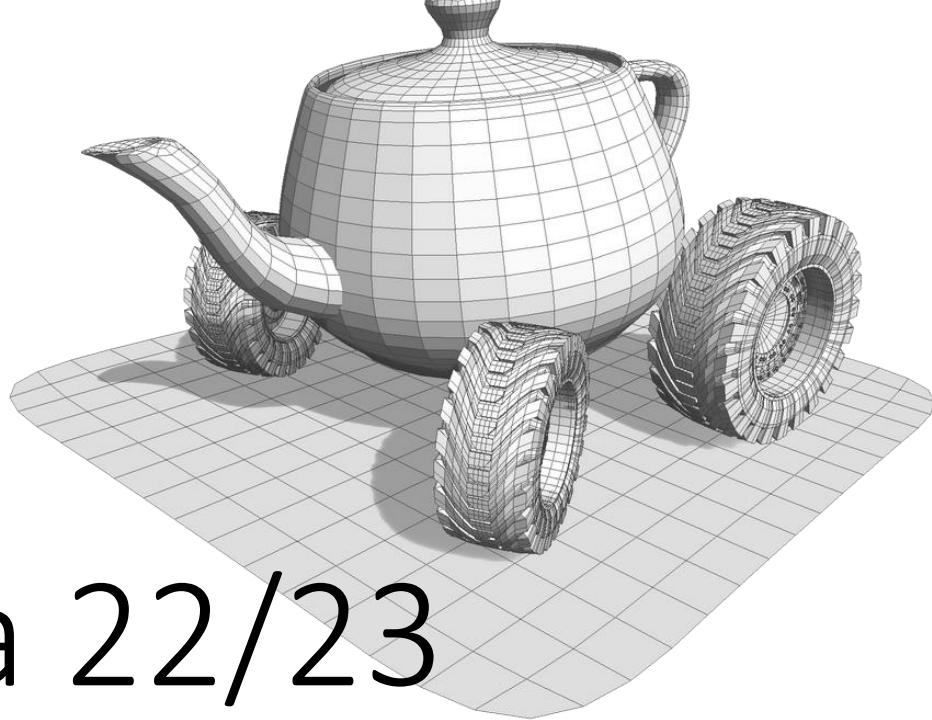
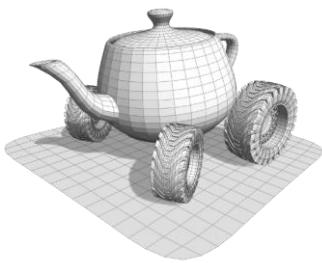


Computer Grafica 22/23

Fabio Ganovelli





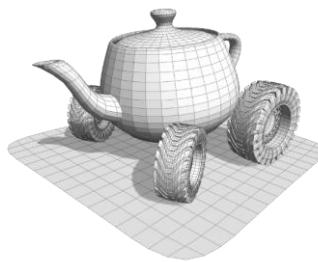
Il docente

Fabio Ganovelli

- Senior Researcher al Consiglio Nazionale delle Ricerche (CNR)
- Area di interesse
 - Computer Graphics & Geometry Processing, modellazione di oggetti deformabili, rendering out-of-core, ricostruzione 3D da immagini

The screenshot shows a web browser window with the title "Fabio Ganovelli Home Page". The address bar indicates the page is not secure (vcg.isti.cnr.it/~ganovelli/). The page content includes a portrait photo of Fabio Ganovelli, his contact information (Istituto di Elaborazione dell'Informazione, Via Moruzzi 1, I-56010 Ghezzano (Pisa) Italy, email: fabio.ganovelli@isti.cnr.it, tel. +39 050 315 2927, fax +39 050 315 2810), and a brief bio stating he is a researcher at the Visual Computing Laboratory, which is part of ISTI (Istituto Scienze e Tecnologie dell'Informazione), which is part of the Italian National Research Council (CNR). He graduated from University of Pisa in 1995, earned his PhD in 2001, and was hired by CNR in 2008. He has been working in Computer since his degree and has published around 50 papers in several subfields including Computer Graphics and Geometry Processing. Below the bio is a navigation menu with links for "Research & Pubbs", "Education", "Community", and "Life". On the right side of the page, there are logos for "FarCry" and "MOLLIC".

Gruppo del docente: VCL vcg.isti.cnr.it



- Visual Computing Lab
 - quasi 30 anni di attività
 - 15 ricerc.
- 3D/2D Digitization
- Geometry processing
- Interactive Graphics
- 3D Graphics for Cultural Heritage
- 3D Fabrication
- ...

The screenshot shows the homepage of the Visual Computing Lab (VCL) at vcg.isti.cnr.it. The page has a yellow header bar with the title "VCG - Home Page". Below the header, there's a navigation bar with links for Publications, Software, Projects, Results, Research, People, and a search icon. The main content area features four images: a person using a tablet to view a 3D model of a classical head, a detailed 3D model of a helmet, a large stone cross, and a bronze statue of a horse mounted on a tripod. Below these images, there's a section titled "Who are we? Where are we going?" containing text about the lab's history and current focus on Cultural Heritage. To the right, there's a "News" section with a link to an award for Paolo Cignoni, and a small image of a 3D-printed object.

VCL vcg.isti.cnr.it

Visual Computing Lab

Publications Software Projects Results Research ▾ People

Who are we? Where are we going?

The history of the Visual Computing Lab of CNR-ISTI began more than 25 years ago, with the friendship and collaboration of Claudio Montani (first head of the lab and former CNR-ISTI Director) and Roberto Scopigno (former head of the VLab, now CNR-ISTI Director). Paolo Cignoni, joined the laboratory at its beginning, and now heads the laboratory.

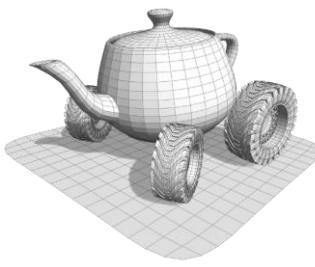
We have initially focused on scientific visualization, then moved to geometry processing, multiresolution representation / visualization, deformable models, 3D digitization, texturing and mesh parameterization.

Currently, our main application domain is Cultural Heritage (that is a rather natural choice since we live in Tuscany), including virtual museums, restoration, documentation and rapid reproduction.

News

Paolo Cignoni: Outstanding Technical Contributions Award

Eurographics conferred to VC Lab Head Paolo Cignoni the 2021 Outstanding Technical Contributions Award for his extraordinary technical contribution to 3D graphics!

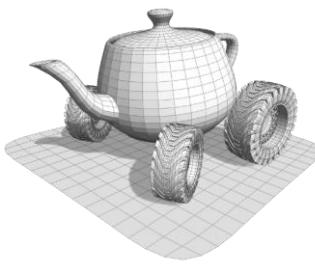


Corsi collegati

- Questo corso sarà molto utile a chi proseguirà il percorso di studi e vorrà includere uno di questi esami
 - 758AA 3D GEOMETRIC MODELING & PROCESSING CIGNONI PAOLO
 - 656AA SCIENTIFIC AND LARGE DATA VISUALIZATION CORSINI MASSIMILIANO, GIORGI DANIELA

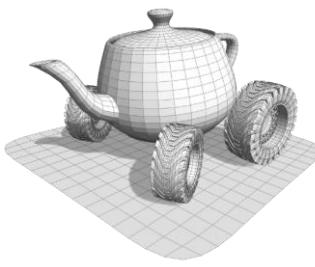
colleghi al VCL

Capo del VCL



Modalità di esame

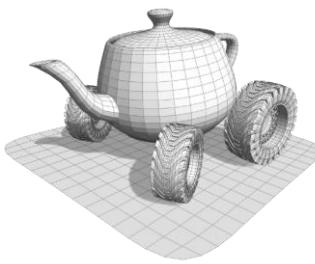
- Scritto:
 - Tipicamente 4 esercizi in 2 ore.
 - Si può usare tutto il materiale e internet.
 - **Non** si posso usare altre teste (nemmeno artificiali)
- Progetto: Il rendering di un simil-videogame
 - Idealmente svolto incrementalmente durante il corso
 - non servono schede grafiche ultimo grido
 - Linguaggi: C++ ma non obbligatorio
 - Sarà chiaro tra poco
- Orale:
 - Brevissimo (5-10 mins) volto solo a verificare che abbiate lavorato al progetto



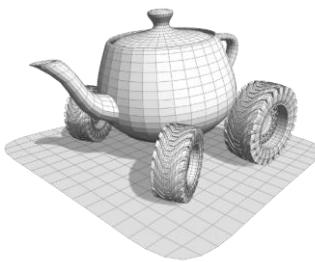
Struttura del corso

- La Computer Graphics è un settore tecnologico/applicativo
- Le conoscenze acquisite possono essere messe in pratica *incrementalmente*
- Il dominio perfetto per un approccio *teaching in context / contextual learning approach*
 - Ci proponiamo di risolvere un problema
 - Impariamo la teoria che serve per risolverlo
- Visti i numeri (di studenti, di insegnanti, di pc) adotteremo una versione «light»
 - Impariamo la teoria su un certo argomento
 - La mettiamo in pratica immediatamente su un progettino che ci seguirà durante tutto il corso

Consigli



- Vi darò ogni settimana un piccolo esercizio, non è un sovraccarico sullo studio, farlo **è studiare**
- Non fate l'errore di separare teoria e pratica per questo corso
 1. Mi studio la teoria per l'esame scritto, poi faccio il progettino → bagno di sangue
 2. Provo via via a far funzionare il codice → passo il **primo** appello con il progettino praticamente finito



Riferimenti

- Introduction to Computer Graphics: a Practical Learning Approach

F.Ganovelli, M.Corsini, S.Pattanaik, and M.Di Benedetto
CRC Press, 2014

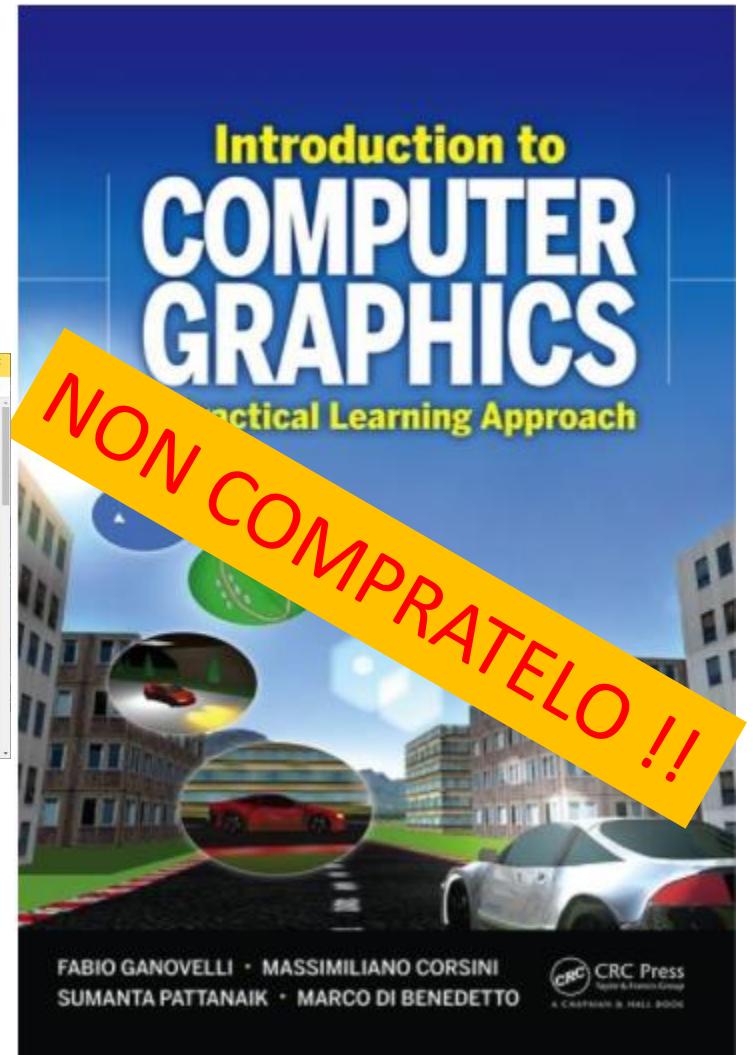
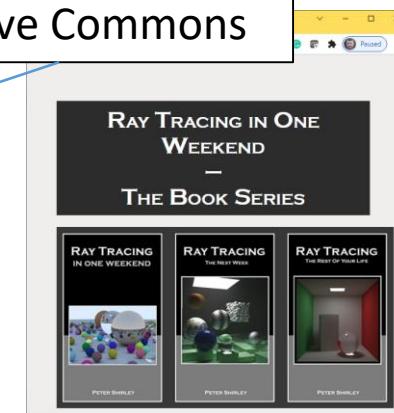
Non serve comprarlo!

- Ray Tracing in one Weekend

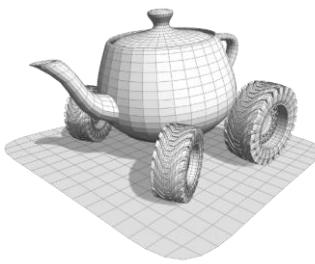
P.Shirley
<http://raytracing.github.io>

- Ray tracing in one Weekend
- Ray Tracing the next week
- Ray tracing for the rest of your life

È Creative Commons

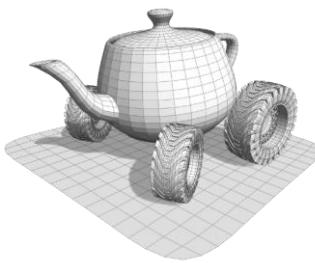


Argomenti

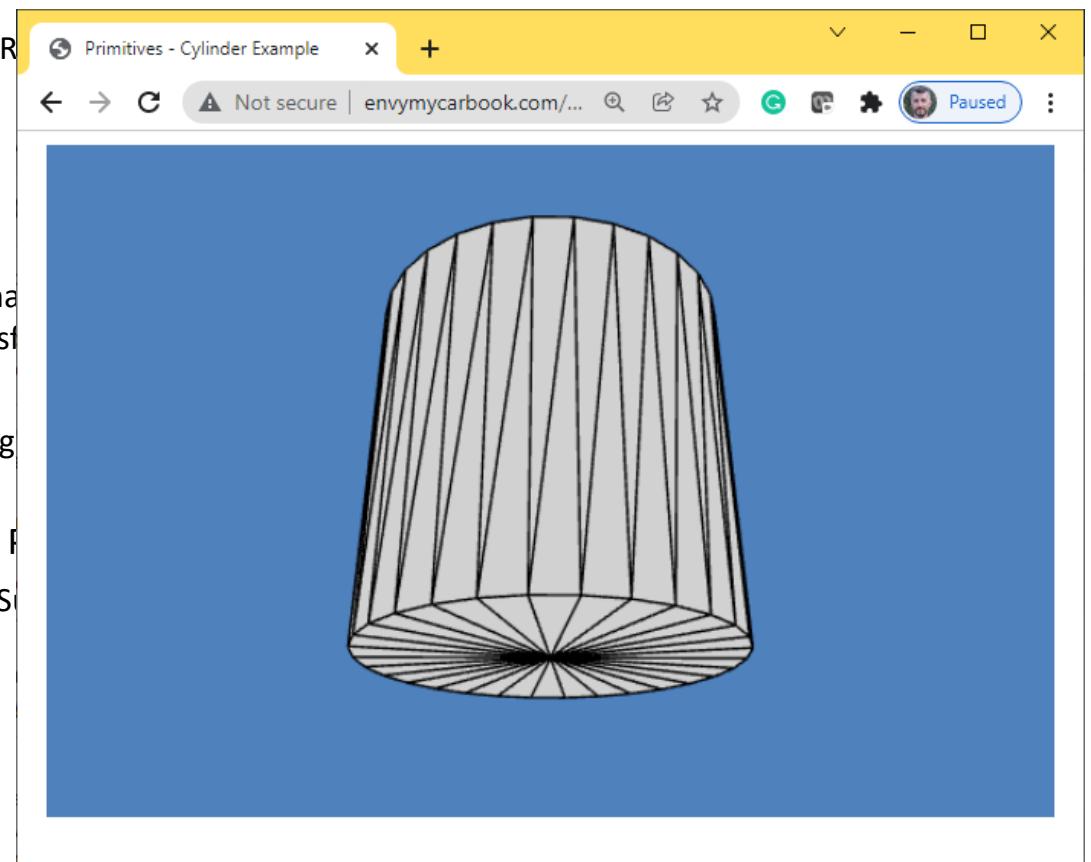


- **Chapter 1** Fondamenti
 - Il sistema di visione umano, Color spaces, illuminant, gamma correction, Rappresentazione delle immagini: immagini raster e immagini vettoriali
 - Pipeline di rendering rasterization based (o proiettiva), Raytracing
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensionali
 - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rendering
 - Trasformazioni di base: I frames, organizzazione gerarchica delle trasformazioni.
 - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettiche. Trasformazioni nella pipeline di rendering
- **Chapter 6** Lighting and Shading
 - Interazione luce/materia: Riflessione, rifrazione, assorbimento, scattering, l'equazione di rendering
 - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Minification. Perspective correct interpolation
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscurrence. Subsurface scattering
- **Chapter 9-10** Utilizzi avanzati
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping
- **Chapter 11+** Global Illumination
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing

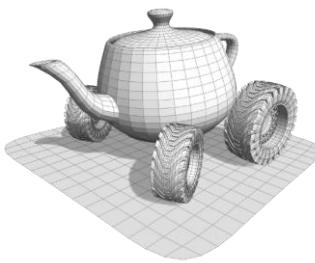
Argomenti



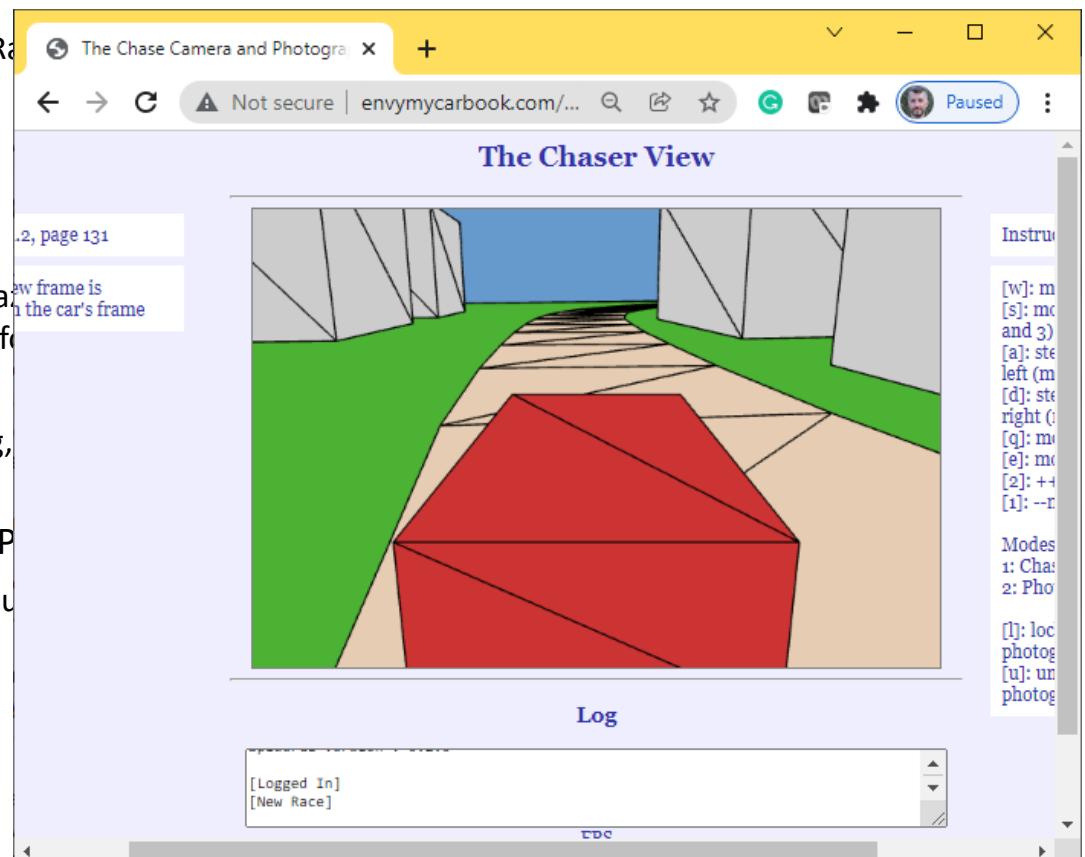
- **Chapter 1** Fondamenti
 - Il sistema di visione umano, Color spaces, illuminant, gamma correction, Ray tracing
 - Pipeline di rendering rasterization based (o proiettiva), Raytracing
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensionali
 - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rendering
 - Trasformazioni di base: I frames, organizzazione gerarchica delle trasformazioni
 - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettiche. Trasformazioni di proiezione
- **Chapter 6** Lighting and Shading
 - Interazione luce/materia: Riflessione, rifrazione, assorbimento, scattering
 - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Minification. Filtering
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscuration. Spherical harmonics
- **Chapter 9-10** Utilizzi avanzati
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping
- **Chapter 11+** Global Illumination
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing



Argomenti



- **Chapter 1** Fondamenti
 - Il sistema di visione umano, Color spaces, illuminant, gamma correction, Ray tracing
 - Pipeline di rendering rasterization based (o proiettiva), Raytracing
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensionali
 - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rendering
 - Trasformazioni di base: i frames, organizzazione gerarchica delle trasformazioni
 - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettiche. Trasformazioni di proiezione
- **Chapter 6** Lighting and Shading
 - Interazione luce/materia: Riflessione, rifrazione, assorbimento, scattering, absorption
 - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Minification. Precompute
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscuration. Surface shading
- **Chapter 9-10** Utilizzi avanzati
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping
- **Chapter 11+** Global Illumination
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing



Argomenti

- **Chapter 1** Fondamenti
 - Il sistema di visione umano, Color spaces, illuminant, gamma correction
 - Pipeline di rendering rasterization based (o proiettiva), Raytracing
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensionali
 - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rendering
 - Trasformazioni di base: I frames, organizzazione gerarchica delle scene
 - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettive
- **Chapter 6 Lighting and Shading**
 - Interazione luce/materia: Riflessione, rifrazione, assorbimento, scattering
 - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Minification
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscurance
- **Chapter 9-10** Utilizzzi avanzati
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping
- **Chapter 11+ Global Illumination**
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing)

The image displays two side-by-side screenshots of a 3D rendering application interface. Both windows have a yellow header bar with the title, a back/forward button, a refresh icon, a search icon, a star icon, a puzzle icon, a user icon, and a 'Paused' button.

Top Window (Add the Sun):

- Title:** Add the Sun
- Content:** A 3D scene showing a red sports car on a road at night. In the background, there is a large white building and some greenery. The sky is dark.
- Left Sidebar (Instructions):**
 - page 193
 - made with boxes is gone. We load model for the
 - is a directional in.
 - uilding look so ? Check Section
 - n and street int color and car dings and tress
- Right Sidebar (Instructions):**
 - [w]: mov and 2)
 - [s]: mov and 3)
 - [a]: steer left (mod
 - [d]: steer right (mod
 - [q]: mov [e]: mov
 - [2]: ++rr
 - [1]: --mo

Modes:
1: Chase
2: Photo
3: The ol
- Bottom Log:** (empty)

Bottom Window (Car's headlights and light in the t):

- Title:** Car's headlights and light in the t
- Content:** The same scene as the top window, but the car's headlights are turned on, casting a bright yellow glow on the road ahead.
- Left Sidebar (Instructions):**
 - 03
 - otlights lights we
 - are area multiple 6.7.6).
- Right Sidebar (Instructions):**
 - [w]: mov and 2)
 - [s]: mov and 3)
 - [a]: steer left (mod
 - [d]: steer right (mod
 - [q]: mov [e]: mov
 - [2]: ++rr
 - [1]: --mo

Modes:
1: Chase
2: Photo
3: The ol
- Bottom Log:** (empty)

Add Reflections to the Car



- Phong lighting, Cook-torrance, Oren-Nayar, Minnaert

Chapter 7 Textures. Concetti di base. Magnification and Minification. Perspective correct interpolation

- Chapter 8 Ombre: Shadow mapping e shadow volumes; Ambient obscurance. Subsurface scattering
- Chapter 9-10 Utilizzi avanzati
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping (1h)
- Chapter 11+ Global Illumination
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing (4h)

Textures to the Terrain, Street

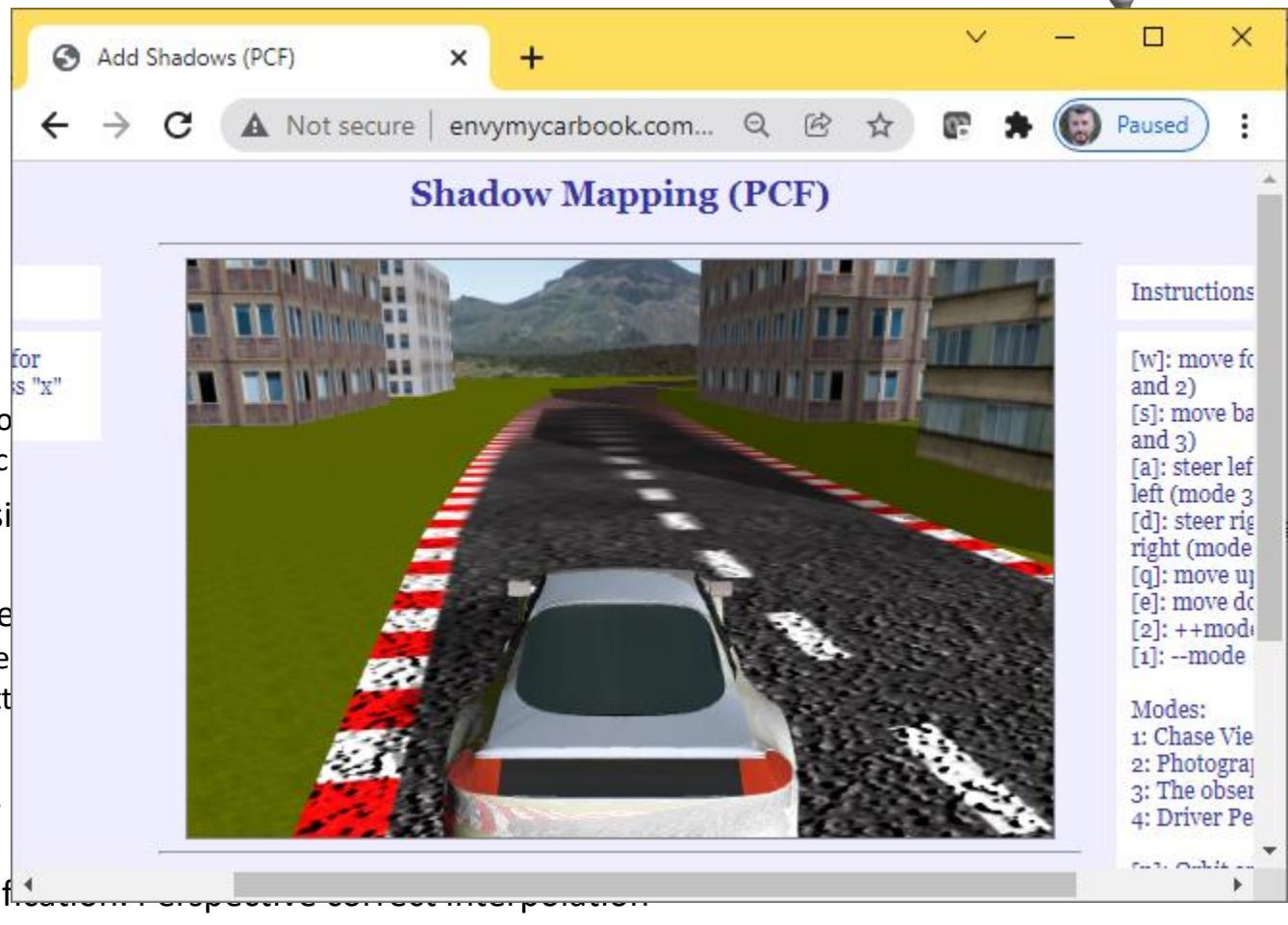
C Not secure | envymycarbook.com... Paused

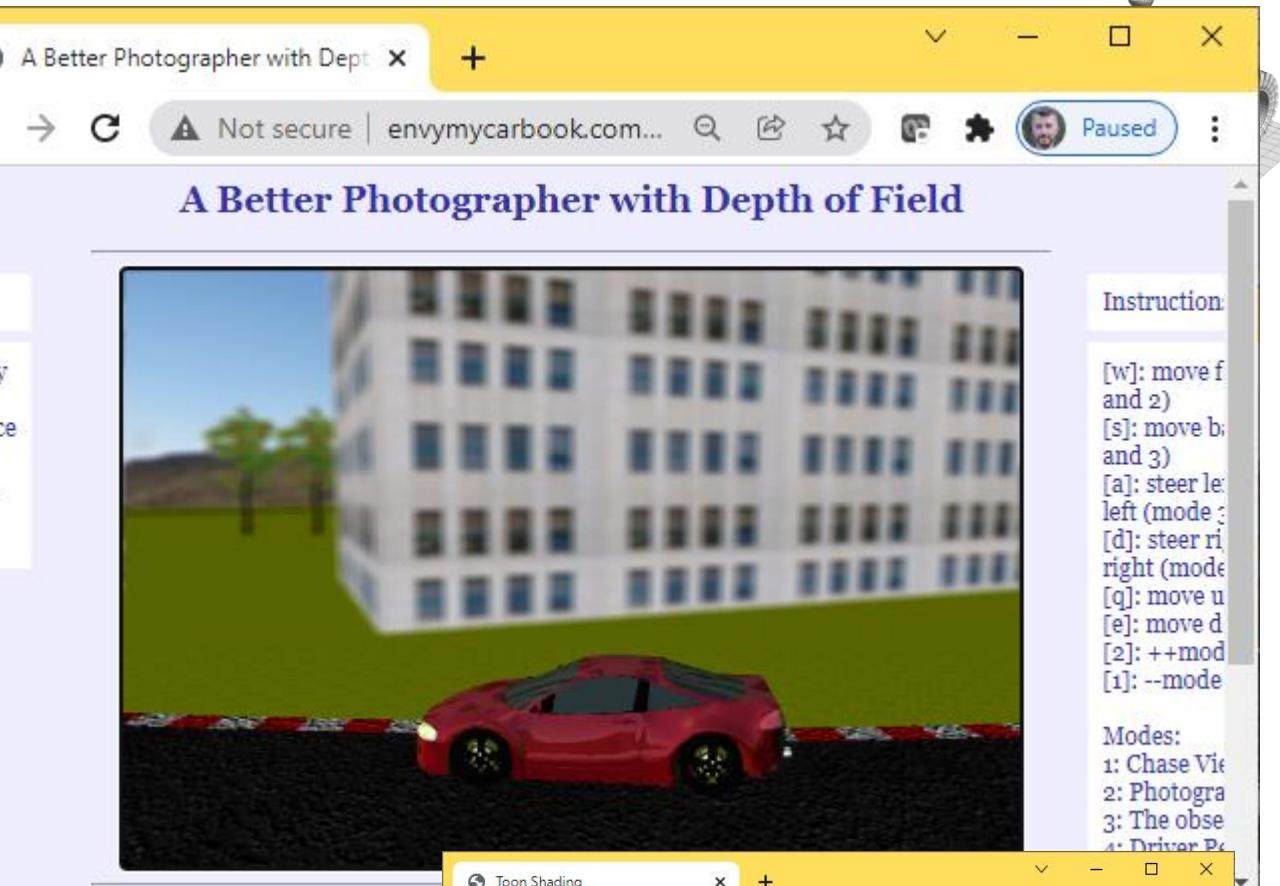
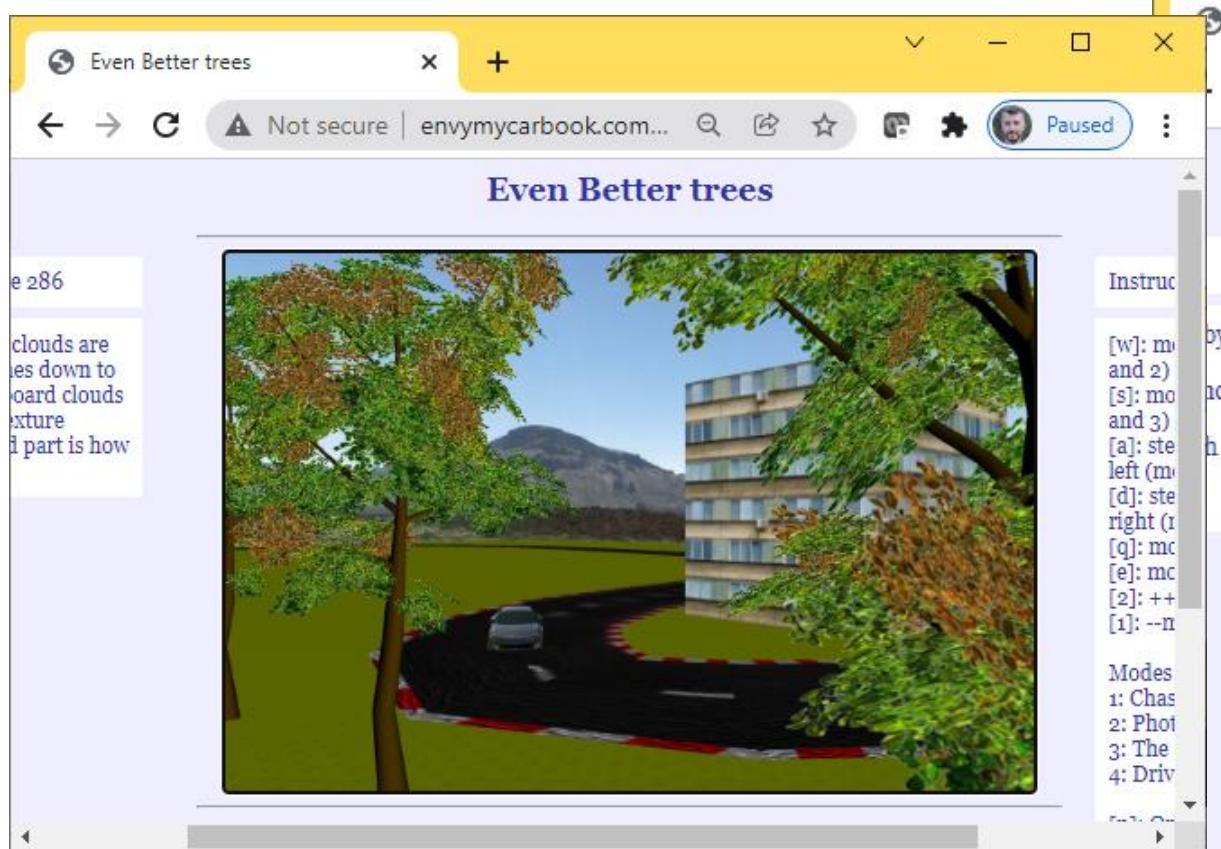
Add Textures to the Terrain, Street and Buildings



Argomenti

- **Chapter 1** Fondamenti
 - Il sistema di visione umano, Color spaces, illuminant, gamma correction
 - Pipeline di rendering rasterization based (o proiettiva), Raytracing
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensionali
 - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rendering
 - Trasformazioni di base: I frames, organizzazione gerarchica delle scene
 - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettive
- **Chapter 6** Lighting and Shading
 - Interazione luce/materia: Riflessione, rifrazione, assorbimento, scattering
 - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Minification
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscuration. Subsurface scattering
- **Chapter 9-10** Utilizzi avanzati
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping (1h)
- **Chapter 11+** Global Illumination
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing (4h)

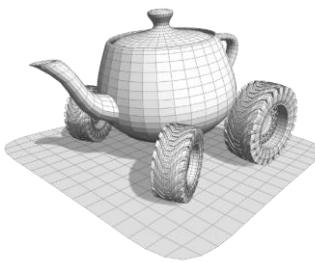




- **Chapter 7** Textures. Concetti di base. Magnification and Mipmapping. Perspective correction.
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscuration. Subsurface scattering
- **Chapter 9-10 Utilizzzi avanzati**
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping
- **Chapter 11+ Global Illumination**
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing (4h)



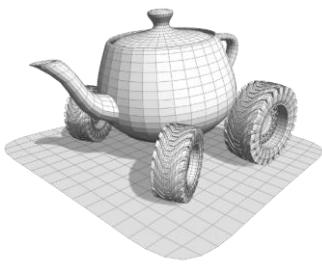
Argomenti



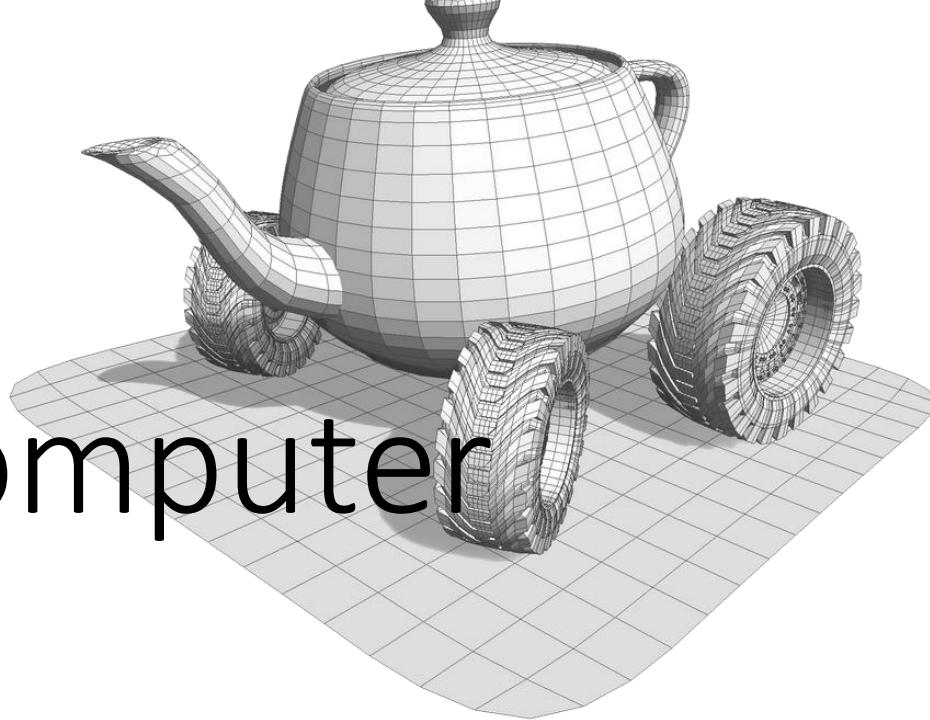
- **Chapter 1** Fondamenti
 - Il sistema di visione umano, Color spaces, illuminant, gamma correction, Rappresentazione delle immagini: immagini raster e immagini vettoriali
 - Pipeline di rendering rasterization based (o proiettiva), Raytracing
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensionali
 - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rendering
 - Trasformazioni di base: i frames, organizzazione gerarchica delle trasformazioni.
 - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettiche. Trasformazioni nella pipeline di rendering
- **Chapter 6** Lighting and Shading
 - Interazione luce/materia: Riflessione, rifrazione, assorbimento, scattering, l'equazione di rendering
 - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Minification
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient occlusion
- **Chapter 9-10** Utilizzi avanzati
 - Depth of field, lens flare, radial distortion
 - Texture e lighting: bump mapping, relief mapping
- **Chapter 11+ Global Illumination**
 - Ray tracing, strutture di accelerazione, path tracing, photon tracing

Where is the car racing??

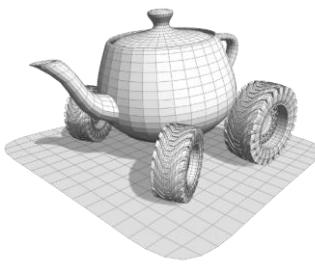




Introduction to Computer Graphics

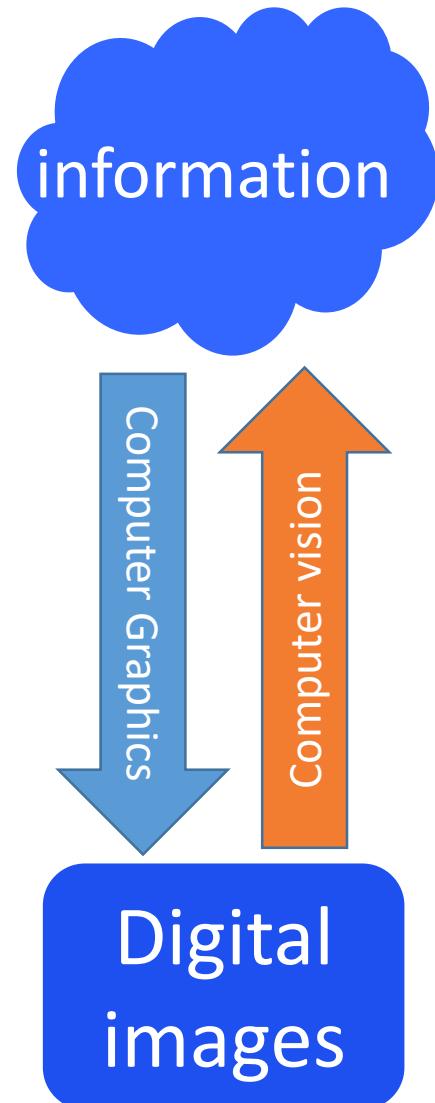
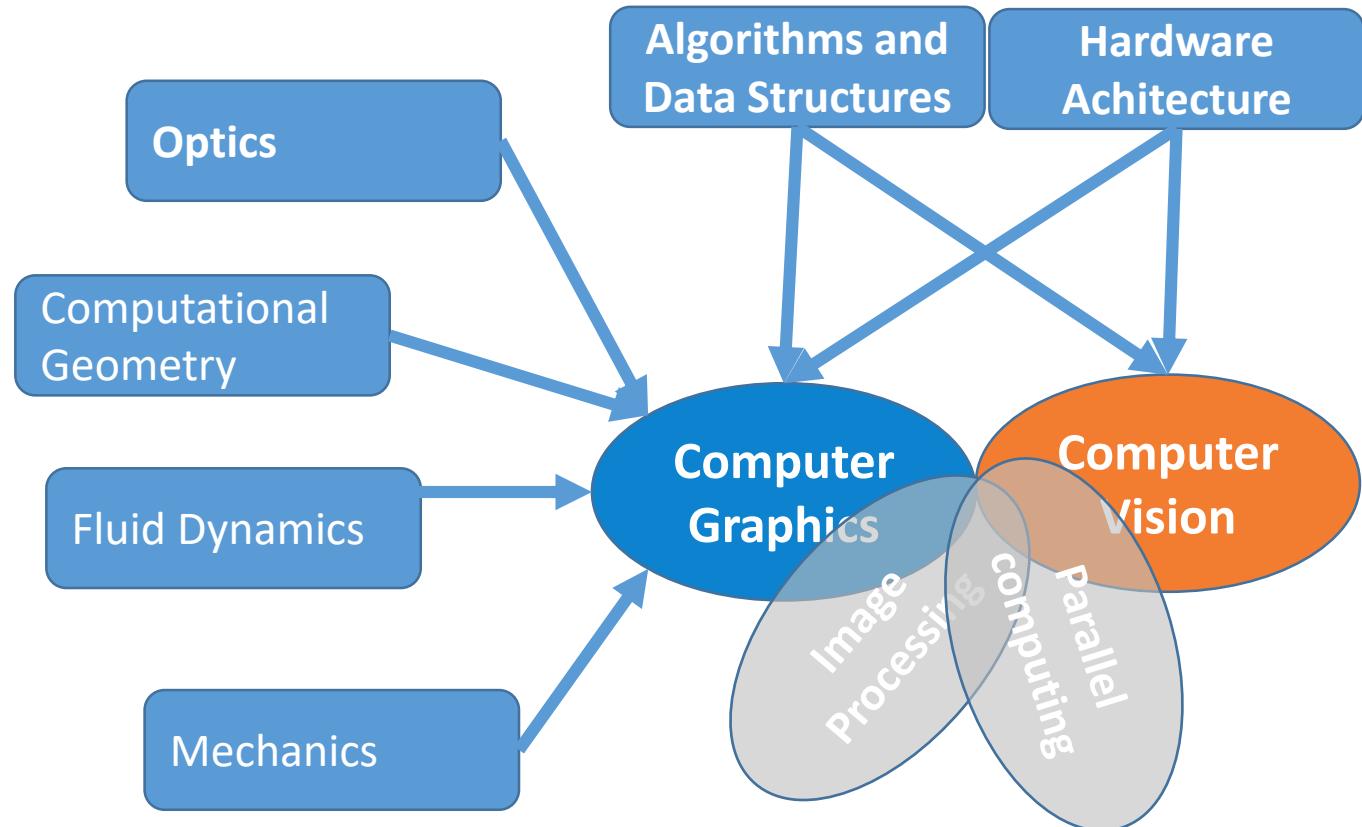
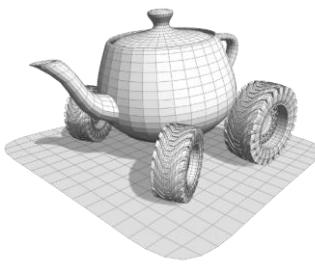


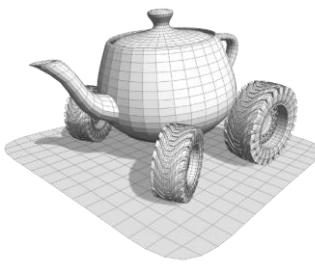
What 3D Computer Graphics is



- A sub-field of computer science which studies methods for digitally synthesizing, manipulating and visualizing **digital content**
- An interdisciplinary field where computer scientists, mathematicians, physicists, engineers, artists and practitioners gather with the common goal of opening a «window» on the «world», where
 - windows stands for: monitor, tablet, phone, visors..anything that can show images really
 - World stand for: a digital model, the result of a scientific simulation, anything for which we can conceive a visual representation

Connections & Relations





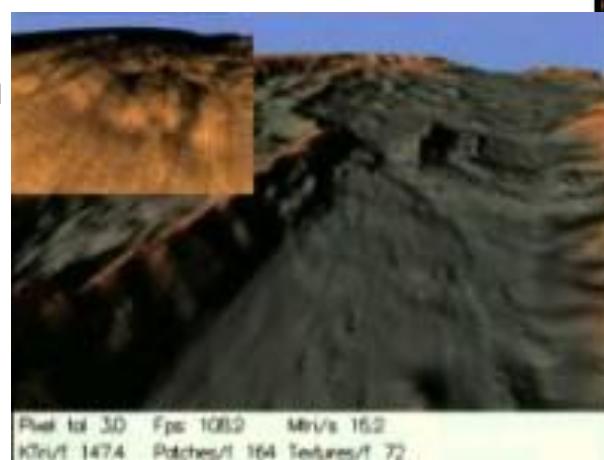
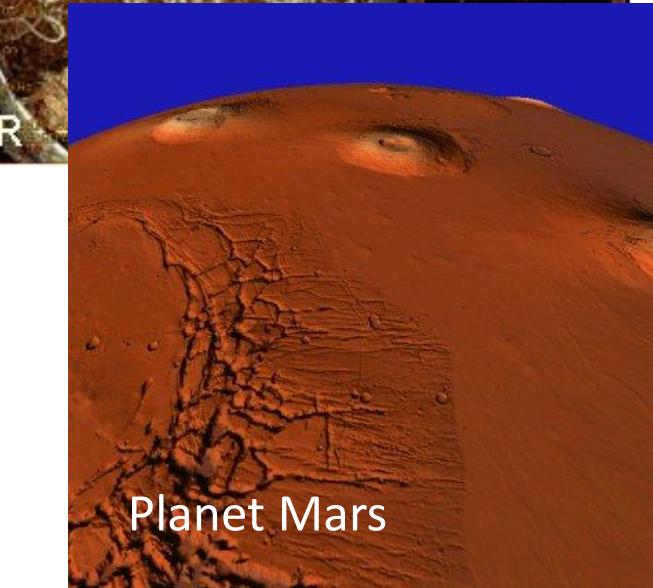
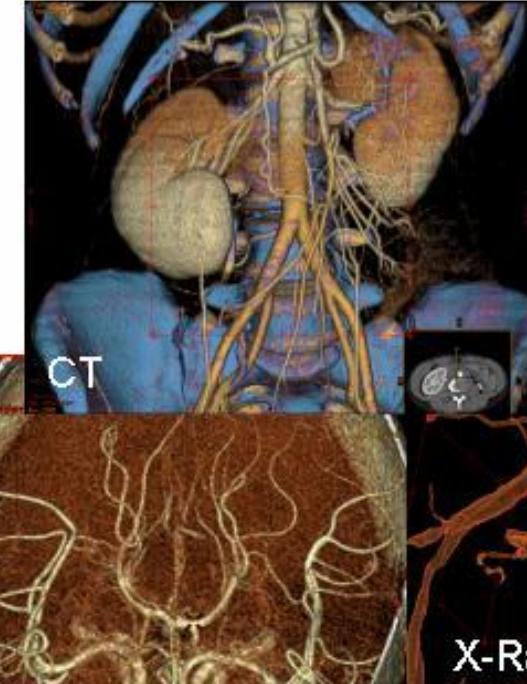
Computer Graphics: applications

- **Scientific**
 - Scientific Visualization
 - Data Visualization
- E-Commerce
 - Product display
- Medicine
 - Diagnosis
 - Virtual Surgery, Telesurgery
- Manufacturing
 - Computer Aided Design
- Entertainment
 - Movies
 - Visual Effects
 - CGI movies
 - Videogames
- Cultural Heritage
 - Virtual Museums
 - Support to restoration
 - Supporto to Study
- Architecture
 - Design
 - Lighting Design
 - Preview

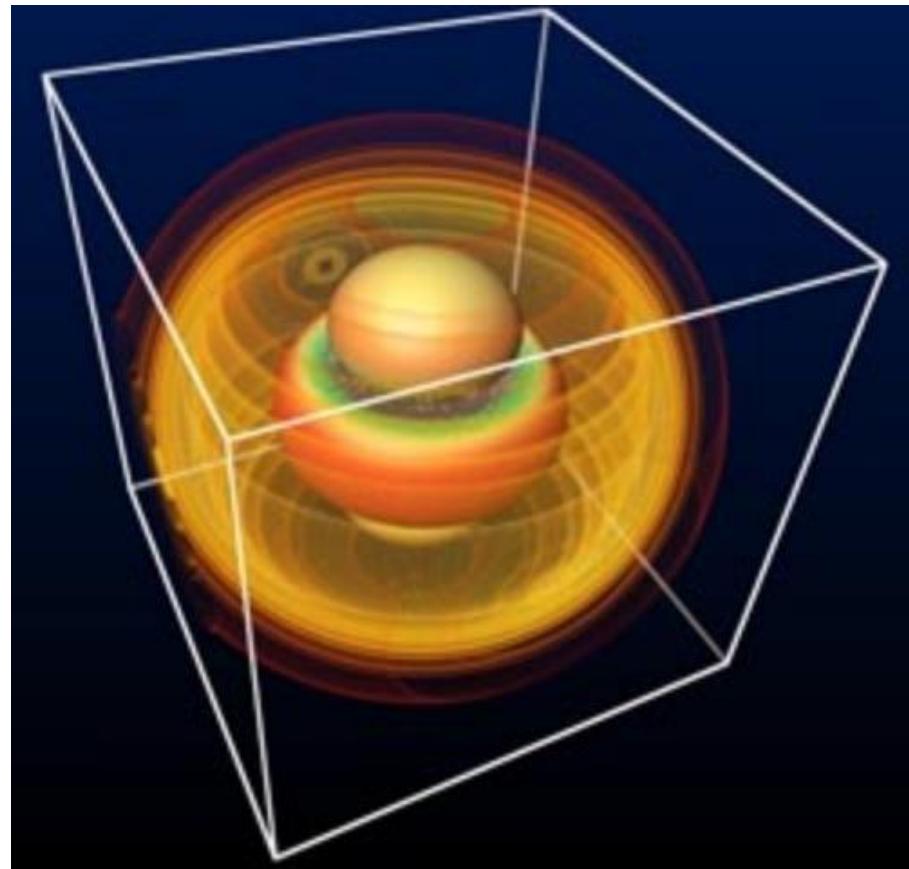
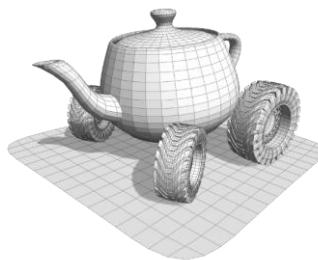
Scientific Visualization

- A.k.a.: SciVis , visual data analysis ...
- Visualization of *scientific data*
 - meteo, medical, biological, chemical, physics, astrophysics, etc
- Data source:
 - output of a simulation
 - *Mesasured with a specific system*

Typically a lot of data

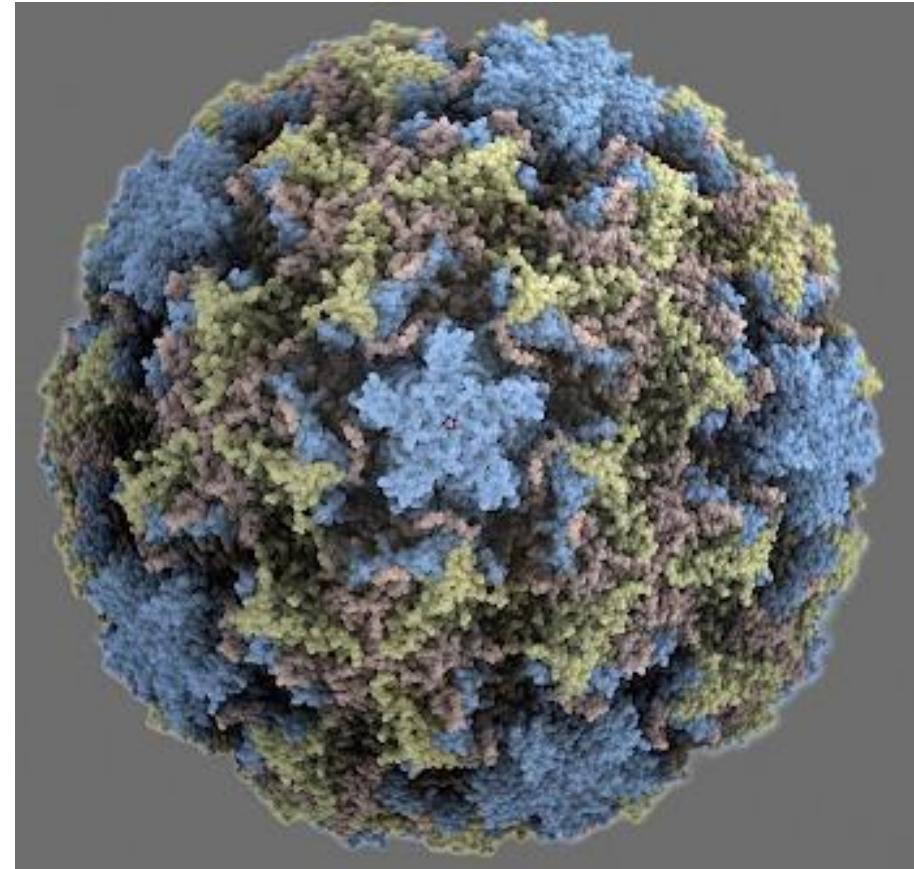


Scientific Visualization



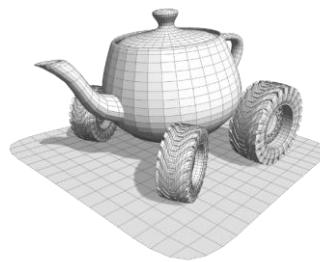
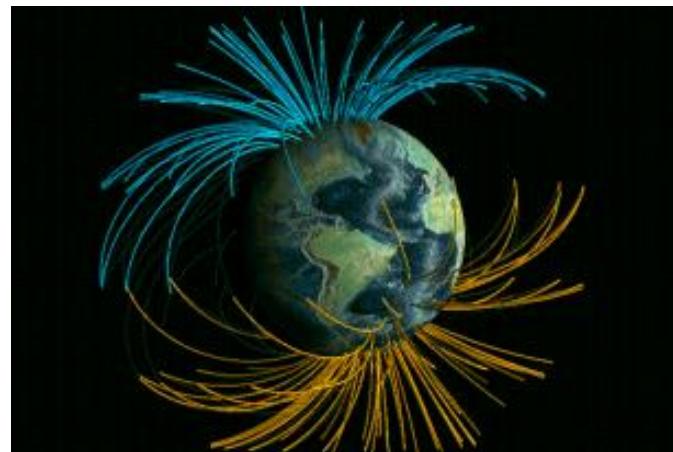
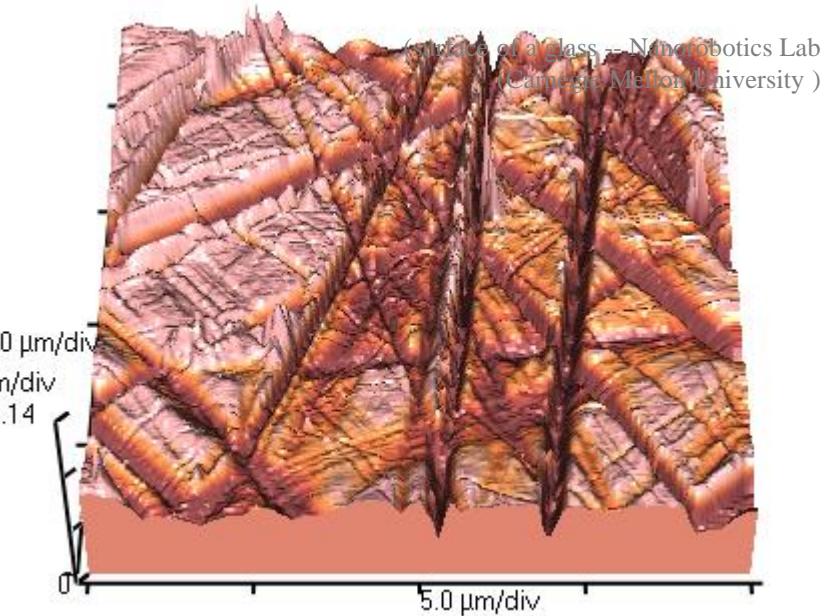
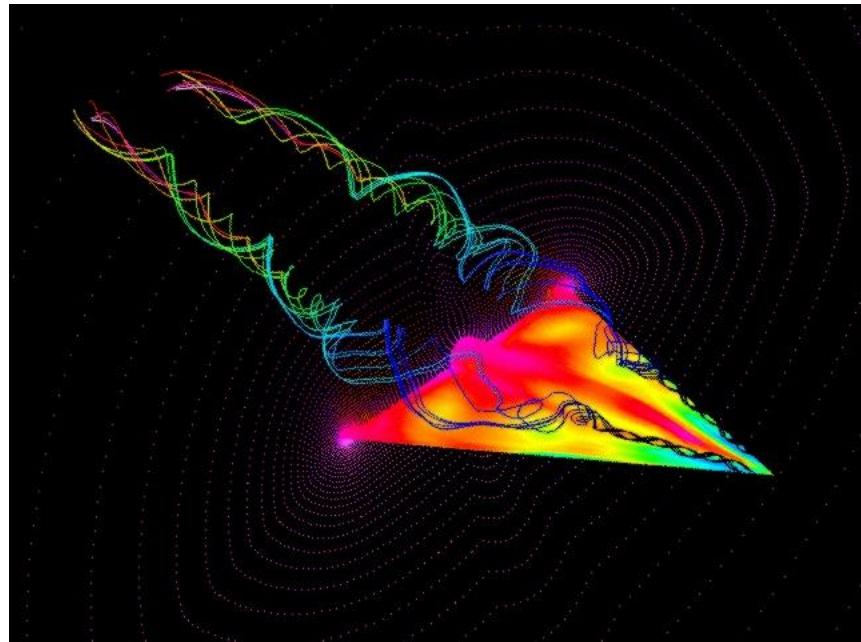
Gravitational waves during a collision of black holes

(Max Planck Institute for Gravitational Physics)

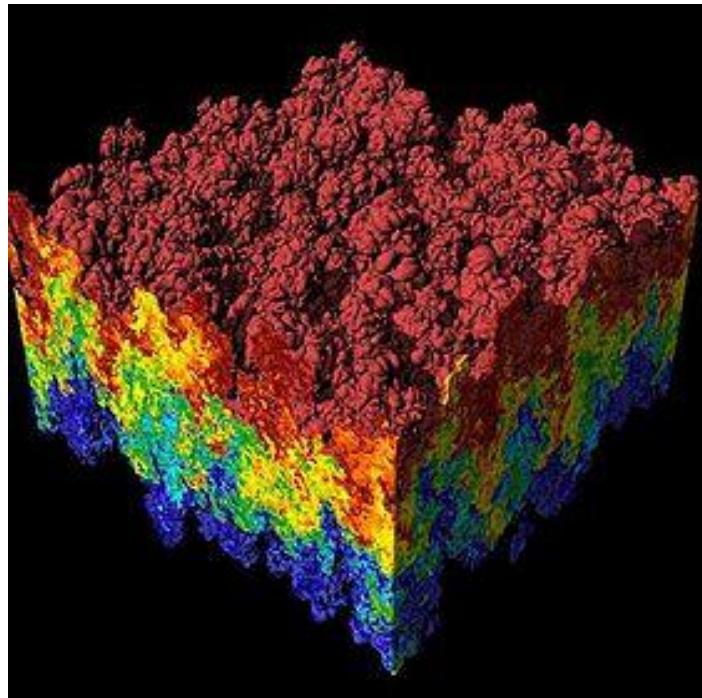
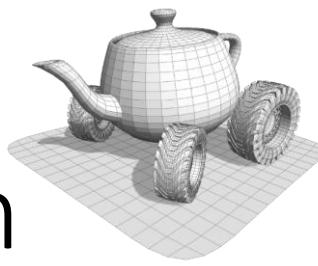


Rhinovirus 3 protein

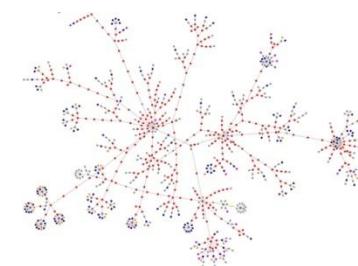
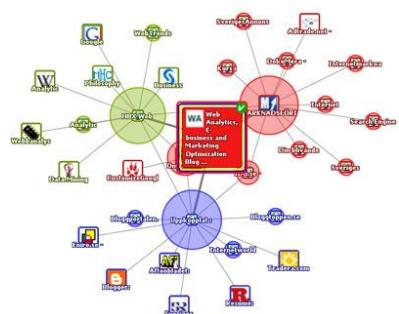
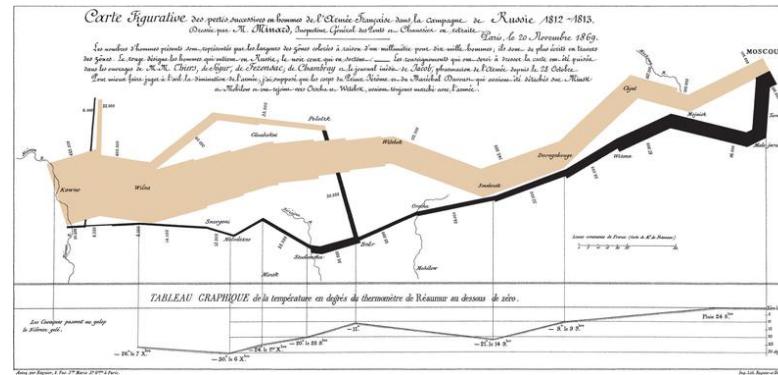
Scientific Visualization

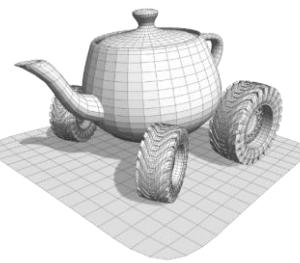


Scientific Visualization vs Data Visualization

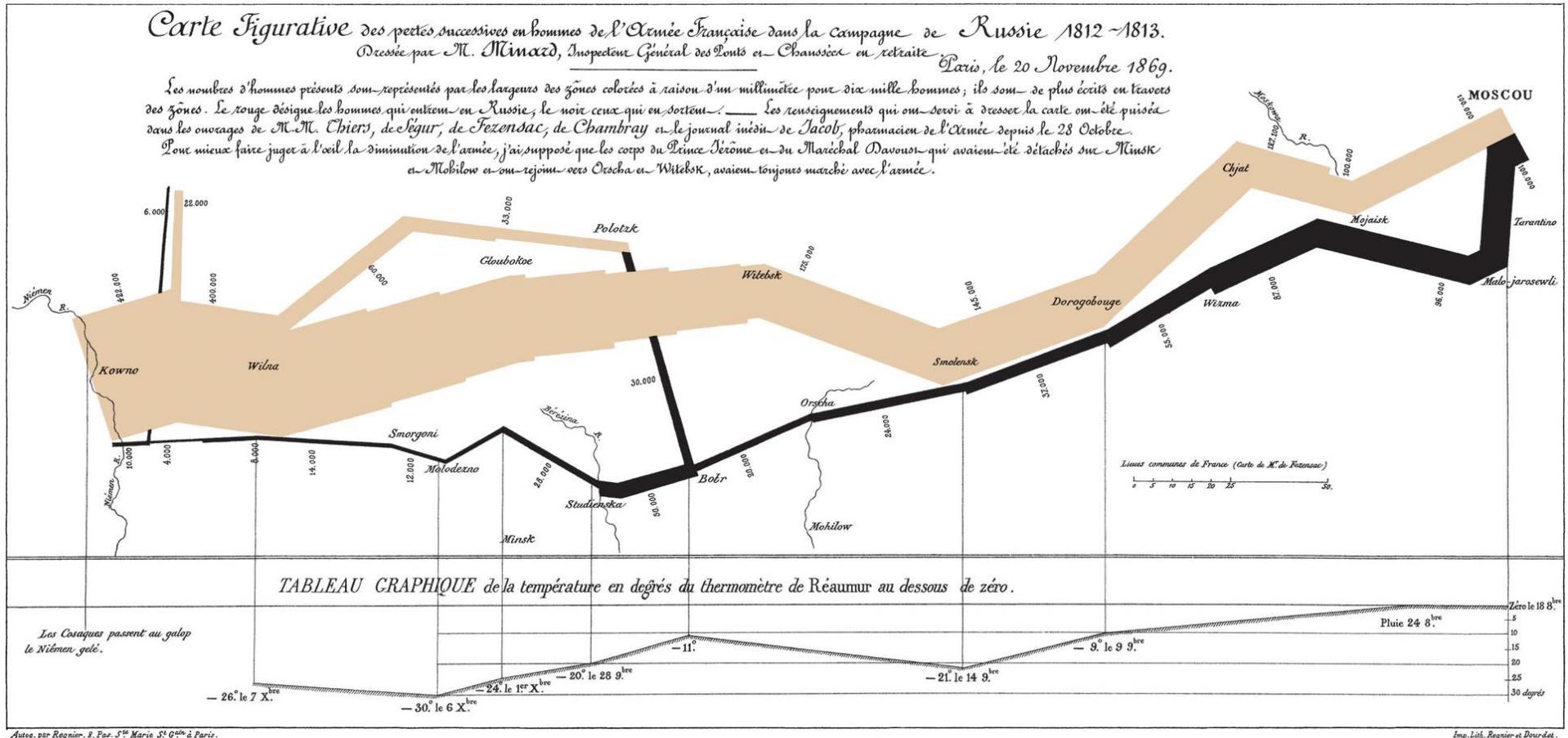


Simulation of the interaction between two fluids with different densities



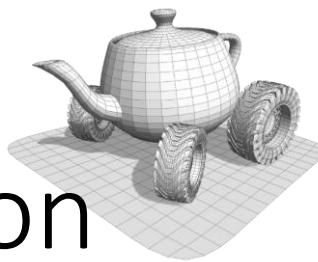


Data Visualization: an early example



Charles Minard, 1869: Napoleon Campaign

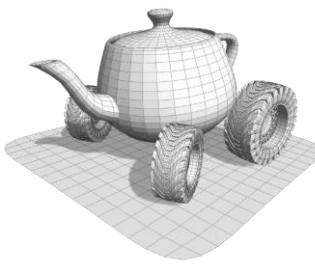
Scientific Visualization VS Data Visualization



- **data** with a natural 3D interpretation
 - Often the time dimension is added
- **Goal:** readability
- **Means:** Realism or Illustrative rendering
- **data** abstract (N dimensional)
- **Goal:** readability
- **Means:** graphs, glyphs...whatever it works

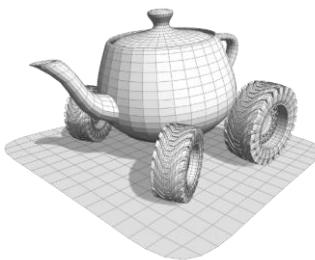
Illustrative rendering





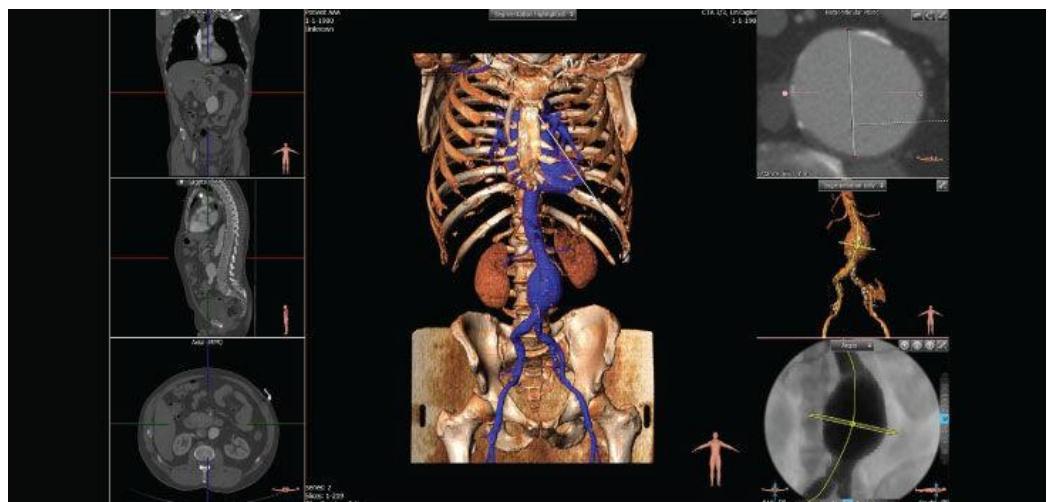
Computer Graphics: applications

- Scientific
 - Scientific Visualization
 - Data Visualization
- E-Commerce
 - Product display
- Medicine
 - Diagnosis
 - Virtual Surgery, Telesurgery
- Manufacturing
 - Computer Aided Design
- Entertainment
 - Movies
 - Visual Effects
 - CGI movies
 - Videogames
- Cultural Heritage
 - Virtual Museums
 - Support to restoration
 - Supporto to Study
- Architecture
 - Design
 - Lighting Design
 - Preview



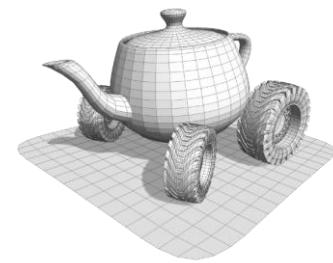
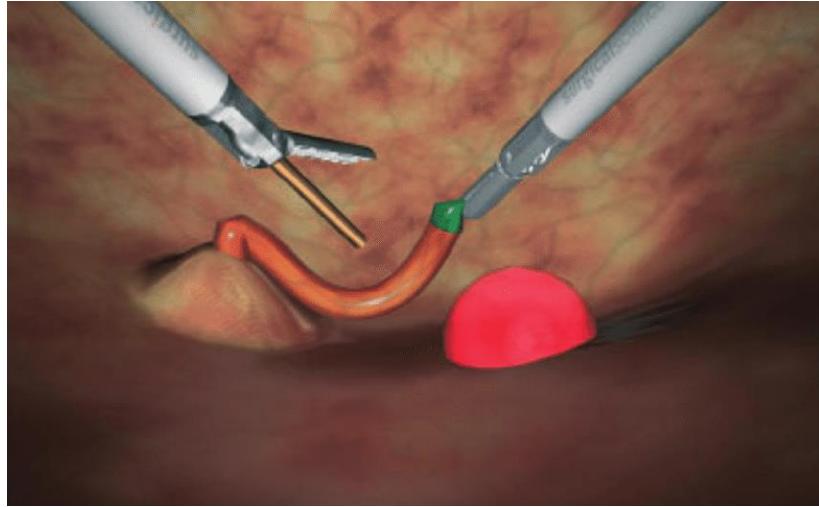
Medical applications

- Diagnostic: 3D models can be created from CT+MRI, ultrasounds
 - Improve human readability

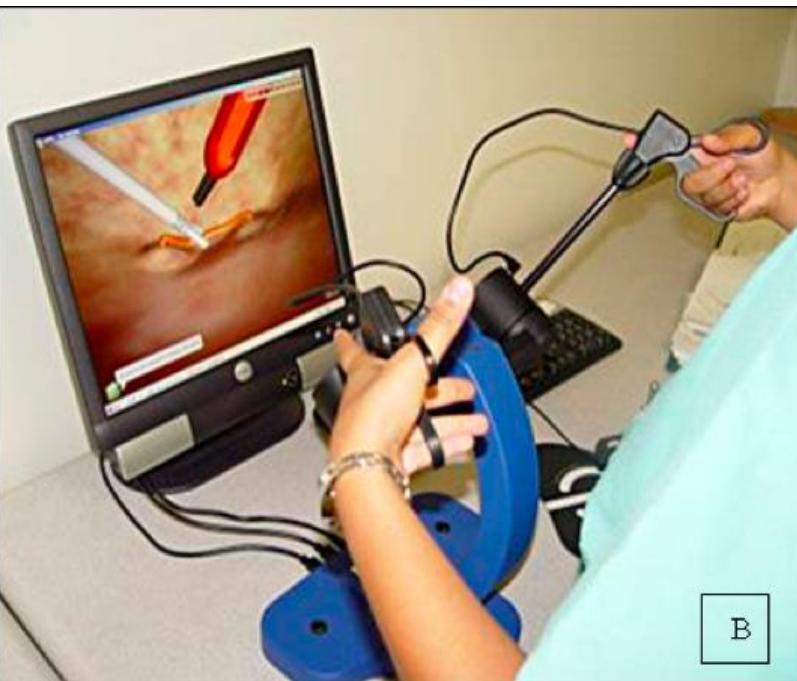


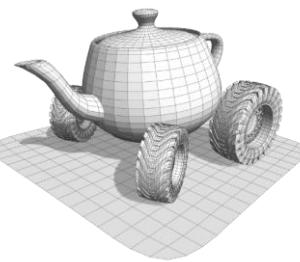
Medical applications

- Virtual surgery simulator
 - E.g. Virtual laparoscopic Cholecystectomy



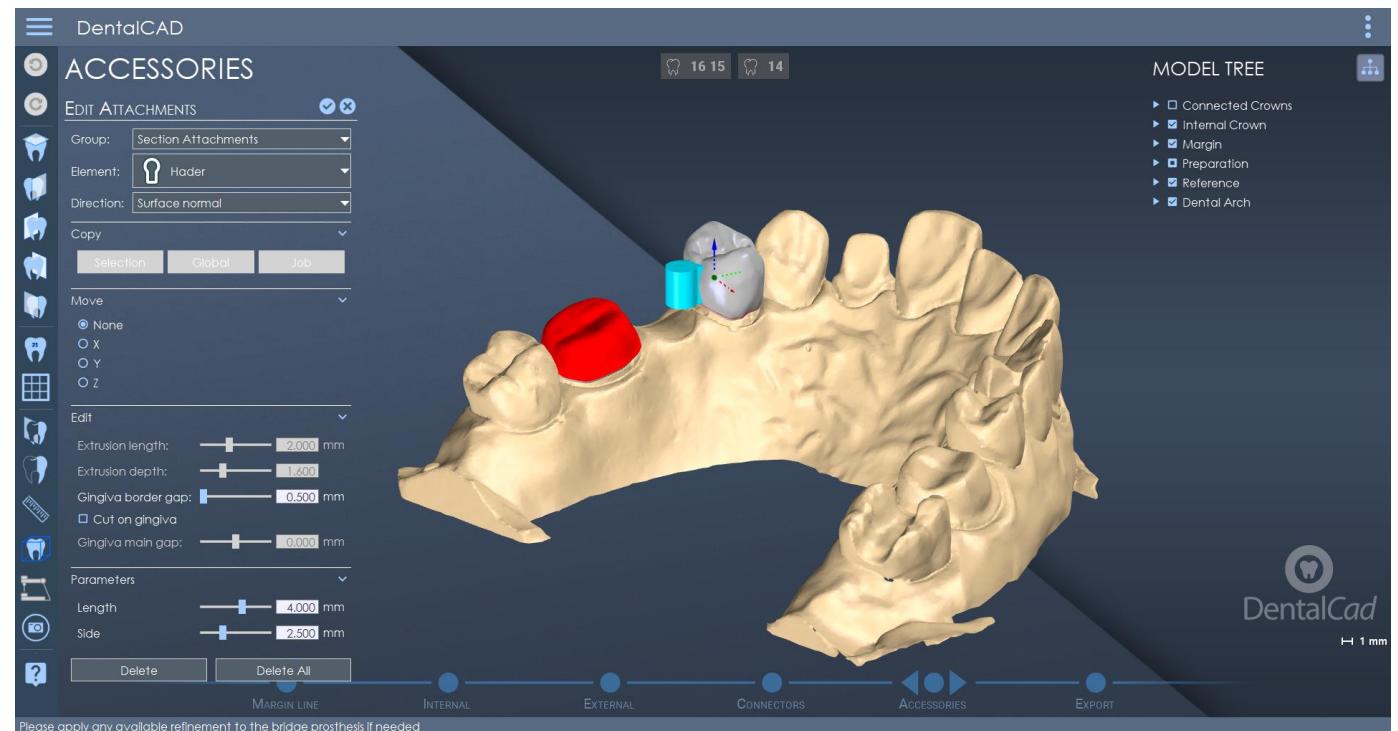
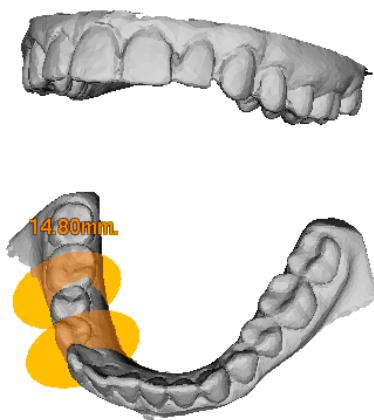
Lap mentor (tm)



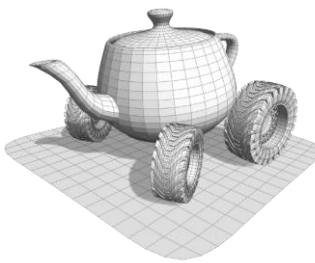


Medical applications

- Surgery planning

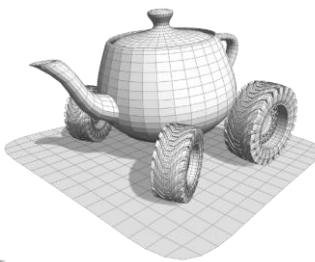


Computer Graphics: applications

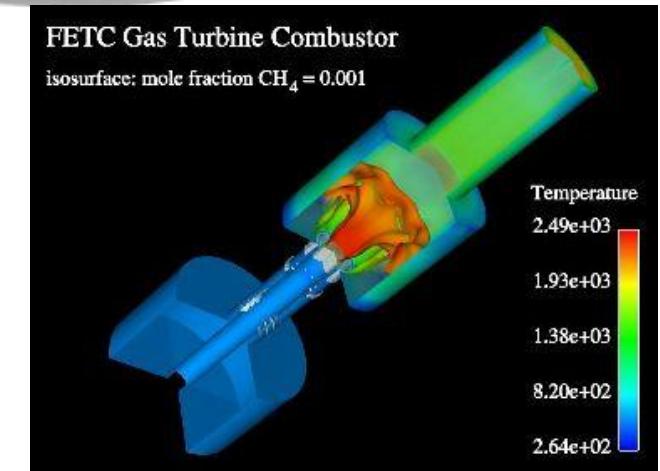
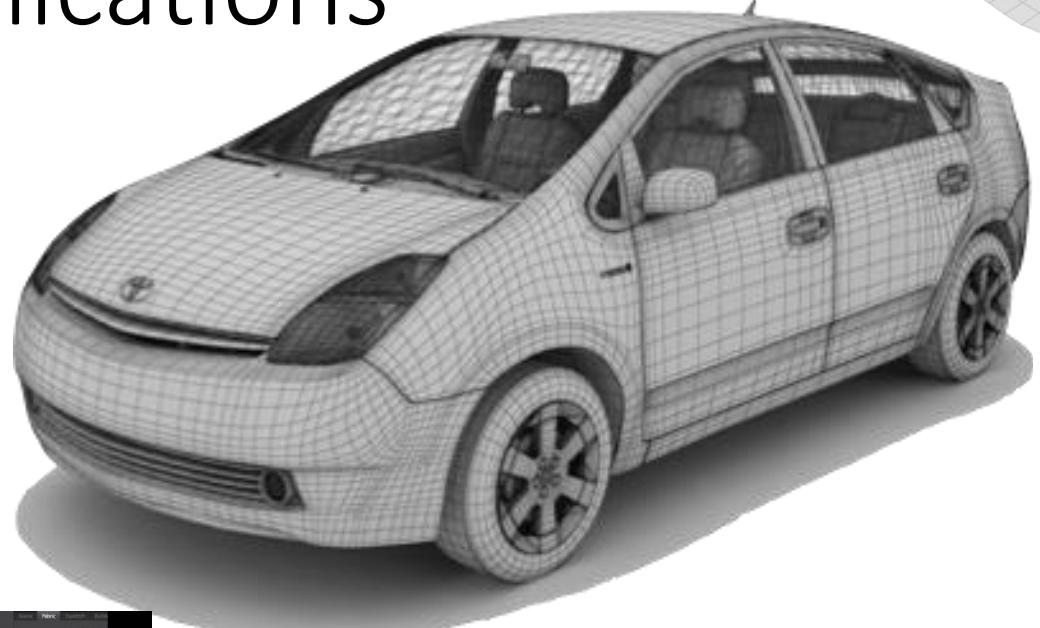
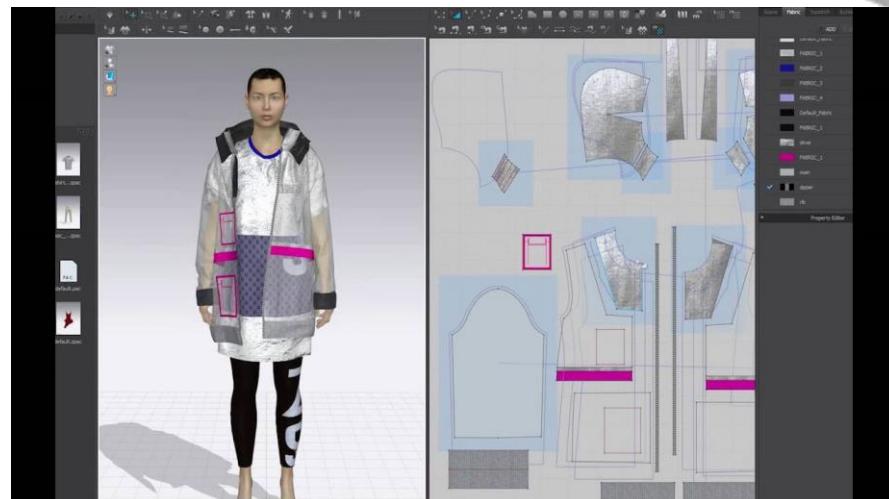


- Scientific
 - Scientific Visualization
 - Data Visualization
- E-Commerce
 - Product display
- Medicine
 - Diagnosis
 - Virtual Surgery, Telesurgery
- Manufacturing
 - Computer Aided Design
- Entertainment
 - Movies
 - Visual Effects
 - CGI movies
 - Videogames
- Cultural Heritage
 - Virtual Museums
 - Support to restoration
 - Supporto to Study
- Architecture
 - Design
 - Lighting Design
 - Preview

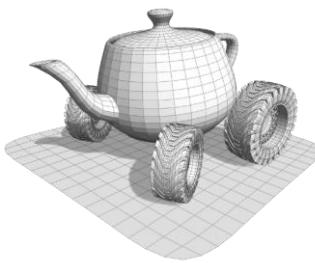
Computer Graphics: Applications



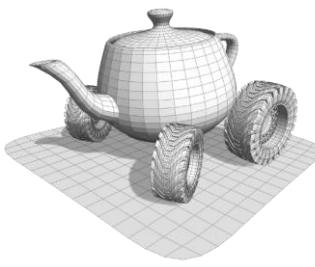
- Manufacturing Industry
 - Computer Aided Design
 - Rapid Prototyping
 - Simulations
(e.g. Finite Element Methods)
 - ...



Computer Graphics: applications



- Scientific
 - Scientific Visualization
 - Data Visualization
- E-Commerce
 - Product display
- Medicine
 - Diagnosis
 - Virtual Surgery, Telesurgery
- Manufacturing
 - Computer Aided Design
- Entertainment
 - Movies
 - Visual Effects
 - CGI movies
 - Videogames
- Cultural Heritage
 - Virtual Museums
 - Support to restoration
 - Supporto to Study
- Architecture
 - Design
 - Lighting Design
 - Preview



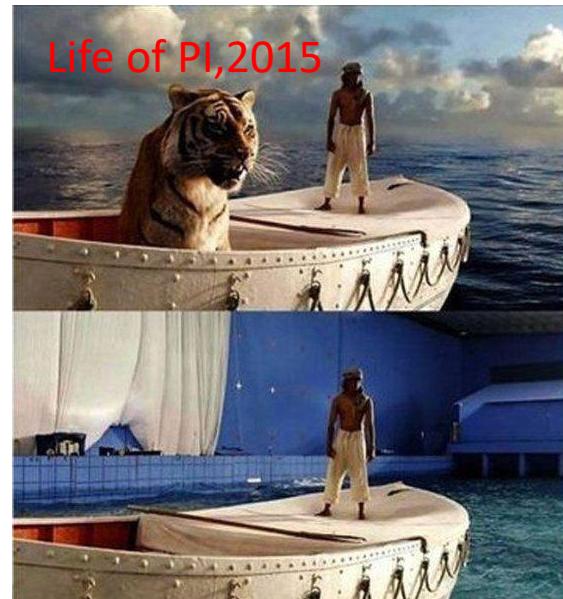
Computer Graphics: Applications

- Entertainment: movie industry
 - visual effects (not special fx)



Visual Effects VS Special Effects

Visual Effects: added in post production, often using Computer Generated Images (CGI)



Special Effects: real things taken on camera. Stunts, ropes, explosions, mad people doing stuff

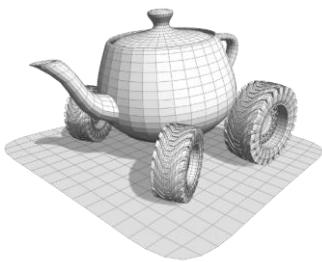




Computer Graphics: Applications

- Entertainment: movie industry
 - CG shorts





Computer Graphics: applicazioni

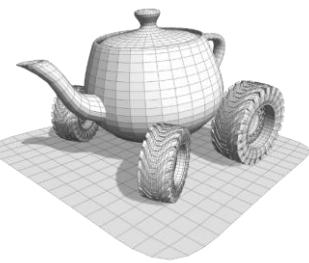
- Entertainment: movie industry
 - CG shorts
 - Feature movies

Toy Story - Pixar 1995



Coco - Pixar 2017





Computer Graphics: applicazioni

- Entertainment: movie industry
 - CG shorts
 - Feature movies
 - Feature movies fotorealistici



Chapter 1: What Computer Graphics is

Final Flight Of the Osiris – Squaresoft 2003

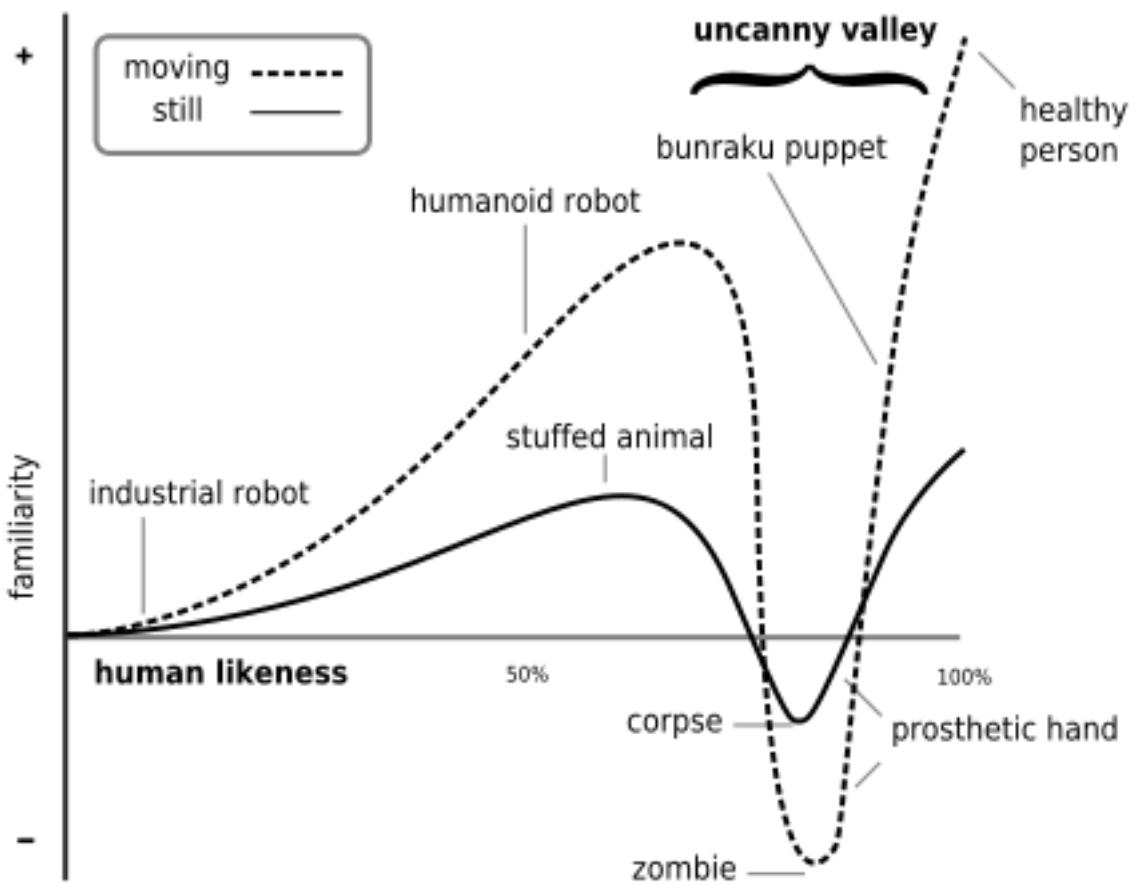


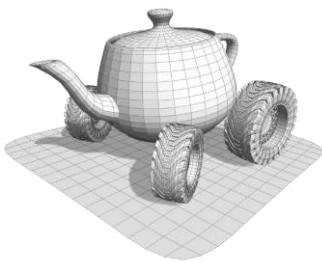
Final Fantasy – Squaresoft 2001



Avatar – ILM 2009

Uncanny Valley



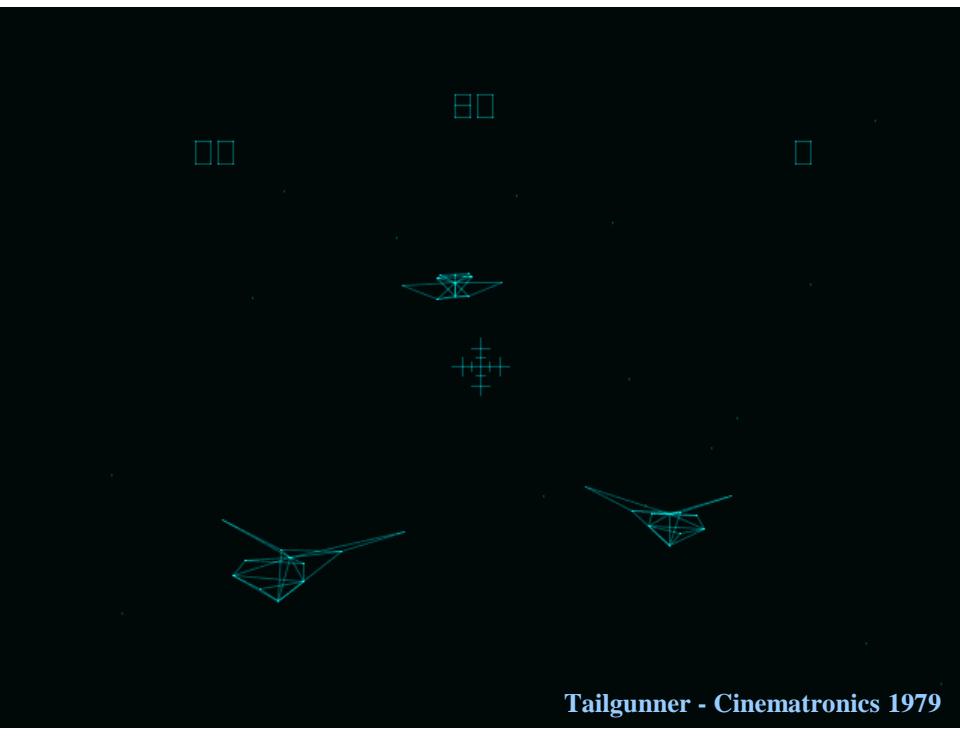


Computer Graphics: applications

- Entertainment: games, the leading business \$\$\$

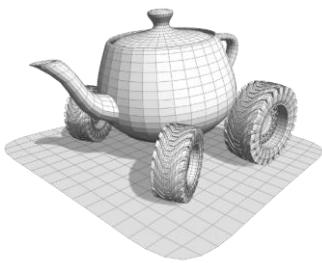


Battlezone – Atari 1980



Tailgunner - Cinematronics 1979

41



Computer Graphics: applicazioni

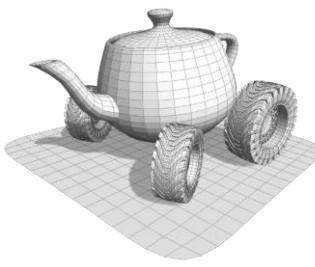
- Entertainment: games, the leading business \$\$\$



Chapter 1: What Computer Graphics

Virtua Fighter - Sega 1993



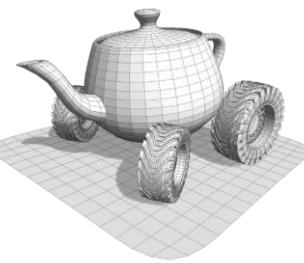


Computer Graphics: applicazioni

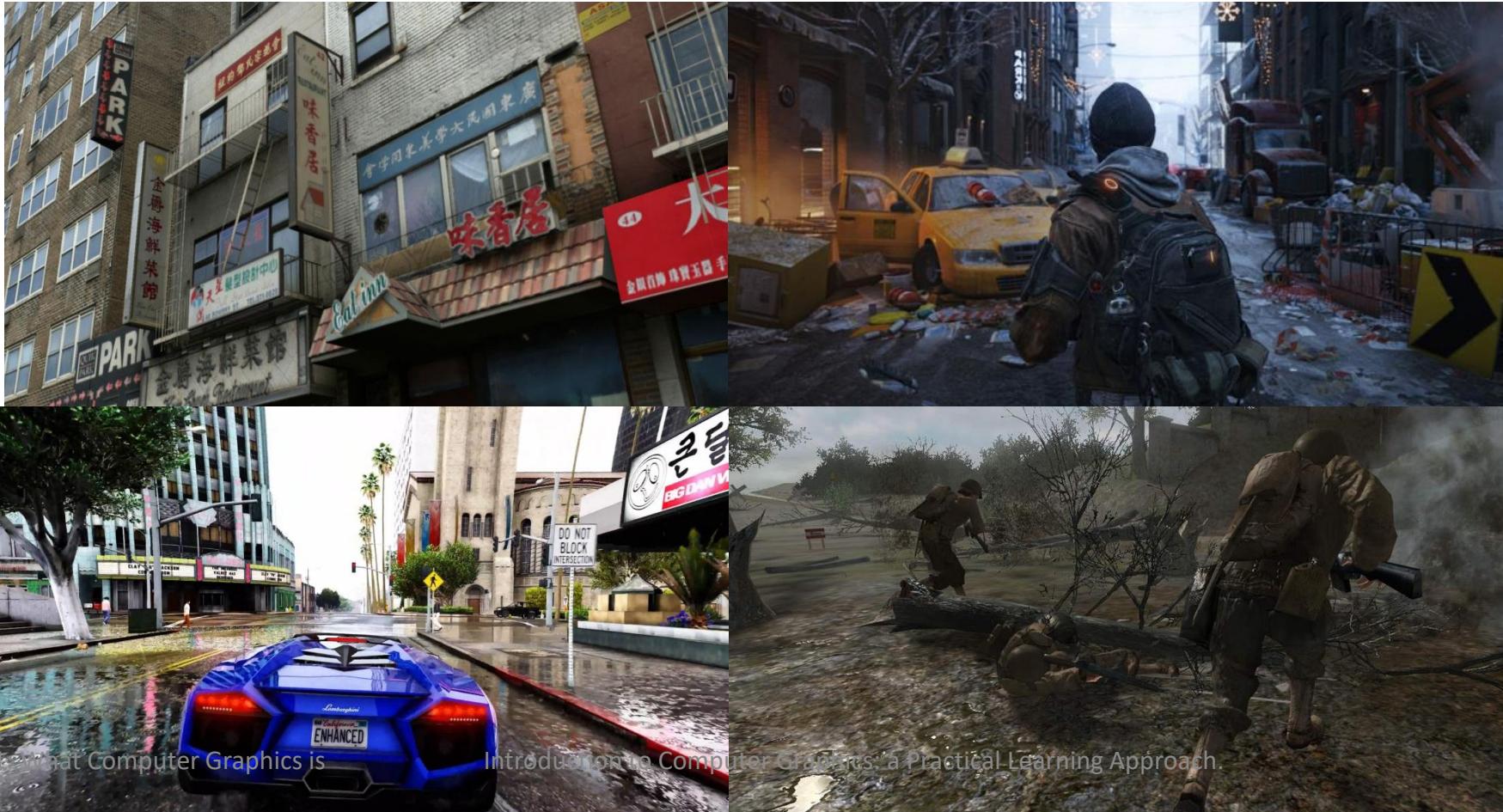
- Entertainment: games, the leading business \$\$\$



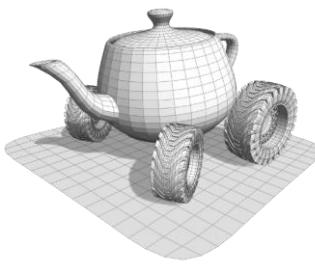
Applicazioni della CG



- Entertainment: games, the leading business \$\$\$

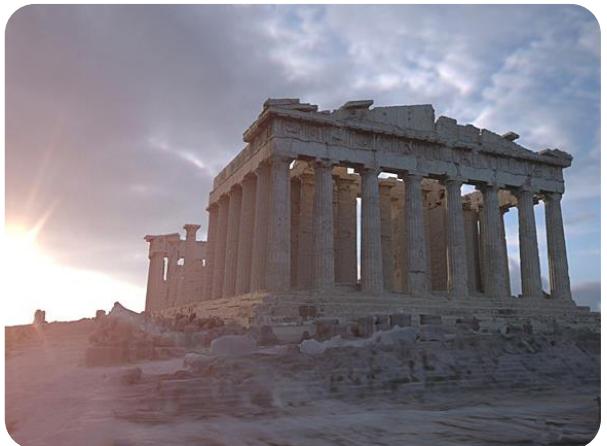
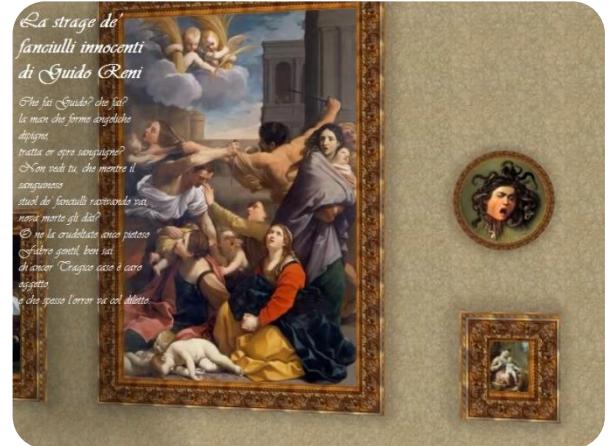
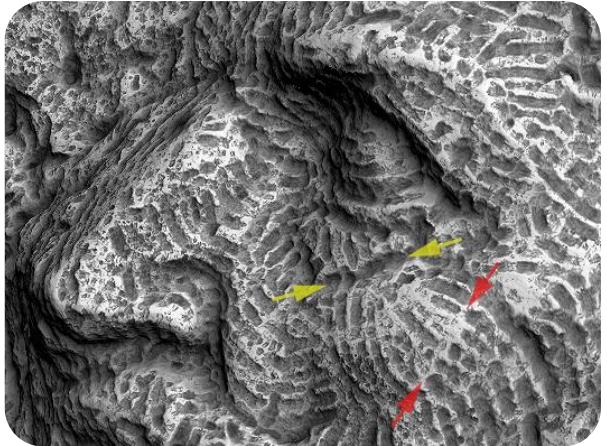
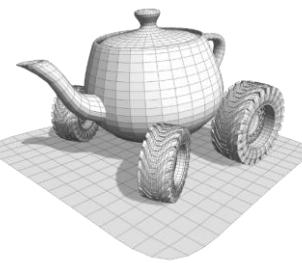


Computer Graphics: applications

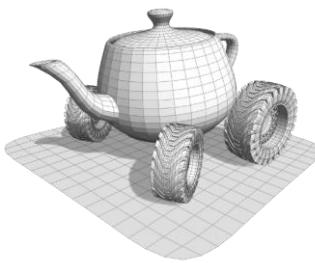


- Scientific
 - Scientific Visualization
 - Data Visualization
- E-Commerce
 - Product display
- Medicine
 - Diagnosis
 - Virtual Surgery, Telesurgery
- Manufacturing
 - Computer Aided Design
- Entertainment
 - Videogames
 - Movies
 - Visual Effects
 - CGI movies
- Cultural Heritage
 - Virtual Museums
 - Support to restoration
 - Supporto to Study
- Architecture
 - Design
 - Lighting Design
 - Preview

Computer Graphics for Cultural Heritage



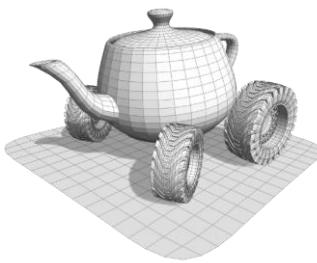
3D models in Cultural Heritage



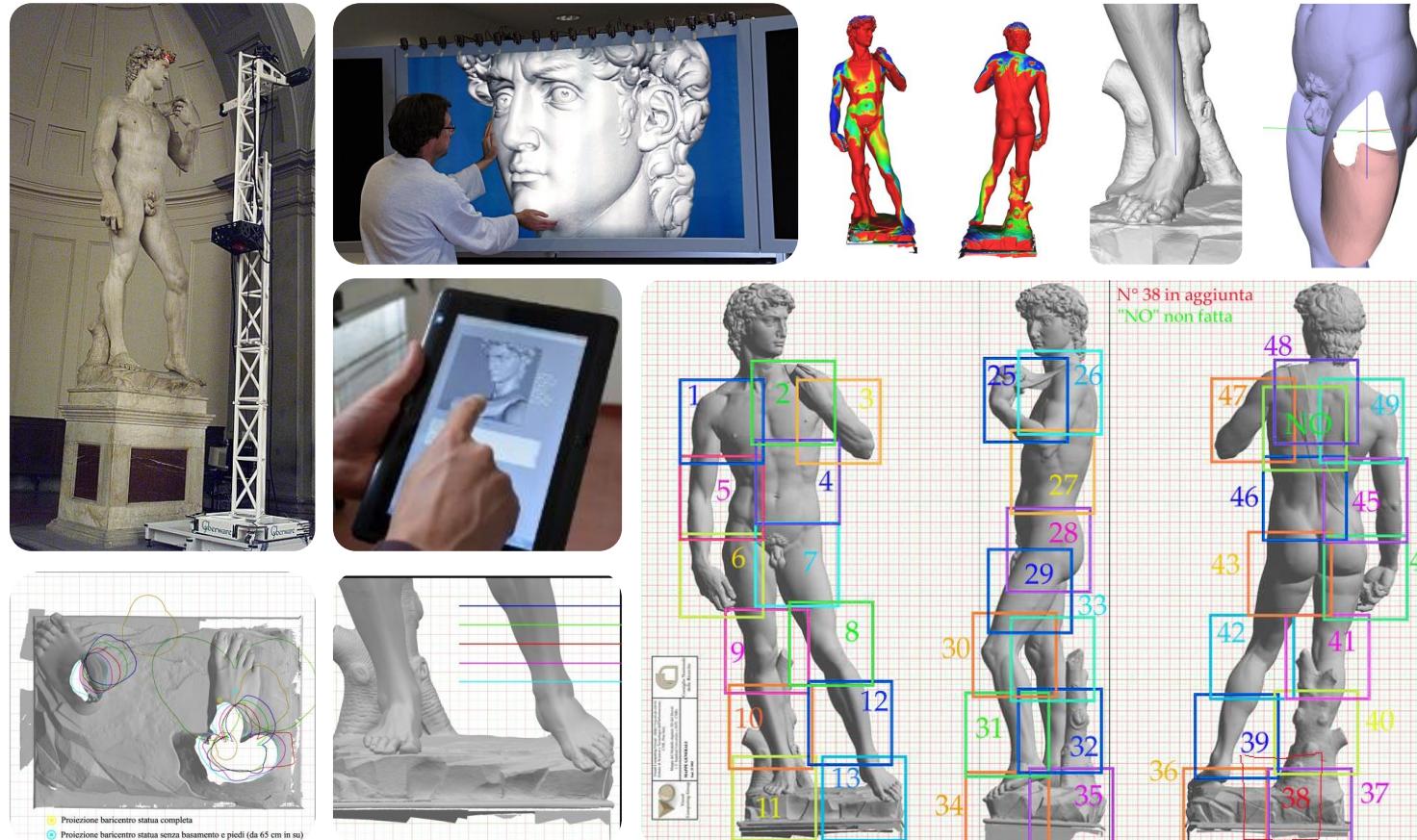
A tool for:

- Fruition / presentation
 - Virtual Museums, interactive kiosks in real museums, web , television...
- Archival
 - Documenting pieces of art
 - Domain onto which to associate other information
- Study
 - Support Restoration
 - Physical Simulation
 - Scientific Visualization

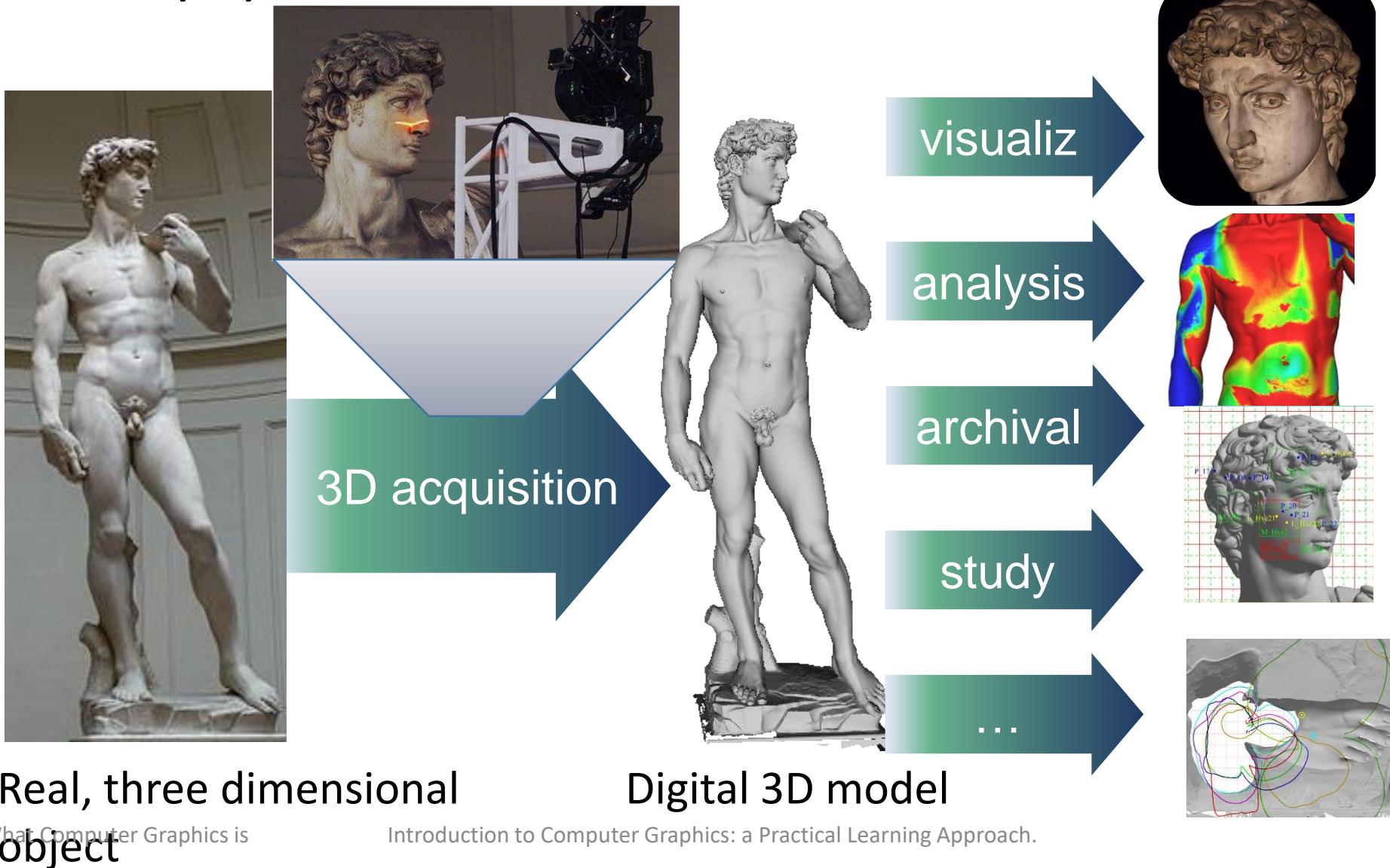
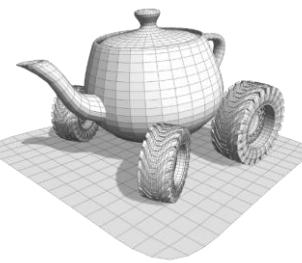
CG for Cultural Heritage

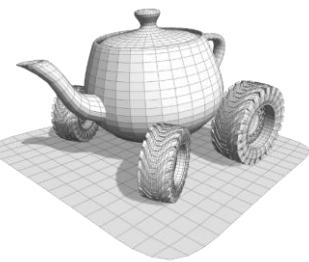


Case study: Michelangelo Project (<https://accademia.stanford.edu/mich/>)

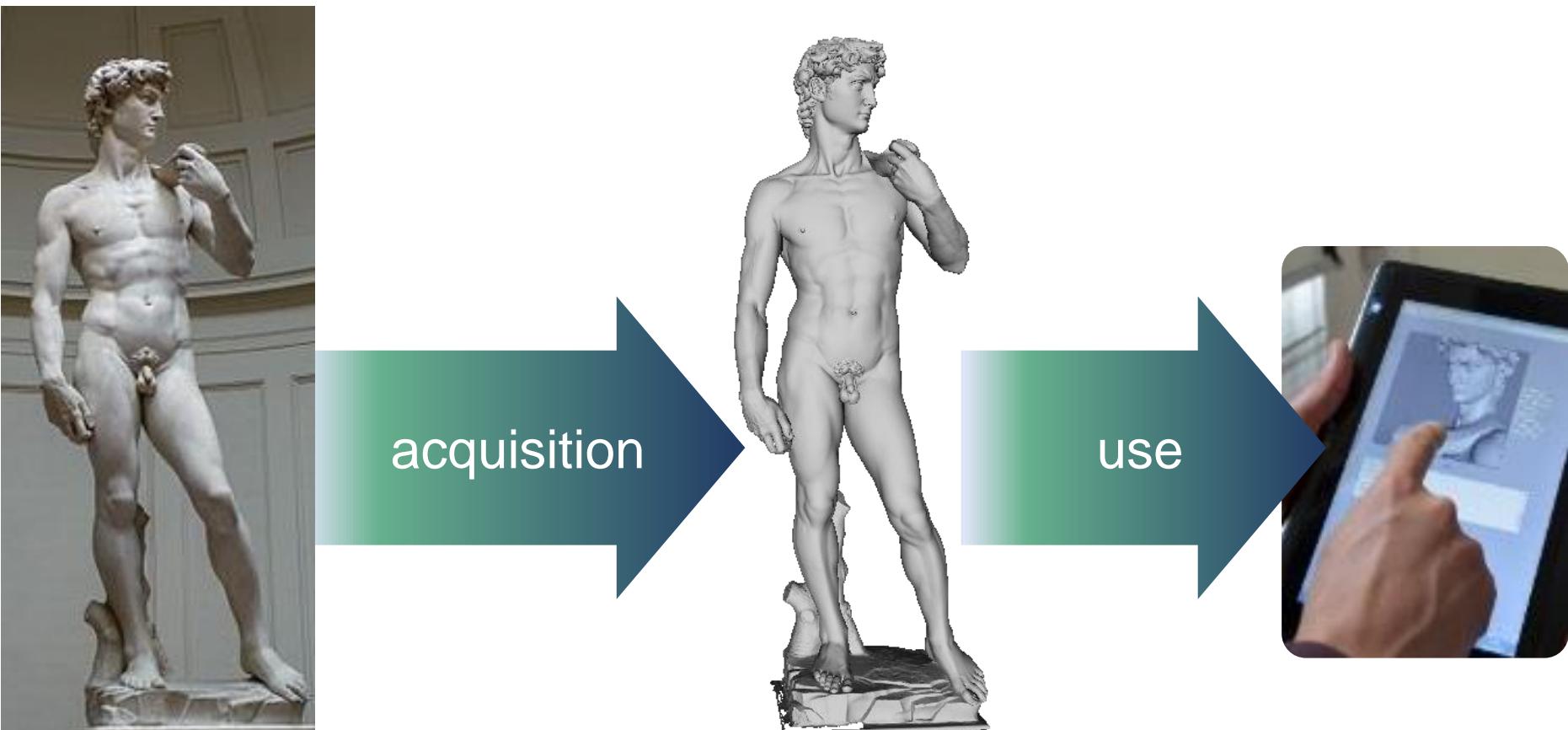


General pipeline scheme



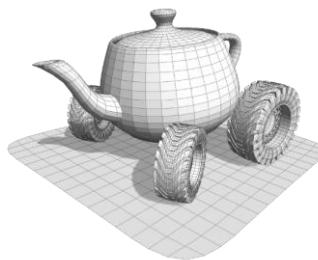


General pipeline scheme



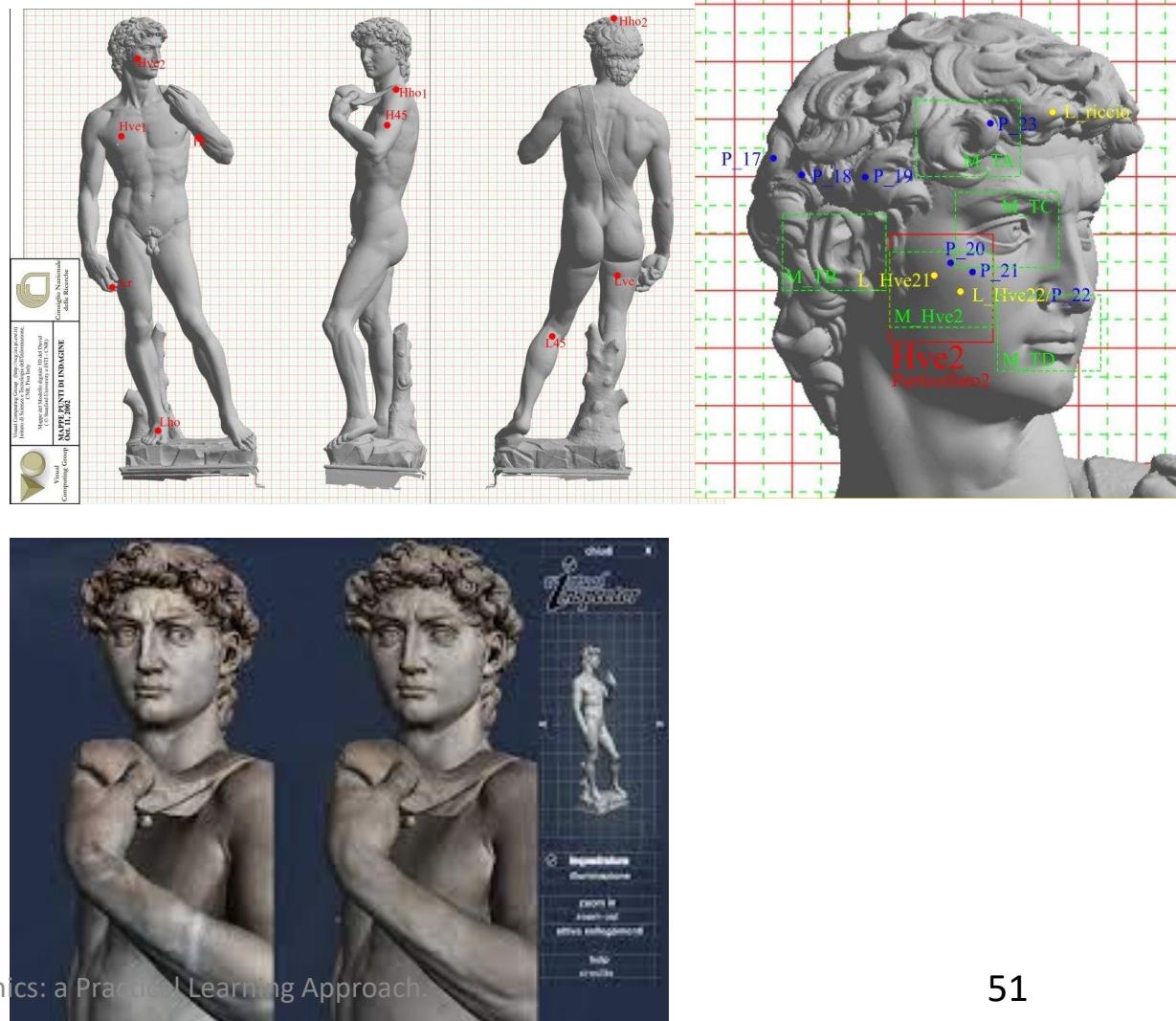
Real, three dimensional
object

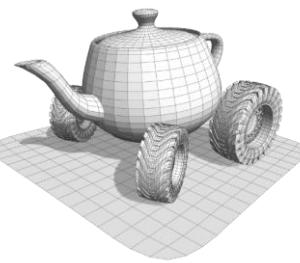
Digital 3D model



Cultural Heritage: Restoration

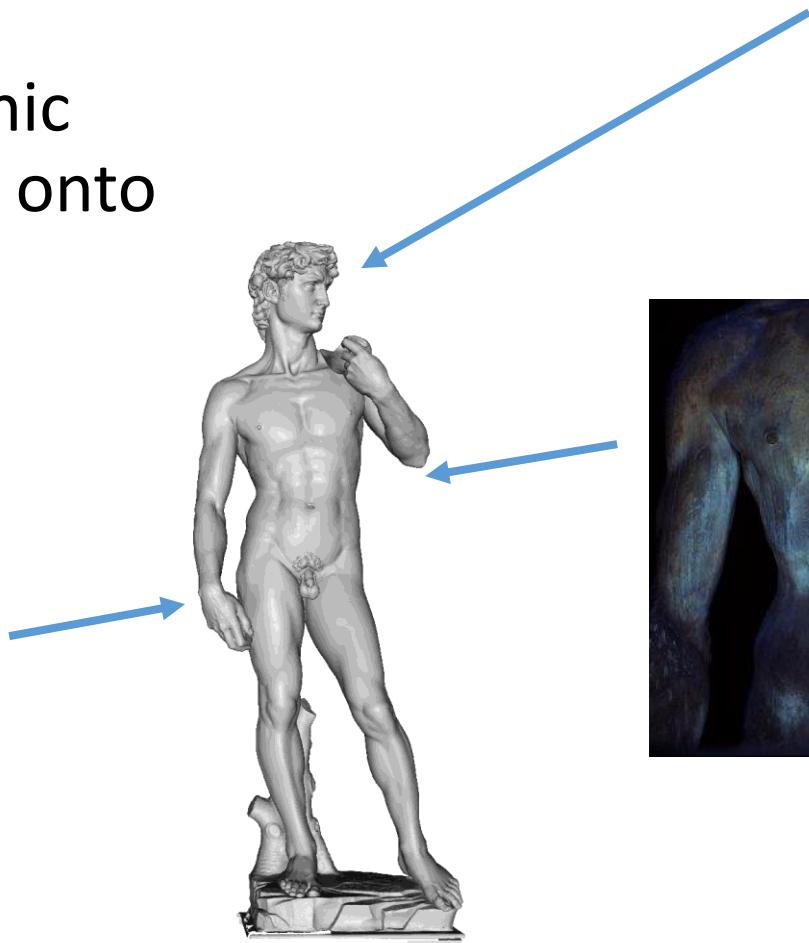
- Traditionally, the restorer had hand-made drawings where to annotate the work to do. This process is now digital
- The statue can be digitalized *before* and *after* the restoration and compared

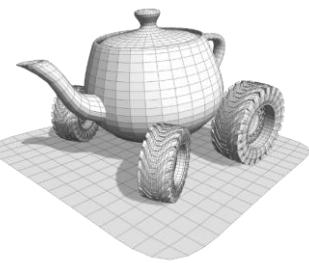




Cultural Heritage: analysis

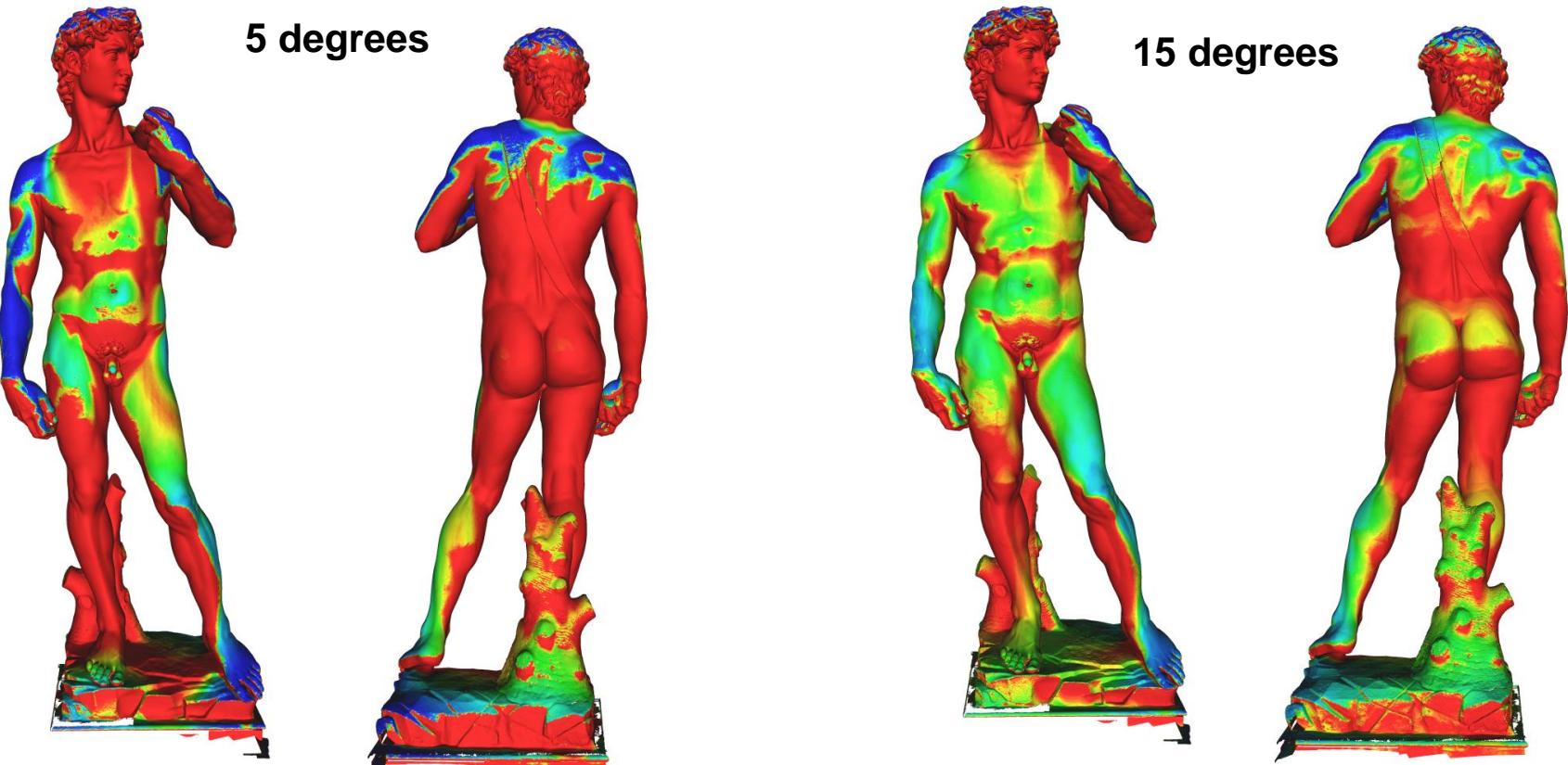
- Ultraviolet Images are acquired to detect organic deposits and «stitched» onto the mesh

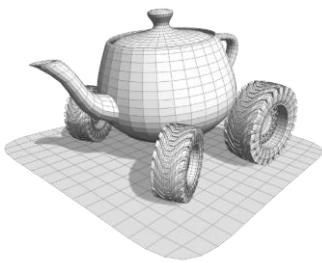




Cultural Heritage: Analysis

- Simulation of deposit of contaminants (dust)

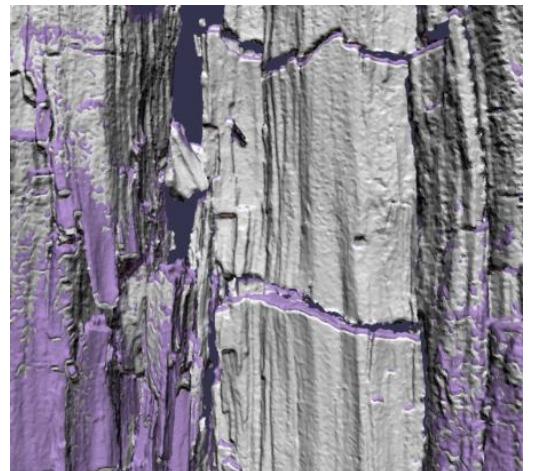




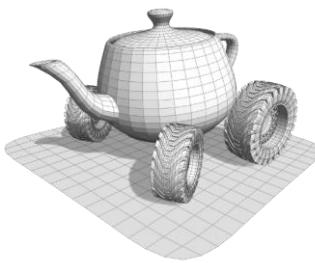
Cultural heritage: Monitoring

- monitoring deformable materials over time
- The Dunarobba Forest: 1 million old trunks suddenly discovered and exposed

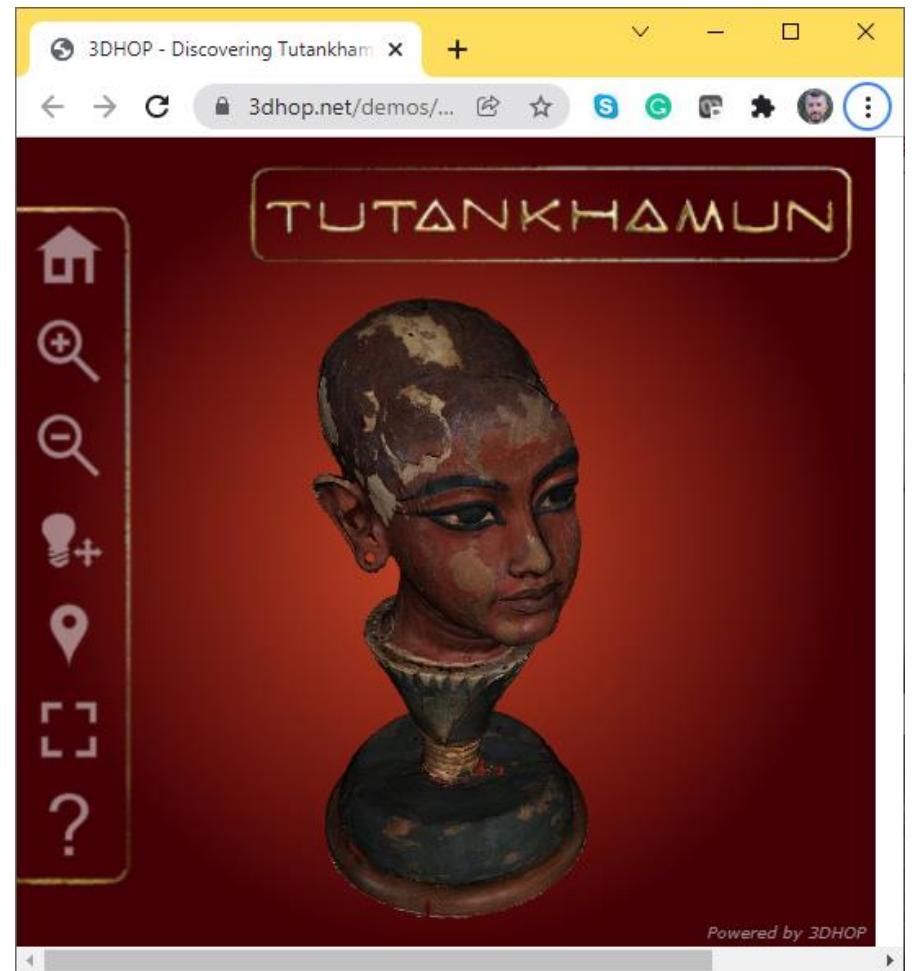
Dunarobba www.forestafossile.it

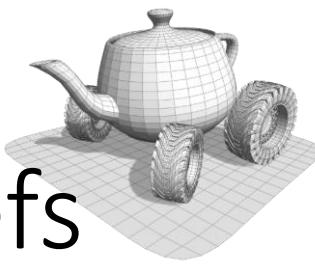


Computer Graphics: applications



- Cultural Heritage on the Web
- «3D Hop»Demo

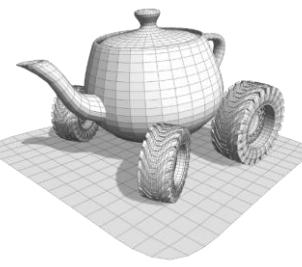




Computer Graphics: applications Base Reliefs

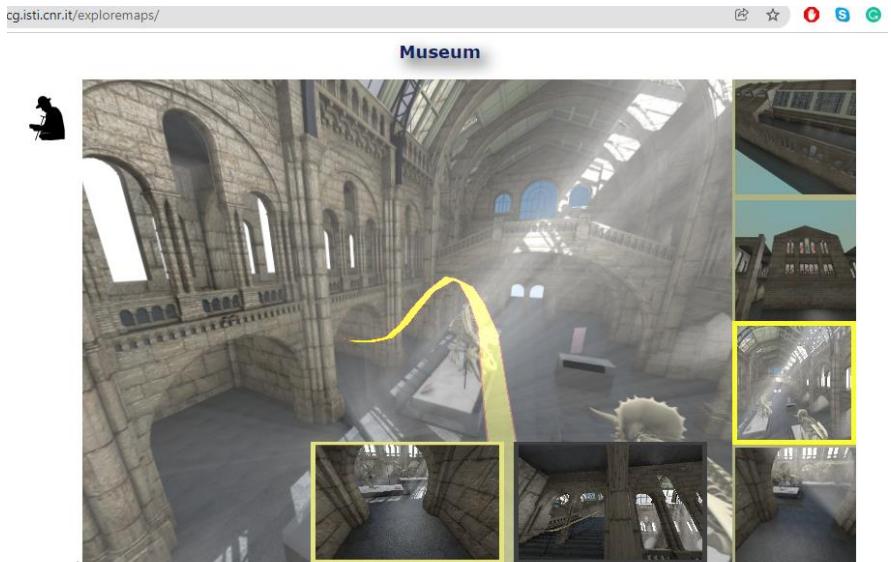
- Relightable Images (**RTI**)
- <http://vcg.isti.cnr.it/relight/comparison/ycc.html>
- No 3D data, just «how the baserelief would look if lit from that direction»
- It is how we perceive the «3D» of baserelief
- Fixed point of view, moving light

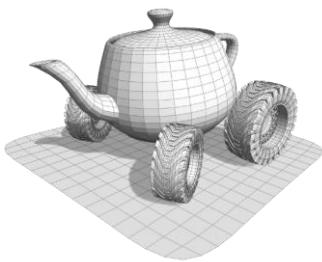




Computer Graphics: applications

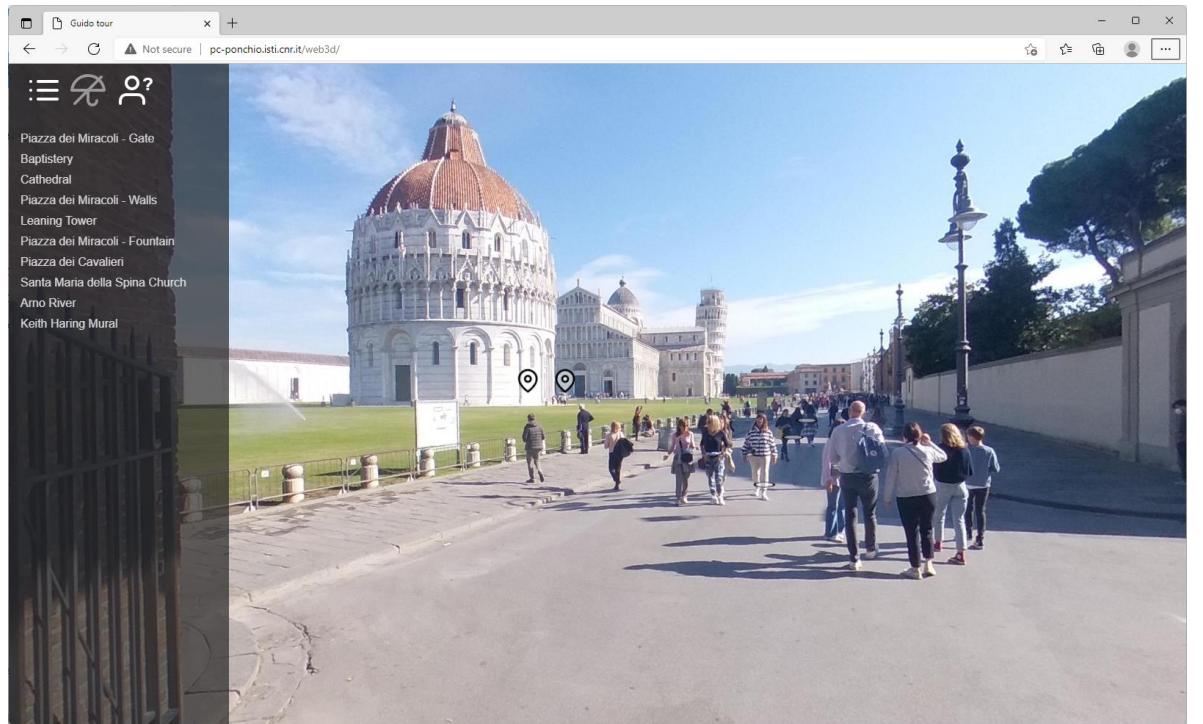
- Virtual Navigation
- eXploreMaps Demo
- No 3D data, only images and videos pre-rednered + constrained position of the observer

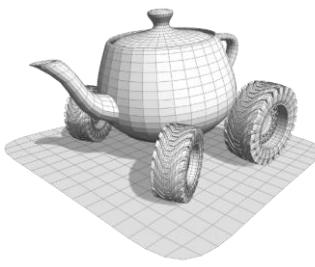




Computer Graphics: applications

- Virtual Navigation II
- [Guido demo](#)
- No 3D data, only
- Guide & followers paradigm
- ..under development..

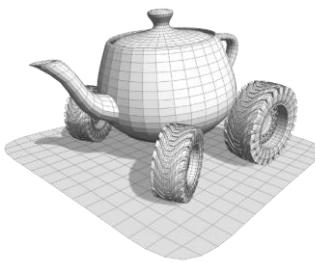




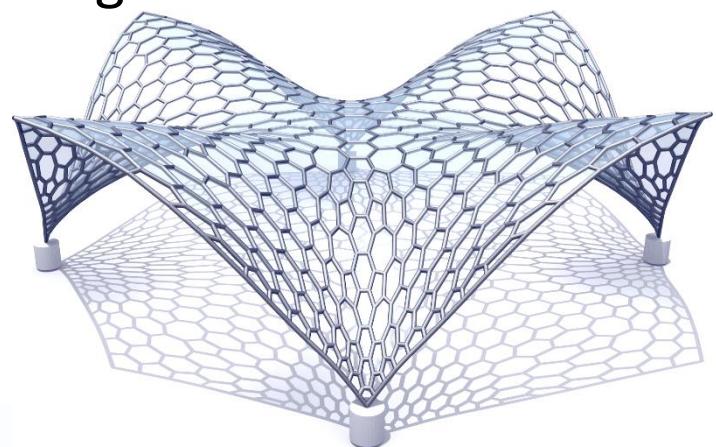
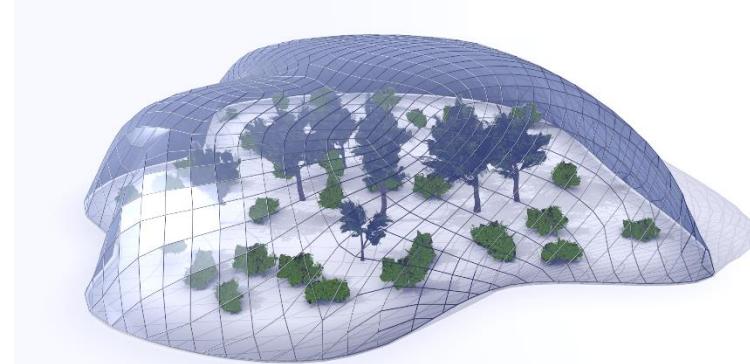
Computer Graphics: applications

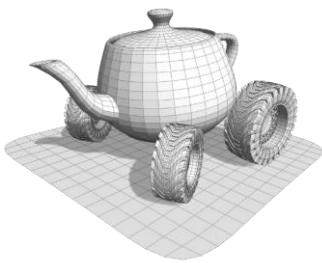
- Scientific
 - Scientific Visualization
 - Data Visualization
- E-Commerce
 - Product display
- Medicine
 - Diagnosis
 - Virtual Surgery, Telesurgery
- Manufacturing
 - Computer Aided Design
- Entertainment
 - Movies
 - Visual Effects
 - CGI movies
 - Videogames
- Cultural Heritage
 - Virtual Museums
 - Support to restoration
 - Supporto to Study
- Architecture
 - Design
 - Lighting Design
 - Preview

Computer Graphics: applicazioni



- Architecture:
 - Support to design





Computer Graphics: applications

- Architecture:

- preview:

- Communication
 - Evaluation
 - Lights design



(a) lighter paints in scene



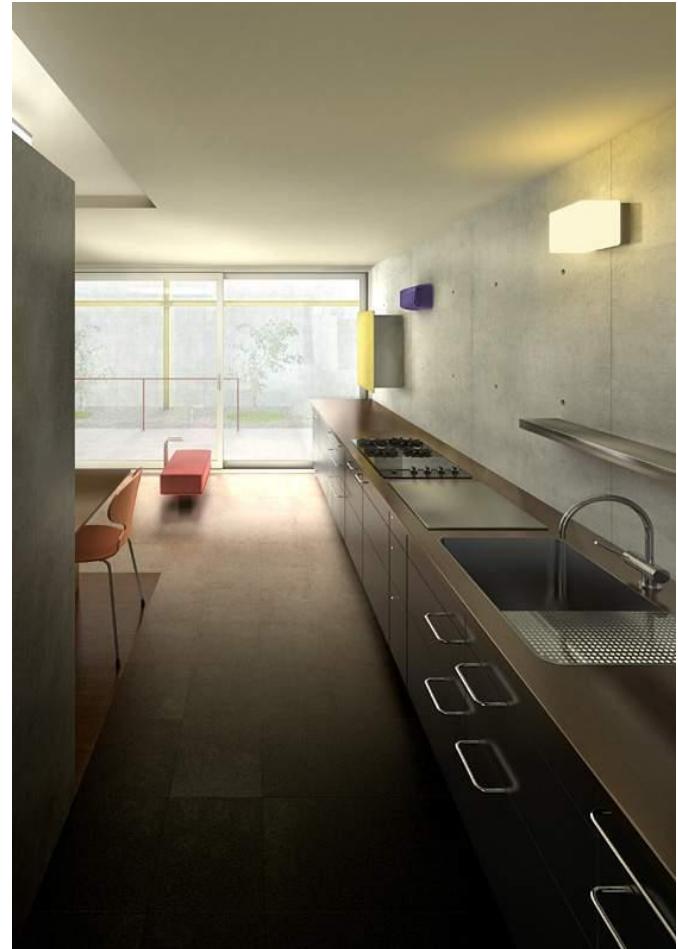
(b) computer places key light

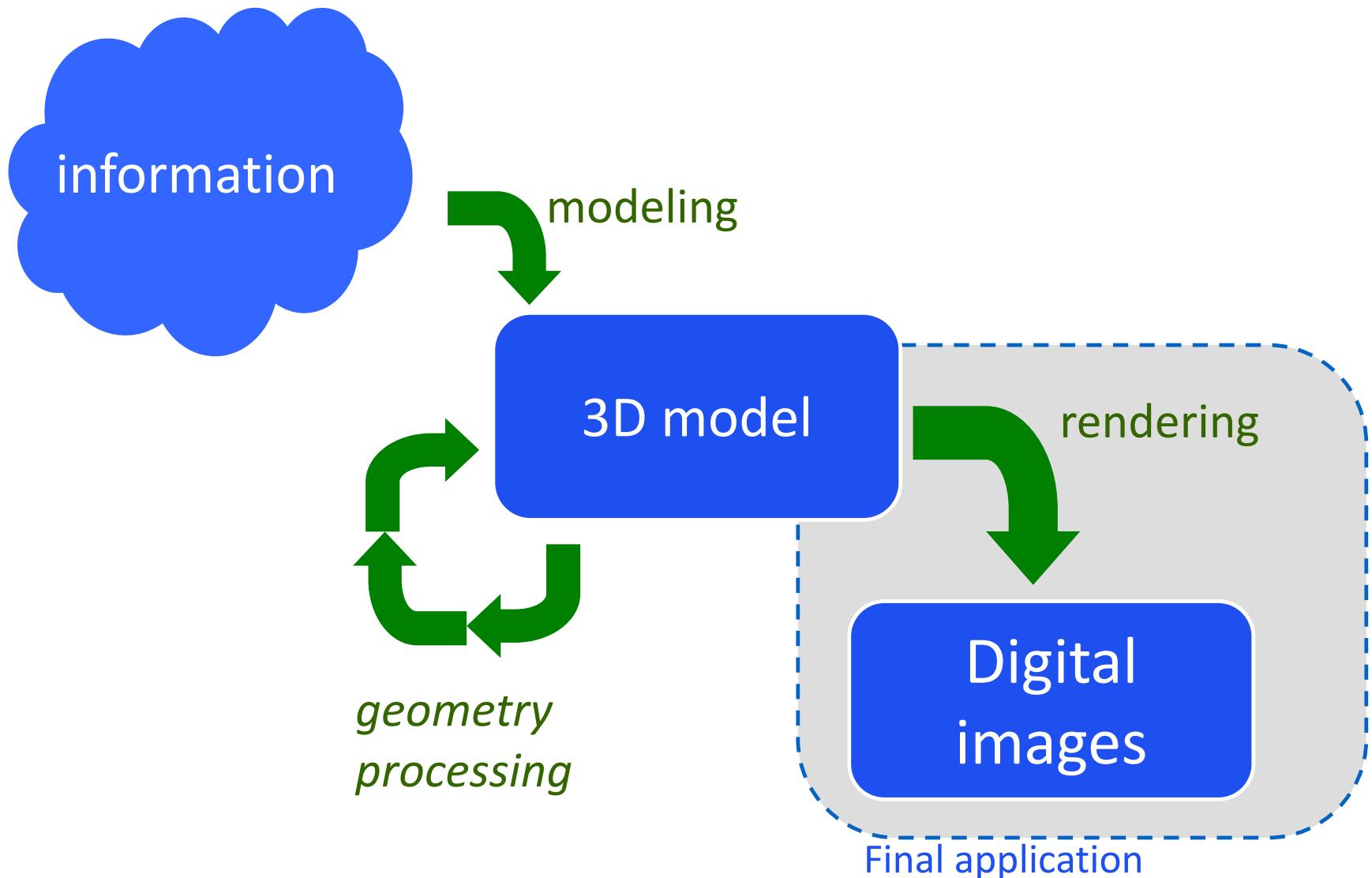
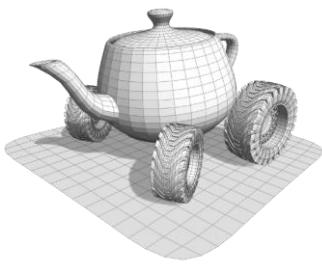


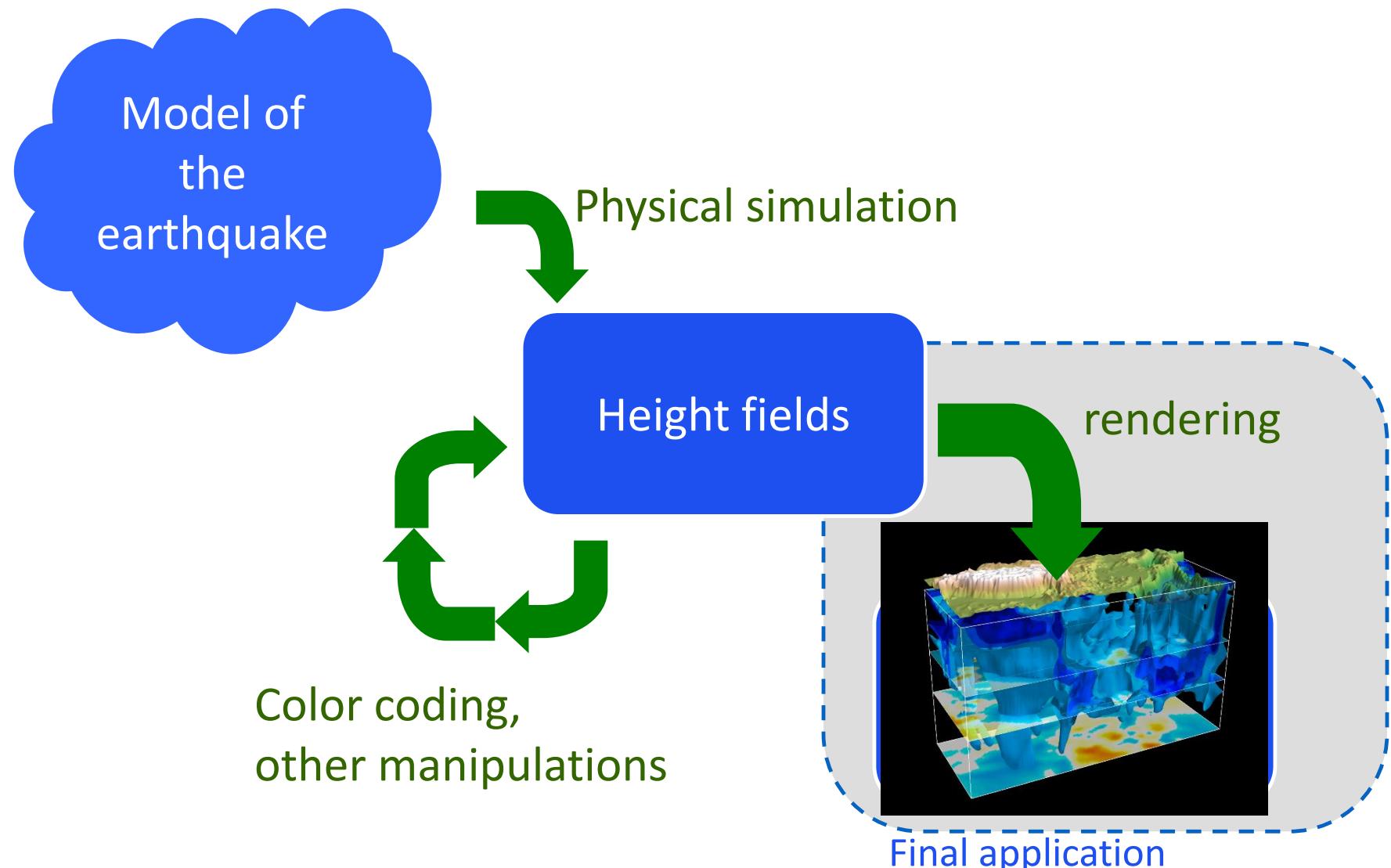
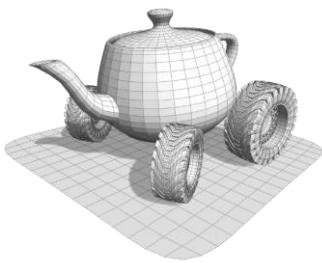
(c) lighter paints more

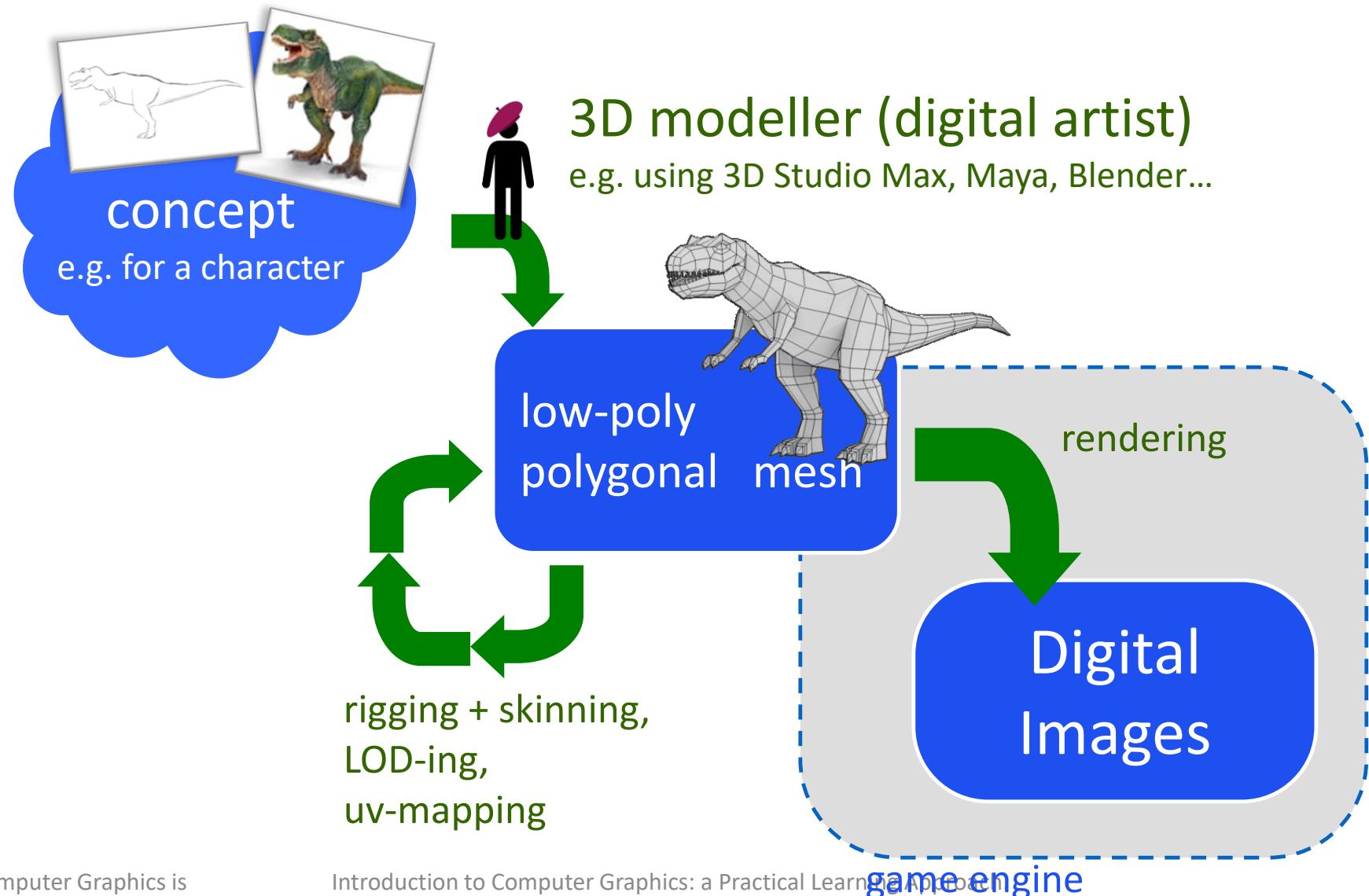
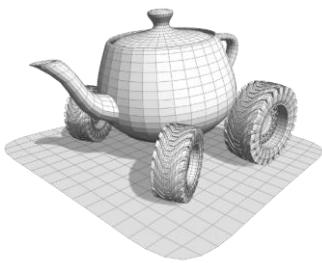


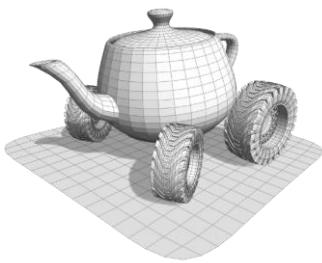
(d) final lit scene





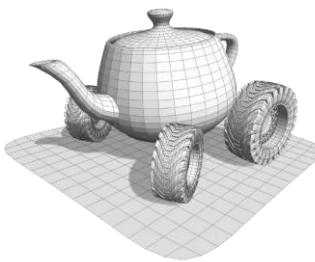






Rendering: a very general term!





Rendering: a very general term!

- Example: in web browsers

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>SIGGRAPH 2005 | Homepage</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="stylesheet" href="index.css" type="text/css" />
<link rel="SHORTCUT ICON" href="favicon.ico" />
<script type="text/javascript" src="flash_detect5.js"></script>
<script type="text/javascript">
<!--

function MM_swapImgRestore() { //v3.0
    var i,x,a=document.MM_sr;
    for(i=0;a&&i<a.length&&(x=a[i])&&x.oSrc;i++) x.src=x.oSrc;
}

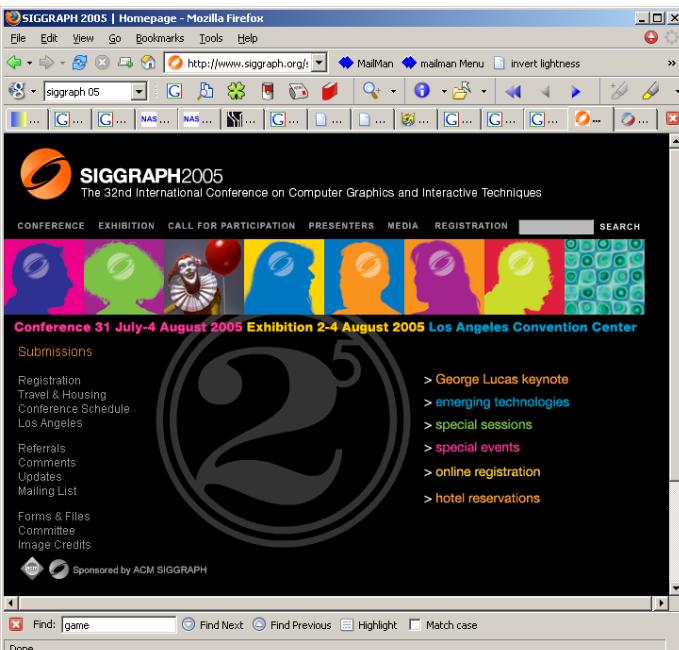
function MM_swapImage() { //v3.0
    var i,j=0,x,a=MM_swapImage.arguments; document.MM_sr=new Array;
    for(i=0;i<(a.length-2);i+=3)
        if ((x=MM_findObj(a[i]))!=null)(document.MM_sr[j++]=x; if(!x.oSrc)
x.oSrc=x.src; x.src=a[i+2]);
}

function MM_findObj(n, d) { //v4.01
    var p,i,x; if(!d) d=document;
    if((p=n.indexOf("?"))>0&&parent.frames.length) {
        d=parent.frames[n.substring(p+1)].document; n=n.substring(0,p);
        if(!(x=d[n])&&d.all) x=d.all[n]; for (i=0;i<d.forms.length;i++)
x=d.forms[i][n];
    }
    ...
}
```

HTML page + images, css, etc

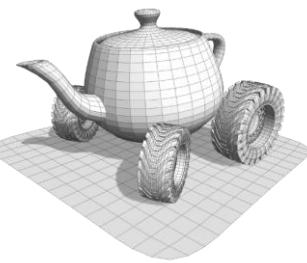
(which models a Web Page)

rendering



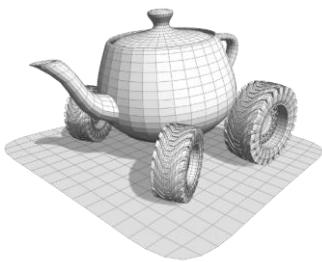
the image appearing in
the browser

Rendering



- In our context





Rendering

- Rendering 3D

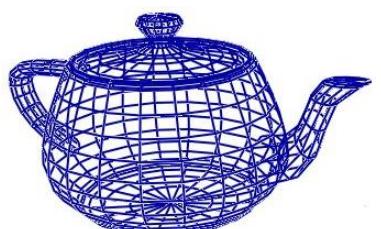
```
Rim: { 102, 103, 104, 105, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }
Body: { 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 }
Lid: { 96, 96, 96, 96, 97, 98, 99, 100, 101, 101, 101, 0, 1, 2, 3 }
{ 0, 1, 2, 3, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117 }
Handle: { 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56 }
{ 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 28, 65, 66, 67 }
Spout: { 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83 }
{ 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95 }

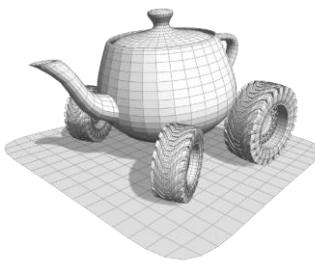
Vertices:
{ 0.2000, 0.0000, 2.70000 }, { 0.2000, -0.1120, 2.70000 }, { 0.1120, -0.2000, 2.70000 }
{ 0.0000, -0.2000, 2.70000 }, { 0.13375, 0.0000, 2.53125 }, { 1.3375, -0.7490, 2.53125 }
{ 0.7490, -1.3375, 2.53125 }, { 0.0000, -1.3375, 2.53125 }, { 1.4375, 0.0000, 2.53125 }
{ 1.4375, -0.8050, 2.53125 }, { 0.8050, -1.4375, 2.53125 }, { 0.0000, -1.4375, 2.53125 }
{ 1.5000, 0.0000, 2.40000 }, { 1.5000, -0.8400, 2.40000 }, { 0.8400, -1.5000, 2.40000 }
{ 0.0000, -1.5000, 2.40000 }, { 1.7500, 0.0000, 1.87500 }, { 1.7500, -0.9800, 1.87500 }
{ 0.9800, -1.7500, 1.87500 }, { 0.0000, -1.7500, 1.87500 }, { 2.0000, 0.0000, 1.35000 }
{ 2.0000, -1.1200, 1.35000 }, { 1.1200, -2.0000, 1.35000 }, { 0.0000, -2.0000, 1.35000 }
{ 2.0000, 0.0000, 0.90000 }, { 2.0000, -1.1200, 0.90000 }, { 1.1200, -2.0000, 0.90000 }
{ 0.0000, -2.0000, 0.90000 }, { -2.0000, 0.0000, 0.90000 }, { 2.0000, 0.0000, 0.45000 }
{ 2.0000, -1.1200, 0.45000 }, { 1.1200, -2.0000, 0.45000 }, { 0.0000, -2.0000, 0.45000 }
{ 1.5000, 0.0000, 0.22500 }, { 1.5000, -0.8400, 0.22500 }, { 0.8400, -1.5000, 0.22500 }
{ 0.0000, -1.5000, 0.22500 }, { 1.5000, 0.0000, 0.15000 }, { 1.5000, -0.8400, 0.15000 }
{ 0.8400, -1.5000, 0.15000 }, { 0.0000, -1.5000, 0.15000 }, { -1.6000, 0.0000, 0.202500 }
{ -1.6000, -0.3000, 0.202500 }, { -1.5000, -0.3000, 0.202500 }, { -1.5000, 0.0000, 0.225000 }
{ -2.3000, 0.0000, 0.202500 }, { -2.3000, -0.3000, 0.202500 }, { -2.5000, -0.3000, 0.225000 }
{ -2.5000, 0.0000, 0.225000 }, { -2.7000, 0.0000, 0.202500 }, { -2.7000, -0.3000, 0.202500 }
{ -3.0000, -0.3000, 0.202500 }, { -3.0000, 0.0000, 0.225000 }, { -2.7000, 0.0000, 1.80000 }
{ -2.7000, -0.3000, 1.80000 }, { -3.0000, -0.3000, 1.80000 }, { -3.0000, 0.0000, 1.80000 }
{ -2.7000, 0.0000, 1.57500 }, { -2.7000, -0.3000, 1.57500 }, { -3.0000, -0.3000, 1.35000 }
{ -3.0000, 0.0000, 1.35000 }, { -2.5000, 0.0000, 1.25000 }, { -2.5000, -0.3000, 1.25000 }
```

3D rendering



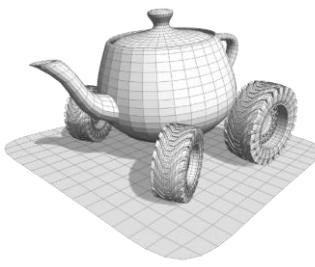
- 3D model
- 3D point
 - primitives...





Rendering Algorithms

- Two types:
 - On-Line Rendering
 - Interactive: 1 – 10 frames per sec ("FPS")
 - Real-Time: 25-100 fps
 - Off-line Rendering
 - From minutes to hours per frame
- Very different:
 - On applications
 - On constraints
 - On visual quality (photorealism)
 - On type and resolution of 3D models used.
- ...or are they??



Real Time VS Offline rendering

- Less and less different...

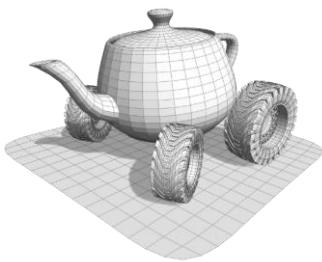
1993



real time



offline



Real Time VS Offline rendering

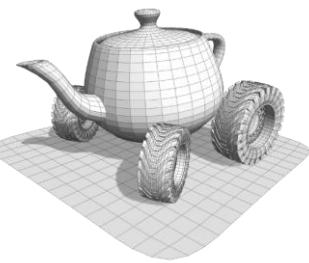
- Less and less different...

2001



real time

offline



Real Time VS Offline rendering

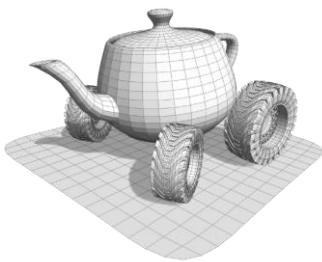
- Less and less different...

2007



real time

offline



Real Time VS Offline rendering

- Less and less different...

2021



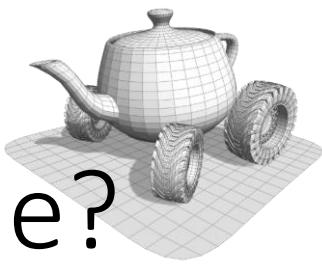
The Matrix Awakens (Unreal engine 5)



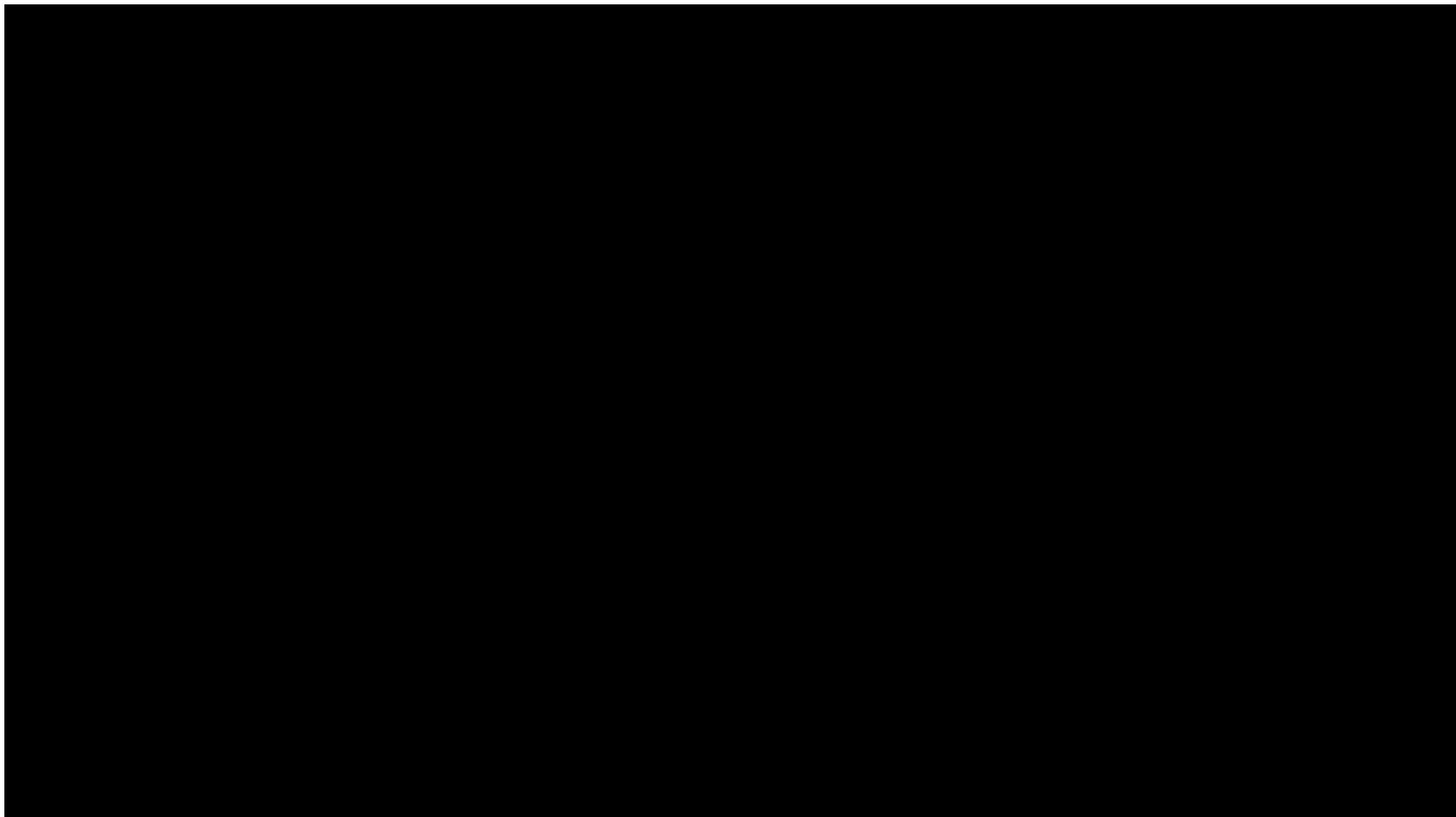
Matrix Resurrections (Warner Bros)

real time

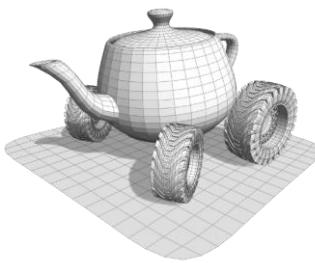
offline



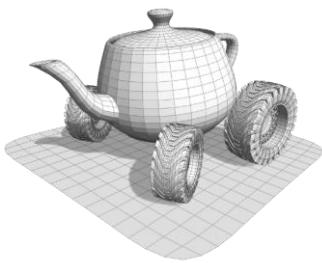
Real Time VS Offline rendering: and this one?



Tools for Computer Graphics



- Approximations/simplifications
 - Of the entities involved
 - Illumination models
 - Fineness of the description of the objects in the scene
 - Of their interactions
 - Light – material (for rendering)
 - Object-object (contact/collision detection)
- Data structures & Algorithms
 - organization of space and entities
 - Hierarchies are a major tool
 - data caching, data loading ...
 - Adaptivity: make so that in the time budget a meaningful rendering is provided

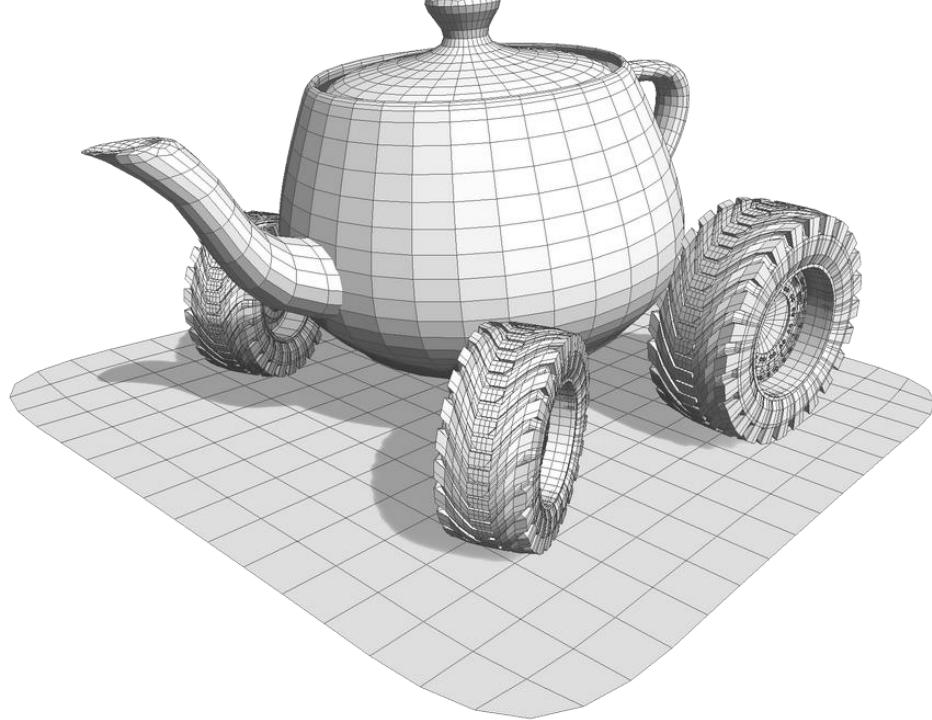


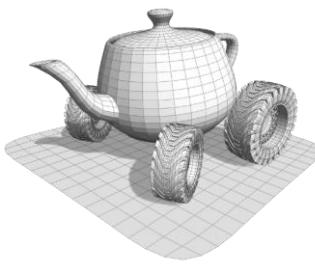
What in this course?

- This is an **introductory** course: hardcore concepts of Computer Graphics
- Will we be able to make a game like the demo we have just seen?
 - No, if only for **3D models and material design**
- We will cover *almost* everything you see in a modern videogame but do not be fooled: **digital assets play a very important role**

Setup

Let's start!

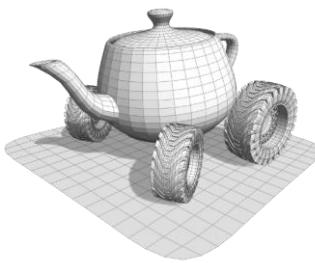




Our Tools (1/3): Environment

- Any IDE for developing with C++
 - **MS Visual Studio** (I'll use this)
 - QTCreator (MS/Linux/Mac)
 - ... whatever you like
- Language
 - C++ is not mandatory but strongly encouraged
 - There are OpenGL API for Javascript, Python, Java, C#, Fortran, Pascal...
 - For most part, there won't be much to do on CPU side, it will be almost all GPU side

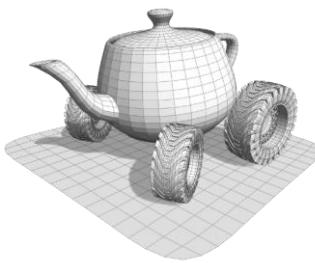
Our Tools (2/3): Libraries



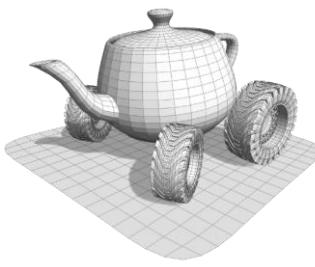
today

- **OpenGL** (www.opengl.org) for programming the graphics pipeline
- **GLFW** (Graphics Library FrameWork) for providing a window to draw to and handling keyboard and mouse events
 - There are alternatives: Glut /freeglut, wxWidgets, Qt, VTK
 - ..but GLFW is currently the de-facto standard in game industry
- **Dear ImGui** for the GUI (buttons, sliders etc..)
 - There are alternatives: AntTweakBar, ..
 - .. But Dear ImGui is currently the de-facto standard in game industry
 - We may not even use a library for a GUI
- **GLM** (header-only library for math)
- **TinyObjLoader** or **VCGLib** for loading 3D models

Our Tools (3/3): version control system



- **Git**
 - The course repository is on **Github** (www.github.com)
<https://github.com/ganovelli/CGCourse2023/tree/main>
 - The repository will be updated by me, you are requested to fork it and to work on your forked version, pulling from the main repo when necessary (that is, when I add something you need)
 - Note: no need to be proficient with git, you will need only very basic commands (commit/push/pull).
- GUIs for Git: **GitHub Desktop**, GitKraken, Sourcetree, Tortoise Git, SmartGit.



Let's setup: fect the repository

For the repository

<https://github.com/ganovelli/CGCourse2023>

CGCourse2023

-- 3dparty
readme.txt

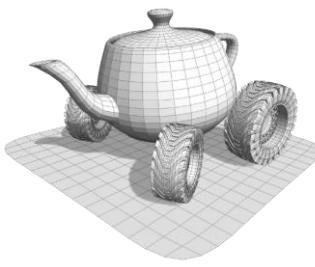
-- src

-- code_0_setup
main.cpp
readme.txt

The screenshot shows a GitHub repository page for 'ganovelli/CGCourse2023'. The repository is public and has 1 star, 1 watch, and 1 fork. It contains 2 branches and 0 tags. The commit history shows the following activity:

Author	Commit Message	Time	Commits
ganovelli	Create readme.txt	8 minutes ago	6 commits
	3dparty	8 minutes ago	
	src/code_0_setup	10 minutes ago	
	.gitignore	2 weeks ago	
	README.md	2 weeks ago	

The sidebar on the right displays repository statistics: Computer Grafica Unip (1 star), Readme (1 watching), and 1 fork. A prominent red circle highlights the 'Create a new fork' button in the sidebar.



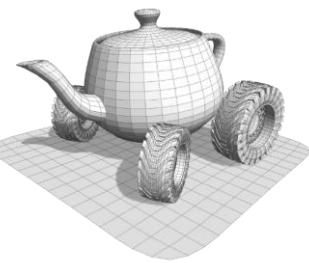
Let's setup: fetch GLFW

- Download GLFW from <https://www.glfw.org>
 - You can download the precompiled binaries and
 - Or you can download the source code and con
- Place .h and .lib in the 3dparty folder:

CGCourse2023

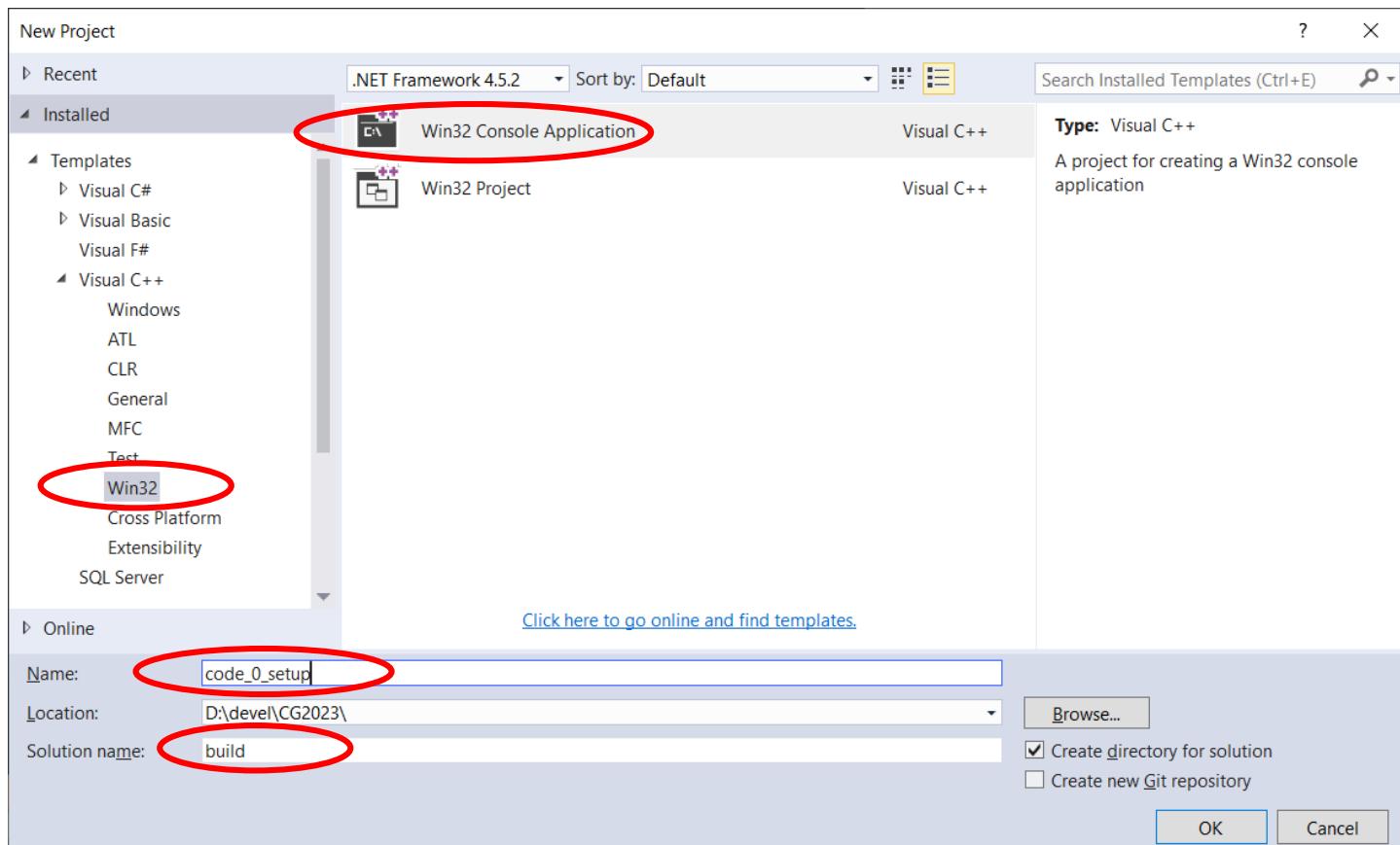
```
-- 3dparty
  -- glfw
    -- include
    -- lib
```

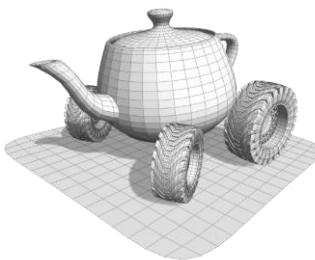
Name	Type
docs	File folder
include	File folder
lib-mingw-w64	File folder
lib-static-ucrt	File folder
lib-vc2012	File folder
lib-vc2013	File folder
lib-vc2015	File folder
lib-vc2017	File folder
lib-vc2019	File folder
lib-vc2022	File folder
LICENSE.md	MD File
README.md	MD File



Let's setup: create a project (1/2)

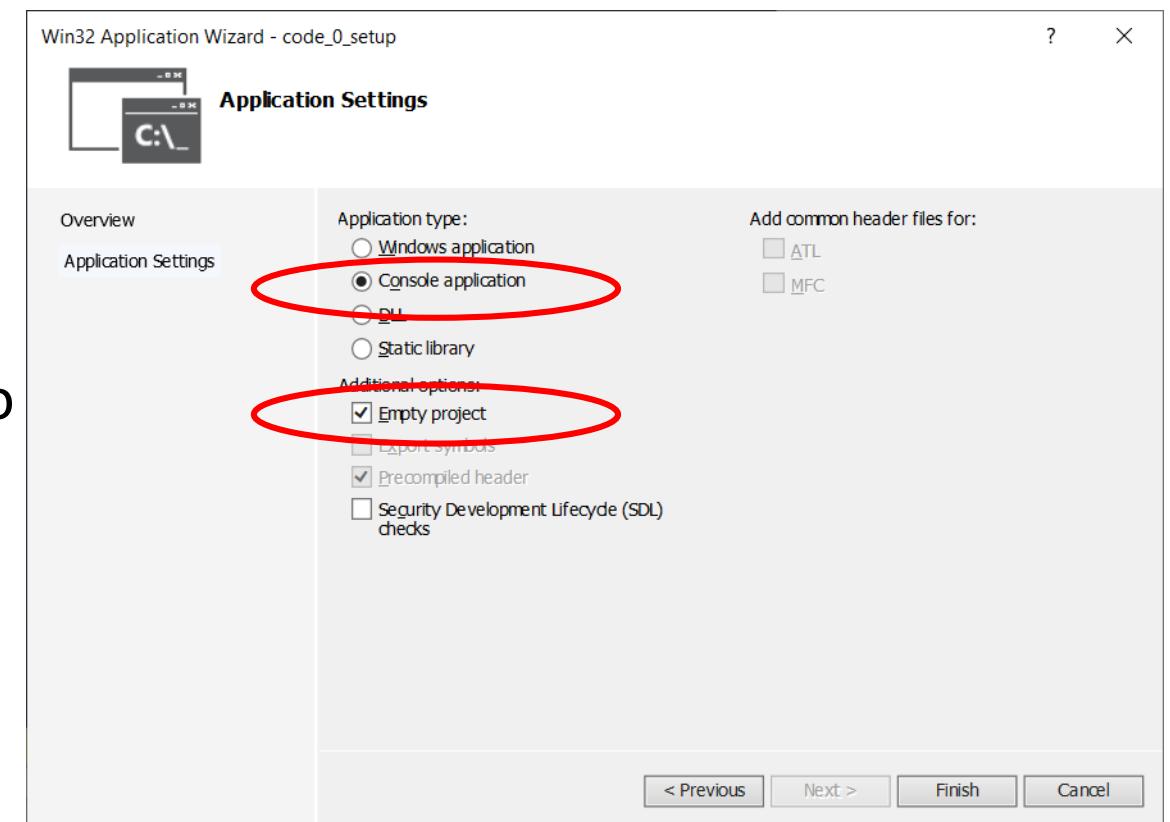
- Create a win32 console application c++ project



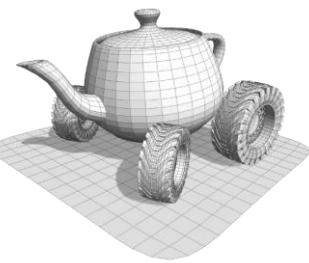


Let's setup: create a project (2/2)

- Create a win32 console application c++ project
- In the project properties:
 - Add the file main.cpp to the source
 - Add glfw/include to the include path
 - Add glfw/lib to the library path
 - Add the glfw3.lib and opengl32.lib to



IT'S DONE !



Let's setup: set include paths

code_0_setup Property Pages

Configuration: Debug Platform: x64 Configuration Manager...

Additional Include Directories: ..\3dparty\glfw\include

Program Database for Edit And Continue (/ZI)

Additional #using Directories

Debug Information Format

Common Language RunTime Support

Consume Windows Runtime Extension

Suppress Startup Banner

Warning Level: Yes (/nologo)

Treat Warnings As Errors: Level3 (/W3)

Warning Version: No (/WX-)

SDL checks: Yes (/sdl)

Multi-processor Compilation

General

Optimization

Preprocessor

Code Generation

Language

Precompiled Headers

Output Files

Browse Information

Advanced

All Options

Command Line

Linker

General

Input

Manifest File

Debugging

System

Optimization

Embedded IDL

Windows Metadata

Advanced

All Options

Manifest Tool

XML Document Generator

Browse Information

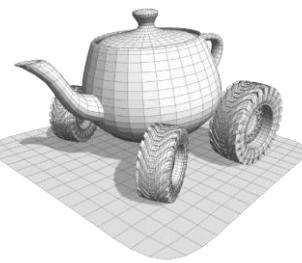
Build Events

Custom Build Step

Code Analysis

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

OK Cancel Apply



Let's setup: set libraries

code_0_setup Property Pages

Configuration: Debug Platform: x64 Configuration Manager...

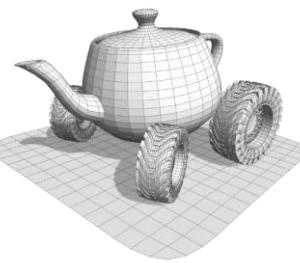
Additional Dependencies
Ignore All Default Libraries
Ignore Specific Default Libraries
Module Definition File
Add Module to Assembly
Embed Managed Resource File
Force Symbol References
Delay Loaded DLLs
Assembly Link Resource

opengl32.lib;glfw3.lib;%AdditionalDependencies%

Additional Dependencies
Specifies additional items to add to the link command line. [i.e. kernel32.lib]

OK Cancel Apply

Navigation pane:
Configuration Properties
General
Debugging
VC++ Directories
C/C++
 General
 Optimization
 Preprocessor
 Code Generation
 Language
 Precompiled Headers
 Output Files
 Browse Information
 Advanced
 All Options
 Command Line
Linker
 General
 Input
 Manifest File
 Debugging
 System
 Optimization
 Embedded IDL
 Windows Metadata
 Advanced
 All Options
 Command Line
 Manifest Tool
 XML Document Generator
 Browse Information
 Build Events
 Custom Build Step
 Code Analysis



Let's setup: set library path

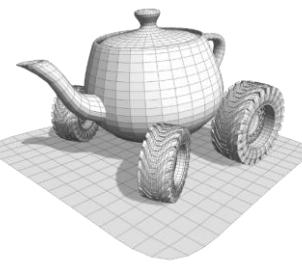
code_0_setup Property Pages

Configuration: Debug Platform: x64 Configuration Manager...

Configuration Properties	Output File	<code>\$(OutDir)\$(TargetName)\$(TargetExt)</code>
General	Show Progress	Not Set
Debugging	Version	
VC++ Directories	Enable Incremental Linking	Yes (/INCREMENTAL)
C/C++	SUPPRESS Startup Banner	Yes (/NOLOGO)
General	Ignore Import Library	No
Optimization	Register Output	No
Preprocessor	Per-user Redirection	No
Code Generation	Additional Library Directories	<code>..\..\3dparty\glfw\lib</code>
Language	Link Library Dependencies	Yes
Precompiled Headers	Use Library Dependency Inputs	No
Output Files	Link Status	
Browse Information	Prevent DLL Binding	
Advanced	Treat Linker Warning As Errors	
All Options	Force File Output	
Command Line	Create Hot Patchable Image	
Linker	Specify Section Attributes	
General		
Input		
Manifest File		
Debugging		
System		
Optimization		
Embedded IDL		
Windows Metadata		
Advanced		
All Options		
Command Line		
Manifest Tool		
XML Document Generator		
Browse Information		
Build Events		
Custom Build Step		
Code Analysis		

Output File
The /OUT option overrides the default name and location of the program that the linker creates.

OK Cancel Apply



Hello 3D CG World! (1/2)

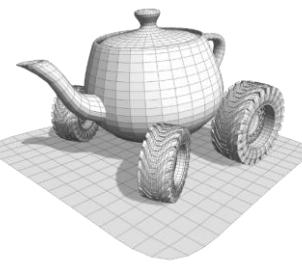
```
#include <GLFW/glfw3.h>

int main(void)
{
    GLFWwindow* window;
    /* Initialize the library */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(1000, 800, "My Race", NULL, NULL);
    if (!window){
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);
```

All the drawing commands
will be output into this
window



Hello 3D CG World! (2/2)

```
/* Loop until the user closes the window */
while (!glfwWindowShouldClose(window))
{
    /* Render here */
    glClear(GL_COLOR_BUFFER_BIT);

    /* Swap front and back buffers */
    glfwSwapBuffers(window);

    /* Poll for and process events */
    glfwPollEvents();
}

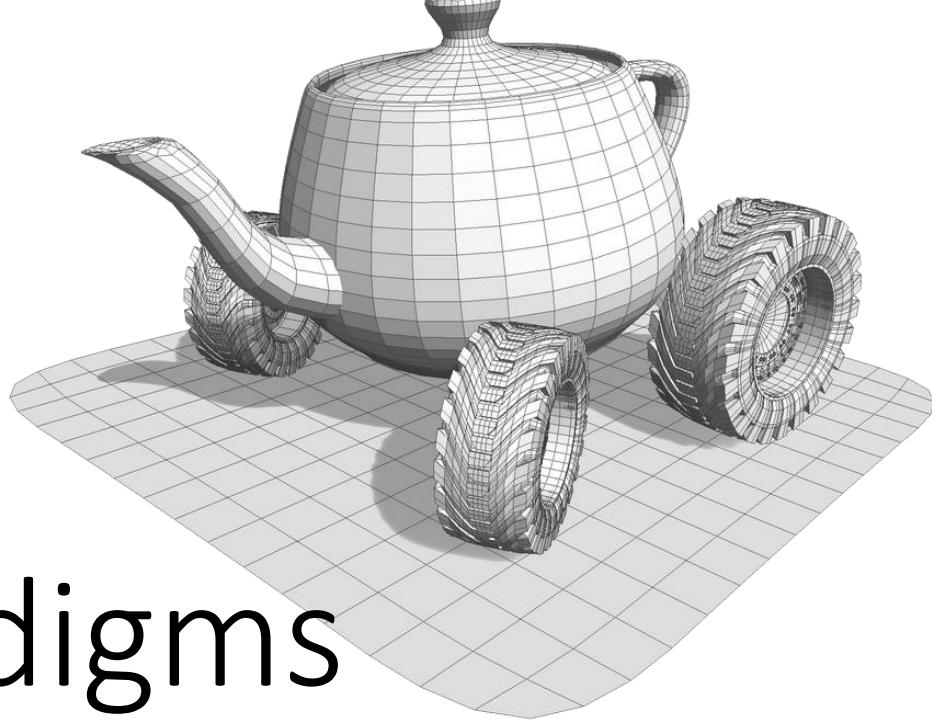
glfwTerminate();
return 0;
}
```

All the rendering of this program: clear the window

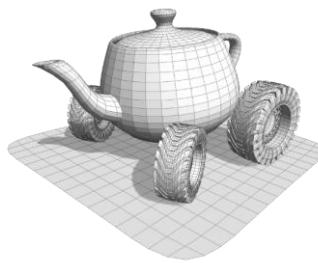
Front buffer: what you see in the window
Back buffer: where the draw commands take effect

Process the events (mouse, keyboards..).

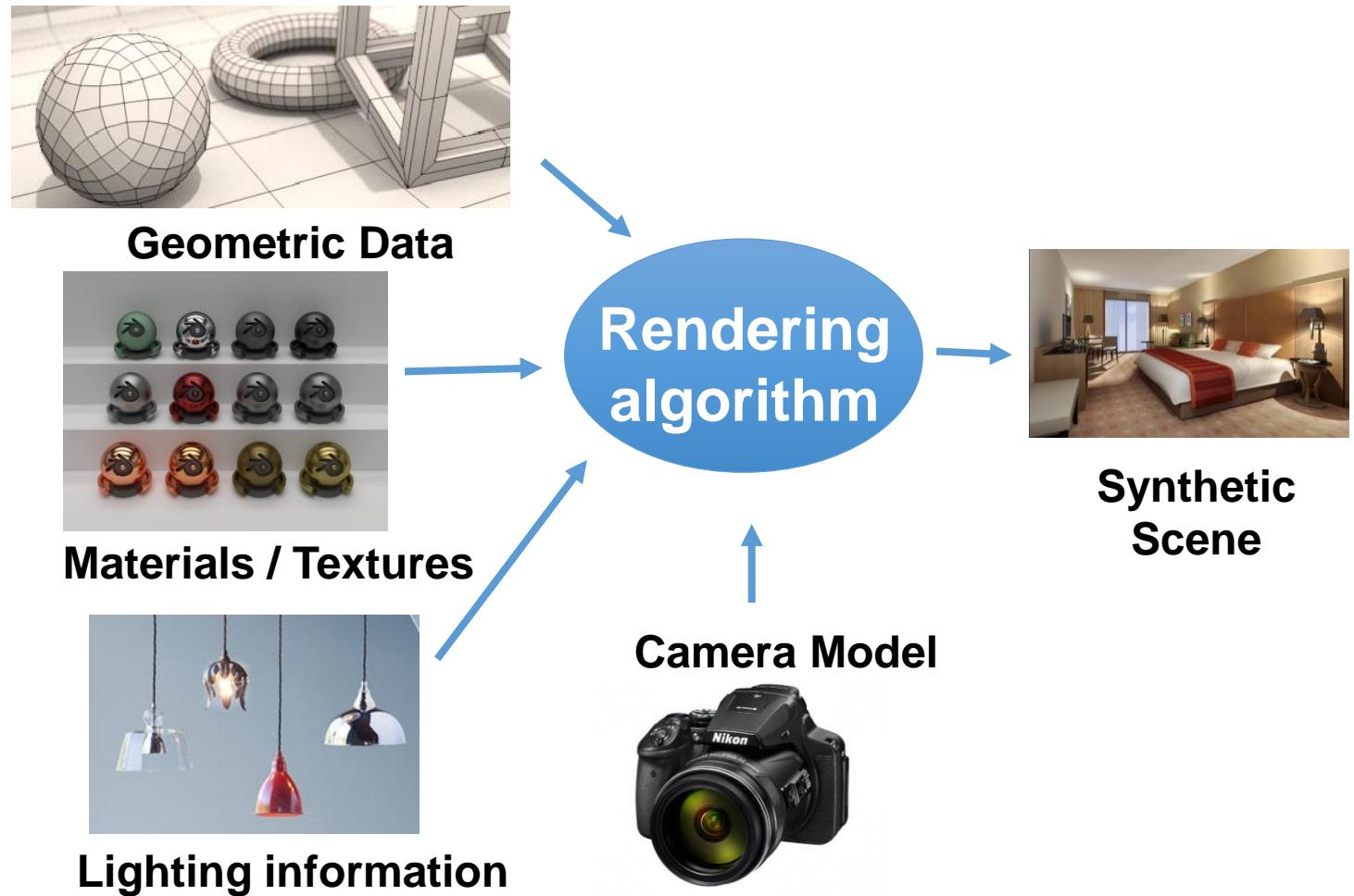
Rendering Paradigms

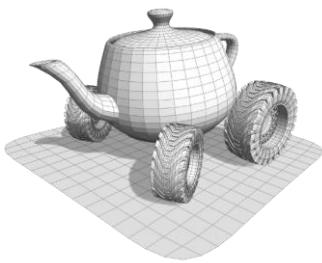


Rendering Algorithms



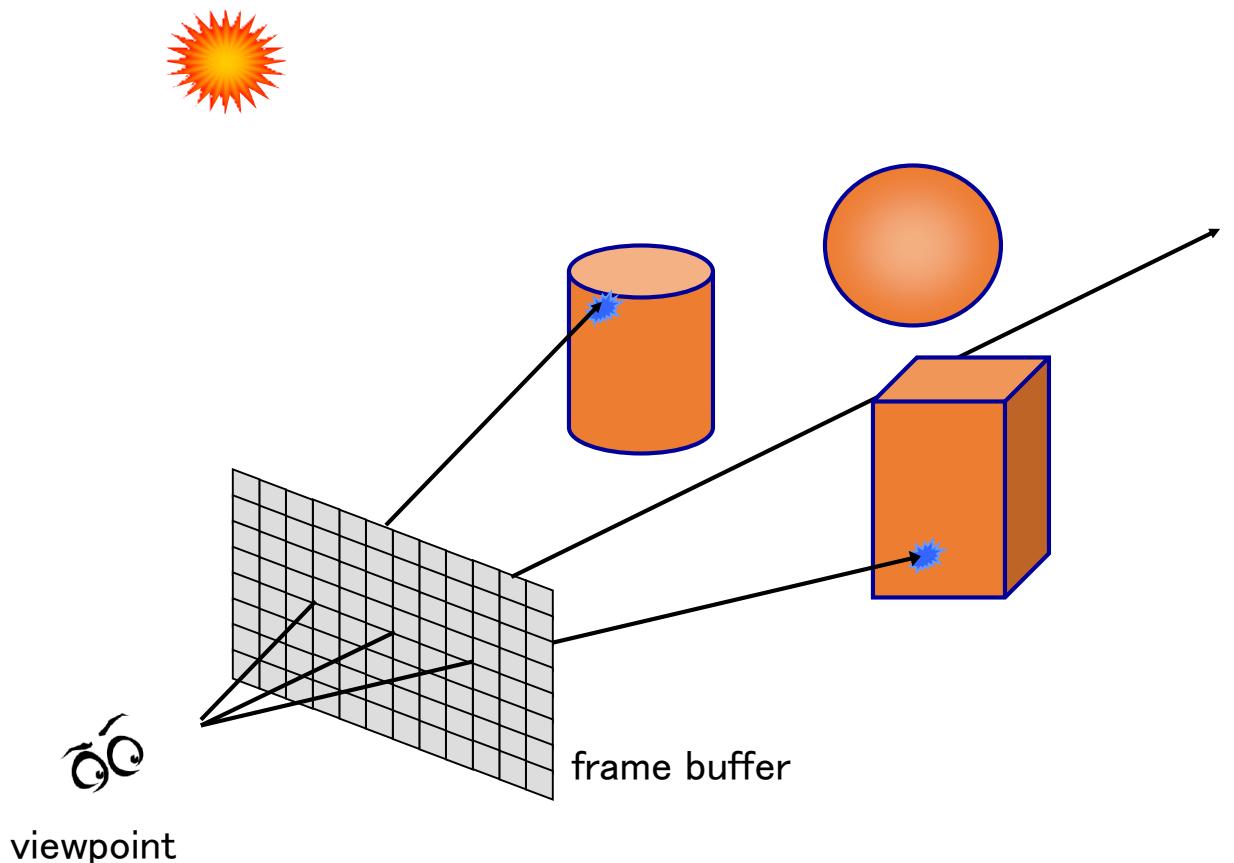
- a **Rendering Algorithm** is a series of steps that turn the digital description of a scene and of our viewing parameters in a raster image
- Two «families»
 - **Ray Tracing**
 - **Rasterization**

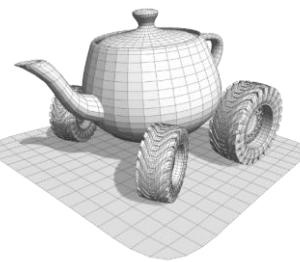




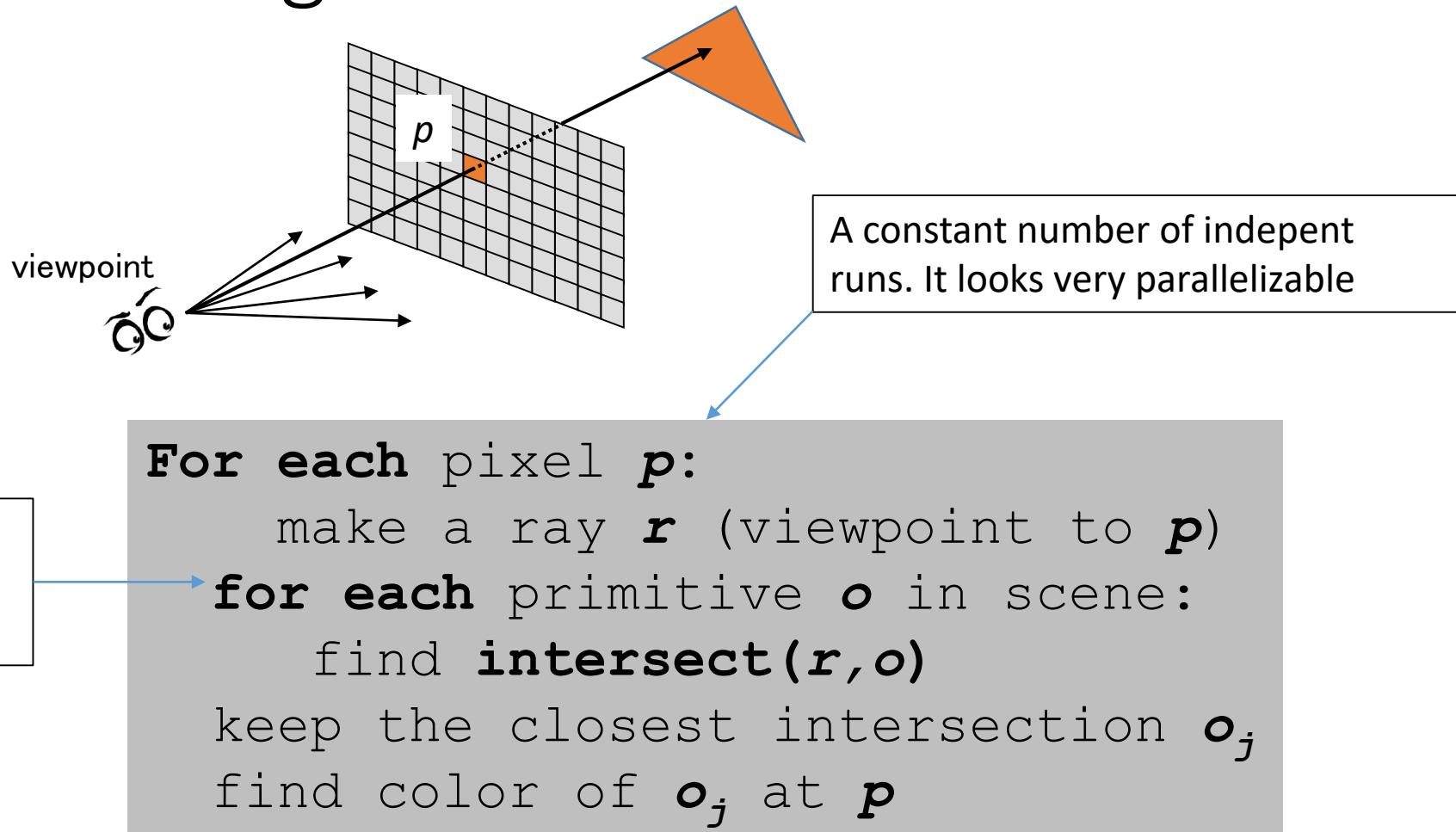
Ray Tracing

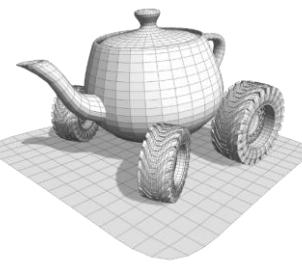
- Key idea: follow the light rays back to where they come from
- For each pixel «shoot» a ray from the eye through that pixel and check if it hits something in the scene





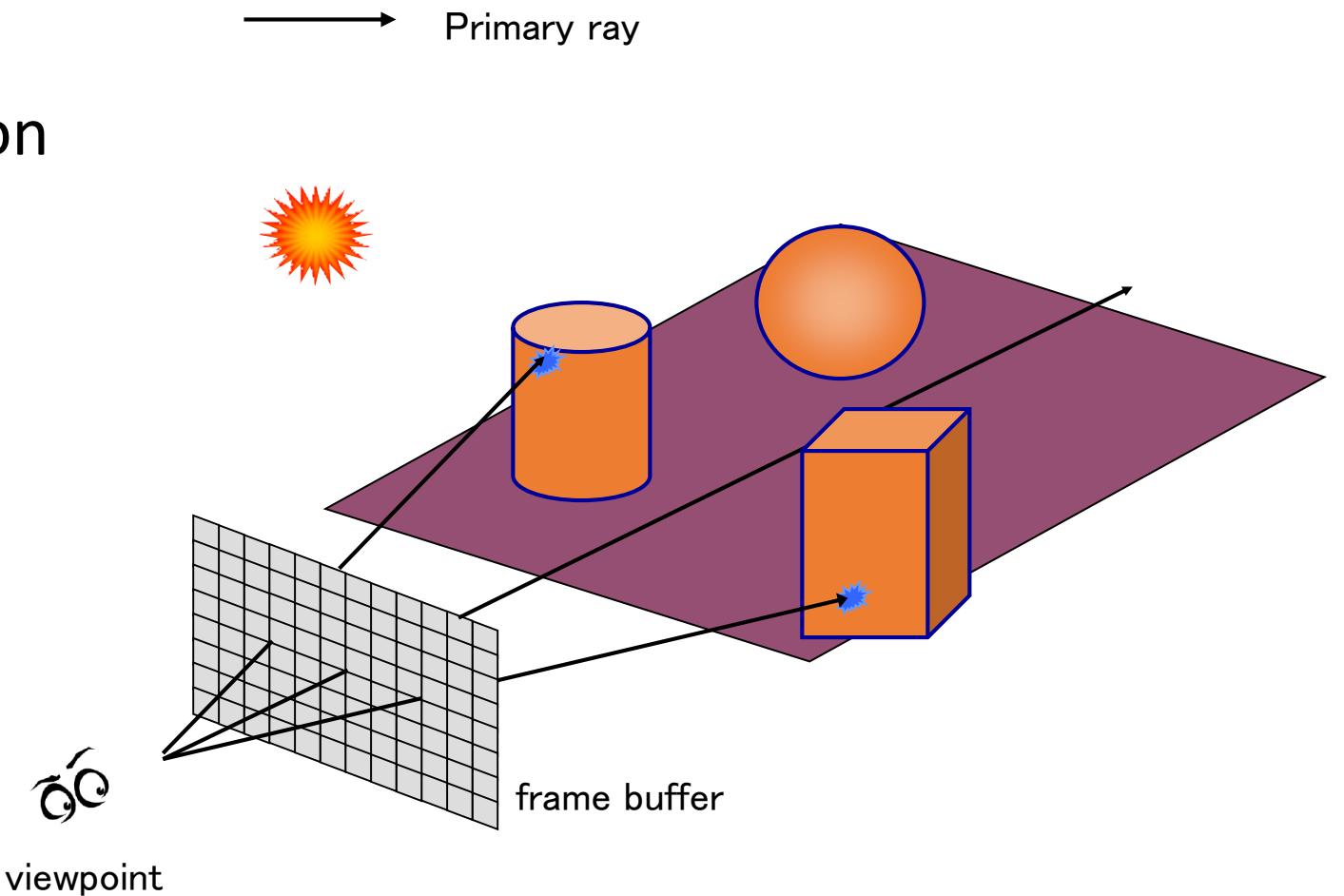
Ray Tracing Pseudocode

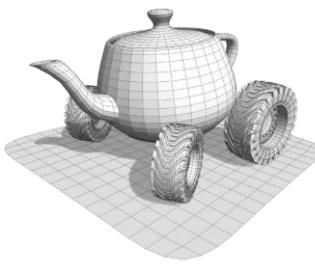




Ray-Tracing

- Implementation cost: it's all about testing the intersection with a ray with the scene

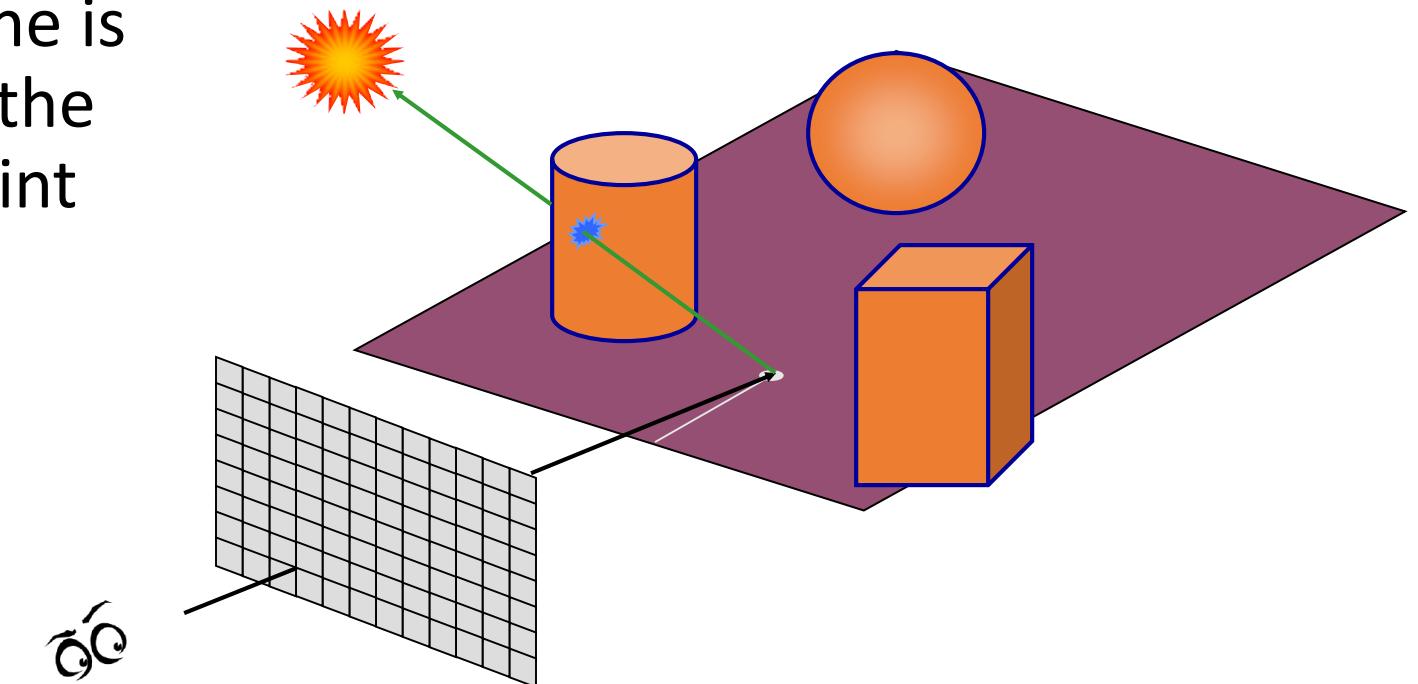


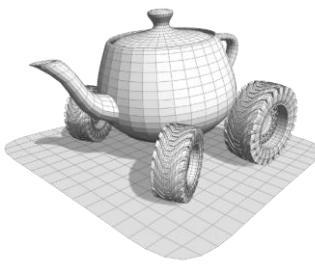


Ray-Tracing

- Easy to handle: shadows (sharp ones)
- If the intersection ray-scene is found, trace another ray (the **shadow ray**) from that point to the position of the lightsources.
If it intersects the scene than is in shadow

→ Shadow ray (if it intersect something it is in shadow)
→ Primary ray

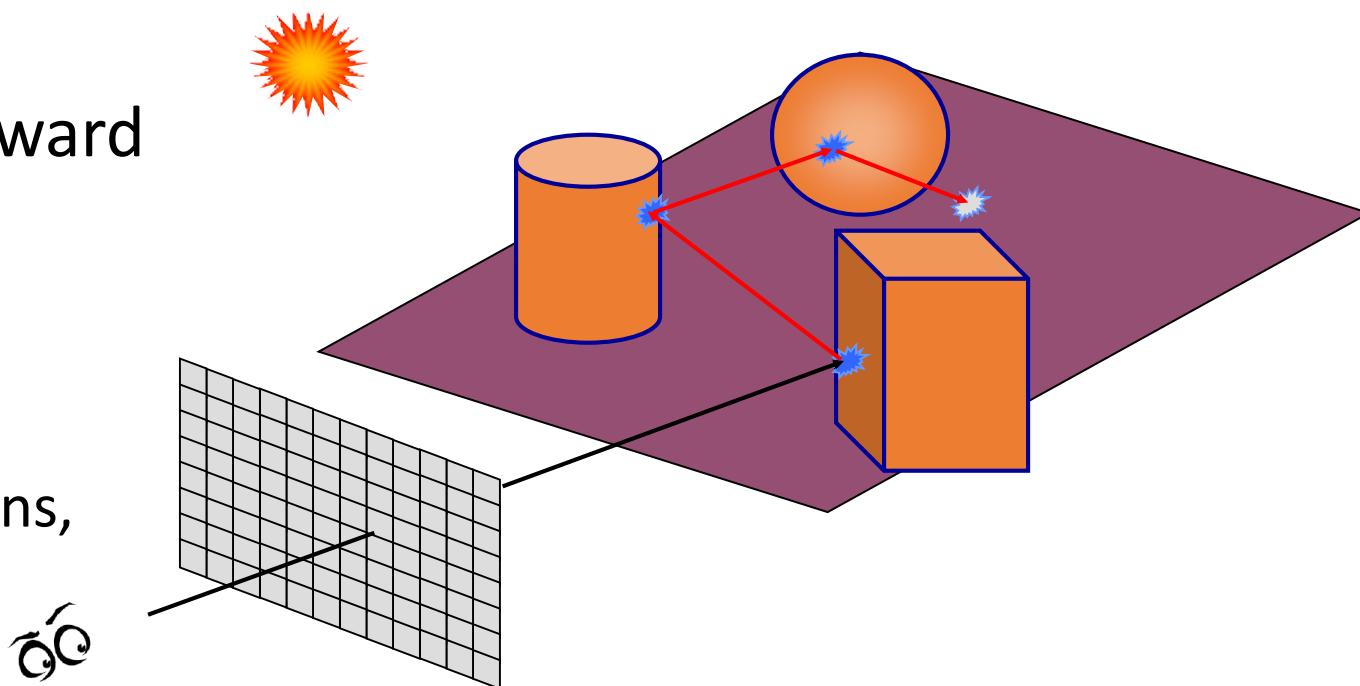


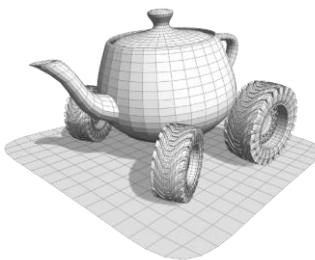


Ray-Tracing

- Easy to handle: multiple specular reflections
- If the intersection ray-scene is found, trace another ray (the **reflection or secondary ray**) toward the specular direction of the incoming ray.
 - Repeat N times
 - N == number of **bounces**
 - Higher N, more realistic reflections, longer run time

→ Reflection ray
→ Shadow ray (if it intersect something it is in shadow)
→ Primary ray

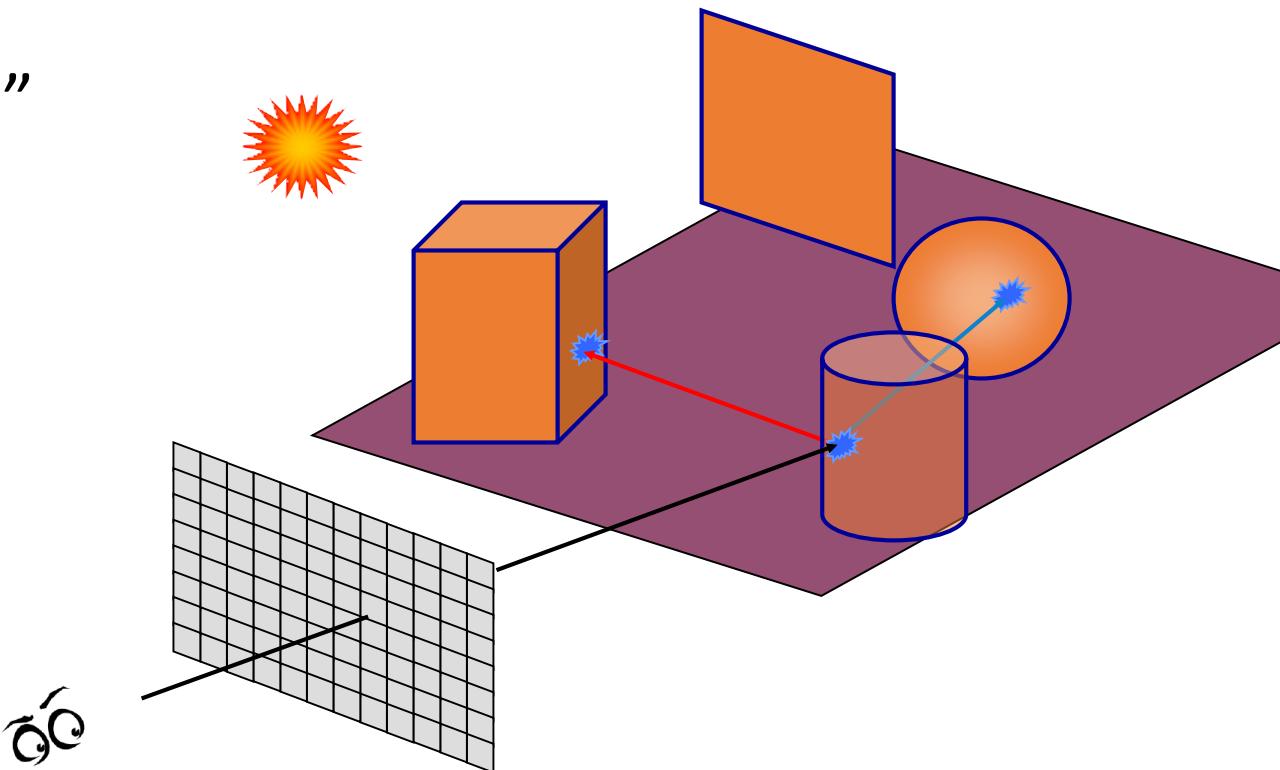


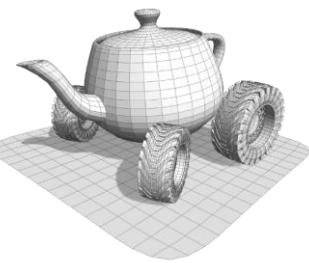


Ray-Tracing

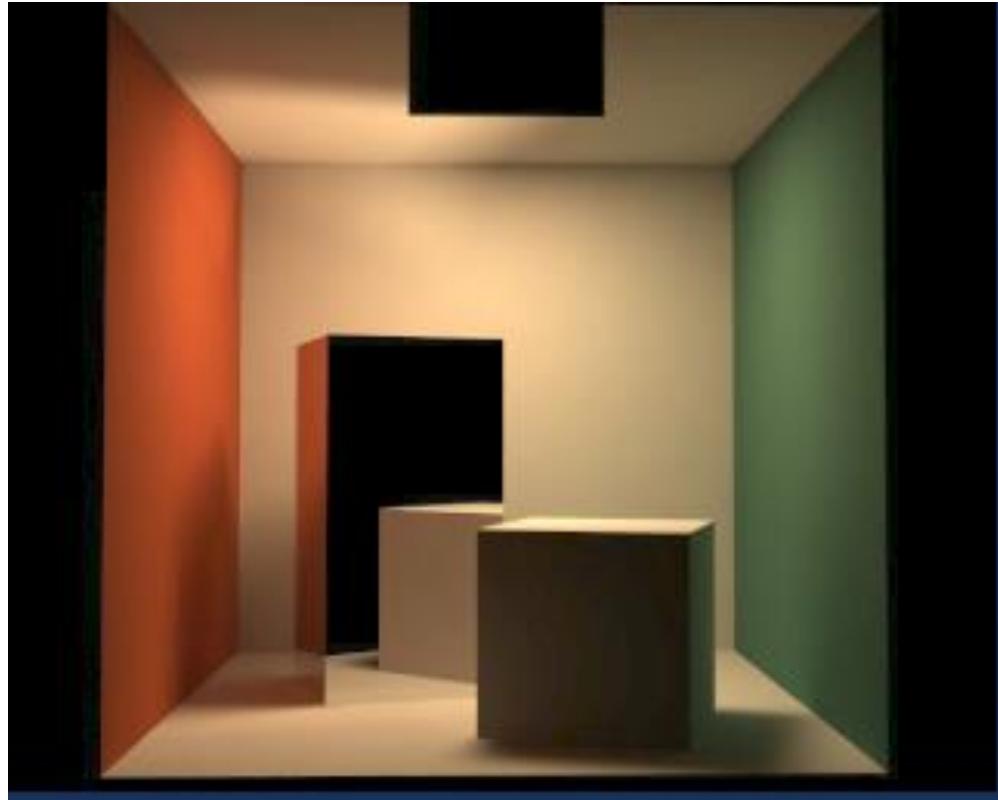
- Easy to handle: refraction
- Exactly like reflection with the bouncing direction going “through” the object
- note: we will study reflections, refractions, shadows, caustics and other mirabilia later on...

→ Refraction ray
→ Reflection ray
→ Shadow ray (if it intersect something it is in shadow)
→ Primary ray

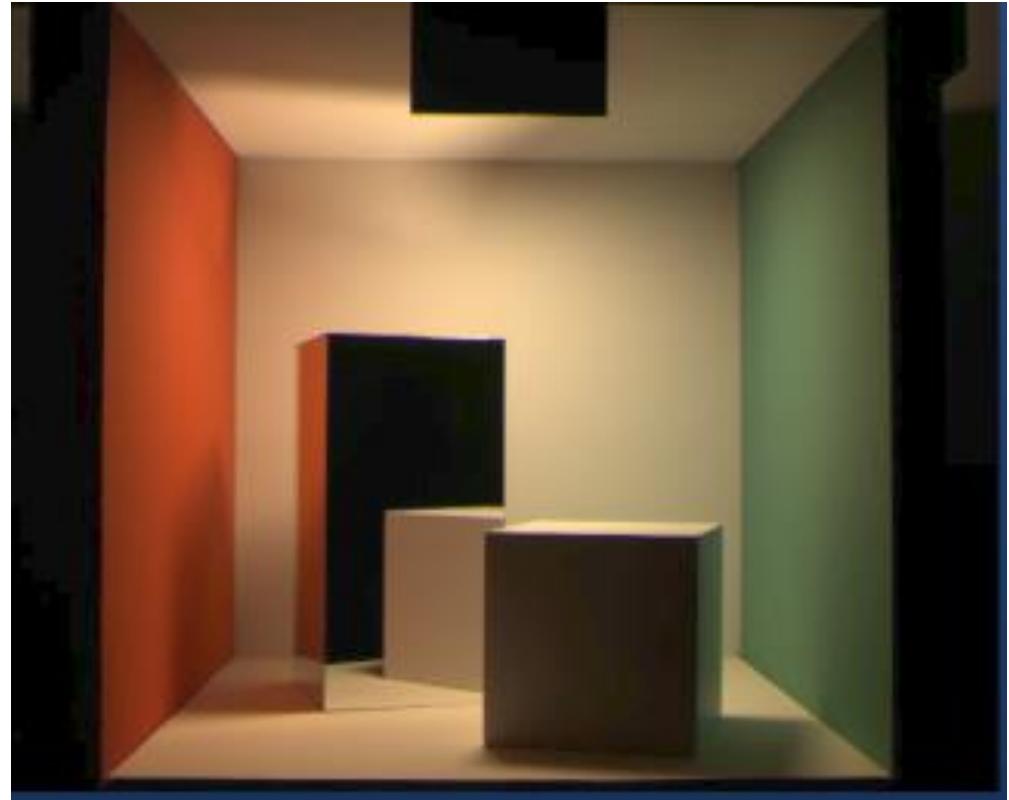




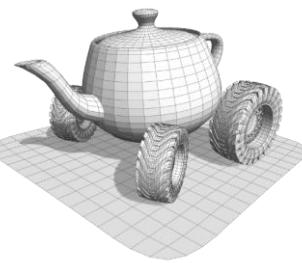
Ray-Tracing: the Cornell Box



Real photograph with
controlled lighting conditions



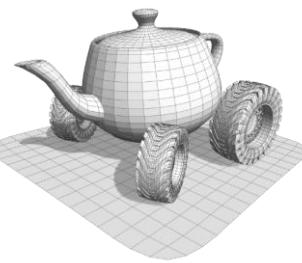
Ray traced rendered model



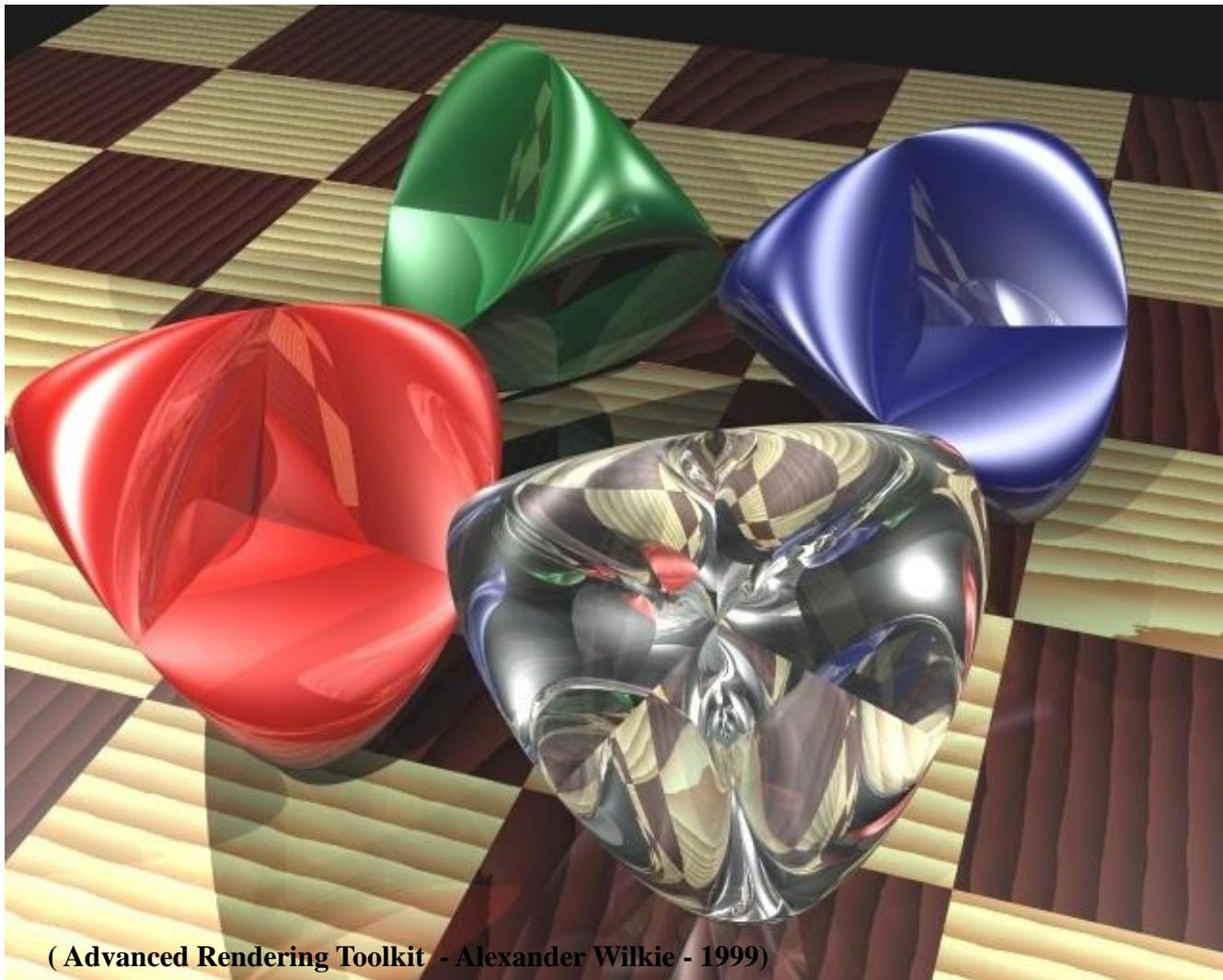
Ray-Tracing: examples



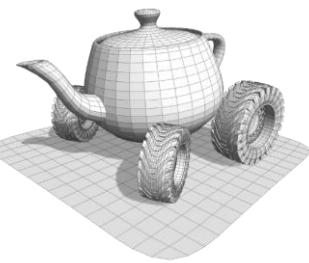
(Advanced Rendering Toolkit - Alexander Wilkie - 1999)



Ray-Tracing: examples

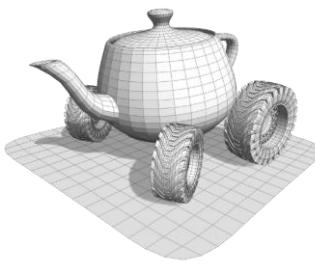


(Advanced Rendering Toolkit - Alexander Wilkie - 1999)



Ray-Tracing: examples





Ray-Tracing : cost

- Main core:
 - Computing the intersection between a 3D RAY e 3D PRIMITIVES
- Computationally complex
 - High cost
 - Yet SUBLINEAR! (w.r.t. to the primitive count)
 - If you are using the right structures: spatial indexing!
- Once used only for off-line
 - Now realtime exploiting GPUs

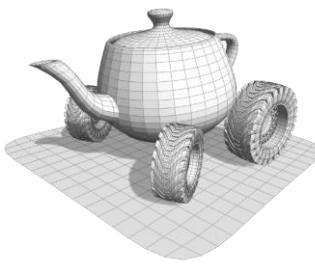
$$RTCost(r) = N \sum_{\forall o \in S} Int(r, o)$$

Cost for a ray

Number of bounces

Const of intersecting ray r with object o

Ray-Tracing : primitives

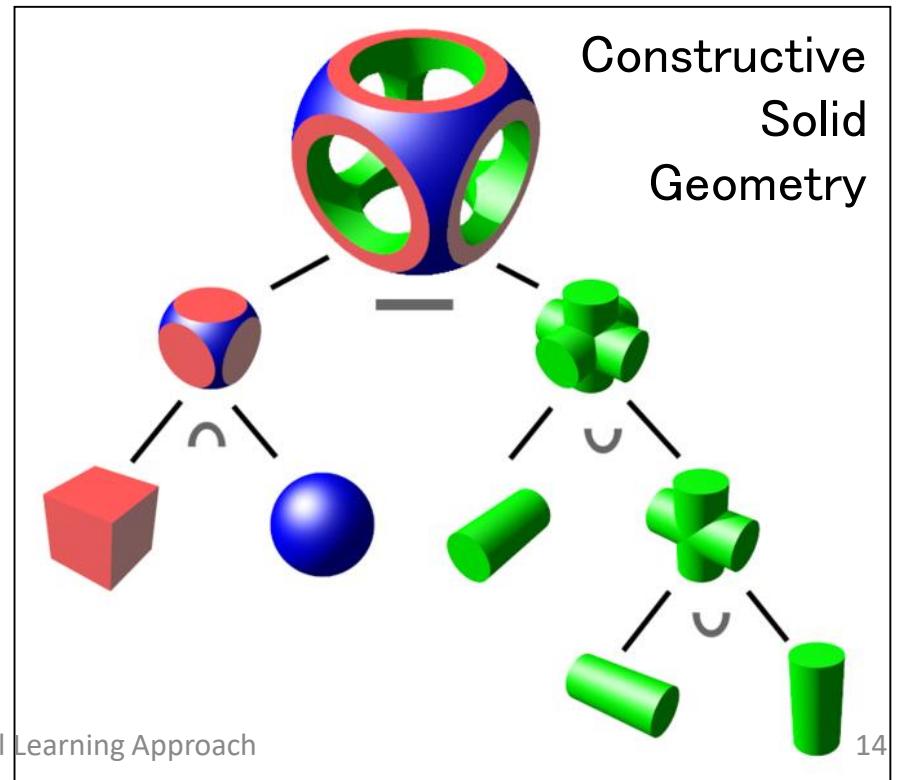


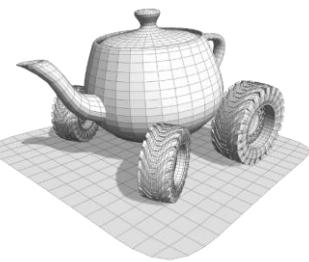
Q: which primitives ?

A: everything that I can intersect with a ray !

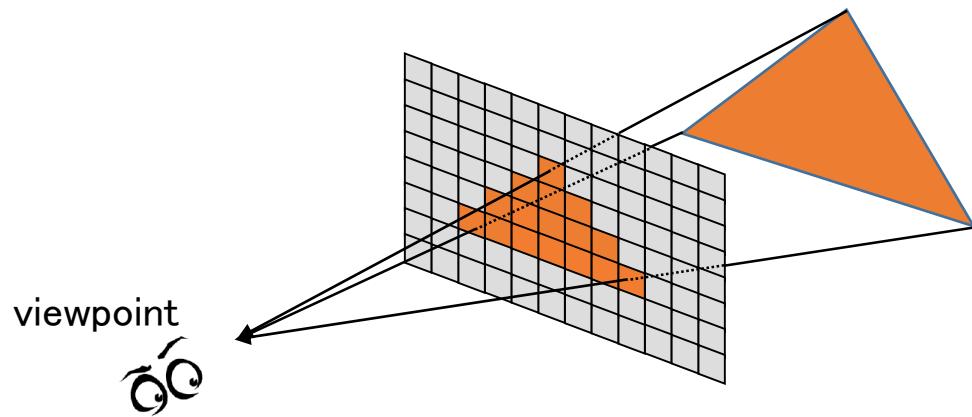
Examples:

- Triangles, quadrilateral etc..
 - Implicit surfaces
 - sphere
 - Constructive Solid Geometry
 - ...



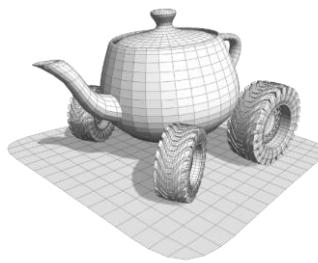


Rasterization-based Pseudocode



```
For each primitive  $t$ :  
    find where  $t$  falls on screen  
    rasterize the 2D shape  
    for each produced pixel  $p$  :  
        find the color for  $t$   
        color  $p$  with it
```

aka T&L: Transform & Lighting...

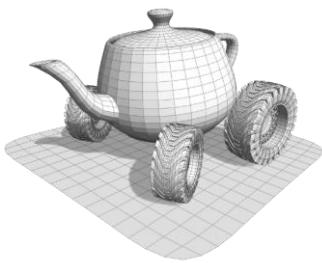


- ***Transform*** :

- Coordinate system transformations
- Goal: project the primitives in screen space
- ... but also to assemble the primitives to compose the scene

- ***Lighting*** :

- Lighting computation
- Goal: compute the final color of each part of the scene
 - To do so we have to take into account:
 - Optical characteristics
 - Lighting of the scene
 - Geometry of the objects



Rasterization based : rendering primitives

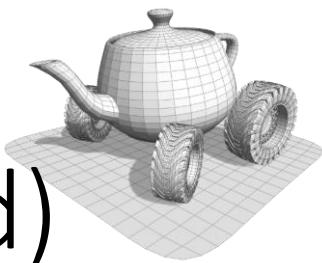
Q: which **rendering primitives** for rasterization-based algorithms ?

A: *everything that we know:*

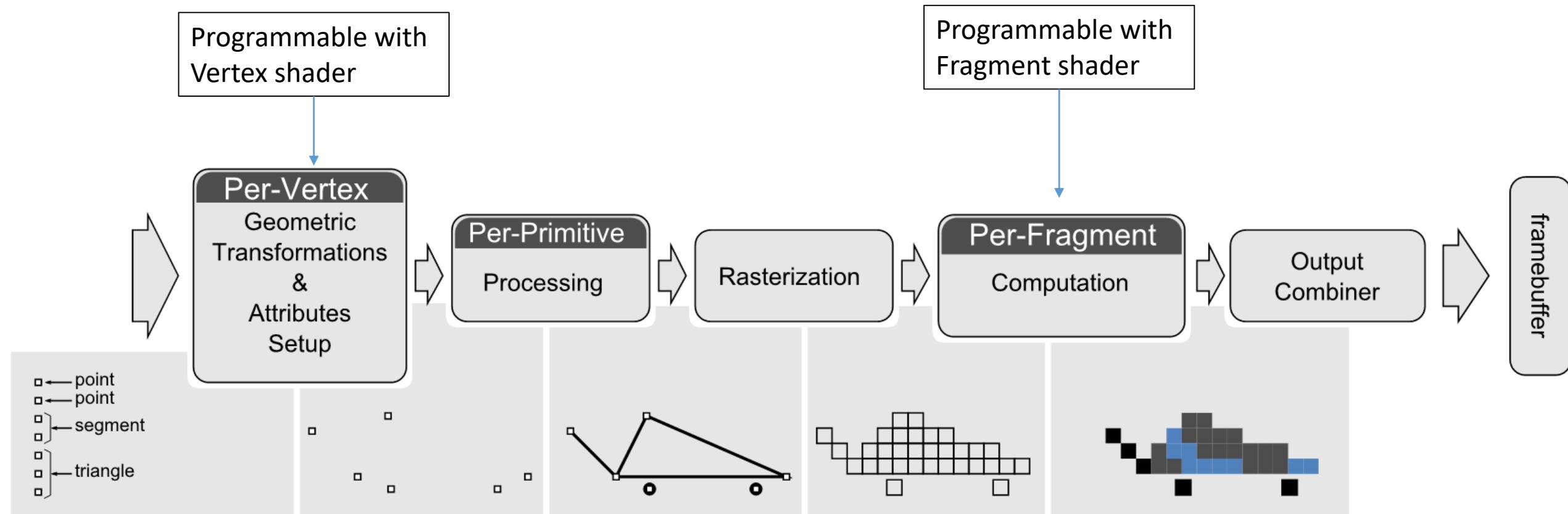
- 1) how to project from 3D to 2D
- 2) «rasterize» in 2D (= convert into pixels)

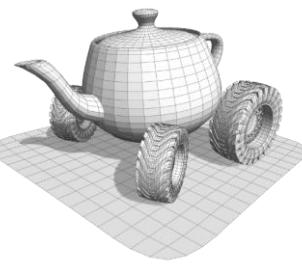
Commonly:

- *SEGMENTS*
- *POINTS*
- *TRIANGLES*

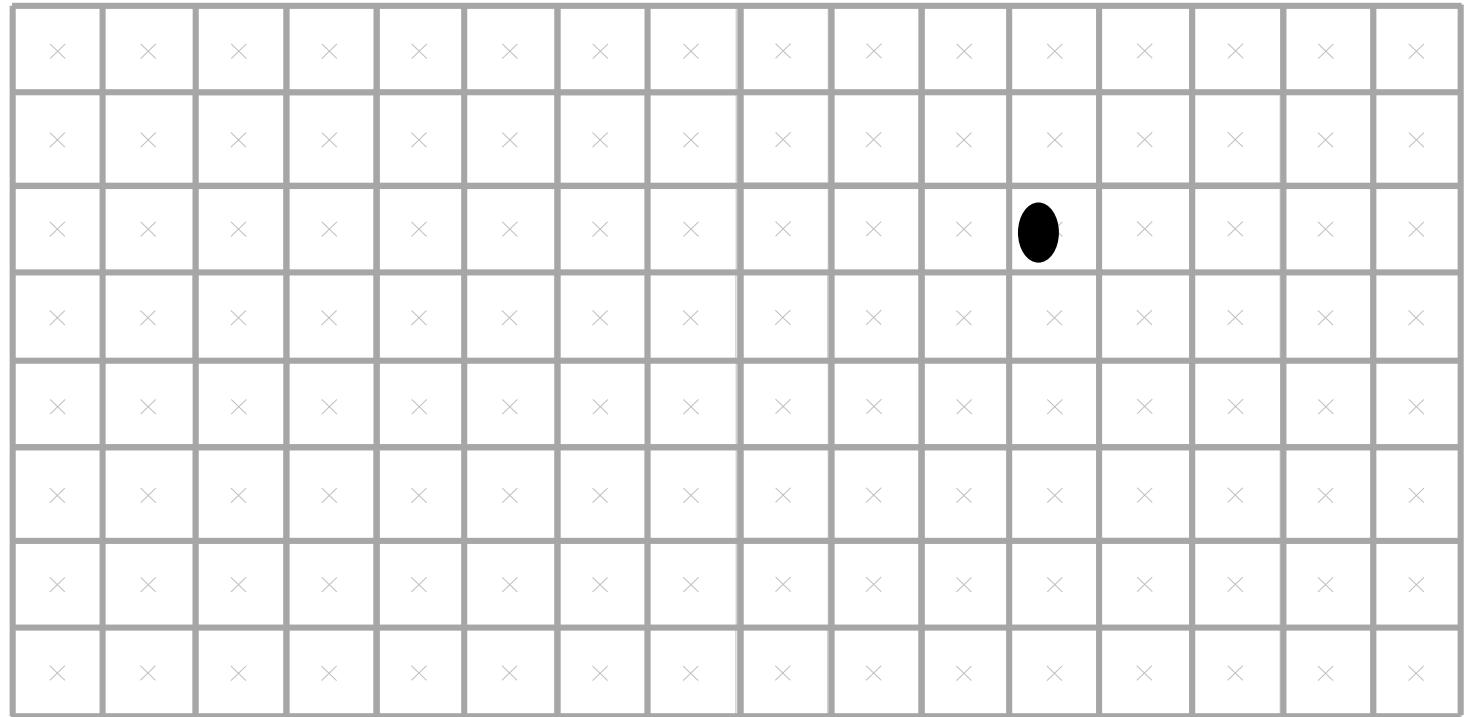


Rasterization-based pipeline (oversimplified)

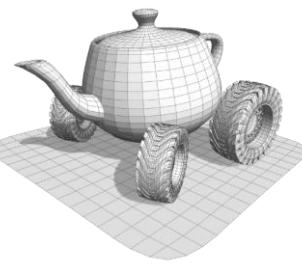




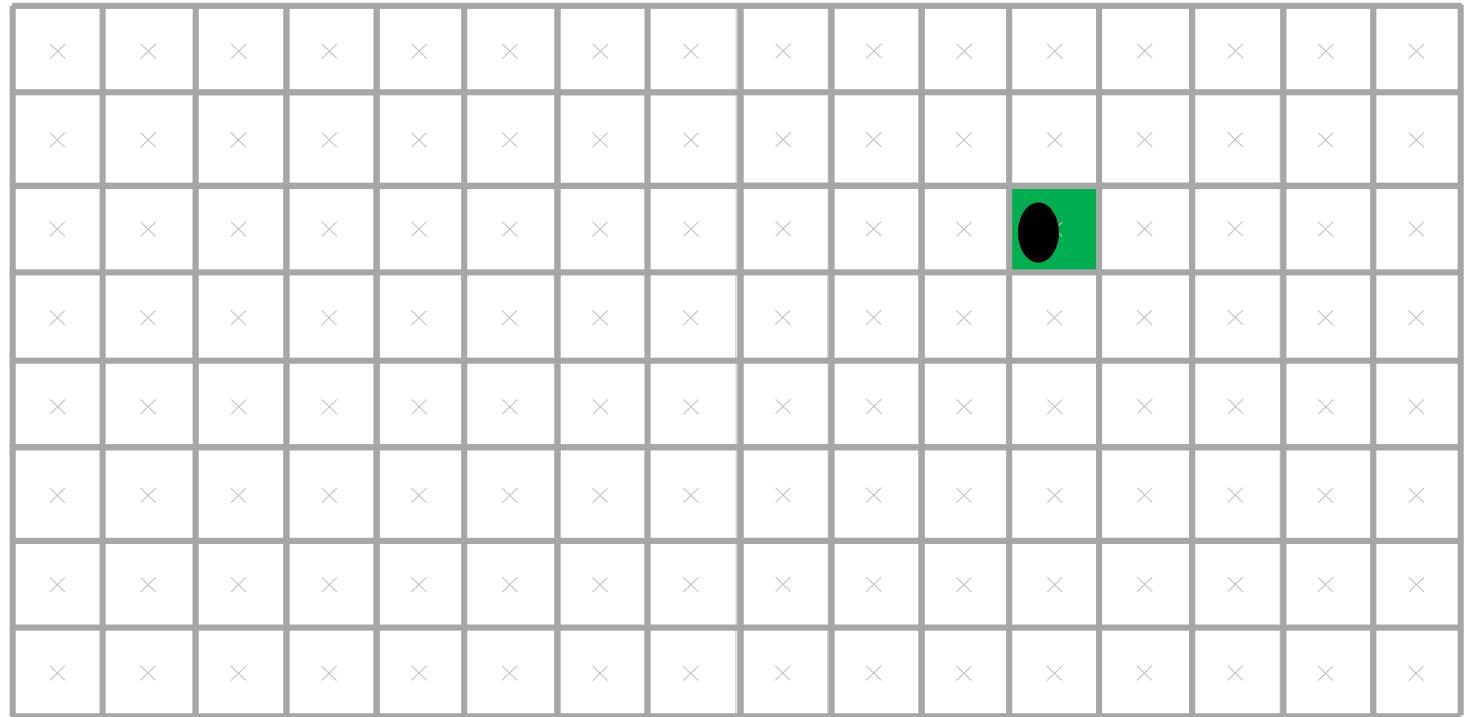
Rasterizing points



Point “splat”



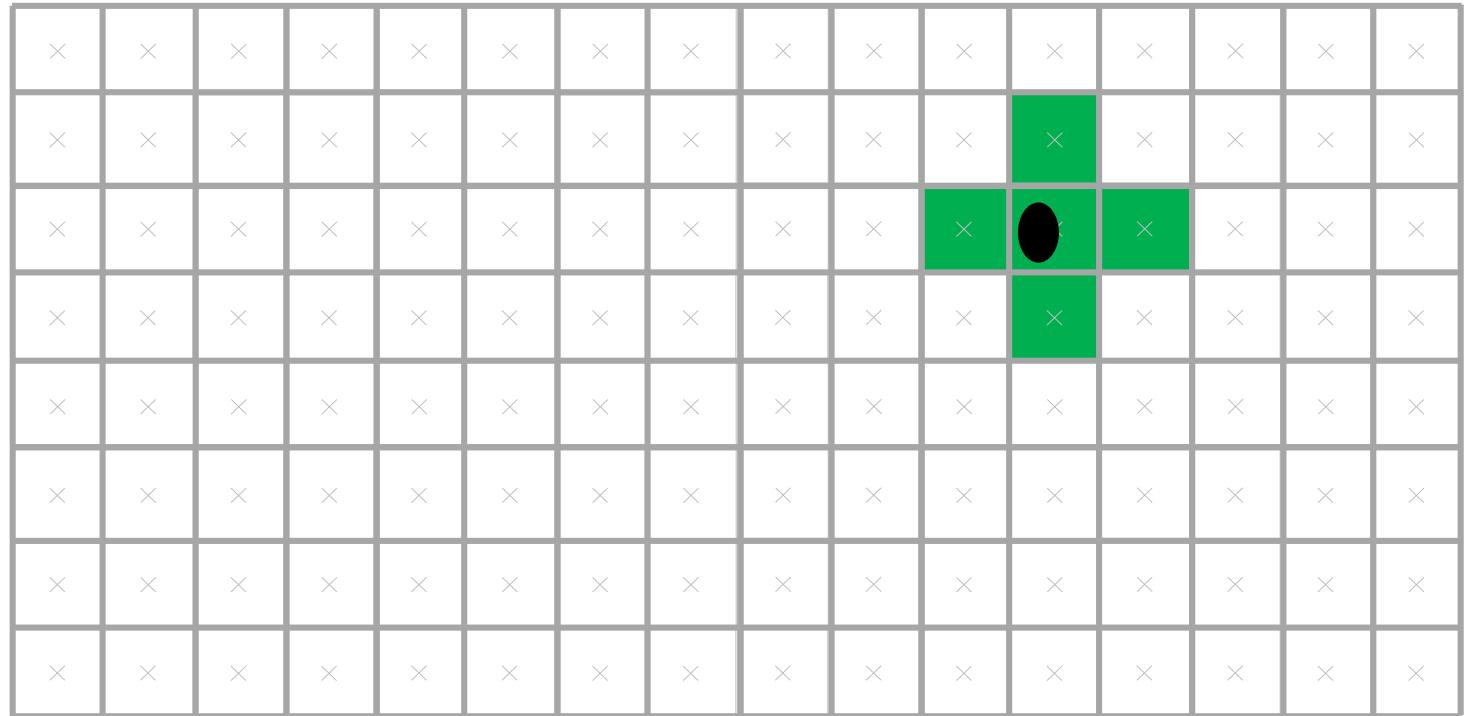
Rasterizing points



Point “splat”



Rasterizing points



Point “splat”

Discrete Differences Analyzer (DDA)

Rasterizing lines

- Line equation:

- Slope $\rightarrow m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$
- $y = y_0 + m(x - x_0)$

```
RasterizeH(x0,y0,x1,y1){
    m = (y1-y0) / (x1-x0);
    x=x0;y=y0;
    do{
        pixel(x,y) = on
        x+=1;
        y=y+m;
    } while(x<x1);
}
```

Assumes:
 $x_0 \leq x_1$
 $-1 \leq m \leq 1$

$$m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

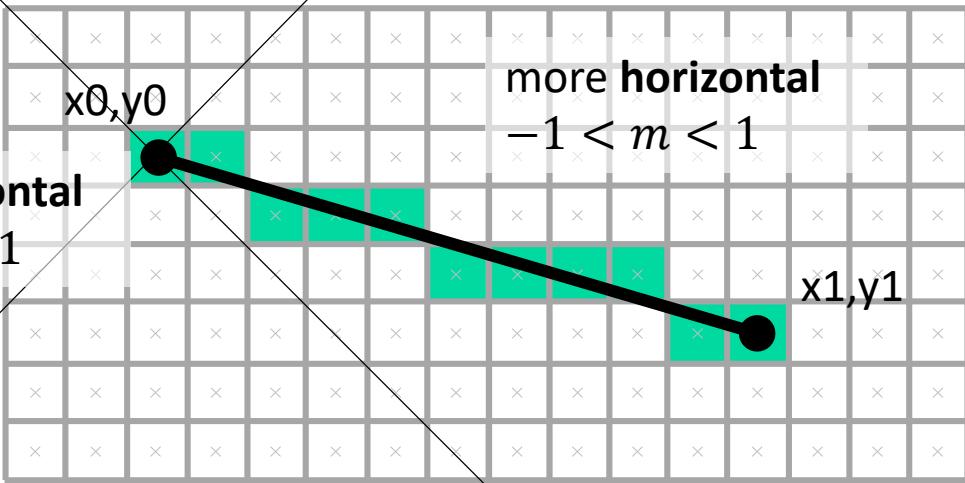
$$-1 < m < 1$$

more vertical

$$-1 < \frac{1}{m} < 1$$

more vertical

$$-1 < \frac{1}{m} < 1$$



more horizontal

$$-1 < m < 1$$

x_0, y_0

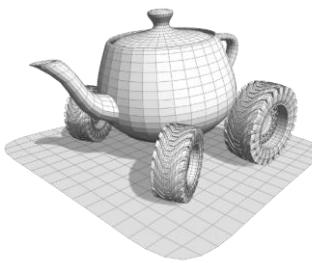
x_1, y_1

$$y = y_0 + m(x - x_0)$$

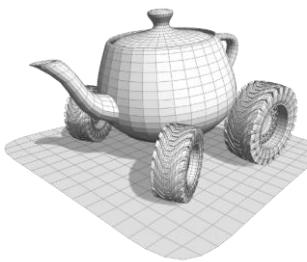
yes

no

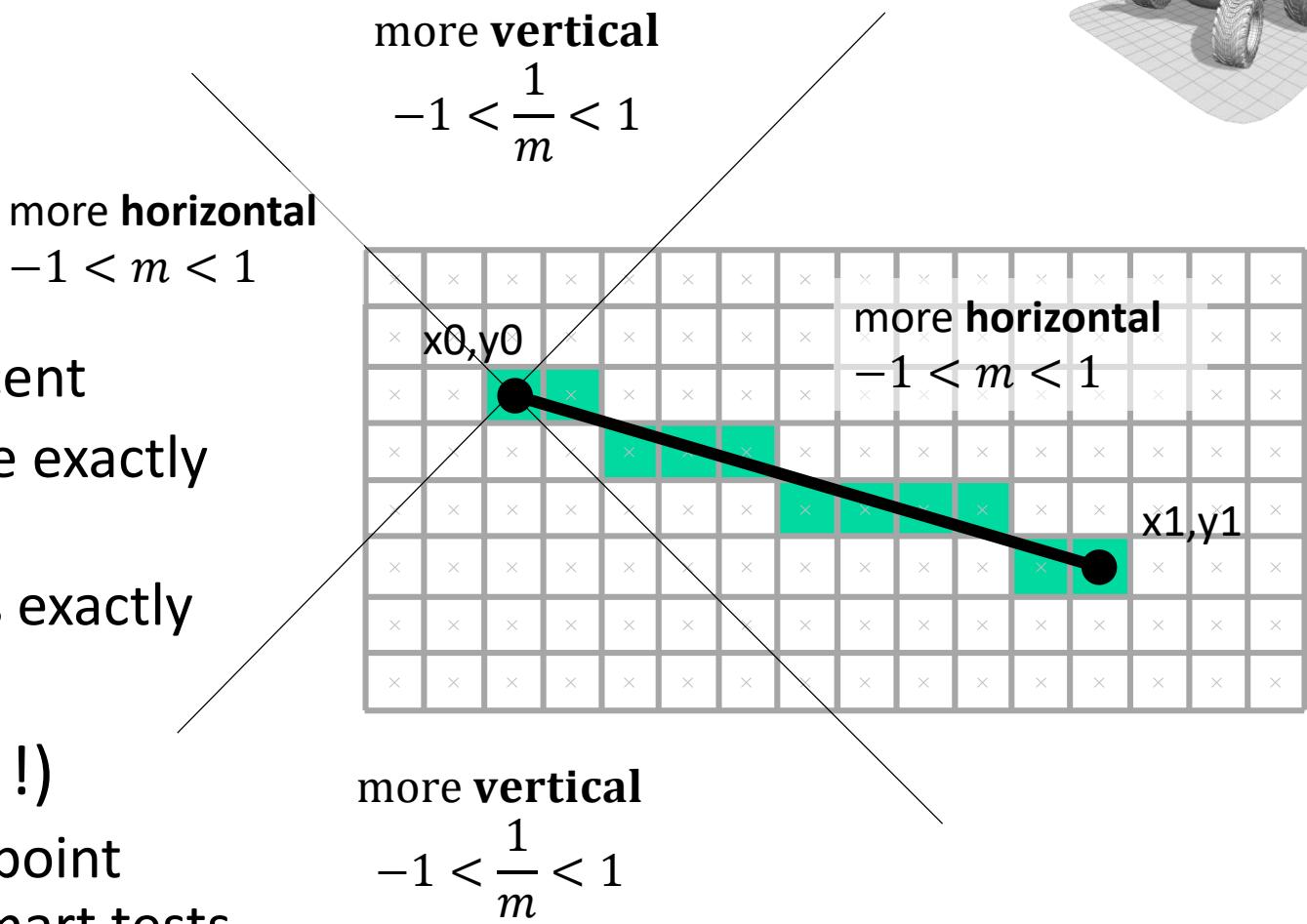
$$x = x_0 + \frac{1}{m} (y - y_0)$$

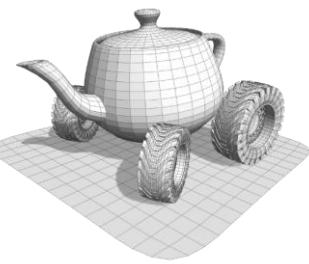


Rasterizing lines

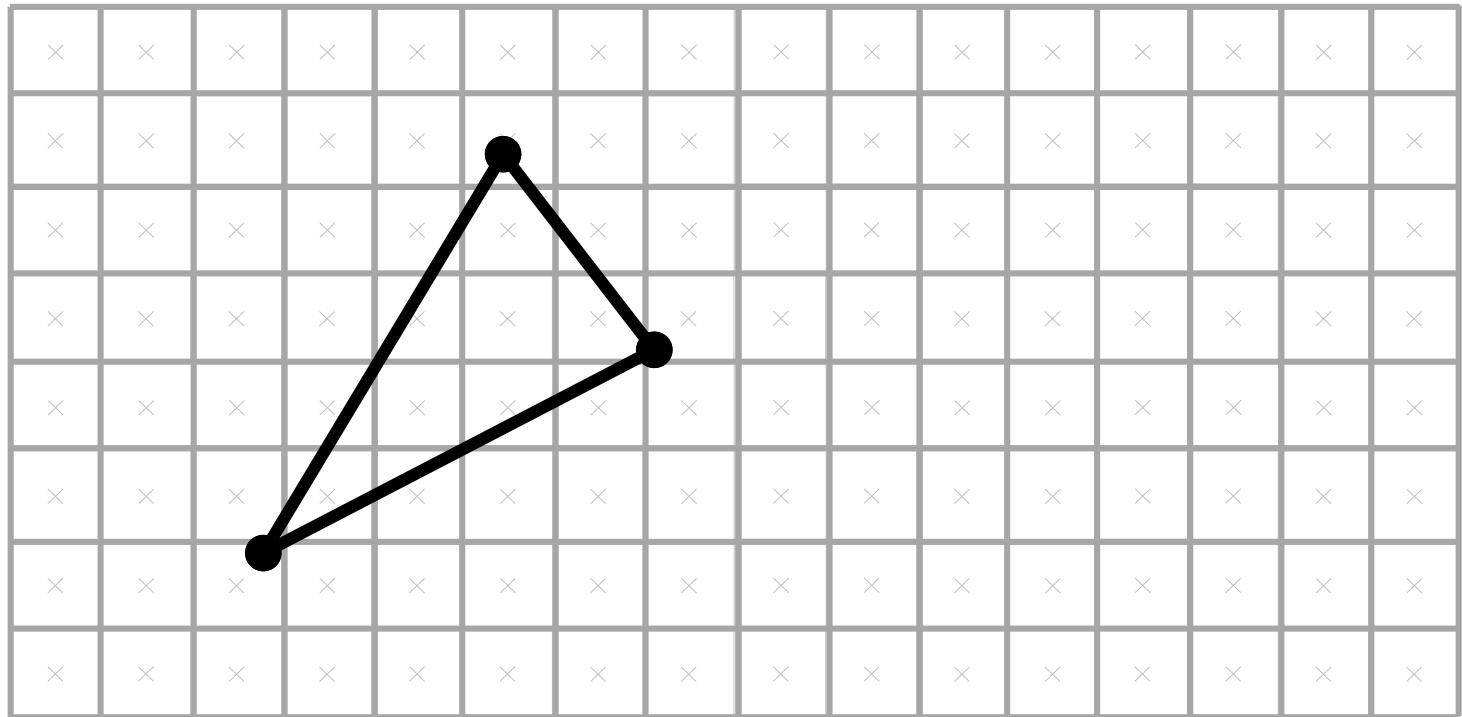


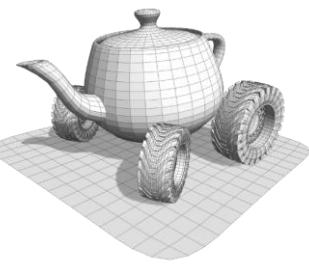
- Guarantees:
 - All consecutive pixels are adjacent
 - For *more horizontal* lines, there exactly **one** pixel per column
 - For *more vertical* lines, there is exactly **one** pixel per row
- Bresenhamn algorithm (1962 !)
 - Perform DDA without floating point operations, only integer and smart tests
 - Strictly sequential, not used in modern GPUs



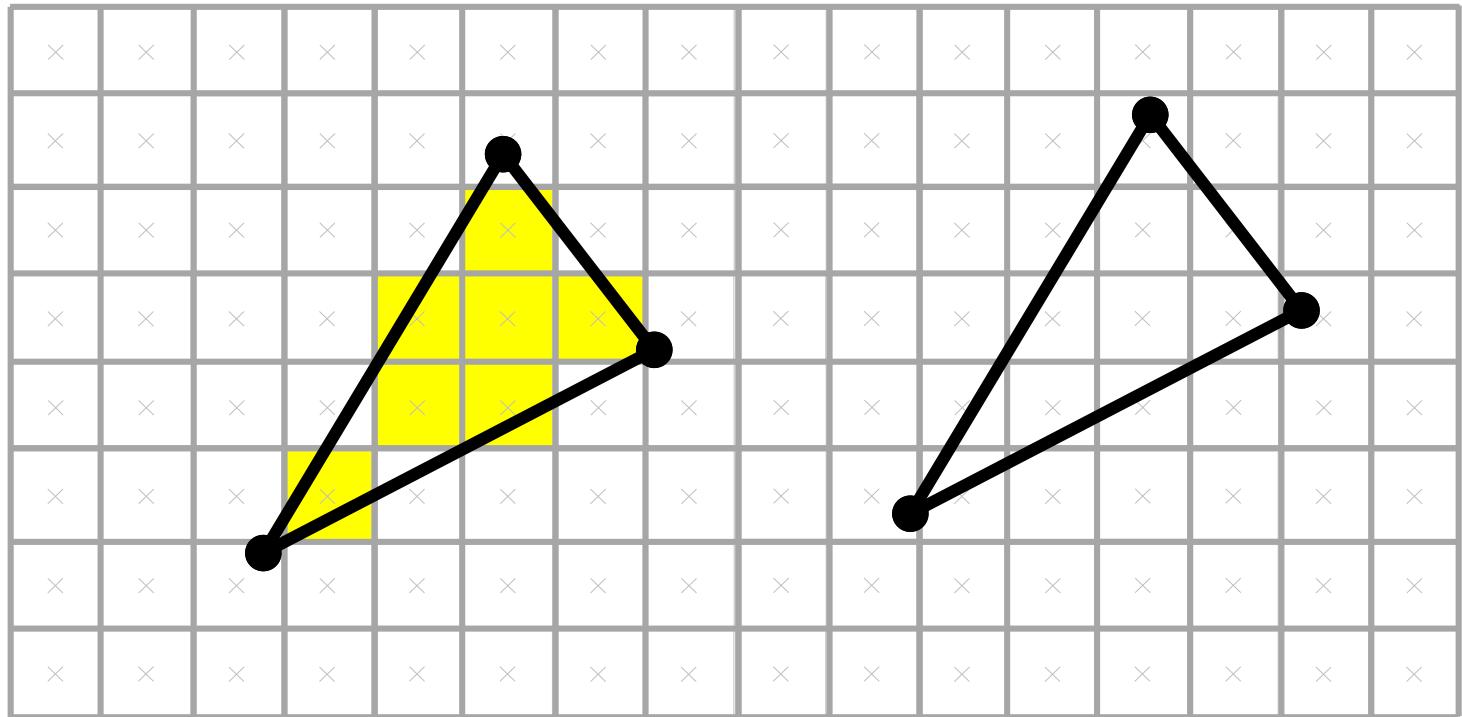


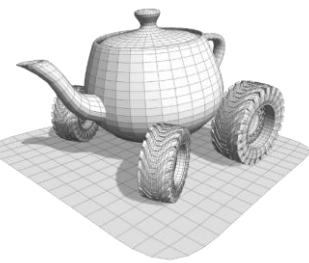
Rasterizing Triangles



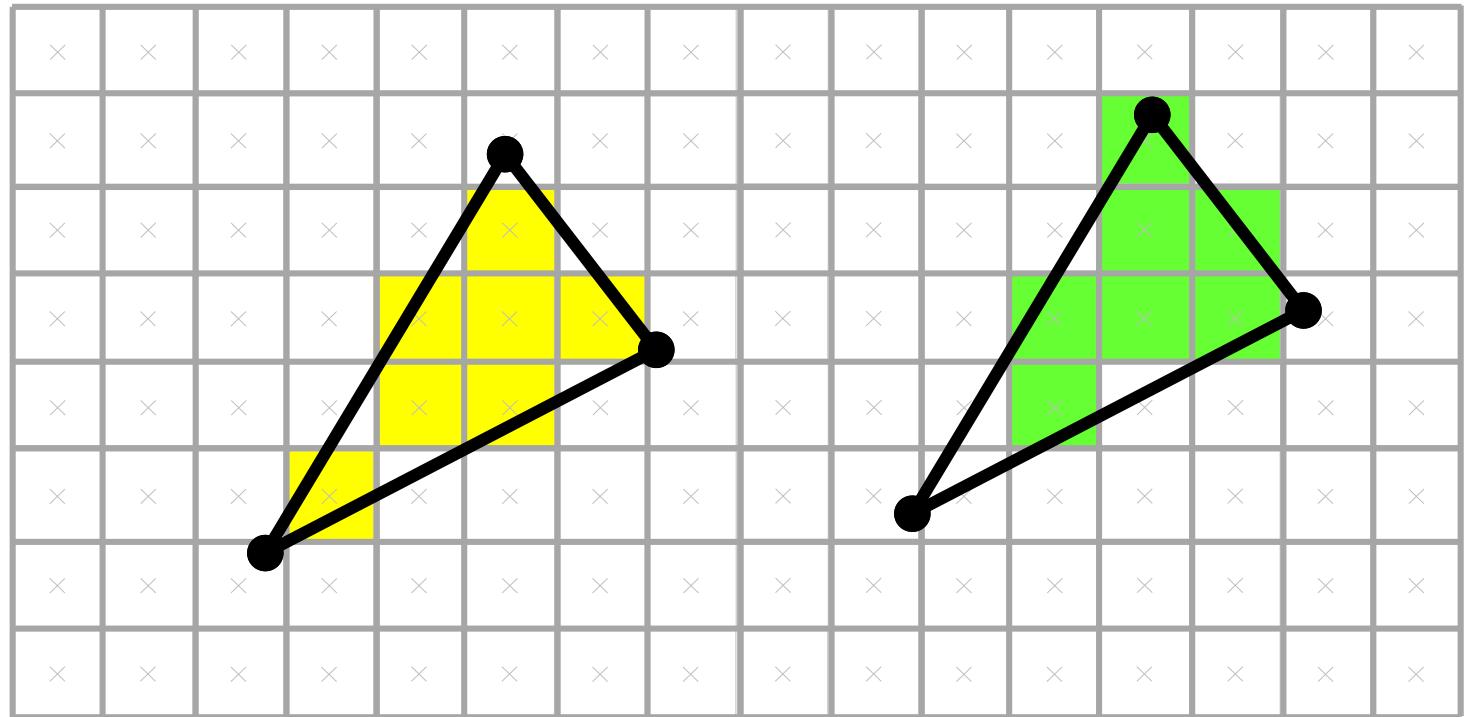


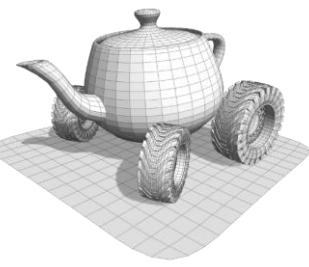
Rasterizing Triangles



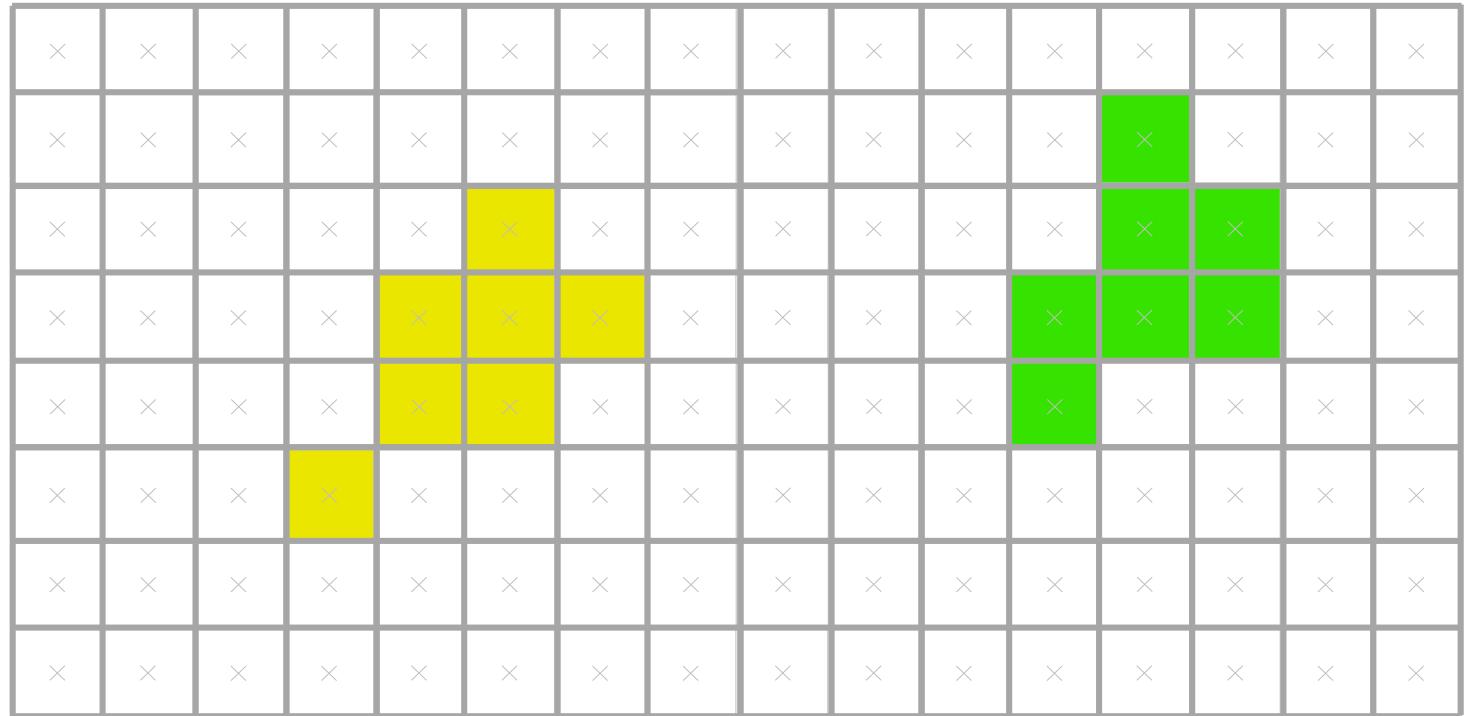


Rasterizing Triangles



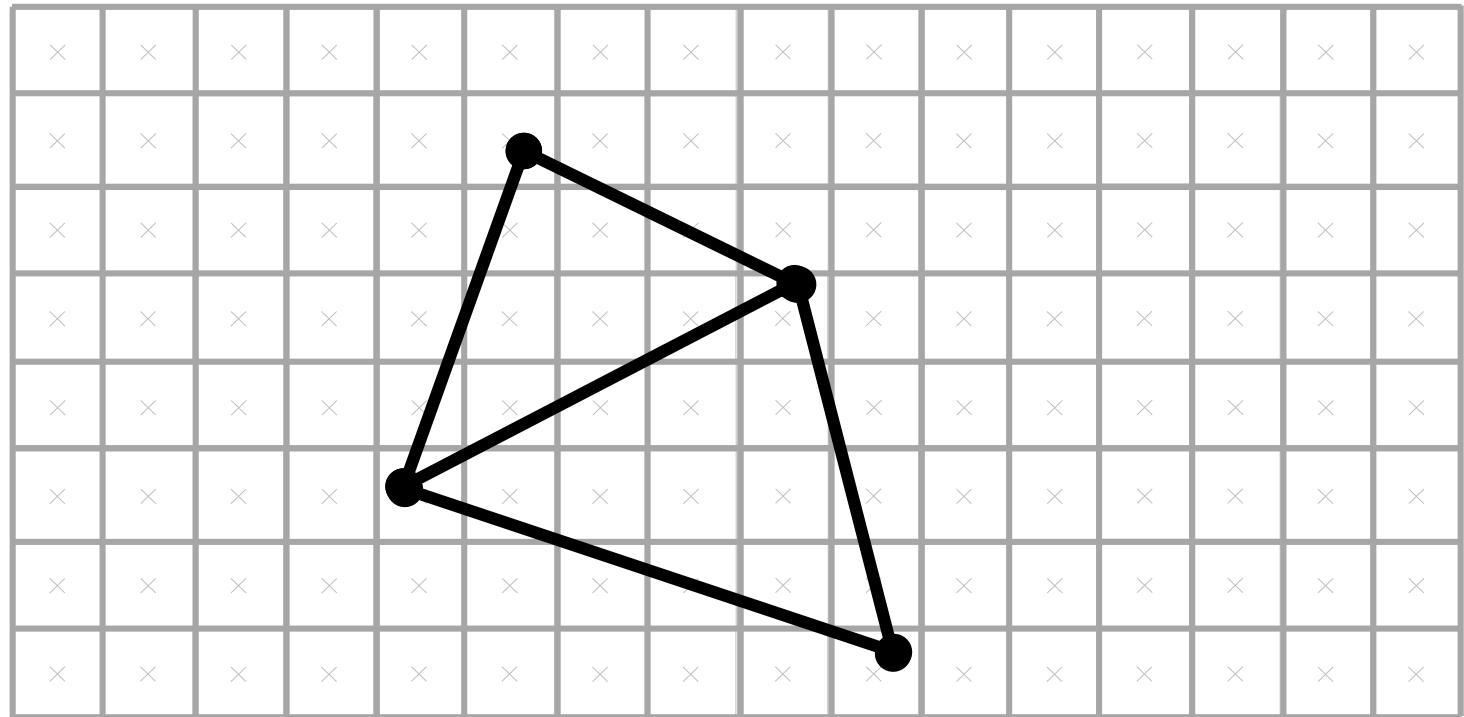


Rasterizing Triangles



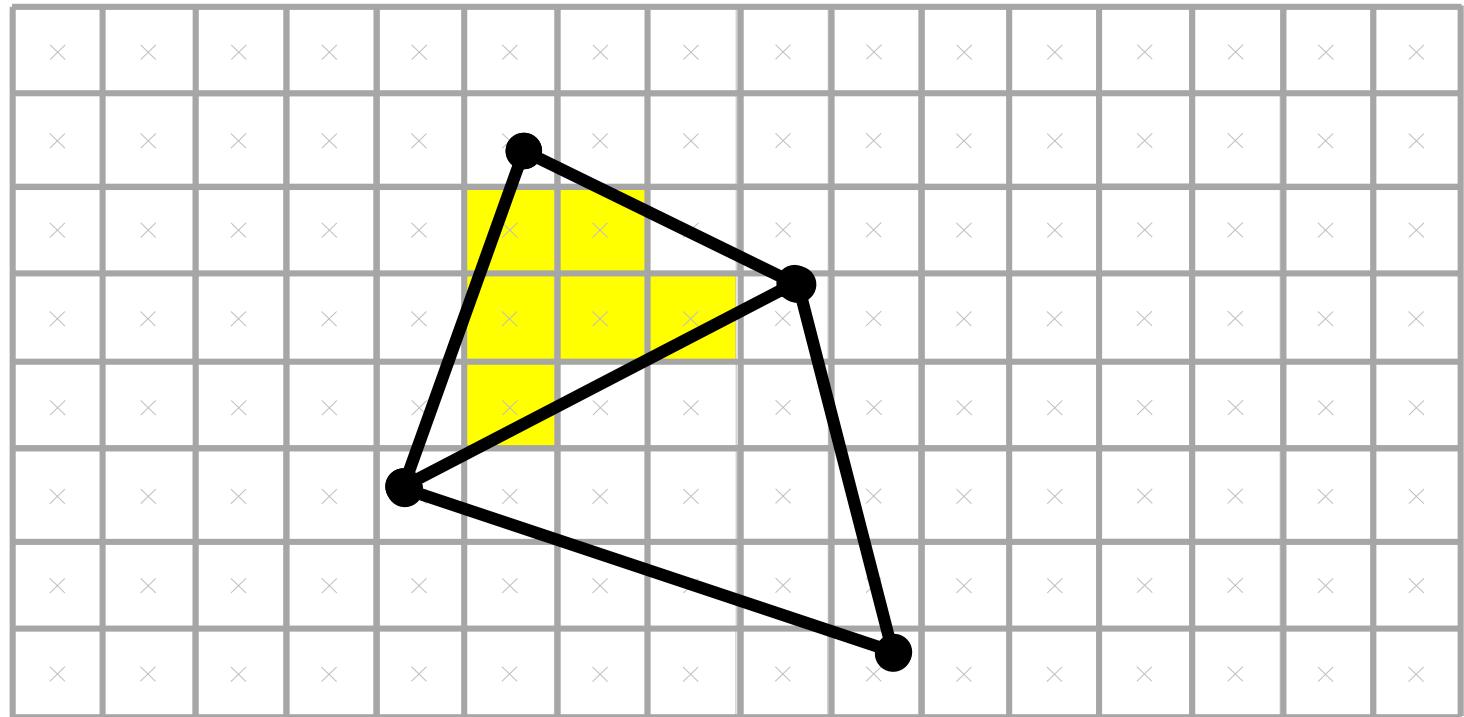


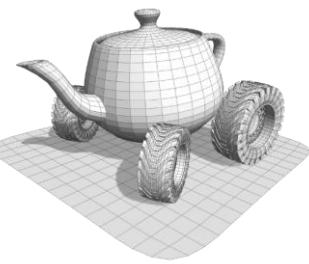
Rasterizing Triangles



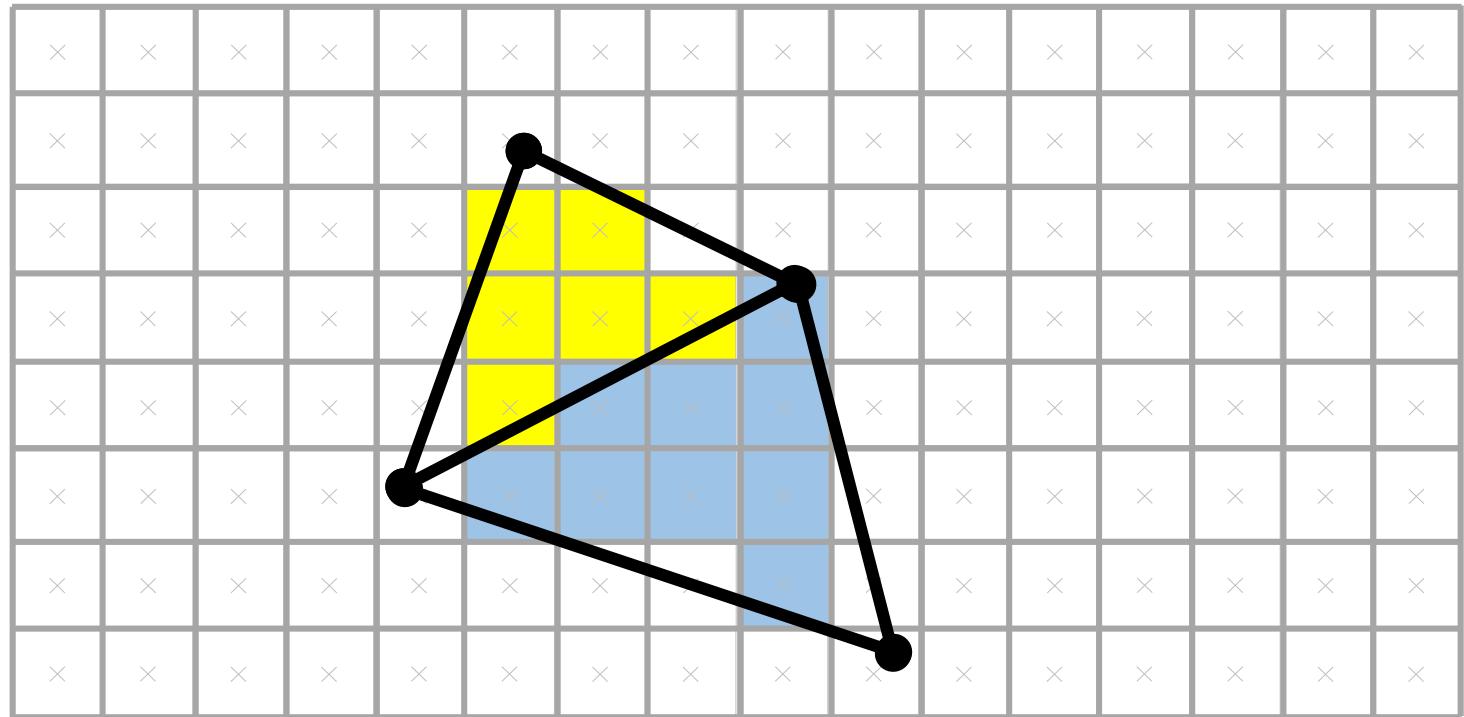


Rasterizing Triangles



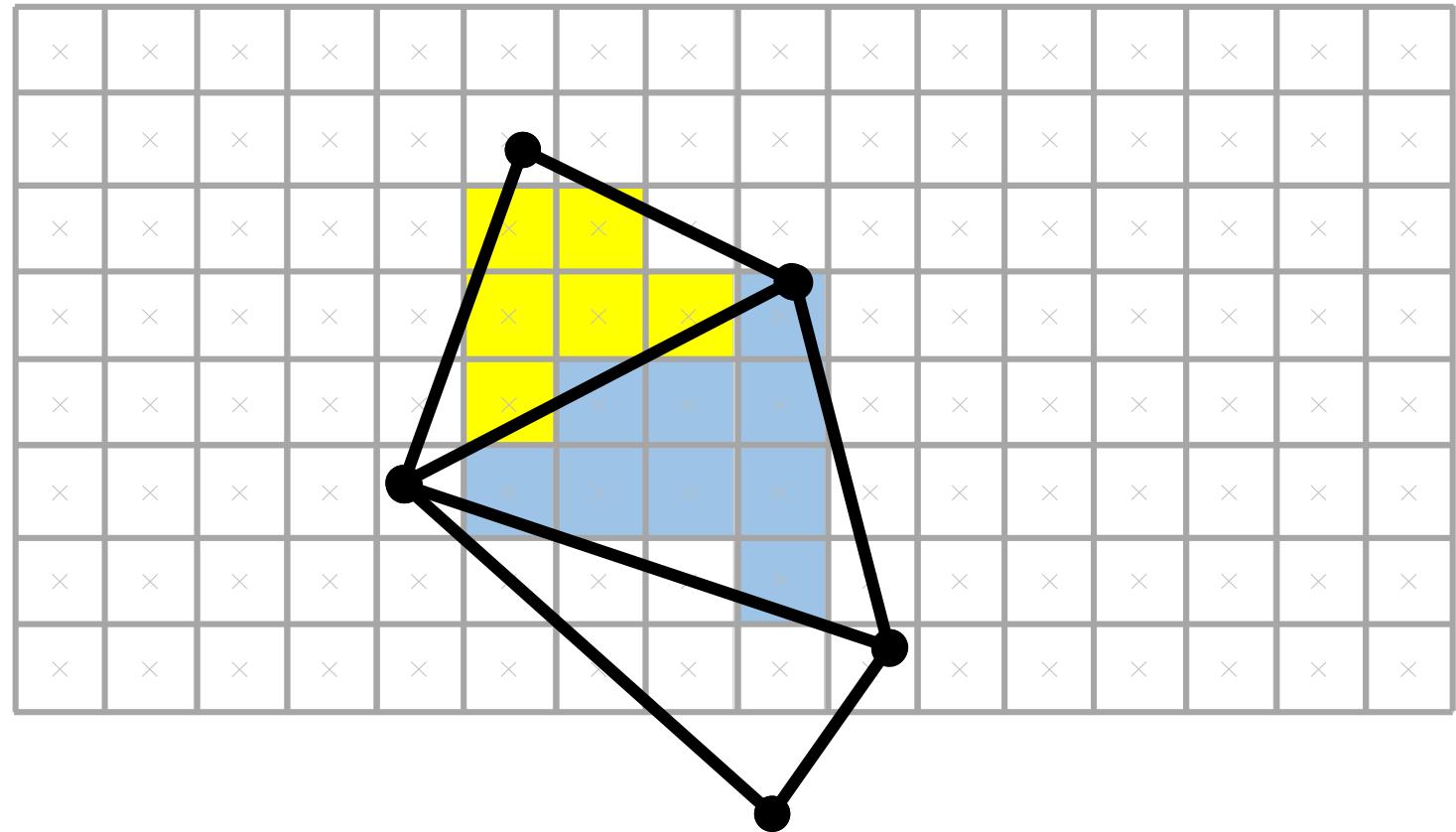


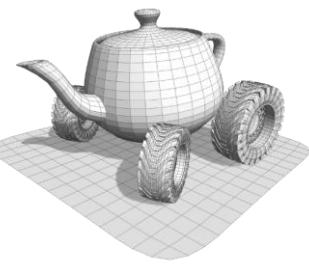
Rasterizing Triangles



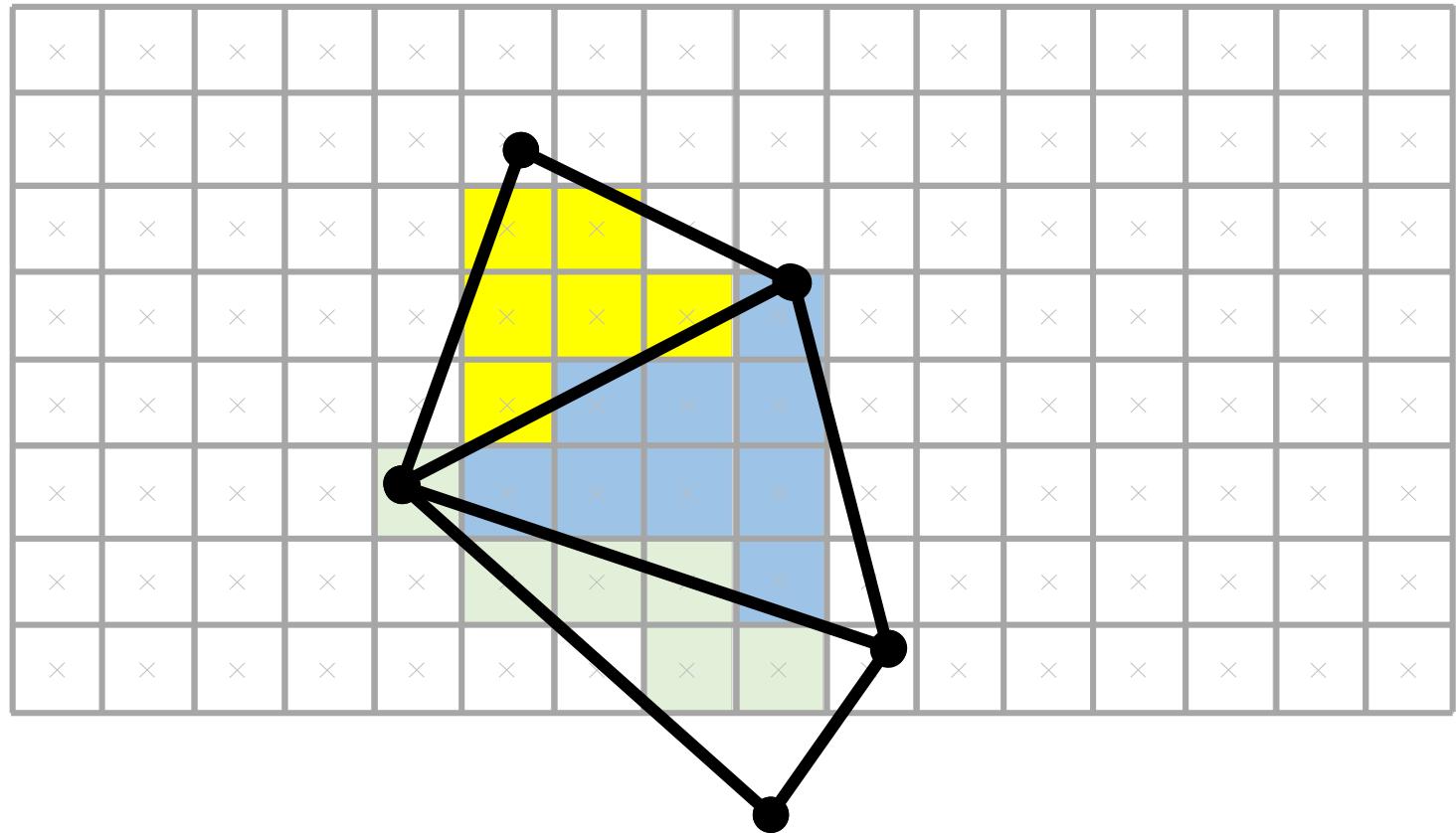


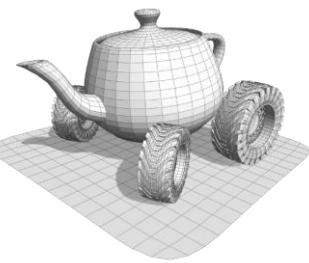
Rasterizing Triangles



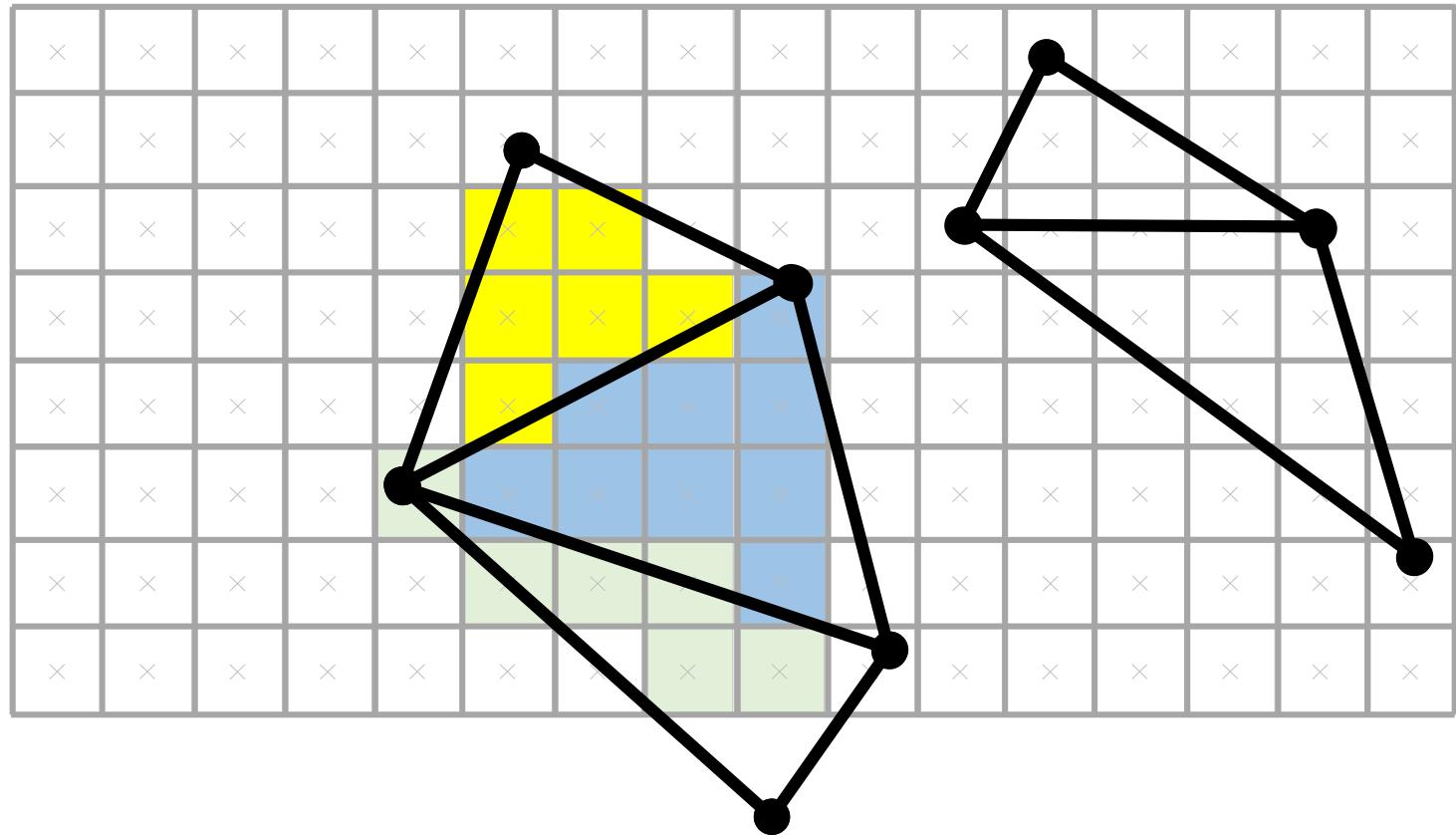


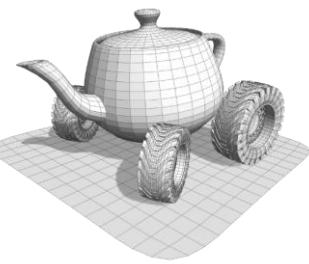
Rasterizing Triangles



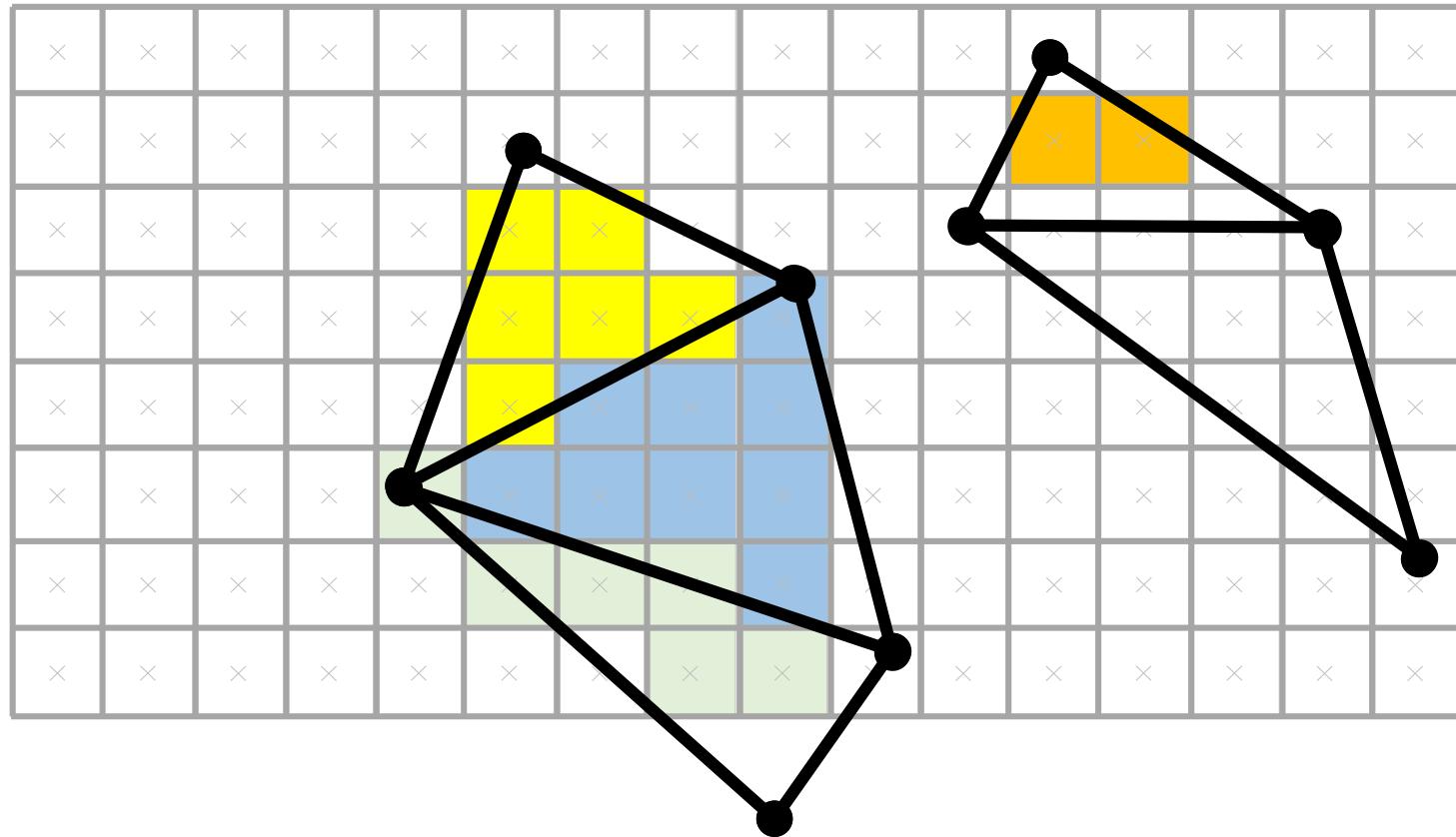


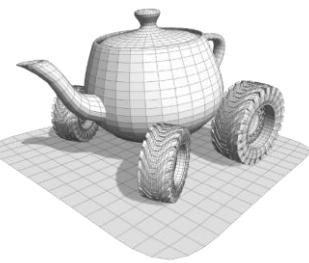
Rasterizing Triangles



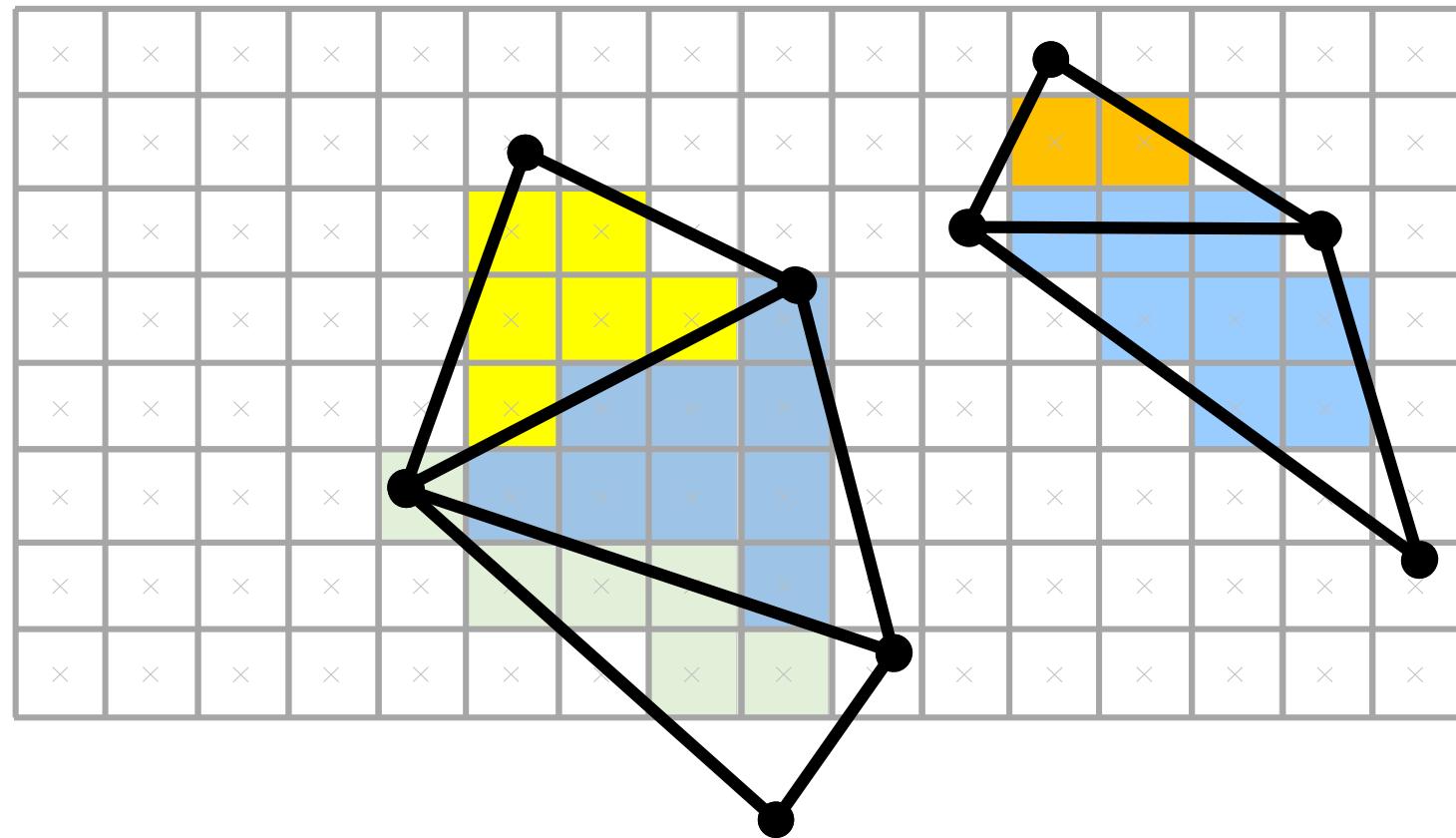


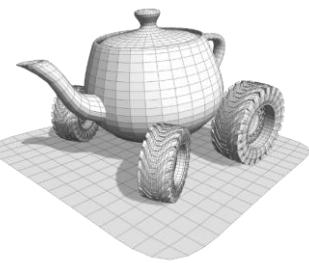
Rasterizing Triangles



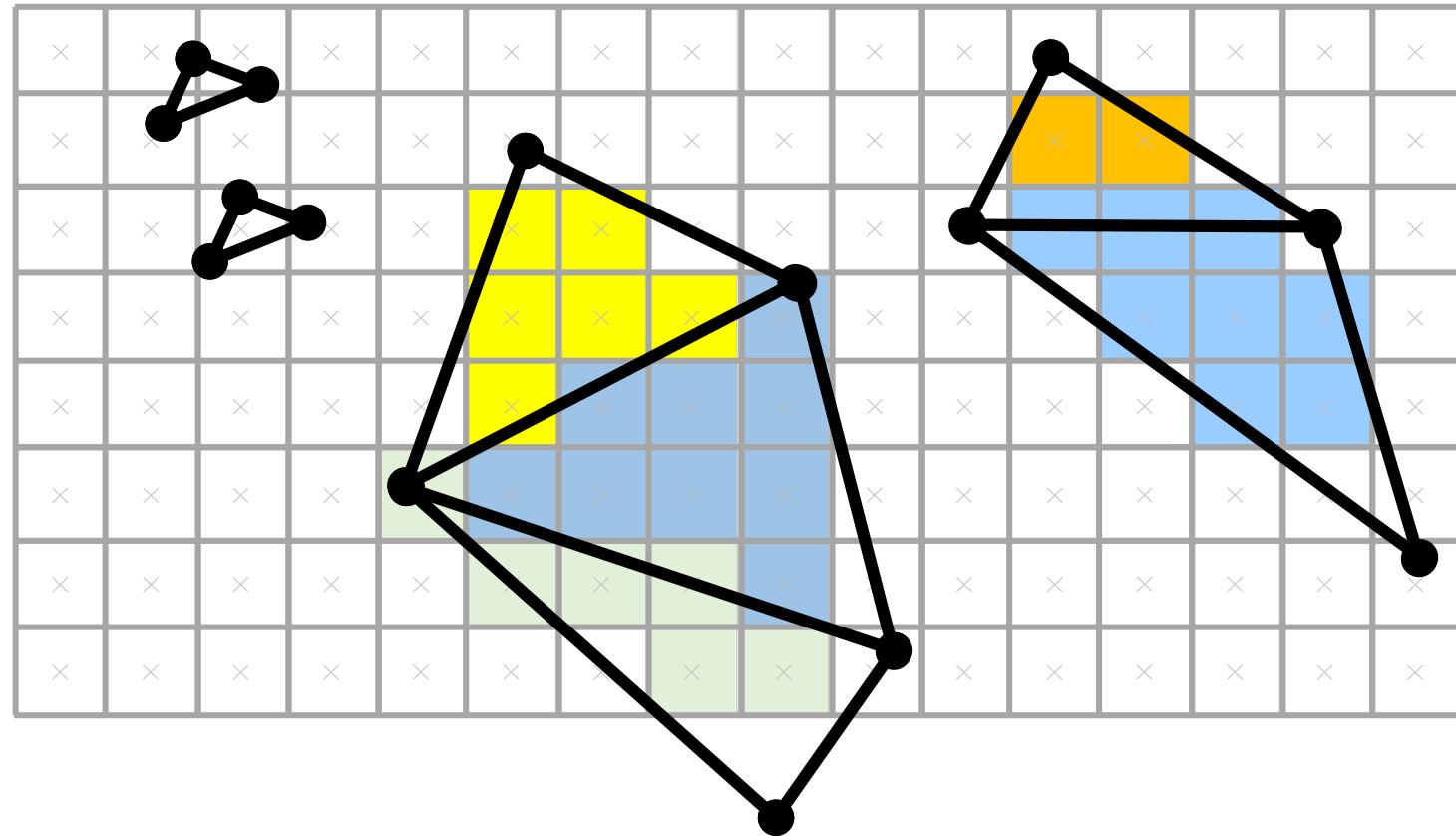


Rasterizing Triangles



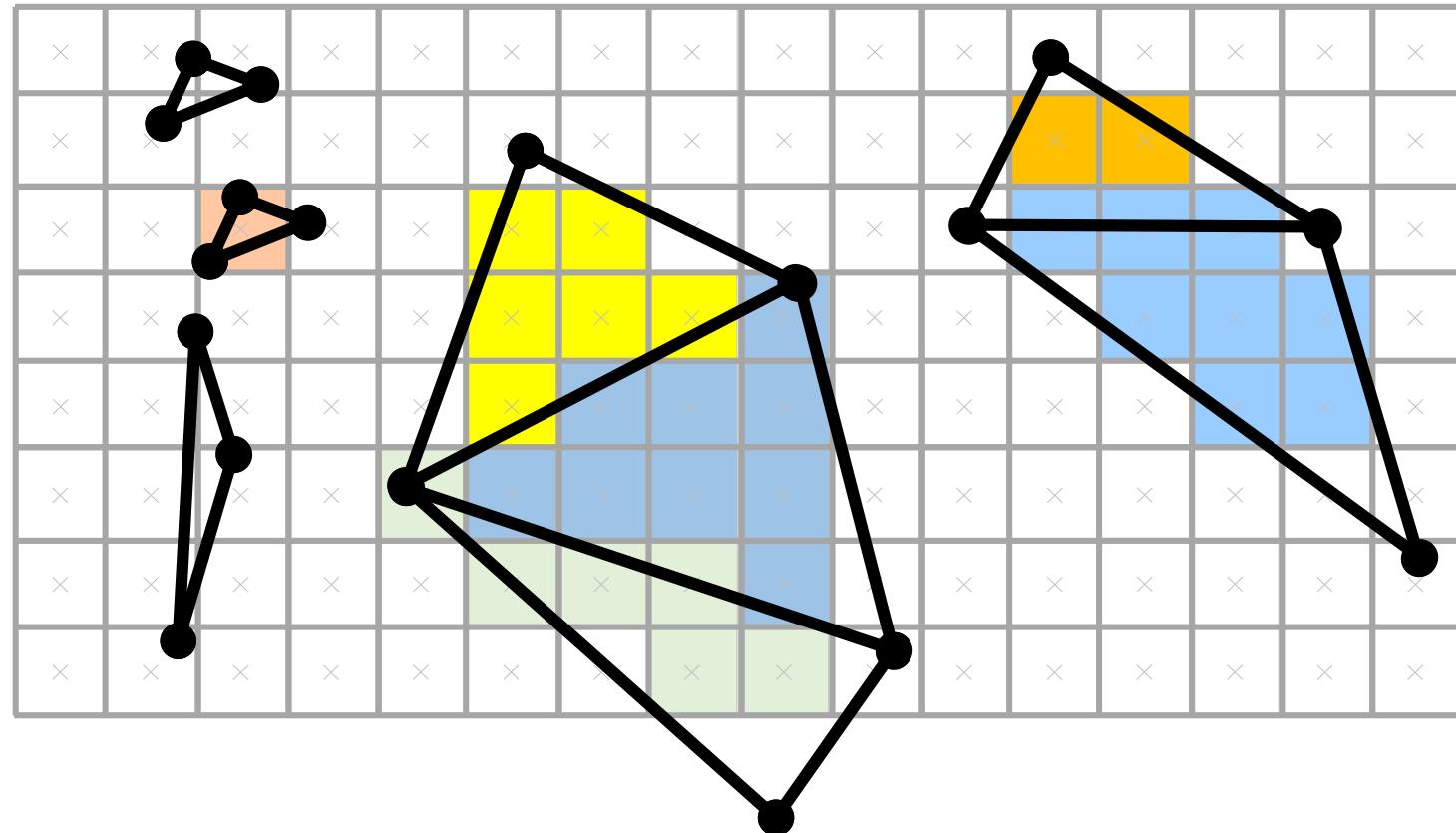


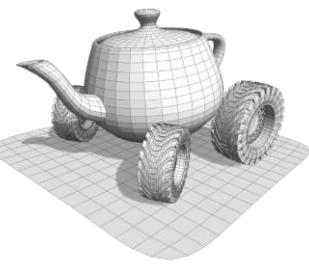
Rasterizing Triangles



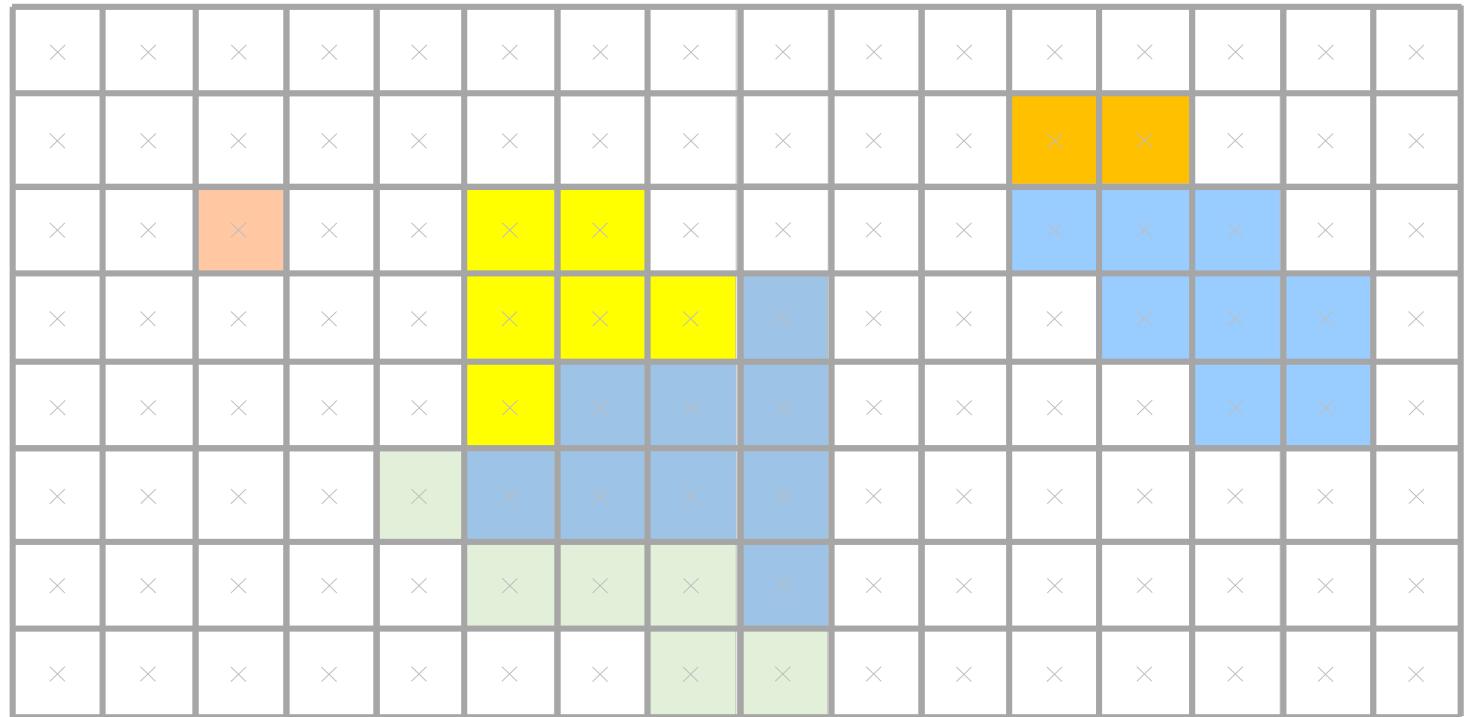


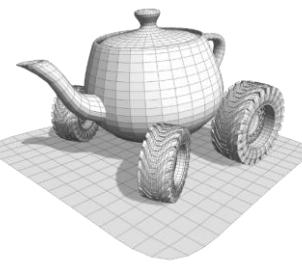
Rasterizing Triangles





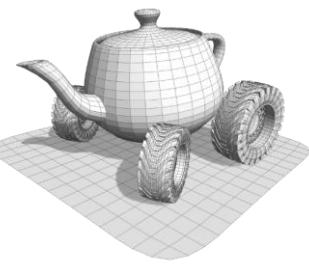
Rasterizing Triangles





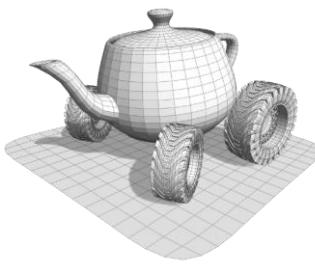
Rasterizing triangles: summarizing (1/2)

- input: three points in 2D (that is, on the screen)
 - Real coordinates coordinates XY
- output: fragments «covered» by the triangles
- A pixel is «covered» **iff** its center is inside the triangle
 - Q: why not just “iff the pixel intersects the triangle” ?
- A triangle can cover any number of fragments
 - No one, some, all of them
- Only fragments within the viewport (that is, the screen) are tested/generated.



Rasterizing triangles: summarizing (2/2)

- Fragments are only generated for the portion of the triangle inside the viewport
 - The part outside is «clipped» away (see «clipping» later on...)
 - If the triangle is entirely outside the viewport no fragment is generated (see «culling» later on)
- Basic test of the rasterizer
 - Is the center of this pixel inside the triangle?
- The union of two triangles sharing one edge (edge == segment connecting two vertices of the triangle) creates one and only one fragment for each pixel
 - There is no overlapping (two triangles generating the two fragments for the same pixel)
 - There are no gaps (that is, pixels covered by the union of triangles with no associated fragment)



Rasterization-based : cost

- Main core:
 - Processing vertices & rasterizing primitives
- Computationally simple
- Still the main choice for online applications

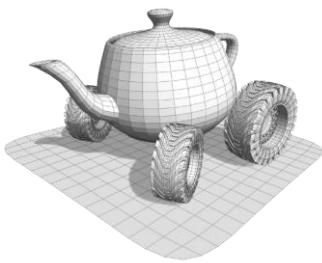
$$CostR = K \#vertex + \sum_{\forall p \in S} R(p)$$

Cost for «transforming» a vertex

Depends on the primitive size on screen

Number of vertices

Cost of rasterizing primitive p



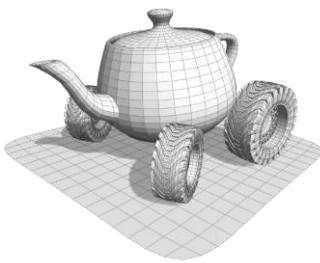
RASTERIZATION

VS

RAY-TRACING

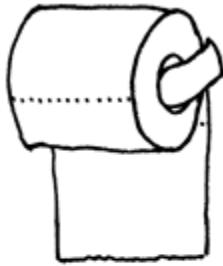


Rendering Algorithms Paradigms



RAY-TRACING

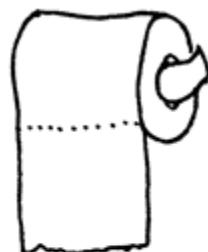
**for each pixel
for each primitive**



Like
this?

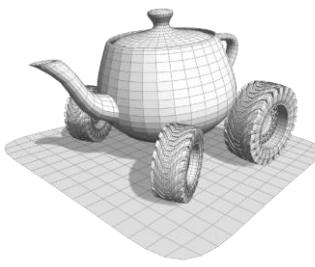
RASTERIZATION
BASED:

**for each primitive
for each pixel**

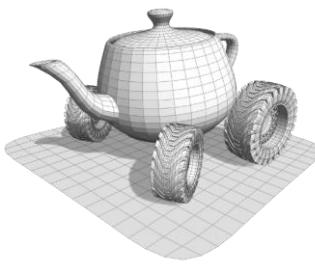


Or like
this?

Advantages of ray tracing



- Conceptually simple algorithm
- Takes into account GLOBAL effects
- More realistic rendering of lighting effects
- *More freedom of primitives*
(intersect it's easier than project+rasterize)
- Potentially great scalability with scene complexity

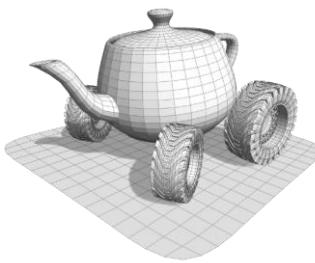


Ray-tracing : simple and elegant?

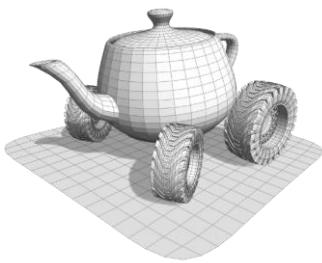
```
typedef struct{double x,y,z}vec;vec U,black,amb=(.02,.02,.02);struct sphere{vec cen,color;double rad,kd,ks,kt,k1,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,1.,5.,0.,0.,0.,.5,1.5,};yx:double u,b,tmin,sqrt(),tan():double vdot(A,B)vec A,B:(return A.x*B.x+A.y*B.y+A.z*B.z:)vec vcomb(a,A,B)double a:vec A,B:(B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return B;)vec vunit(A)vec A:(return vcomb(1./sqrt(vdot(A,A)),A,black));}struct sphere*intersect(P,D)vec P,D:(best=0;tmin=1e30;s=sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u:tmin;return best;)vec trace(level,P,D)vec P,D:(double d,eta,e:vec N,color;struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D)):else return amb;color=amb;eta=s->ir;d=-vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d=-d;l=sph+5;while(l-->sph)if((e=l->k1*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e,l->color,color);U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,color,vcomb(s->k1,U,black))));}main(){printf("%d %d\n",32,32):while(yx<32*32)U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}//*minray!*/
```

A whole raytracer on a business card :-P !
(by Paul Heckbert)

Advantages of rasterization

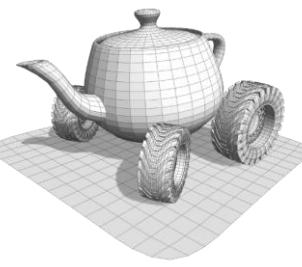


- More easily parallelizable?
- More controllable complexity
- Better-suited for graphics cards (for now)
- Easier with dynamic data?
- Better treatment of anti-aliasing (more later on)
- Better scalability with image resolution



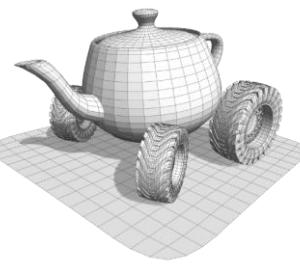
Historic View

- Ray-tracing
 - algorithm of choice for OFFLINE rendering
 - used for MOVIES (and CGI static images)
 - Pixar, Dreamworks, etc. favourite
 - well known ray-tracers: POV-ray, renderman, YafaRay...
- Rasterization
 - algorithm of choice for REAL-TIME rendering
 - used for GAMES (and previews, and 3D on web...)
 - Nvidia, ATI, etc. favourite
 - First to get specialized HW



The commonplaces

- **Raytracing:**
 - good for complex visual effects
 - shadows, specular reflections, refractions...
 - SW-based (CPU)
 - *therefore*: off-line, hi-quality renderings!
- **Rasterization:**
 - fast!
 - for each primitive, process only a few pixels
(instead of: for each pixel, process *all* primitives)
 - HW-based (GPU)
 - *therefore*: real-time, compromise-quality renderings!



Reality : Ray-Tracing

.. not necessarily SLOW!

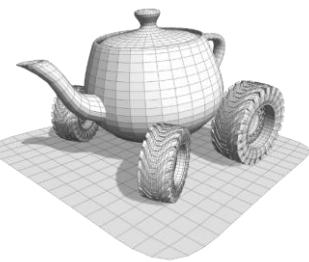
- algorithm and data structures for efficient “**spatial queries**”
- even **sub-linear** with number of primitive!

.. not necessarily SW:

- Specialized HW?



OpenRT Project
inTrace Realtime Ray Tracing Technologies GmbH
MPI Informatik, Saarbruecken - Ingo Wald 2004

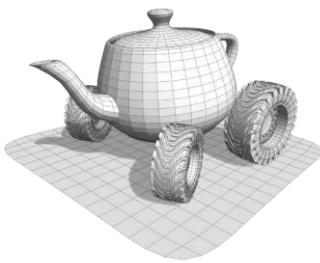


Reality: Rasterization based

- ..not necessarily without complex visual effect
(they just require some approx and a few advanced algorithms)



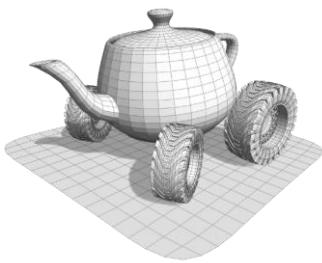
The reality



“ *Rasterization is fast,
but needs cleverness
to support complex visual effects.*

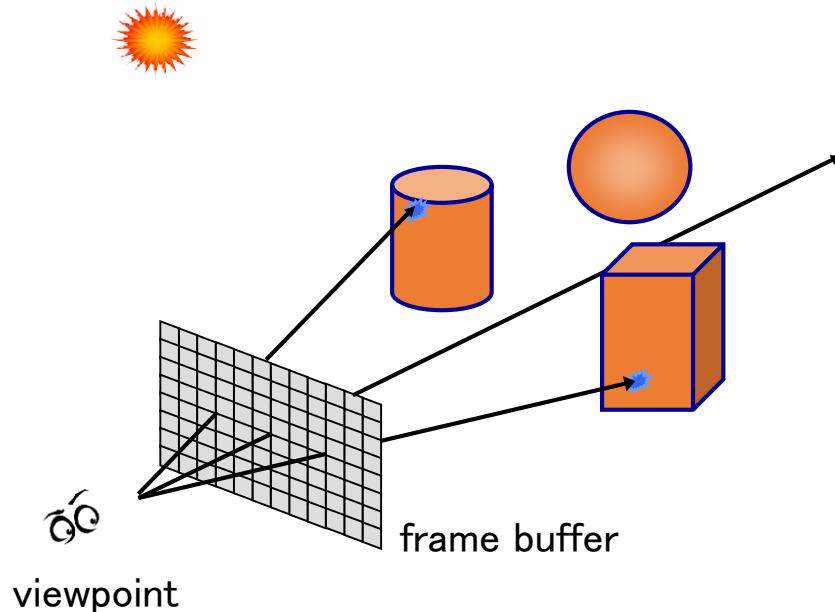
*Ray tracing supports complex visual effects,
but needs cleverness
to be fast.*

David Luebke (NVIDIA)



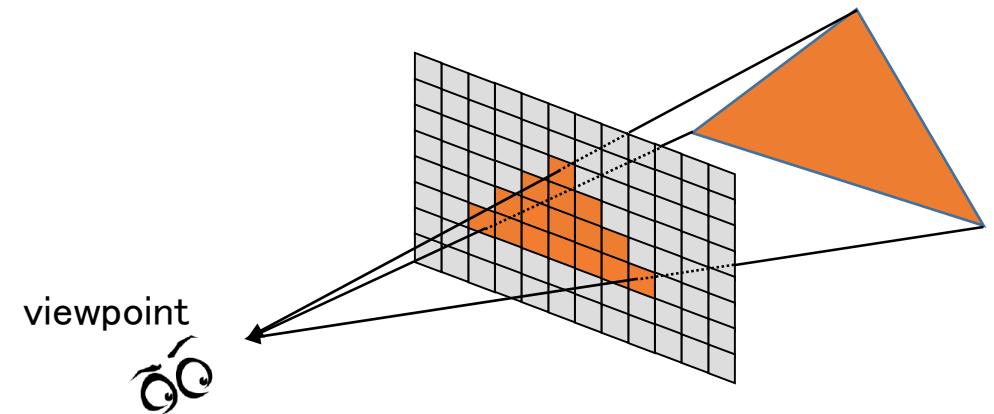
Why rasterization «won» so far?

- Won means «all the graphics boards have been designed for rasterization up until now»



Ray tracing

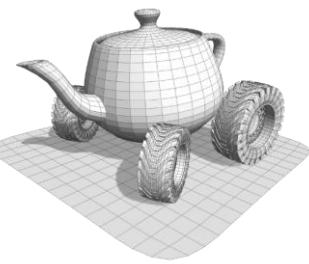
Chapter 1: Rendering Paradigms



Rasterization

52

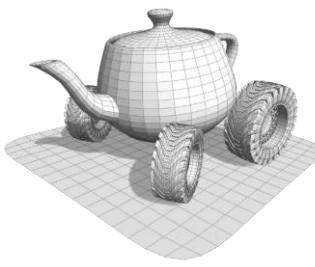
Introduction to Computer Graphics: a Practical Learning Approach



Why rasterization «won»* so far? (1/2)

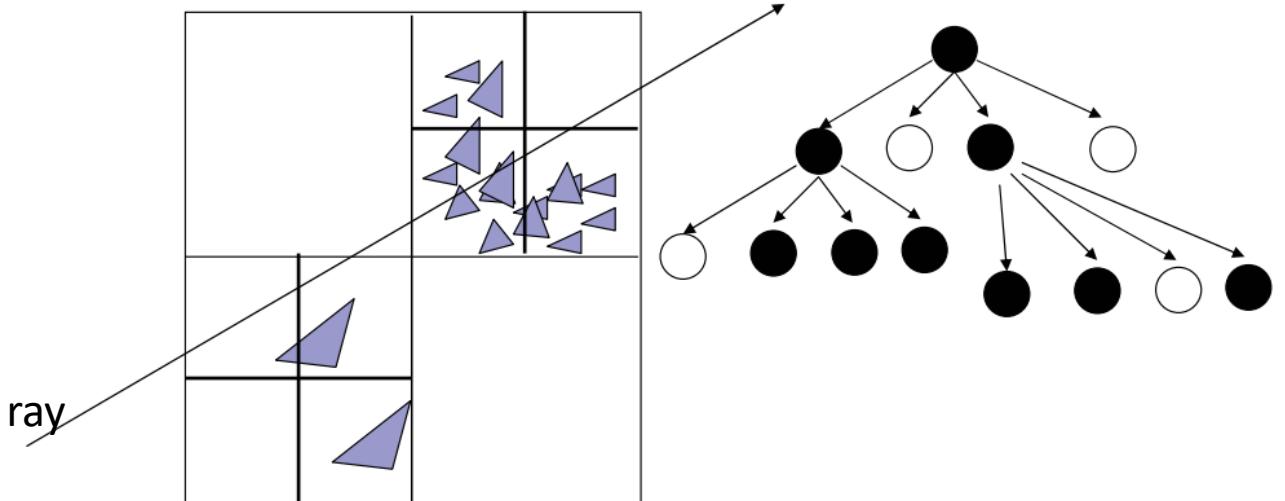
- Both need to find which portions of the entire description of the scene are *visible* from the current point of view. This problem is referred to as the **Hidden Surface Removal**
- Both Ray Tracing and Rasterization need to process every geometric primitive in the scene, but:
 - Ray Tracing needs the whole scene in memory *unless* ad hoc algorithms are used (hierarchies of bounding volumes...more later on)
 - Ray Tracing is naturally parallelizable but we cannot (yet) process *all* pixels in parallel with the number of rays required (several per pixels x N bounces)
 - Rasterization *runs* over all the primitives once and it's done

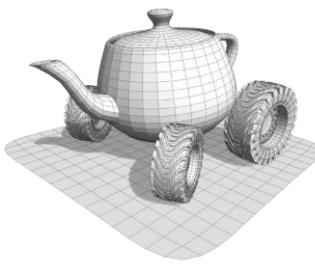
* Won means «all the graphics boards have been designed for rasterization up until now»



Ray Tracing acceleration

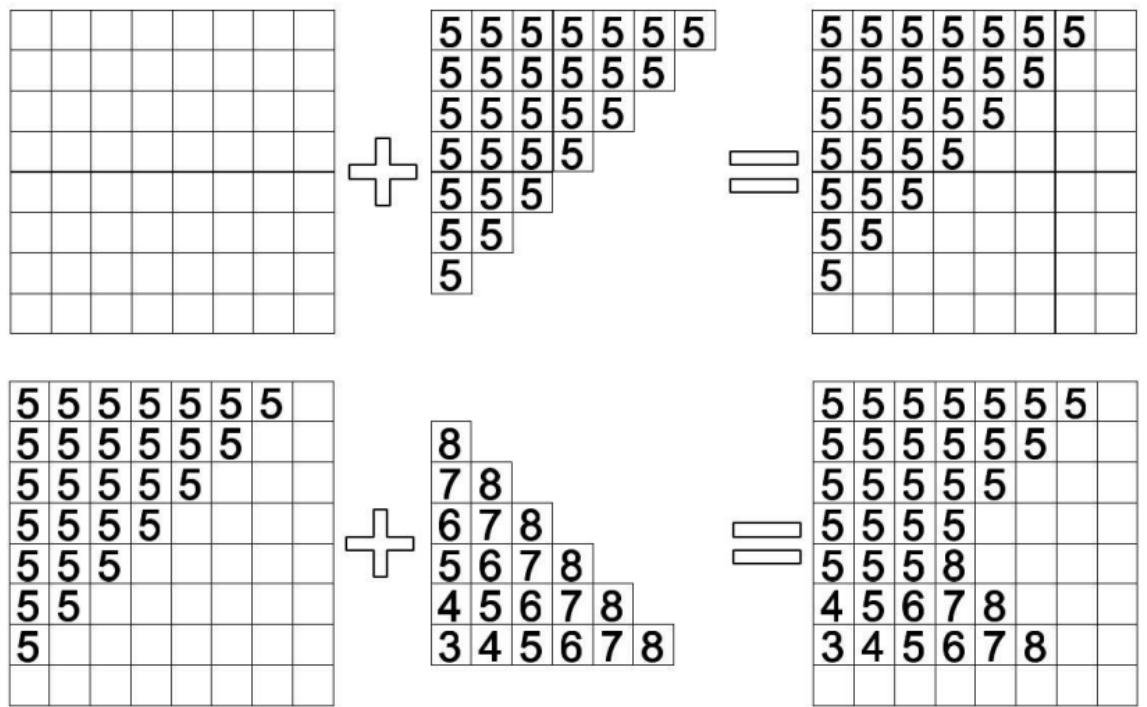
- **Bounding Volumes Hierarchies (BVH)**
 - Bounding Volume of a region of space: a geometric entity (e.g. A box, a sphere) that includes said region
 - Once built, a **BVH** reduce the cost of the intersection to $O(\log n)$ (on average)

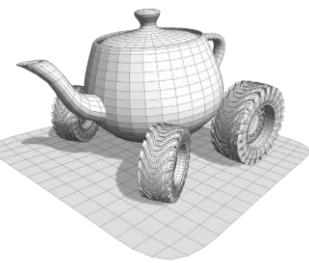




Rasterization acceleration: Z-Buffer

- The rasterization algorithm computes, for each pixel, the *distance* of the corresponding surface from the point of view
- Such distance is written into a memory buffer the same size of the screen, called **Depth (or Z) Buffer**
- Unless the Z-Buffer already contains a smaller value for that pixel

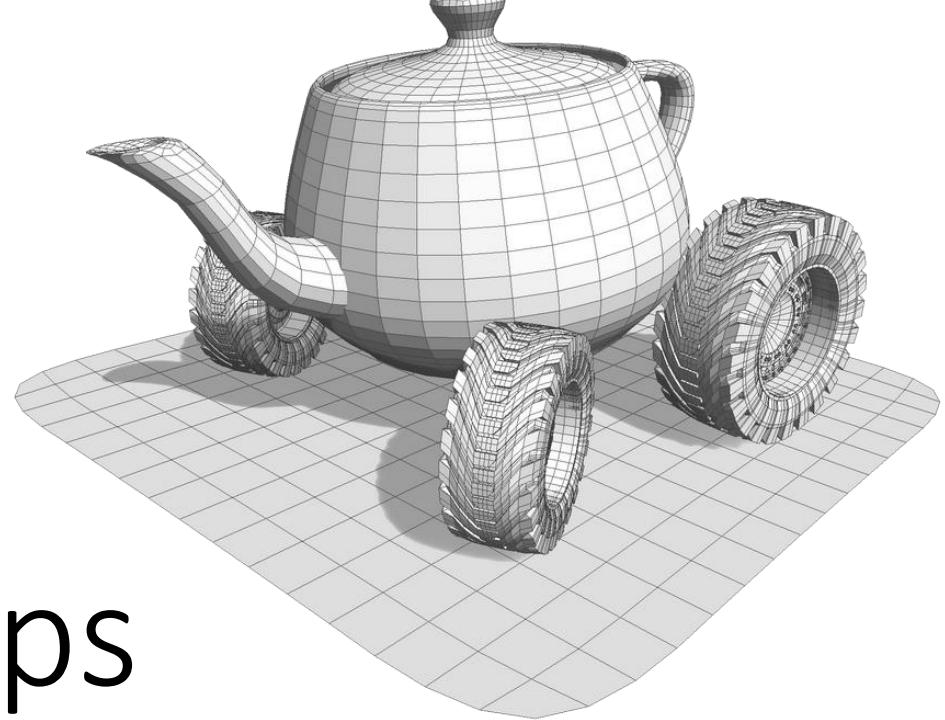




Why rasterization «won»* so far? (2/2)

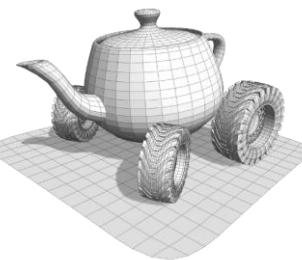
- Both need to find which portions of the entire description of the scene are *visible* from the current point of view. This problem is referred to as the **Hidden Surface Removal**
- Both Ray Tracing and Rasterization need to process every geometric primitive in the scene, but:
 - Ray Tracing needs the whole scene in memory *unless* ad hoc algorithms are used (hierarchies of bounding volumes...more later on)
 - Ray Tracing is naturally parallelizable but we cannot (yet) process *all* pixels in parallel with the number of rays required (several per pixels x N bounces)
 - Rasterization *runs* over all the primitives once and it's done
- Rasterization with z-buffer is an extremely simple and has a more predictable cost

* Won means «all the graphics boards have been designed for rasterization up until now»



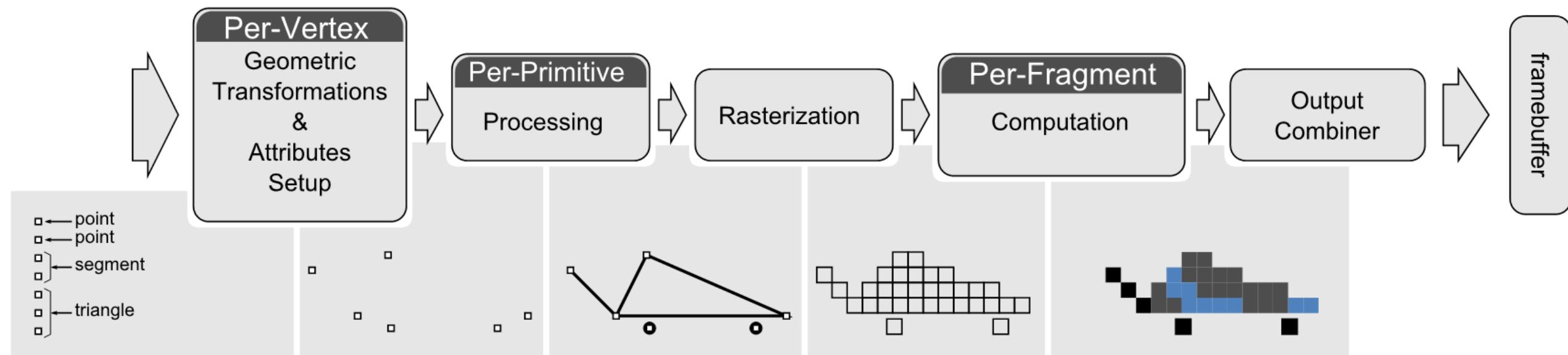
The First Steps

Let's draw a triangle!

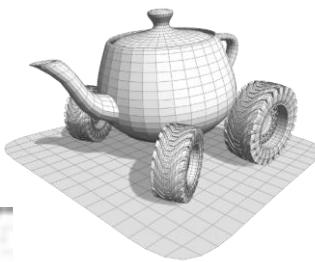


Rasterization-based pipeline

- How is the rasterization based pipeline implemented?

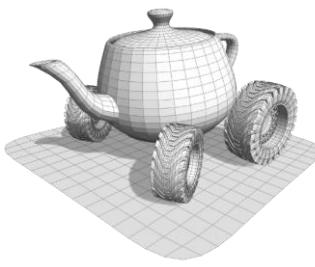


OpenGL



- Initially developed by Silicon Graphics
- From 2002 to 2006:
OpenGL **A**rchitecture **R**eview **B**oard
 - Updating specifications
 - 90% industry, 10% academy
 - Each company is a member
- from 2006 is the Khronos Group
 - Some branded extensions by nVidia





Other versions

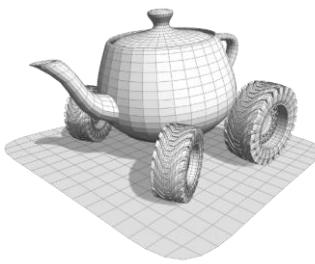


- per **embedded devices**
- Basically a subset of OpenGL



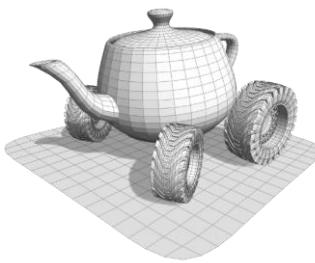
- Specifically written for the **web**
- Based on the version 2.0 of OpenGL|ES
- **HTML5**
- Binding for **JavaScript**
- **No plugin required**

OpenGL alternatives



- by Khronos (again)
- In a way is a more advanced version of OpenGL
 - Lower Level, more control on memory
 - “bytecode” for the shaders, no driver compiling high level language
 - better debugging
 - unified mobile / embedded / desktop
- It will probably replace OpenGL in games but not for *general purpose* applications and not so fast..

OpenGL alternatives

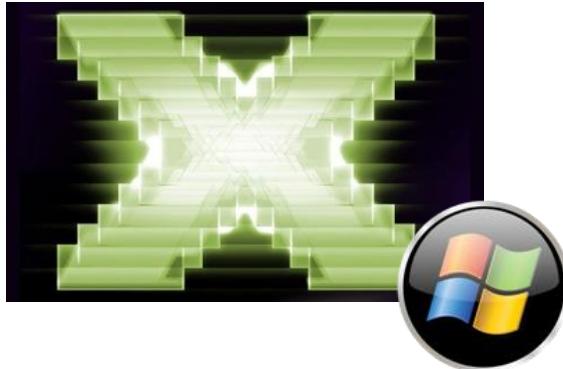


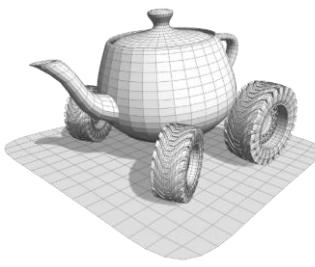
- **Direct3D**

- Microsoft only
- Part of DirectX
- Same goals as OpenGL
 - API to use the same GPUs
 - Similar organizations
 - C / C++
- Most common alternative to OpenGL
 - Direct3D = industrial standard (e.g. )
 - OpenGL = industrial + academic standard

- **Metal**

- Apple only
- Meant to replace OpenGL & Vulkan for Mac





Our First Triangle: the “deprecated” way

- **“Immediate mode”:** provide render data at every frame

Hey OpenGL, I'm beginning to provide the vertices to draw triangles

here's the first one ...

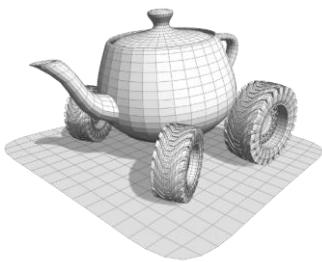
here's the second one ...

here's the third one ...

I'm done, please
draw the triangle

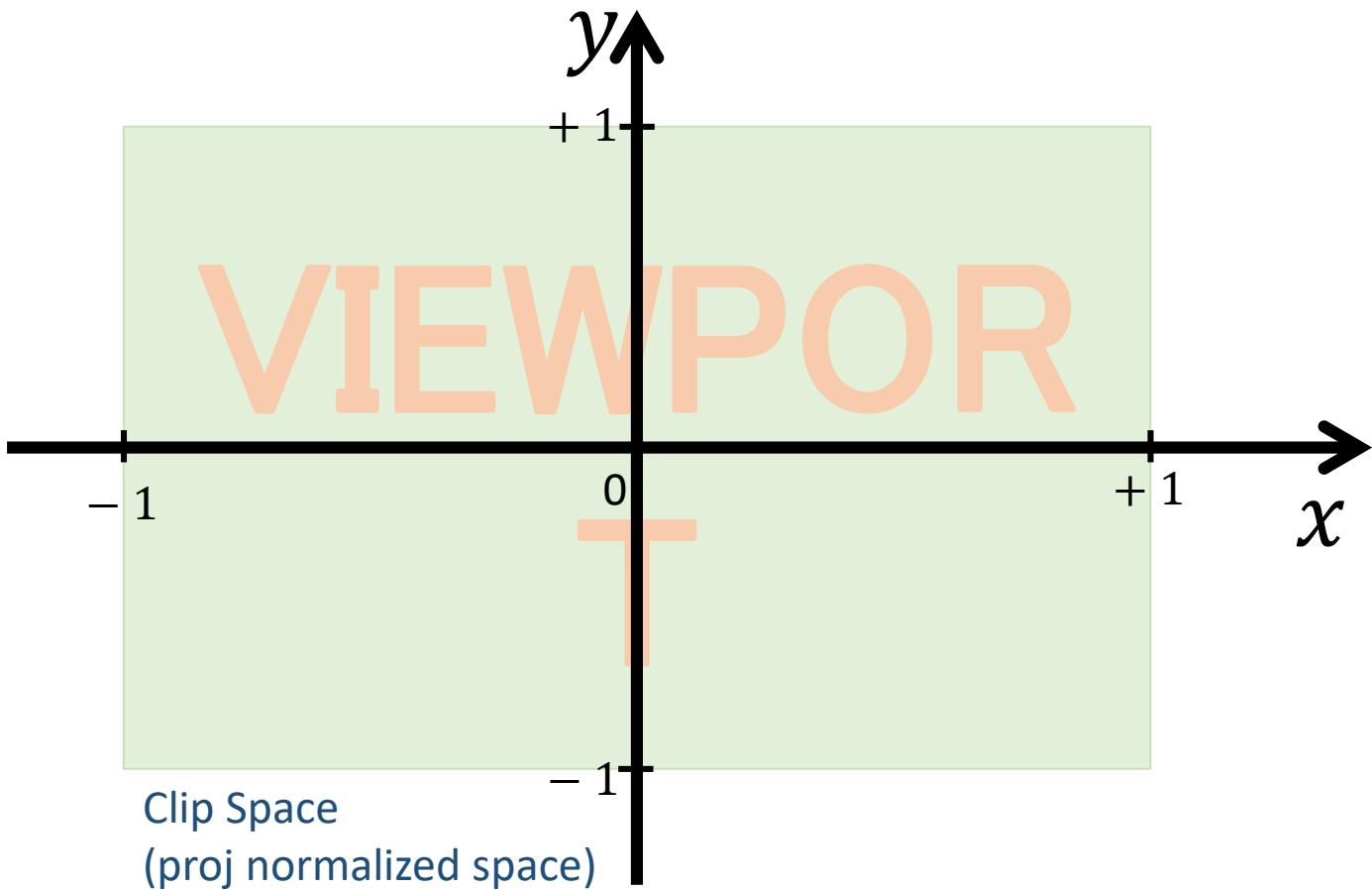
```
while (!glfwWindowShouldClose(window))  
{  
    /* Render here */  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    glBegin(GL_TRIANGLES);  
    glVertex3f(0.0, 0.0, 0.0);  
    glVertex3f(0.5, 0.0, 0.0);  
    glVertex3f(0.5, 0.5, 0.0);  
    glEnd();  
  
    /* Swap front and back buffers */  
    glfwSwapBuffers(window);  
}
```

It could be:
GL_LINES,
GL_QUADS
etc..

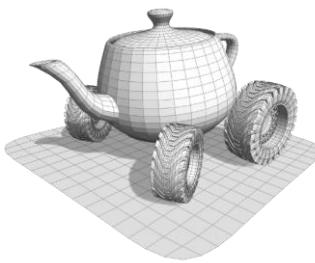


Clip Space

- All our coordinates will ultimately be projected in **clip space**, that is $[-1,1] \times [-1,1]$
- In this first program we write them directly in clip space



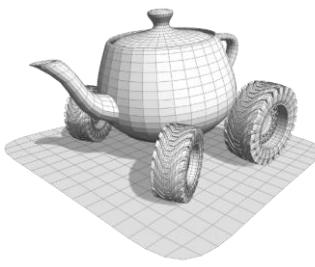
Our First Triangle: the “deprecated” way



- **“Immediate mode”:** provide render data at every frame
 - E.g. vertex data are kept in client memory and sent to the GPU at *every frame*
 - Lots of API calls, executed by the slow CPU

```
while (!glfwWindowShouldClose(window))  
{  
    /* Render here */  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    glBegin(GL_TRIANGLES);  
    glVertex3f(0.0, 0.0, 0.0);  
    glVertex3f(0.5, 0.0, 0.0);  
    glVertex3f(0.5, 0.5, 0.0);  
    glEnd();  
  
    /* Swap front and back buffers */  
    glfwSwapBuffers(window);  
}
```

Immediate mode



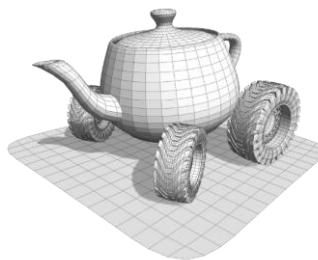
Our First Triangle: the “proper” way

- “**Retained mode**”: provide render data once (or rarely)
 - E.g. vertex data are transferred to GPU memory *once*
 - Fewer API calls

```
float * render_data = create_render_data_in_RAM();
handle = create_buffer_in_VideoRAM();
transfer_data_from_RAM_to_VideoRAM(render_data, handle);

while (!glfwWindowShouldClose(window))
{
    /* Render here */
    glClear(GL_COLOR_BUFFER_BIT);
    draw_render_data(handle);

    /* Swap front and back buffers */
    glfwSwapBuffers(window);
}
```



Our First Triangle: the “proper” way

Creates a memory buffer on GPU memory
(just the reference, size is unknown)

Binds the target `GL_ARRAY_BUFFER` to
the buffer just created

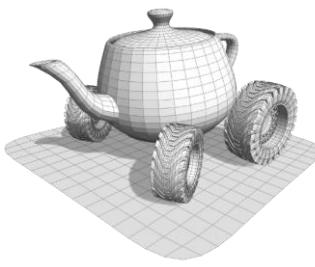
Loads the data into the buffer

`positionAttribIndex` is the name of
the array of position in the next stage of
the pipeline. Tells OpenGL we are going to
use it. Turn its slot “on”

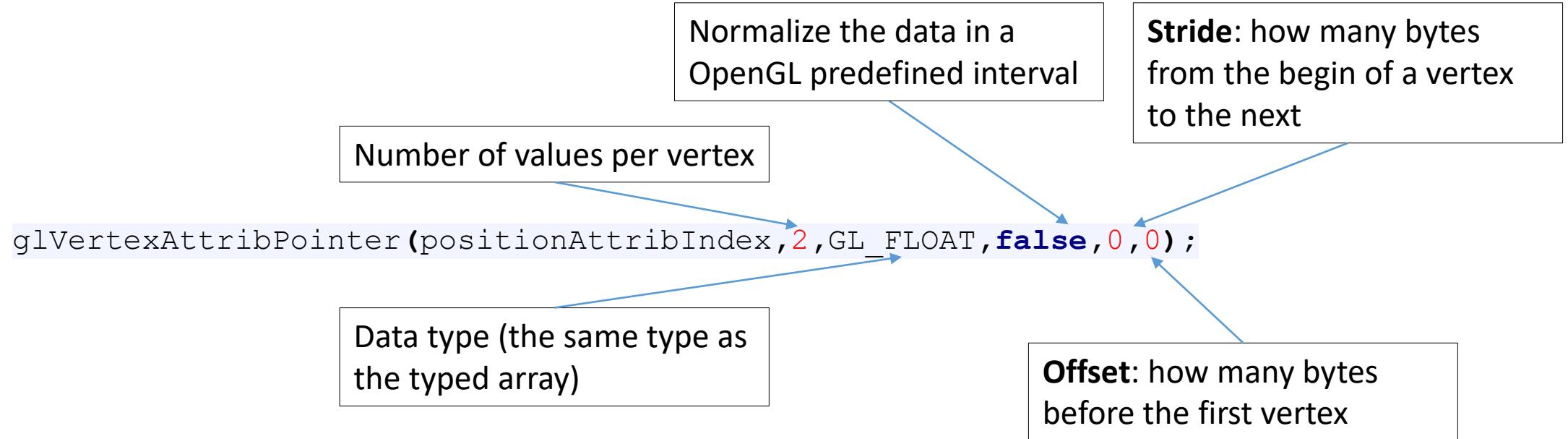
Tell OpenGL how to interpret the data
(more on slide 14)

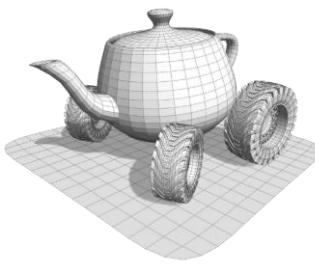
```
/* create render data in RAM */  
GLuint positionAttribIndex = 0;  
float positions[] = { 0.0, 0.0, // 1st vertex  
                      0.5, 0.0, // 2nd vertex  
                      0.5, 0.5 // 3rd vertex  
};  
  
/* create a buffer for the render data in video RAM */  
GLuint positionsBuffer;  
glCreateBuffers(1, &positionsBuffer);  
glBindBuffer(GL_ARRAY_BUFFER, positionsBuffer);  
  
/* declare what data in RAM are filling the buffer in video RAM */  
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 6, positions, GL_STATIC_DRAW);  
glEnableVertexAttribArray(positionAttribIndex);  
  
/* specify the data format */  
glVertexAttribPointer(positionAttribIndex, 2, GL_FLOAT, false, 0, 0);  
  
/* issue the draw command */  
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Hints OpenGL that there data are not
going to be changed. OpenGL
implementation can optimize the data
organization as a consequence



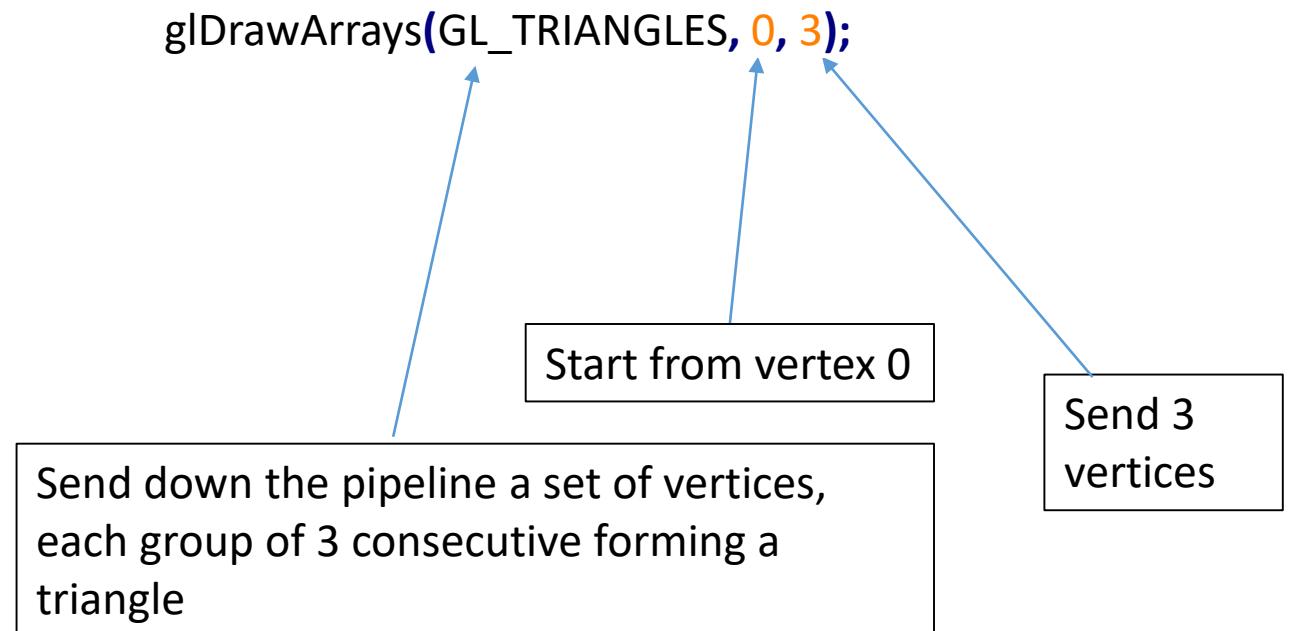
Describing the render data

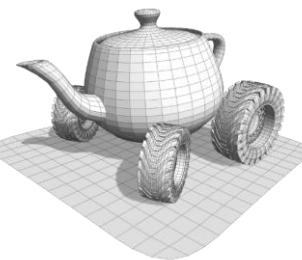




Our First Triangle: the draw command

- `glDrawArrays` (and similar commands) tells opengl to render the primitives for which the data have already been defined
- This function is invoked at every frame



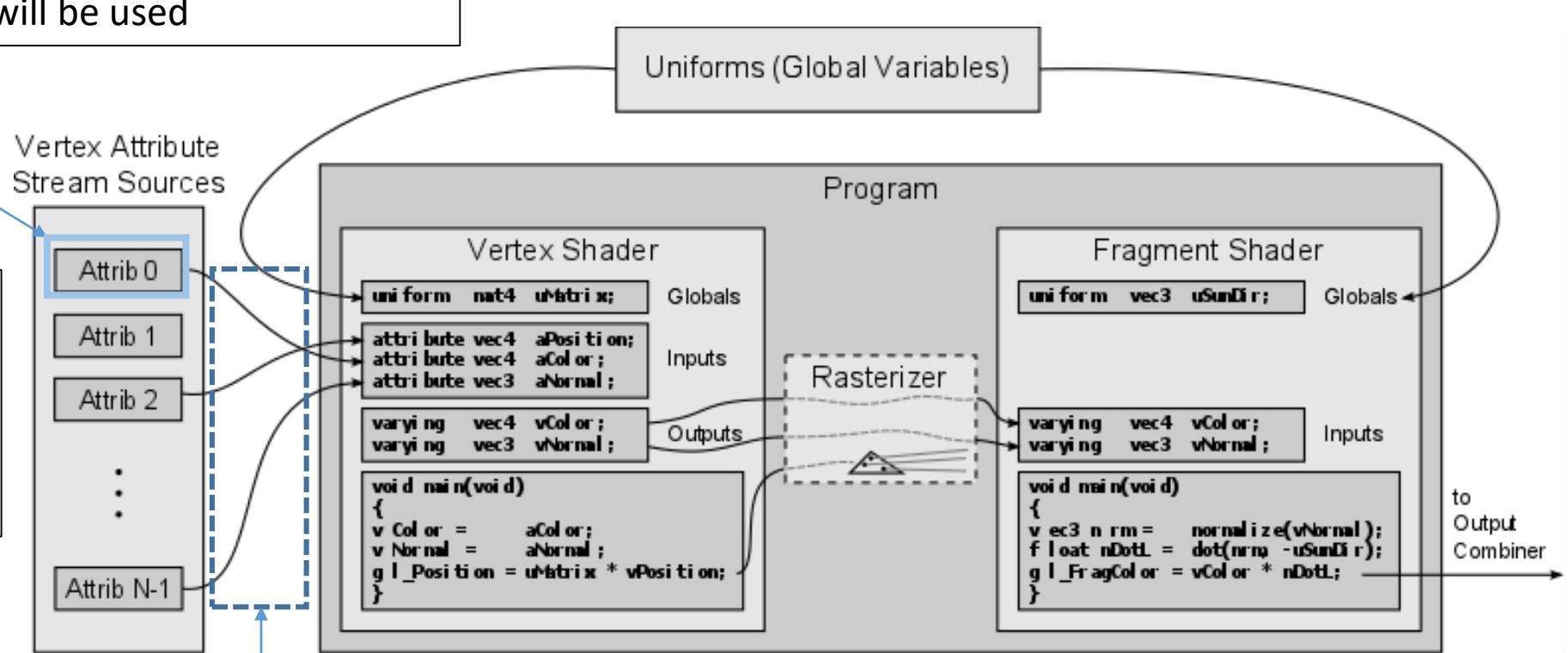


Our First Triangle: Vertex Flow

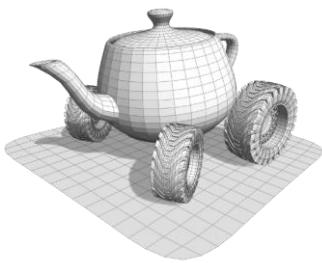
`glEnableVertexAttribArray(0)`

means the slot 0 will be used

A vertex is a set of attributes. It's up to the developer to decide what they are

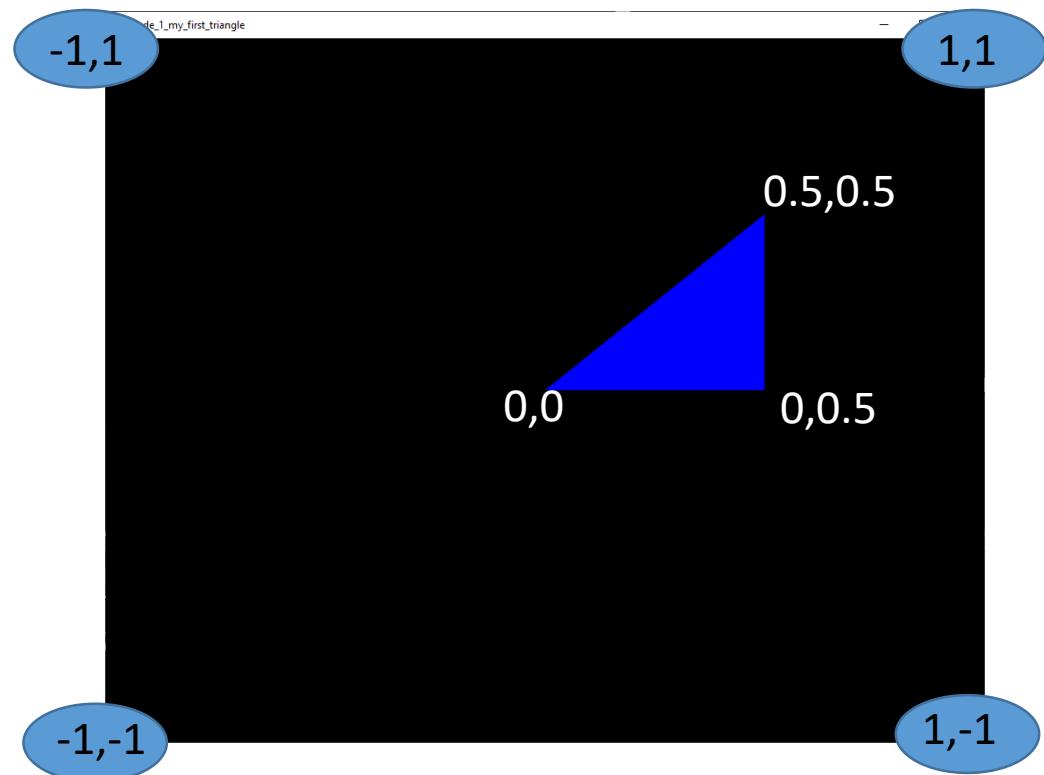


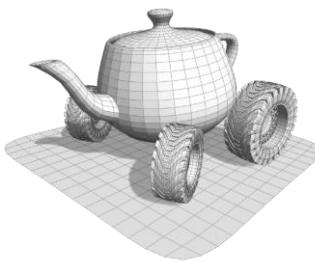
How to decide which slot is connected to which attribute? More on this later



Our First Triangle: Result

- Finally!





OpenGL

- Very common pattern:
 - Create an instance of some type of GL object
 - Bind a GL target to that specific instance
 - Run commands on the target

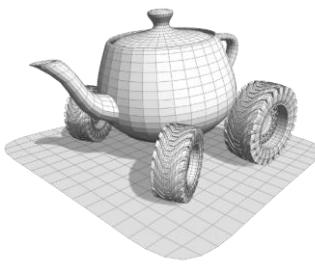
functions `glBind*` bind some GL target to a specific instance of it. From now on, all calls acting on the target `GL_ARRAY_BUFFER` affect `positionsBuffer`

functions `glCreate*` create **some GL object** and provide a **reference** to it

```
glCreateBuffers(1, &positionsBuffer);  
glBindBuffer(GL_ARRAY_BUFFER, positionsBuffer);
```

```
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 6, positions, GL_STATIC_DRAW);
```

The command `glBufferData` affects `positionsBuffer`

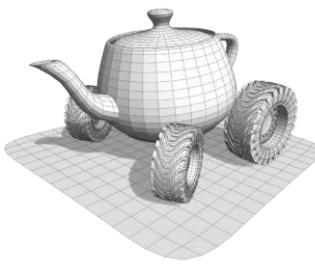


Our Second Triangle: Add Color

- Things to do:

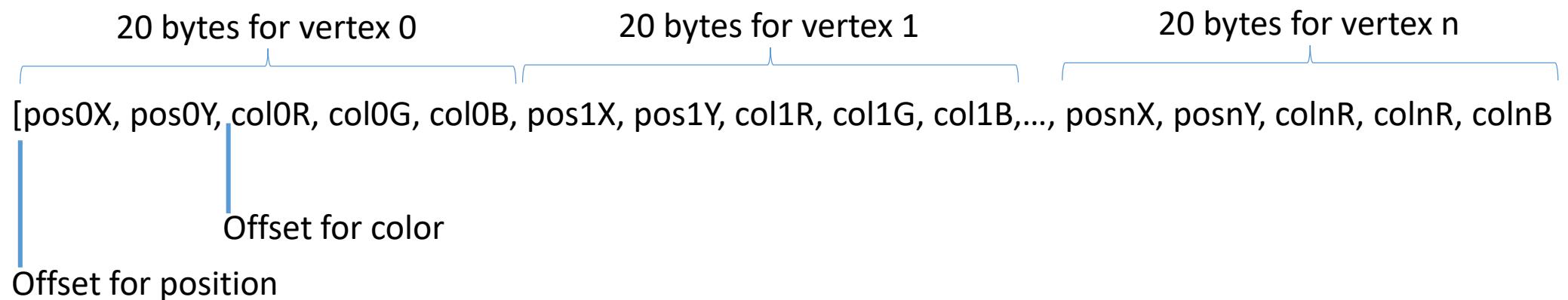
1. Create and fill a buffer for the color
2. Enable the attribute and tell OpenGL how to read it from the buffer

```
GLuint colorAttribIndex = 1;  
float colors[] = { 1.0, 0.0, 0.0, // 1st vertex  
                  0.0, 1.0, 0.0, // 2nd vertex  
                  0.0, 0.0, 1.0 // 3rd vertex  
};  
/* create a buffer for the render data in video RAM */  
GLuint colorsBuffer;  
glCreateBuffers(1, &colorsBuffer);  
glBindBuffer(GL_ARRAY_BUFFER, colorsBuffer);  
  
/* declare what data in RAM are filling the buffering video RAM */  
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 9, colors,  
GL_STATIC_DRAW);  
glEnableVertexAttribArray(colorAttribIndex);
```

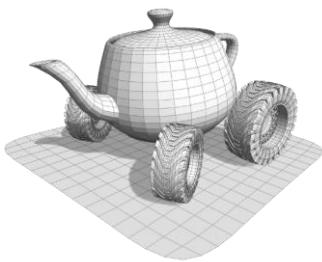


Our Second Triangle: Stride and Offset

- Stride and offset can be used to compact more data in the same array
 - For example we also have a color attribute for each vertex

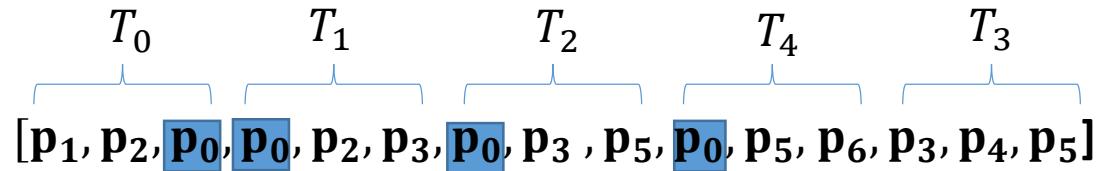


```
glVertexAttribPointer(positionAttribIndex, 2, GL_FLOAT, false, 20, 0);
glVertexAttribPointer(colorAttribIndex, 3, GL_FLOAT, false, 20, 8);
```

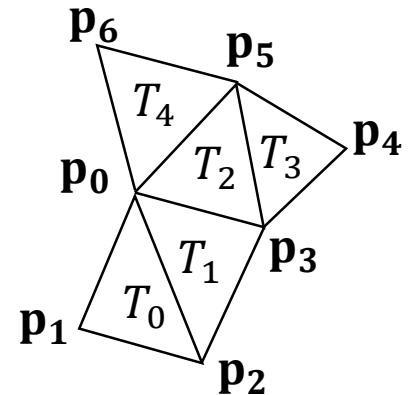


Indexed data structures (1/2)

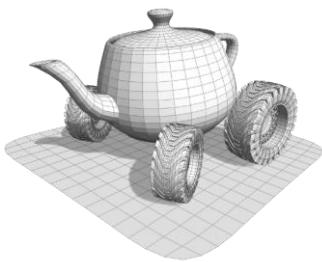
- Vertices are usually “shared” by more triangles
 - That is, repeated in the ARRAY_BUFFER for each triangle



- We can specify the positions only once and use an array of indices to specify the triangles
 - And index is an unsigned integer, a position is 3 floats
- $[p_0, p_1, p_2, p_3, p_4, p_5, p_6]$



i0	i1	i2
0	1	2
0	2	3
0	3	5
0	5	6
3	4	5



Indexed data structures (2/2)

Non indexed data

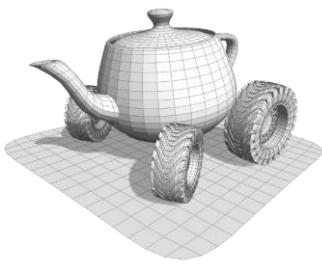
```
GLuint positionAttribIndex = 0;  
float positions[] = { 0.0, 0.0, // 1st vertex  
                      0.5, 0.0, // 2nd vertex  
                      0.5, 0.5, // 3rd vertex  
                      0.0, 0.0, // 4th vertex  
                      0.5, 0.5, // 5th vertex  
                      0.0, 0.5, // 6th vertex  
};  
GLuint positionsBuffer;  
glCreateBuffers(1, &positionsBuffer);  
// .... Other code here...  
  
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Indexed data

```
GLuint positionAttribIndex = 0;  
float positions[] = { 0.0, 0.0, // 1st vertex  
                      0.5, 0.0, // 2nd vertex  
                      0.5, 0.5, // 3rd vertex  
                      0.0, 0.5, // 4th vertex  
};  
GLuint positionsBuffer;  
glCreateBuffers(1, &positionsBuffer);  
// .... Other code here...  
GLuint indices[] = { 0, 1, 2, 0, 2, 3 };  
GLuint indexBuffer;  
glCreateBuffers(1, &indexBuffer);  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) * 6, indices, GL_STATIC_DRAW);  
  
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, NULL);
```

The table of indices

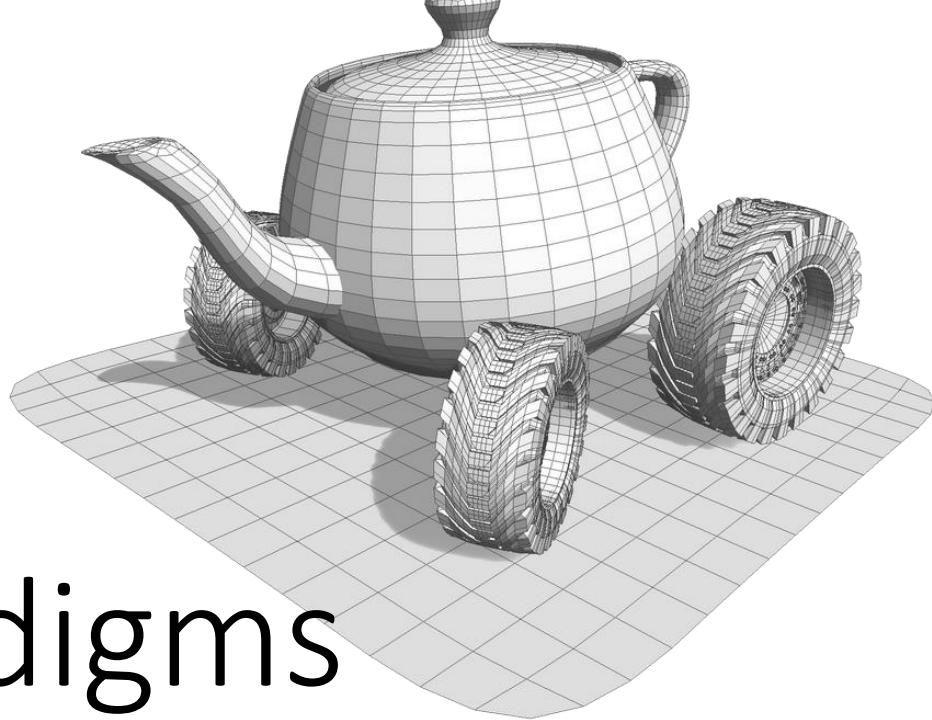
Number of indices



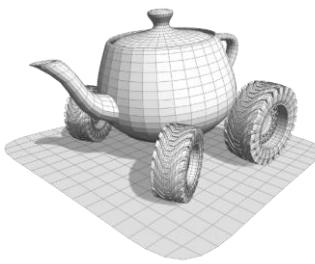
Exercises

1. Change the code of `code_1_my_first_triangle/main.cpp` so that only one `ARRAY_BUFFER` is used (that is, only one `gl.bindBuffer` and only one `gl.bufferData`)

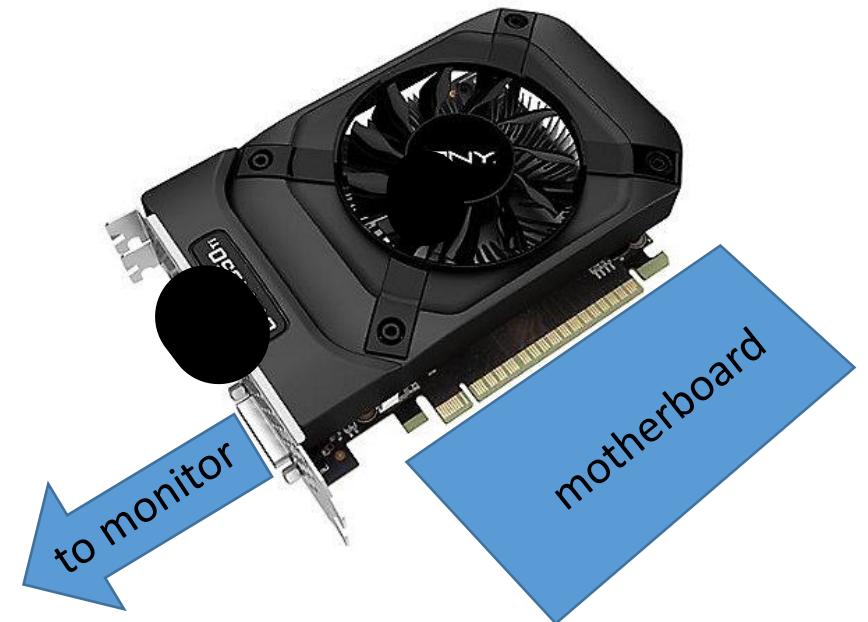
Rendering Paradigms



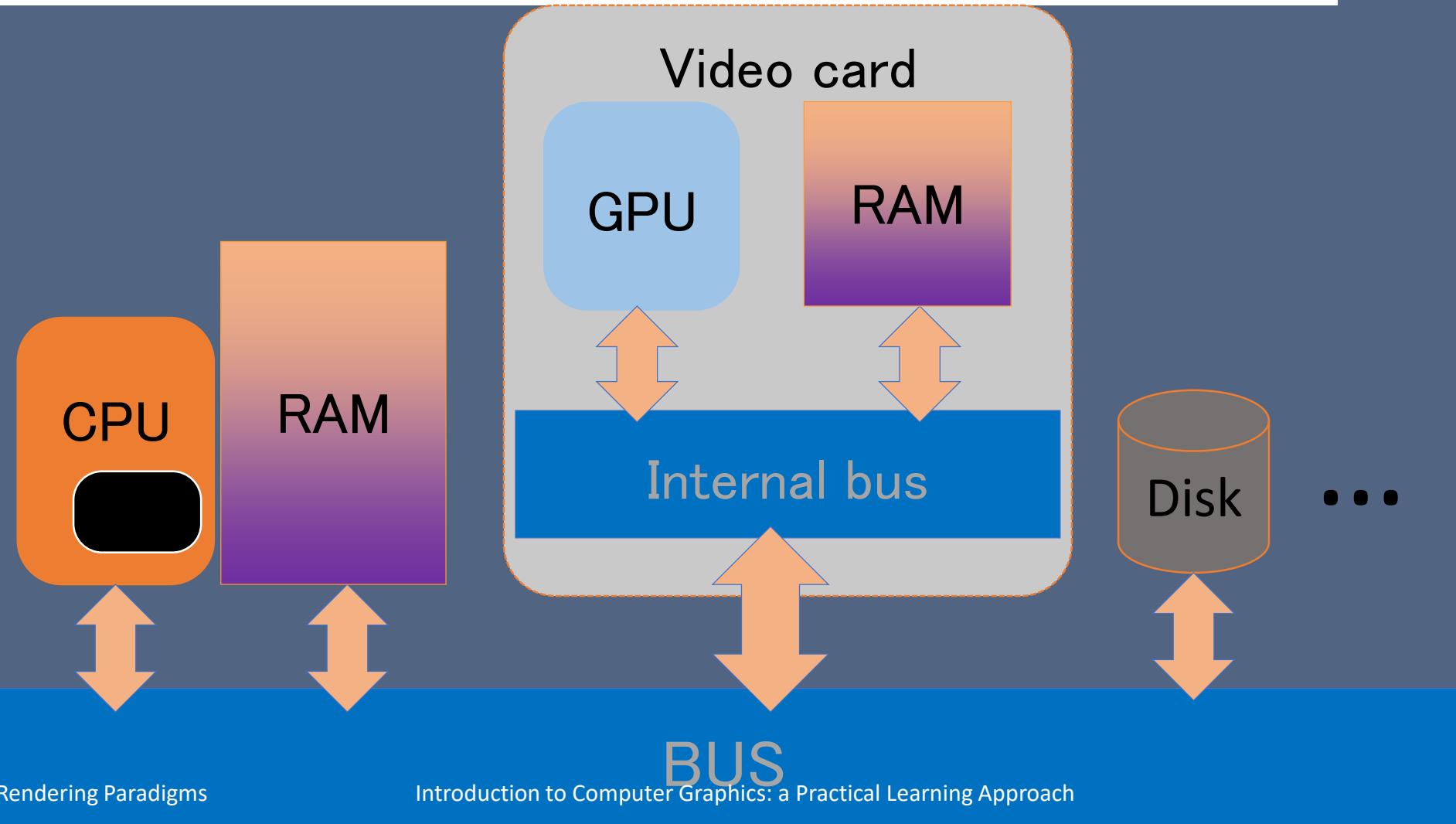
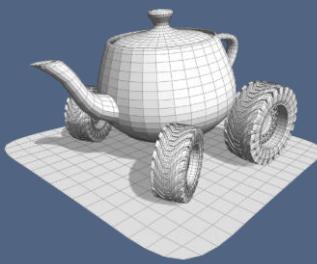
Specialized Hardware for rendering



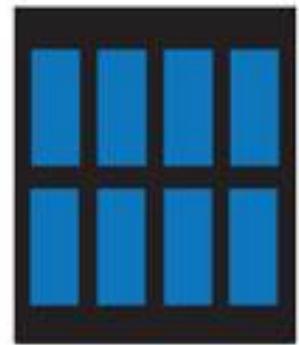
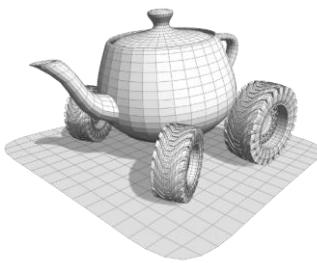
- GPU:
 - Graphics Processing Unit
 - Is what CPU is to the motherboard
 - *Specialized Instruction Set*
- Pipelined architecture
- SIMD computation model
 - Because CG is mostly highly parallel
- Has its own RAM memory level
 - "RAM CPU" vs "RAM GPU"



A over-simplified scheme

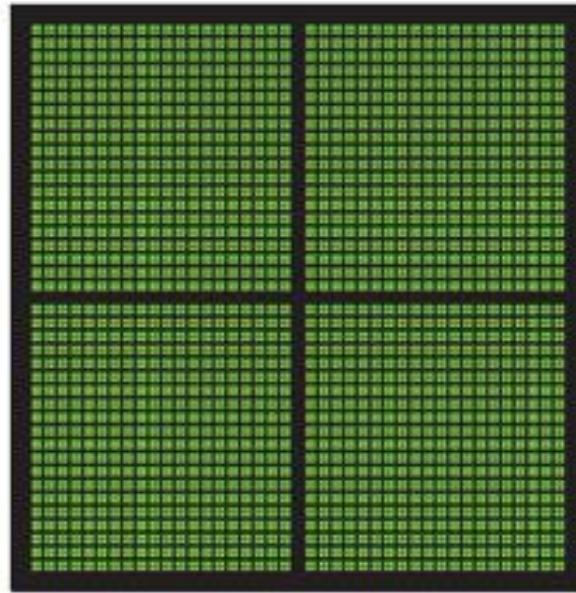


CPU vs GPU

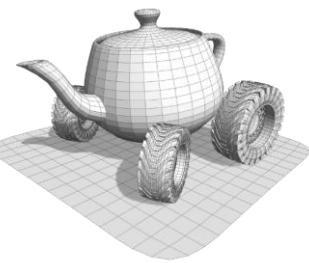


CPU
MULTIPLE CORES

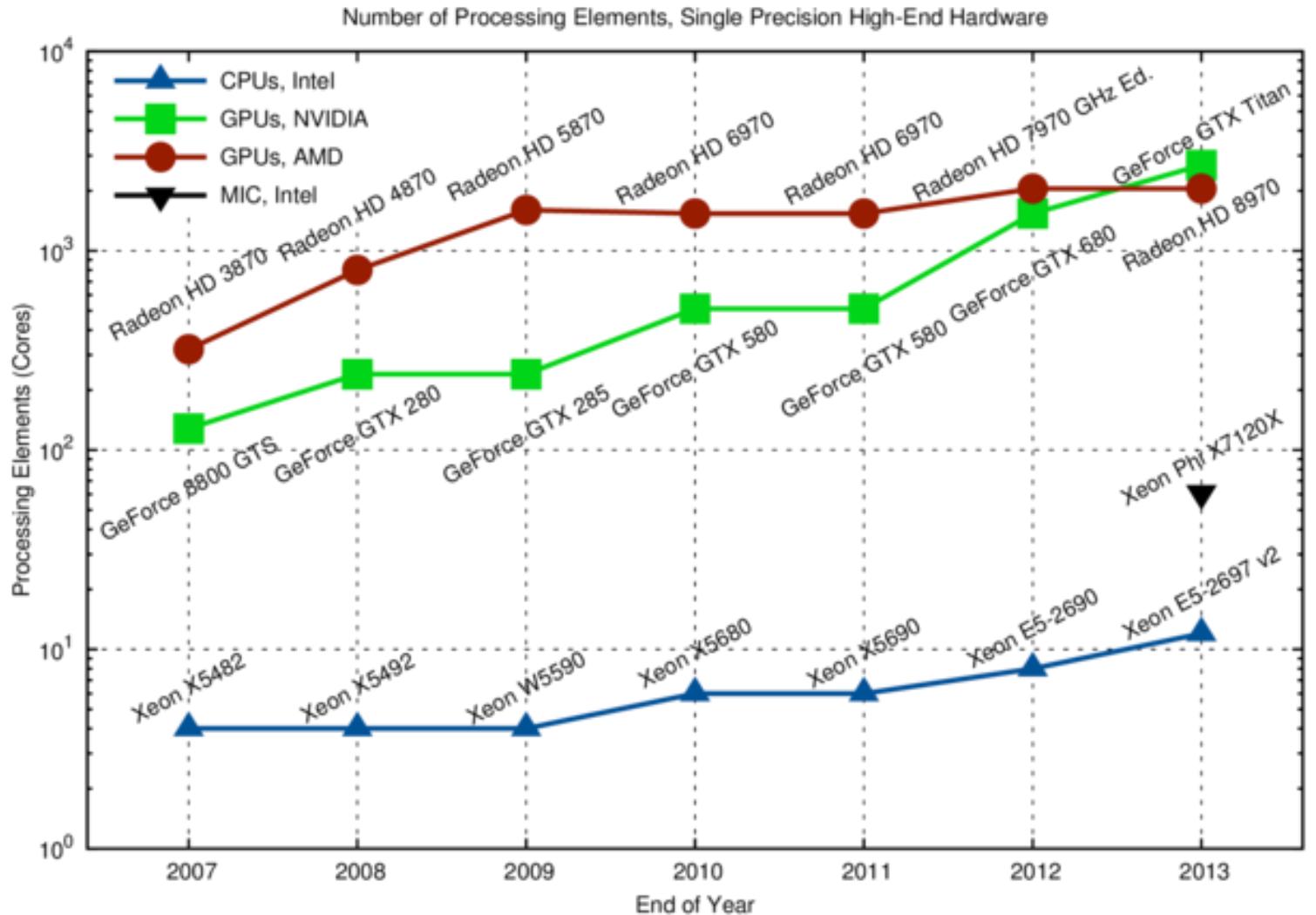
VS



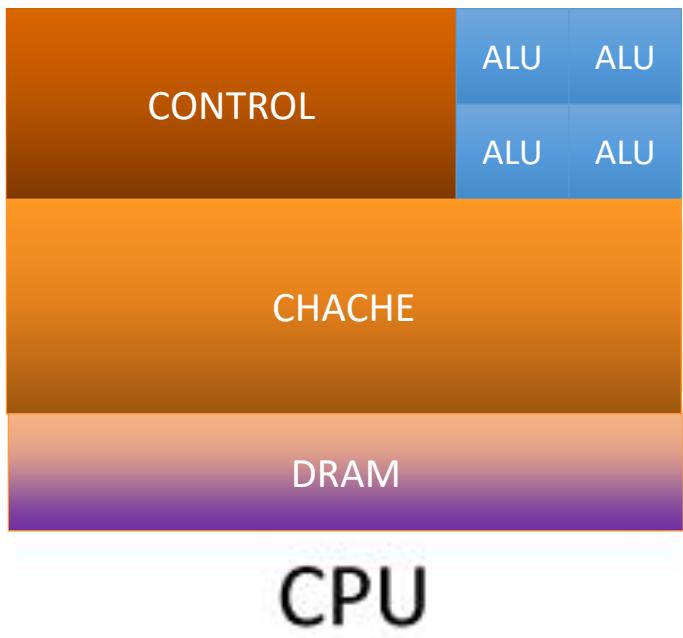
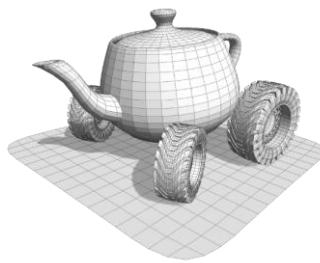
GPU
THOUSANDS OF CORES



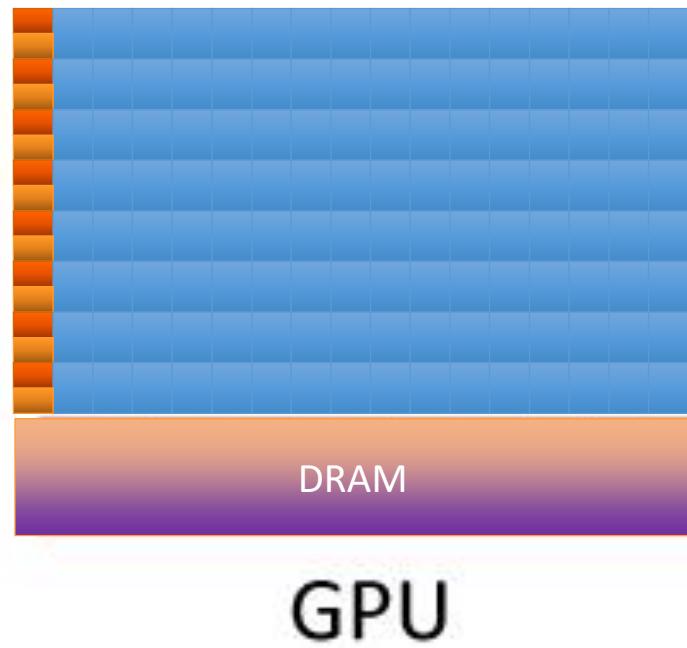
CPU vs GPU: how many proc. elements



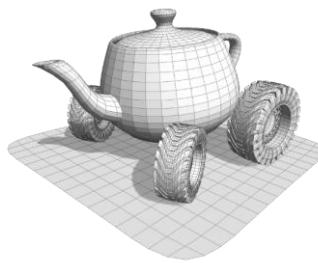
CPU vs GPU



VS



CPU vs GPU



- Transistors:
 $>80\%$
control + cache

→ flexibility

CPU

VS

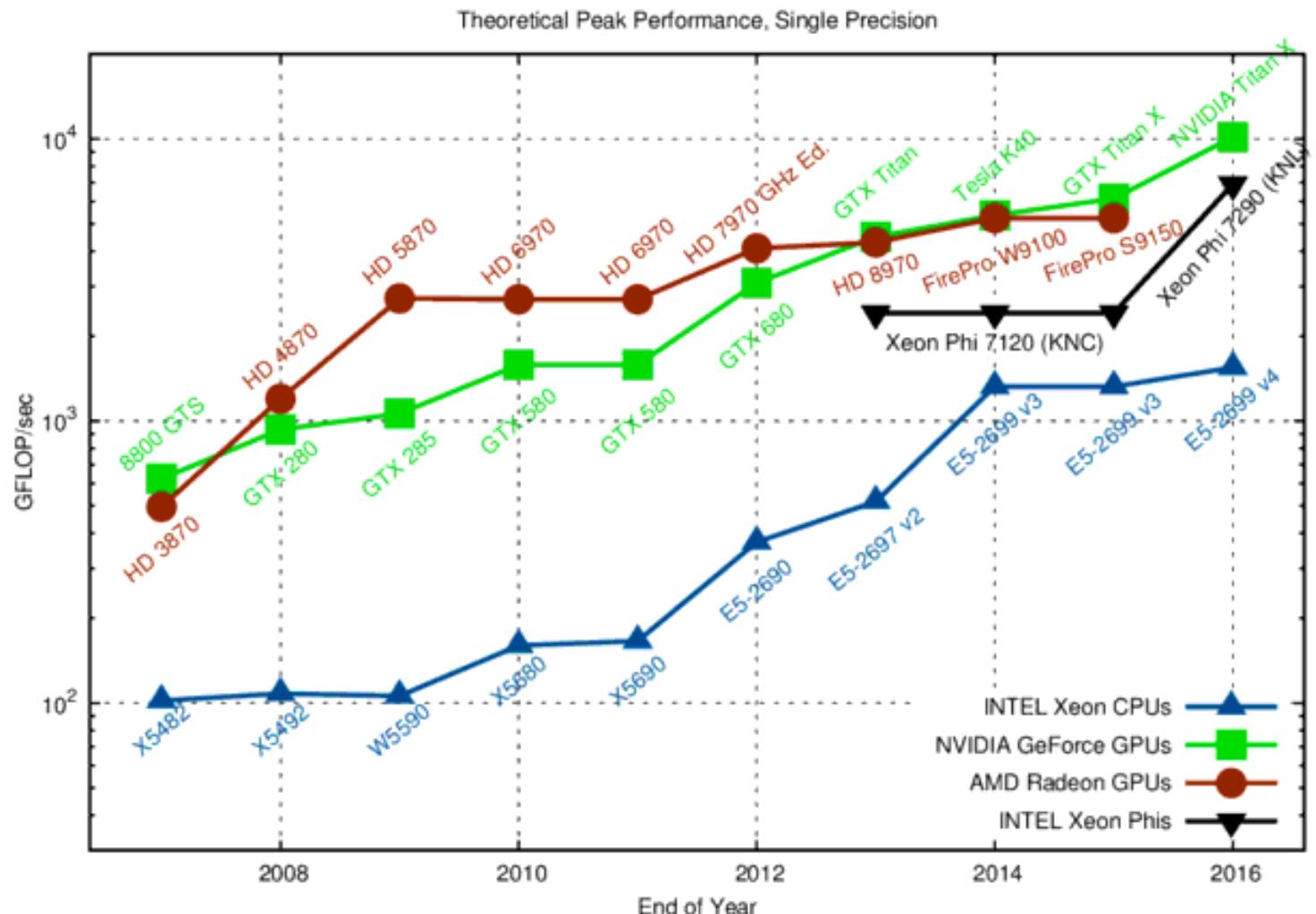
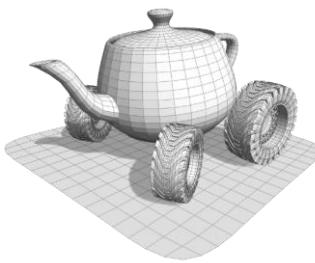
- Transistors:
 $\sim 80\%$ ALU

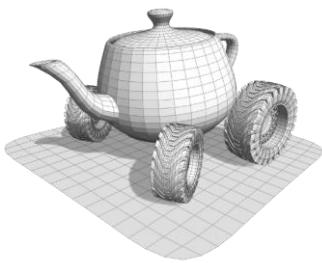
→ power (FLOPS)

GPU

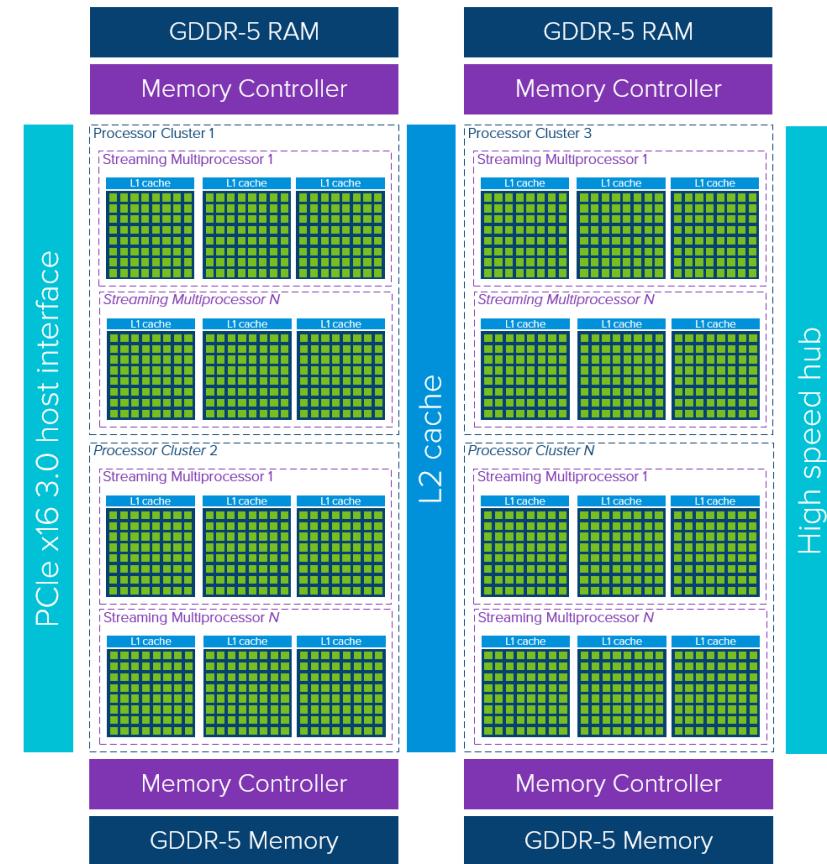
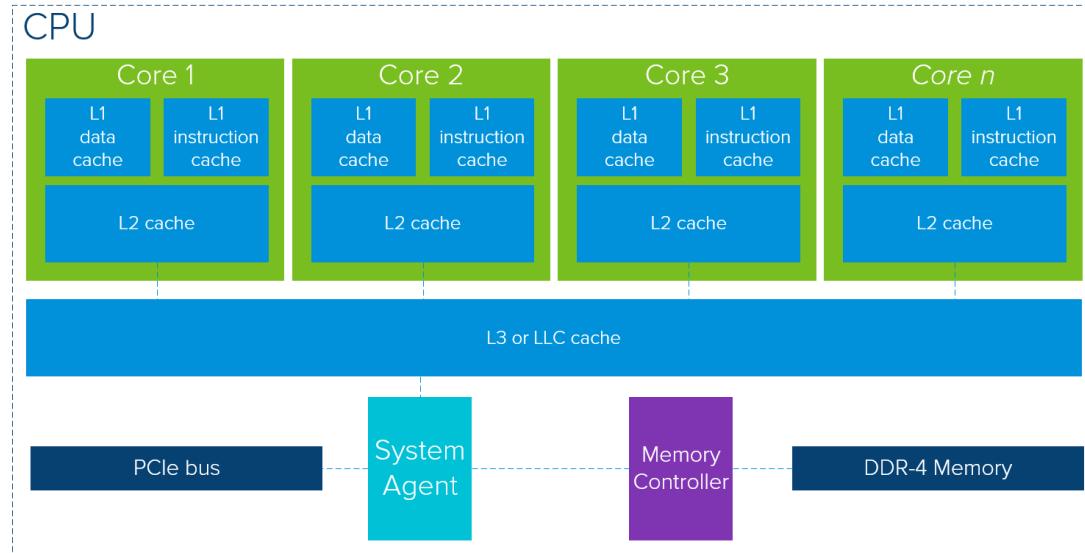
1.4 G transistors → 1 TeraFLOP

CPU vs GPU

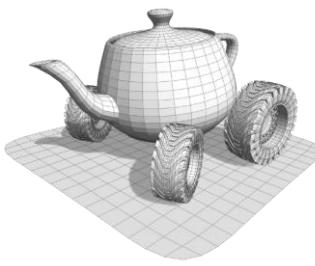




CPU vs GPU

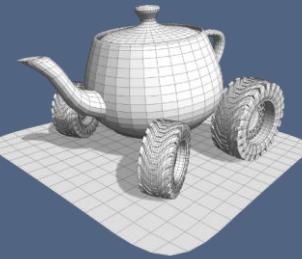


Specialized Hardware for Rendering

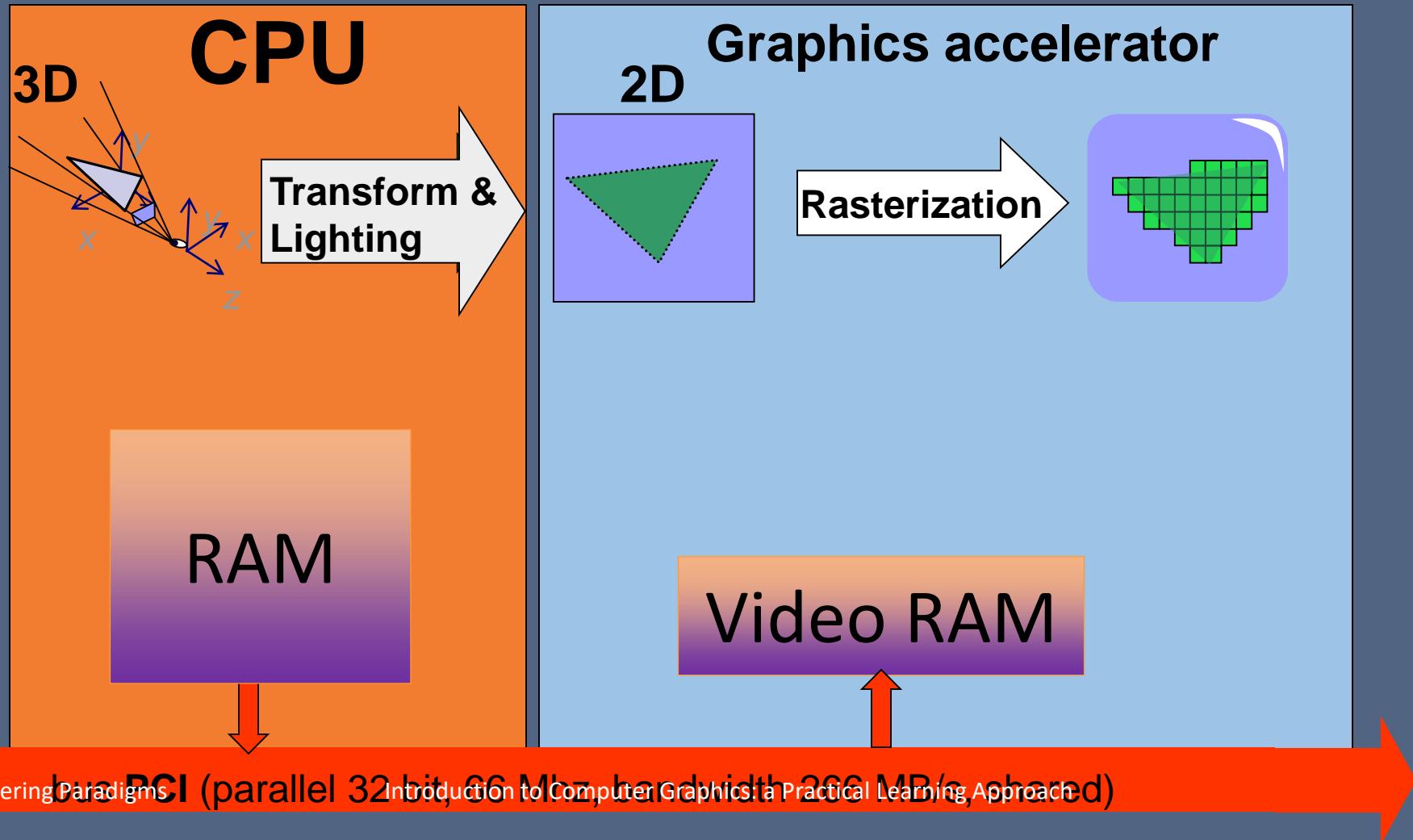


- Pros: efficiency
 - Specialized instruction set
 - hard-wired most common operations
 - Parallel computing, that is:
 1. between CPU e GPU
 - GPU does rendering
 - CPU+RAM free to do other things while GPU renders
 2. between multiple GPUs on the same bus (not very common)
 3. Among the stages of the rendering pipeline
 4. Within each stage of the pipeline
 5. instruction level: atomic operation on 4 values
- Cons: rigidity
 - Harder to use it for general purpose computation (not so much anymore..)

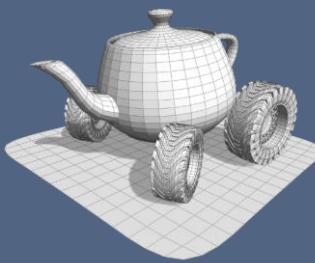
Some history: 1995-1997



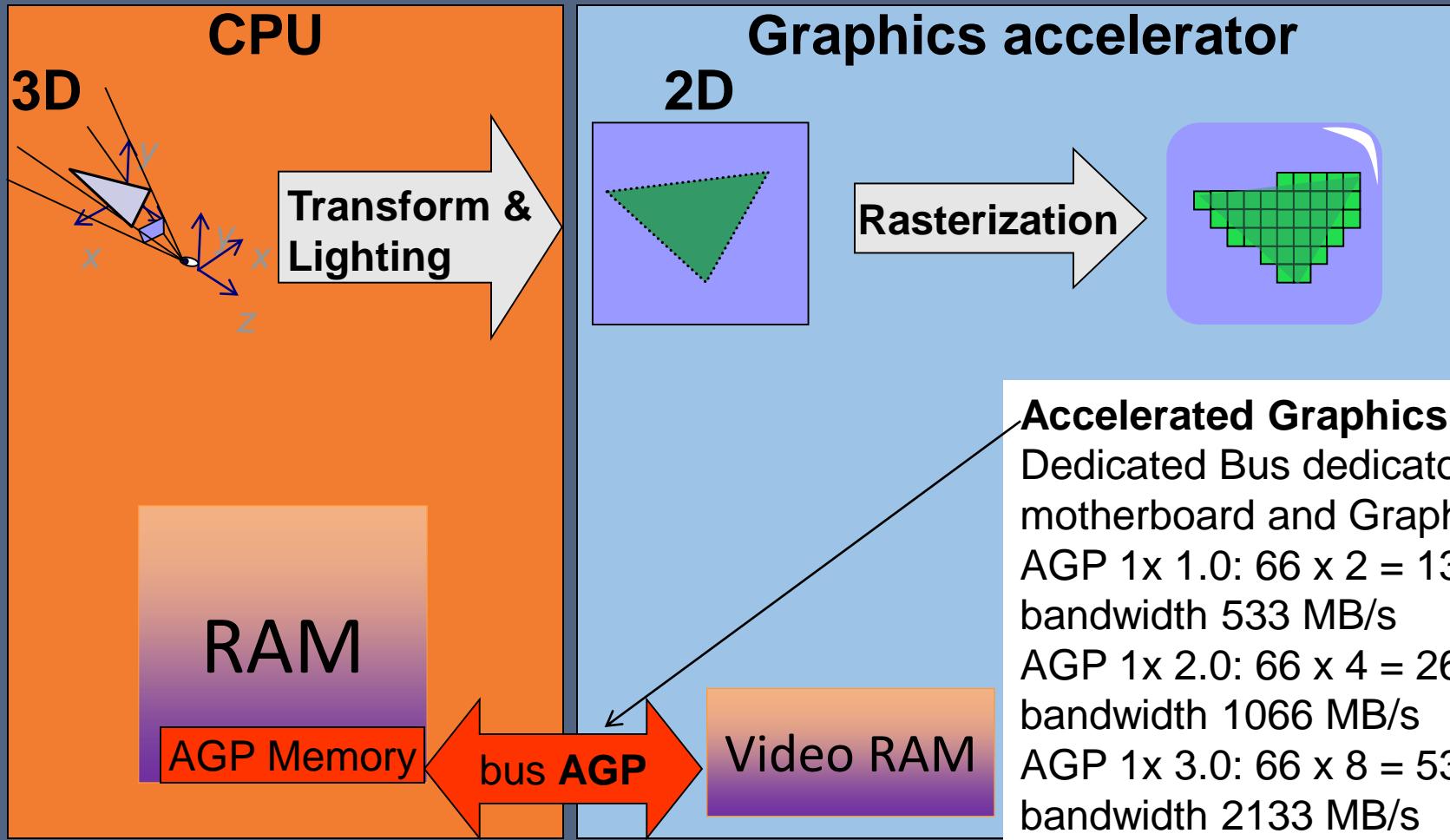
- 1995-1997:
3DFX Voodoo



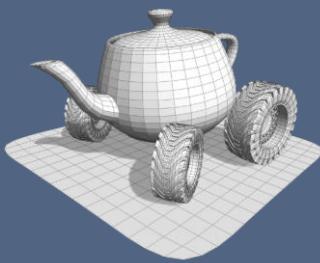
Some history: 1998



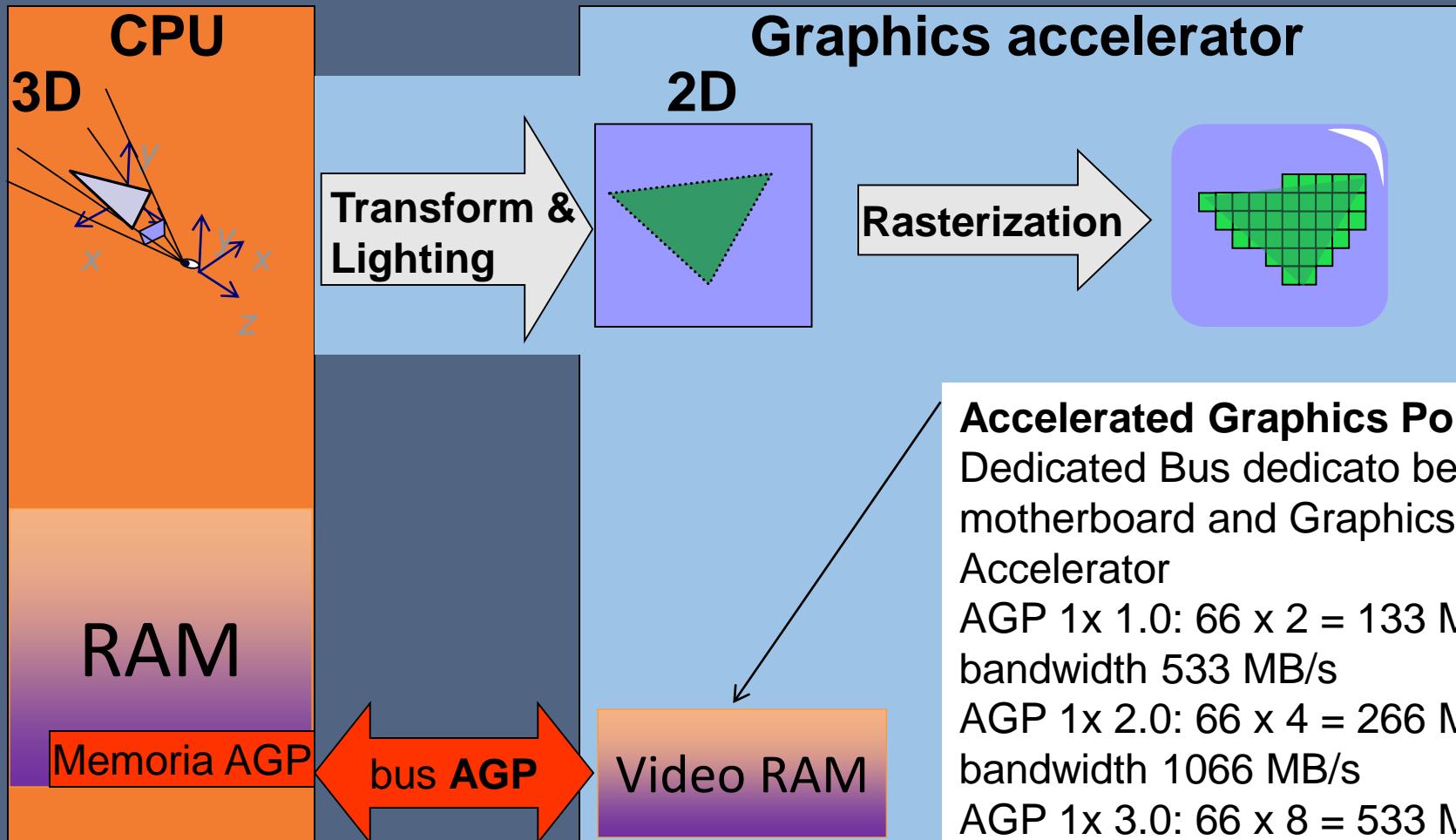
- 1995-1997:
3DFX Voodoo
- 1998: NVidia
TNT, ATI Rage



Some history: 1998



- 1995-1997:
3DFX Voodoo
- 1998: NVidia
TNT, ATI Rage



Accelerated Graphics Port

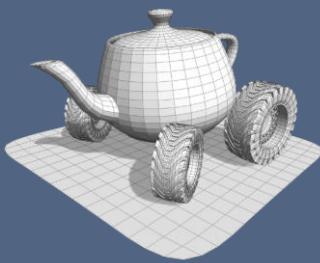
Dedicated Bus dedicated between motherboard and Graphics Accelerator

AGP 1x 1.0: $66 \times 2 = 133$ MHz,
bandwidth 533 MB/s

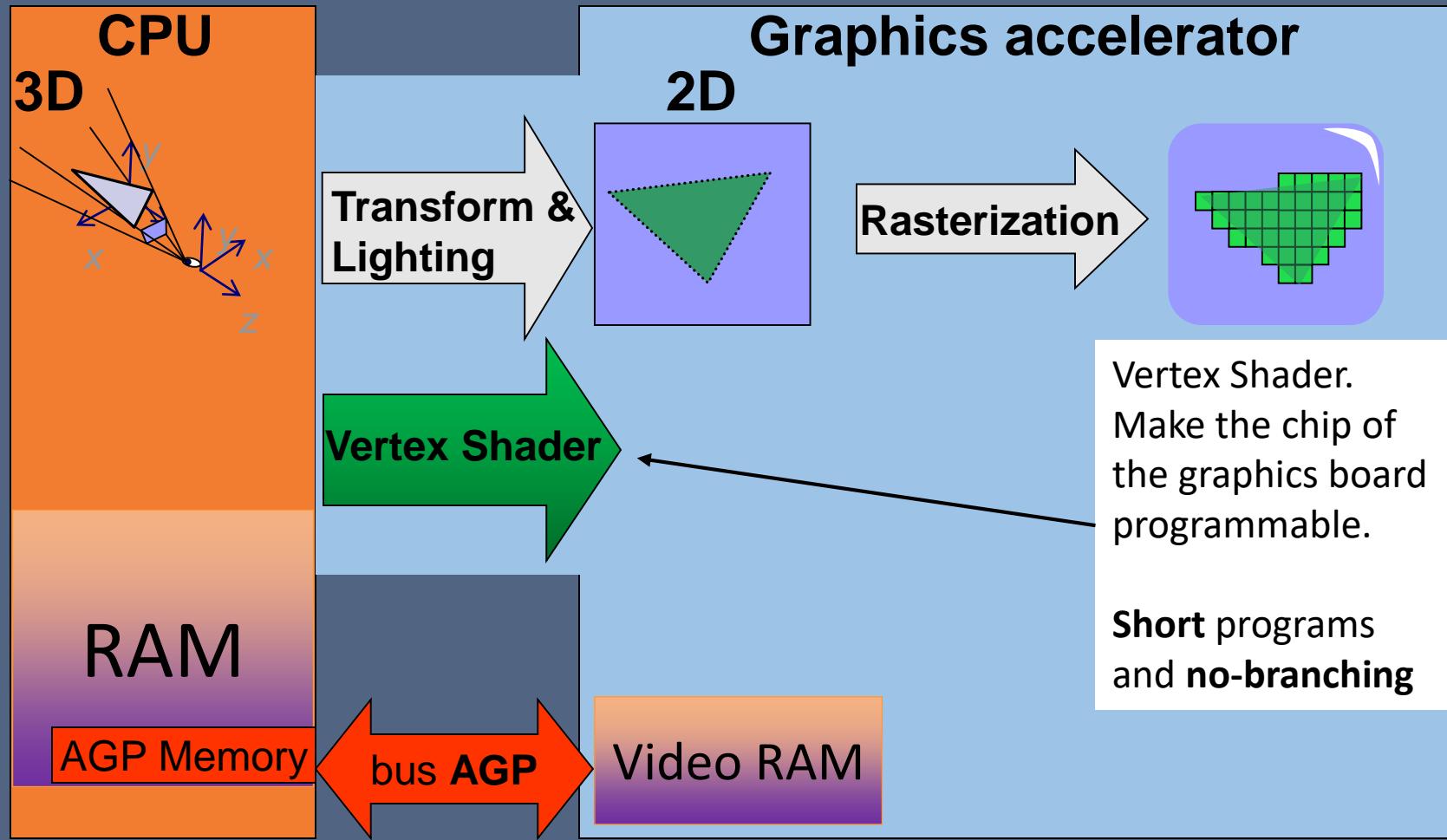
AGP 1x 2.0: $66 \times 4 = 266$ MHz,
bandwidth 1066 MB/s

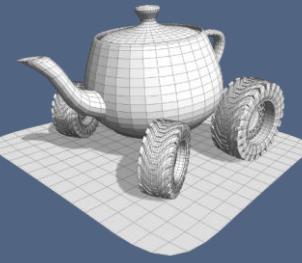
AGP 1x 3.0: $66 \times 8 = 533$ MHz,
bandwidth 2133 MB/s

Some history: 2001



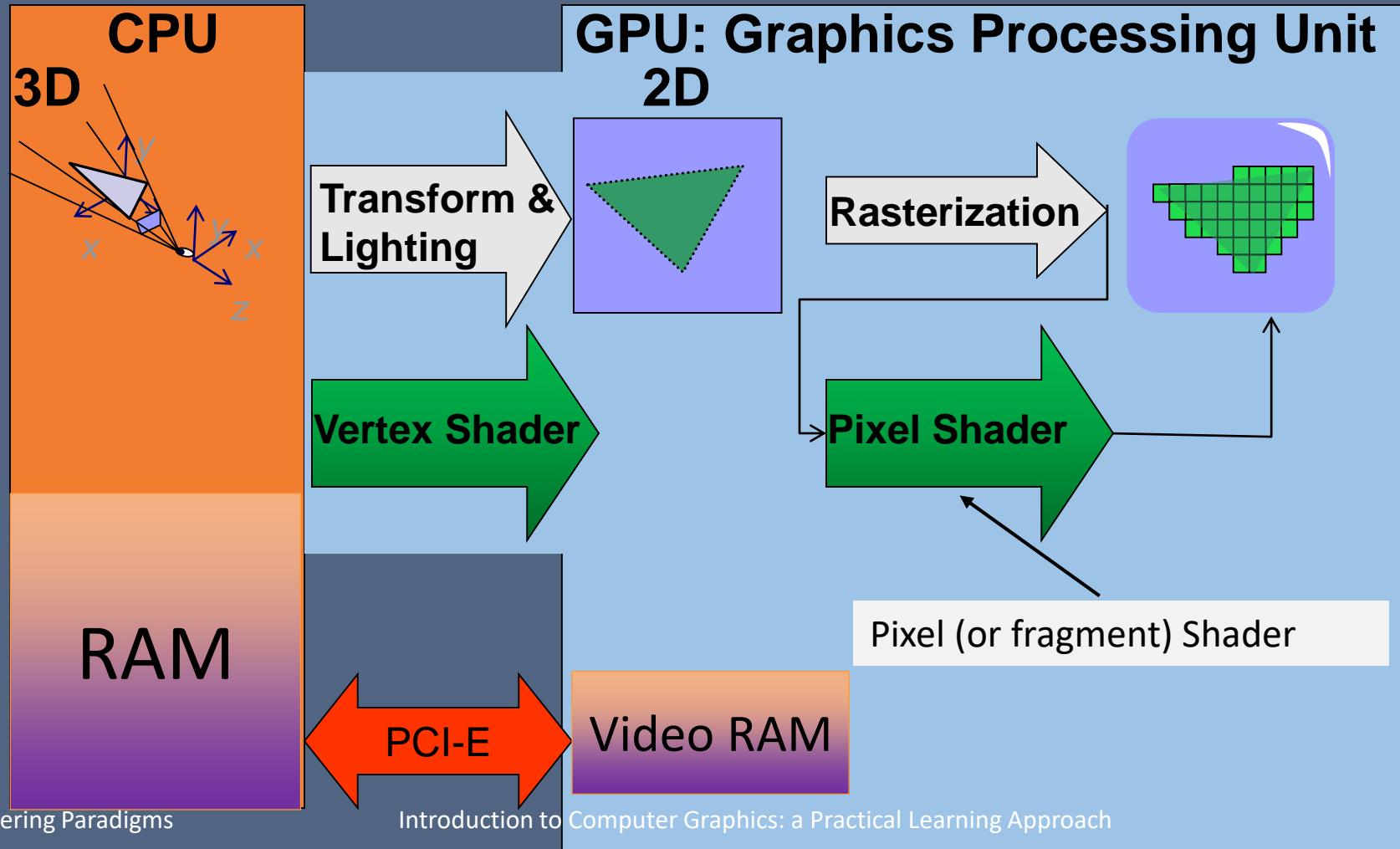
- 1995-1997:
3DFX Voodoo
- 1998: NVidia
TNT, ATI Rage
- 2001: GeForce
3, ATI Radeon
8500



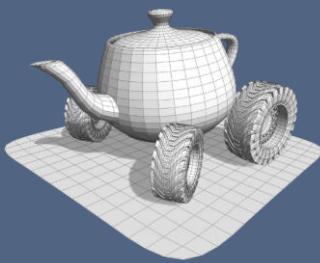


Some history: 2002-2003

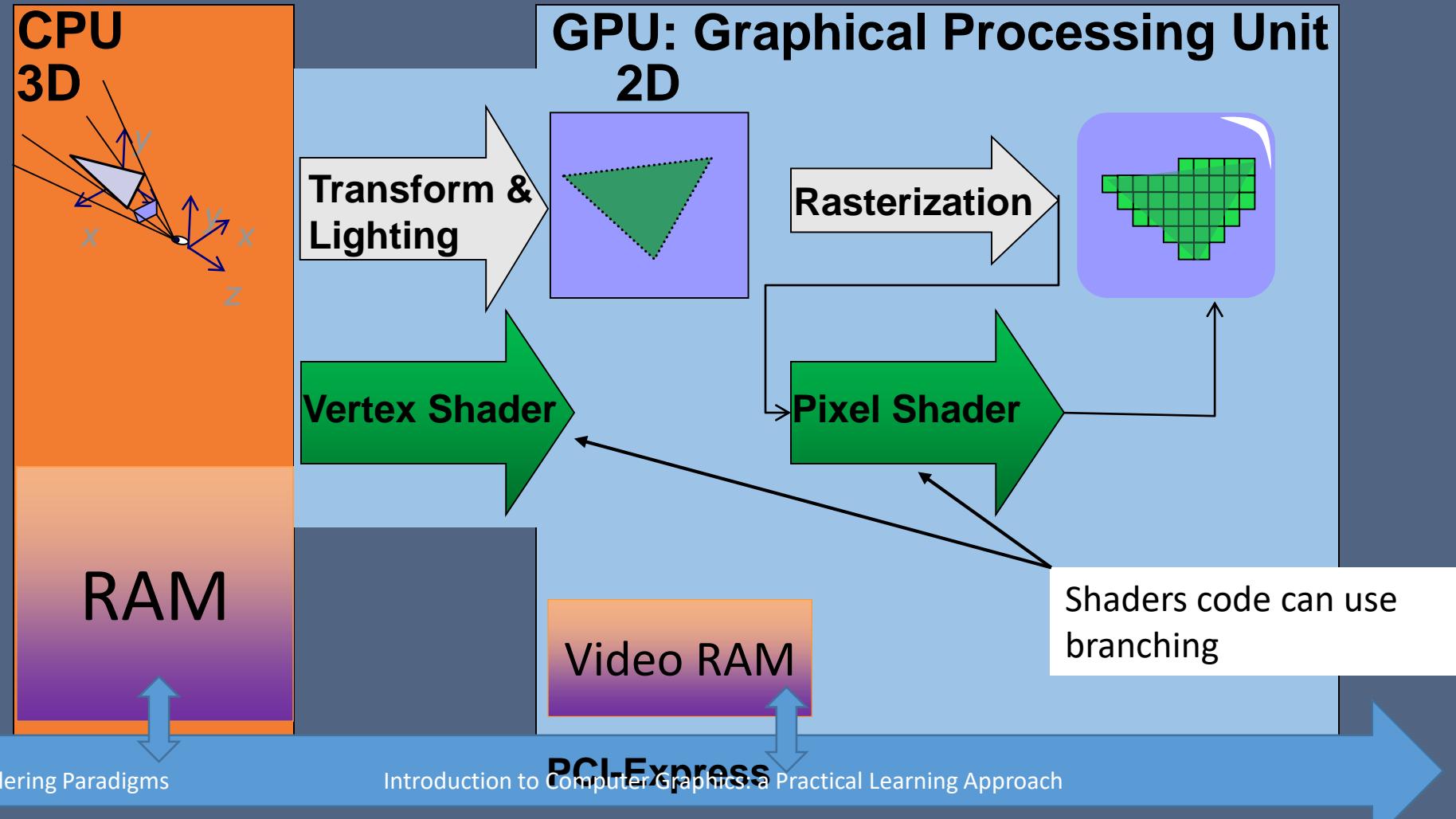
- 1995-1997:
3DFX Voodoo
- 1998: NVidia
TNT, ATI Rage
- 2001: GeForce
3, ATI Radeon
8500
- 2002-3:
GeForceFX,
Radeon 8500



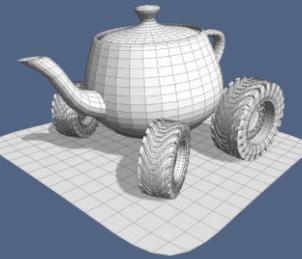
Some history: 2004-2005



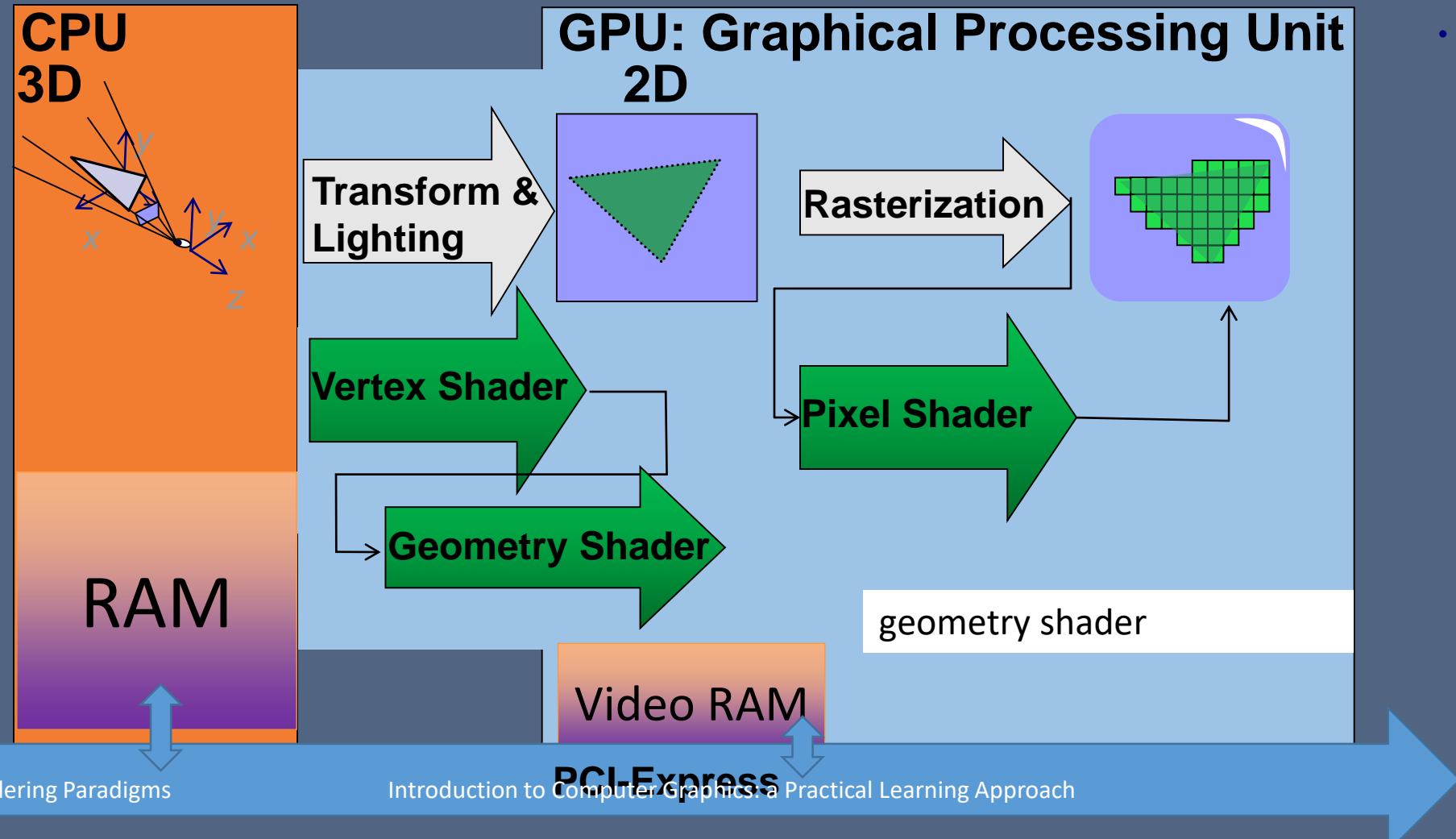
- 1995-1997:
3DFX Voodoo
- 1998: NVidia
TNT, ATI Rage
- 2001: GeForce
3, ATI Radeon
8500
- 2002-3:
GeForceFX,
Radeon 8500
- GeForce 6800-
7800, ATI
Radeon 9200
- 2004-5
GeForce 6800-
7800, ATI
Radeon 9200



Some history: 2007

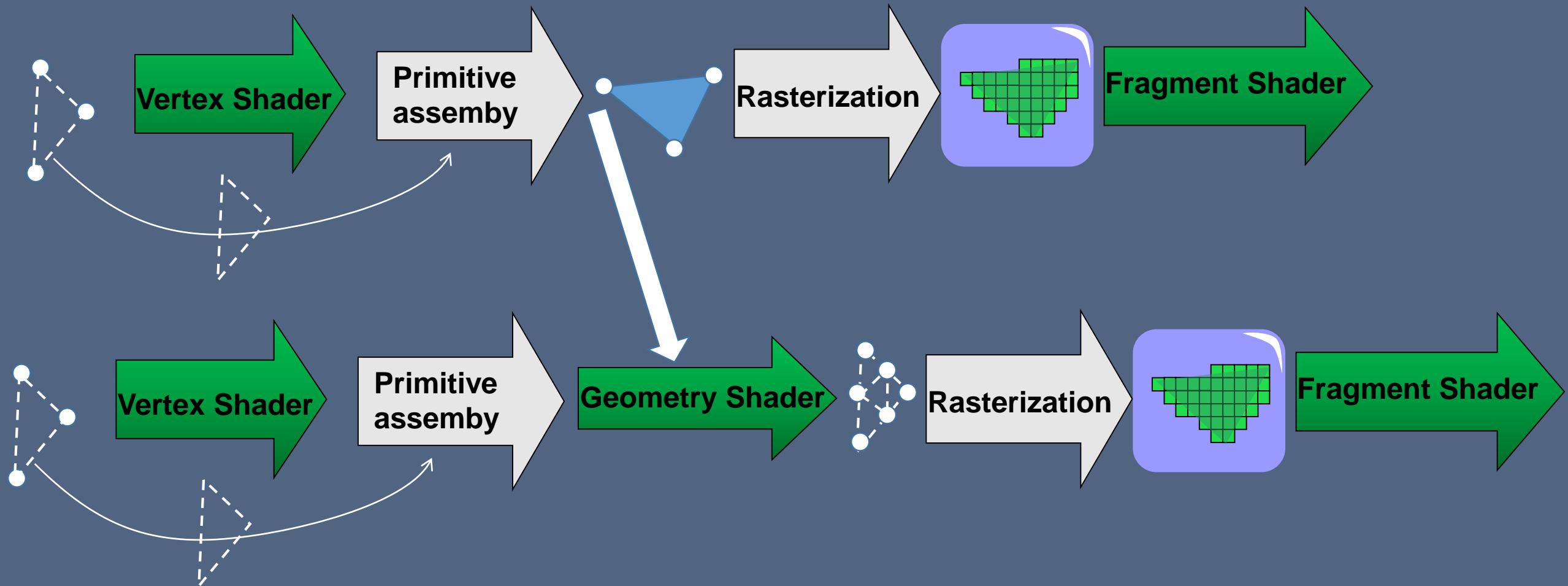
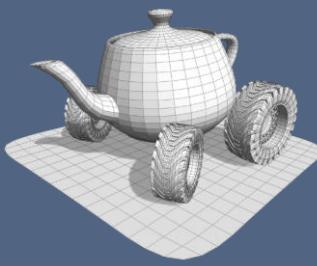


- 1995-1997: 3DFX Voodoo
- 1998: NVidia TNT, ATI Rage
- 2001: GeForce 3, ATI Radeon 8500
- 2002-3: GeForceFX, Radeon 8500
- GeForce 6800-7800, ATI Radeon 9200
- 2004-5: GeForce 6800-7800, ATI Radeon

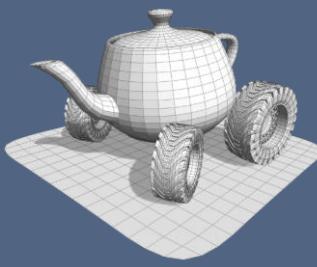


- 2007: NVidia 8800

Geometry Shader

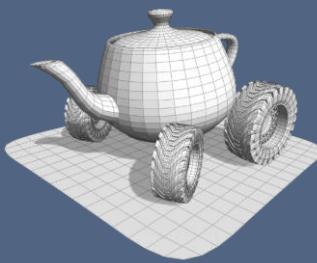


Compute Shaders

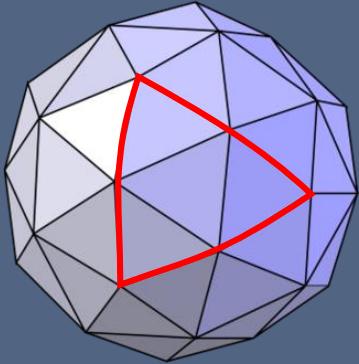
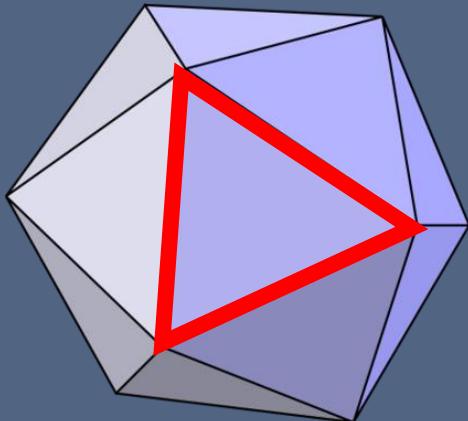


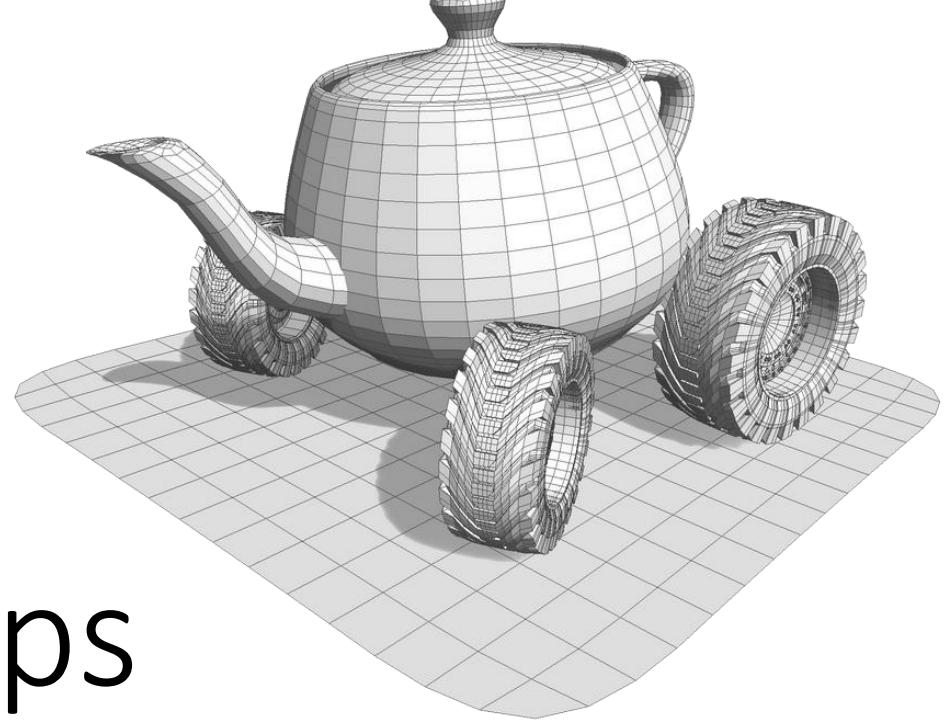
- With all this flexibility, people started to program the shaders to do generic, not necessarily CG, computation
 - General Purpose GPU (GPGPU)
- Since OpenGL 4.3 (2012) **Compute Shaders** have been introduced.
- Compute Shaders computation model
 - Partition the program execution in a *3 dimensional grid*, each cell of the grid with a predefined number of executions
 - Run the grid in parallel
 - Access Video Memory as we do with RAM

Another shader we won't use right now



- Tessellation Shaders
 - Same job you can do with a Geometry Shader but with hardware *dedicated* support
 - For example Subdivision Surfaces (see Geometry Processing course)

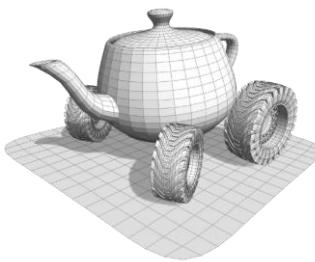




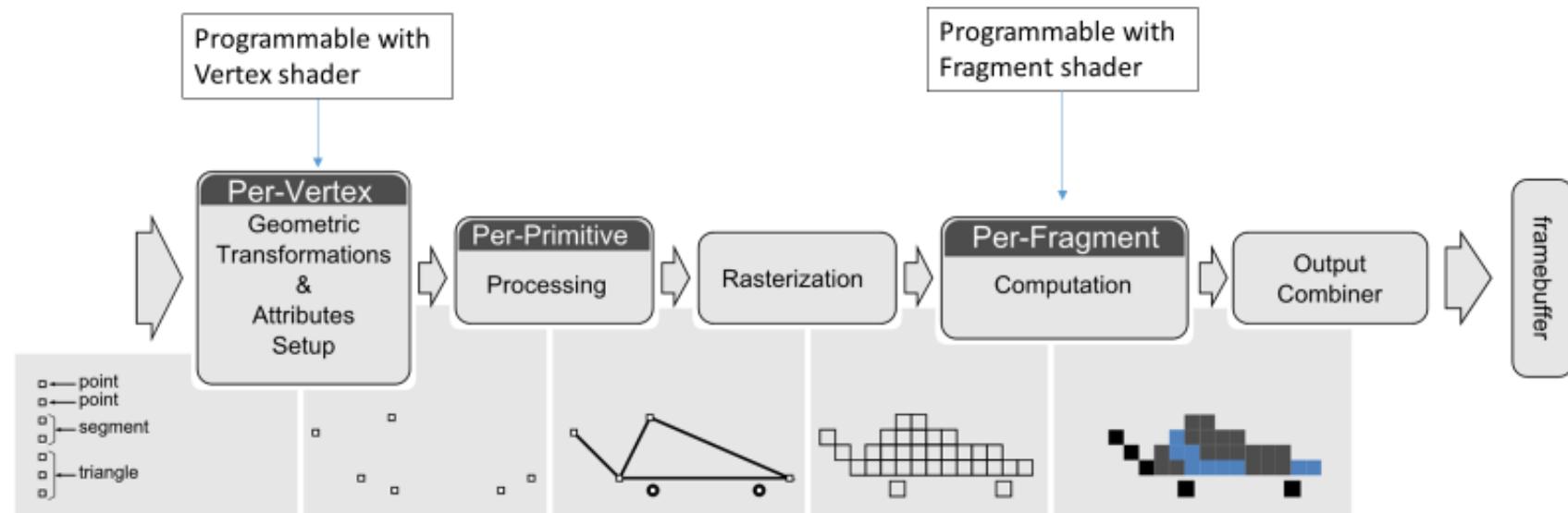
The First Steps

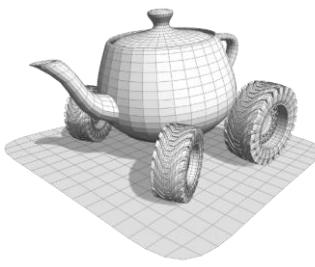
Let's draw a triangle!

Recall the pipeline..



Rasterization-based pipeline (oversimplified)





Shader Program: the Vertex Shader

- GLSL: a c++ - like language to program the pipeline
- A GLSL program is compiled and linked and then runs on the GPU
- Vertex-shader
 - Input: set of attributes per vertex
 - Output: set of attributes per vertex

gl_* are names of built-in variables

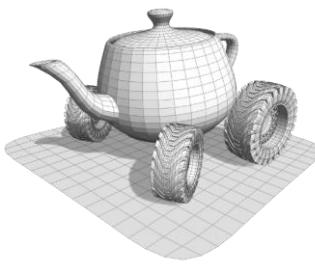
```
/* create a vertex shader */
std::string vertex_shader_src = "\n    in vec2 aPosition;\n    void main(void)\n    {\n        gl_Position = vec4(aPosition, 0.0, 1.0);\n    }\n";
const GLchar* vs_source = (const GLchar*)vertex_shader_src.c_str();
```

GLSL code

```
GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
```

```
glShaderSource(vertex_shader, 1, &vs_source, NULL);
```

```
glCompileShader(vertex_shader);
```

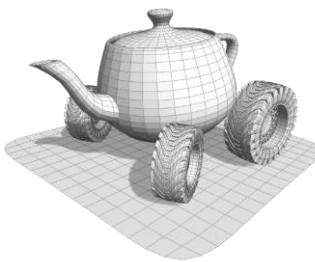


Shader Program: the Fragment Shader

- GLSL: a c++ - like language to program the pipeline
- A GLSL program is compiled and linked and then runs on the GPU
- Fragment-shader
 - Input: set of attributes per fragment
 - Output: set of attributes per fragment

```
/* create a fragment shader */  
std::string fragment_shader_src = "\\\n    out vec4 color;\n    void main(void)\\n    {\n        color = vec4(0.0, 0.0, 1.0, 1.0);\\n    }";  
const GLchar* fs_source = (const GLchar*)fragment_shader_src.c_str();  
GLuint fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);  
glShaderSource(fragment_shader, 1, &fs_source, NULL);  
glCompileShader(fragment_shader);
```

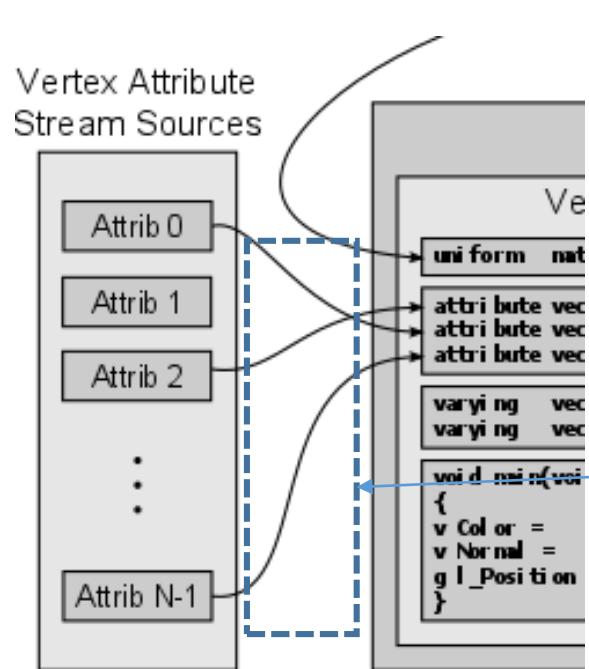
GLSL code



Shader Program

Tells OpenGL to create a **program** object

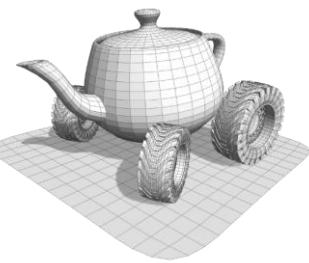
Attaches vertex and fragment shaders to the program



```
GLuint program_shader = glCreateProgram();  
glAttachShader(program_shader, vertex_shader);  
glAttachShader(program_shader, fragment_shader);  
glBindAttribLocation(program, positionAttribIndex,"aPosition");  
glLinkProgram(program_shader);
```

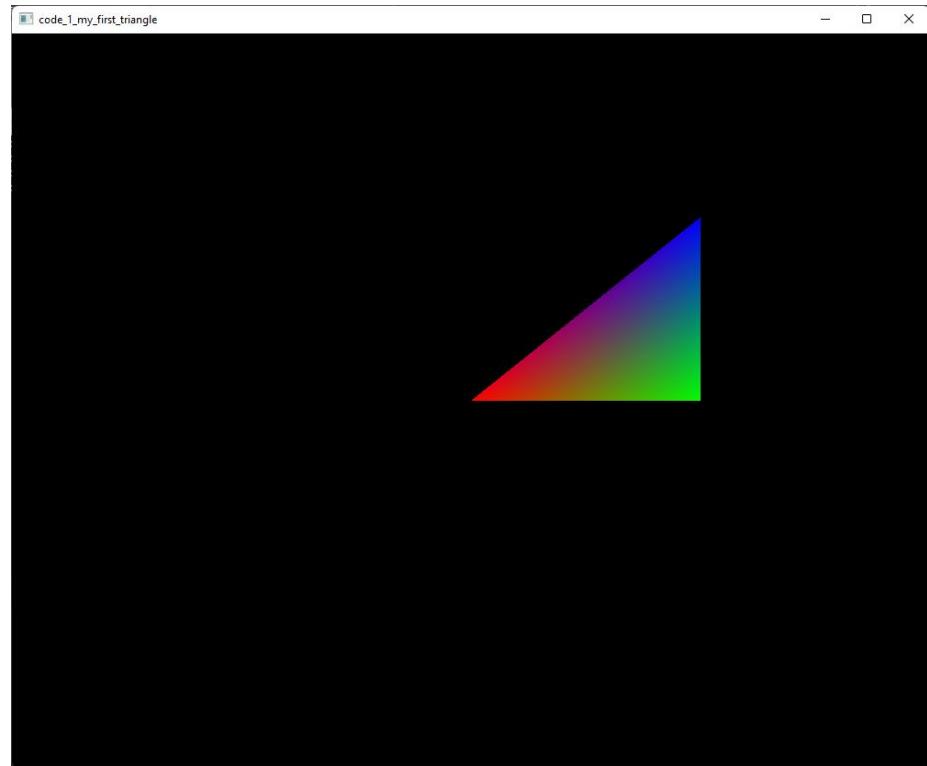
Link the program

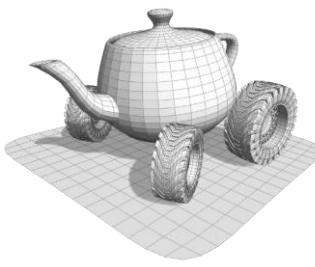
Connects the attribute in the vertex shader to the slot enabled with
 `glEnableVertexAttribArray(positionAttribIndex)`



Our Second Triangle: Add Color

- Same as before, but we want to also assign a color to each vertex and *interpolate* inside the triangle





Data Flow

Qualifier for input values

```
var vsSource = `  
    in vec2 aPosition;  
    in vec3 aColor;
```

One vertex-shader runs for each vertex

```
        out vec3 vColor;  
  
    void main(void)  
    {  
        vColor = aColor;  
        gl_Position = vec4(aPosition, 0.0, 1.0);  
    }  
`;
```

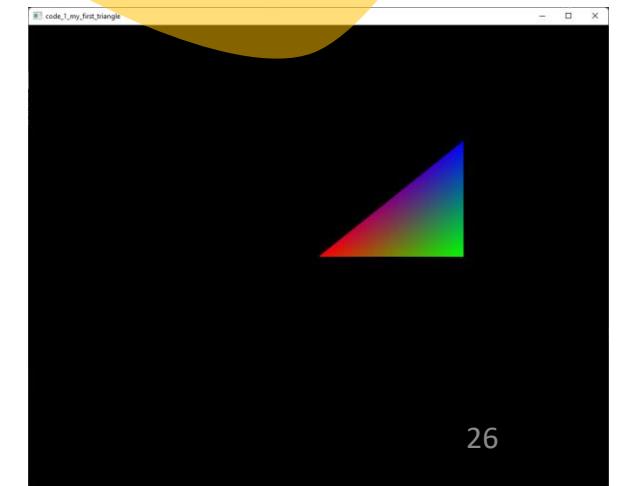
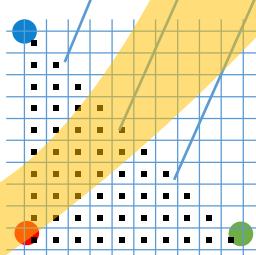
```
var fsSource = `  
    out vec4 color;  
    in vec3 vColor;  
  
    void main(void)  
    {  
        color = vec4(vColor, 1.0);  
    }  
`;
```

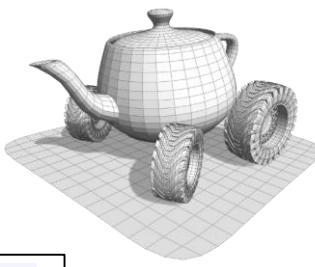
One fragment-shader runs for each fragment

vColor is interpolated

Primitive assembler

Rasterizer





Shader: Uniform variables

- Both Vertex and Fragment shaders may declare **uniform variables**
- A uniform variable is a *read-only* variable that is set from the client program and holds the same value for all vertices (or fragments)

Ex: translate the vertex position by uDelta

```
var vsSource = `  
in vec2 aPosition;  
in vec3 aColor;  
uniform vec3 uDelta;  
  
out vec3 vColor;  
  
void main(void)  
{  
    vColor = aColor;  
    gl_Position = vec4(aPosition+ uDelta, 0.0, 1.0);  
}  
`;
```

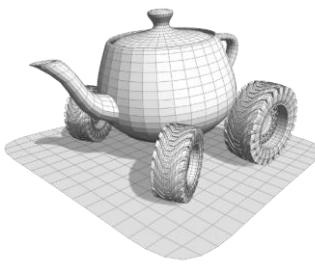
C++

```
GLint loc = glGetUniformLocation(program_shader,"uDelta");  
glUniform1f(loc,0.5);
```



Debugging tips

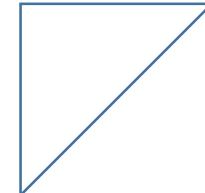
- Client-side (CPU)
 - Usual stuff: allow step-by-step, reads variables, call stack etc
- OpenGL-side
 - `glGetError()` returns a code reporting the last error flag raised by some call
 - `glGet*` are all functions querying the internal state of WebGL:
for example: `glGetProgramInfoLog(...)`;
 - In `src/common/debugging.h` you'll find a bunch of wrapping function ready to use
- GLSL side
 - Trickier: GLSL code is executed on the GPU and parallel. Not great tools for debugging
 - Learn to use the fragment shader output as log and other tricks we will learn along the way



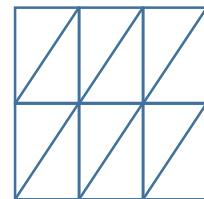
Exercises

1. In `code_3_box2D` implement the function `create_box2d(int xsize, int ysize)` that sets up the buffers for rendering a box as the ones shown below. Then make the call to draw it
 - The box goes from x coordinate -0.1 to 0.1 and from y coordinates -0.1 to 0.1
 - The «n» parameter tells by how many rectangles (that is, triangles pairs) the box is made of

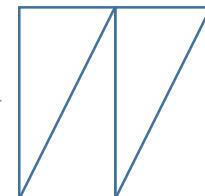
create_box2d(1,1)



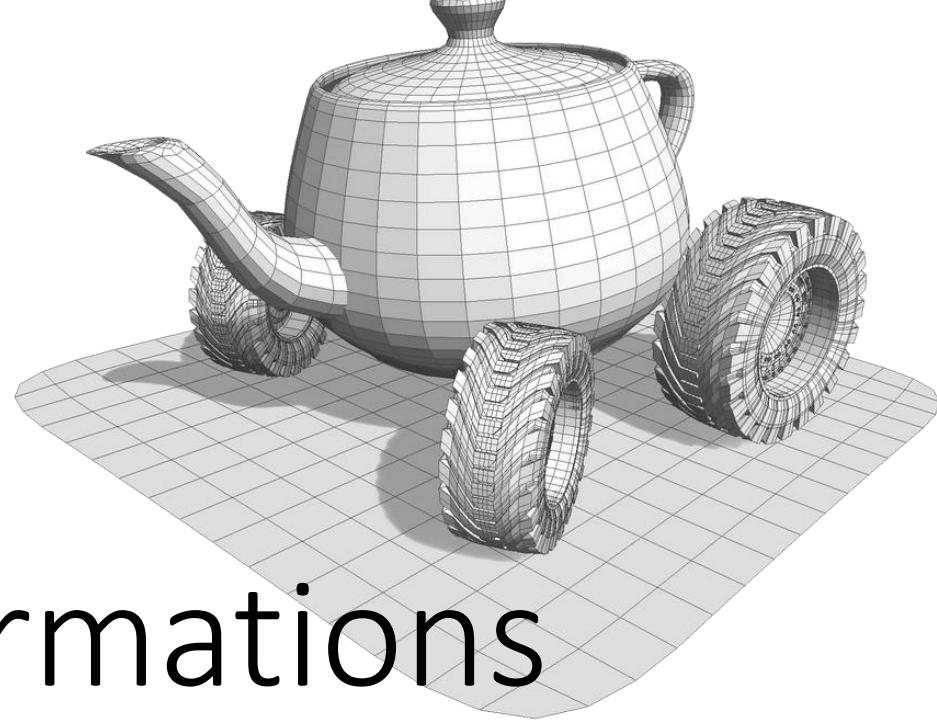
create_box2d(3,2)



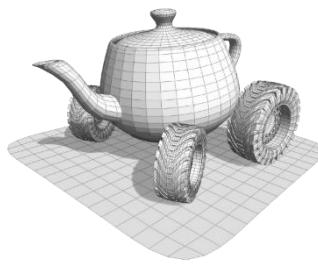
create_box2d(2,1)



Geometric Transformations



Geometric entities: points and vectors



Points: a point represents a **position** in space

We will use **bold** for points: $\mathbf{p}, \mathbf{q} \dots$

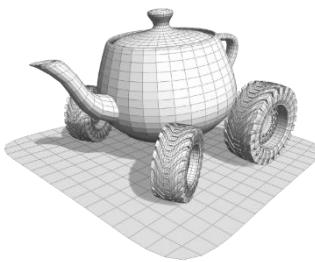


Vectors: a vector represents a **displacement** in space

- It has direction
- It has magnitude (*length*)
- It does **not** have a position

We will use ***italic bold*** for vectors:
 $\mathbf{\boldsymbol{v}}, \mathbf{\boldsymbol{u}} \dots$





Operations on points and vectors

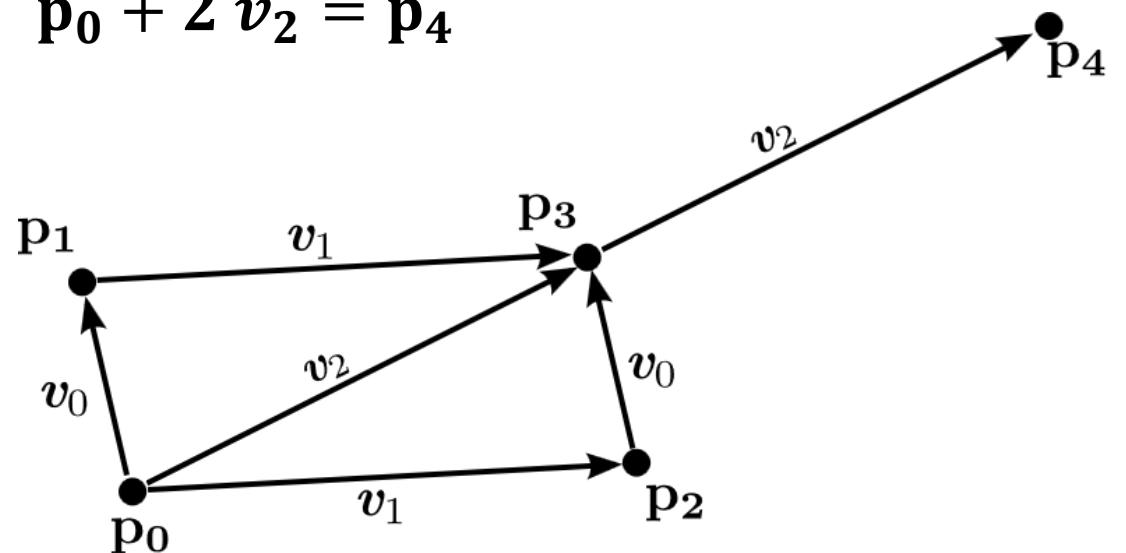
- point + vector returns a point
- point – point returns a vector
- vector + vector returns a vector
- scalar vector returns a vector

$$\mathbf{p}_0 + \mathbf{v}_0 = \mathbf{p}_1$$

$$\mathbf{p}_1 - \mathbf{p}_0 = \mathbf{v}_0$$

$$\mathbf{v}_0 + \mathbf{v}_1 = \mathbf{v}_1 + \mathbf{v}_0 = \mathbf{v}_2$$

$$\mathbf{p}_0 + 2 \mathbf{v}_2 = \mathbf{p}_4$$



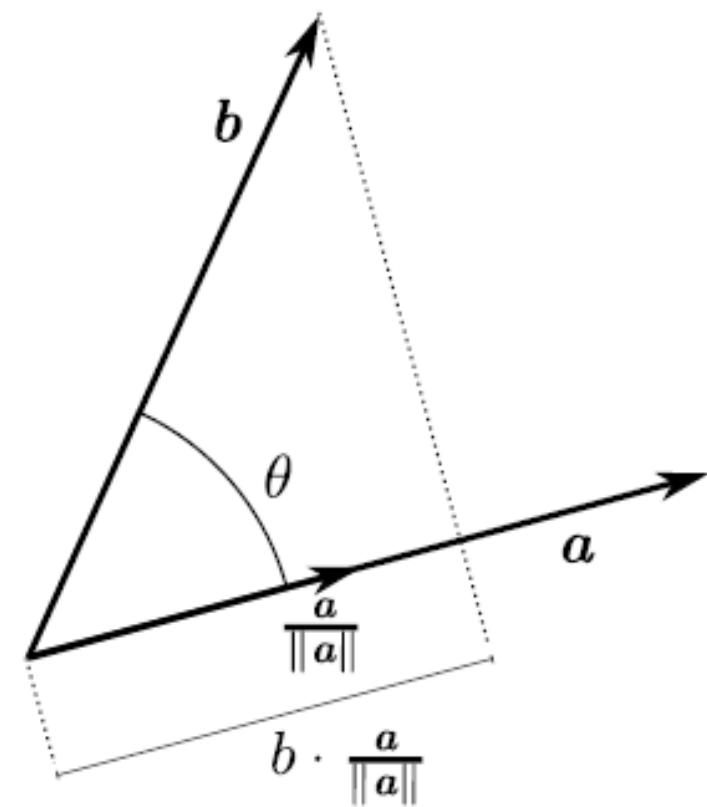


Dot product (1/3)

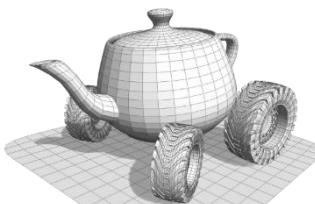
- The **dot** product between two vectors is a scalar defined as

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i + \cdots + a_n b_n$$

- It holds that: $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$ which gives a way to compute the angle between two vectors and the length of the respective orthogonal projections on one another



$$\theta = \arccos \left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \right)$$

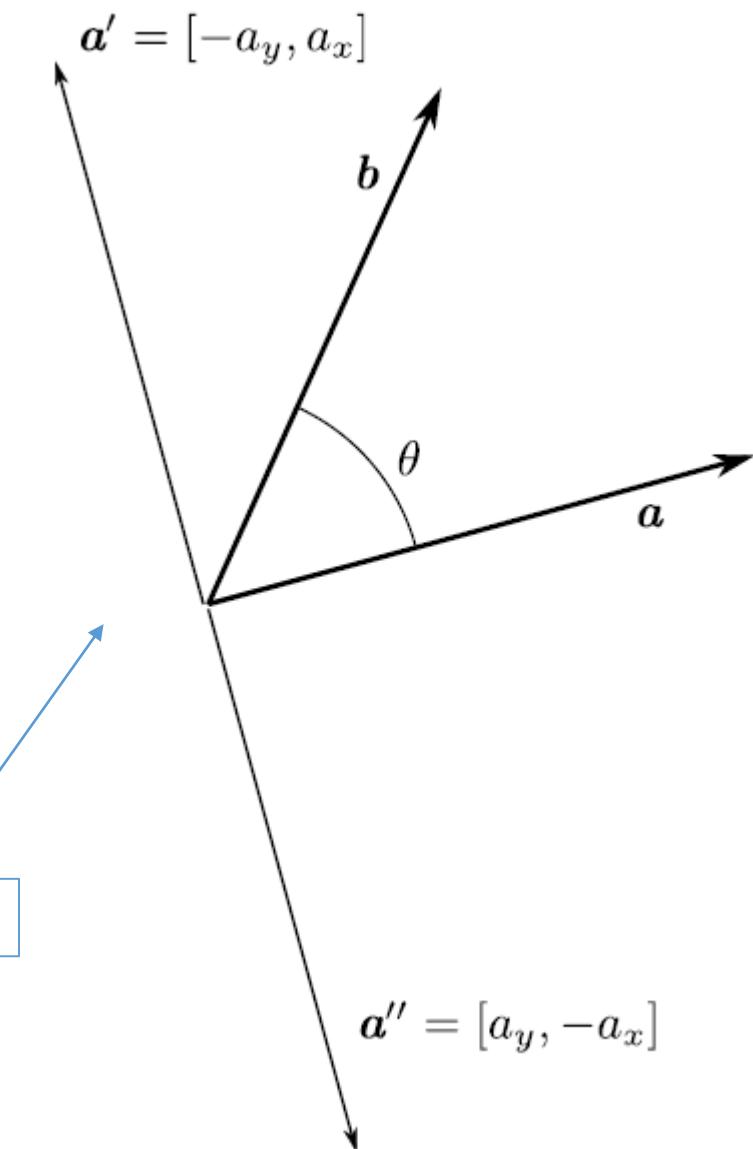


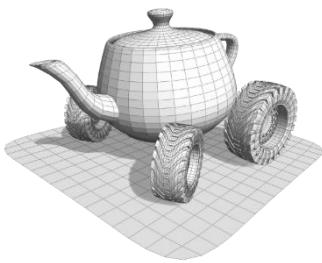
Dot product (2/3)

- The **dot** product is 0 when the vectors are orthogonal
- Given a vector, we can easily find an orthogonal non-zero vector

$$[\dots, a_i, \dots, a_j, \dots] \cdot [0, \dots, 0, a_j, 0, \dots, 0, -a_i, 0, \dots] = \\ = a_i a_j - a_j a_i = 0$$

In 2 dimensions





Dot product (3/3)

- The dot product is maximum for parallel vectors and minimum for antiparallel vectors

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$$

$$\theta = 0 \Rightarrow \mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\|$$



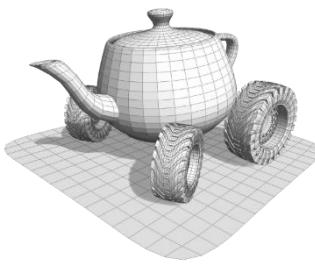
parallel

$$\theta = \pi \Rightarrow \mathbf{a} \cdot \mathbf{b} = -\|\mathbf{a}\| \|\mathbf{b}\|$$



antiparallel

- commutative: $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$
- Distributive over summation: $\mathbf{a} \cdot (\mathbf{c} + \mathbf{d}) = \mathbf{a} \cdot \mathbf{c} + \mathbf{a} \cdot \mathbf{d}$



Cross product (1/3)

- The **cross** product is a binary operator between two vectors in three dimensional space computed as:

$$a \times b = [a_y b_z - b_y a_z, -a_x b_z + b_x a_z, a_x b_y - b_x a_y]$$

- Mnemonic rule: write the 3×3 matrix with variables i, j, k in the first row and compute the determinant.
The multipliers of i, j, k are the x, y, z coordinates of the cross product

$$\det \begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} = i(a_y b_z - b_y a_z) - j(a_x b_z - b_x a_z) + k(a_x b_y - b_x a_y)$$

Cross product (2/3)

- The **cross** product gives a vector that is **orthogonal** to both \mathbf{a} and \mathbf{b}

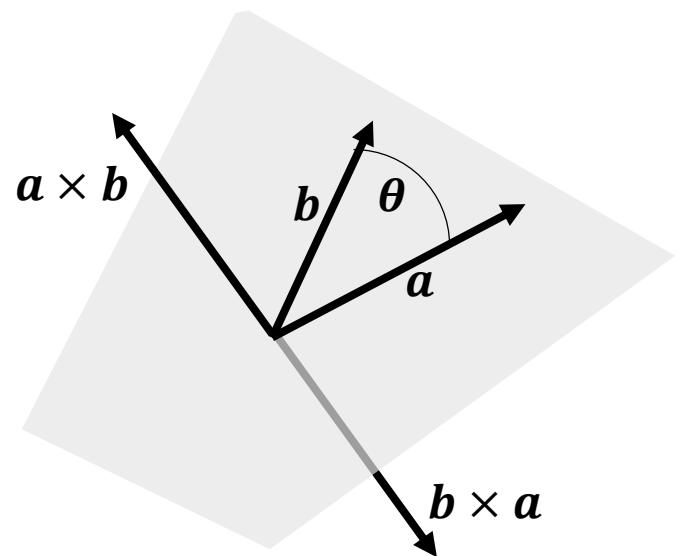
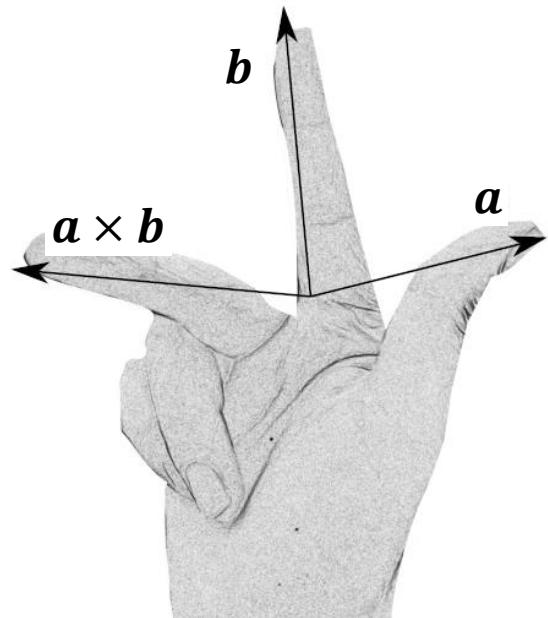
$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{a} = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{b} = 0$$

- The cross product is anticommutative:

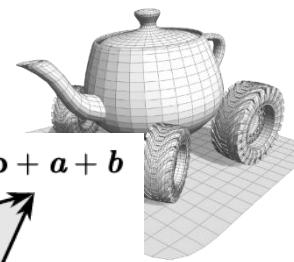
$$\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$$

- It distributes over summation:

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$$



Cross product (3/3)

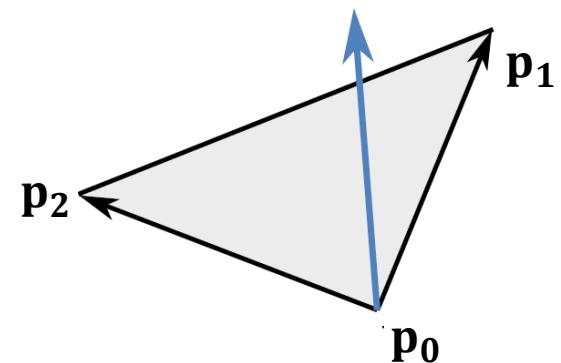
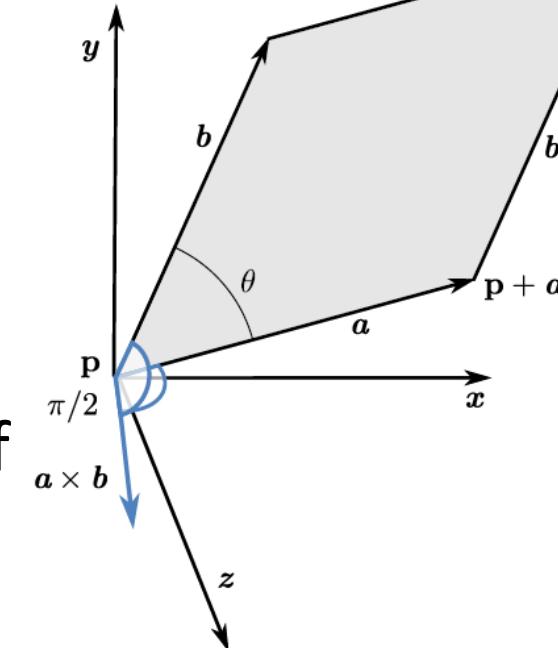


- Norm of cross product:

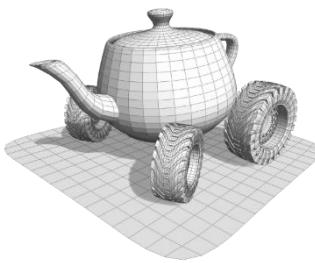
$$\|a \times b\| = \|a\| \|b\| \sin(\theta)$$

- The norm of a cross product equals the area of the parallelogram $\mathbf{p}, \mathbf{p} + \mathbf{a}, \mathbf{p} + \mathbf{a} + \mathbf{b}, \mathbf{p} + \mathbf{b}$
- ..which means that for a triangle $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$:

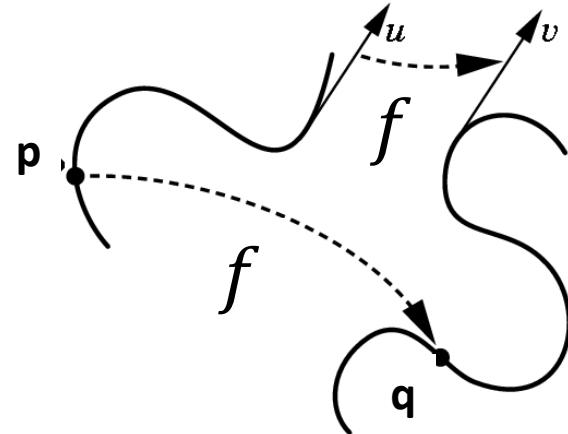
$$\|(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)\| = 2 \text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$$



Geometric transformations

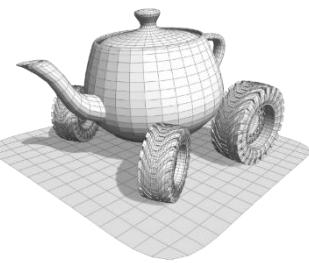


- A **geometric transformation** is a function that maps points to points and vectors to vectors
- We will (mostly) use a specific type of transformations
 - Luckily, the easiest to handle
 - When we say “transform an object” we mean to apply the same transformation to all the points of that object



$$\mathbf{q} = f(\mathbf{p})$$

$$\mathbf{v} = f(\mathbf{u})$$

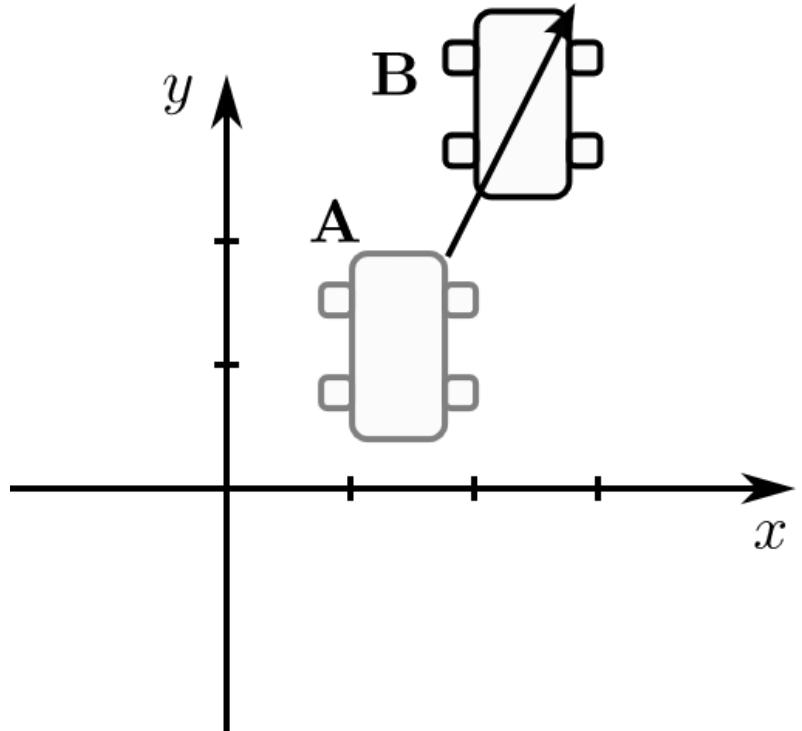


Translation

- Translation is typically used to *move* objects around

$$T_v(\mathbf{p}) = \mathbf{p} + \mathbf{v} = \begin{bmatrix} p_x \\ p_y \end{bmatrix} + \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \end{bmatrix}$$

Translation vector



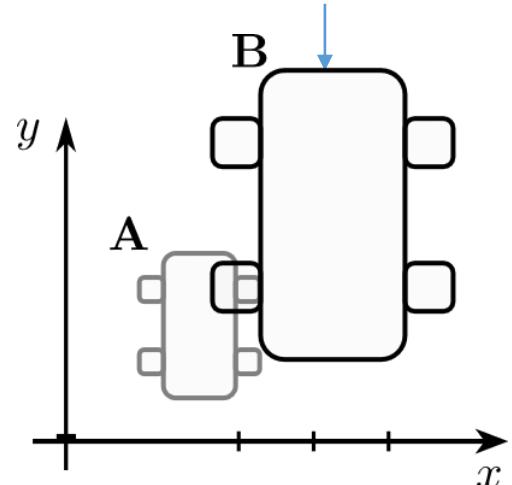
Scaling

- Scaling changes the size of an object
- ...by multiplying for a scaling factor the components of all its points

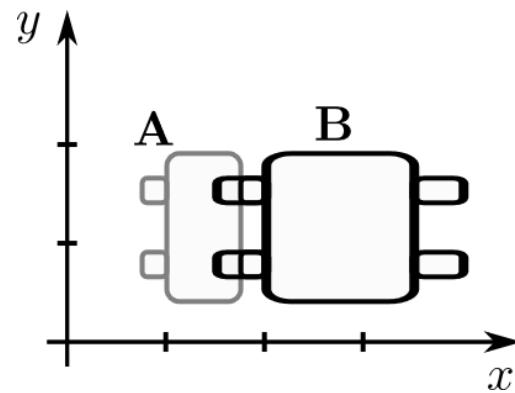
$$S_{(s_x, s_y)}(\mathbf{p}) = \begin{bmatrix} s_x \cdot p_x \\ s_y \cdot p_y \end{bmatrix}$$

- If $s_x = s_y$ than the scaling is **uniform** (or **isotropic**)
 - it means object's proportions do not change
- if $s_x \neq s_y$ than the scaling is **non uniform** (or **anisotropic**)
 - it means object's proportions do change

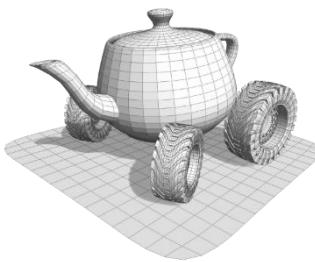
Is this what you expected? Be honest..



Uniform scaling

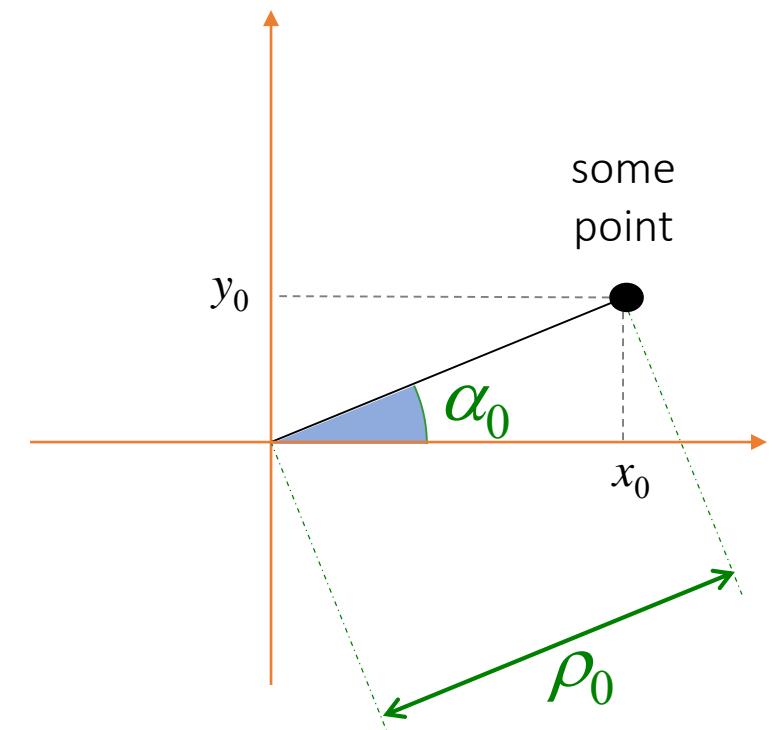
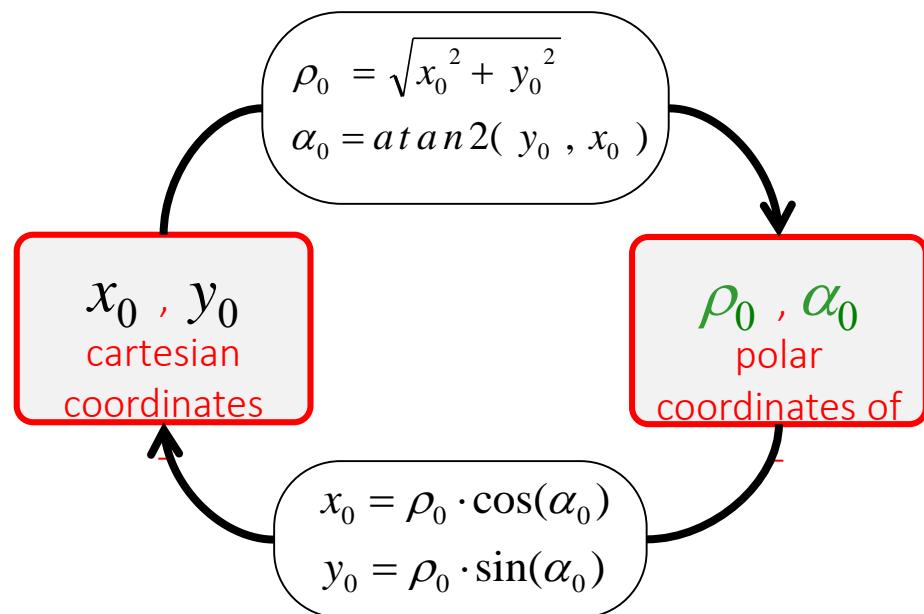


Non uniform scaling



Rotation: Polar Coordinates

- Polar Coordinates: express points and vectors by the tuple (α, ρ)
 - α is the angle with the x -axis
 - ρ is the distance from the origin



Rotation

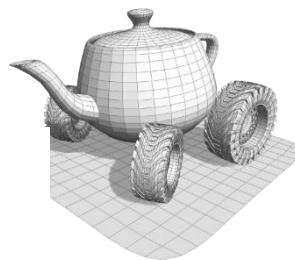
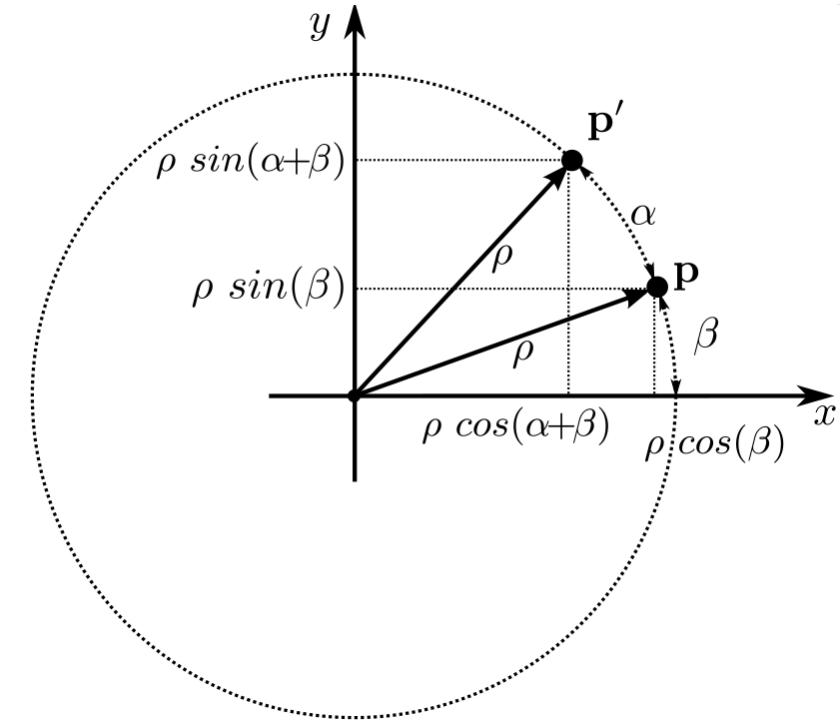
A rotation by α of a point

$$\mathbf{p} = \begin{bmatrix} \rho \cdot \cos(\beta) \\ \rho \cdot \sin(\beta) \end{bmatrix}$$

means: $\mathbf{R}_\alpha(\mathbf{p}) = \begin{bmatrix} \rho \cdot \cos(\beta + \alpha) \\ \rho \cdot \sin(\beta + \alpha) \end{bmatrix}$

$$\boxed{\cos(\beta + \alpha) = \cos \beta \cos \alpha - \sin \beta \sin \alpha} \Rightarrow$$

$$\begin{aligned} p'_x &= \rho \cdot \cos(\beta + \alpha) \\ &= \rho \cos(\beta) \cos(\alpha) - \rho \sin(\beta) \sin(\alpha) \\ &= \rho \frac{p_x}{\rho} \cos(\alpha) + \rho \frac{p_y}{\rho} \sin(\alpha) \\ &= \boxed{p_x \cos(\alpha) - p_y \sin(\alpha)} \end{aligned}$$



Rotation

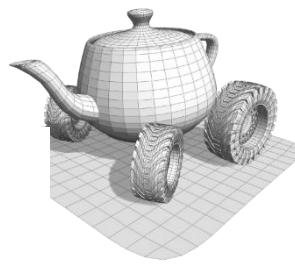
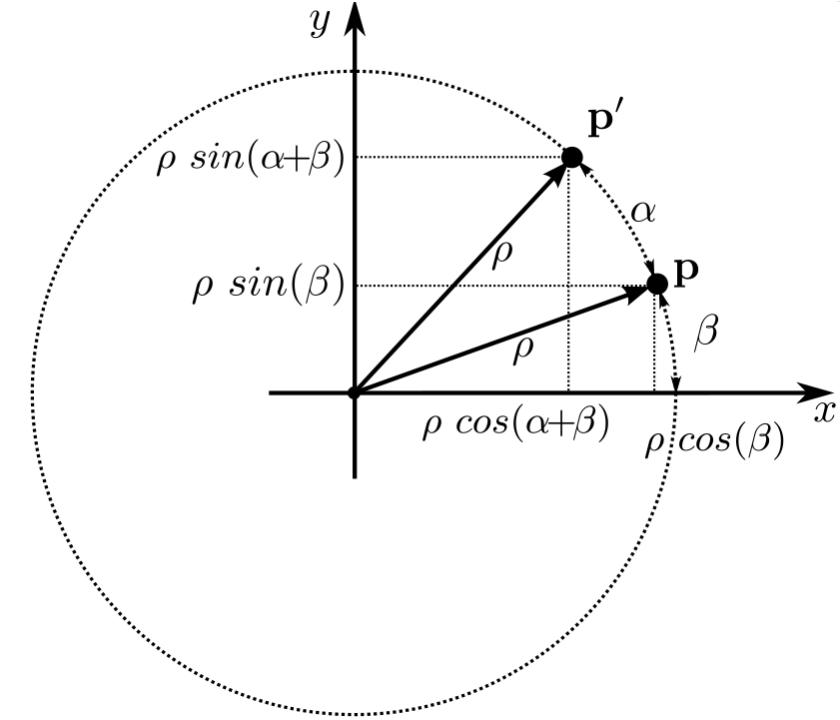
A rotation by α of a point

$$\mathbf{p} = \begin{bmatrix} \rho \cdot \cos(\beta) \\ \rho \cdot \sin(\beta) \end{bmatrix}$$

means: $\mathbf{R}_\alpha(\mathbf{p}) = \begin{bmatrix} \rho \cdot \cos(\beta + \alpha) \\ \rho \cdot \sin(\beta + \alpha) \end{bmatrix}$

$$\sin(\beta + \alpha) = \sin(\alpha) \cos(\beta) + \sin(\beta) \cos(\alpha) \Rightarrow$$

$$\begin{aligned} p'_y &= \rho \cdot \sin(\beta + \alpha) \\ &= \rho \sin(\alpha) \cos(\beta) + \rho \sin(\beta) \cos(\alpha) \\ &= \rho \sin(\alpha) \frac{p_x}{\rho} + \rho \frac{p_y}{\rho} \cos(\alpha) \\ &= \boxed{p_x \sin(\alpha) + p_y \cos(\alpha)} \end{aligned}$$



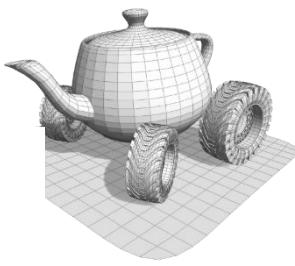
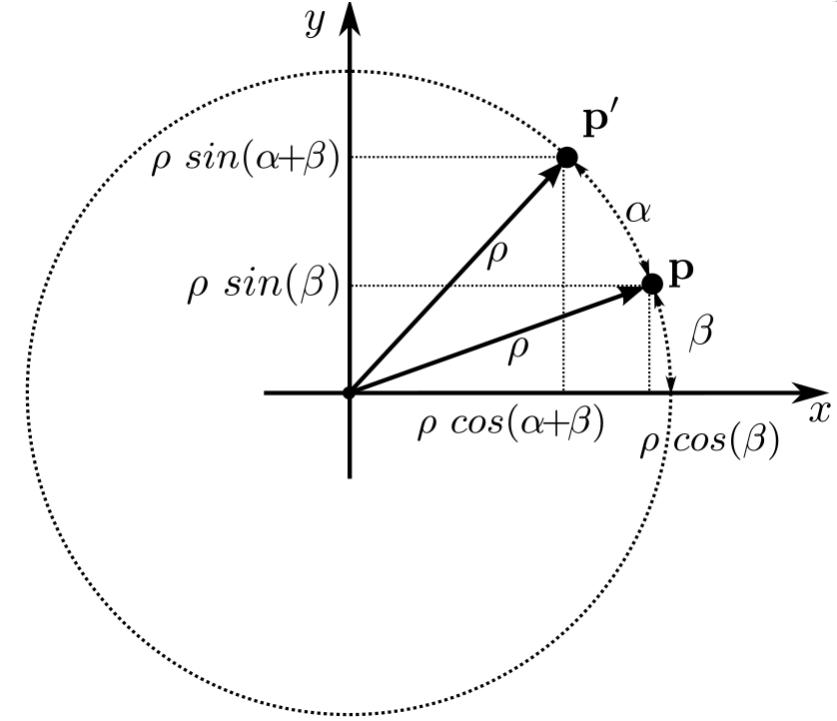
Rotation

A rotation by α of a point

$$\mathbf{p} = \begin{bmatrix} \rho \cdot \cos(\beta) \\ \rho \cdot \sin(\beta) \end{bmatrix}$$

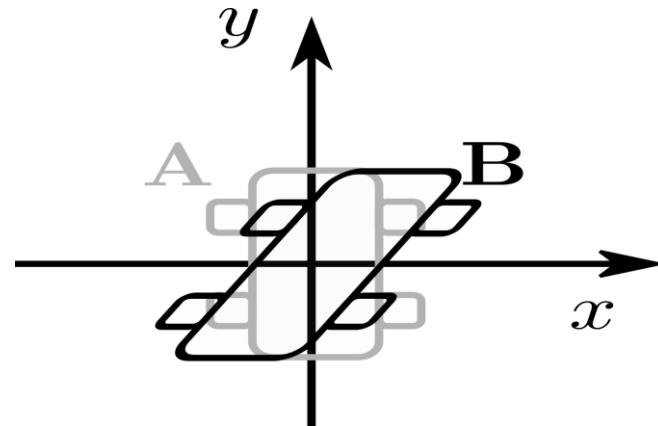
means: $\mathbf{R}_\alpha(\mathbf{p}) = \begin{bmatrix} \rho \cdot \cos(\beta + \alpha) \\ \rho \cdot \sin(\beta + \alpha) \end{bmatrix}$

$$\mathbf{R}_\alpha(\mathbf{p}) = \begin{bmatrix} \cos(\alpha)p_x - \sin(\alpha)p_y \\ \sin(\alpha)p_x + \cos(\alpha)p_y \end{bmatrix}$$

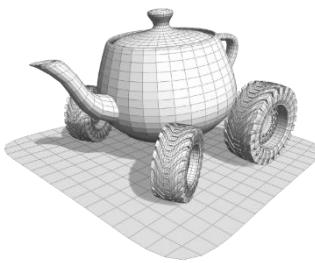
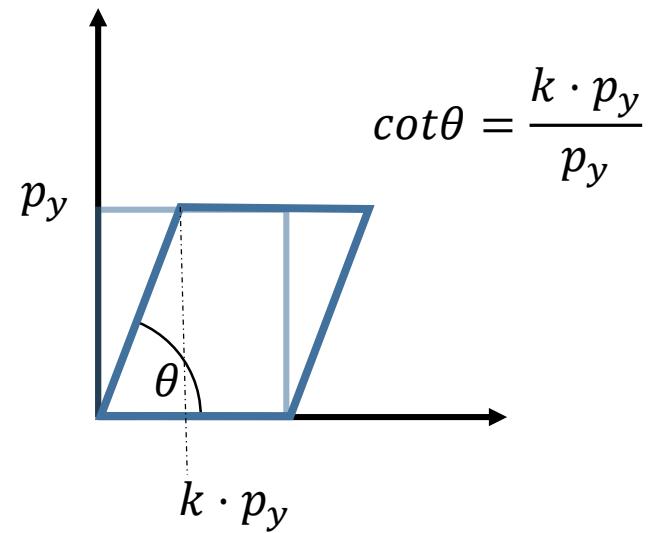


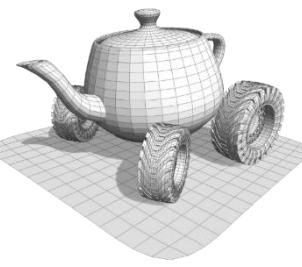
Shear

- Translation of each point along an axis proportionally to the value of the other
- with $k = \cot\theta$ a 90° angle becomes θ



$$Sh_{k,y}(\mathbf{p}) = \begin{bmatrix} \mathbf{p}_x + k \cdot \mathbf{p}_y \\ \mathbf{p}_y \end{bmatrix}$$





Transformations with matrices

- Note that *scaling* and *rotations* are both expressed as

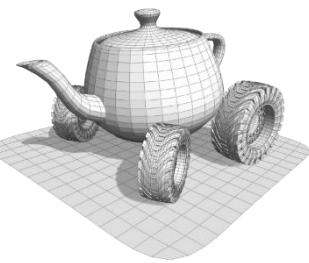
$$p'_x = a_{xx} p_x + a_{xy} p_y$$

$$p'_y = a_{yx} p_x + a_{yy} p_y$$

- Which can be conveniently written as

$$\mathbf{p}' = \begin{bmatrix} a_{xx} & a_{xy} \\ a_{yx} & a_{yy} \end{bmatrix} \mathbf{p}$$

Transformation matrix



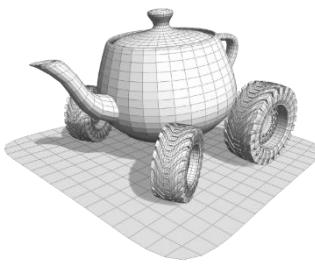
Transformations in matrix form

rotation matrix

$$\mathbf{R}_\alpha = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

$$\mathbf{S}_{(s_x, s_y)} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Scaling matrix



Transformations with matrices

- Matrix-vector product expresses linear *combinations* of the vector terms
- translation **cannot** be expressed in that way, unless we use **homogeneous coordinates**

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

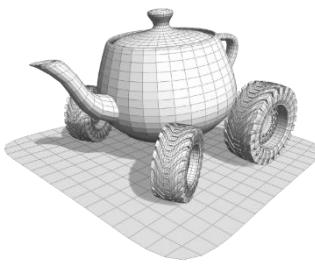
It's a point

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

It's a vector

It works for the point
vector operands

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$



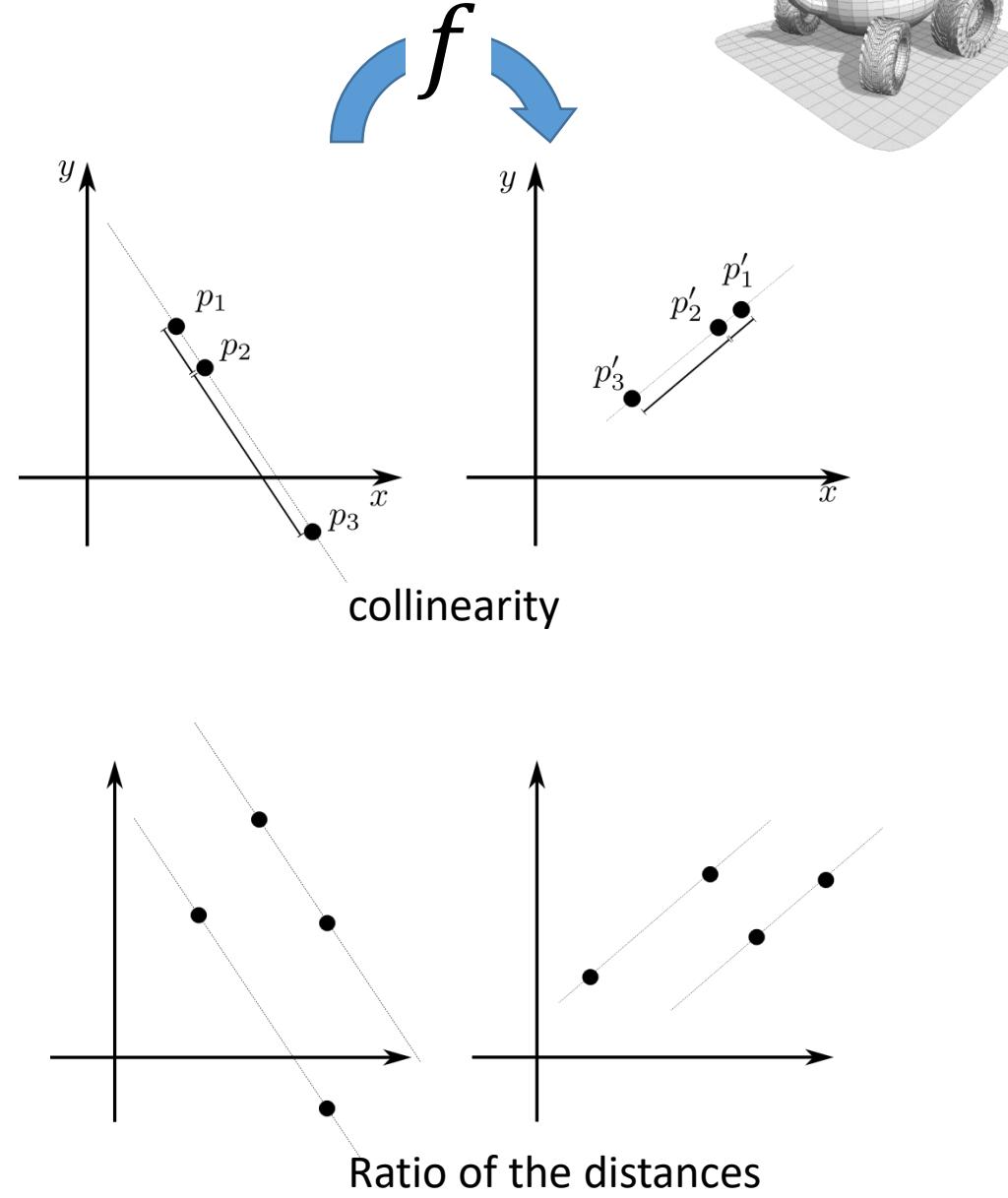
Translation in matrix form

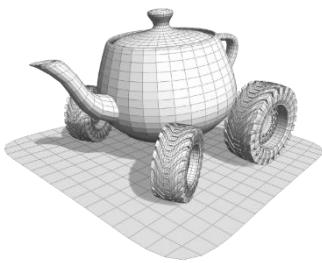
- We need to add one row and one column to our matrix
- The **translation matrix** is $T_v \mathbf{p} = \begin{bmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \\ 1 \end{bmatrix}$
- Recap: translation, rotation, shearing and scaling are all written in the form

$$\begin{bmatrix} a_{xx} & a_{xy} & v_x \\ a_{yx} & a_{yy} & v_y \\ 0 & 0 & 1 \end{bmatrix}$$

Affine transformations (1/3)

- A transformation is **affine** iff:
 - It preserves collinearity. Three collinear points are still collinear after the transformation
 - It preserves the ratio between the distances of pair of points on parallel lines
 - It preserves parallelism. Parallel lines remain parallels
- Translation, scaling and rotation are **affine transformations**
 - And..





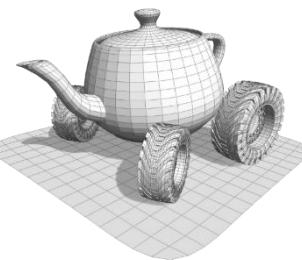
Affine transformations (2/3)

- Every transformation expressed as a matrix of the form

$$M = \begin{bmatrix} a_{xx} & a_{xy} & v_x \\ a_{yx} & a_{yy} & v_y \\ 0 & 0 & 1 \end{bmatrix}$$

These need to be 0

with M non singular (that is invertible: $\det(M) \neq 0$) is an affine transformation



Affine transformations (3/3)

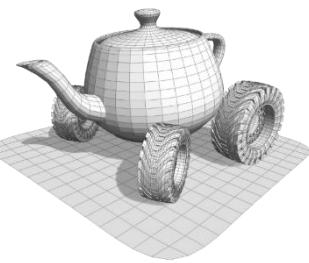
- Familiarize with matrices representing affine transformations

The upper left part of the matrix will contain scaling, rotation and shearing

The last column will contain the translation term

$$\begin{bmatrix} a_{xx} & a_{xy} & v_x \\ a_{yx} & a_{yy} & v_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{xx}p_x + a_{xy}p_y + v_x \\ a_{yx}p_x + a_{yy}p_y + v_y \\ 1 \end{bmatrix}$$
$$= \begin{bmatrix} [a_{xx} \ a_{xy}] \\ [a_{yx} \ a_{yy}] \\ 1 \end{bmatrix} \mathbf{p} + \mathbf{v}$$

These will be 0



Composition of Transformations (1/2)

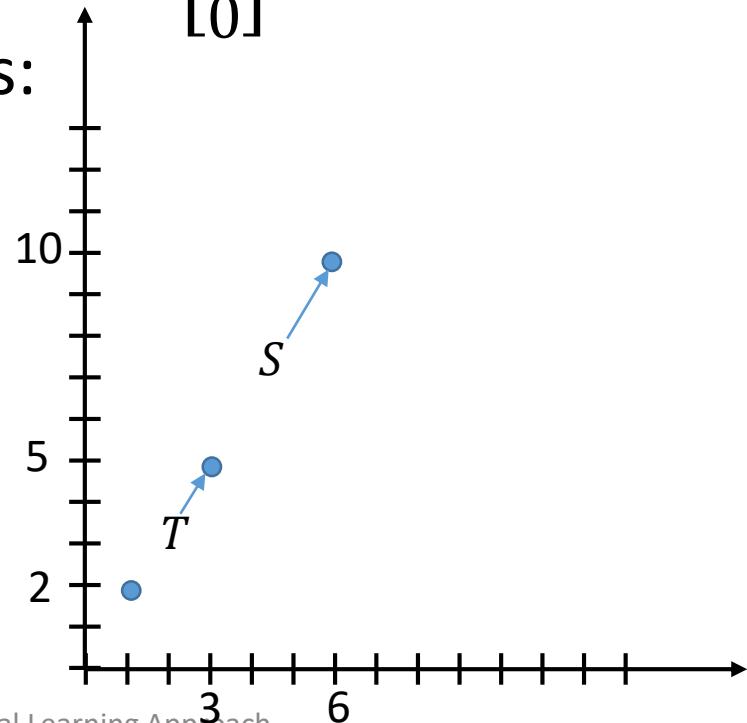
- Composing transformations means to apply a *sequence* of them

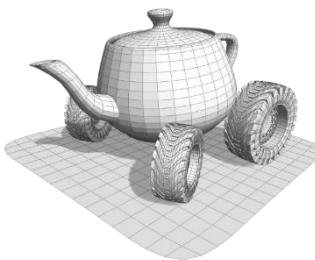
- Example: translate point $p = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ by vector $v = \begin{bmatrix} 2 \\ 3 \\ 0 \end{bmatrix}$ and **then** scale the translated point by 2 along both axes:

$$S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

$$p' = S \cdot T \cdot p = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 4 \\ 0 & 2 & 6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$



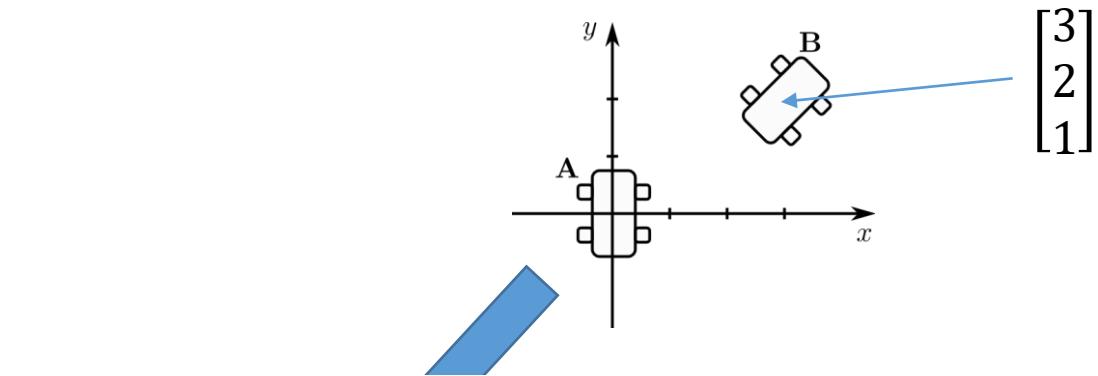


Composition of Transformations (2/2)

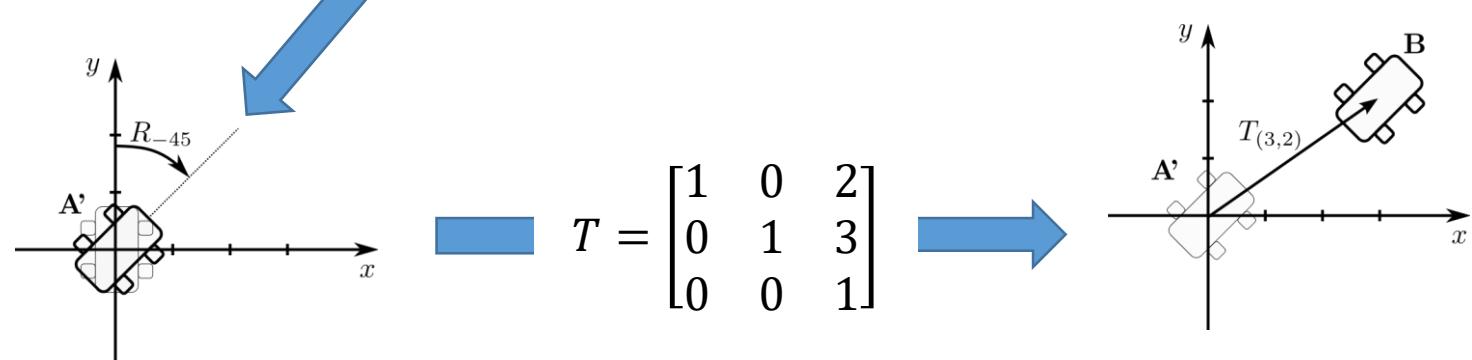
- Example: move the car from A to B

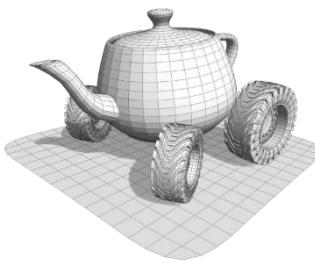
- B is rotated by 45° clockwise

- B is centered into $\begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$



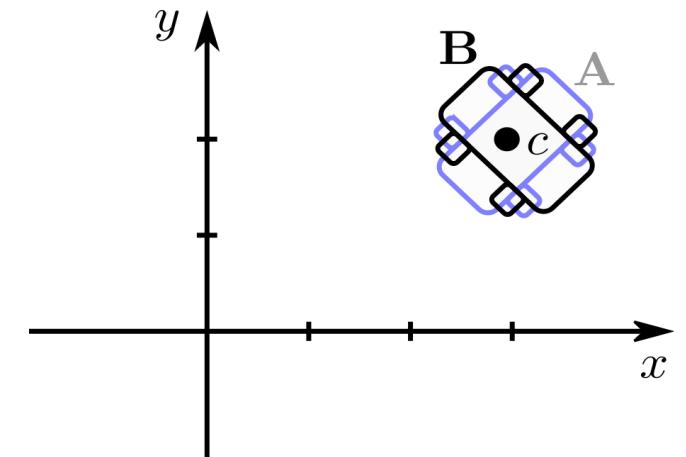
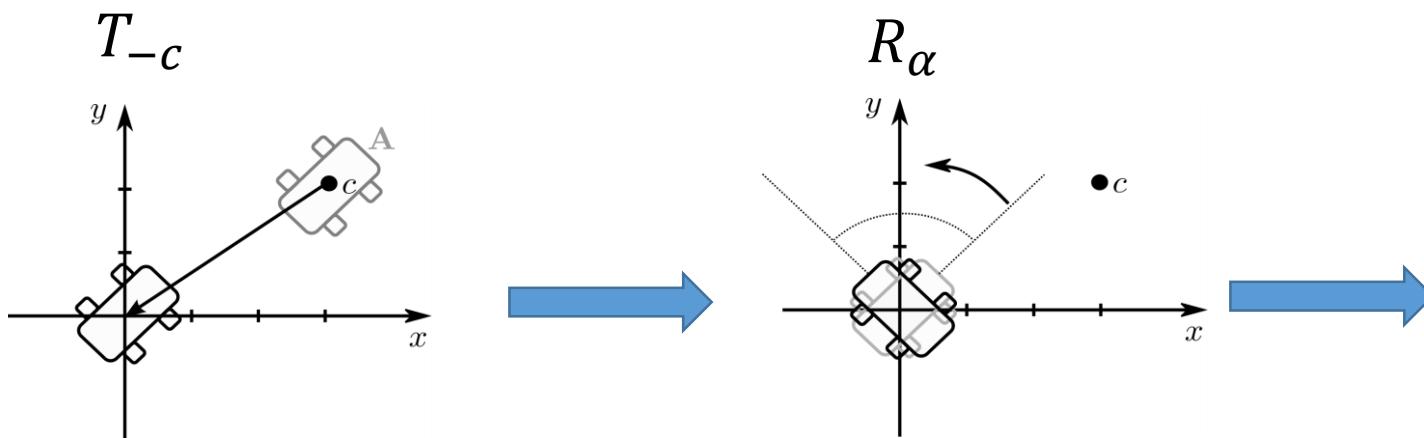
$$R_{-45^\circ} = \begin{bmatrix} \cos(-45^\circ) & -\sin(-45^\circ) & 0 \\ \sin(-45^\circ) & \cos(-45^\circ) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



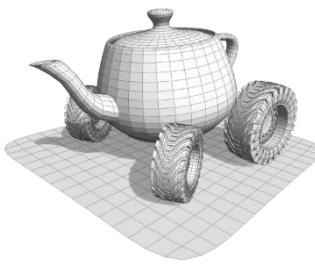


Rotation around a specific point

- Rotate A to B around point c
 - 1. Translate so that c is brought to the origin
 - 2. Rotate
 - 3. Translate it back to c



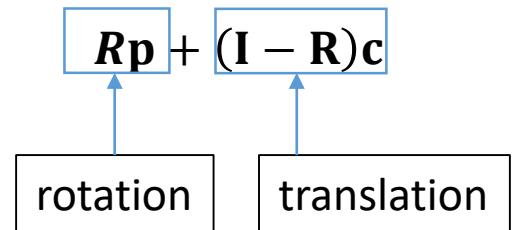
$$R_{\alpha,c} = T_c R_\alpha T_{-c}$$



Form of a rotation matrix around a point

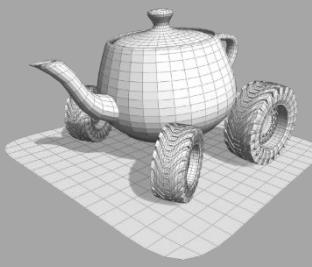
- Let's apply the same transformations in Cartesian coordinates
 - That is, without extending the column with 1s and 0s
- A rotation followed by a translation is referred to as as **rototranslation**
- every composition of affine transformations is an affine transformation

$$\mathbf{p}' = (\mathbf{R}(\mathbf{p} - \mathbf{c})) + \mathbf{c} = \mathbf{Rp} - \mathbf{Rc} + \mathbf{c} =$$



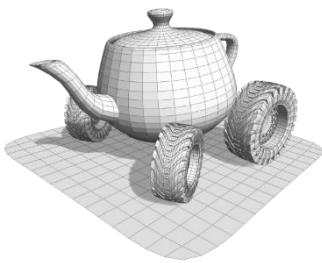
$$R_{\alpha,c} = \begin{bmatrix} R_\alpha & (I - R_\alpha)c \\ 0 & 1 \end{bmatrix}$$

How convenient! 3 transformations are still just one 4x4 matrix

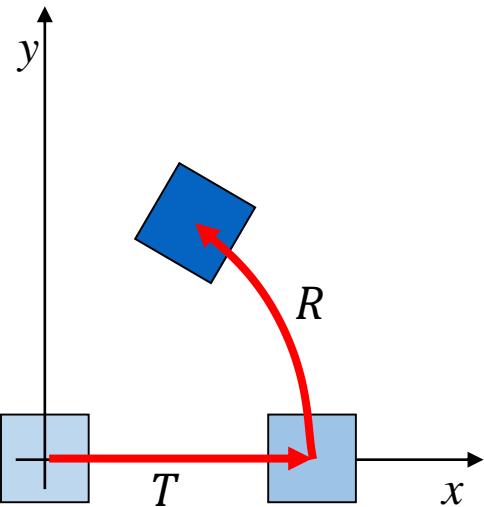


Recall: properties of matrix operations

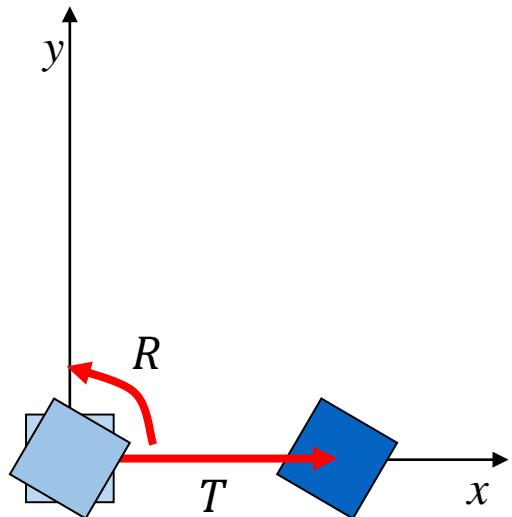
- Matrix product is **associative**: $(A B)C = A (B C)$
- Matrix product is **distributive**: $A(B + C) = AB + AC$
- Matrix product is **not commutative**: $A B \neq B A$
 - ..in general, of course it is in special circumstances: one of the two I = or the identity, two rotations around the same axis, two scalings ..etc
- The order in which transformations are applied *matters*



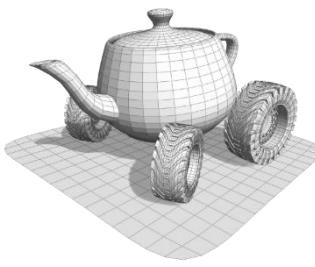
Order of transformations matters



Translation followed by a rotation



Rotation followed by a translation



Inverse of affine transformations (1/2)

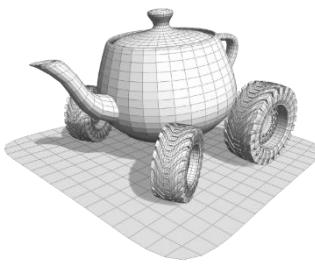
- Since an affine transformation is represented with a matrix A , its inverse is the inverse of the matrix A^{-1} , so that is

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

$$(AB)^{-1} = B^{-1}A^{-1}$$

The inverse of a translation is trivial

$$A = \begin{bmatrix} M & \mathbf{t} \\ 0 & 1 \end{bmatrix}^{-1} = \left(\begin{bmatrix} I & \mathbf{t} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} M & \mathbf{0} \\ 0 & 1 \end{bmatrix} \right)^{-1} = \begin{bmatrix} M & \mathbf{0} \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} I & \mathbf{t} \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} M & \mathbf{0} \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} I & -\mathbf{t} \\ 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} M^{-1} & -M^{-1}\mathbf{t} \\ 0 & 1 \end{bmatrix}$$



Inverse of affine transformations (2/2)

- **Rotation:** the rotation matrix is **orthonormal**, that is, its rows and columns have unitary norm and their dot product is 0

- If a matrix M is orthonormal its inverse is its transposed matrix: $M^{-1} = M^T$, so:

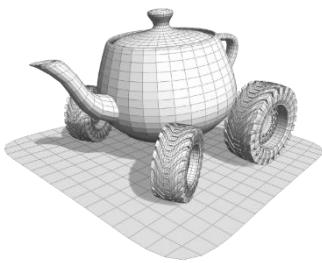
$$\mathbf{R}_\alpha^{-1} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}^T = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

- **Scaling:**

$$\mathbf{S}_{(s_x, s_y)}^{-1} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}^{-1} = \begin{bmatrix} s_x^{-1} & 0 \\ 0 & s_y^{-1} \end{bmatrix}$$

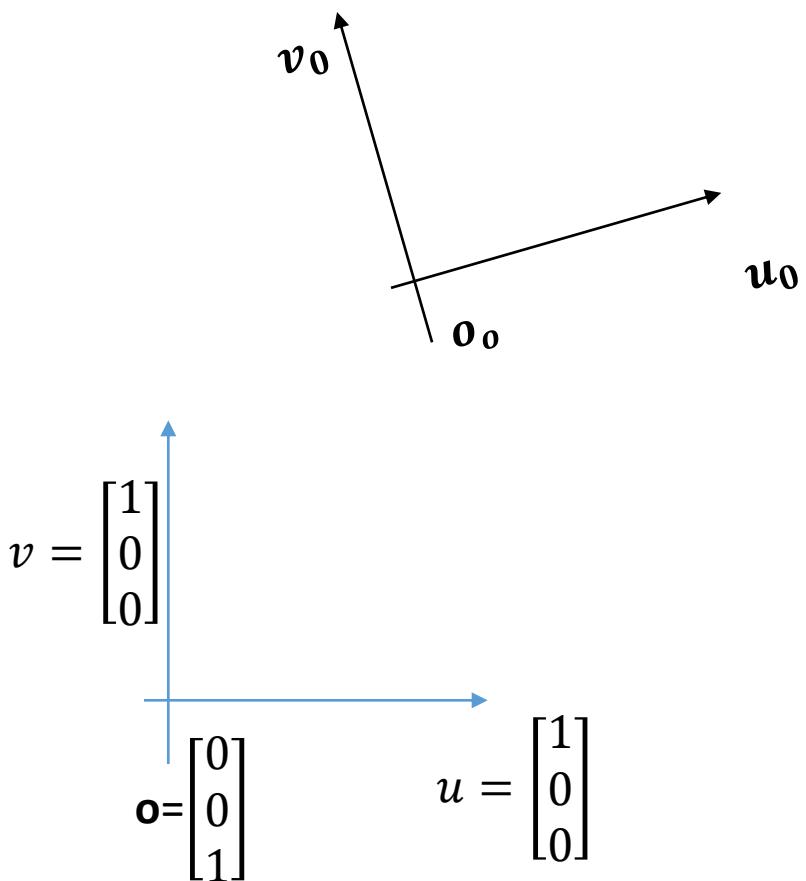
- **Shear:**

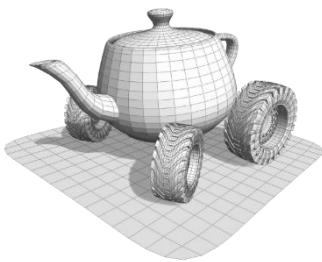
$$\mathbf{Sh}_h^{-1} = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & -h \\ 0 & 1 \end{bmatrix}$$



Frames

- A **frame** consists of a point, called **origin** and 2 independent vectors called **axis** (3 in 3D)
- A frame defines what the coordinates of a point are
- A frame (origin and axis) is itself expressed in some frame
- We call **canonical frame** the frame with origin in $[0,0,1]^T$ and axis $u = [1,0,0]^T$ and $v = [0,1,0]^T$





From a frame to its canonical frame

- Given a frame $F = \{\mathbf{o}, \mathbf{u}, \mathbf{v}\}$ and the coordinates of a point $[x, y]^T$ in F , what are its coordinates in the canonical frame?

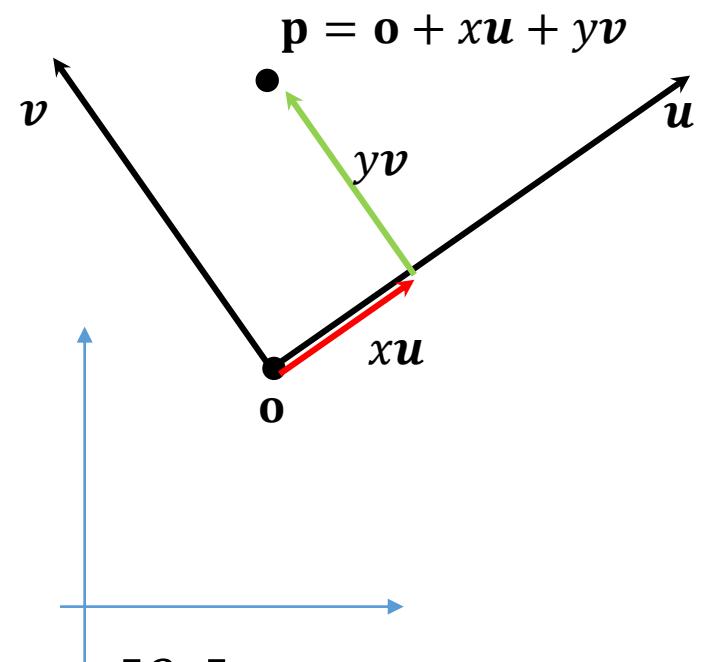
$$\mathbf{p} = \mathbf{o} + x\mathbf{u} + y\mathbf{v}$$

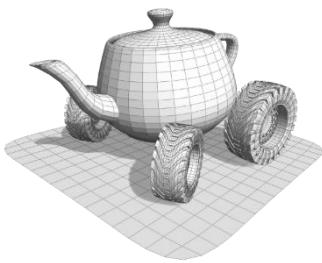
- In matrix form:

Axes of the frame as columns

$$\mathbf{p} = \begin{bmatrix} u_x & v_x \\ u_y & v_y \\ 0 & 0 \end{bmatrix} \begin{bmatrix} o_x \\ o_y \\ 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = x \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} + y \begin{bmatrix} v_x \\ v_y \\ 1 \end{bmatrix} + 1 \begin{bmatrix} o_x \\ o_y \\ 1 \end{bmatrix}$$

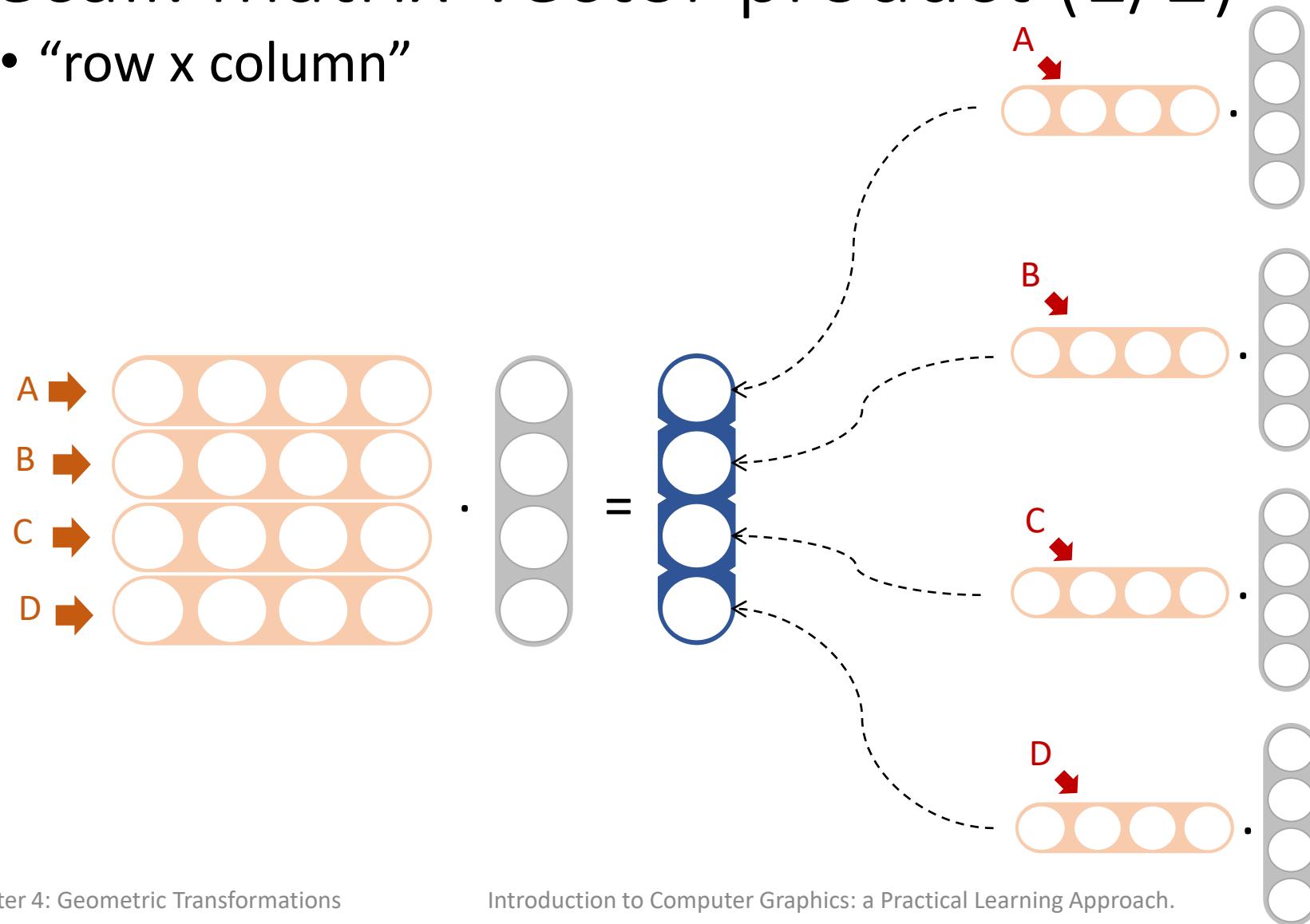
Origin as translation

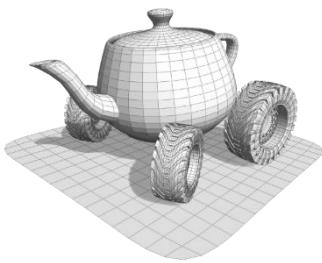




Recall: matrix-vector product (1/2)

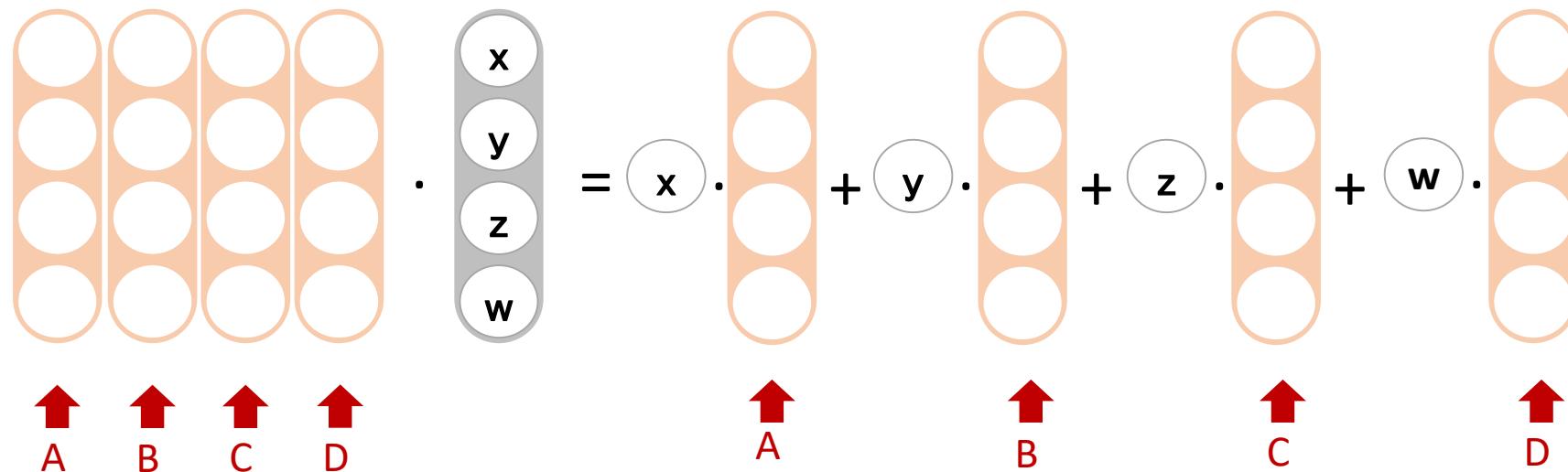
- “row x column”

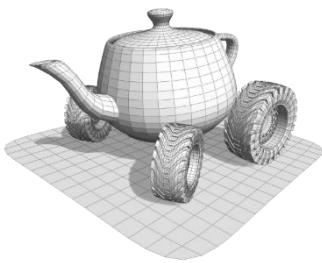




Recall: Matrix-vector product

- “row x column”
- Linear combination of the matrix column vectors



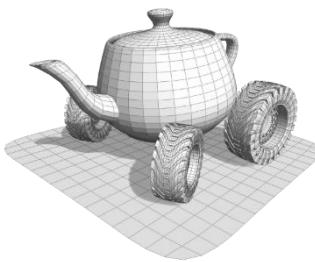


Frames as matrices

- a frame $F = \{\mathbf{o}, \mathbf{u}, \mathbf{v}\}$ can be conveniently represented as a matrix with the axis and origin as columns:

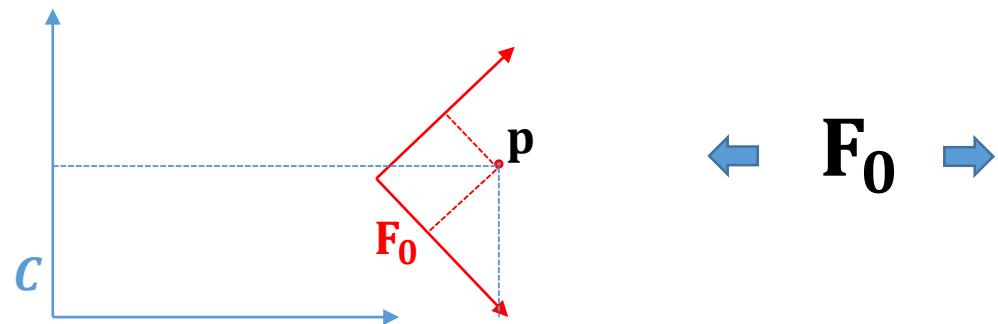
$$\mathbf{F} = \begin{bmatrix} u_x & v_x & o_x \\ u_y & v_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Remember:
 - If \mathbf{p}_F are coordinates of point \mathbf{p} expressed in the frame F , $F\mathbf{p}_F$ are the coordinates of \mathbf{p} expressed in the canonical frame
 - If \mathbf{p}_C are coordinates of point \mathbf{p} expressed in the canonical frame C , $F^{-1}\mathbf{p}_C$ are the coordinates of \mathbf{p} expressed in the frame F

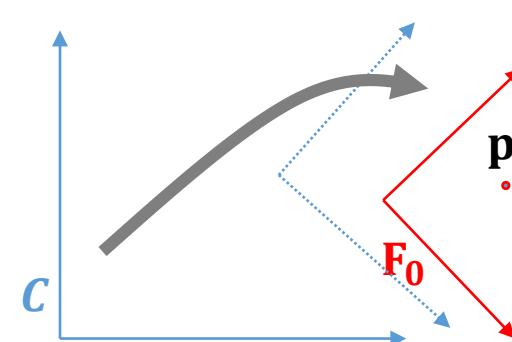


Frames & Transformations

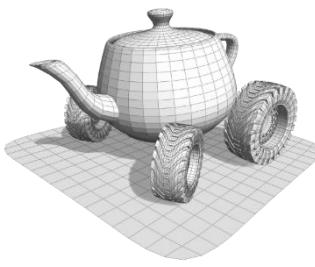
- Expressing coordinates from one frame to another is referred to as **changing frame**
- Changes of frame are affine transformations, it's merely a matter of which inner representation better suit our goal/intuition



It transforms the coordinates from F_0 to C



It transforms C into F_0 !



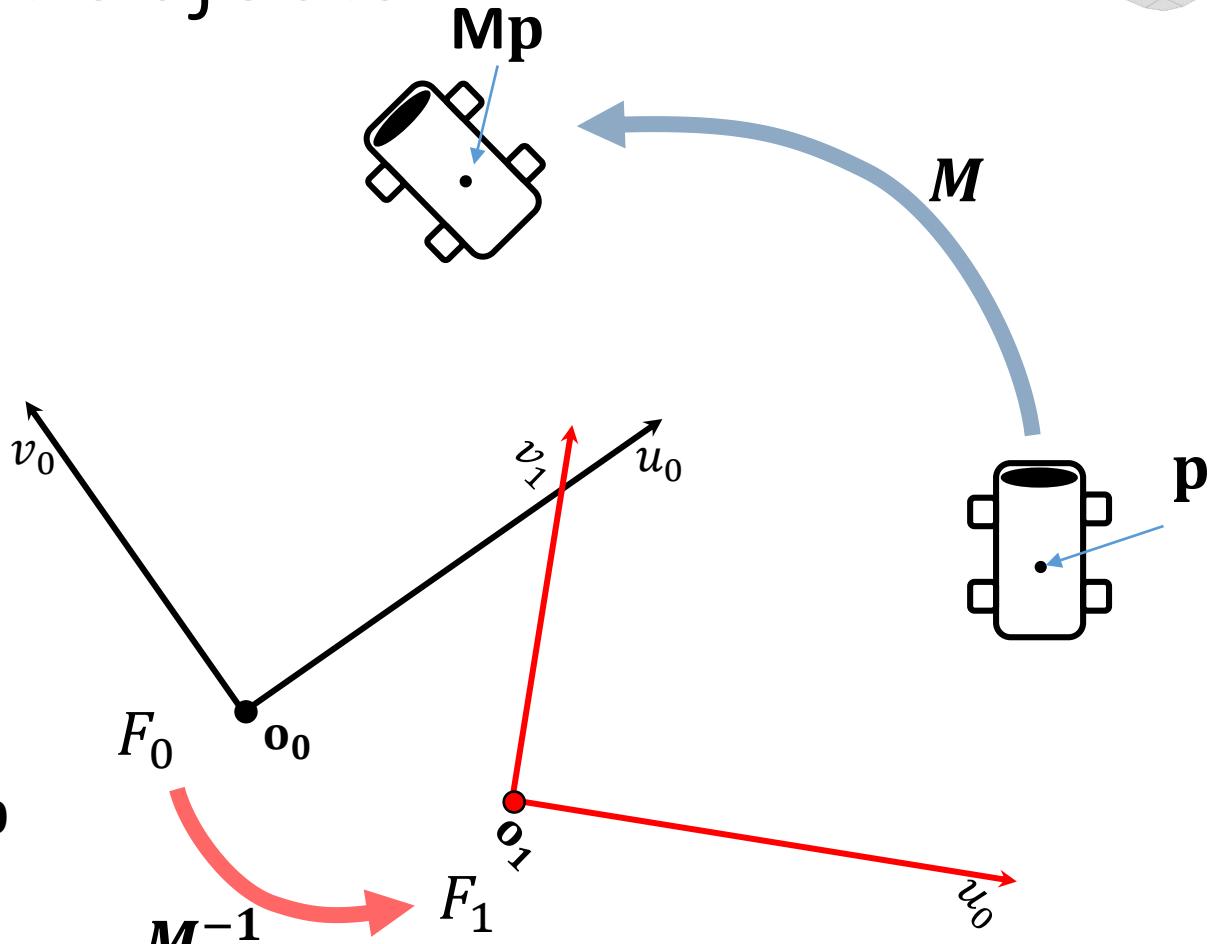
Transforming Frames & objects

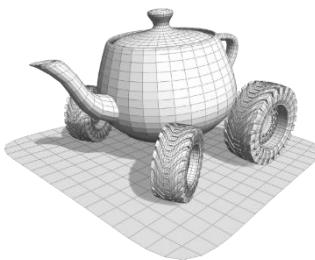
- Applying a transformation M to an object will change its coordinates in the frame F_0
- The same coordinates are obtained by applying the transformation M^{-1} to the frame F_0

$$(Mp)_0 = F_0^{-1}Mp = F_1^{-1}p \Rightarrow F_1 = M^{-1}F_0$$

Coordinates of Mp in F_0

Coordinates of p in F_1





From a frame to *another* frame

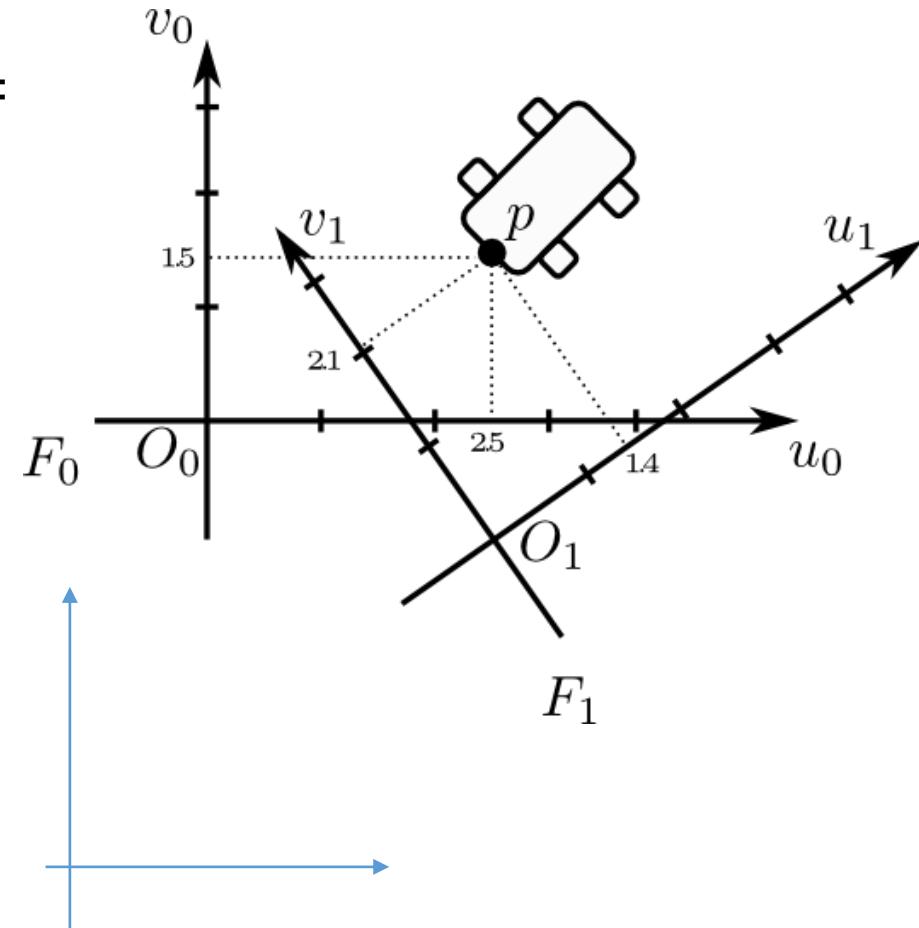
- Given the frames $F_0 = \{O_0, u_0, v_0\}$ and $F_1 = \{O_1, u_1, v_1\}$ what are the coordinates in F_1 of a point \mathbf{p}_0 in F_0 ?
- Note: the coordinates in the canonical frame are the same:

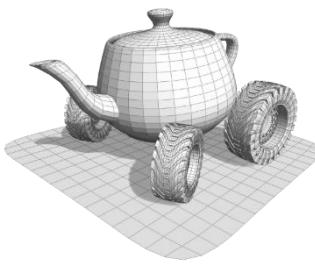
$$\mathbf{p} = F_0 \mathbf{p}_0 = F_1 \mathbf{p}_1$$

coordinates in
the canonical
frame



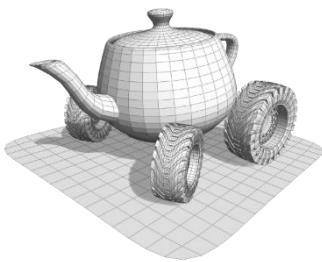
$$\mathbf{p}_1 = F_1^{-1} F_0 \mathbf{p}_0$$





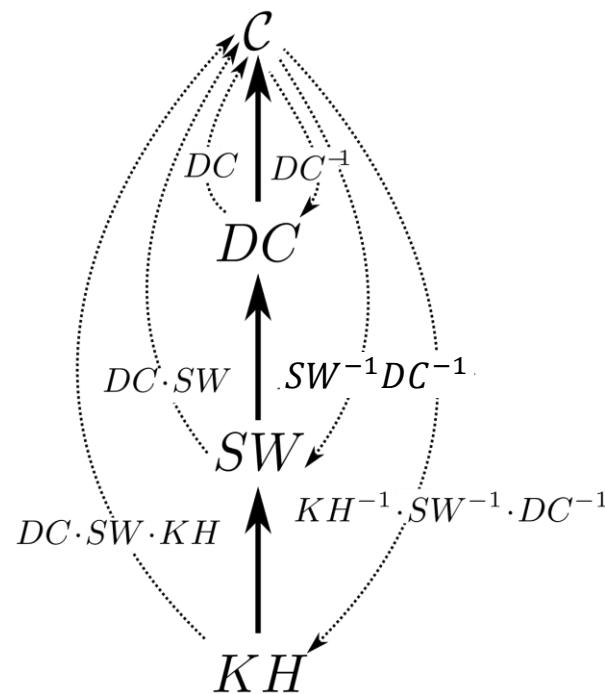
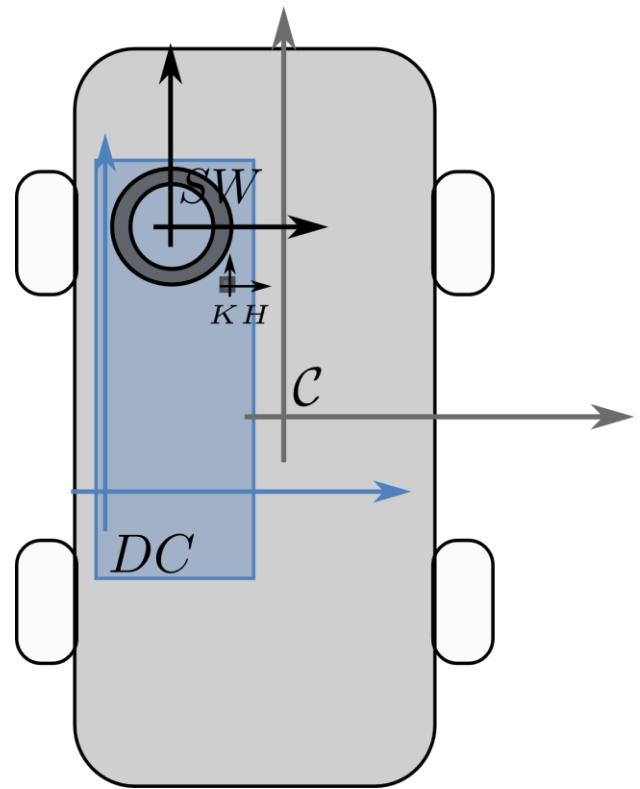
Frames & Transformations

Type of frames	example	transformation
axis aligned orthonormal frame	$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$	translation
Orthonormal frame centered at the origin	$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	rotation
Axis aligned, non unitary orthogonal frames centered at the origin	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	scaling
Non orthogonal non unitary frames	$\begin{bmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	shear

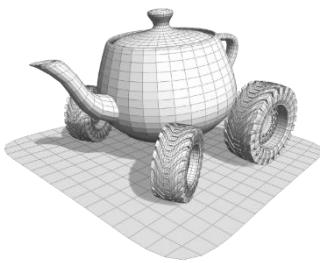


Hierarchies of frames

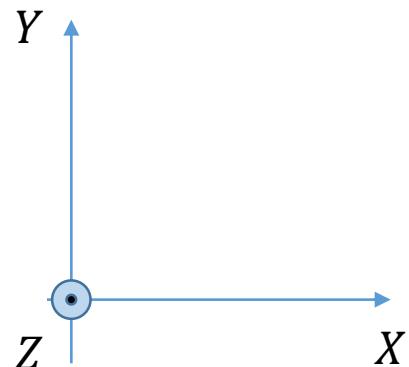
- It's convenient to *think* the scene as a hierarchy of frames: each frame is expressed in its parent's frame
- Example: the front-right wheel is at $[1,0.5,1.0]^T$ w.r.t. the frame of the car
- A directed graph may be useful to represent relations and transformations

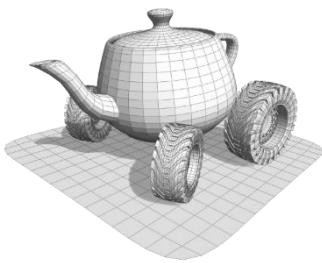


The third dimension



- Everything we saw extend to 3D, we only need to add one more row and column to the transformation matrices
- From now on, we will deal with 4×4 matrices and 4 dimension vectors
- Only rotations need some more care...(next)





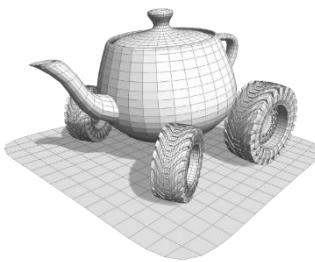
Rotations in 3D

- In 3D the rotations are w.r.t to an axis passing through the origin
 - The 2D rotations correspond to the 3D rotations around the Z-axis
 - Rotations around the principal axes:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

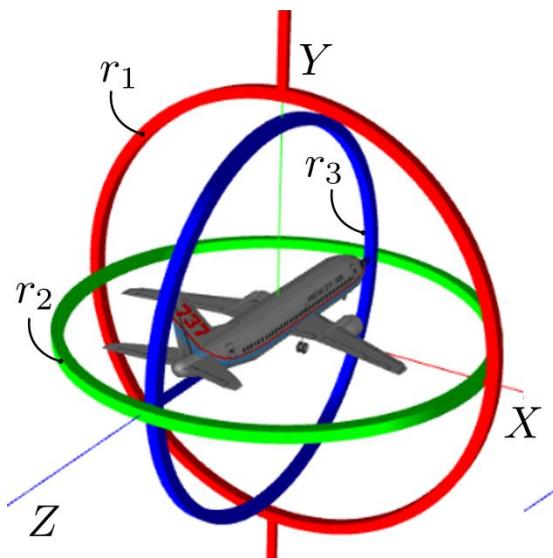
$$R_y(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

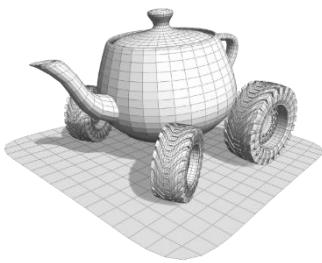
$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Euler Angles Rotations

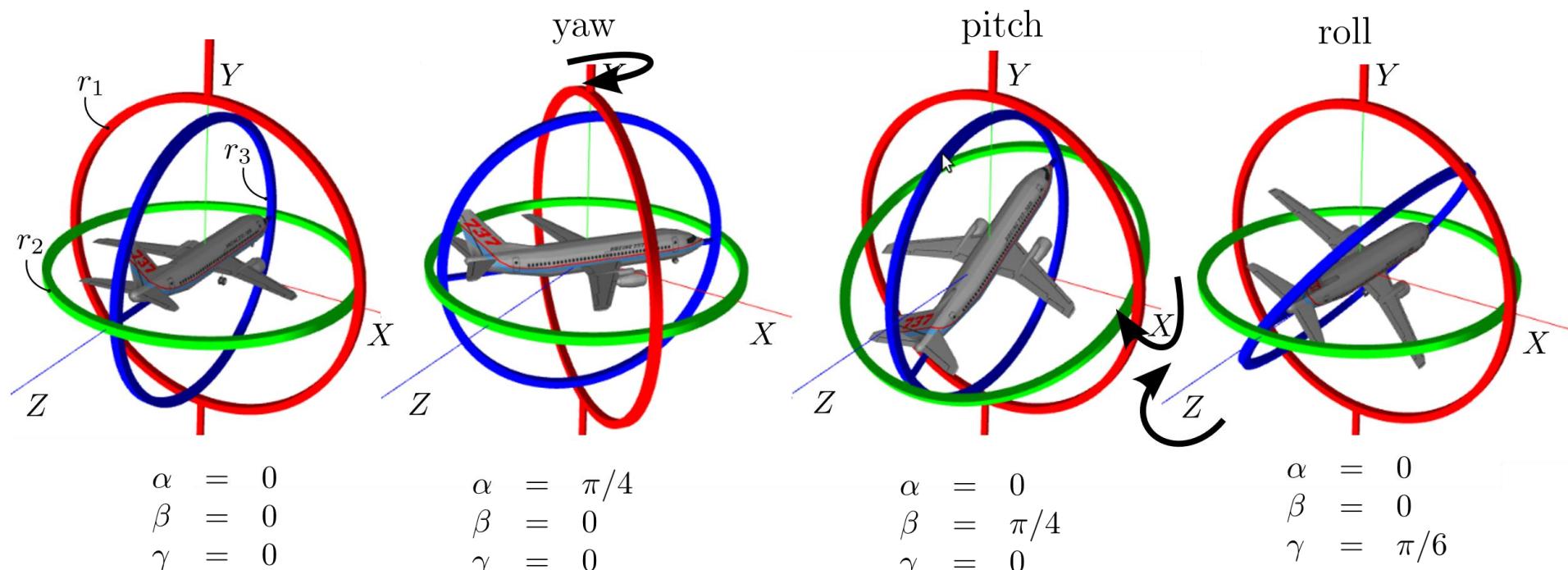
- Assume the object is mounted on a gimbal
- **Gimbal:** a system of 3 concentric rings where each one is bound to its outer ring through a support that allows rotation
- The rotation is a composition of:
 - yaw: rotation around axis y
 - pitch: rotation around axis x
 - roll: rotation around axis z

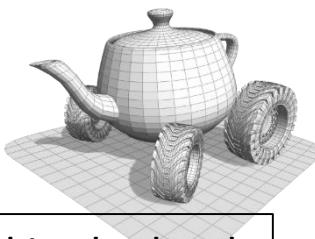




Euler Angles Rotations

- demo





Frames of the gimbal

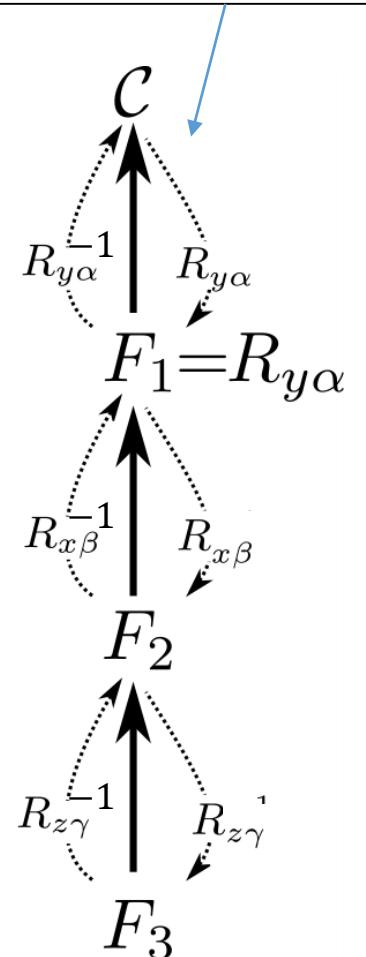
- Consider each ring as associate to its own frame
 - Ring 1 to F_1 , and F_1 rotates by α around its y axis (yaw)
 - Ring 2 to F_2 , and F_2 rotates by β around its x axis (pitch)
 - Ring 3 to F_3 , and F_3 rotates by γ around its z axis (roll)
- The rotation α applied to F_1 also rotates F_2 and F_3
- The rotation β applied to F_2 also rotates F_3
- The rotation γ applied to F_3 rotates the aircraft

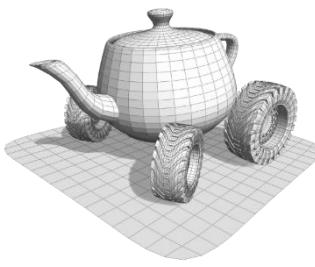
$$F_1 = R_{y\alpha} C$$

$$F_2 = R_{x\beta} \ R_{y\alpha} \ C$$

$$F_3 = R_{z\gamma} \ R_{x\beta} \ R_{y\alpha} C$$

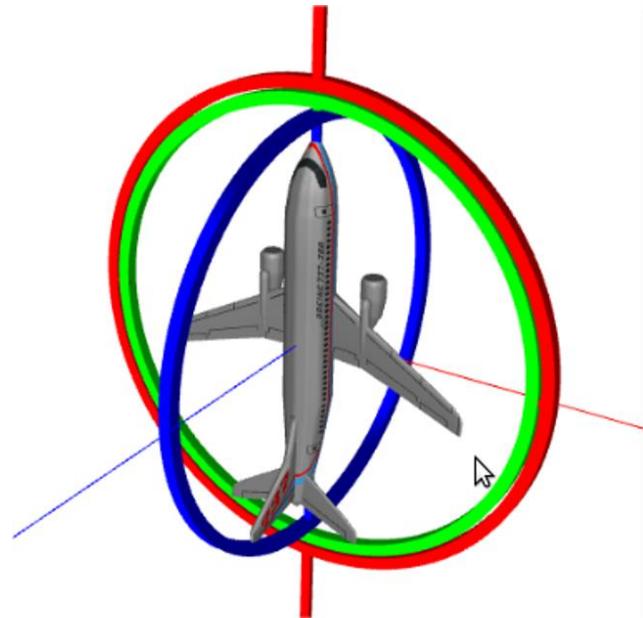
-1s are inverted in the book



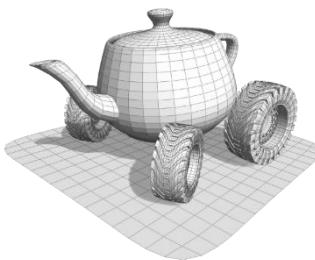


Gimbal lock

- α , β , and γ define a *surjective* function to the space of rotation
 - It means that *every* rotation can be obtained
- However, the function is not *injective*
 - it means that more than a combination of α , β , and γ define the same rotation
- For example: if $\beta = 90^\circ$ then all values of α and γ such that $\alpha = -\gamma$ give the same rotation
- So a degree of freedom is lost. This effect is known as **Gimbal Lock**

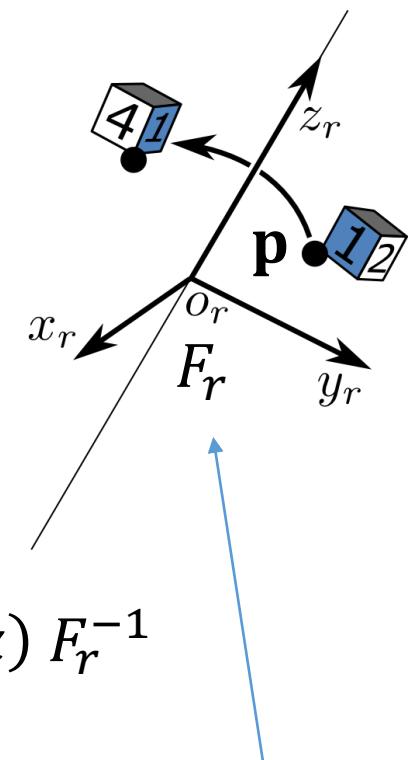
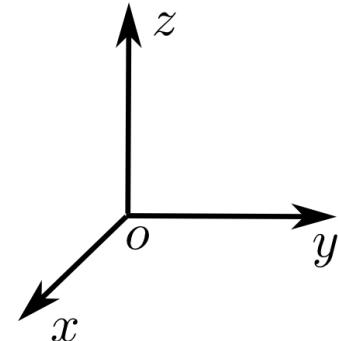


$$\begin{aligned}\alpha &= 0 \\ \beta &= \pi/2 \\ \gamma &= 0\end{aligned}$$



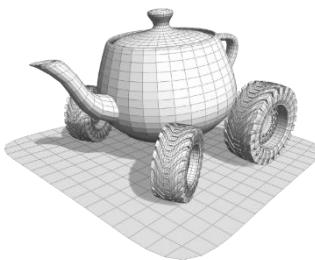
Rotation around a generic axis (1/2)

- Find the rotation around a generic axis $\mathbf{o}_r + d \mathbf{z}_r$
- Same technique as for 2D rotations around a point:
 1. Consider a frame F_r such that its z axis is z_r . Apply the transformation that bring F_r to the canonical frame
 2. Apply the rotation matrix for the Z axis
 3. Apply the inverse of the transformation applied in 1



$$R_{(o_r, z_r)}(\alpha) = F_r \ R_z(\alpha) F_r^{-1}$$

Where does F_r come from?



Building a frame from a point and an axis

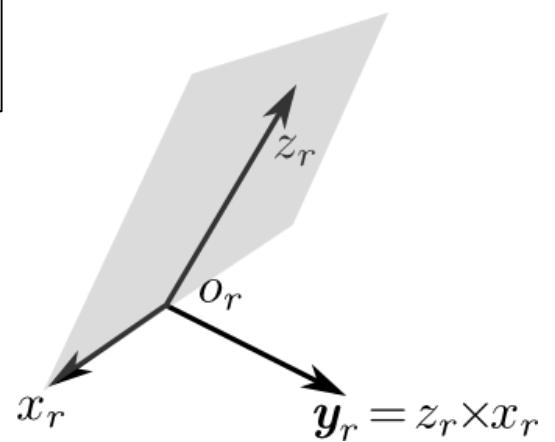
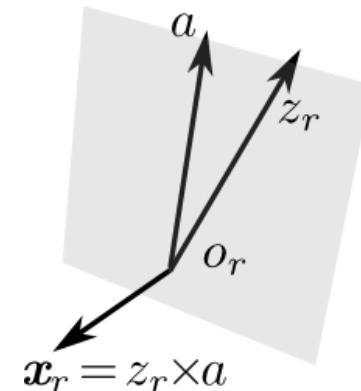
- Recall: the vector product of two vectors is orthogonal to both:

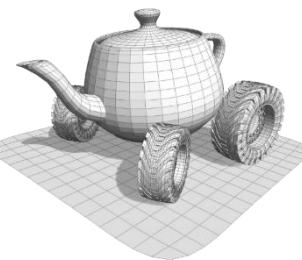
1. Pick a vector $a \neq z_r$

Normalization is not

Note: if a is too near to be collinear with z_r numerical stability can be an issue.
A typical choice is the principal axis most orthogonal to z_r , that is, the one with the non 1 coordinate corresponding to the smalles absolute value of coordinates of z_r

Example: if $z_r = \begin{bmatrix} 0.9 \\ 0.2 \\ -0.4 \\ 0 \end{bmatrix}$ then $a = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$





Rotation around a generic axis (2/2)

- Another way to do it (for axes passing through the origin)

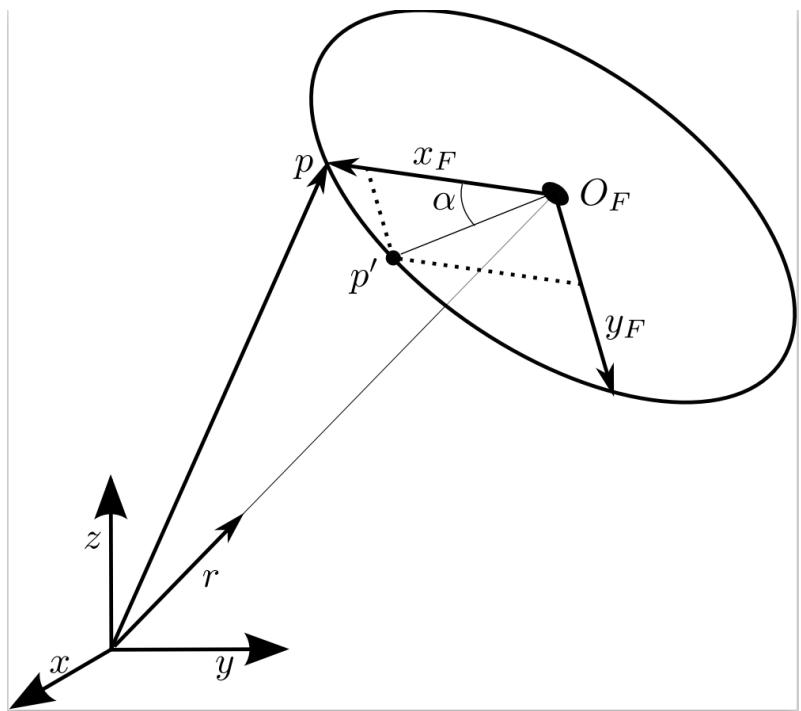
$$\mathbf{p}' = \begin{bmatrix} \mathbf{x}_F & \mathbf{y}_F & \mathbf{r} & O_F x \\ 0 & 0 & 1 & O_F y \\ 0 & 0 & 1 & O_F z \end{bmatrix} \begin{bmatrix} \cos \alpha \\ \sin \alpha \\ 0 \\ 1 \end{bmatrix} = \cos \alpha \ \mathbf{x}_F + \sin \alpha \ \mathbf{y}_F + 0 \ \mathbf{r} + \mathbf{O}_F$$

$$\mathbf{O}_F = (\mathbf{p} \cdot \mathbf{r}) \ \mathbf{r}$$

$$\mathbf{x}_F = \mathbf{p} - \mathbf{O}_F = \mathbf{p} - (\mathbf{p} \cdot \mathbf{r}) \ \mathbf{r}$$

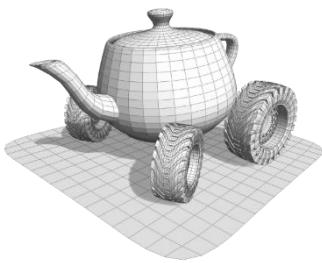
$$\mathbf{y}_F = \mathbf{r} \times \mathbf{x}_F = \mathbf{r} \times (\mathbf{p} - \mathbf{O}_F) =$$

$$= \mathbf{r} \times (\mathbf{p} - \mathbf{O}_F) = \mathbf{r} \times \mathbf{p} - \mathbf{r} \times (\mathbf{p} \cdot \mathbf{r}) \ \mathbf{r} = \mathbf{r} \times \mathbf{p}$$



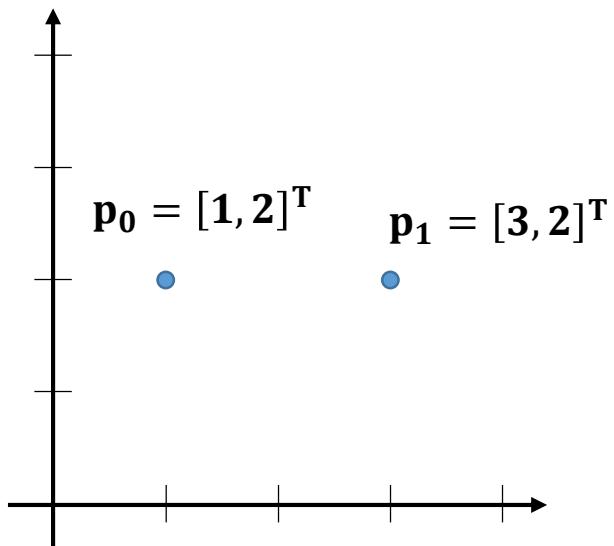
Rodriguez formula:

$$\mathbf{p}' = \cos \alpha \ \mathbf{p} + (1 - \cos \alpha)(\mathbf{p} \cdot \mathbf{r}) \mathbf{r} + \sin \alpha \ (\mathbf{r} \times \mathbf{p})$$

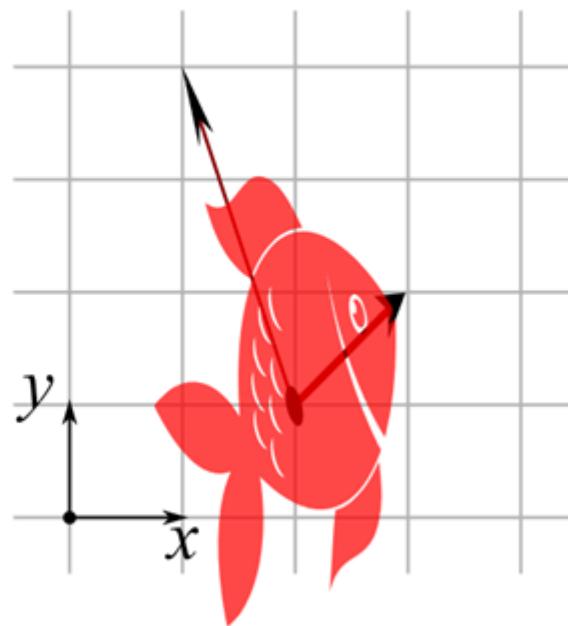
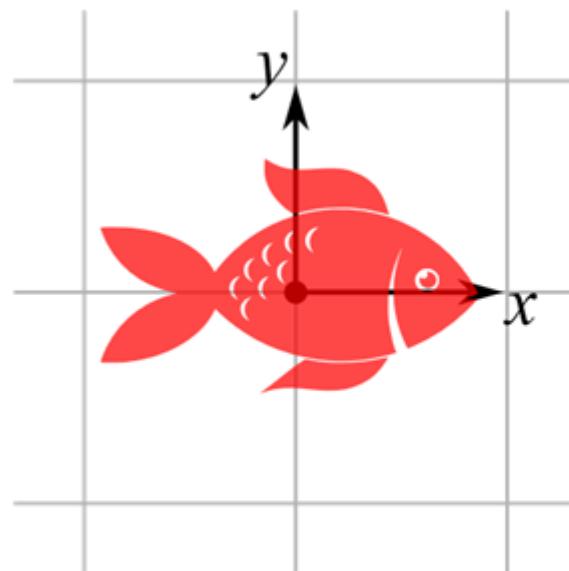
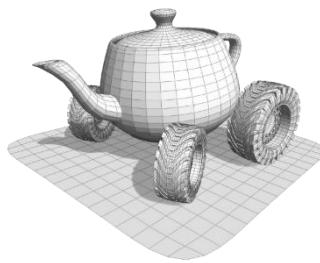


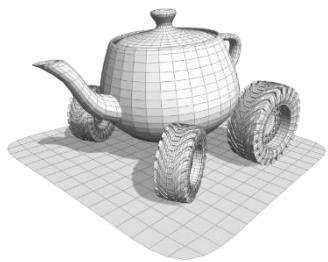
Exercise

- Find a **translation** matrix that transforms p_0 into p_1
 - How many exist?
- Find a **scaling** matrix that transforms p_0 into p_1
 - How many exits?
- Find a **rototranslation** matrix that transforms p_0 into p_1
 - How many exists?



Exercise





A sinistra è raffigurato l'oggetto PesceRosso nel suo sistema di riferimento (l'asse Z, omesso, è ortogonale agli altri due).

A destra, il pesce trasformato nello spazio Mondo (cioè nel frame canonico).

Guardando il disegno, ricostruisci la matrice che trasforma il pesce dal frame locale allo spazio mondo M , oppure (a tua scelta) la sua inversa M^{-1}

$$M^e = \begin{pmatrix} \quad & \quad & 0 & \quad \\ \quad & \quad & 0 & \quad \\ 0 & 0 & 1 & 0 \\ \quad & \quad & \quad & \quad \end{pmatrix}$$

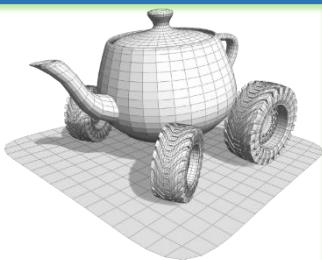
Descrivi le coordinate in **spazio mondo**...

- ...dell'origine del frame locale del pesce:
(, ,)
- ...della posizione della scaglia del pesce che (nel suo spazio) è **il punto** di coordinate (0,6 , 0,1 , 0,2):
(, ,)
- ...della differenza fra le due punte della pinna caudale del pesce, che (nel suo spazio) è **il vettore** di coordinate (0 , 0,5 , 0):
(, ,)

Descrivi come trovare le coordinate, in **spazio pesce**, una bollicina d'aria la cui posizione (in spazio mondo) è (1,2 , 0)

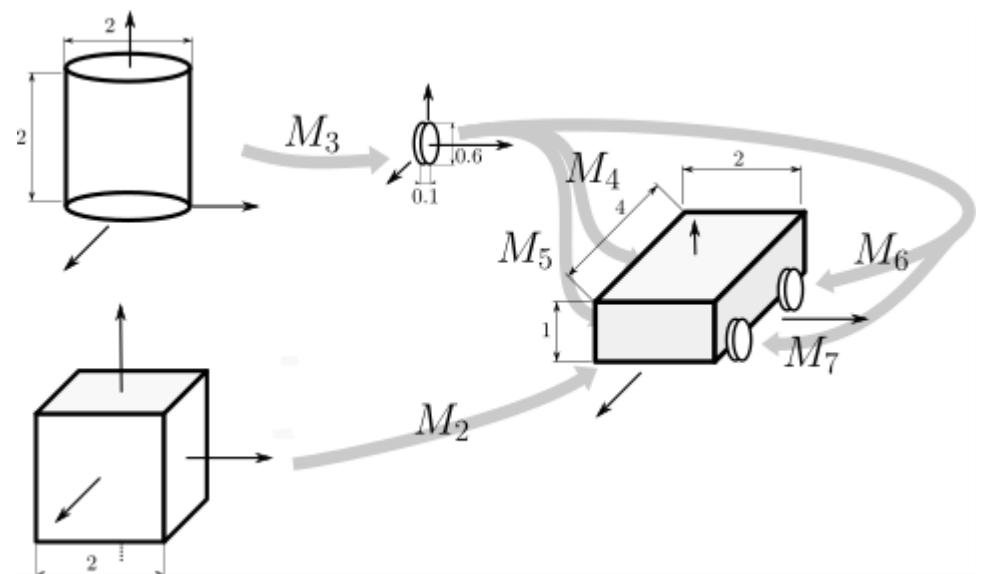
* (, ,)

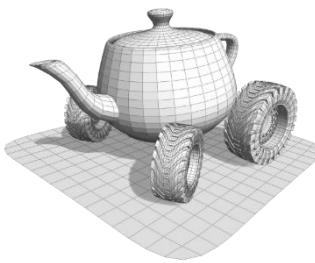
Il conto che hai completato qui sopra produce la risposta cercata in coordinate (, ,)



Drawing a «car»

- Goal: to show a car by drawing transformed cylinders and boxes
- Roles of transformations shown in the picture
 - M_2 scale and translate the cube
 - M_3 rotate and scale the cylinder
 - M_4, M_5, M_6, M_7 translate the scaled cylinders to be the wheels





Rotations with Quaternions

- Quaternions are an *extension of the complex numbers* which allow to efficiently represent rotations in 3D
- Note: complex number already do it in 2D!

$$\mathbf{p} = p_x + i p_y$$

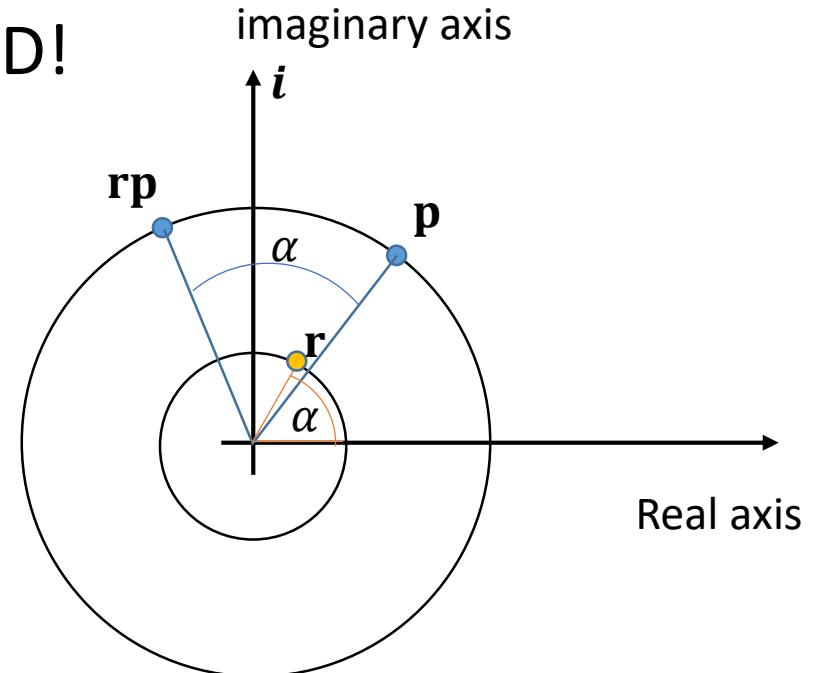
$$\mathbf{r} = \cos\alpha + i \sin(\alpha)$$

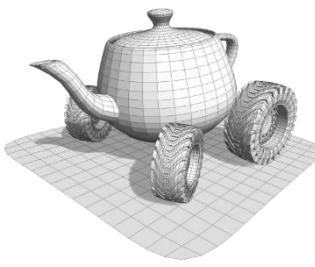
$$(\cos\alpha + i \sin\alpha)(p_x + i p_y) =$$

$$\cos\alpha p_x + i^2 \sin\alpha p_y + i(\sin\alpha p_x + \cos\alpha p_y) =$$

$$i^2 = -1$$

$$\cos\alpha p_x - \sin\alpha p_y + i(\sin\alpha p_x + \cos\alpha p_y)$$





Quaternions (definition) (1/2)

- A quaternion is a quadruple made with one real number and 3 imaginary coefficients

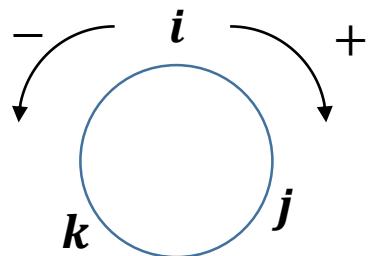
$$q = w + i x + j y + k z$$

$$ii = jj = kk = ijk = -1$$

$$ij = k, \quad ji = -k$$

$$jk = i, \quad kj = -i$$

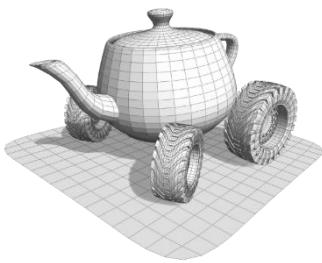
$$ki = j, \quad ik = -j$$



mnemonic rule:
clockwise-> positive
counterCW-> negative

- It is usually written as:

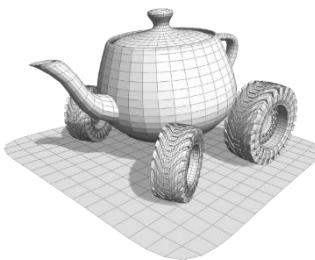
$$q = (w, x, y, z) = (s, v), s \in \mathbb{R}, v \in \mathbb{R}^3$$



Quaternions (operations) (2/2)

$$\mathbf{q} = (w, \mathbf{x}, \mathbf{y}, \mathbf{z}) = (s, \mathbf{v}), s \in \mathbb{R}, \mathbf{v} \in \mathbb{R}^3$$

- sum of two quaternions: $\mathbf{q}_1 + \mathbf{q}_2 = (s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2)$
- product of two quaternions: $\mathbf{q}_1 \mathbf{q}_2 = (s_1 s_2 - \mathbf{v}_1 \mathbf{v}_2, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$
- Conjugate of a quaternion: $\bar{\mathbf{q}} = (s, -\mathbf{v})$
- norm of a quaternion: $|\mathbf{q}| = \sqrt{\bar{\mathbf{q}} \mathbf{q}} = \sqrt{s^2 + \mathbf{v}^2}$
- Inverse of a quaternion: $\mathbf{q}^{-1} = \frac{\bar{\mathbf{q}}}{|\mathbf{q}|^2}$



Quaternions

- *Unit length* quaternions represent axis-angle rotation
- Given an angle θ and a unit vector \boldsymbol{v} , the unit quaternion

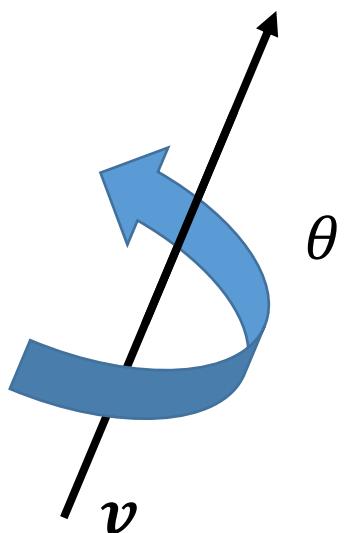
Are we sure it is a unit quaternion?

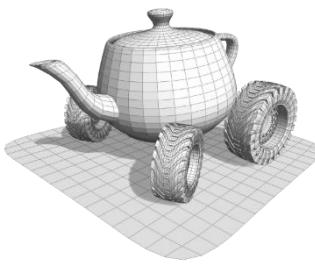
$$\mathbf{q} = \left(\cos \frac{\theta}{2}, \sin \frac{\theta}{2} \boldsymbol{v} \right) \quad \mathbf{q} = \\ \left(\cos \frac{\theta}{2}, \sin \frac{\theta}{2} \boldsymbol{v} \right)$$

Parameters of axis-angle rotations

represents the rotation around the vector \boldsymbol{v} by θ

- How is it done? How is a quaternion “applied” to a 3D point?





Rotating Pure Quaternions

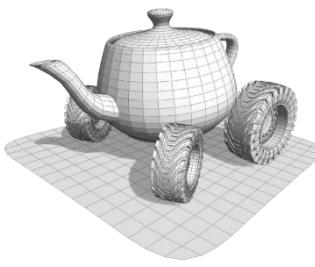
- A **pure quaternion** is a quaternion which real part is 0

$$\mathbf{p} = (0, \mathbf{v}_p)$$

- If we interpret quaternion \mathbf{p} as a point (that is, the origin translated by \mathbf{v}_p), and \mathbf{q} is the unit quaternion $\mathbf{q} = (\cos \frac{\theta}{2}, \sin \frac{\theta}{2} \mathbf{v})$ then

$$\mathbf{p}' = \mathbf{q} \mathbf{p} \bar{\mathbf{q}} = (\cos \frac{\theta}{2}, \sin \frac{\theta}{2} \mathbf{v}) (0, \mathbf{v}_p) (\cos \frac{\theta}{2}, -\sin \frac{\theta}{2} \mathbf{v})$$

where \mathbf{p}' equals to \mathbf{p} rotated around \mathbf{v} by θ



Conversions

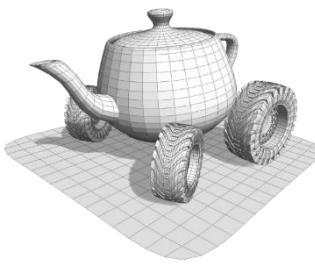
- Between a unit quaternion and axis angle rotation $\mathbf{q} = (s, \mathbf{v}_q)$?

$$\theta = 2 \operatorname{atan2}(s, \|\mathbf{v}_q\|)$$
$$\mathbf{r} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Non dovete memorizzarla

- Between a unit quaternion and a 3x3 matrix:

$$R_{\mathbf{q}} = \begin{bmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_z q_w) & 2(q_x q_z + q_w q_y) \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2(q_x^2 + q_y^2) \end{bmatrix}$$



Rotation with quaternions

- the quaternions have useful properties that make them a strong candidate to use it for rotations
 - The multiplication of two unit quaternions is a unit quaternion, that is, we can concatenate any number of rotations and we will still have a rotation

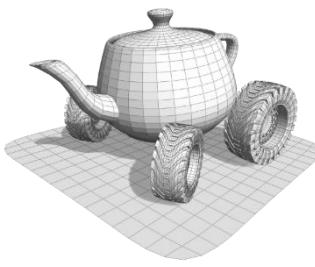
$$\mathbf{q} = \mathbf{q}_4 \mathbf{q}_3 \mathbf{q}_2 \mathbf{q}_1$$

Which means that the rotation:

$$\mathbf{q}_4 \mathbf{q}_3 \mathbf{q}_2 \mathbf{q}_1 \mathbf{p} \bar{\mathbf{q}}_1 \bar{\mathbf{q}}_2 \bar{\mathbf{q}}_3 \bar{\mathbf{q}}_4$$

Is the same as:

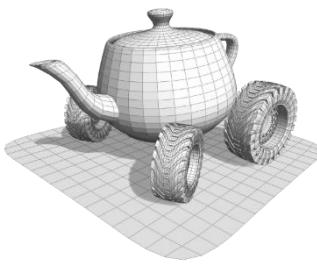
$$\mathbf{q} \mathbf{p} \bar{\mathbf{q}}$$



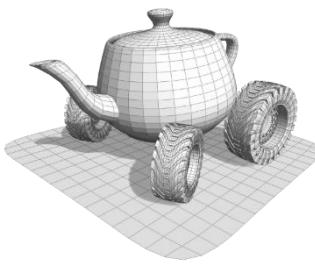
Reasons for quaternions

- Fast to multiply
 - less floating point than matrix multiplication
- More concise notation for rotation composition
 - Try to compose rotations with the Rodriguez formula
- Numerically stable for small increments
- Can convert to matrices and axis-angle
- Main reason: good for interpolating between rotations (see next..)

Interpolating Rotations



- A rotation is a (affine) transformation, what does it mean to interpolate between rotations?
- Recall: a rotation is also just an orthonormal frame centered at the origin. Interpolating between rotations means interpolating between two frames



Interpolating with matrices

- Linear interpolation does not work for matrices:

$$R_t = R_0 (1 - t) + R_1 t$$

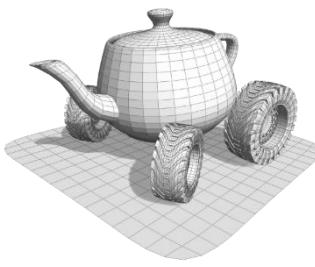
example:

$$0.5I + 0.5R_z(\pi/2) = 0.5 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + 0.5 \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ -0.5 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Average between canonical frame and
Rotation by 90° around the z axis

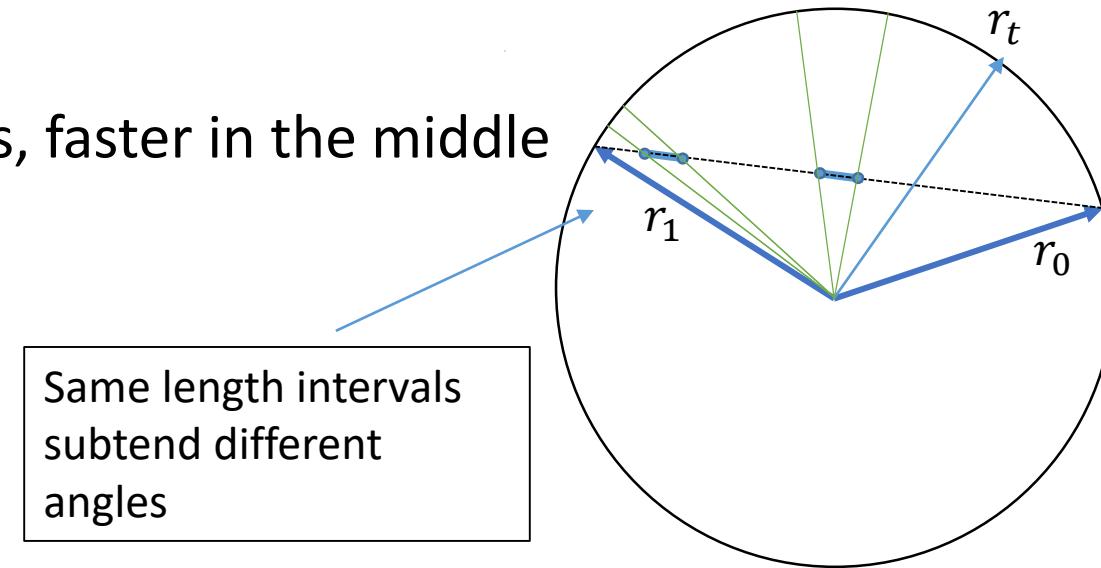
The result is **not** a rotation matrix

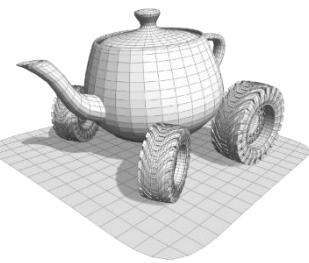
- Could we normalize the result?
 - It only works for small rotations



Interpolating with angle-axis

- Rotation angle can be interpolated linearly: $\theta_t = \theta_0(1 - t) + \theta_1 t$
- Axis can also be interpolated and normalized: $r_t = \frac{r_0(1-t)+r_1 t}{\|r_0(1-t)+r_1 t\|}$
- However, the speed won't be constant interpolation.
 - Slower near the endpoints, faster in the middle



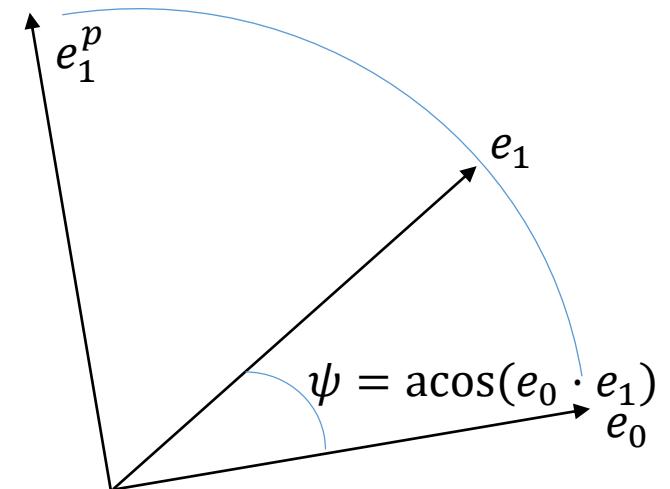


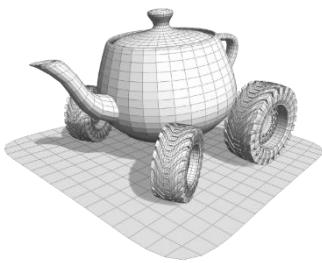
Interpolating rotations

- For constant speed: Spherical Linear Interpolation between vectors e_0 and e_1
 1. Build an orthonormal basis from e_0 and e_1
 2. Interpolate the angle between the vectors

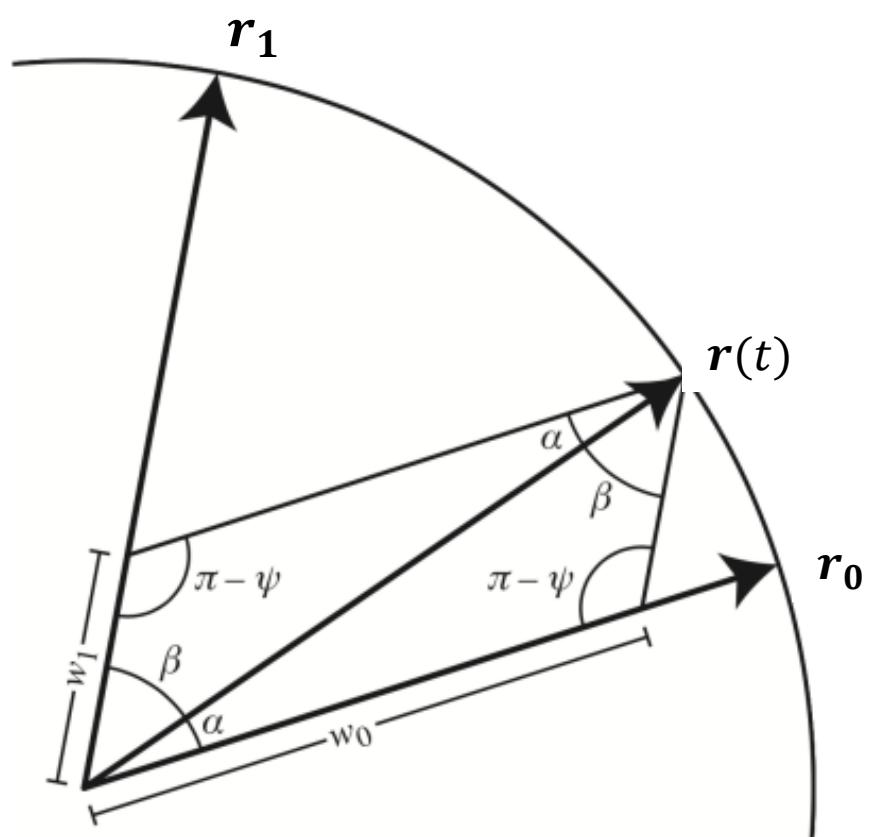
$$e_1^p = \frac{e_1 - (e_0 \cdot e_1)e_0}{|e_1 - (e_0 \cdot e_1)e_0|}$$

$$e(t) = \cos(t\psi) e_0 + \sin(t\psi) e_1^p$$





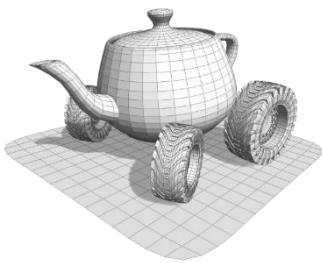
Slerp: Spherical Linear Interpolation (1/3)



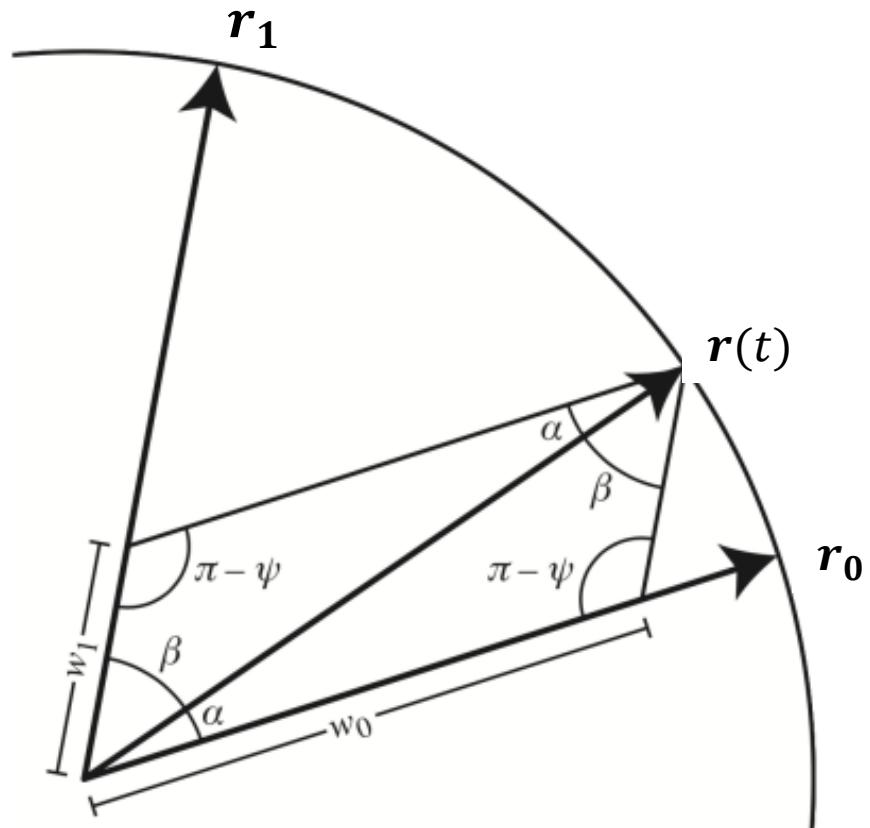
- $R = \{r_0, r_1, 0\}$ is a frame.
- The vector $r(t)$ is expressed in terms of coordinates of frame R as:

$$r(t) = 0 + \omega_0(t) r_0 + \omega_1(t) r_1$$

- What are $\omega_0(t)$ and $\omega_1(t)$?



Slerp: Spherical Linear Interpolation (2/3)



$$r(t) = \omega_0(t)r_0 + \omega_1(t)r_1$$

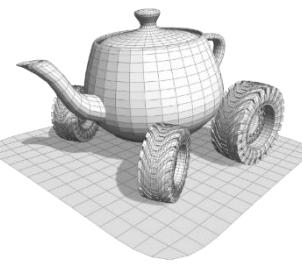
$$\psi = \arccos(r_0 \cdot r_1) = \alpha(t) + \beta(t)$$

$$\sin \psi = \frac{\sin \alpha(t)}{\omega_1} \Rightarrow \omega_1 = \frac{\sin \alpha(t)}{\sin \psi}$$

$$\sin \psi = \frac{\sin \beta(t)}{\omega_0} \Rightarrow \omega_0 = \frac{\sin \beta(t)}{\sin \psi}$$

$$\alpha(t) = t \psi \quad \beta(t) = (1 - t) \psi$$

$$r(t) = \frac{\sin t\psi}{\sin \psi} r_0 + \frac{\sin(1-t)\psi}{\sin \psi} r_1$$



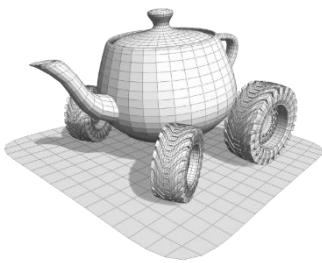
Slerp: Spherical Linear Interpolation (3/3)

$$\mathbf{r}(t) = \frac{\sin t\psi}{\sin \psi} \mathbf{r}_0 + \frac{\sin(1-t)\psi}{\sin \psi} \mathbf{r}_1$$

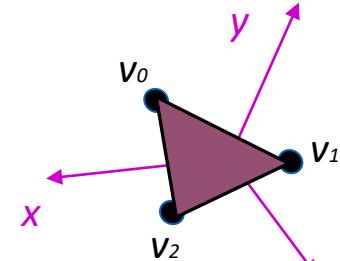
- It works in any dimension
- It can be used to directly interpolate between unit quaternions

$$\begin{aligned}\psi &= \text{acos}(\mathbf{q}_0 \cdot \mathbf{q}_1) \\ \mathbf{q}(t) &= \frac{\sin t\psi}{\sin \psi} \mathbf{q}_0 + \frac{\sin(1-t)\psi}{\sin \psi} \mathbf{q}_1\end{aligned}$$

- Traverses a great arc on the sphere of unit quaternions
- It gives uniform angular velocity about a fixed axis



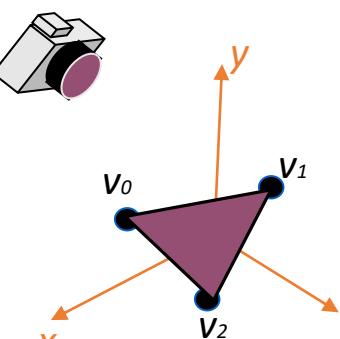
Transformations in the pipeline



Object space



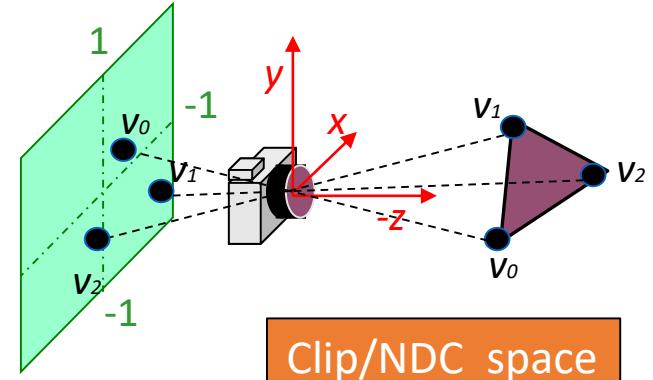
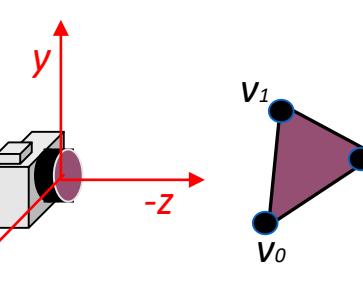
- 0) **Modelling** transform
- 1) **View** transform
- 2) **Projection** transform
- 3) **Viewport** transform



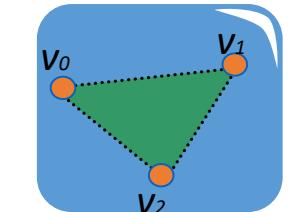
World space



View space

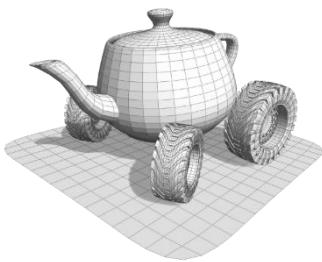


Clip/NDC space

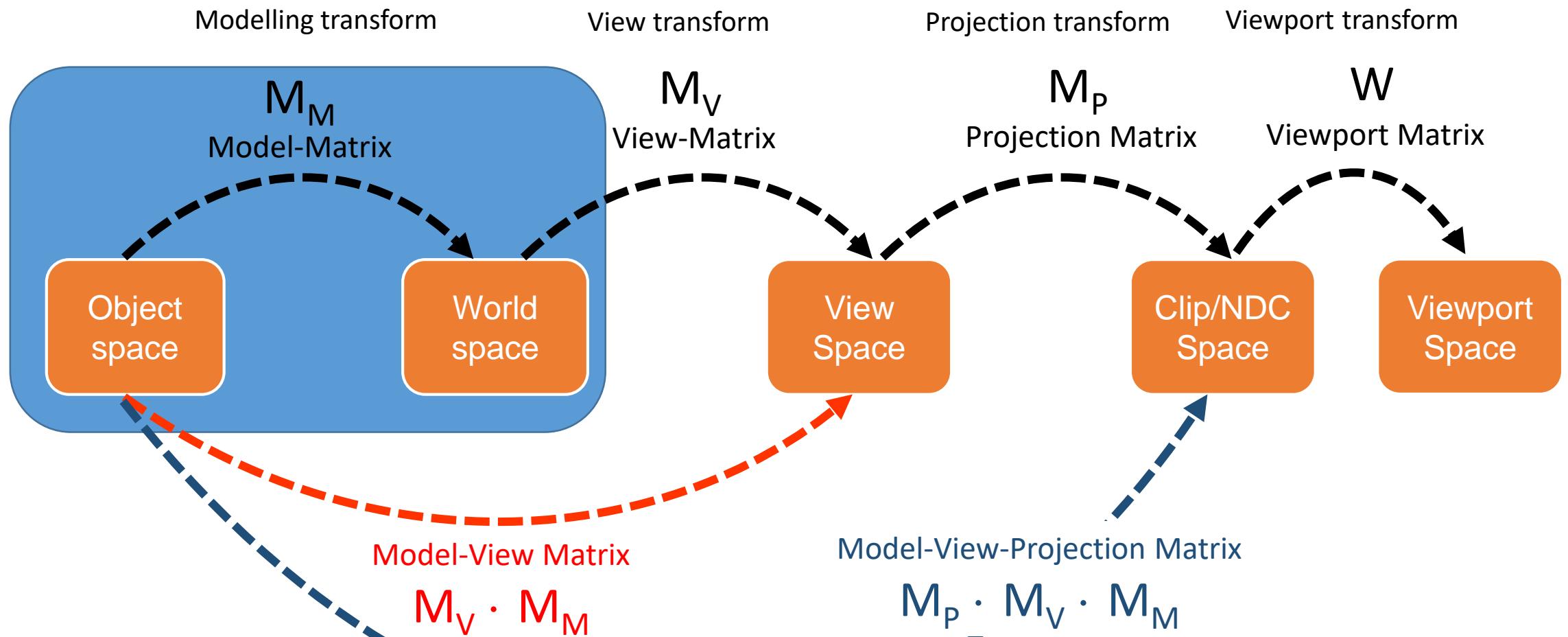


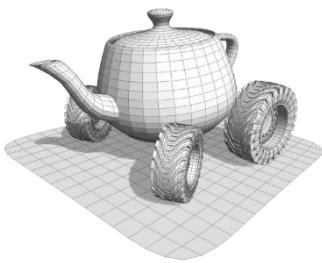
Screen space





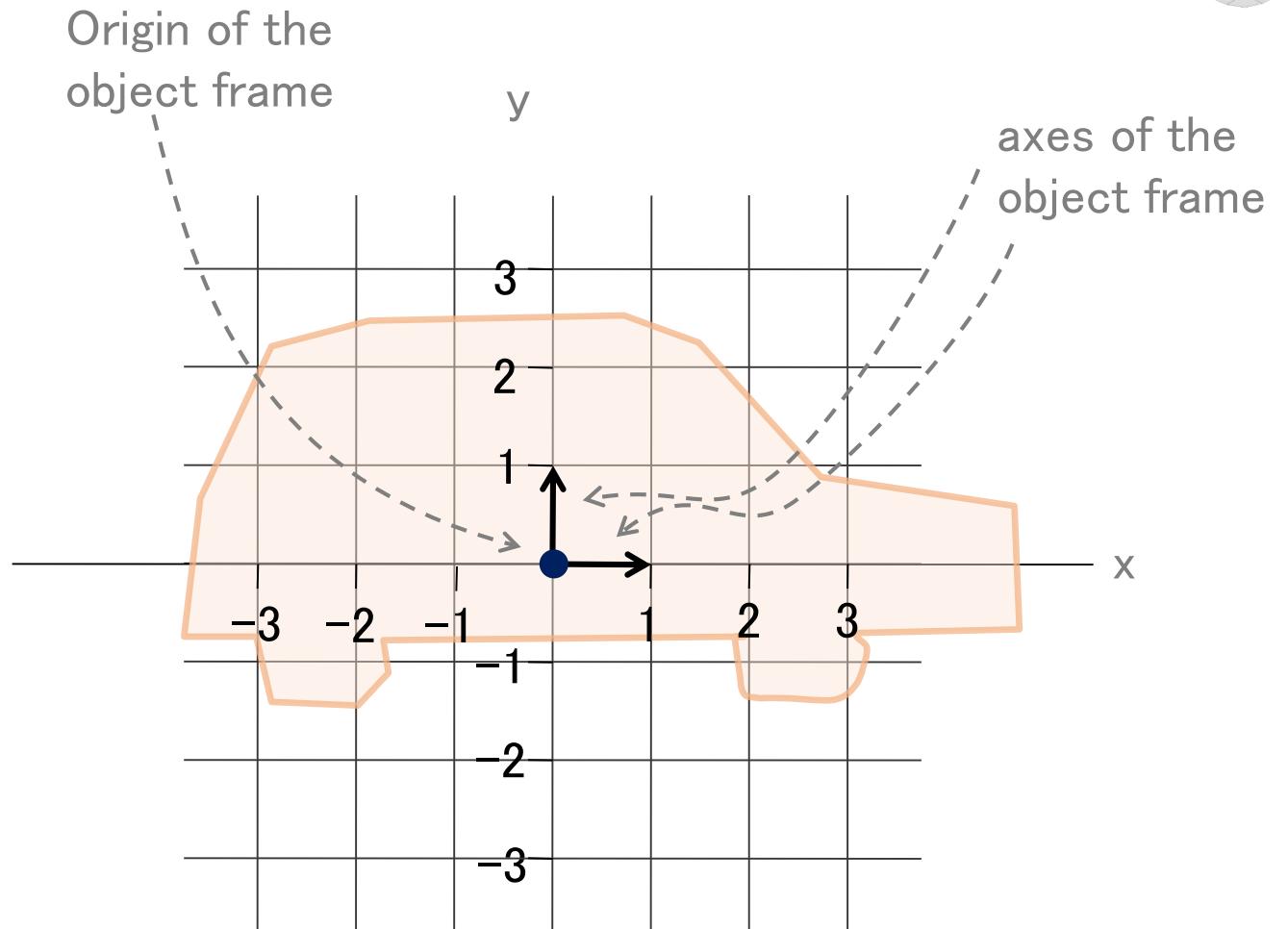
Transformations in the pipeline

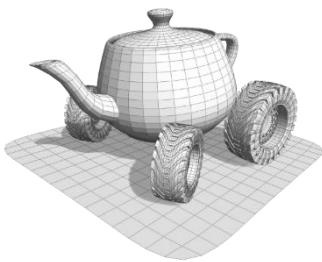




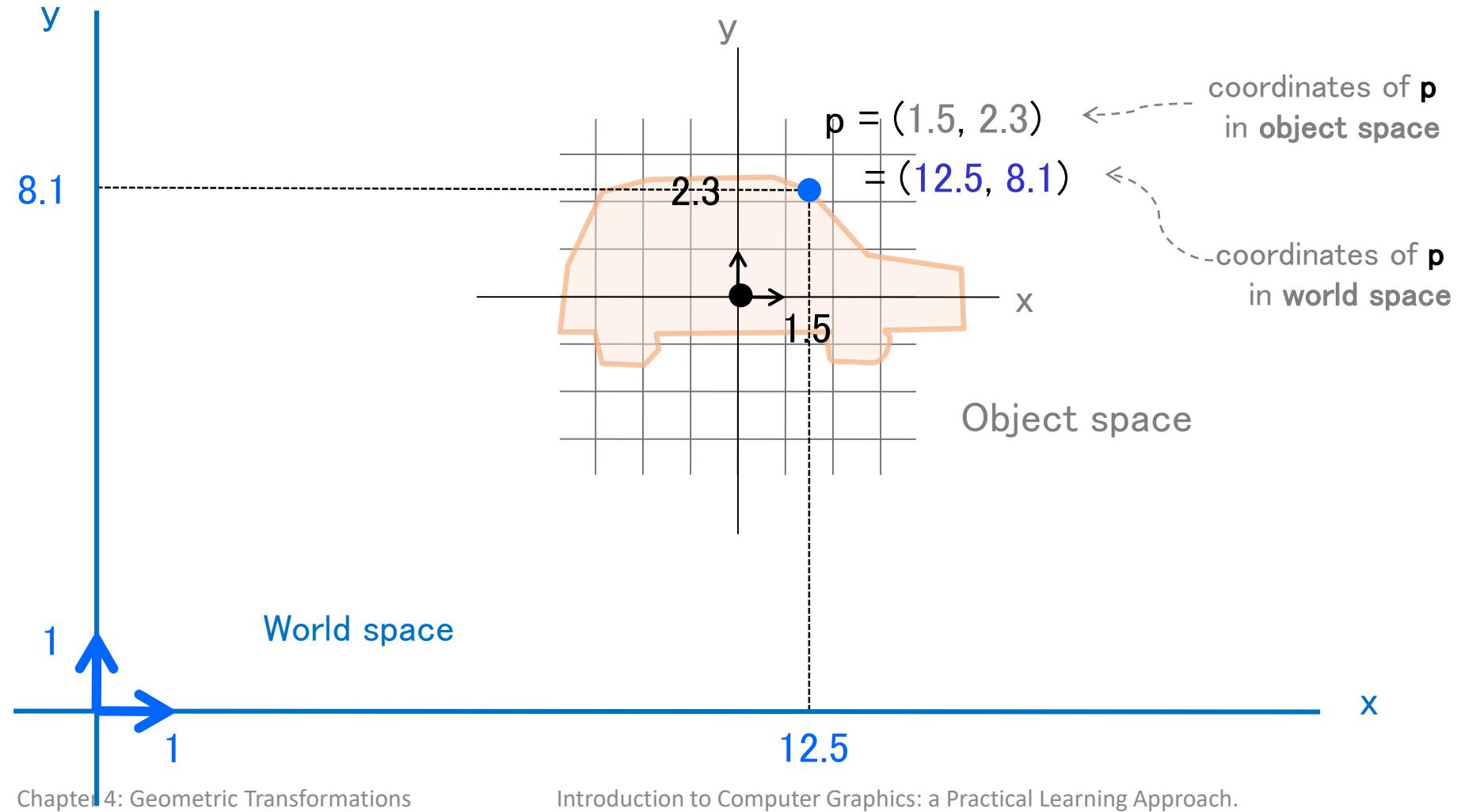
Object space

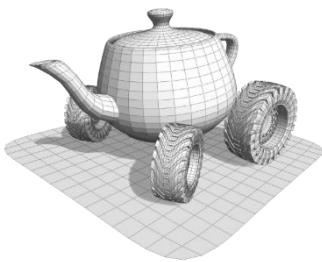
- **Object space:** space where the object is “created”
- **Object frame:** the canonical frame where the object space coordinates are expressed
- Typical choice: define the object around the origin and aligned with the axes



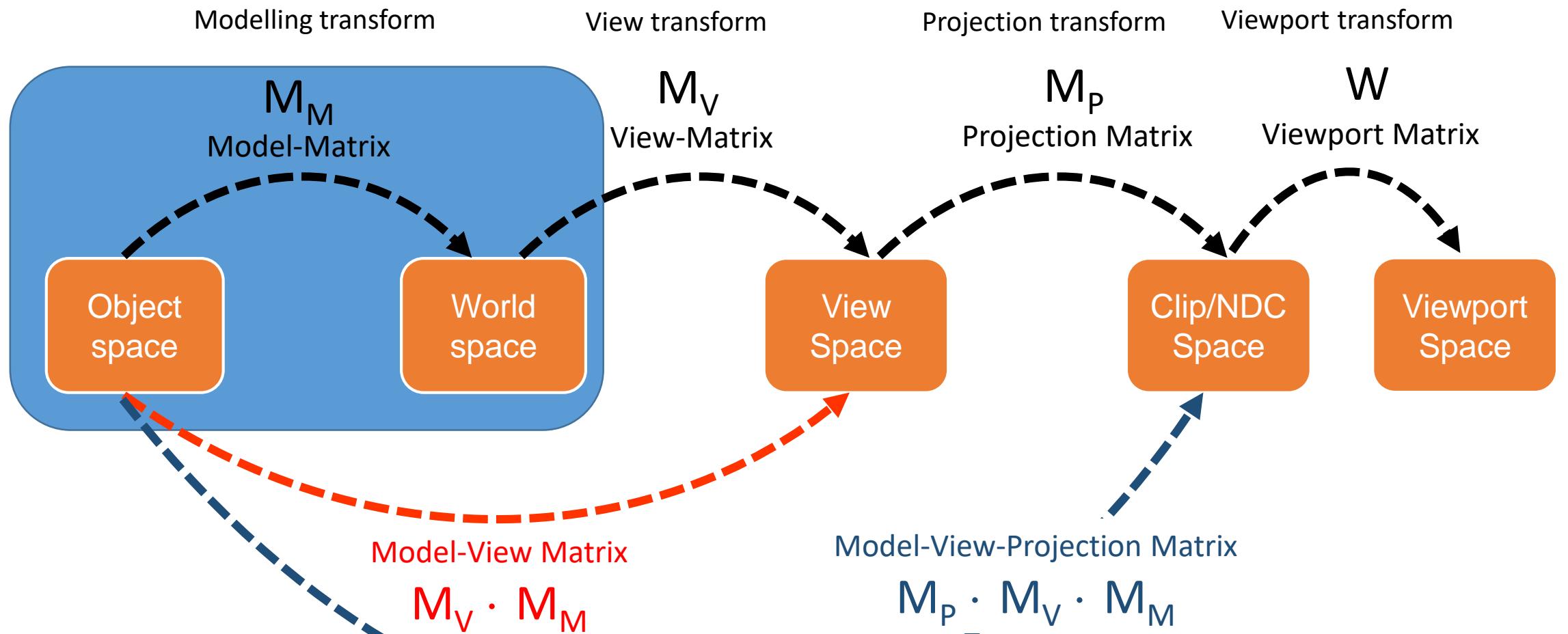


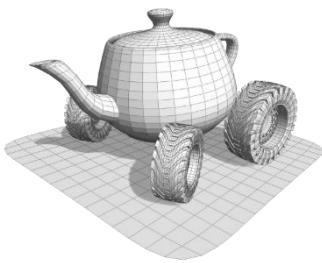
From Object Space to World Space





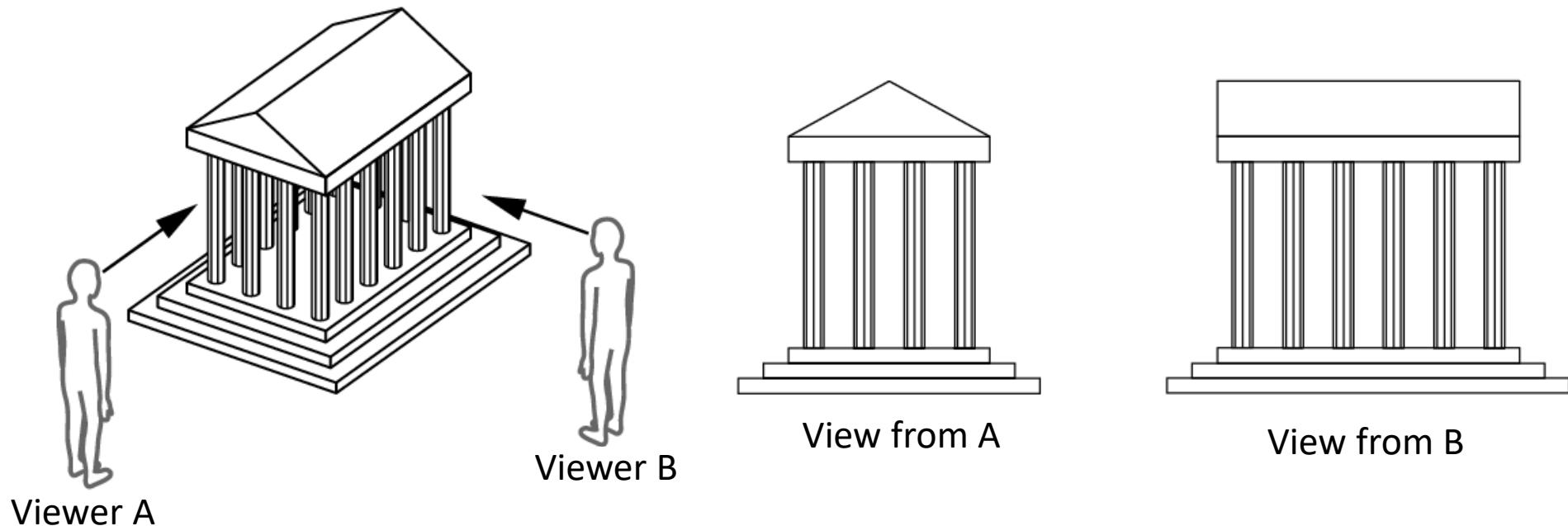
Transformations in the pipeline

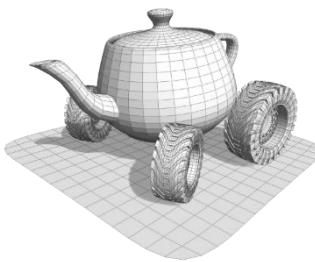




View Transform

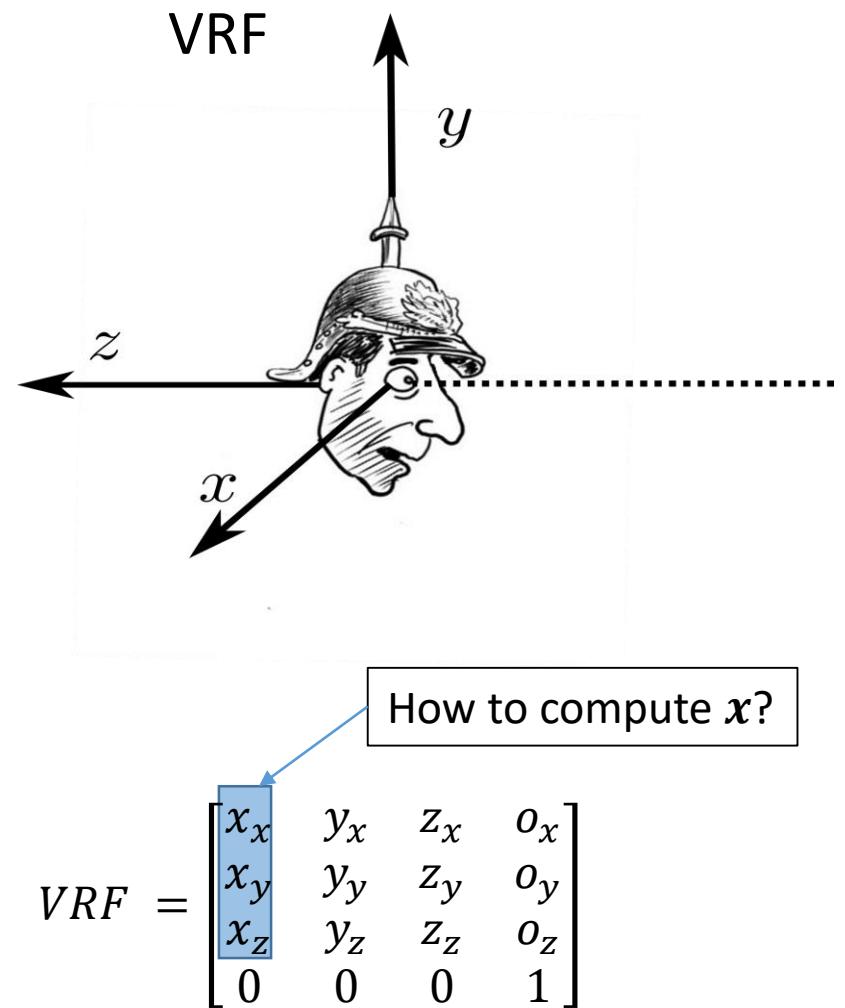
- It depends from where is the “camera” and how it is oriented

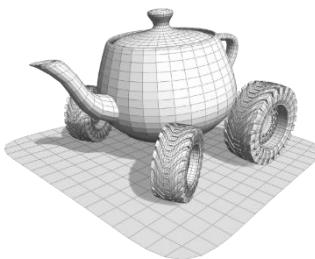




View Reference Frame

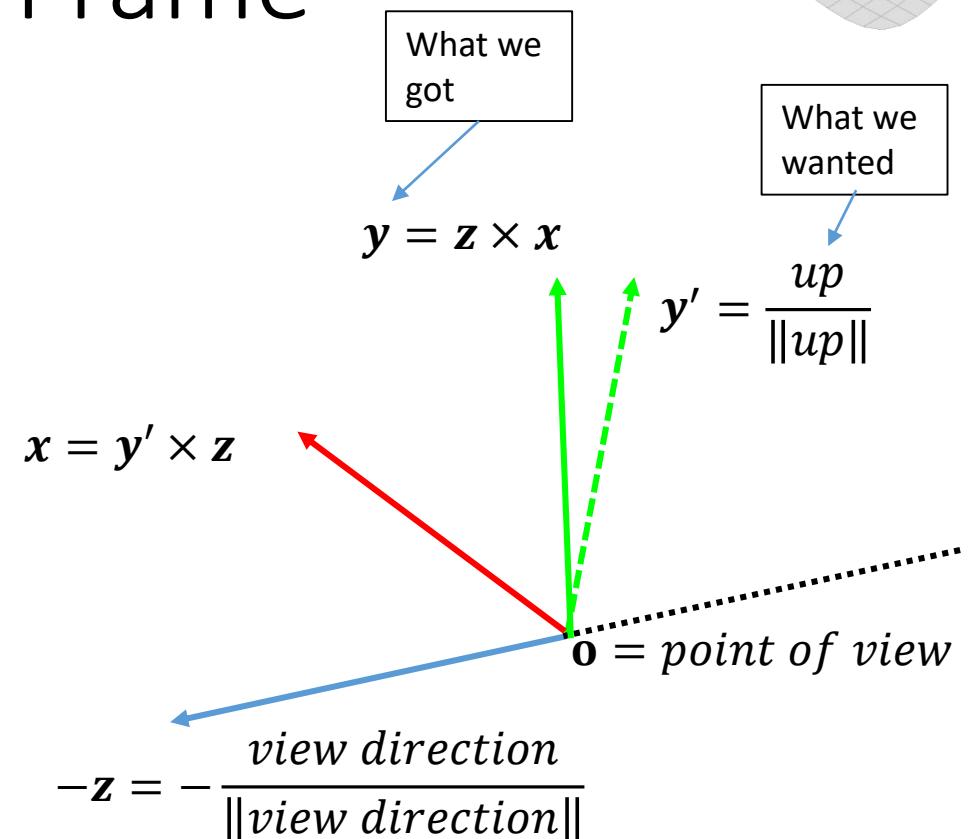
- Position and orientation of the camera is defined by a (unitary orthonormal) frame
 - The **point of view** is the origin of the frame
 - The **viewing direction** is the $-z$ direction
 - The spike of the pickelhube helmet is the **Y** direction
 - Conventionally called the **up** direction

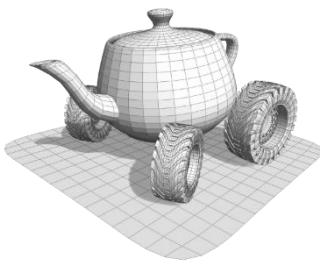




Building the View Reference Frame

- The missing axis x is obtained by means of vector product. Be careful:
 - normalize *view direction* and *up*
 - Usually the *up* input vector is more a *desiderata* and it's not perpendicular to the view direction
 - but we want a orthonormal frame! Then:
 - Compute $\mathbf{x} = \mathbf{y}' \times \mathbf{z}$, which is orthogonal to \mathbf{z}
 - Compute $\mathbf{y} = \mathbf{x} \times \mathbf{x}$
- The parameters describing the V_{RF} are called **extrinsic** camera parameters

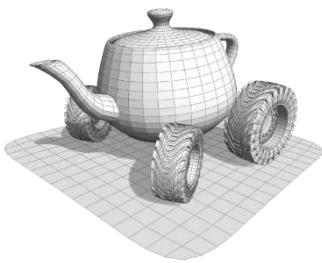




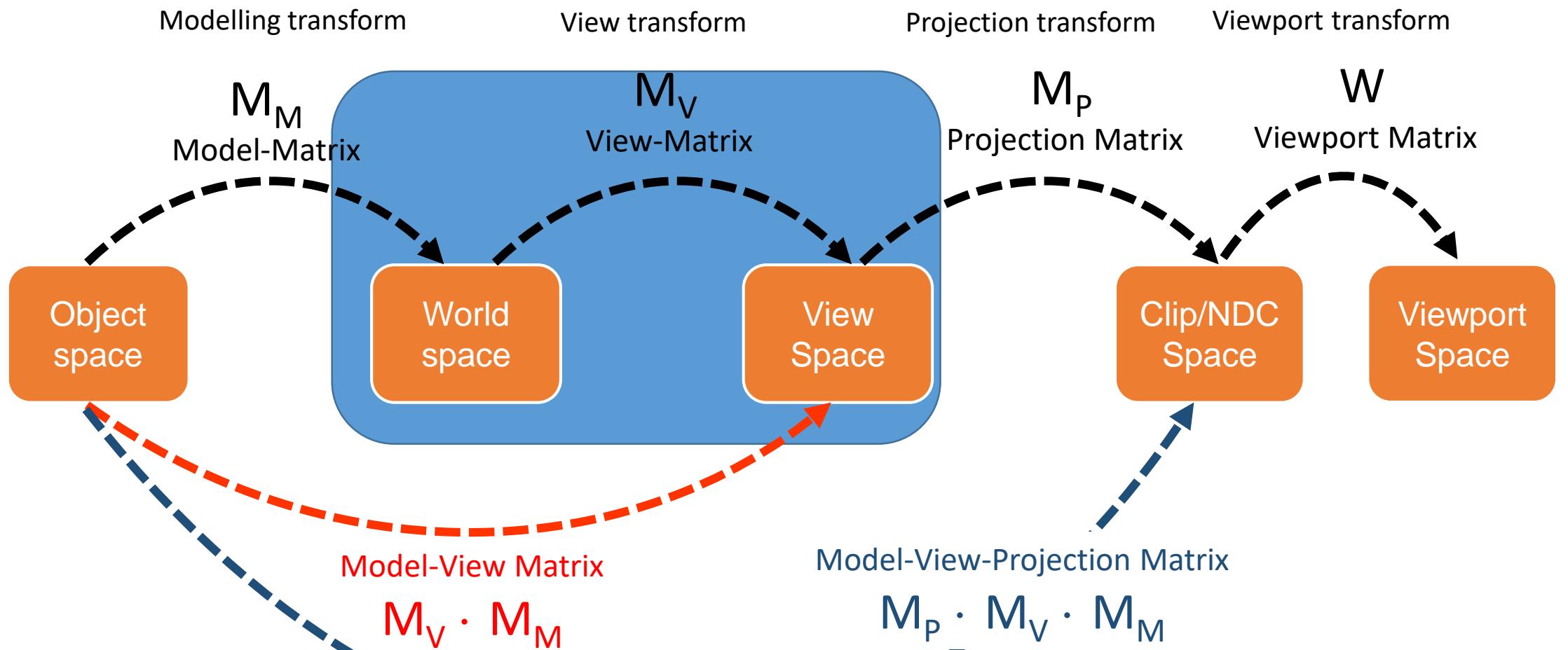
View Reference Frame and View Transform

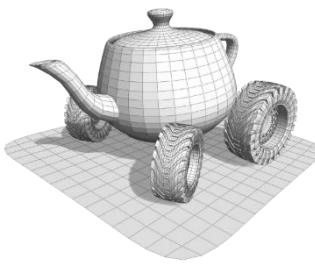
- The matrix encoding the View Reference Frame transforms the coordinates from the View Reference Frame to world space (that is, with coordinates in the canonical frame)
- We want the opposite

$$M_v = V_{RF}^{-1}$$



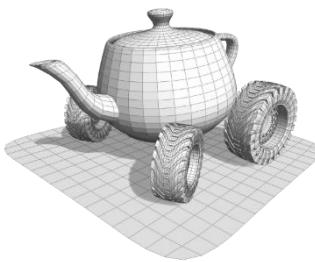
Transformations in the pipeline





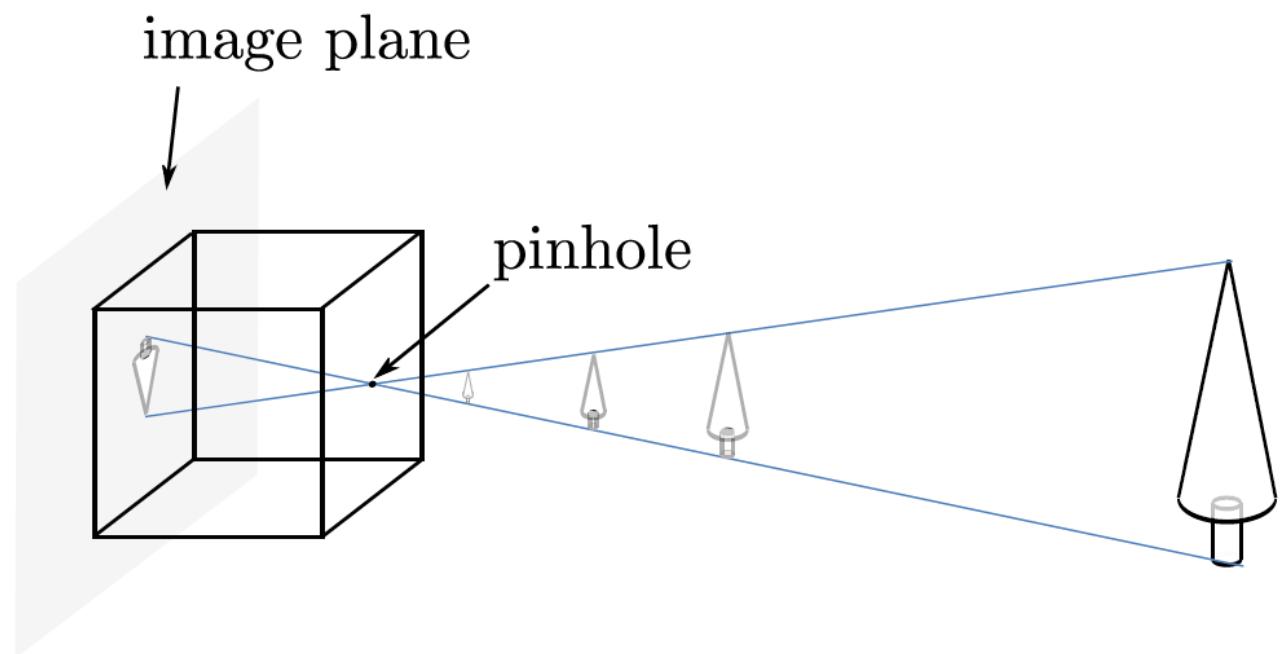
Projection Transform

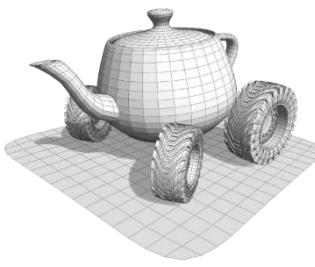
- The View Reference Frame «places» the camera into the scene, but it does not say «what» type of camera
 - It's a fisheye camera (large grandangular)
 - It's a telescope?
 - It's a microscope?
 - ...
- The role of the camera is to map the world it sees into a plane. The way it does it depends on its **intrinsics** parameters



Pinhole camera model

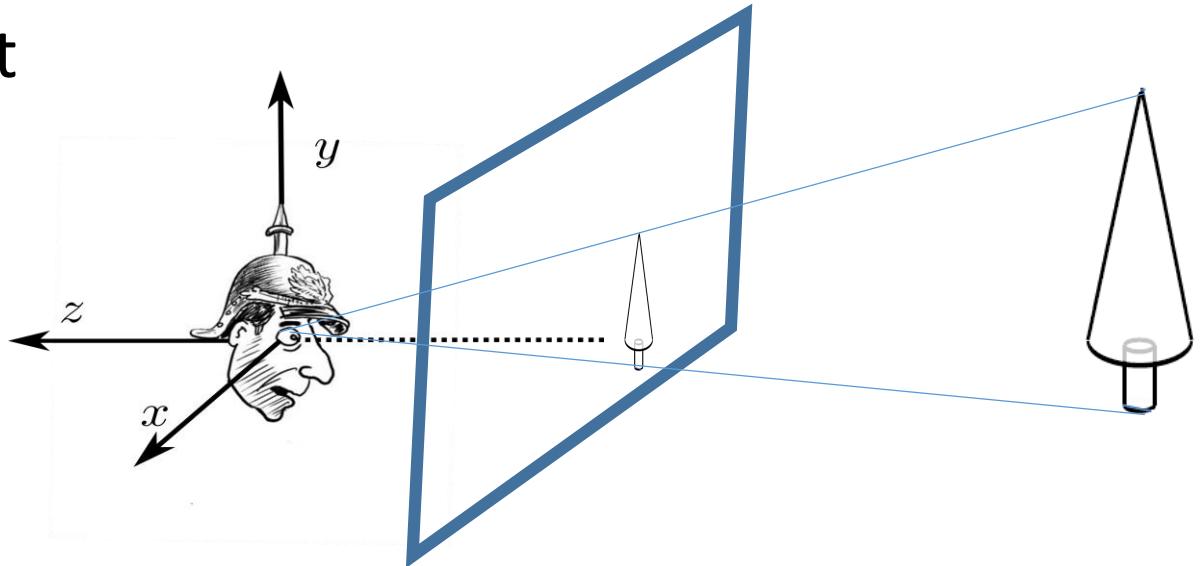
- **Pinhole camera:** a closed box with a «infinitely» small hole through which light enters
- Each entering direction maps to a point in the opposite side
- Image is formed upside down (and left to right)

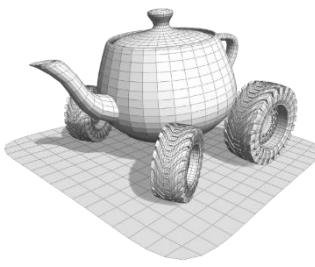




eye-in-front-of-the-window camera model

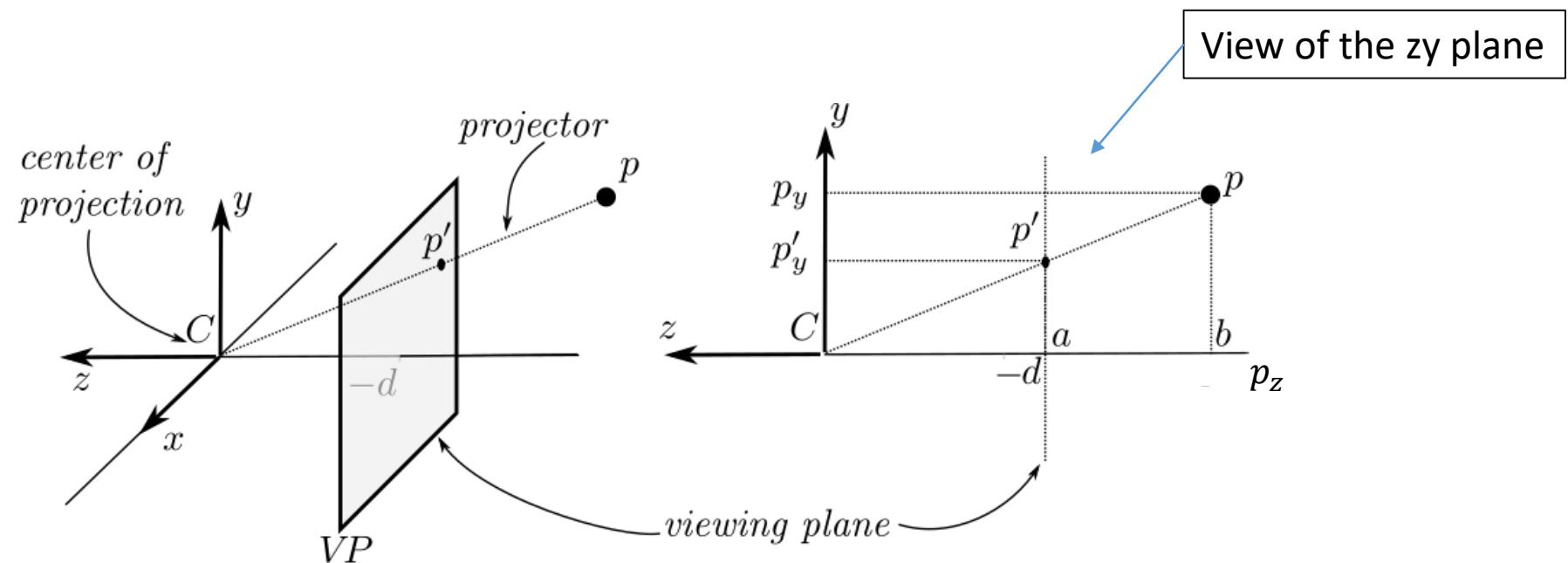
- We stand in front of a window with one eye closed
- Same as the pinhole camera but no reversing of the image

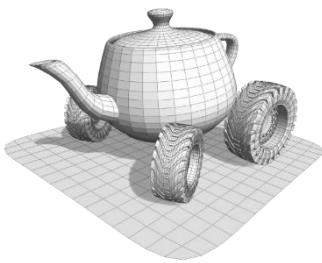




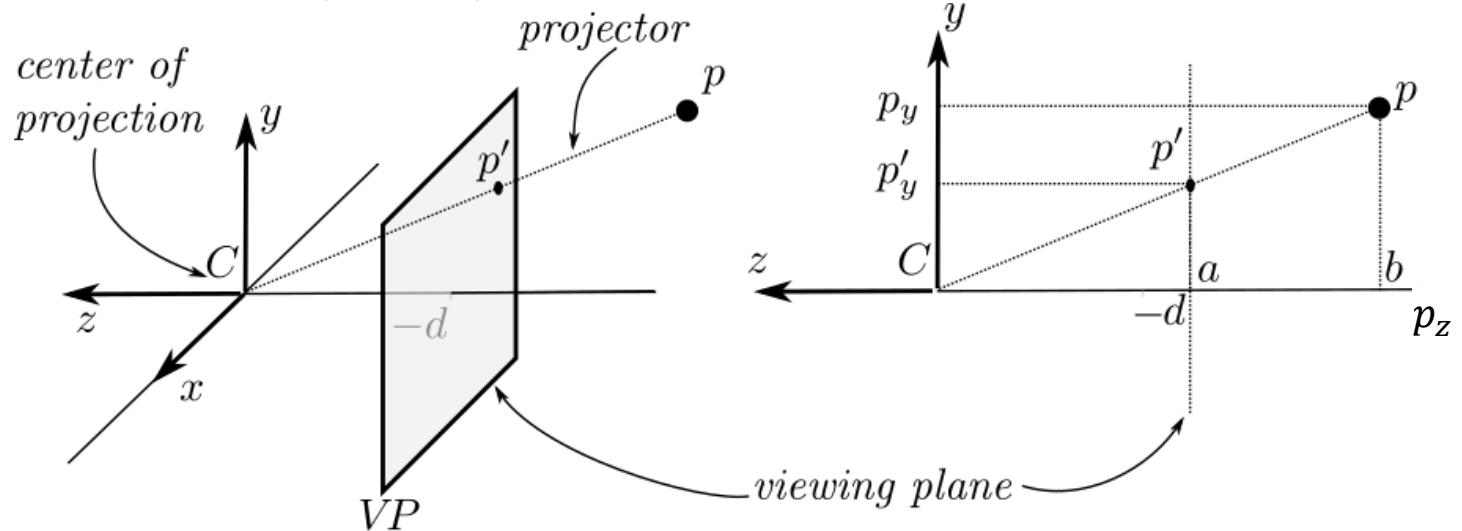
Perspective projection

- The line passing through a point \mathbf{p} and the center of projection is called projector
- The point \mathbf{p} is projected into the plane of the window, called **projective plane**
- What are the coordinates of such projection?





Perspective projection

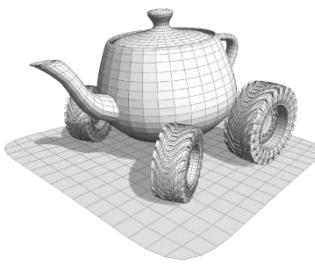


- Let us see the y coordinate. Triangles Cap' and Cbp are **similar** (that is, same internal angles), so the ratio of their side is the same for all sides:

$$p'_y : d = p_y : -p_z \Rightarrow p'_y = -\frac{p_y}{p_z/d}$$

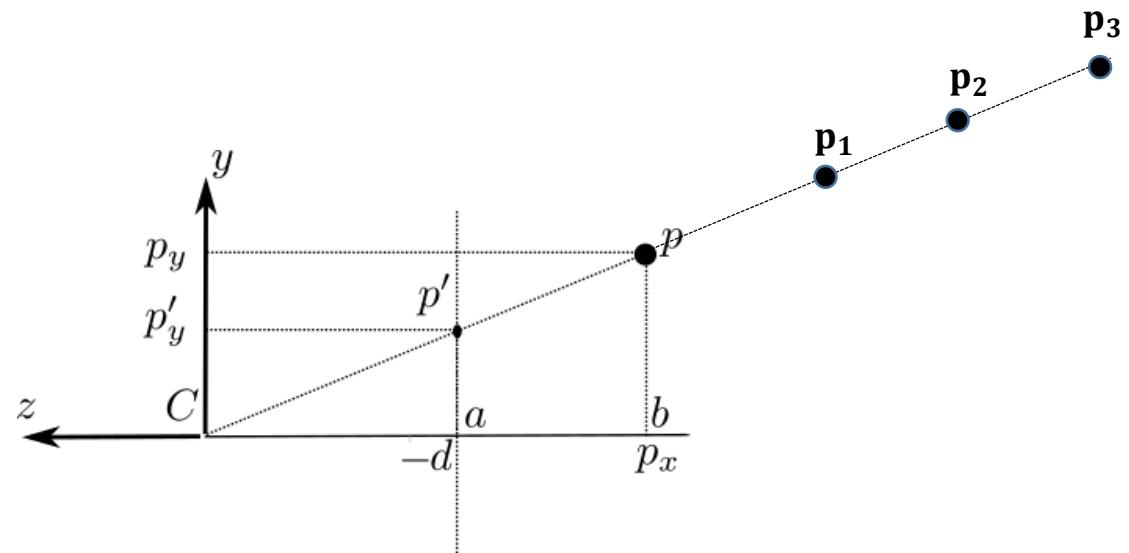
$$p'_x = -\frac{p_x}{p_z/d}$$

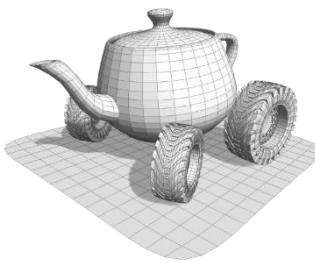
Same thing for the x coordinate



Perspective projection

- Note: all points on the line from C to \mathbf{p} project onto \mathbf{p}'
- Which means that \mathbf{p}_1 , \mathbf{p}_2 , and \mathbf{p}_3 are all the same point \mathbf{p}' in the projective plane
- How can we express this equivalency?



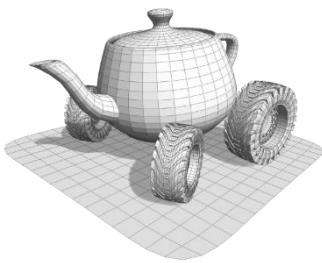


Homogeneous coordinates (again)

- So far we expressed points in homogeneous coordinates having **1** as their last component
- In this case we say that the point is in its **canonical form**
- However, in homogenous coordinates:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda x \\ \lambda y \\ \lambda z \\ \lambda \end{bmatrix} \quad \forall \lambda \neq 0$$

- And the canonical form is obtained by dividing by the last component



Homogeneous coordinates:example

$$\begin{bmatrix} 20 \\ 10 \\ 30 \\ 10 \end{bmatrix} \quad \begin{bmatrix} 2 \\ 1 \\ 3 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 0.2 \\ 0.1 \\ 0.3 \\ 0.1 \end{bmatrix} \quad \begin{bmatrix} 4 \\ 2 \\ 6 \\ 2 \end{bmatrix} \dots$$

Canonical
form

Some of the
homogeneous coordinates of
the **point**
with Cartesian coordinates

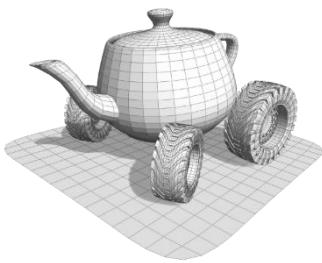
$$\begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \end{bmatrix}$$

The unique
homogeneous coordinates of
vector
with Cartesian coordinates

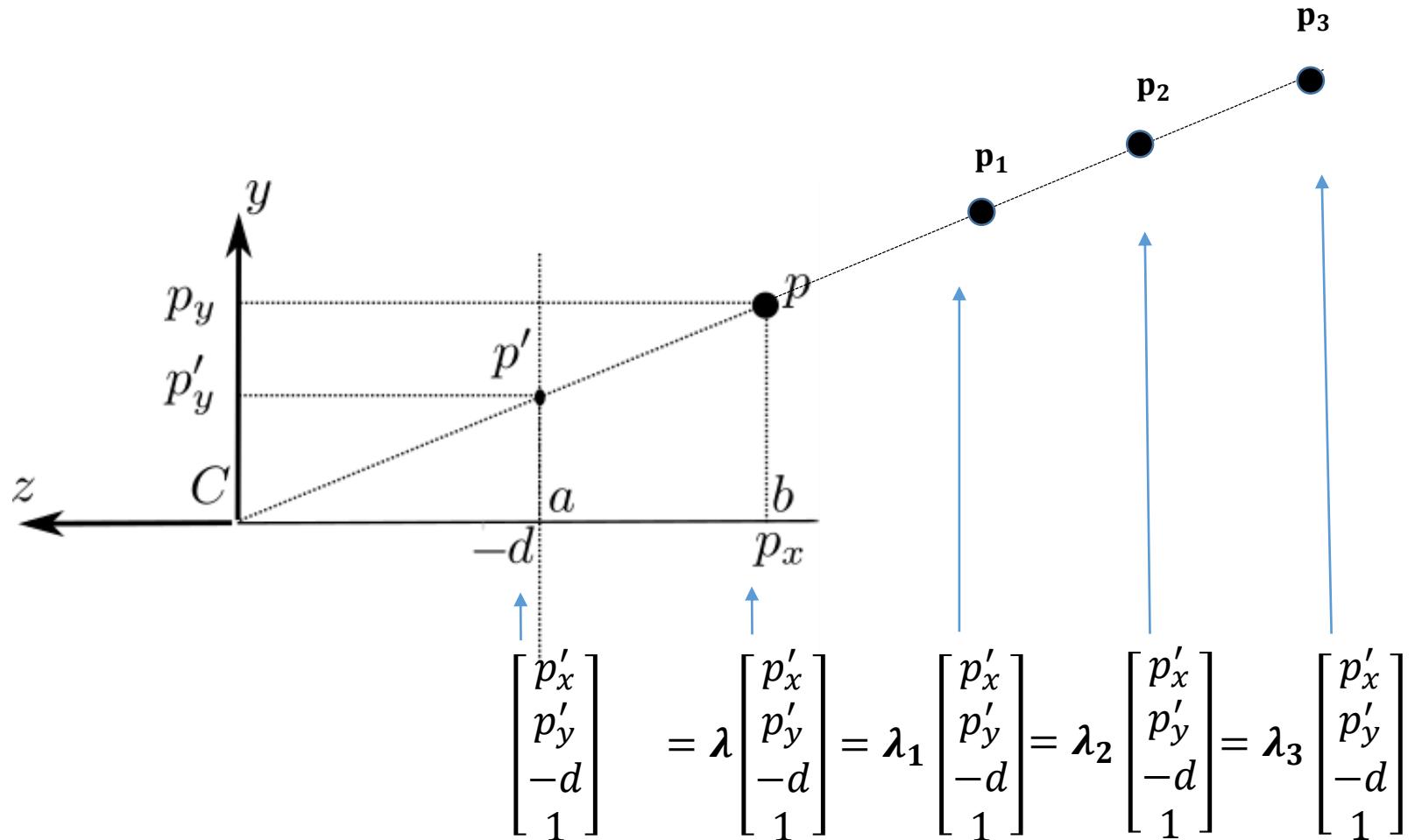
$$\begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

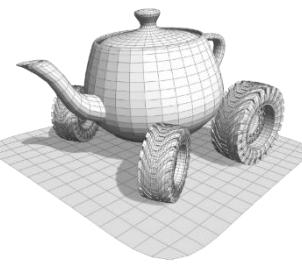
point

vector



Equivalent homogenous coordinates





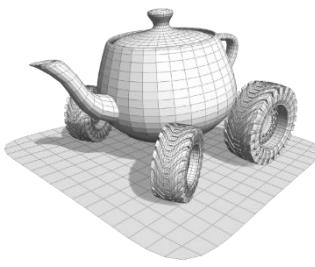
Perspective projection

- In homogenous coordinates:

$$\mathbf{p}' = \begin{bmatrix} -\frac{p_x}{p_z/d} \\ -\frac{p_y}{p_z/d} \\ -\frac{p_z}{p_z/d} \\ -d \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} -\frac{p_x}{p_z/d} \\ -\frac{p_y}{p_z/d} \\ -\frac{p_z}{p_z/d} \\ -d \\ 1 \end{bmatrix} \quad \forall \lambda \neq 0$$

- For $\lambda = -\frac{p_z}{d}$

$$\mathbf{p}' = \begin{bmatrix} -\frac{p_x}{p_z/d} \\ -\frac{p_y}{p_z/d} \\ -\frac{p_z}{p_z/d} \\ -d \\ 1 \end{bmatrix} = -\frac{\mathbf{p}_z}{d} \begin{bmatrix} -\frac{p_x}{p_z/d} \\ -\frac{p_y}{p_z/d} \\ -\frac{p_z}{p_z/d} \\ -d \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ -\mathbf{p}_z/d \\ 1 \end{bmatrix}$$



Perspective projection

- In matrix form

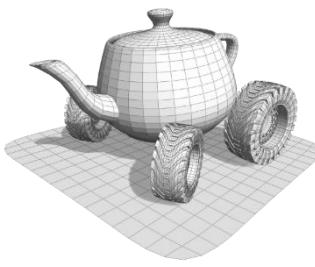
$$\text{perspective projection}$$
$$\mathbf{p}' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ -\frac{p_z}{d} \end{bmatrix} = \begin{bmatrix} -\frac{p_x}{p_z/d} \\ -\frac{p_y}{p_z/d} \\ -d \\ 1 \end{bmatrix}$$

Is it affine?

Is it invertible?

Canonical form

Perspective division



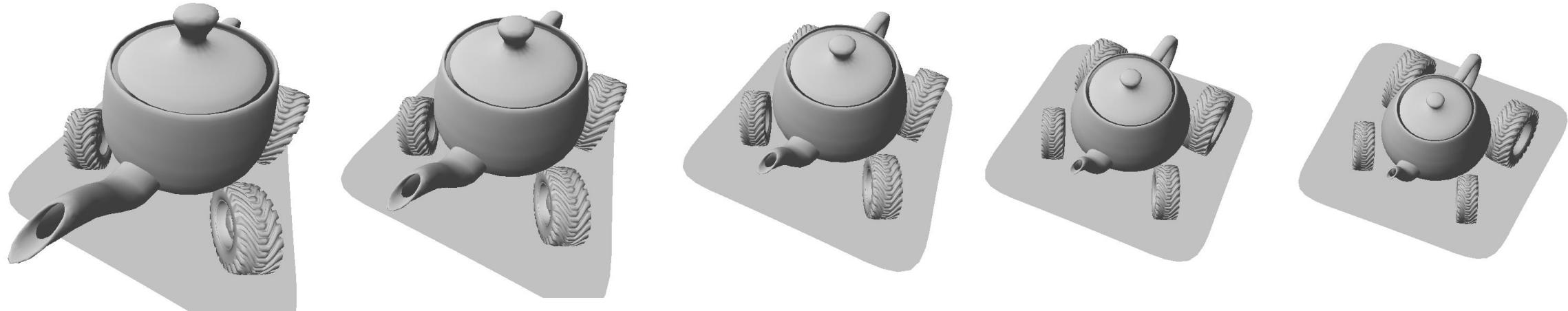
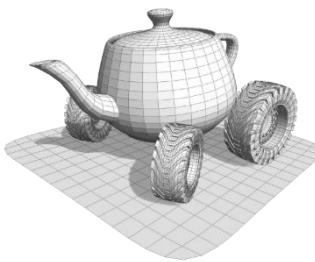
Perspective projection

- Equivalent matrix form

$$\mathbf{p}' = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} d p_x \\ d p_y \\ d p_z \\ -p_z \end{bmatrix} = \begin{bmatrix} -\frac{p_y}{p_z/d} \\ -\frac{p_y}{p_z/d} \\ -d \\ 1 \end{bmatrix}$$

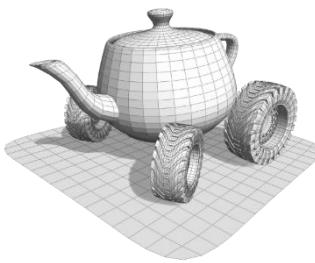
- perspective projection does not preserve length, angles, ratio of the distances.
- Perspective projection does preserve collinearity

Effect of d



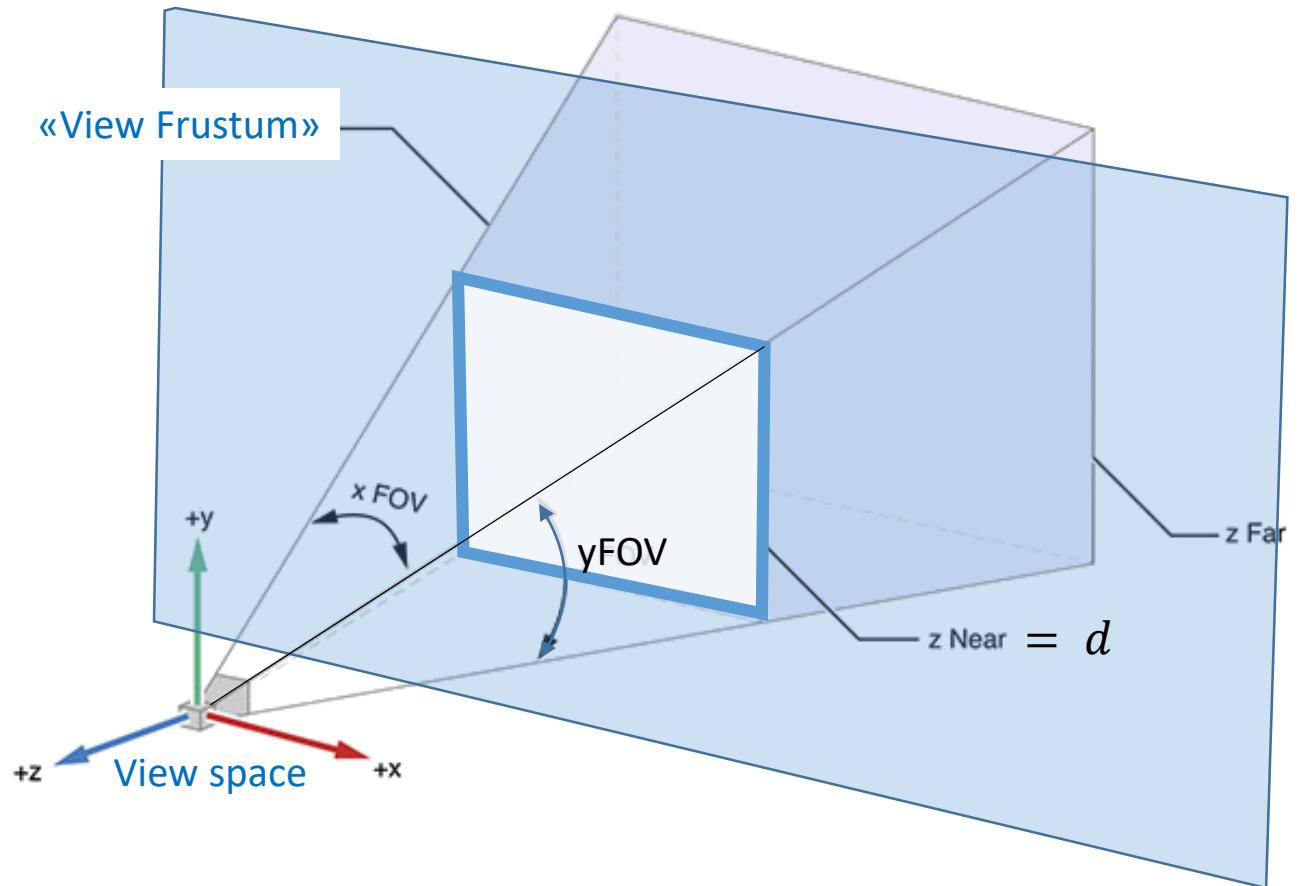
Increasing d

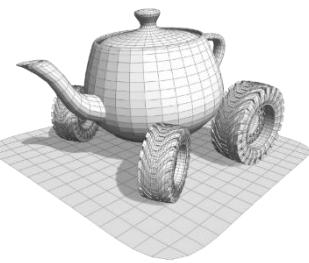
- Small values of d show the grandangular effect
- Greater values of d go towards **orthographics projection** (more later), that is, the projectors tend to become parallel



Defining the window

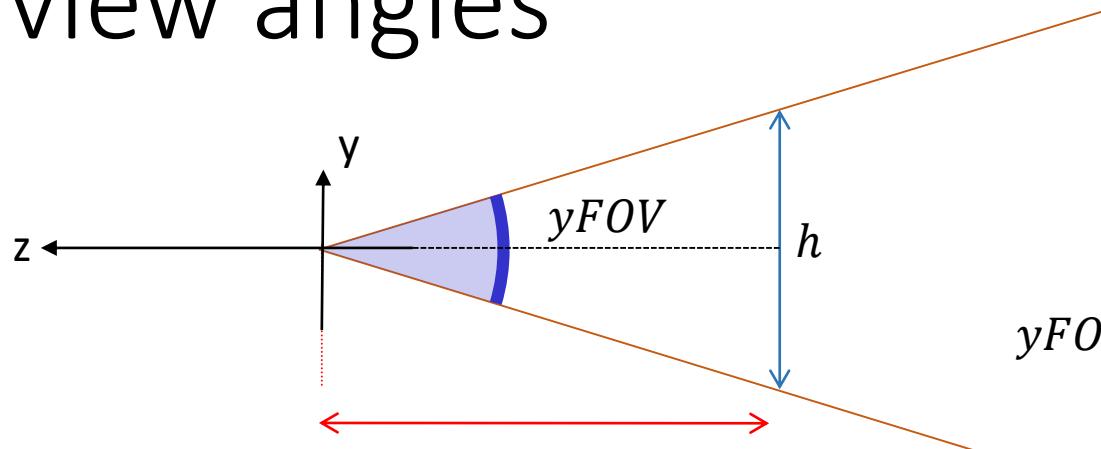
- d tell us the distance of the projective plane from the eye
- How the size of the window is defined?





Field of view angles

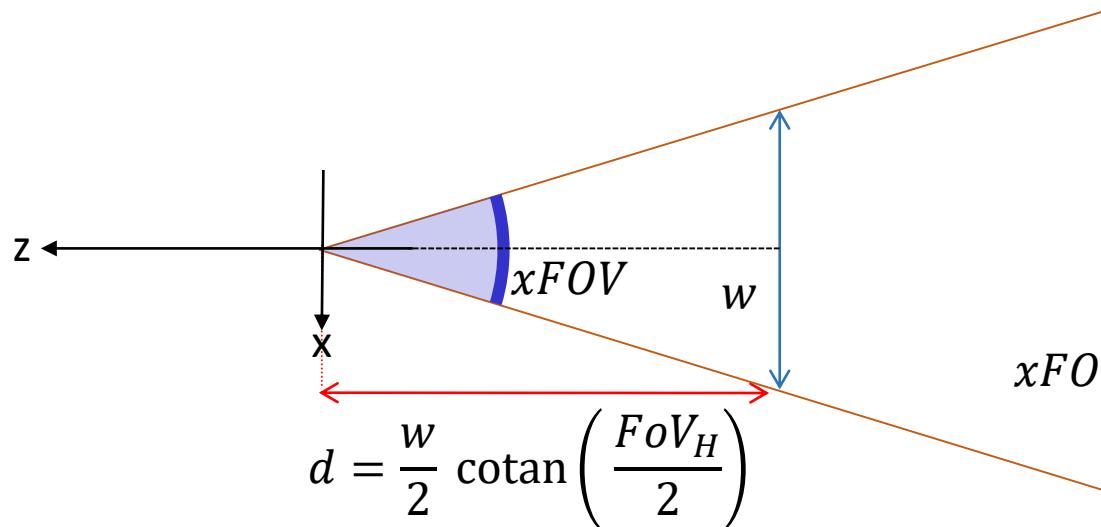
View from
the side



$$yFOV = 2 \arctan\left(\frac{h}{2d}\right)$$

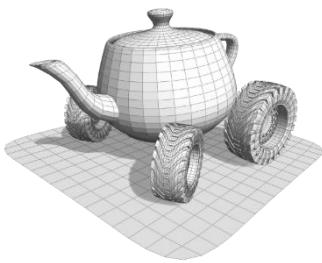
$$d = \frac{h}{2} \cotan\left(\frac{FoV_V}{2}\right)$$

View from
above



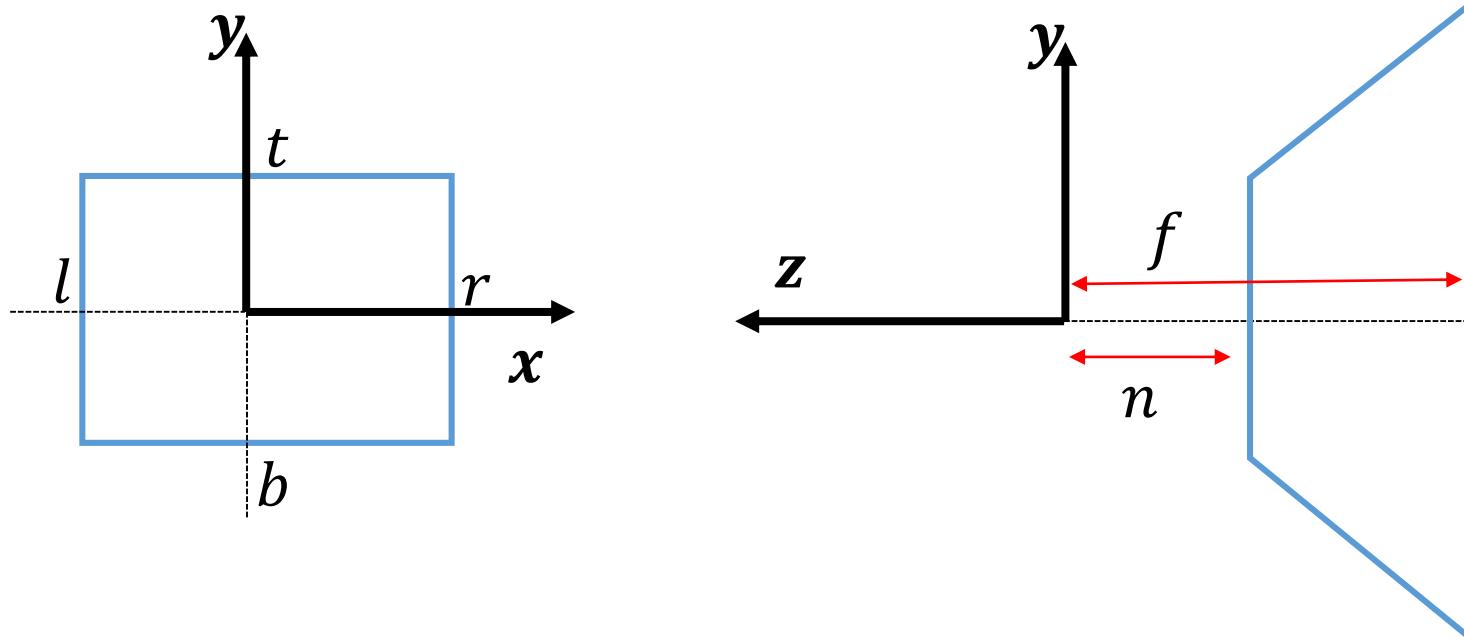
$$xFOV = 2 \arctan\left(\frac{w}{2d}\right)$$

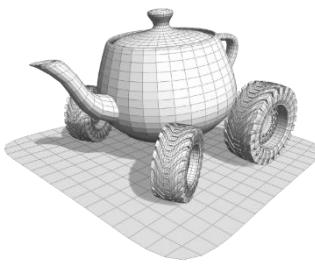
$$d = \frac{w}{2} \cotan\left(\frac{FoV_H}{2}\right)$$



Window coordinates in view space

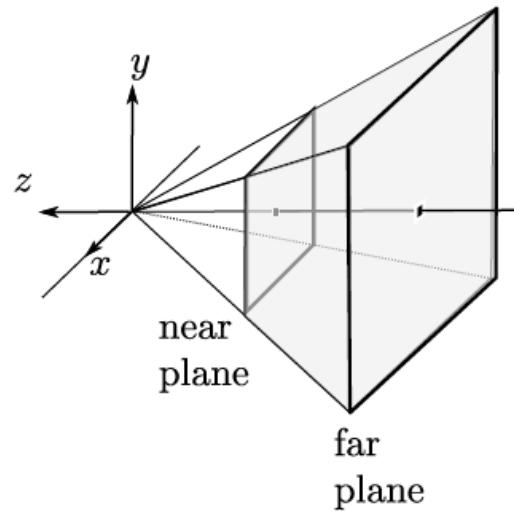
r right
 l left
 t top
 b bottom
 n near
 f far



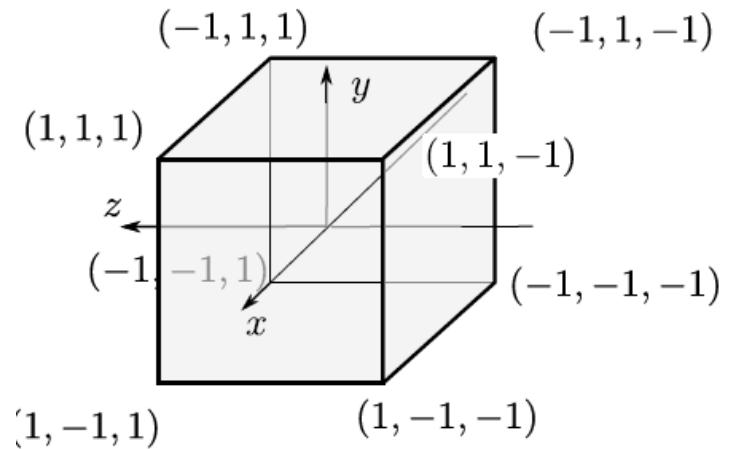


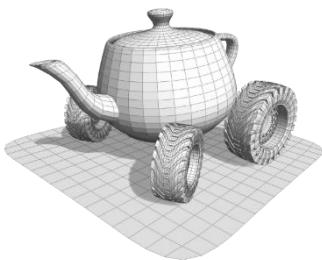
The Normalized Device Coordinates space

- The NDC is the space where primitives (partially) outside the view volume are **clipped** (hence the other name: clip space)
- This is an hardwired fixed functionality and the space is the cube of size 2 centered at the origin (we will see why in a few lectures)

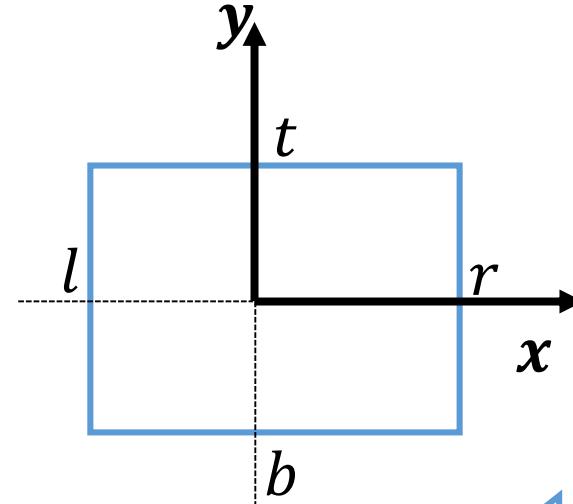


The projection matrix does this transformation



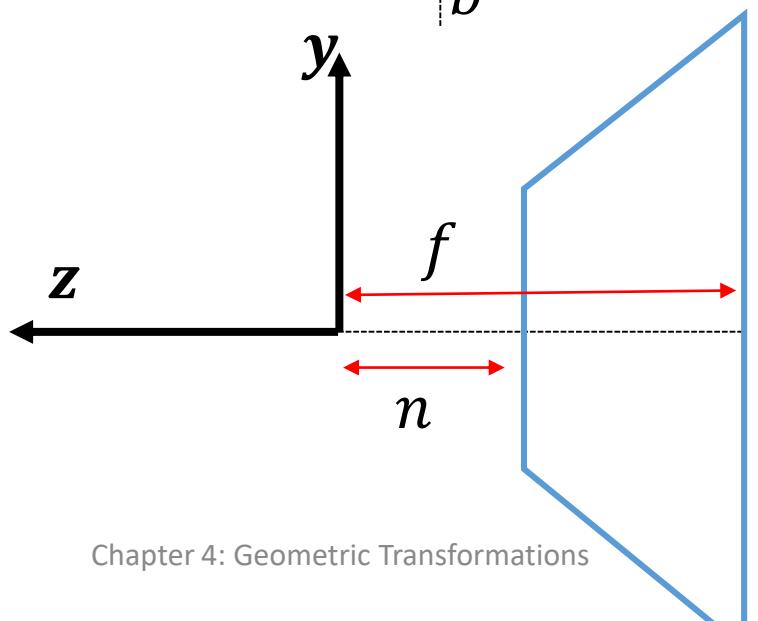


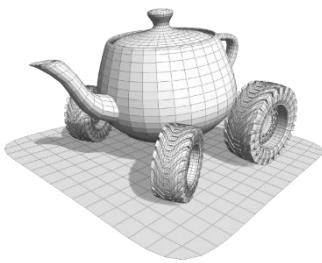
The OpenGL perspective projection matrix



r right
 l left
 t top
 b bottom
 n near
 f far

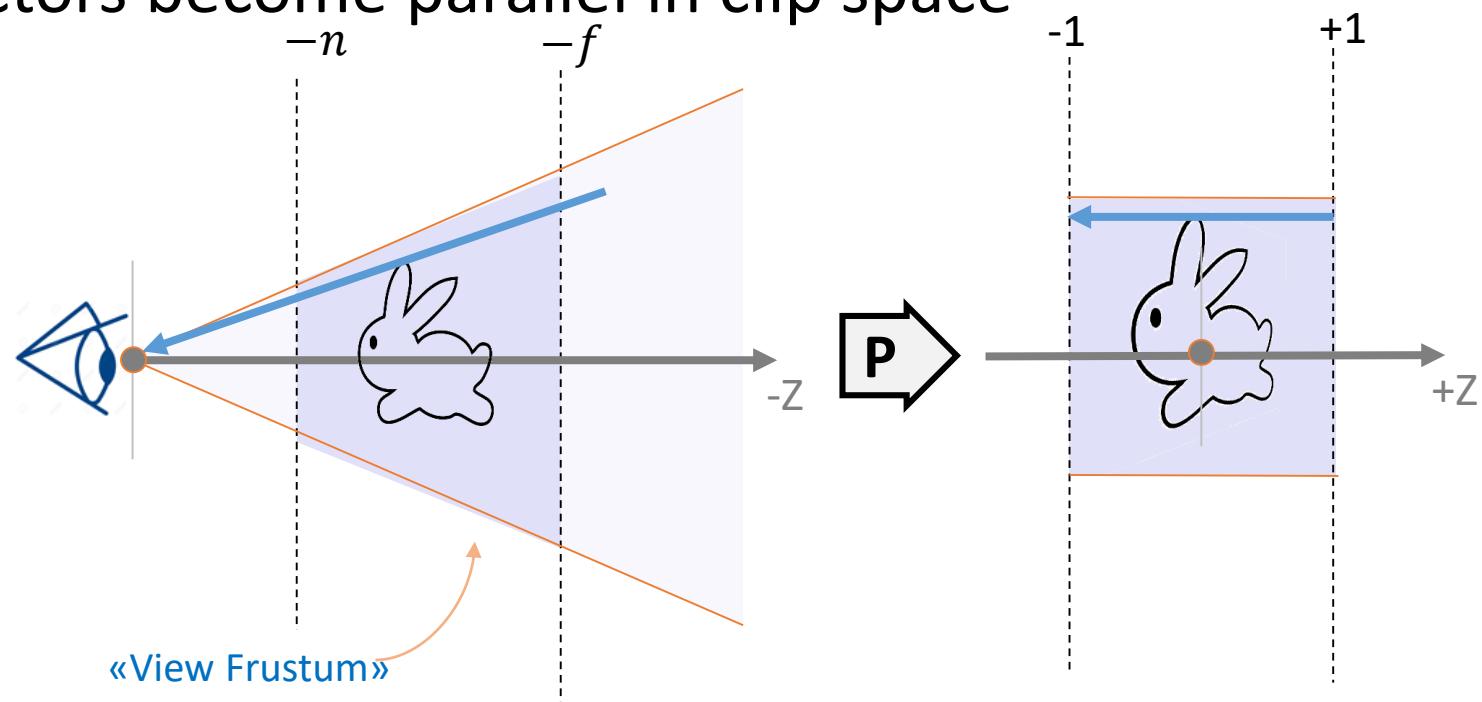
$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

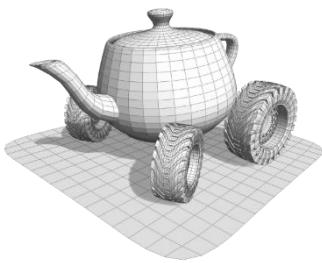




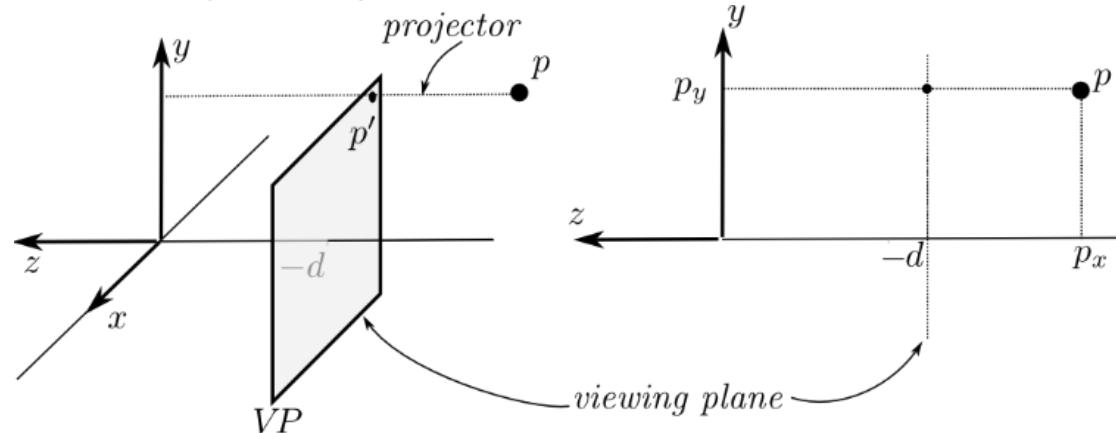
View Space to Clip space distortion

- Perspective projection makes things closer to the viewer to appear bigger (it makes sense)
- Projectors become parallel in clip space

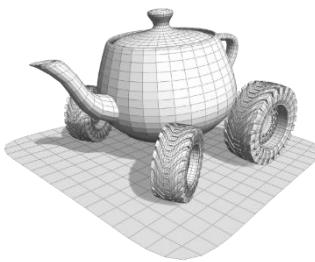




Orthographic projection



- In the **orthographics projection** the projectors are parallel.
- In the eye-in-front-of-the-window metaphor, we will have no lenses and our retina should be as big as the window (not a great metaphor)
- In terms of perspective projection, is the same as if $d \rightarrow \infty$

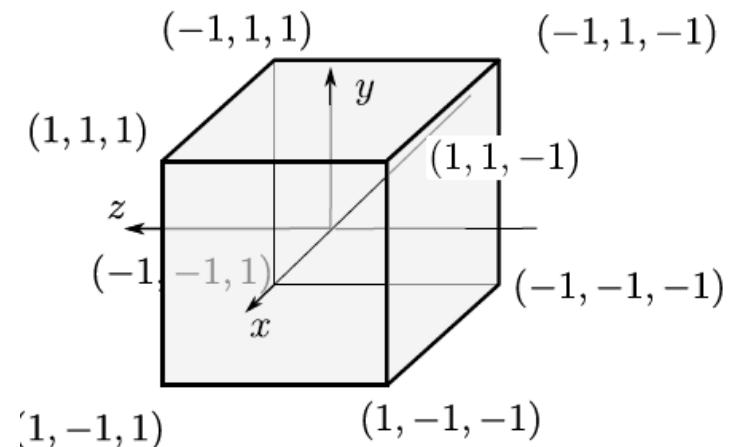
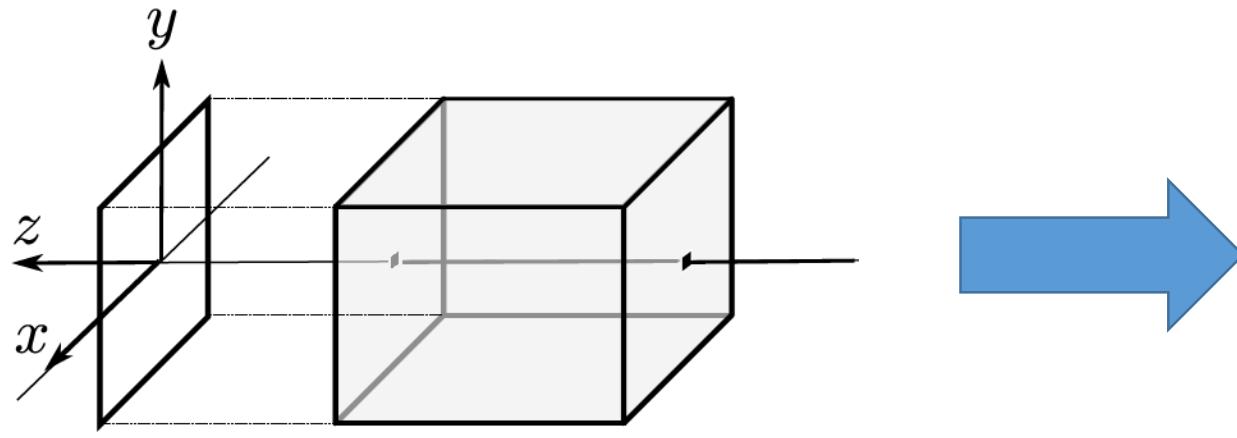


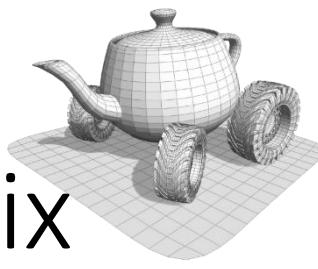
Orthographics projection matrix

- We just need to drop the z coordinates

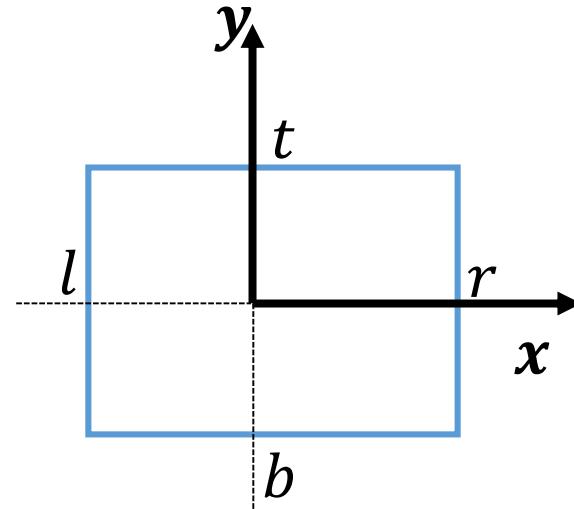
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ -d \\ 1 \end{bmatrix}$$

- However, the actual transformation need to map the viewing volume to NDC:

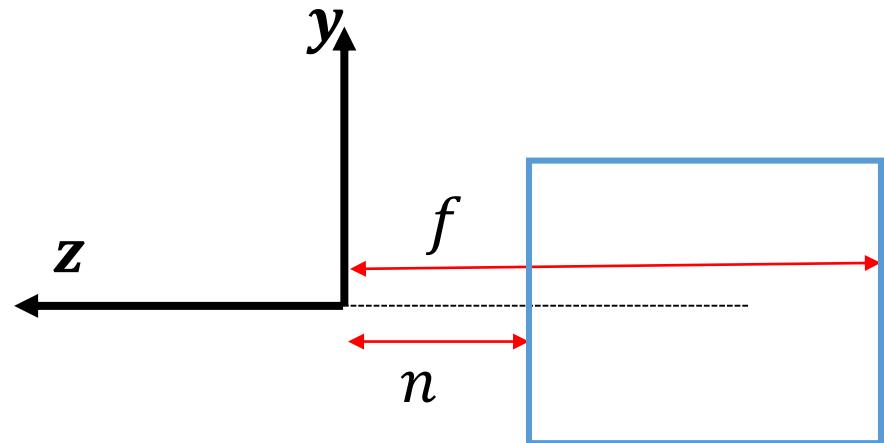




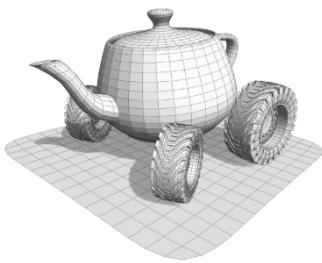
The OpenGL orthographics projection matrix



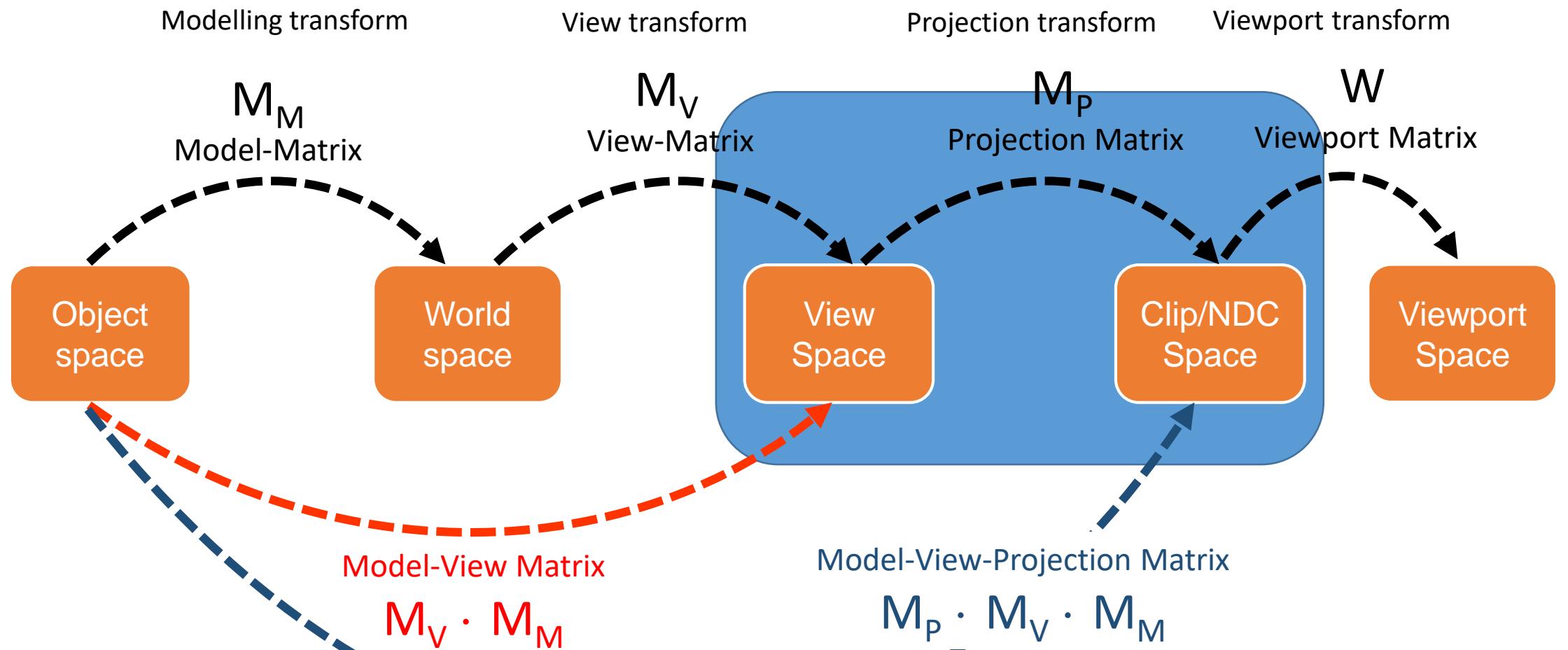
r right
 l left
 t top
 b bottom
 n near
 f far

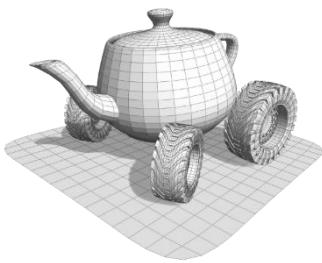


$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



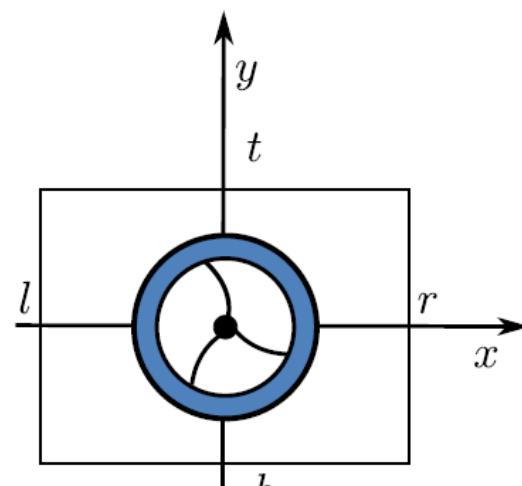
Transformations in the pipeline



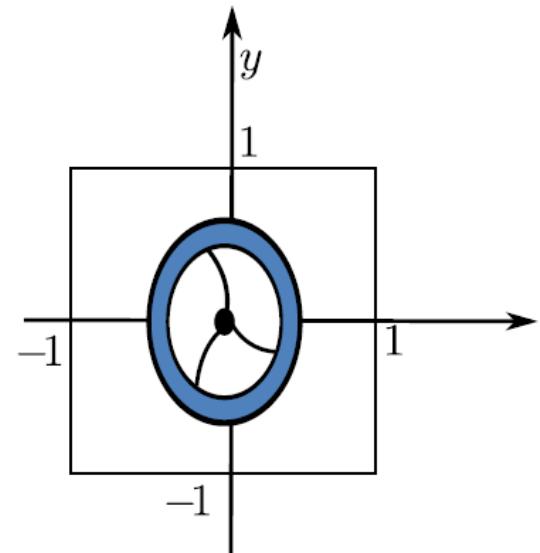


NDC to viewport (1/2)

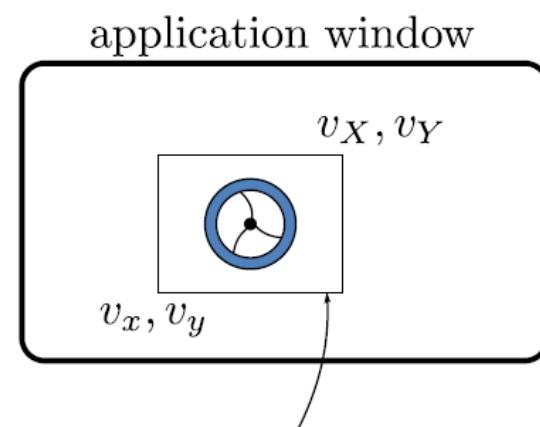
- The **viewport** is the rectangular region of the screen (specified in pixels) we want to render to.
- The viewport transformation maps the *near* side of the NDC to the viewport



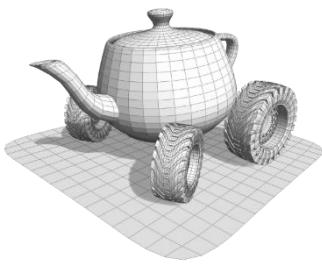
viewing window



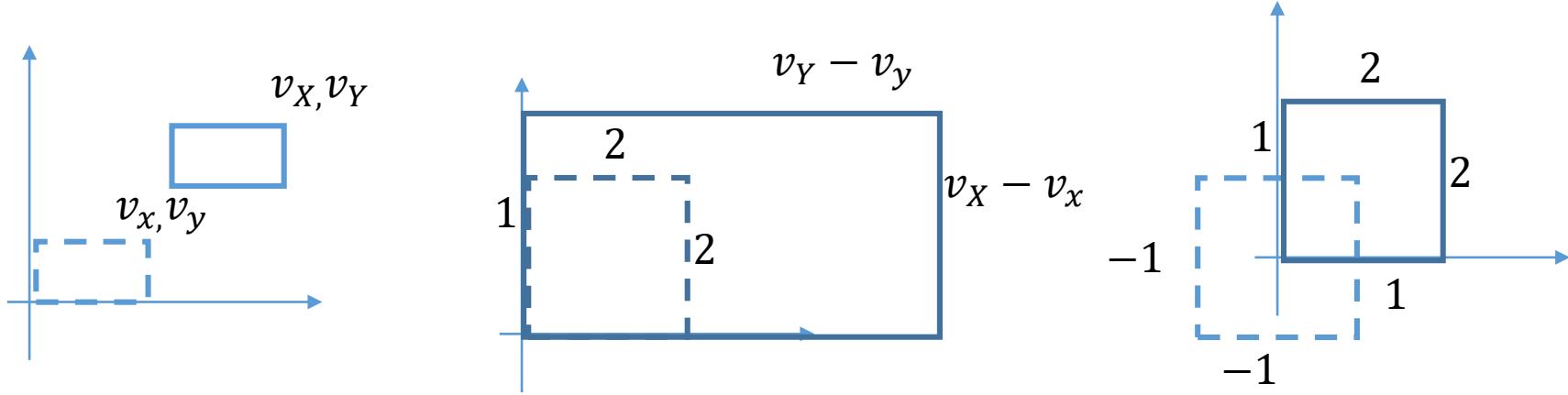
canonical view volume



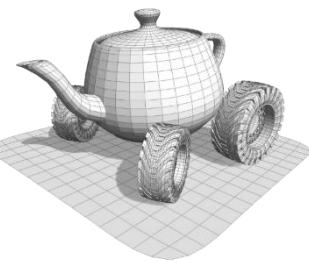
viewport



NDC to viewport (2/2)



$$\begin{aligned}
 W &= \begin{bmatrix} 1 & 0 & v_x \\ 0 & 1 & v_x \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{v_X - v_x}{2} & 0 & 0 \\ 0 & \frac{v_Y - v_y}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \frac{v_X - v_x}{2} & 0 & \frac{v_X - v_x}{2} v_x \\ 0 & \frac{v_Y - v_y}{2} & \frac{v_Y - v_y}{2} v_y \\ 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$



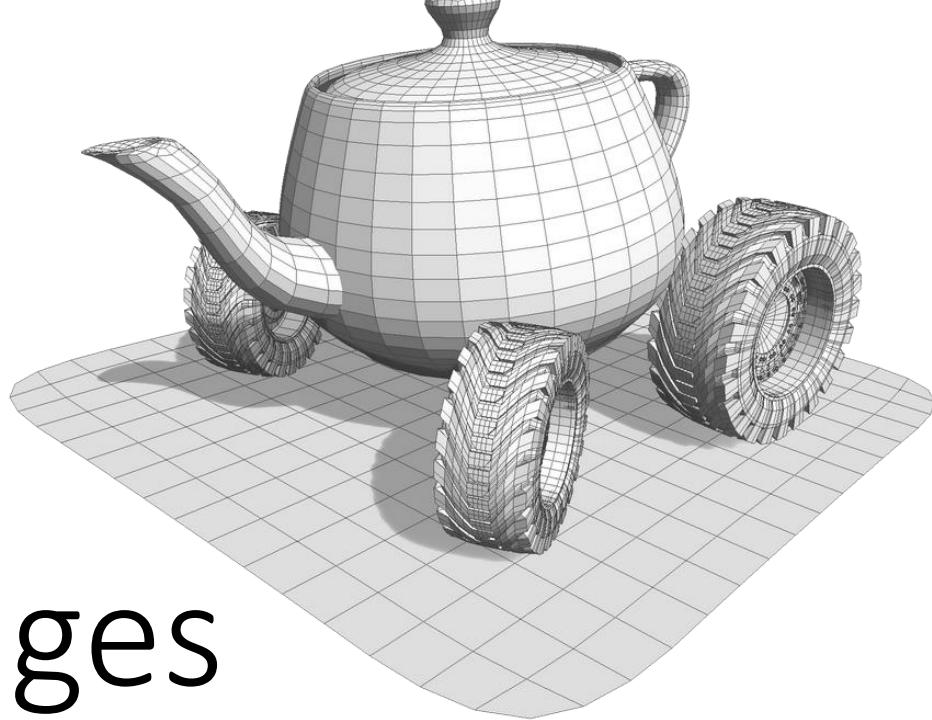
Properties preserved by transformations

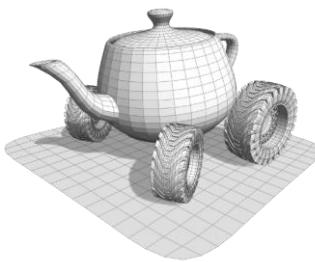
A cheat sheet to remember what transformations preserve what properties

Ratio between couple of points on parallel lines

Transformations	length	angles	ratio	collinearity
translation	Yes	Yes	Yes	Yes
rotation	Yes	Yes	Yes	Yes
uniform scaling	No	Yes	Yes	Yes
non uniform scaling	No	No	Yes	Yes
shearing	No	No	Yes	Yes
orthogonal projection	No	No	Yes	Yes
generic affine transformation	No	No	Yes	Yes
perspective transformation	No	No	No	Yes

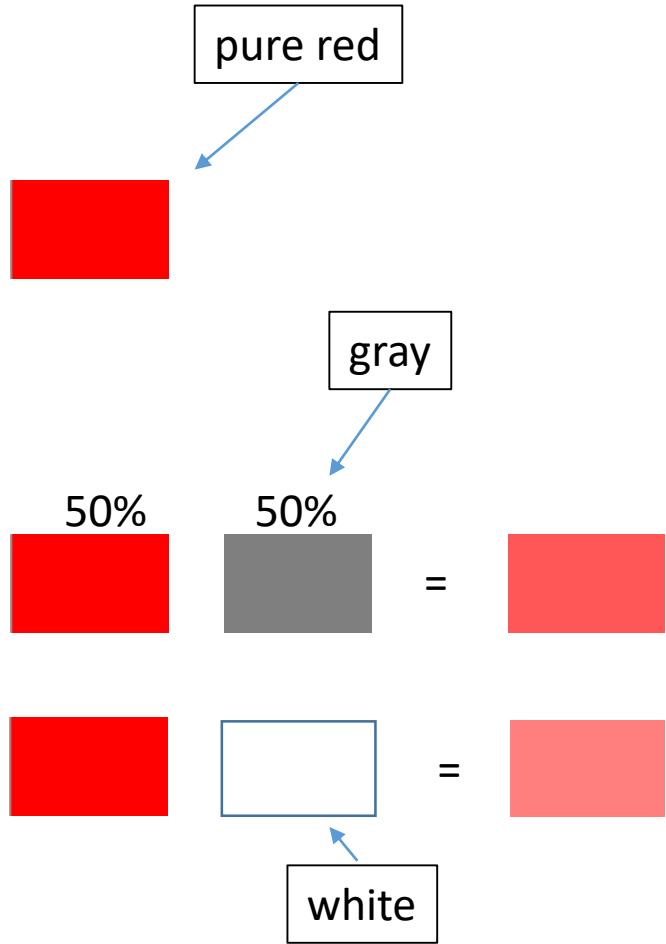
Color and Images



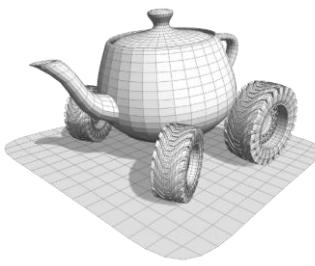


How we talk about color

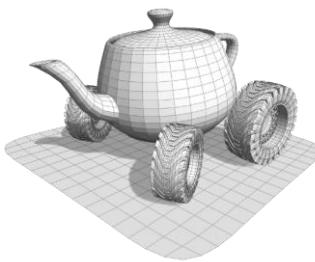
- We characterize a color with three properties:
 - Hue: “the degree to which a stimulus can be described as similar to or different from stimuli that are described as red, orange, yellow, green, blue, violet ..” (*from Wikipedia*)
 - Saturation: how much the color looks «pure»
 - The less the color is saturated, the more it goes toward gray
 - Brightness: how much the color looks «intense»
 - How «whitish» it is



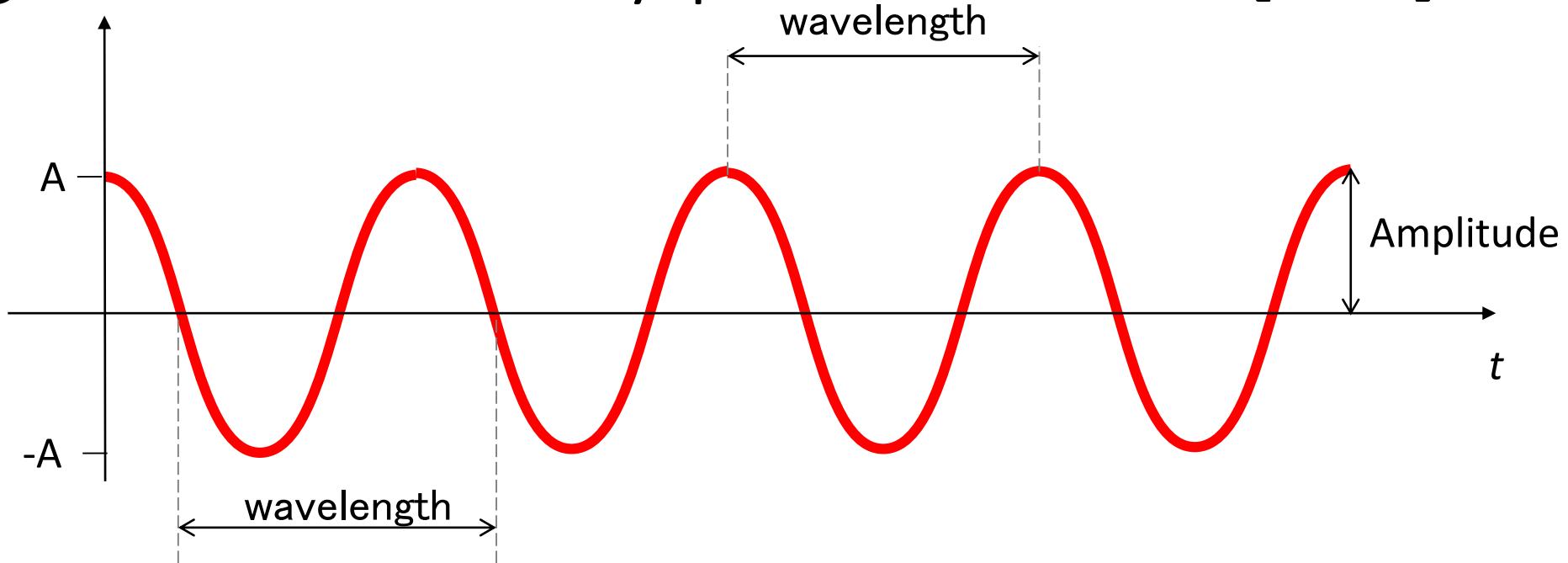
Color: Physics and Perception



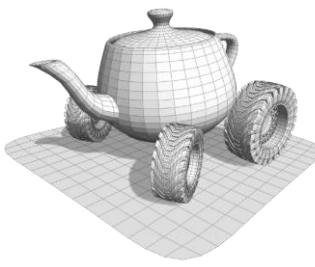
- What's the color of that object? The answer depends on two things:
 - Physics: wavelength of the electromagnetic radiation within the visible spectrum, that is, the color of the light, reaching our eyes from the object
 - Senses & Perception: how said radiation creates a stimulus in our «hardware» (senses) and how our brain interprets such a stimulus (perception)



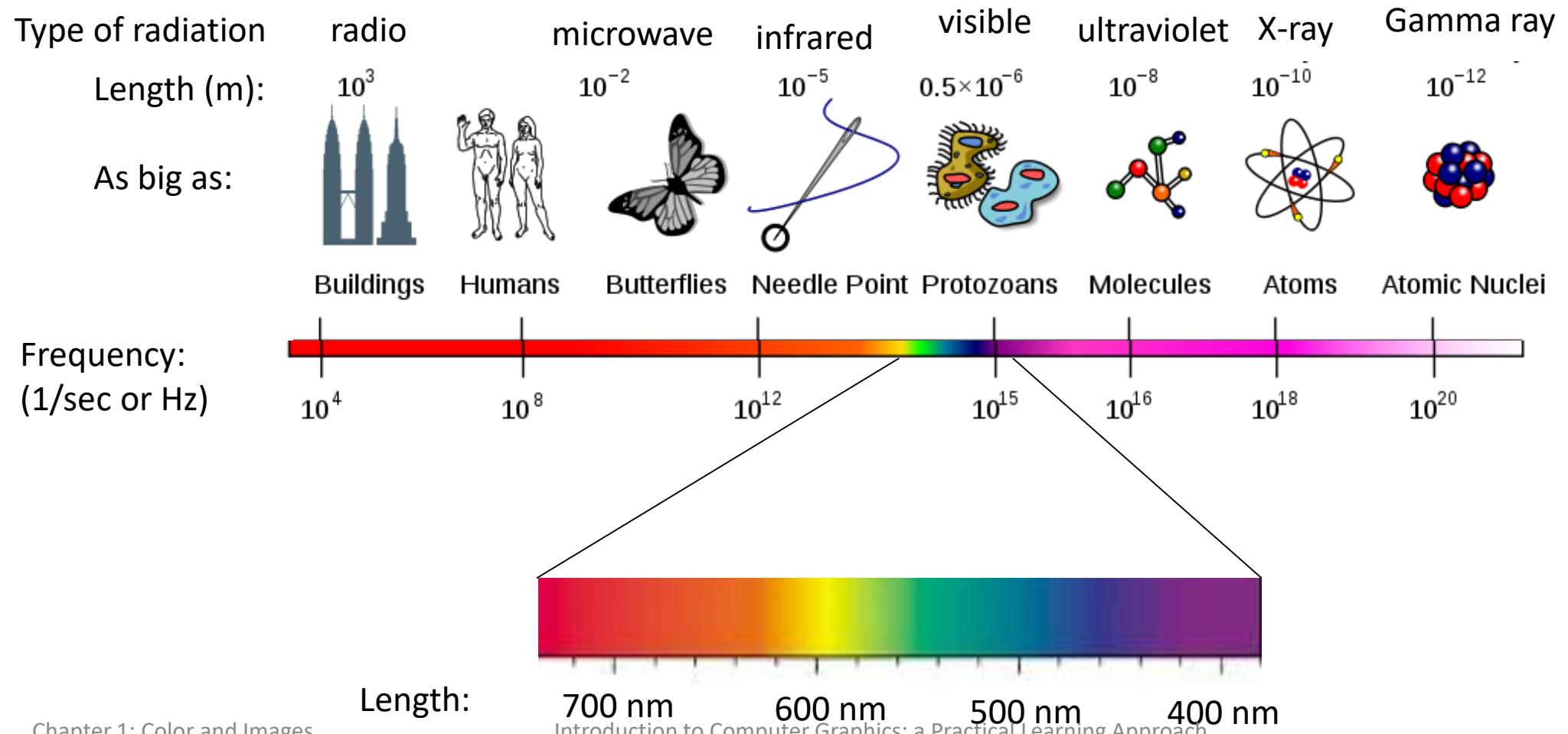
Light as ondulatory phenomenon [1/2]

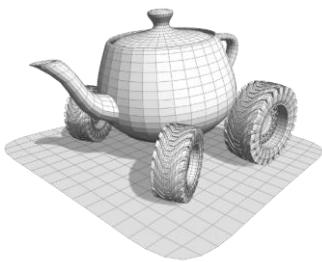


- Light is a ondulatory phenomenon characterized by
 - Amplitude: the value of the peak of each wave [V/m]
 - Wavelength [m]: distance between two consecutive peaks
 - Frequency = speed / wavelength [Hz]



Light as ondulatory phenomenon (2/2)

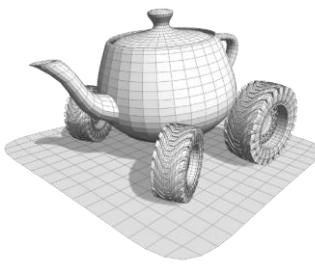




Visible Spectrum

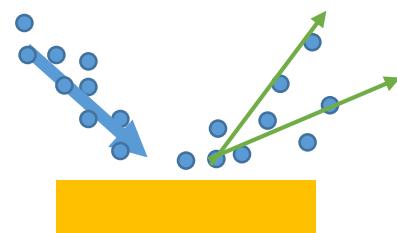


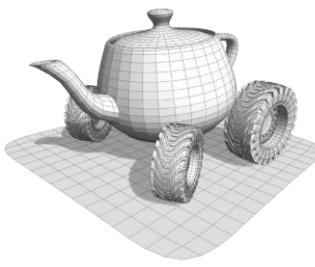
	Wavelength	frequency
Red	~ 625 – 740 nm	~ 480 – 405 THz
Orange	~ 590 – 625 nm	~ 510 – 480 THz
Yellow	~ 565 – 590 nm	~ 530 – 510 THz
Green	~ 520 – 565 nm	~ 580 – 530 THz
Blue	~ 445 – 520 nm	~ 675 – 580 THz
Indigo	~ 425 – 445 nm	~ 700 – 675 THz
Violet	~ 380 – 425 nm	~ 790 – 700 THz



Light as particles

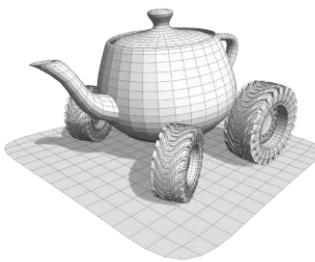
- Photon
 - A quantum of light. A no-mass *light* particle that travels at speed of light carrying an amount of energy.
- Light beam
 - many photons (e.g. $5 * 10^{50} / s$) which travel through void and media. Each photon has its own frequency/energy/color
 - Its color depends on how the energies of the photons is distributed over the wavelengths
- Photons change trajectory when they *hit* something





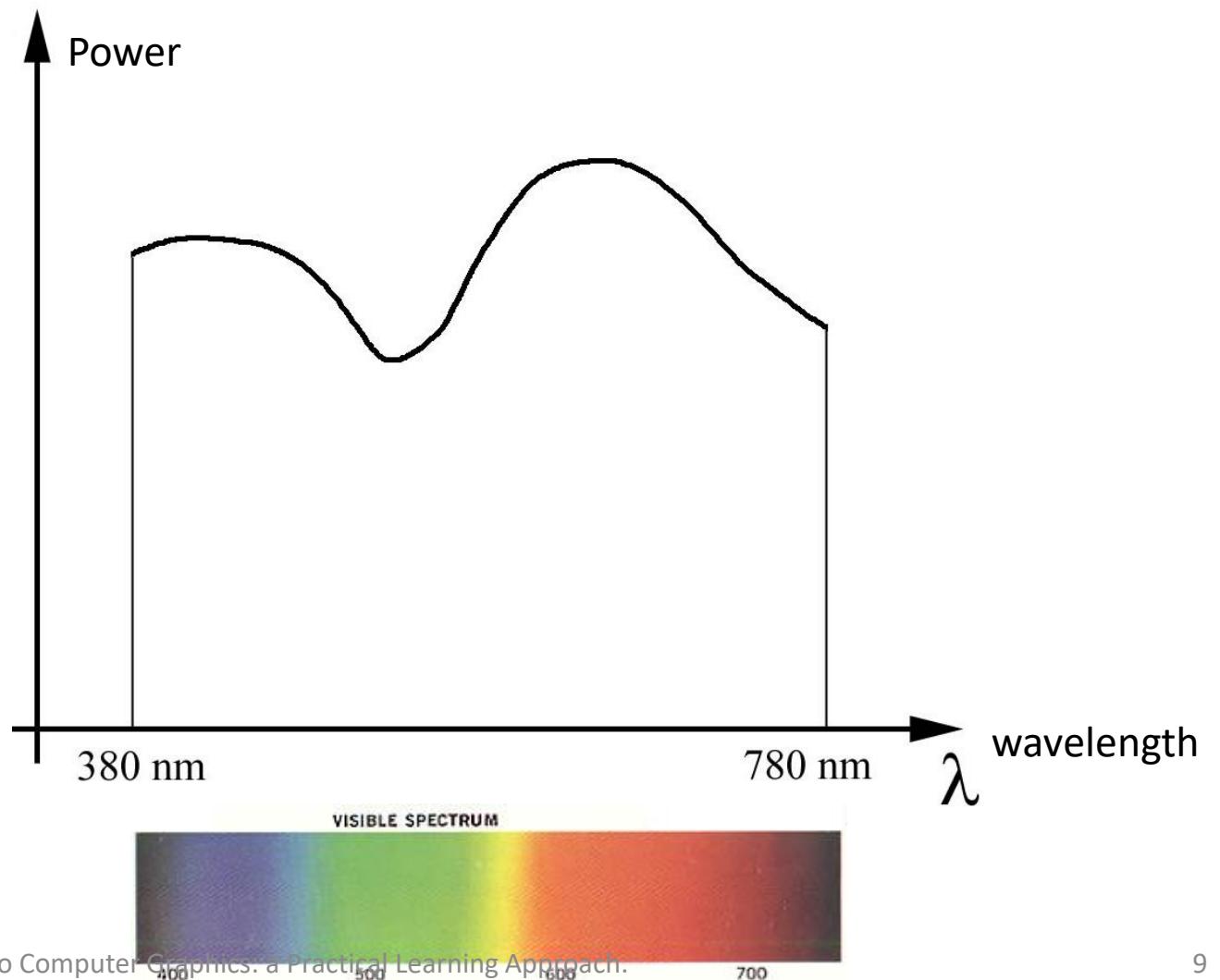
Wave – Particles duality

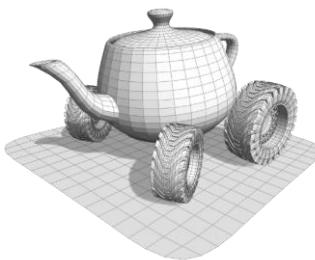
- So, is a quantum of light a *wave* or a *particle*? It's both.
- Certain phenomena are best described by thinking light in terms of electromagnetic waves, e.g. color composition, diffraction
- Certain phenomena are best described by thinking light in terms of photons traveling in straight lines: Ray-optics (or Geometrical optics)
 - computing the distribution of light on a scene



Spectral Power Distribution

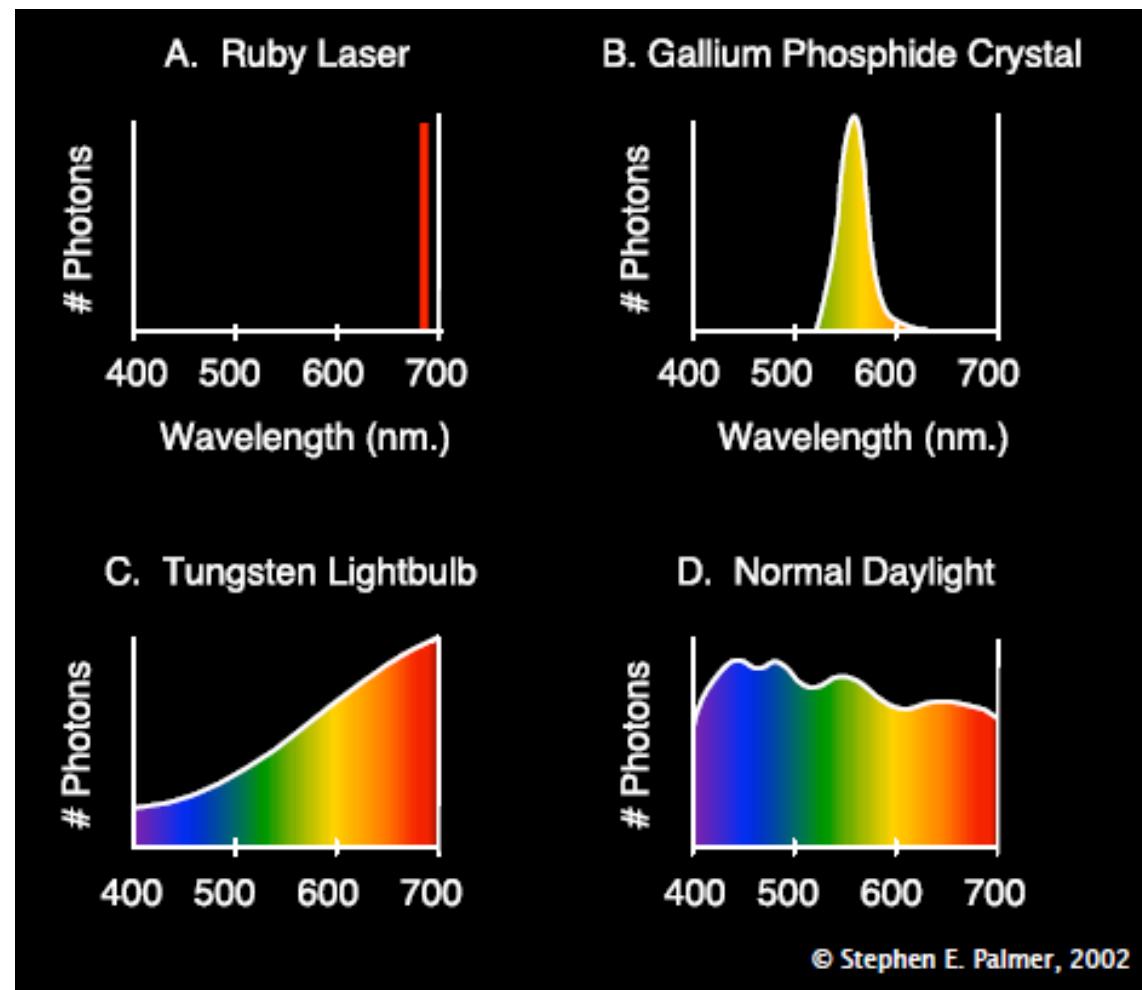
- SPD: distribution of the total energy over the spectrum of frequencies

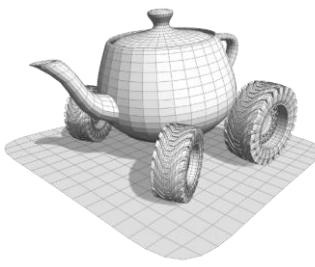




Spectral Power Distribution

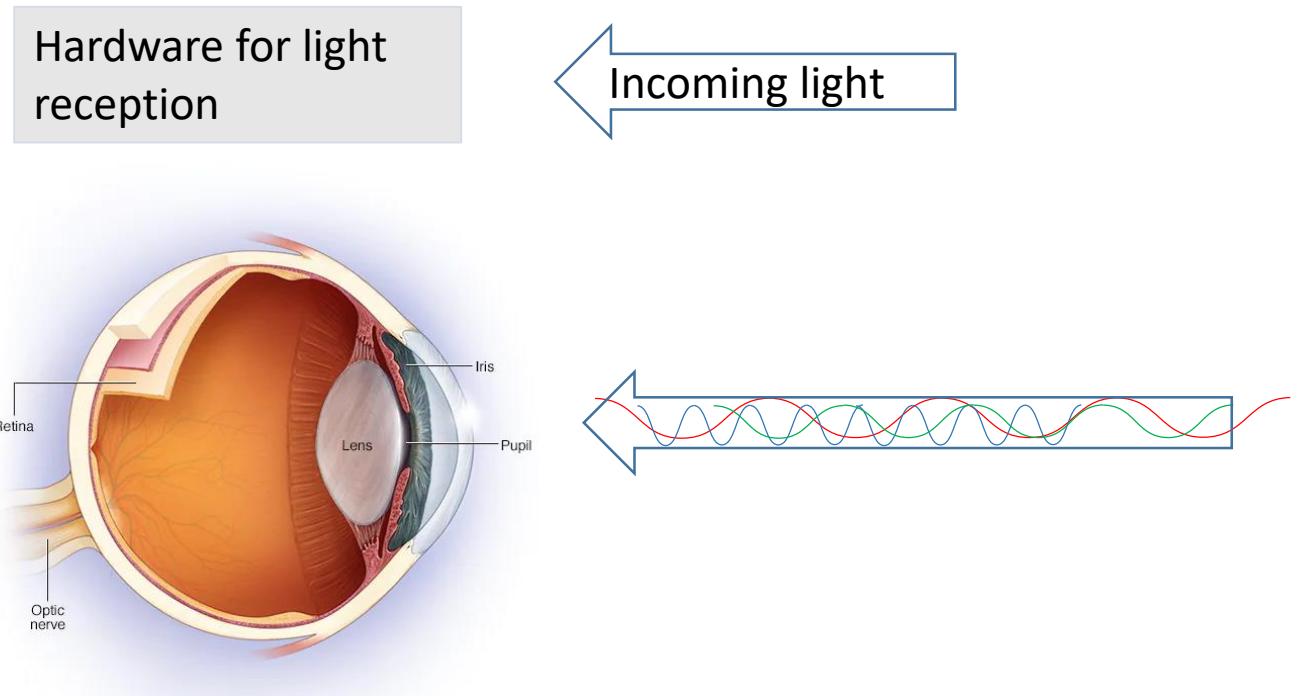
- Different light sources have different SPD

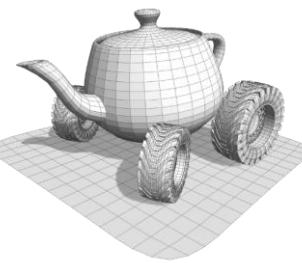




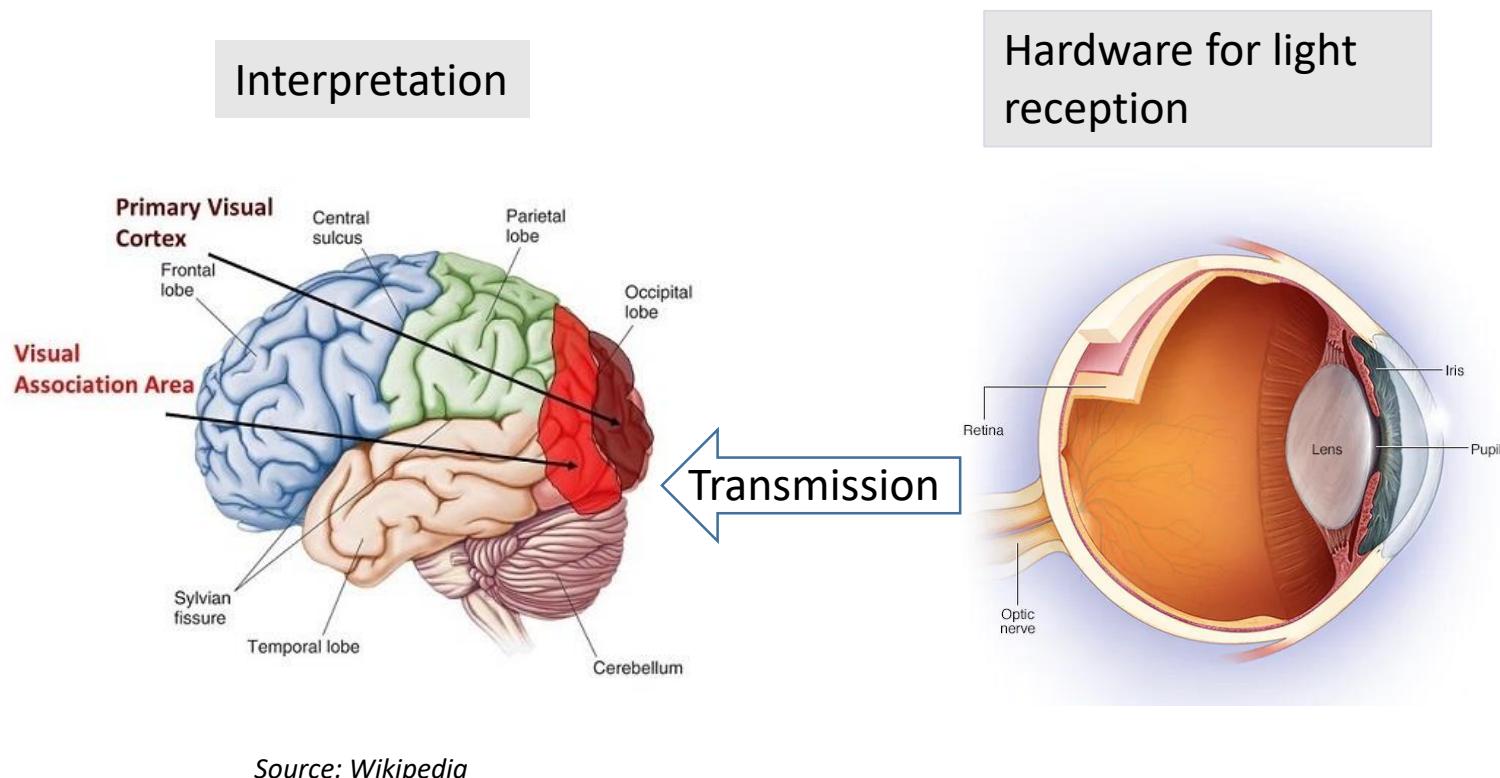
How do we see light?

- Light enters the eye through a tiny opening: the **pupil**
- Light finds the **lens** which will distribute the light all over the **retina**
- The retina contains receptors which are activated by light on the basis of its frequency: **rods and cones**

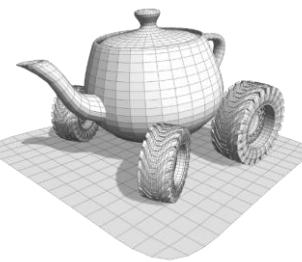




How do we see light?

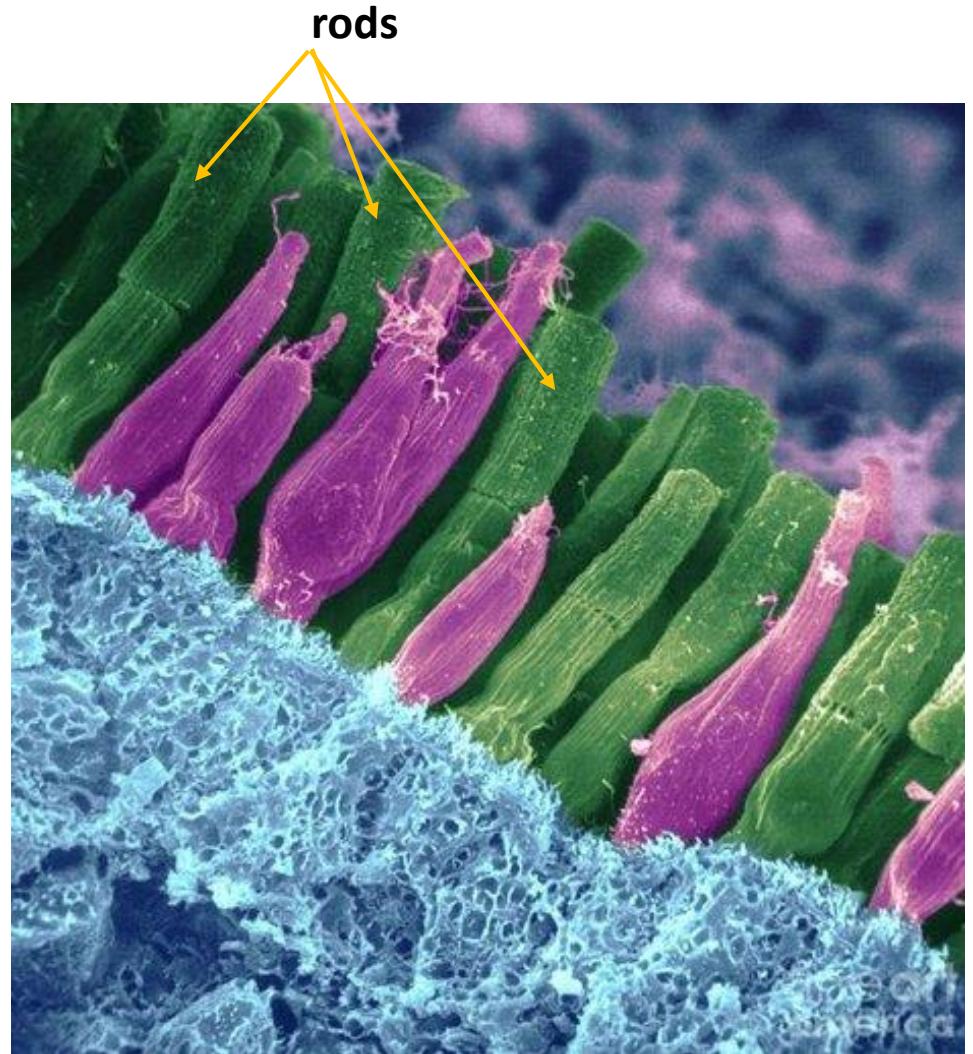


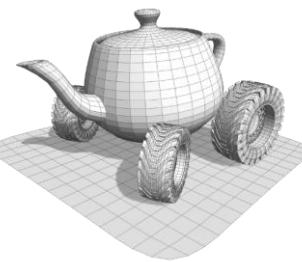
- The **optic nerve** brings the signal provided by rods and cones to the brain
- The brain reads the signals and creates the image you see



Rods and cones

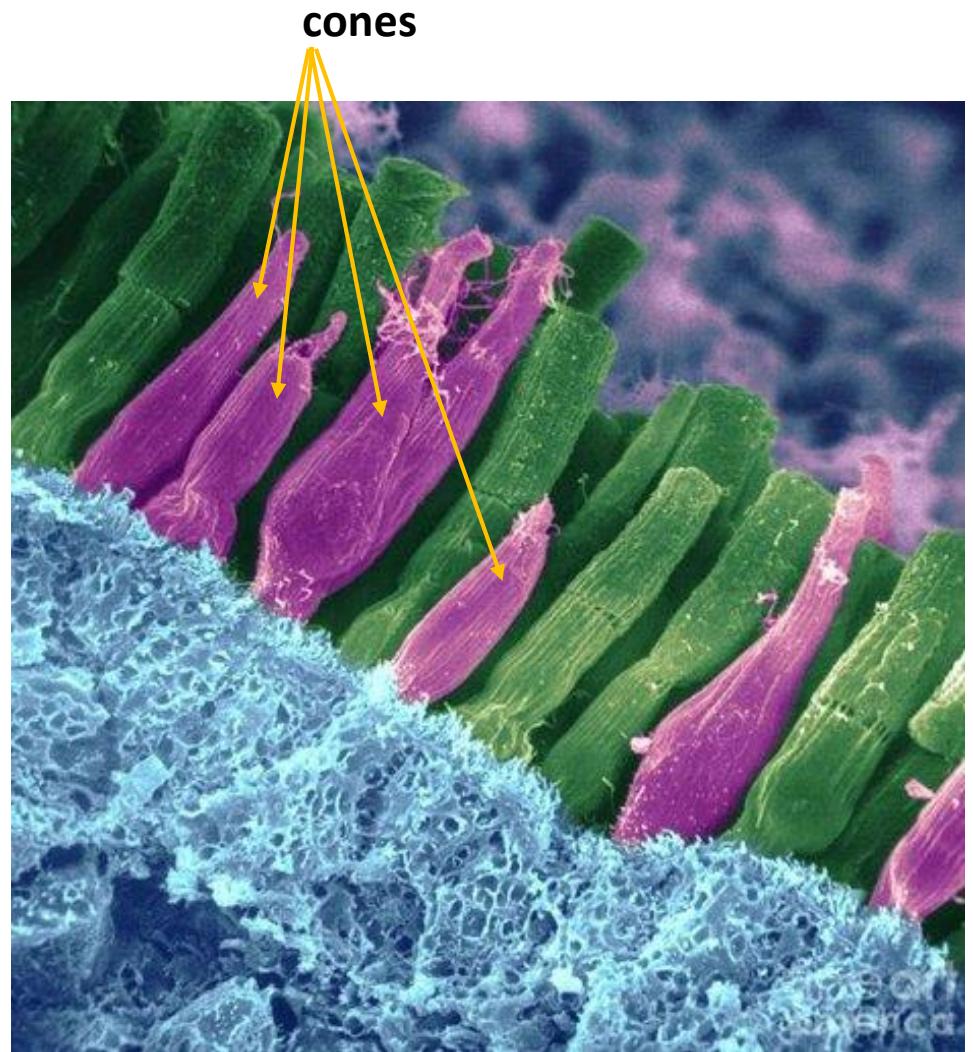
- Rods
 - 120 millions in the retina
 - 1000 times more sensitive to light than cones
 - Discriminates between brightness level in low illumination
 - More sensitive to high frequencies (short wavelengths)

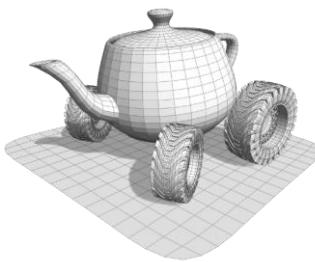




Rods and cones

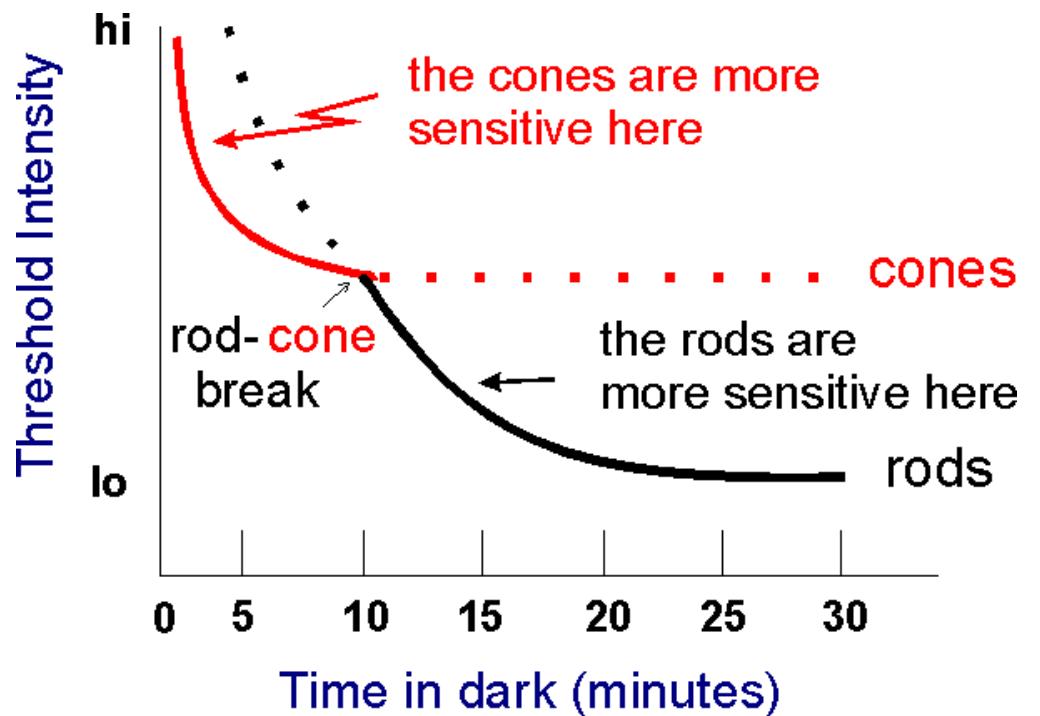
- Cones
 - 6 millions in the retina
 - Mostly concentrated in the central part of the retina: the **fovea**
 - Discriminate between light frequencies (that is, color)
 - Three types of cones:
 - Red (64%) Long, peak in 560 nm – 580 nm
 - Green (32%) Middle: peak in 530 nm – 540 nm
 - Blue (2%) Short: peak in 420 nm – 440 nm



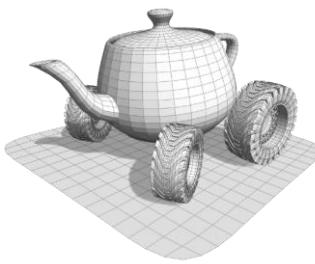


Rods, Cones, and light

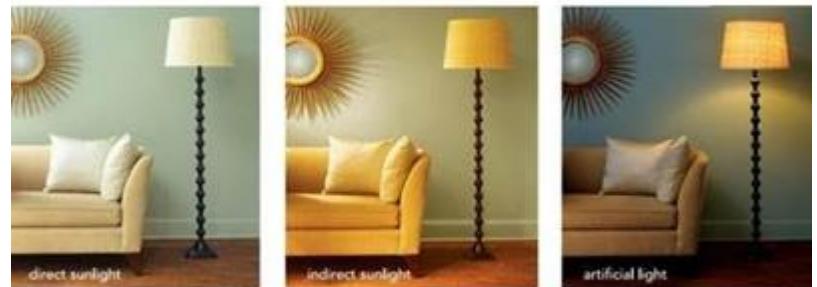
- The dark room experiment
 - Go from the outside in a dark room with no light
 - How bright does a light need to be to be detected?
 - And after 1 minute in the dark...
 - And after 2 minutes in the dark?
 - And after 3 minutes?
 - Cones adapt faster, so the first few minutes of adaptation reflect cone-mediated vision.
 - Rods work slower, but since they can perform at much lower levels of illumination, they take over after the initial cone-mediated adaptation period.



Color of an object



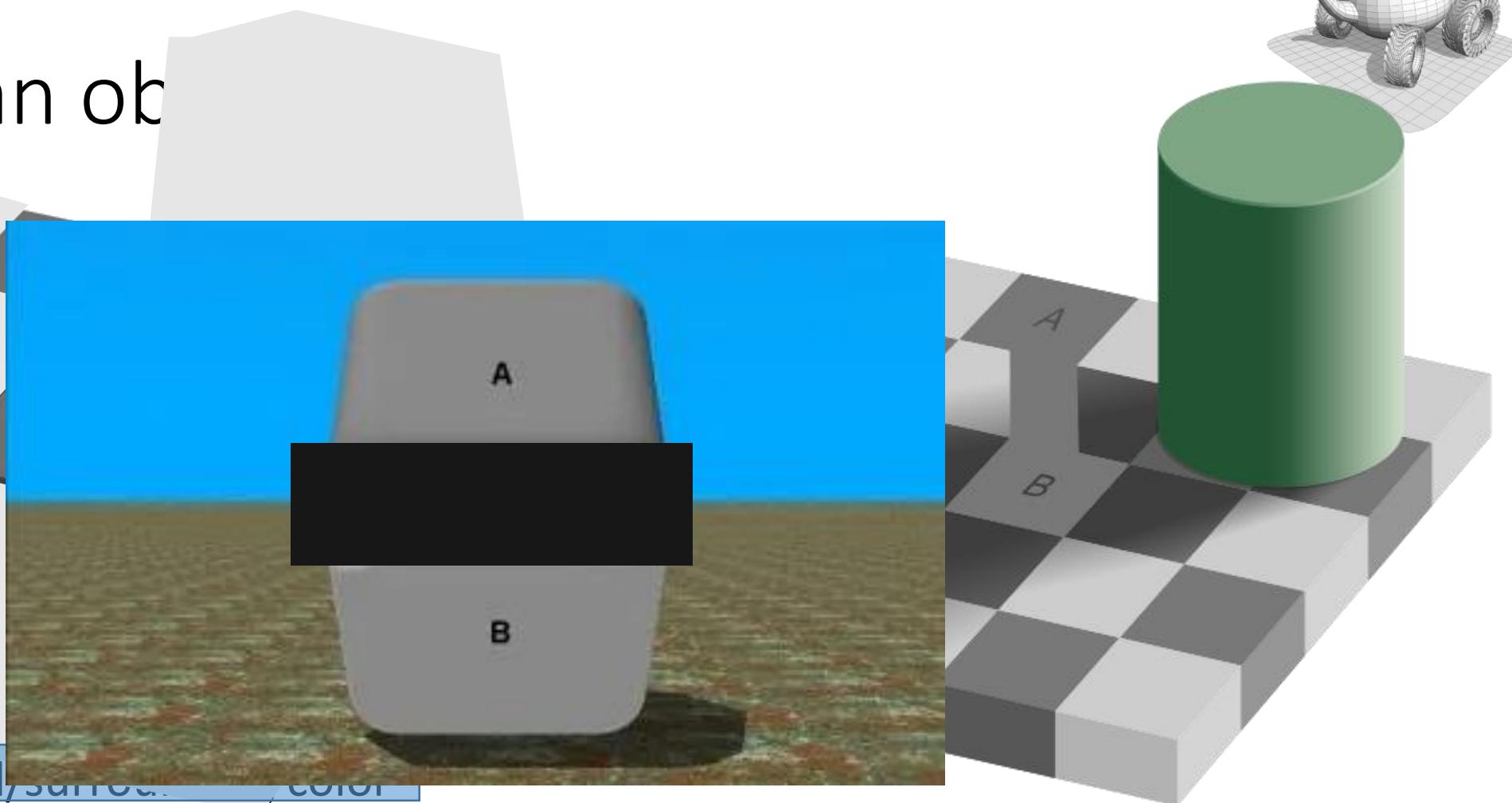
- Objects do not have color.
- They have material properties and reflected light depends on
 - Object material properties
 - That is, how they reflect light
 - Characteristics of light (intensity, color)
 - **Illuminant metamerism:** perceived color changes with illumination
 - Background/surrounding color
 - Observer metamerism



Color of an object

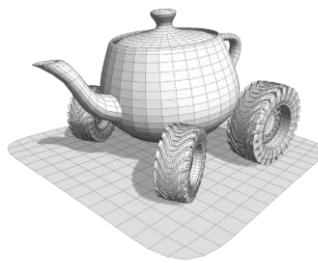
- Object color
- Transparency
- Background, surface, lighting, color
- Observer metamerism

A and B are the same luminance

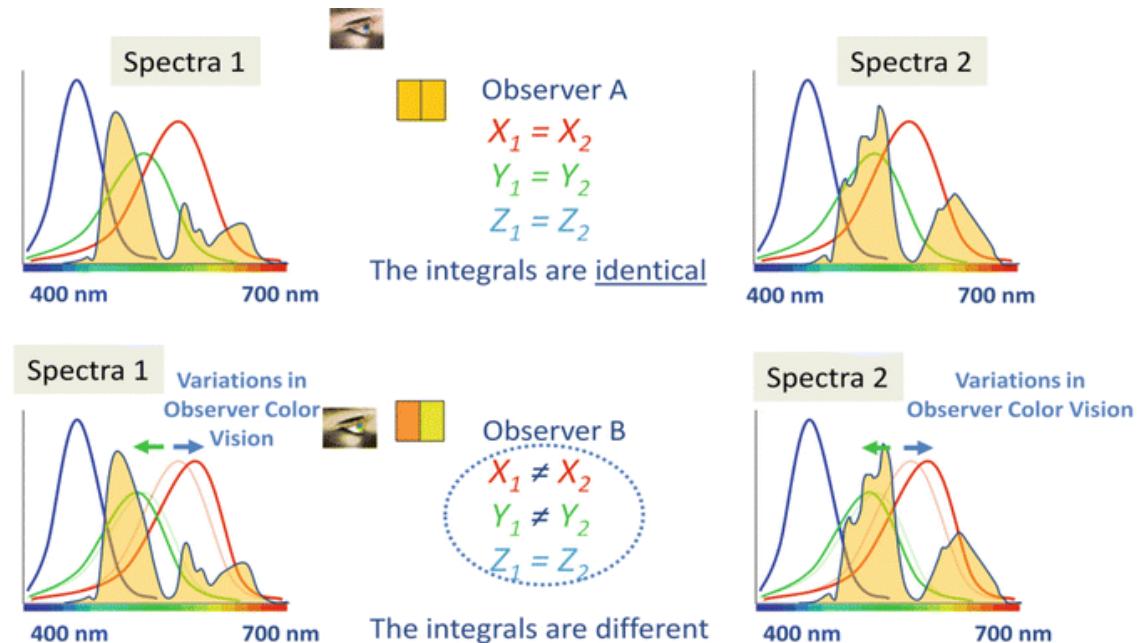


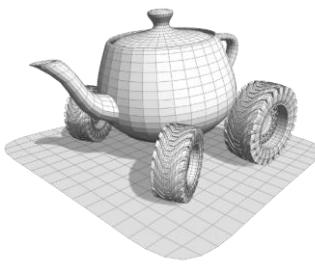
By Original: Edward H. Adelson, vectorized by Pbroks13. - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=75000950>

Color of an object



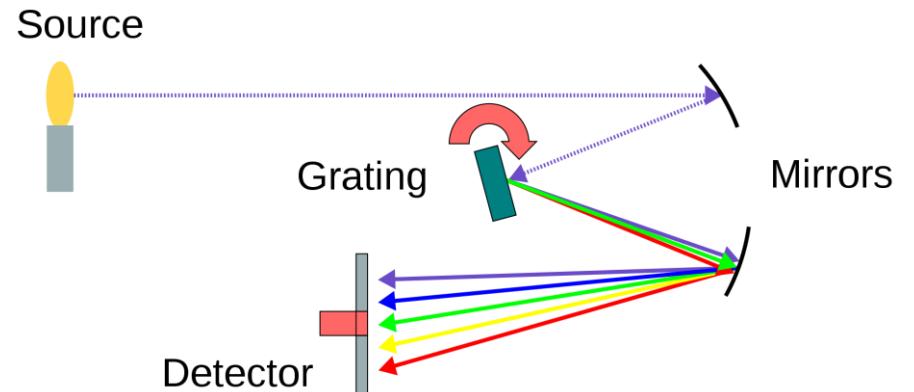
- Objects do not have color.
- They have material properties and reflected light depending on
 - Object material properties
 - That is, how they reflect light
 - Characteristics of light (intensity, color)
 - **Illuminant metamerism:** perceived color changes with illumination
 - Background/surrounding color
 - **Observer metamerism**





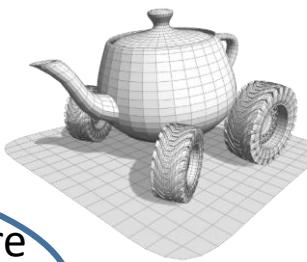
How do we perceive color?

- A **spectrometer** is an instrument used to measure properties of light over a specific portion of the electromagnetic spectrum
 - In short, it decomposes the incoming light its set of frequencies and it measures their intensity

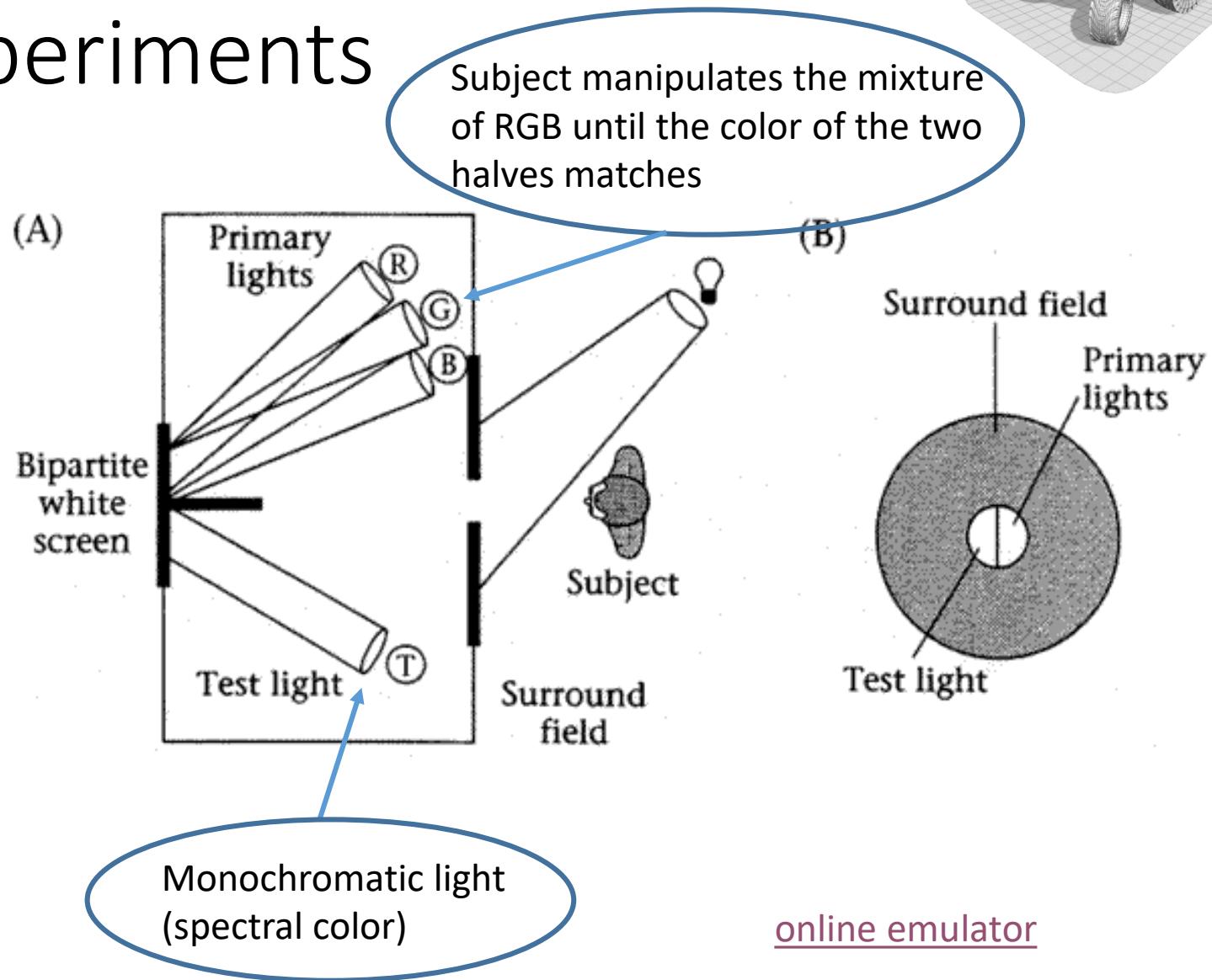


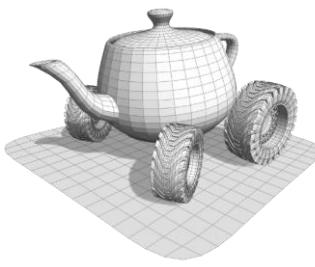
- We do a similar thing (kind of) but only with 3 detectors (receptors)
- Our brain combines the *stimulus* of our receptors to form a color.
how?

Color Matching Experiments



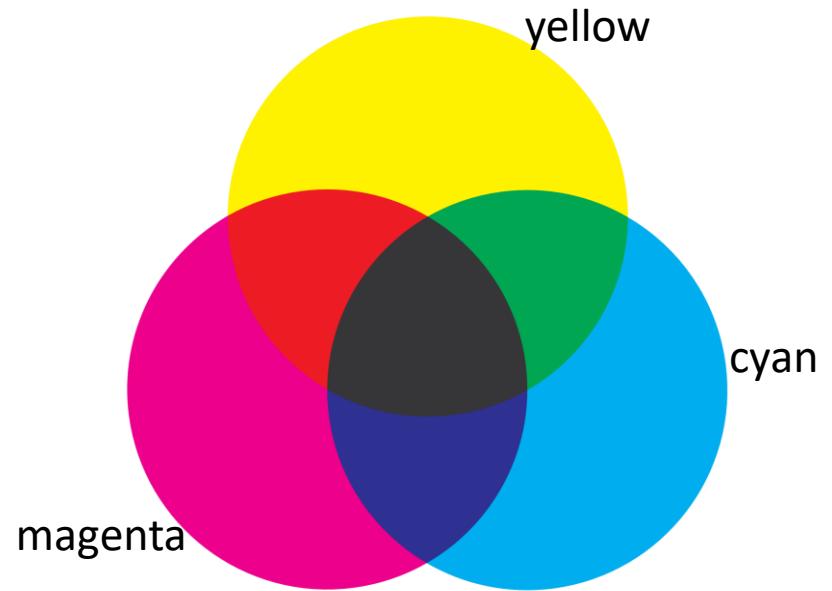
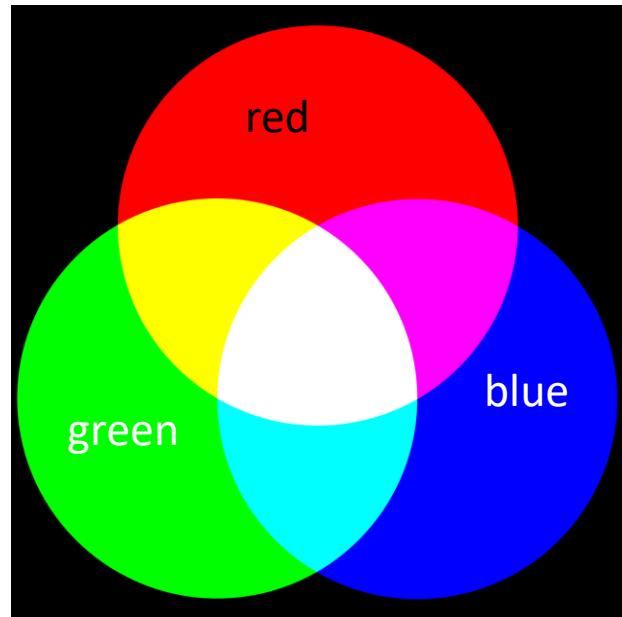
- The **color matching experiments** were carried out (independently) by W. D. Wright and J. Guild to constructing a formal, quantitative, rigorous system for characterizing colors: the **CIE color spaces**.

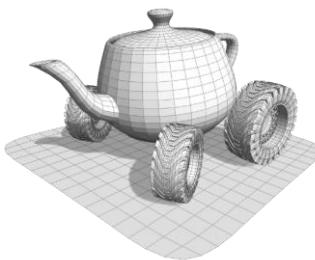




Additive and Subtractive primaries

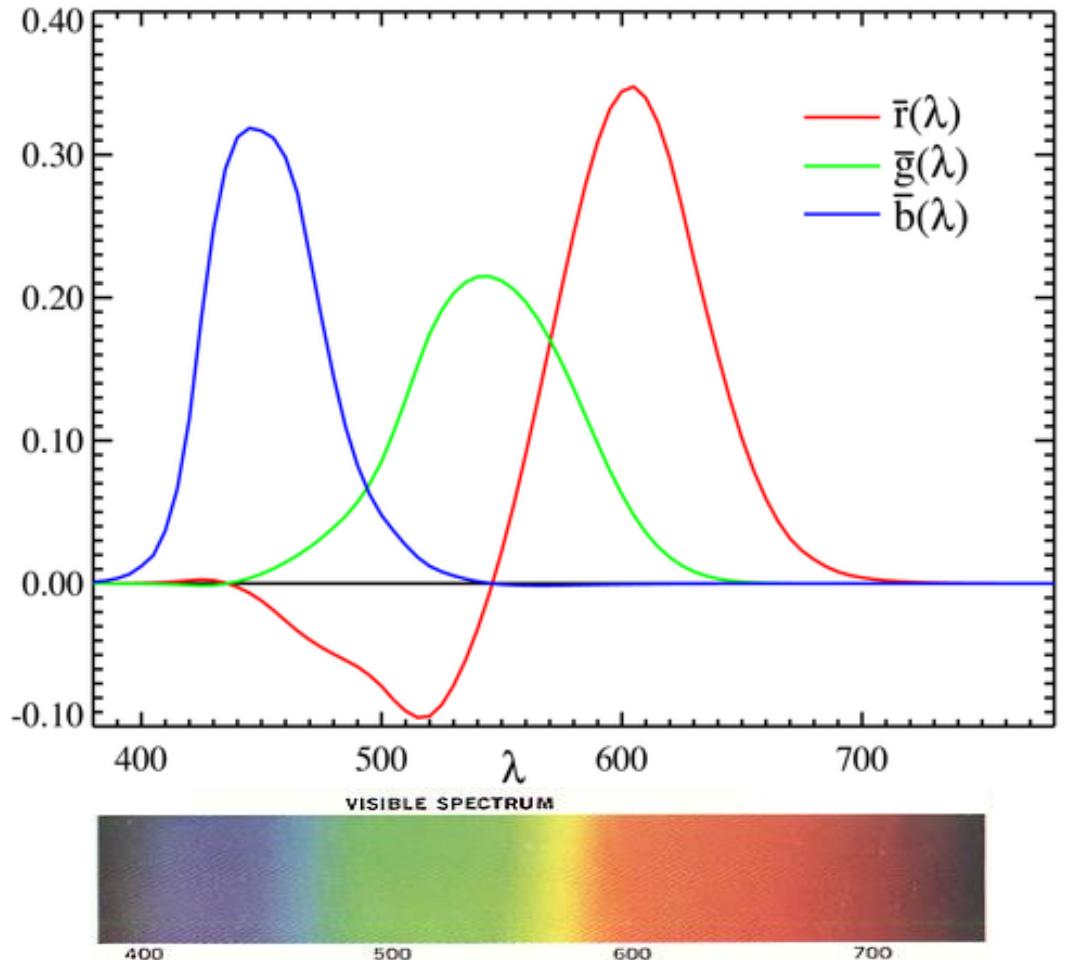
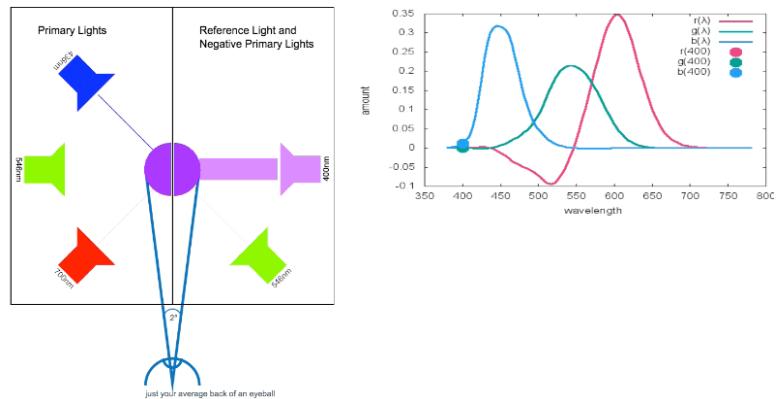
- Additive: project colored lights on a white wall (in a dark room)
- Subtractive: put ink of different colors to a white piece of paper

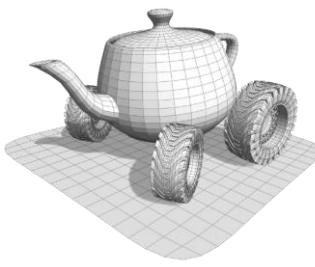




Color Matching Functions

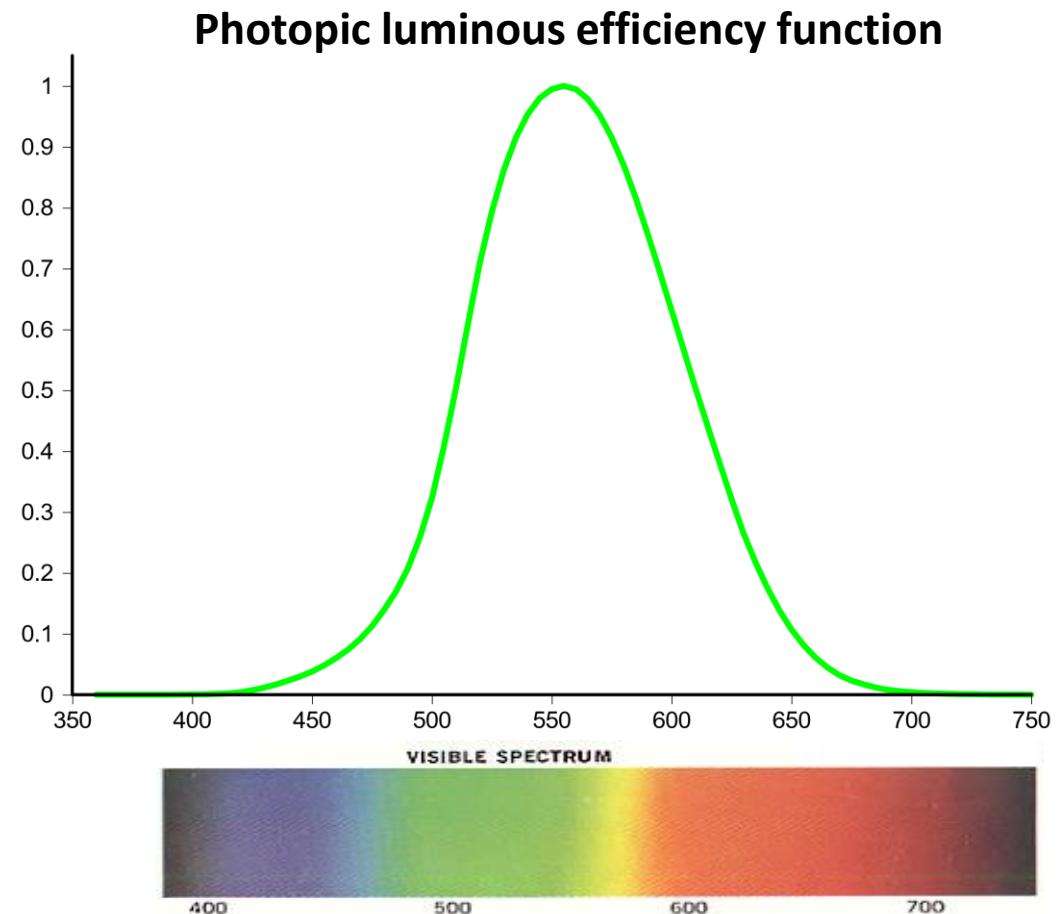
- plotting the RGB values for each test frequency we obtain the **CIE RGB Color Matching Functions**
- *Negative values* for a light mean that the corresponding light has been added to the test light

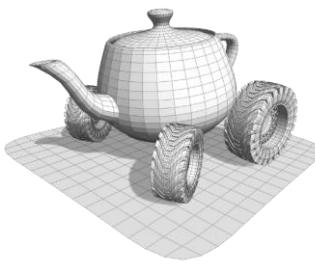




Illuminant (1/2)

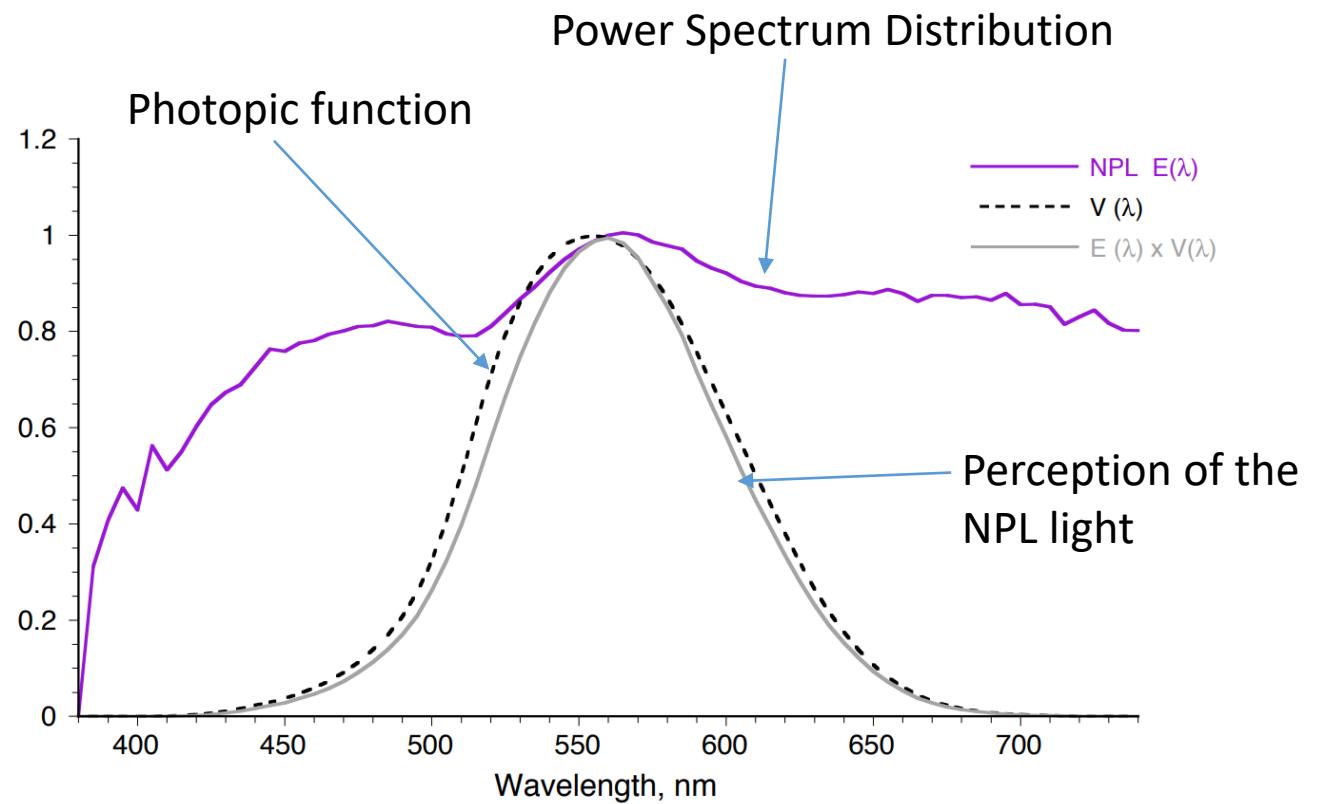
- **Brightness:** how *bright* a light is *perceived*
- **Photopic Luminous Efficiency Function:** brightness at each wavelength

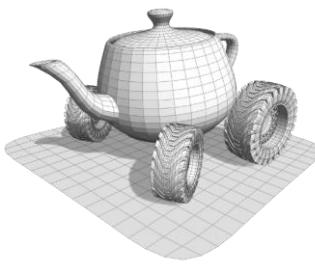




Illuminant (2/2)

- Ideally, a light with a constant PSD which used as the test light
- National Physical Laboratory (NPL) standard white light was used
- Nowadays there are other solutions (more later)





Trichromatic coefficients (1/2)

- They wanted that setting equal coefficients of R,G and B were perceived as white
 - Which was not the case (see the photopic luminous function)
- So they find the “multipliers” to apply to the value of R,G and B so that

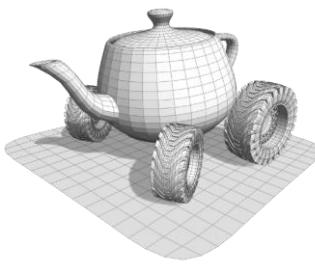
$$\text{White} = r m_r R \oplus g m_g G \oplus b m_b B$$

so that: $r m_r = g m_g = b m_b$

Handles manoeuvred by the observers

$a \oplus b$ means perception of lights a and b superimposed

Other values have been found in other experiments



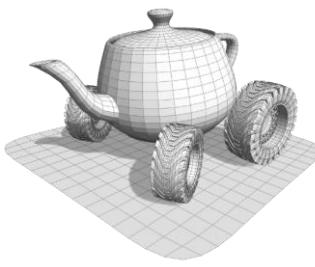
Trichromatic coefficients (2/2)

- Given the observer input r, g, b for a spectral color C , the **trichromatic coefficients α, β , and γ** are defined as

$$\alpha = \frac{r m_r}{r m_r + g m_g + b m_g} \quad \beta = \frac{g m_g}{r m_r + g m_g + b m_g} \quad \gamma = \frac{b m_b}{r m_r + g m_g + b m_g}$$

$$C = \alpha R \oplus \beta G \oplus \gamma B \quad \alpha + \beta + \gamma = 1$$

- The effect of this procedure is to throw away the luminance and just keep the mixture of colors



Luminance for a wavelength

- Let C_λ be the spectral color for the wavelength λ

$$C_\lambda = \alpha R \oplus \beta G \oplus \gamma B$$

- The luminance of C_λ is found as:

$$L_\lambda = \alpha m_r + \beta m_g + \gamma m_b$$

- Example:

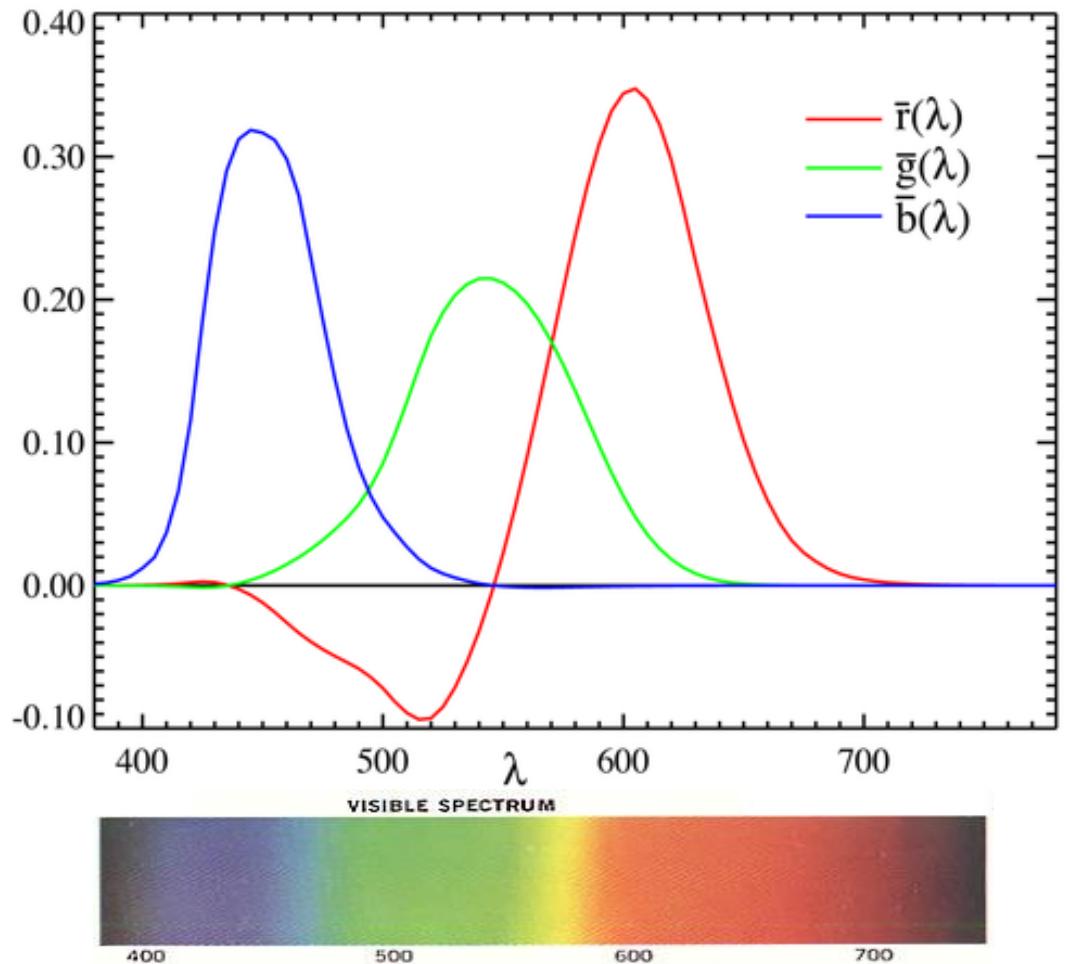
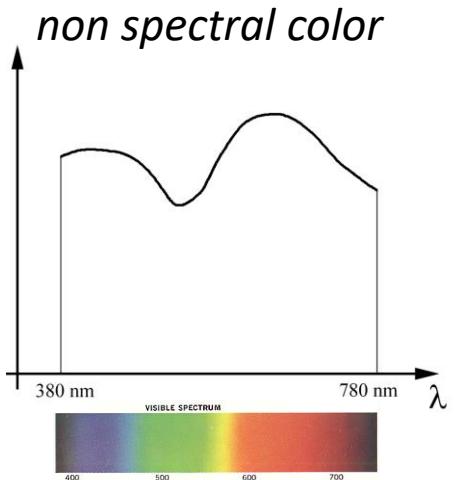
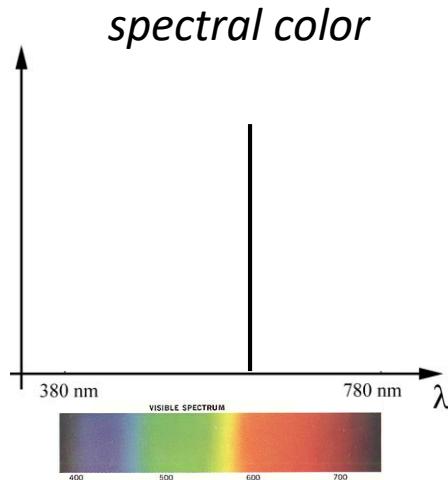
$$C_{500} = -0.94 R \oplus 1.17 G \oplus 0.77 B$$

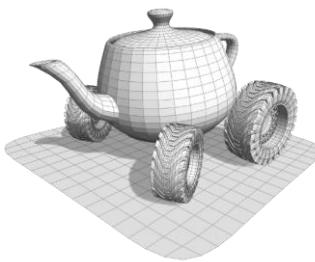
$$L_{500} = -0.94 \times 1 + 4.39 \times 1.17 + 0.048 \times 0.77 = 4.236$$



Tristimulus values: CIE RGB color space

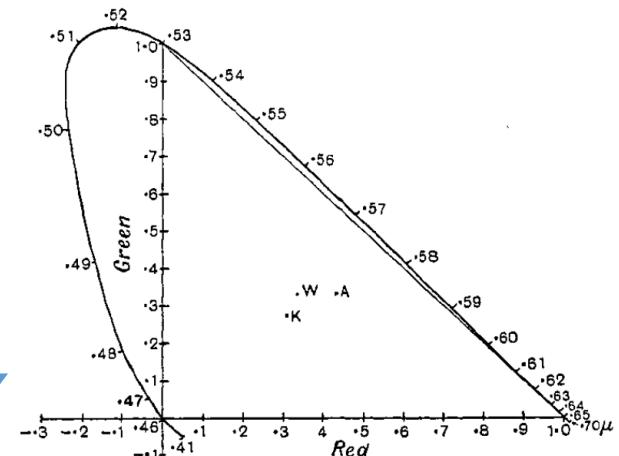
- **Tristimulus values:** amount of Red, Green and Blue to make a generic color
 - That is, not just the spectral colors





Tristimulus values: CIE RGB color space

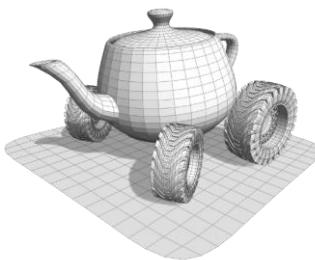
- The values for R, G, and B are found by **projecting** the PSD on the color matching functions
- RGB has negative values to represent colors near the spectral colors



$$R = \sum_{\lambda=380}^{\lambda=780} \bar{r}(\lambda)E(\lambda)$$

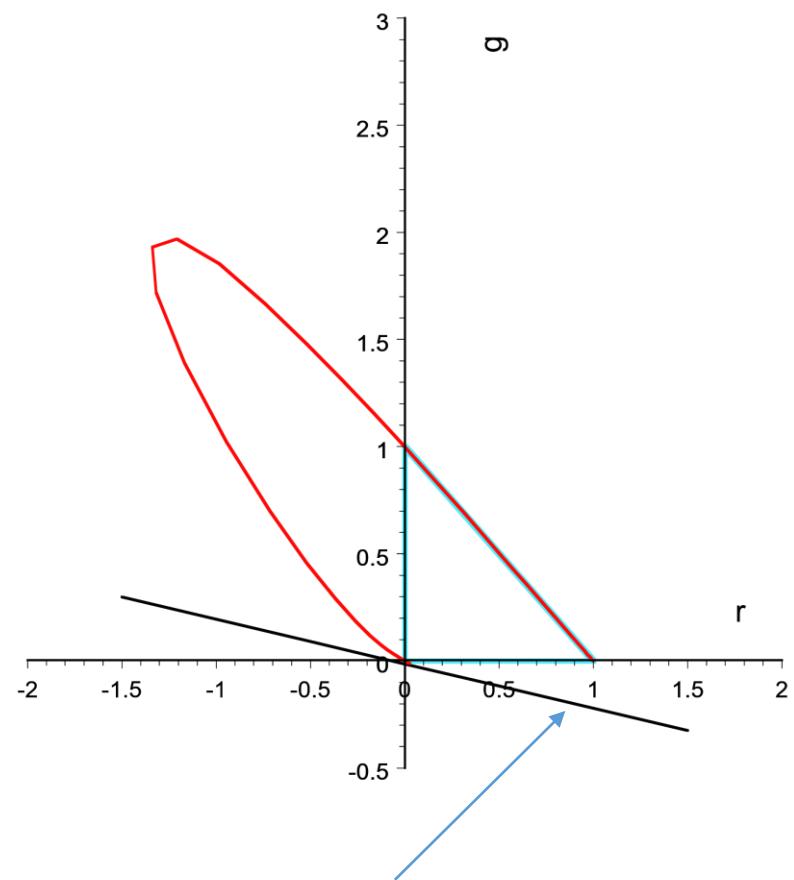
$$G = \sum_{\lambda=380}^{\lambda=780} \bar{g}(\lambda)E(\lambda)$$

$$B = \sum_{\lambda=380}^{\lambda=780} \bar{b}(\lambda)E(\lambda)$$

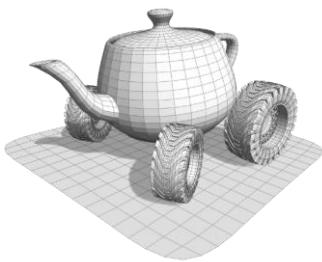


CIE XYZ color system

- CIE XYZ is a transformation of CIE RGB so that
 - Every visible color is specified by positive coordinates
 - Luminance is only encoded in one parameter (Y)
- It's a purely geometric transformation

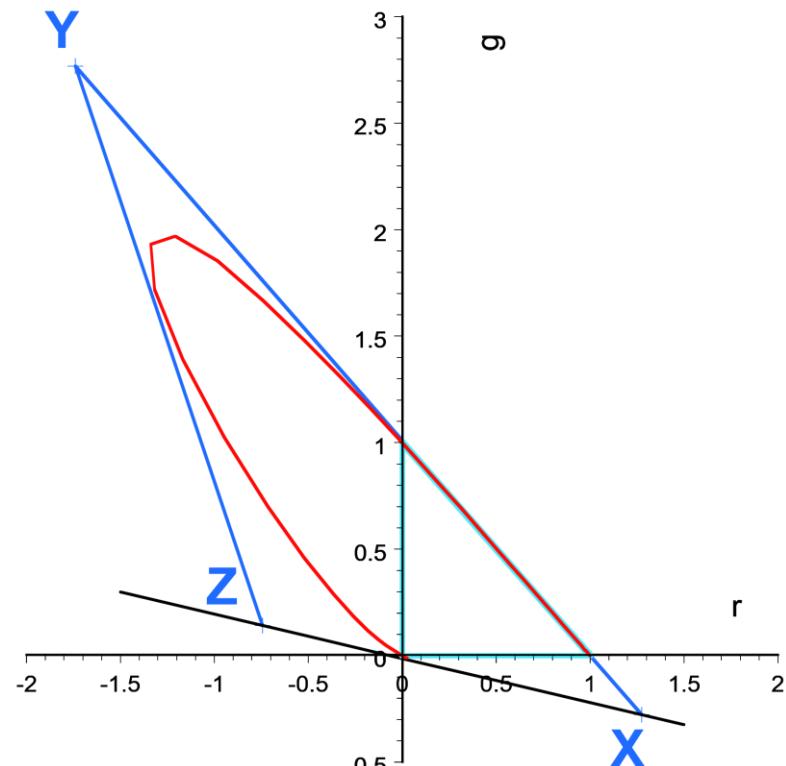


Alychine: line of zero luminance. It is found by solving $L(r,g,b) = 0$



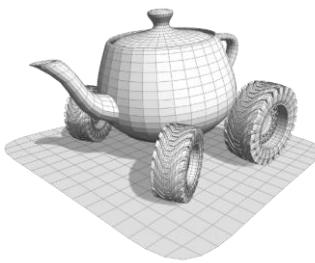
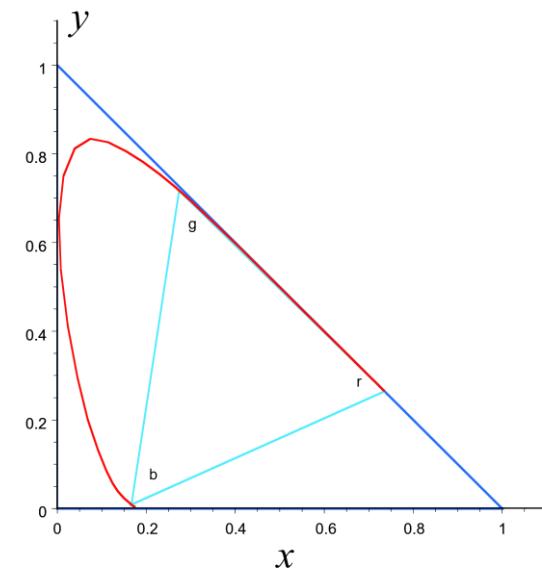
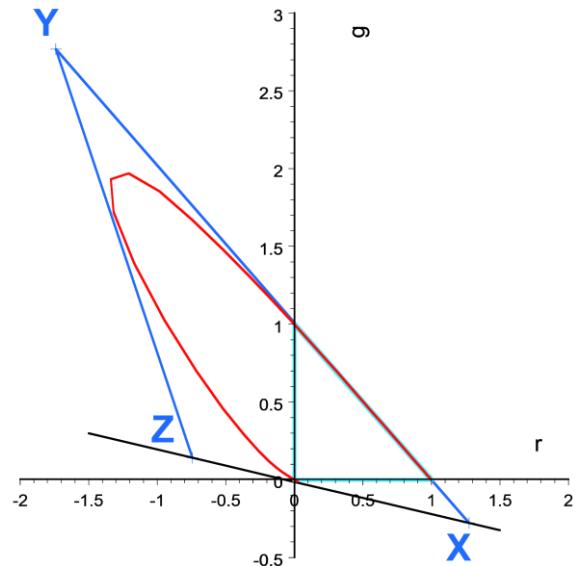
CIE XYZ color space

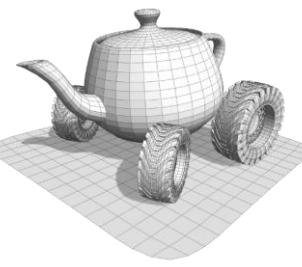
- CIE XYZ is a transformation of CIE RGB so that
 - Every visible color is specified by positive coordinates
 - Luminance is all encoded in one parameter (Y)
- It's a purely geometric transformation
- X,Y and Z are **imaginary colors**
 - X and Z don't even have luminance



CIE XYZ color space

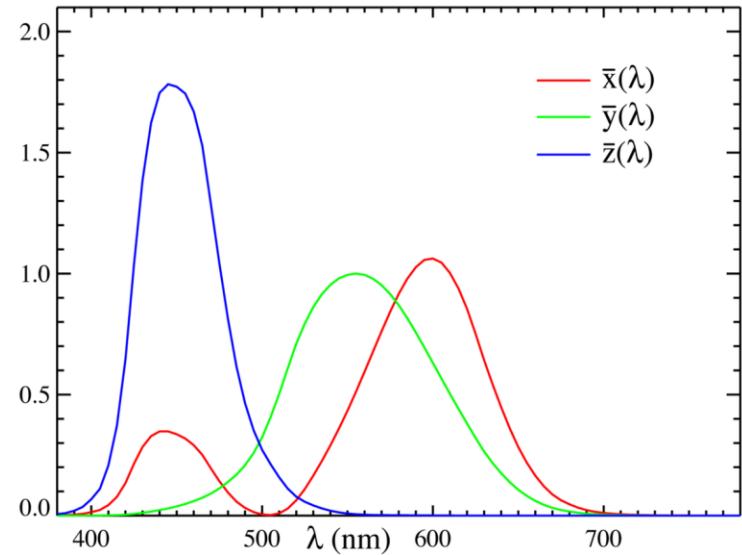
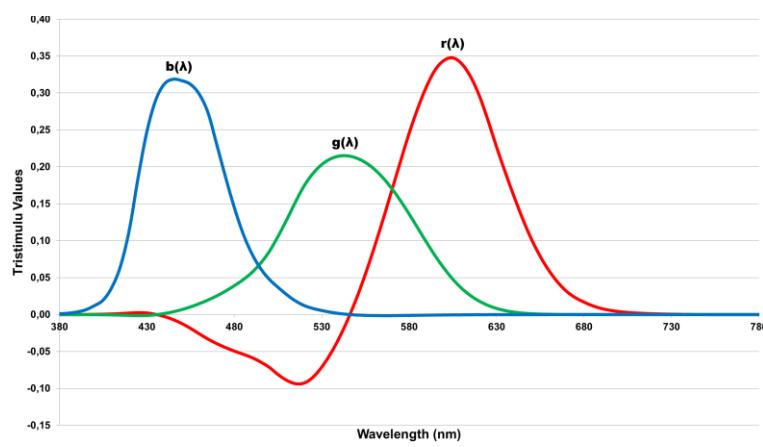
- CIE XYZ is a transformation of CIE RGB so that
 - Every visible color is specified by positive coordinates
 - Luminance is only encoded in one parameter (Y)
- It's a purely geometric transformation
- X,Y and Z are **imaginary colors**
 - X and Z don't even have luminance

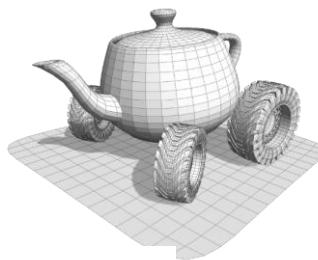




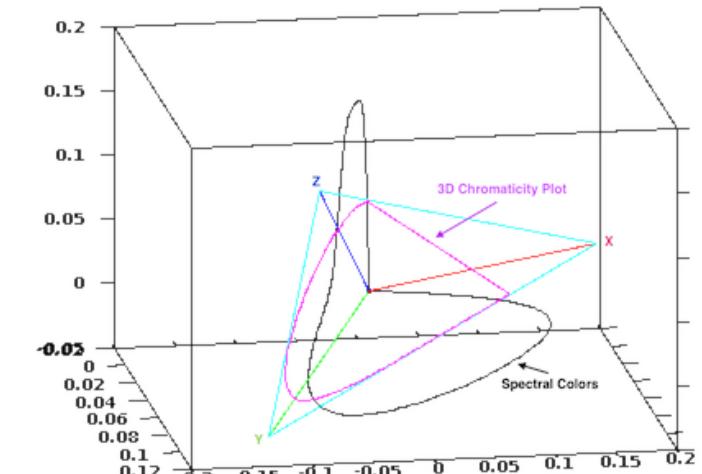
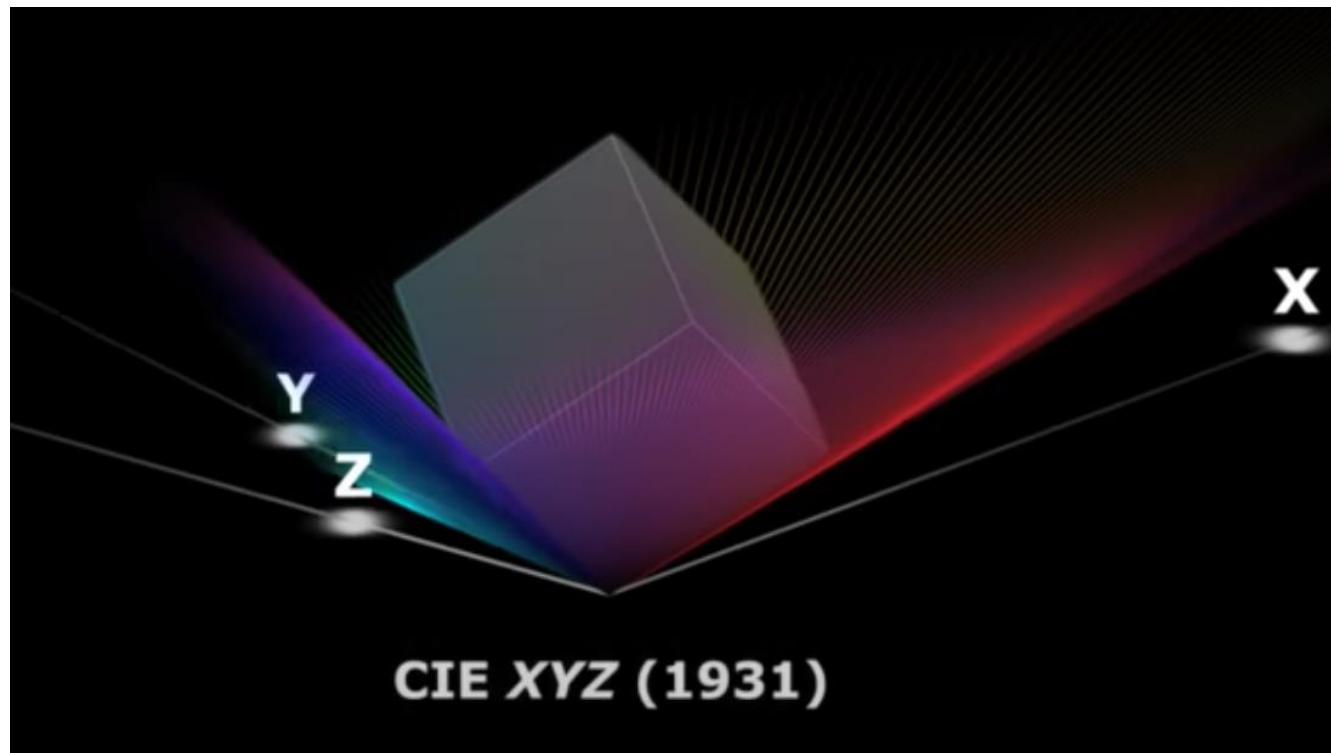
Color matching functions

- CIE RGB
 - Negative values
- CIE XYZ
 - Only positive values

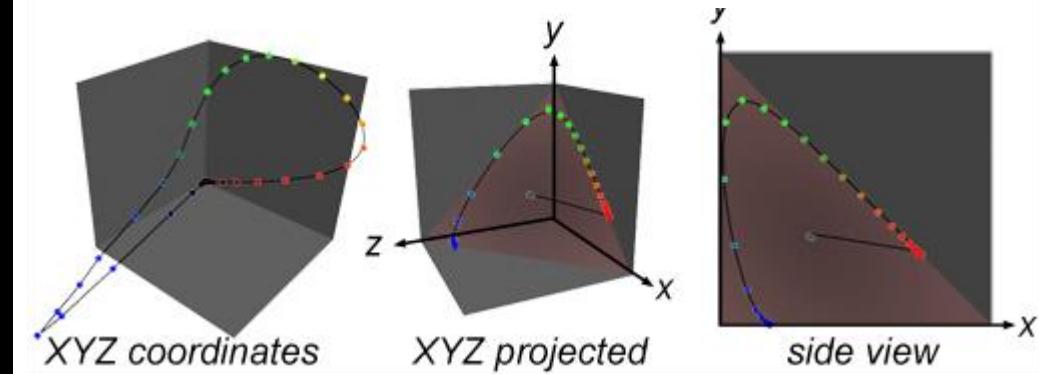




Color spaces CIE RGB and CIE XYZ



1apixel.com



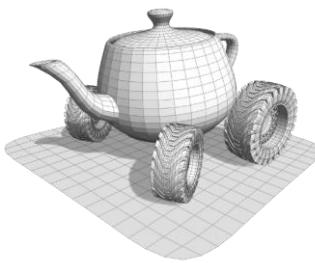
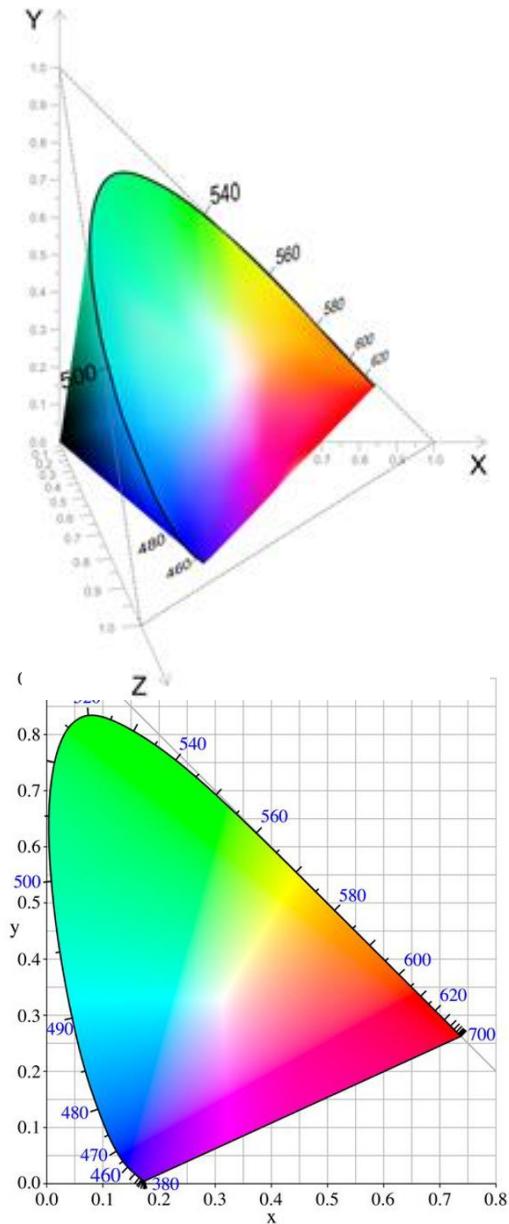
Youtube - @jselan

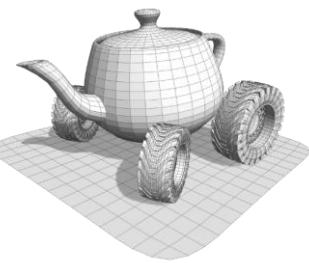
CIE xy chromaticity diagram

- Normalizing XYZ we obtain

$$x = \frac{X}{(X+Y+Z)} \quad y = \frac{Y}{(X+Y+Z)} \quad z = \frac{Z}{(X+Y+Z)}$$

- x and y are enough to specify the hue and saturation
- A complete color is specified by xyY
 - Y changes the luminance
 - xy changes the hue and saturation





Conversions

From XYZ to xyY

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

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

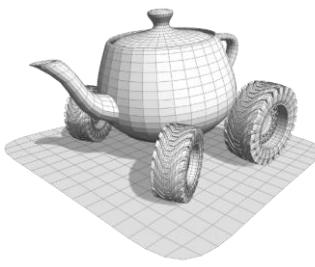
$$Y = Y$$

From xyY to XYZ

$$X = \frac{xY}{y}$$

$$Y = Y$$

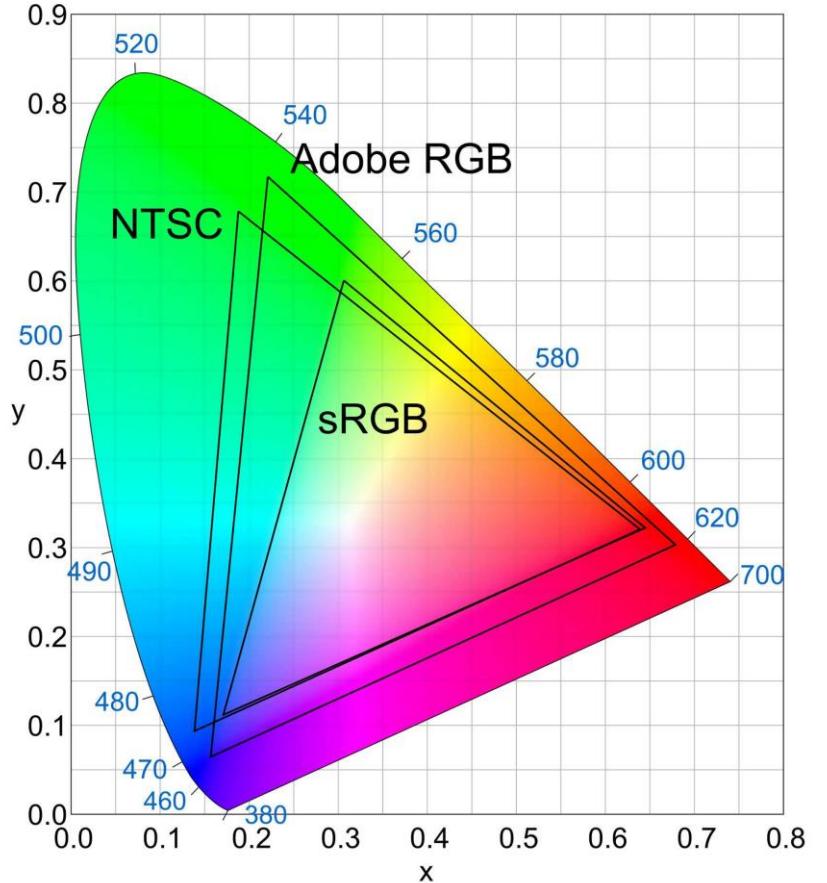
$$Z = \frac{(1 - x - y)Y}{y}$$



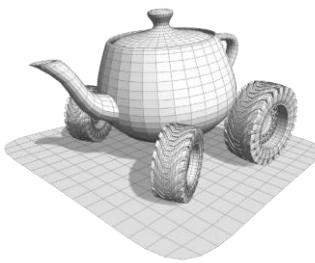
Gamut

- CIE XYZ includes **all** visible colors. Are our devices able to reproduce them all?
- **Gamut:** the set of colors defined by a color space / produced by a device/ perceived by an observer
- sRGB (HP & MS 1996, 's' is for standard) is the most commonly used

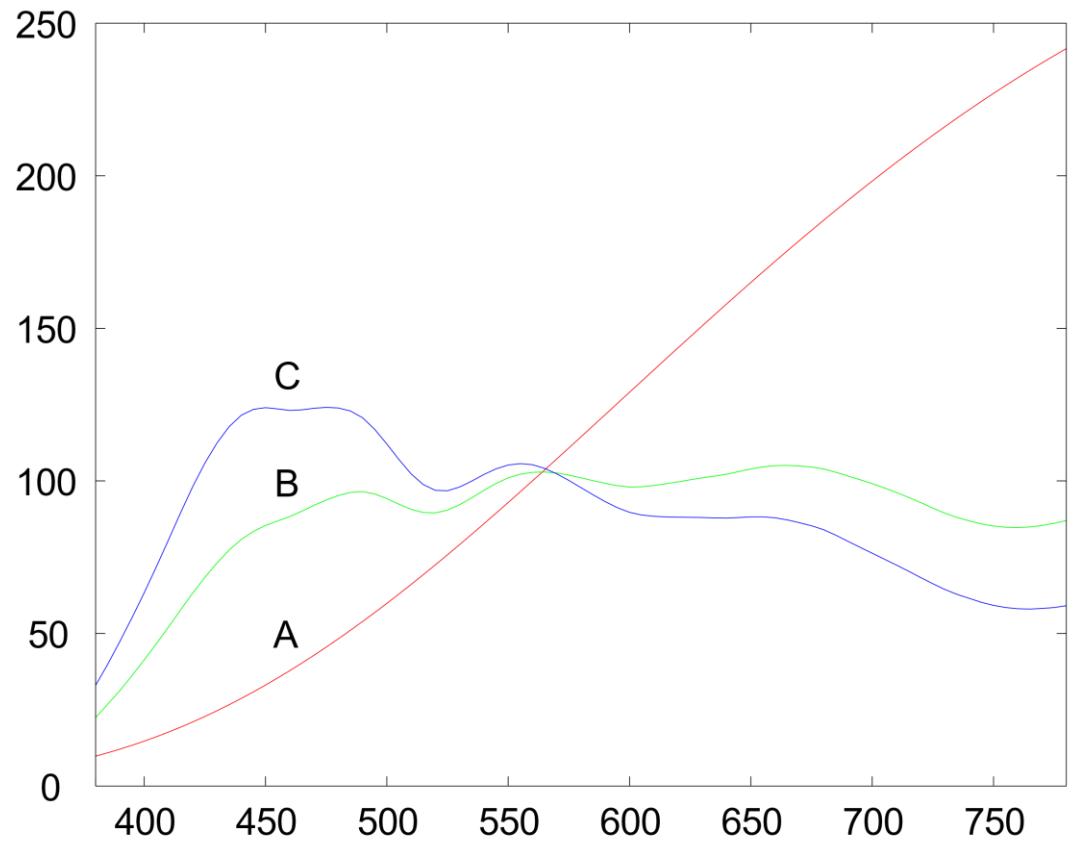
Chromaticity diagram

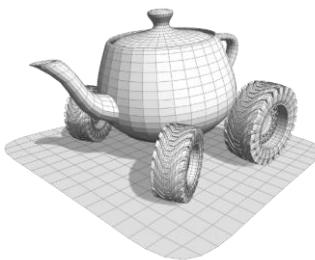


The illuminant



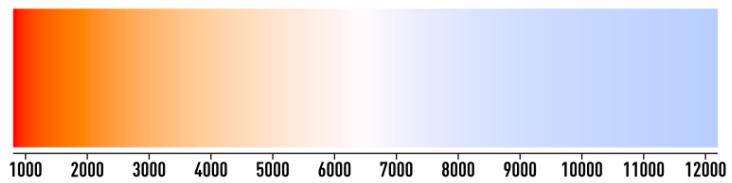
- **White point:** the tristimulus values associated to the illuminant
- The illuminant is a part of a color space. NPL was used in the original experiments. Nowadays we have many
 - A: tungsten-filament bulb lighting
 - B,C: daylight at noon and average
 - D series: currently used illuminants



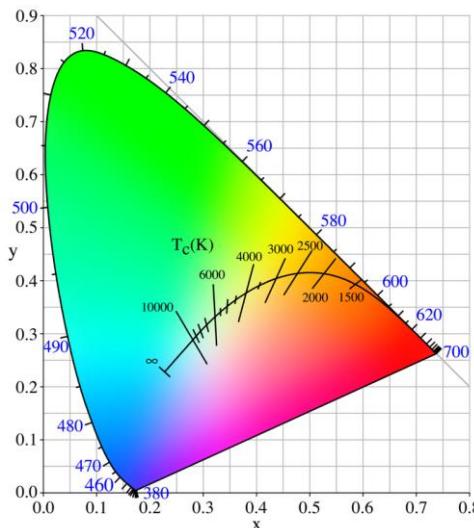
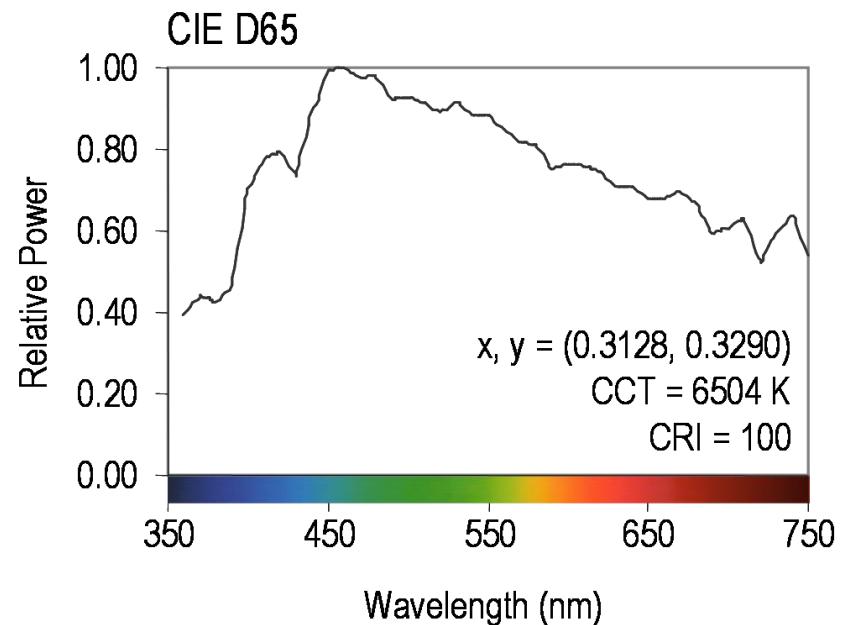


D65 Illuminant

- '65' stands for hundreds of Kelvins
- **Kelvin:** temperature (= Celsius + 273.15)
- **Black-body radiation:** temperature-dependent electromagnetic radiation of an opaque, non-reflective body



"color" of a black body as a function of its temperature

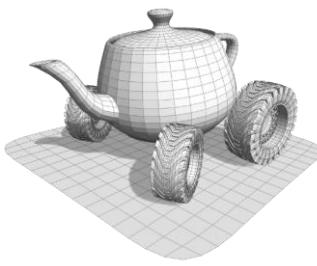


suggerimento

- Bel documentario su come gli altri animali sfruttano il colore per sopravvivere



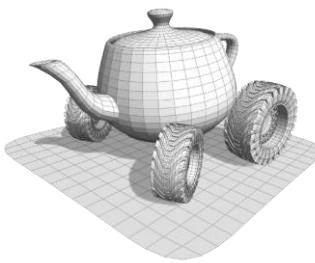
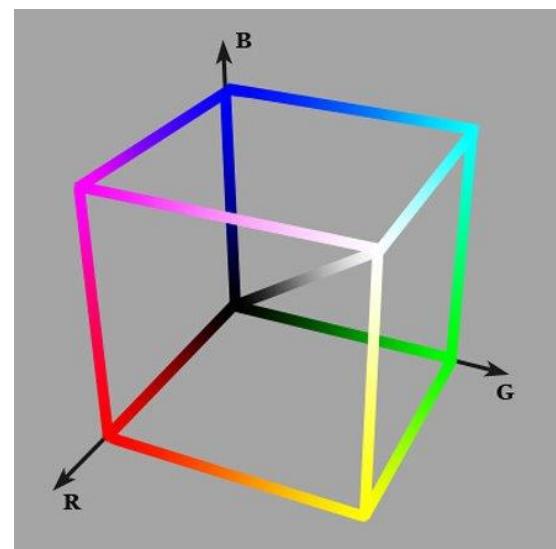
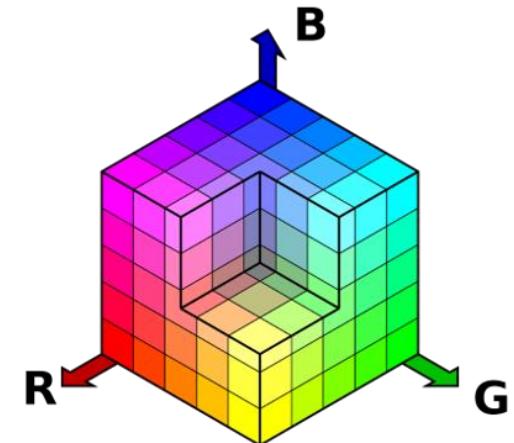
Practical Color Spaces



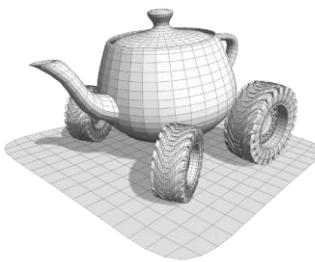
- CIE XYZ is a standard for encoding colors, but not very practical for:
 - picking a color for your digital drawing
 - Creating the color with a physical device (that is, the monitor)

Color Spaces: RGB

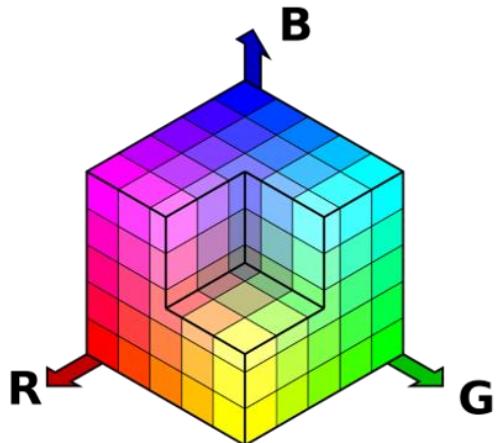
- The RGB color space is represented as a orthogonal reference system with axis Red, Green and Blue.
- The values range in [0,0,0] to [1,1,1]
- A every point in this range is a color
 - (0,0,0) is **black**
 - (1,1,1) is white
 - (v,v,v) is?



Color Spaces: RGB

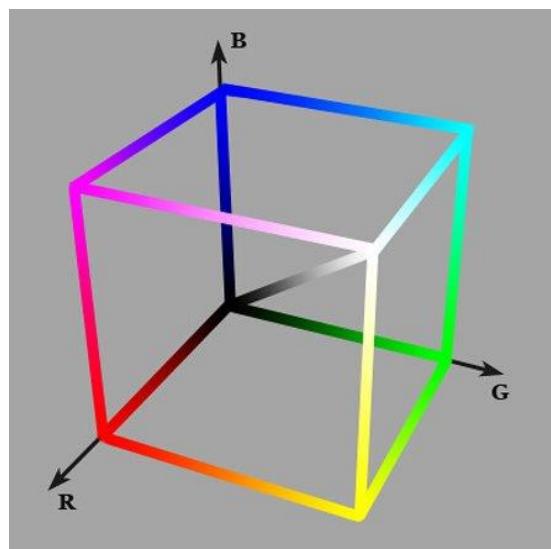


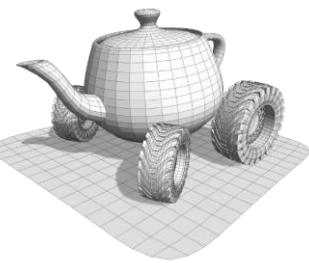
- Fully saturated primary colors correspond to the three corners of the cube



- What's in the remaining three?

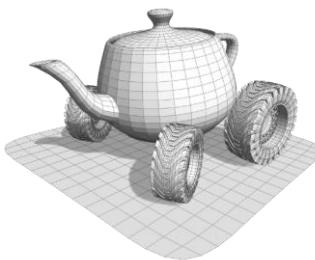
- Cyan = $(1,1,1) - (1,0,0)$
- Magenta = $(1,1,1) - (0,1,0)$
- Yellow = $(1,1,1) - (0,0,1)$





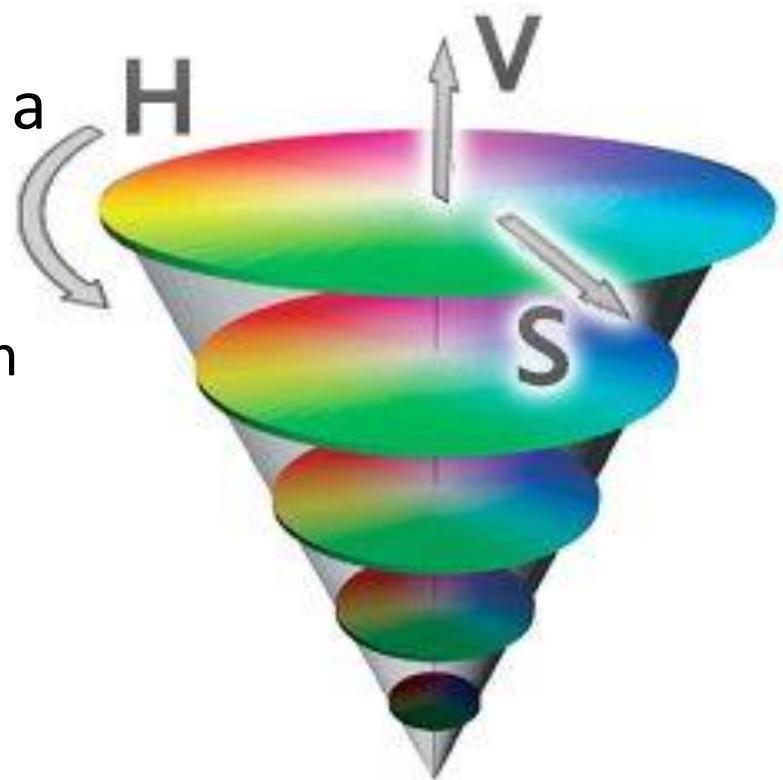
Color Spaces: RGB

- Color space RGB «mimics» our vision system by using quantities directly related to our cone receptors.
- It encode color
- But it not very intuitive when it comes to choosing a color
 - Beside the obvious one or except if you spent ages using it
 - Try finding RGB for brown or pink

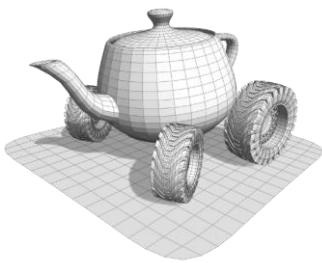


Color Spaces: HSV

- **Hue, Saturation and Value** may be thought as they way we would combine a pure color and a grey value (from black to white) to obtain a desired color
 - The percentage of grey determines the saturation (no grey -> saturation=1, all grey->saturation = 0)
 - The value of grey determines the lightness of the color (white-> a light *shade*, black-> a dark shade)

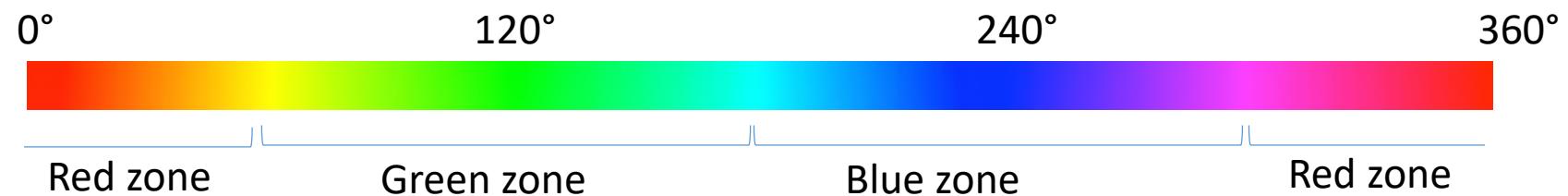


HSV palette demo <https://alloyui.com/examples/color-picker/hsv.html>



RGB to Hue

- The Hue is defined by a piecewise linear interpolation

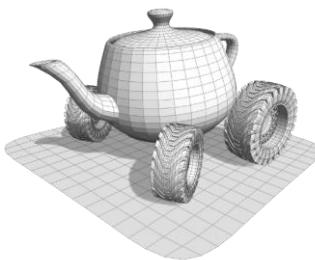


$$M = \max\{R, G, B\}$$

$$m = \min\{R, G, B\}$$

$$\Delta = M - m$$

$$H = \begin{cases} \text{undefined} & \text{if } \Delta = 0 \\ 60^\circ \left(\frac{G-B}{\Delta} \right) & \text{if } M = R \\ 60^\circ \left(\frac{B-R}{\Delta} \right) + 120^\circ & \text{if } M = G \\ 60^\circ \left(\frac{R-G}{\Delta} \right) + 240^\circ & \text{if } M = B \end{cases}$$



RGB to Saturation and Value

- Value is taken as the value of the highest channel
- Saturation is the ratio of Δ and Value.
 - It is 1 when $m = 0$

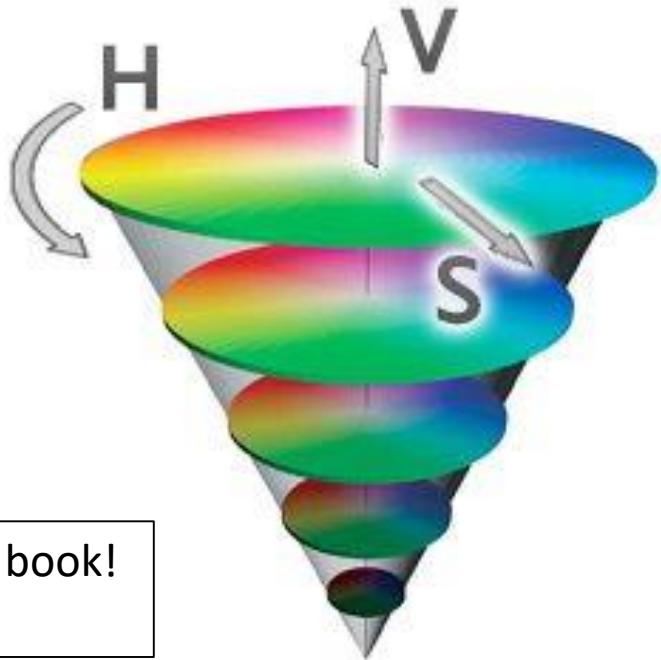
$$M = \max\{R, G, B\}$$

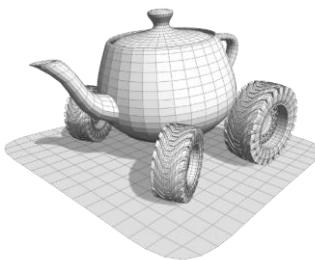
$$m = \min\{R, G, B\}$$

$$\Delta = M - m$$

$$S = \begin{cases} 0 & \text{if } V = 0 \\ \frac{\Delta}{V} & \text{otherwise} \end{cases}$$
$$V = M$$

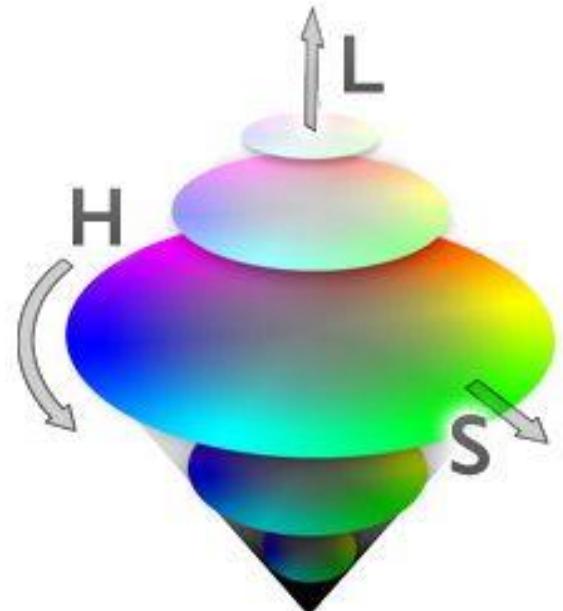
This is wrong in the book!
 Δ instead of V





Color Spaces: HSL

- **Hue Saturation Lightness** is a similar color spaces
 - Hue is exactly the same
 - $L > 0.5$: saturation decreases towards brighter colors
 - $L < 0.5$: saturation decreases towards darker colors



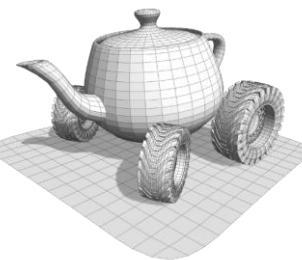
$$M = \max\{R, G, B\}$$

$$m = \min\{R, G, B\}$$

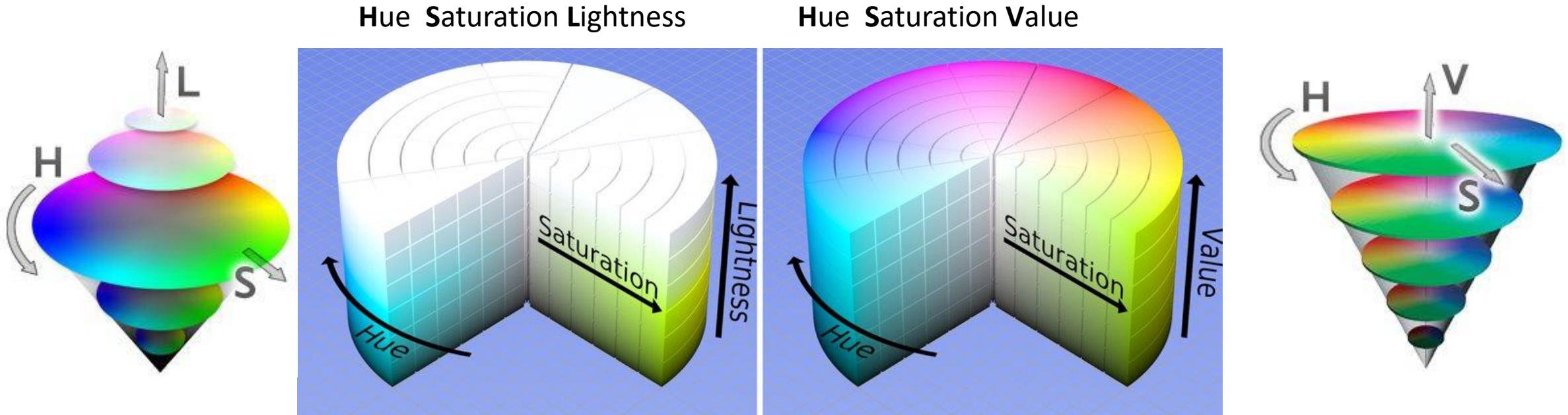
$$\Delta = M - m$$

$$L = \frac{M + m}{2}$$
$$S = \begin{cases} 0 & \text{if } \Delta = 0 \\ \frac{\Delta}{1 - |2L - 1|} & \text{otherwise} \end{cases}$$

[RGB to HSL converter | color conversion \(rapidtables.com\)](https://www.rapidtables.com/color/rgb-to-hsl.html)



Color Spaces: HSL & HSL comparison



from RGB to →

$$L = \frac{M+m}{2}$$

$$S = \begin{cases} 0 & \text{if } \Delta = 0 \\ \frac{\Delta}{1-|2L-1|} & \text{otherwise} \end{cases}$$

$$H = \begin{cases} \text{undefined} & \text{if } \Delta = 0 \\ 60^\circ \left(\frac{G-B}{\Delta} \right) & \text{if } M = R \\ 60^\circ \left(\frac{B-R}{\Delta} \right) + 120^\circ & \text{if } M = G \\ 60^\circ \left(\frac{R-G}{\Delta} \right) + 240^\circ & \text{if } M = B \end{cases}$$

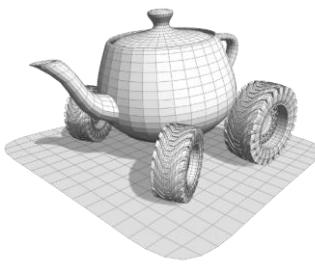
$$S = \begin{cases} 0 & \text{if } V = 0 \\ \frac{\Delta}{V} & \text{otherwise} \end{cases}$$

$$V = M$$

$$M = \max\{R, G, B\}$$

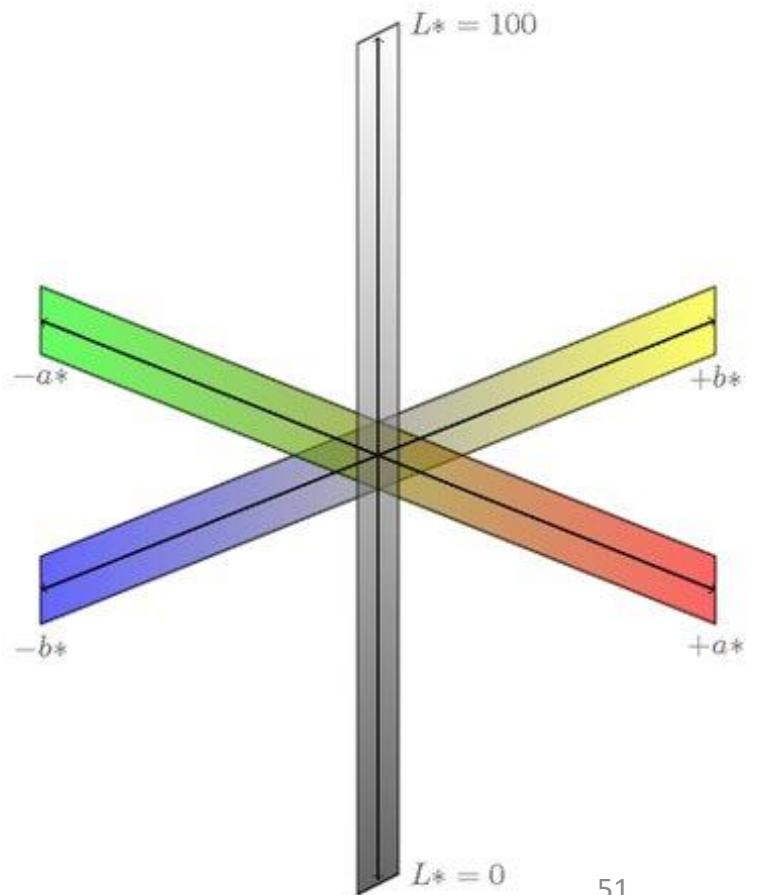
$$m = \min\{R, G, B\}$$

$$\Delta = M - m$$

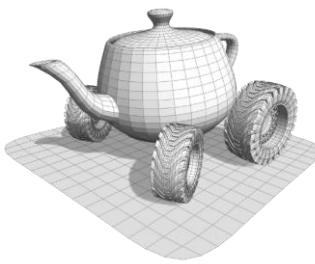


Color Spaces: CIELab

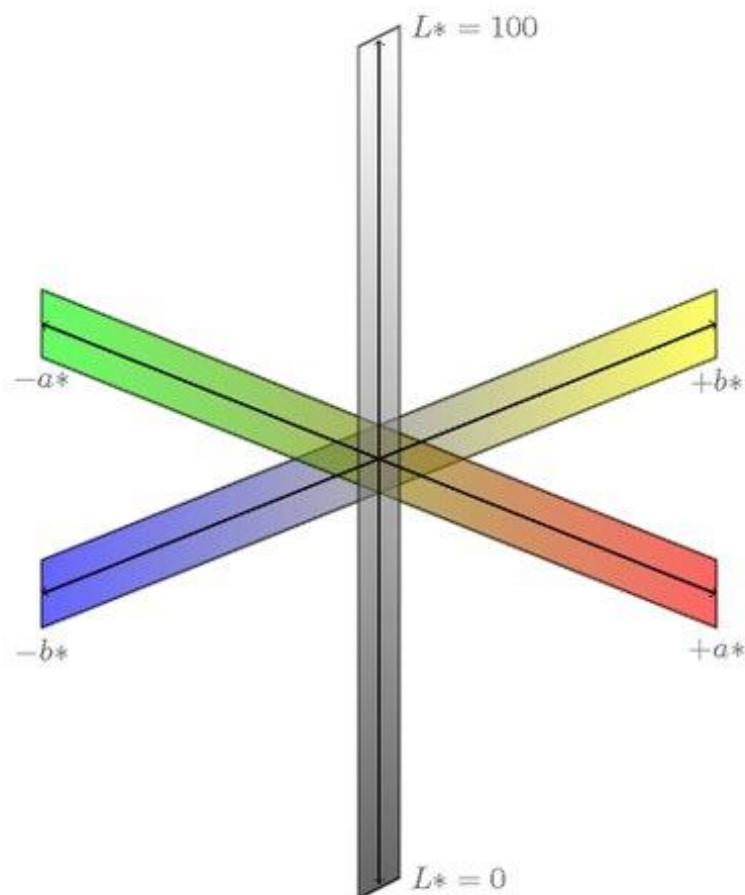
- In RGB, HSL and HSV the "distance" between two colors is not consistent with perception
 - e.g. distant colors may look similar
- **CIELab** (1976) was defined so that Euclidean distance between points in space (colors) reflected human perception
 - Distant colors in CIELab perceived different
 - Close colors in CIELab perceived similar

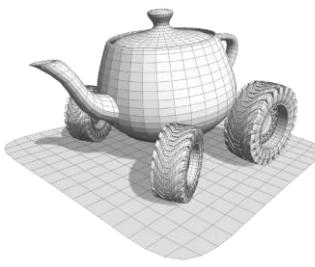


Color Spaces: CIELab



- CIELab color is defined as the triple L^* , a^* and b^*
 - L^* lightness, a^* and b^* chromaticity
- It defines the color by using the **opponent process** color theory that *cones* and *rods* are linked to form opponent signals:
 - From black to white (L^*)
 - From green to red (a^*)
 - From blue to yellow (b^*)





Color Spaces: CIELab

- CIELab is defined in terms of CIEXYZ
- $X_n, Y_n, \text{ and } Z_n$ are illuminant dependent normalization factors

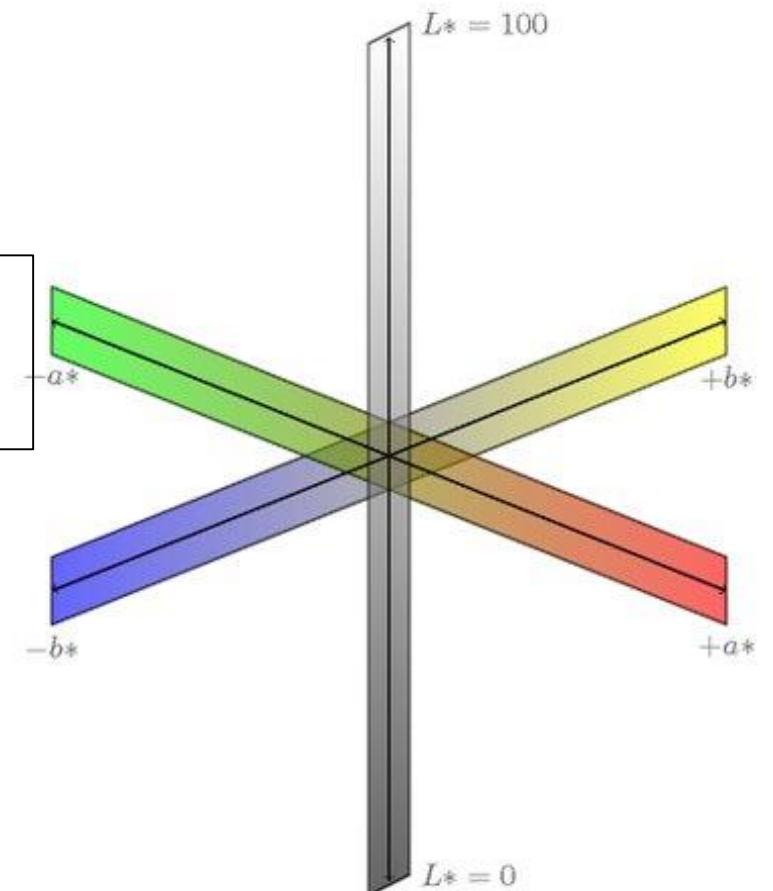
$$L^* = 116 f(X/X_n) - 16$$

$$a^* = 500 f(X/X_n) - f(Y/Y_n)$$

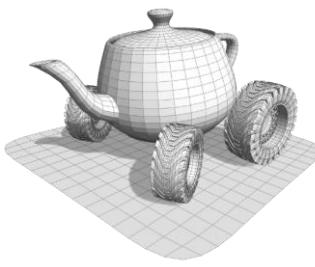
$$b^* = 200 f(Y/Y_n) - f(Z/Z_n)$$

$$f(x) = \begin{cases} x^{1/3} & \text{if } x > 0.008856 \\ 7.787037x + 0.137931 & \text{else} \end{cases}$$

Formule che **non** vi
verranno chieste
all'esame



<http://davidjohnstone.net/pages/Ich-lab-colour-gradient-picker>

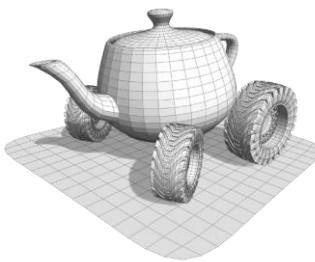


Color differences

- The Euclidean distance is meaningful from a perceptual point of view for CIE Lab color space.
 - Good for specifying color tolerances, color codes, *pseudocoloring* (using sequences of colors to represent data values, possibly with perceptually-equal steps)
- *Delta E (1976)* is defined as:

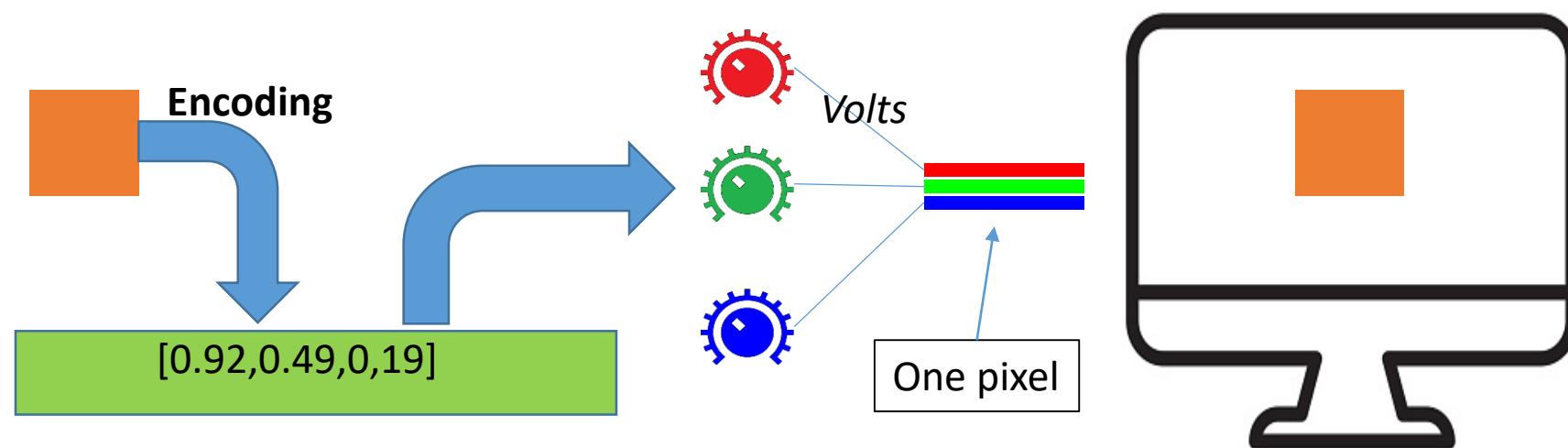
$$\Delta E_{ab}^* = \sqrt{(L_2^* - L_1^*)^2 + (a_2^* - a_1^*)^2 + (b_2^* - b_1^*)^2}$$

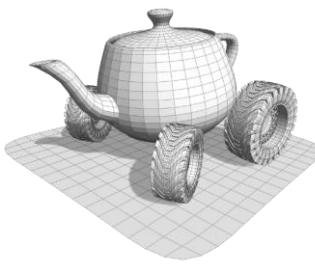
- $\Delta E < 1$: not perceptible, $1 < \Delta E < 2$ close observation needed to perceive the difference, $2 < \Delta E < 10$ different but similar color
- *Note:* Delta E is not perceptually uniform as originally intended, hence superseded by 1994 and 2000 specifications



Encoding and displaying colors

- **Encoding:** to decide what number to store to represent a certain color
- **Displaying:** to read the stored color and send the corresponding signal to display



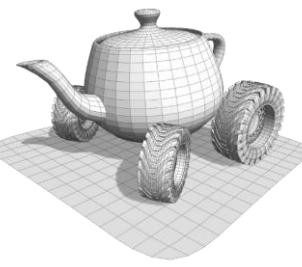


Encoding and displaying colors

- Eye is sensitive to **ratios** of intensities, not to **absolute** intensities
- How do we space intensity so that they "look" uniformly spaced?

$$I_0, \quad I_1 = I_0 r, \quad I_2 = I_1 r = I_0 r^2, \dots, \quad I_n = I_0 r^n$$

- For $r > 1.01$: humans "notice" the change of intensity

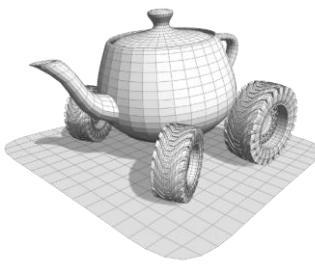


Encoding and displaying colors

- Let us say we have a device capable of displaying intensities between I_{\min} (darkest) and I_{\max} (brightest)
- how big needs n to be so that we can see a continuously changing intensity?

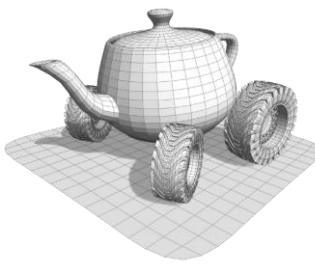
$$I_{\max} = I_{\min} r^n \quad \Rightarrow \quad n = \log_{1.01} \frac{I_{\max}}{I_{\min}}$$

- $\frac{I_{\max}}{I_{\min}}$ = **dynamic range**: ratio between **max** and **min** intensity



Encoding and displaying colors

- Typical dynamic ranges
 - Desktop display in typical conditions: 20:1
 - Photographic print: 30:1
 - Desktop display in good conditions: 100:1
 - High-end display under ideal conditions: 1000:1
 - Digital cinema projection: 1000:1
 - Photographic transparency (directly viewed): 1000:1
 - **High Dynamic Range** display: 10,000:1



Encoding and displaying colors

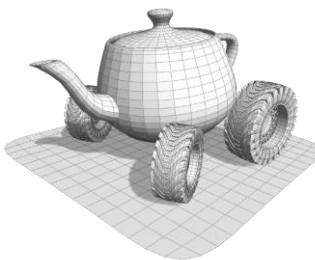
- Linear encoding: intensity values are just mapped proportionally
 - Lots of bits are wasted
- Let us map intensities to our levels $j, j+1, \dots, n$

$$j = \log_r \frac{I_{imm}}{I_0}$$

Value I want to display



- In the real world, for historical/engineering/inertial reasons, **gamma correction** is used

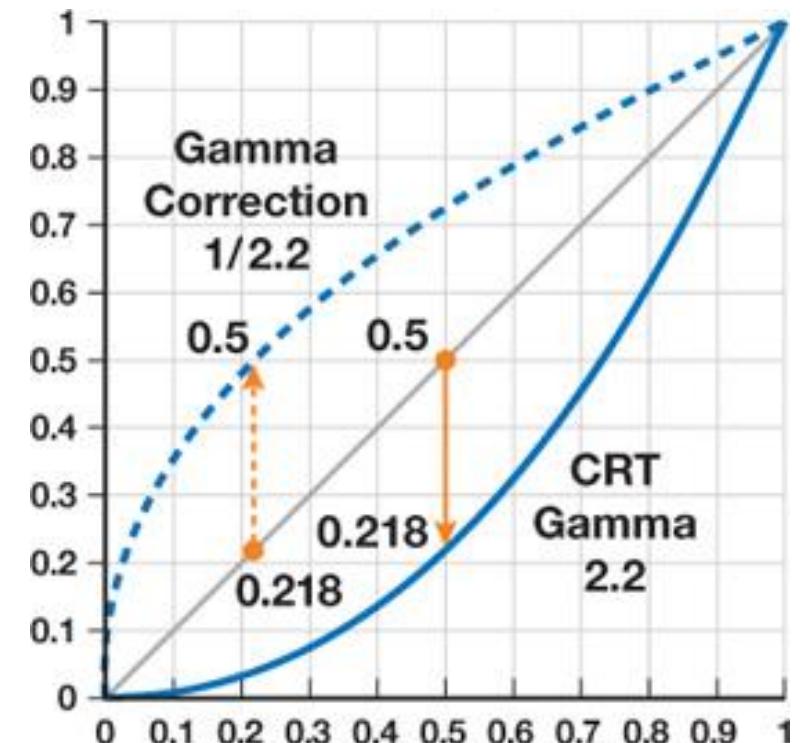


Gamma correction (the historical reason)

- Cathode-Ray Tube monitors show an exponential relation between voltage and intensity

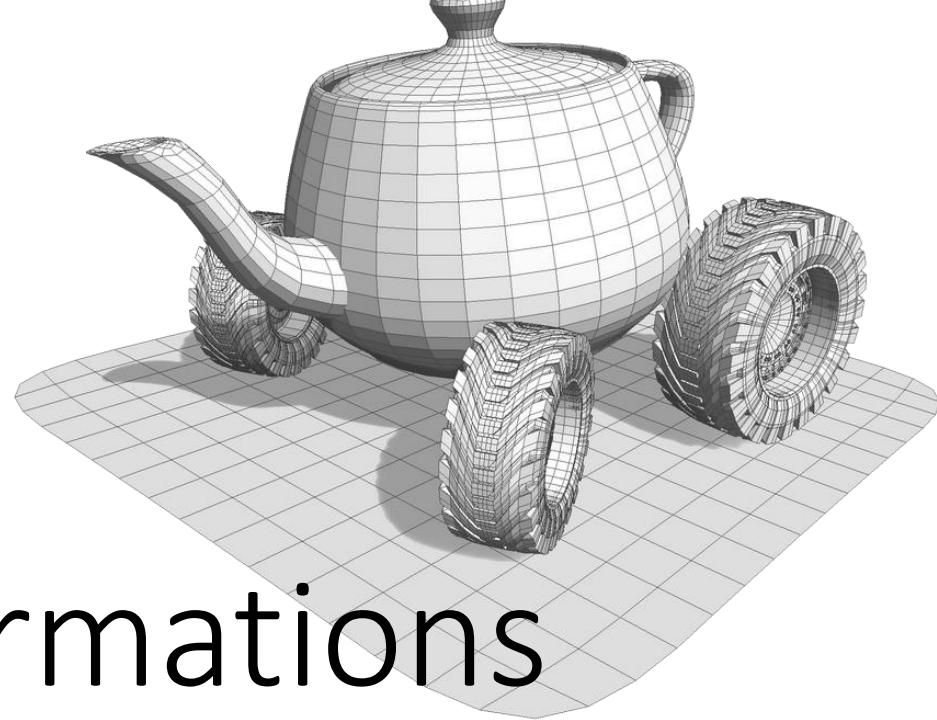
$$I_{output} = V^\gamma$$

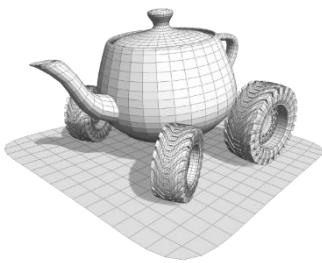
- You need to account for it by encoding $I = V^{\frac{1}{\gamma}}$ so that when it's displayed with the right intensity
- For Catode Ray Tube monitors: $\gamma = 2.2$



Geometric Transformations

At work





Rendering of a scene

for each object to be drawn:

Setup the transformation matrix to render it in the **proper place**, from the desired point of **view** and with the desired **projection**

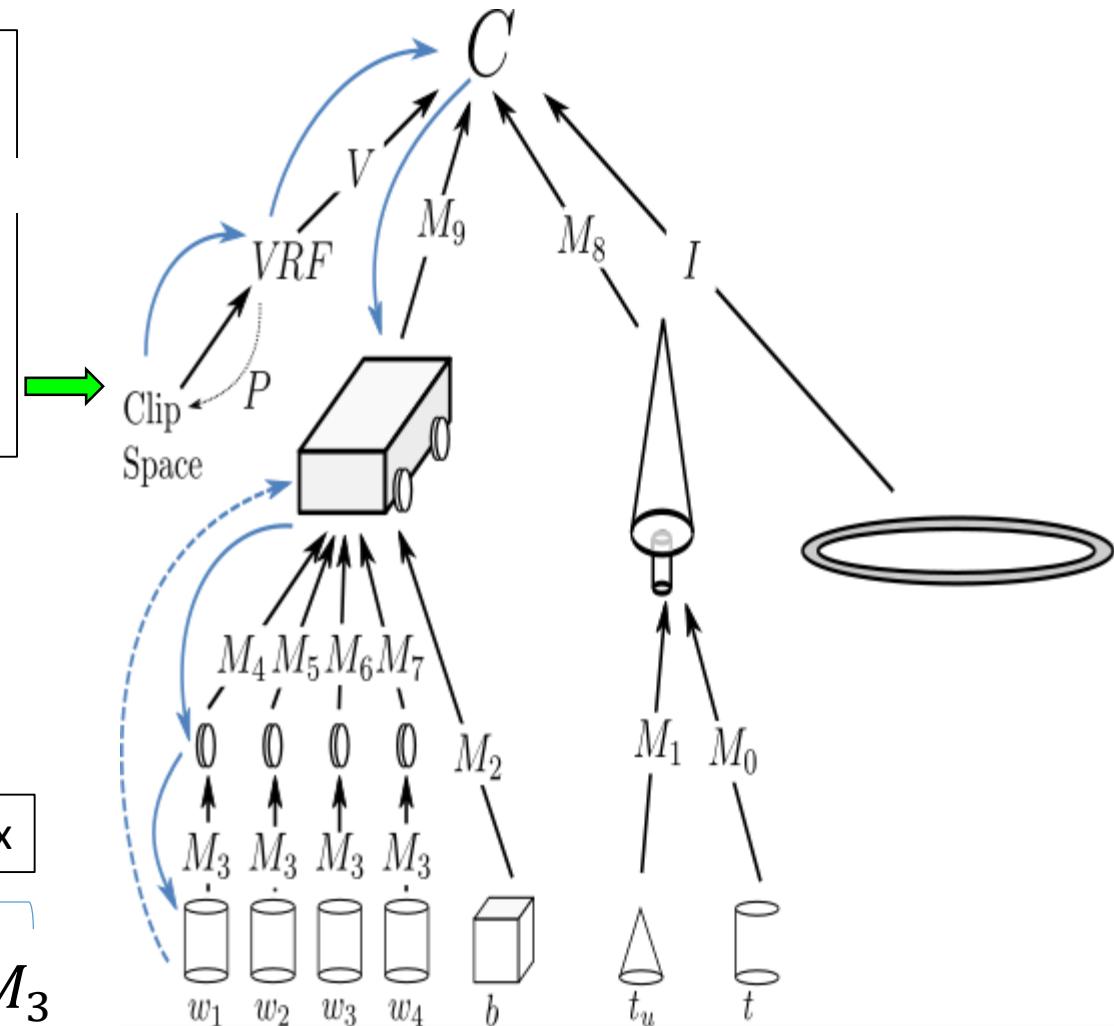
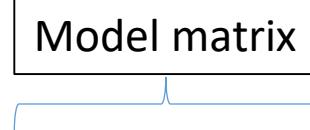
Draw the object

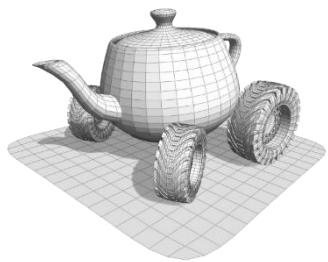
- The transformation matrix is obtained by visiting the scene graph from Clip Space to the object and cumulating matrix product on the right side

Example for wheel w1



$$P \cdot V^{-1} \cdot M_9 \cdot M_4 \cdot M_3$$





Rendering of a scene

Pseudo code for drawing the wheels

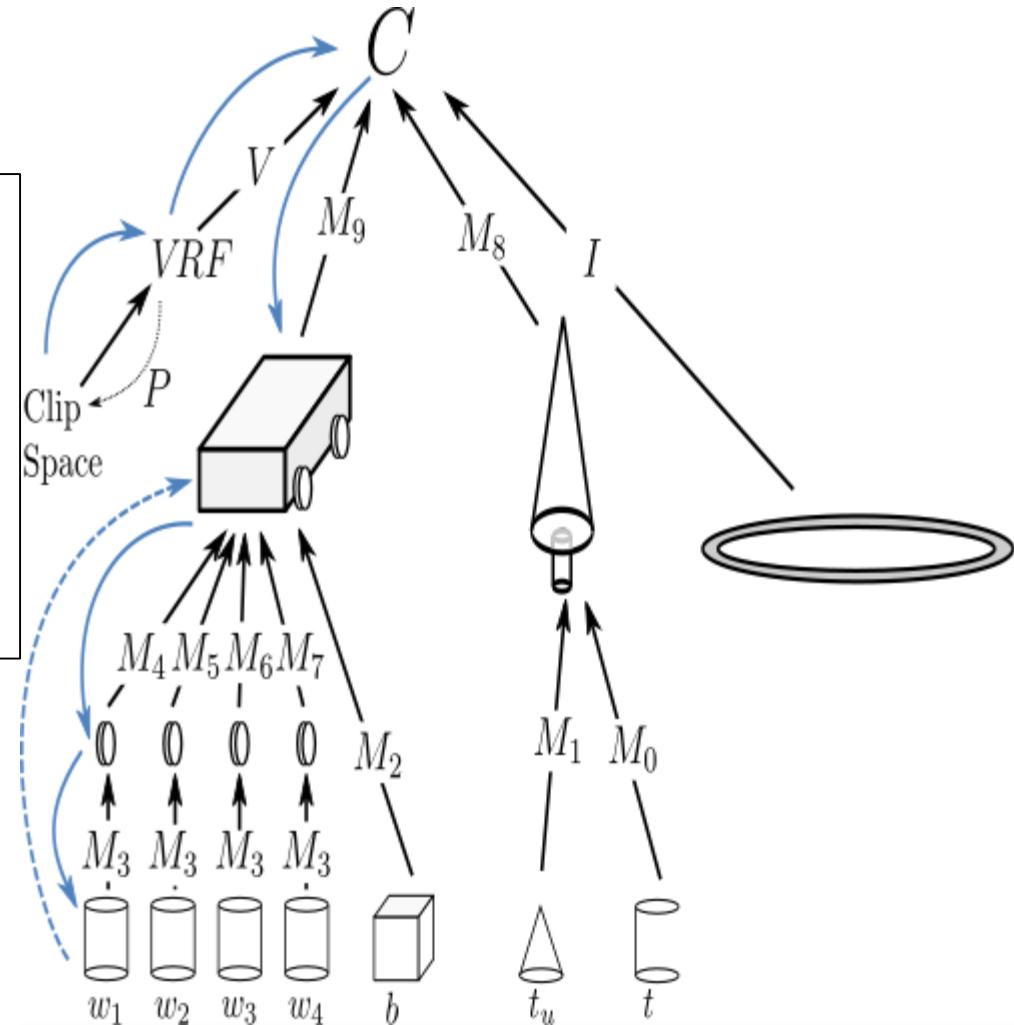
```
M = P*V_1*M9*M4*M3
glUniform4fv(LocM,1,false,M);
draw_wheel();
```

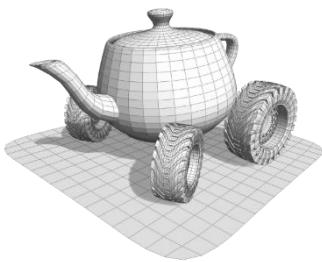
```
M = P*V_1*M9*M5*M3
glUniform4fv(LocM,1,false,M);
draw_wheel();
```

```
M = P*V_1*M9*M6*M3
glUniform4fv(LocM,1,false,M);
draw_wheel();
```

```
M = P*V_1*M9*M7*M3
glUniform4fv(LocM,1,false,M);
draw_wheel();
```

Matrices up to the deepest common ancestor are the same
There is no reason to remultiply them everytime





Rendering of a scene

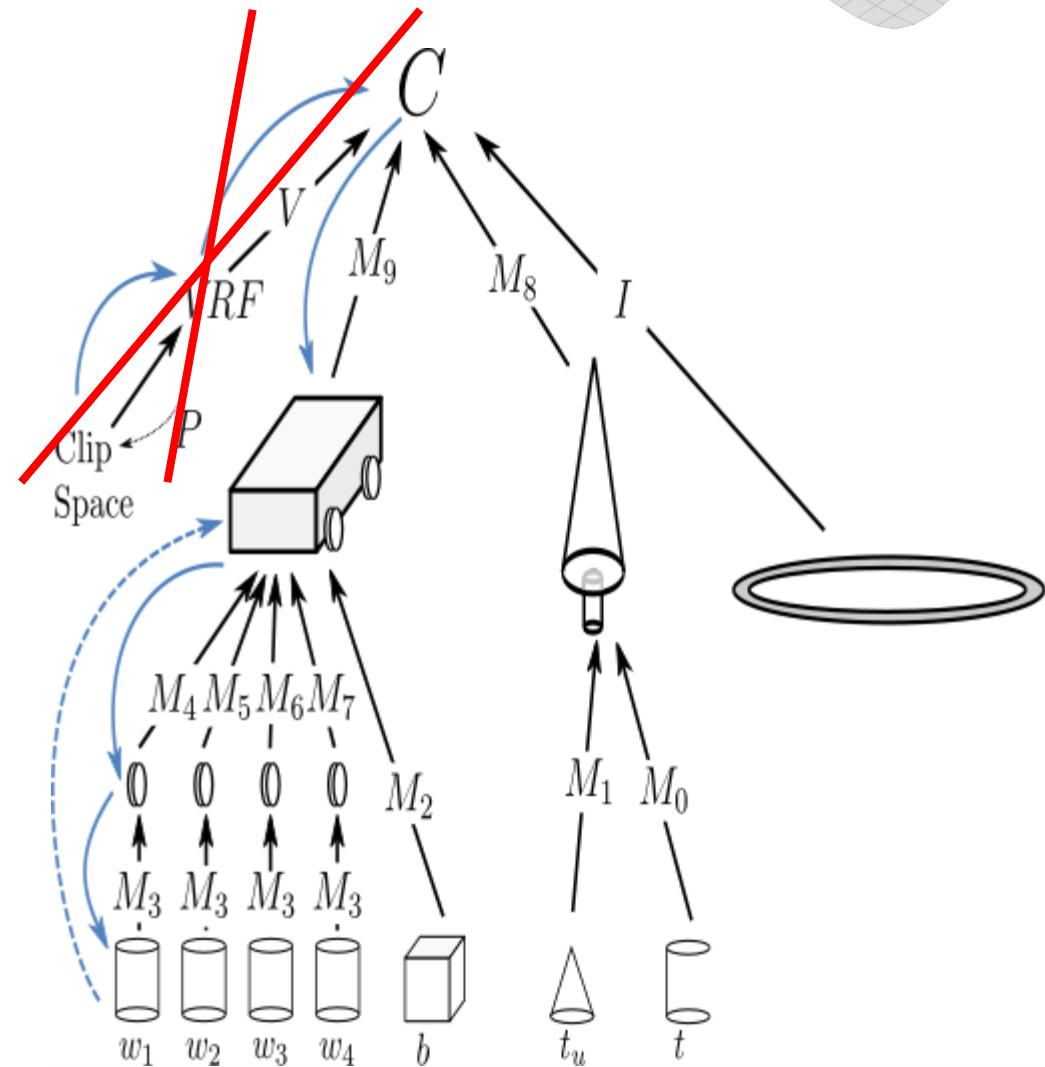
Pseudo code for drawing the scene

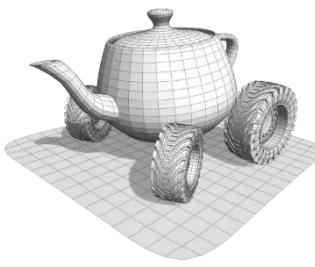
```

visit (currM,node){
    if(node is a leaf){
        draw(node);
    }else
        for each children c of node
            M = transformation from c to node;
            visit(currM*M,c);
    }
}

draw_scene(){
    PM = P*V^-1;
    node = C;
    visit(PM,node);
}
  
```

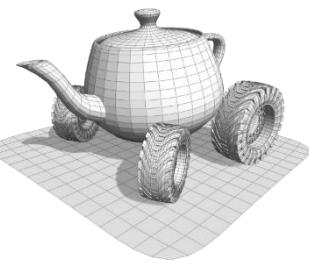
Thanks to recursion stack,
we are doing only the
necessary multiplications
only once





Matrix Stack (1/2)

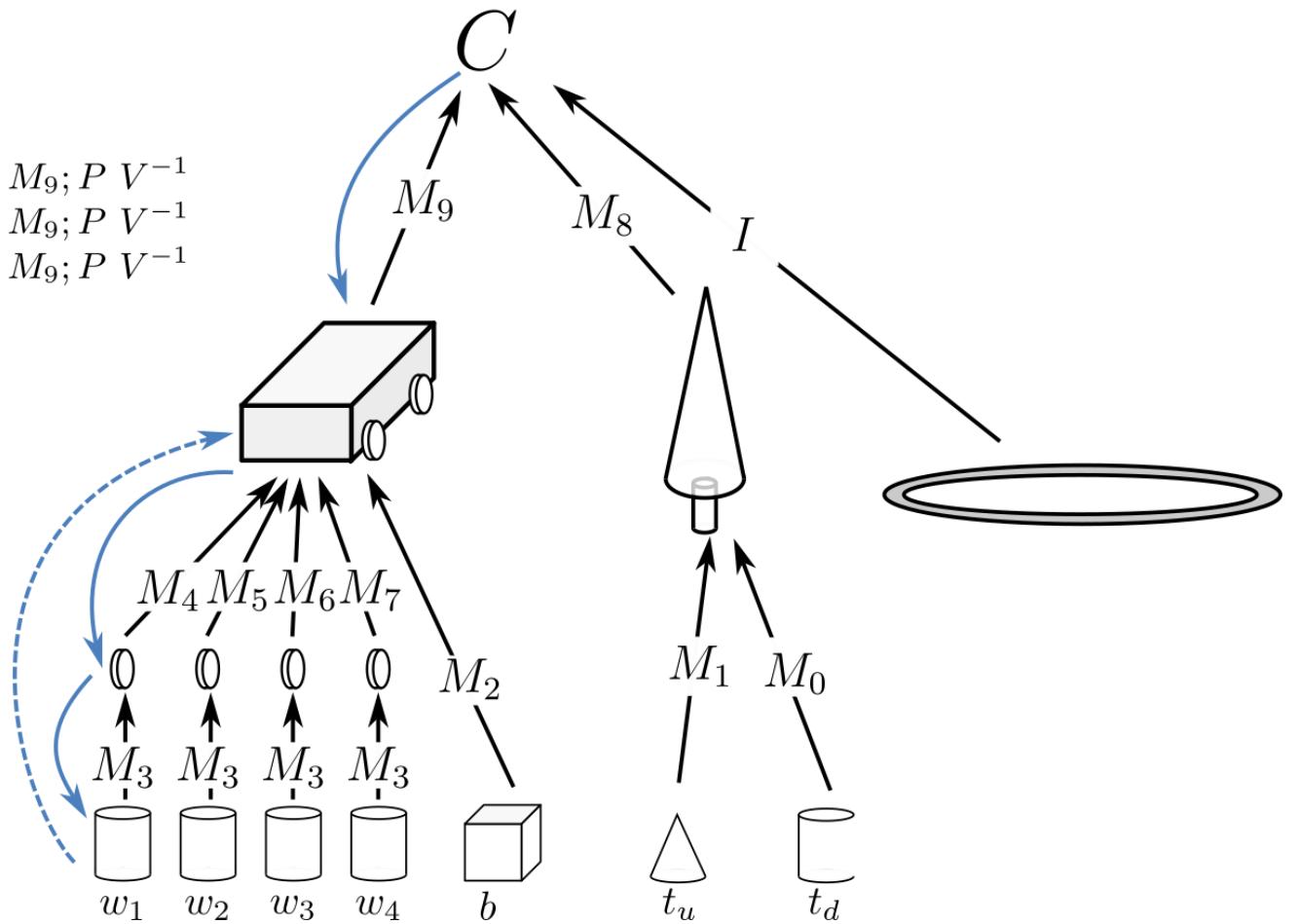
- Explicitly creating the graph data structure and implementing the visit is unnecessarily cumbersome for our goal, we just need the «stack»
- A **matrix stack** is a simple LIFO data structure to store the matrices as per our needs. The top of the stack is referred to as **current matrix** and it's the matrix that will be used at drawing time
 - `stack.load(M)`: replace the **current** matrix with `M`
 - `stack.mul(M)`: replace the **current** matrix with `current*M`
 - `stack.push()`: add one level to a stack and put the current on it
 - `stack.pop()`: remove the top of the stack

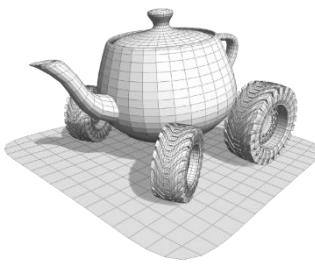


Matrix stack (2/2)

Op	Matrix stack	
	current	
set $P V^{-1}$	$P V^{-1}$	\emptyset
push	$P V^{-1}$	\emptyset
mul M_9	$P V^{-1} M_9$	$P V^{-1}$
push	$P V^{-1} M_9$	$P V^{-1} M_9; P V^{-1}$
mul M_4	$P V^{-1} M_9 M_4$	$P V^{-1} M_9; P V^{-1}$
mul M_3	$P V^{-1} M_9 M_4 M_3$	$P V^{-1} M_9; P V^{-1}$
draw w_1	.	.
pop	$P V^{-1} M_9$	$P V^{-1}$

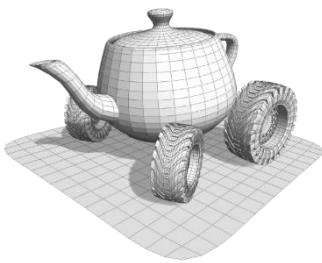
this matrix is used as model matrix when w_1 is drawn





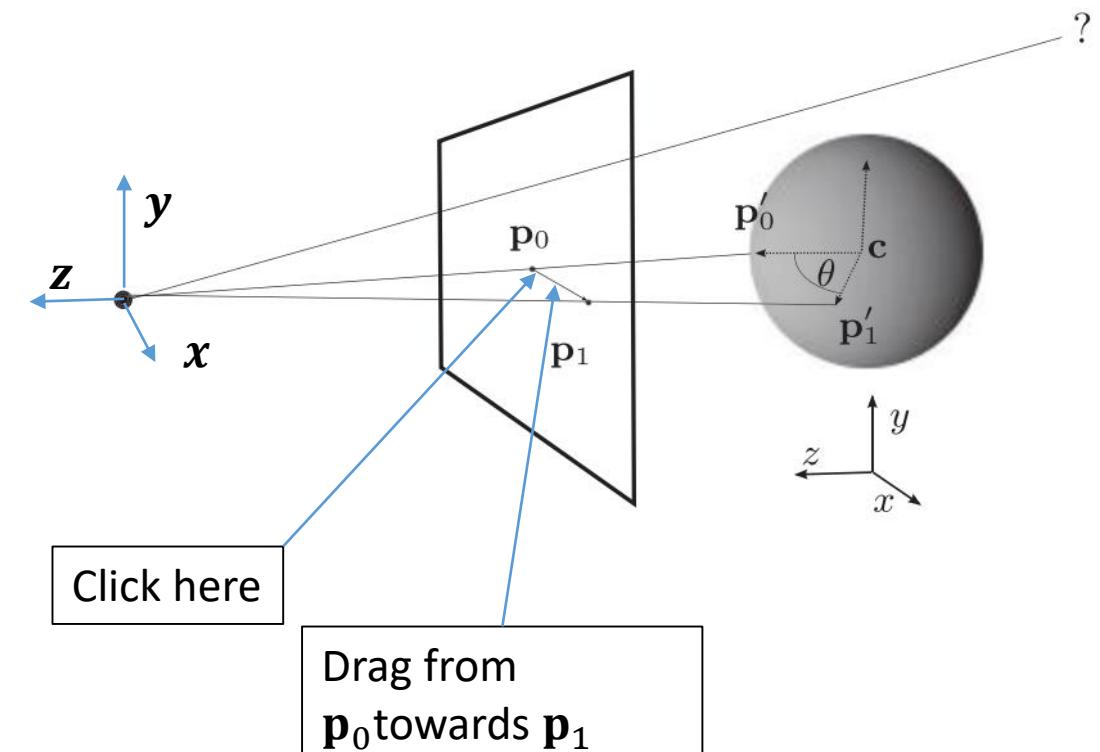
Watching the scene

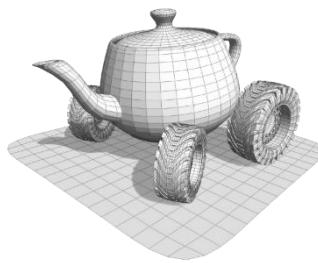
- In any application we need a way to watch the scene
 - Specifying the point and direction of view (Model View Matrix)
 - Specify the camera (Projection Matrix)
 - For our simple pinhole camera
- Two paradigms:
 - **World-in-hand:** we hold the scene in the palm of our hands and rotate, translate and scale as we please (although scaling breaks the metaphor).
 - Typical in the CAD applications
 - **Camera-in-hand:** we hold the camera and go around the scene
 - Typical in video games (e.g. first person shooter)



World-in-hand: the Virtual Trackball

- **Virtual Trackball** idea: we place an ideal (virtual) sphere in front of viewer
 - That is, down the $-z$ direction
- We consider the sphere “glued” to the world.
 - Rotating the sphere means rotating the world
- We make the sphere rotate by click&drag





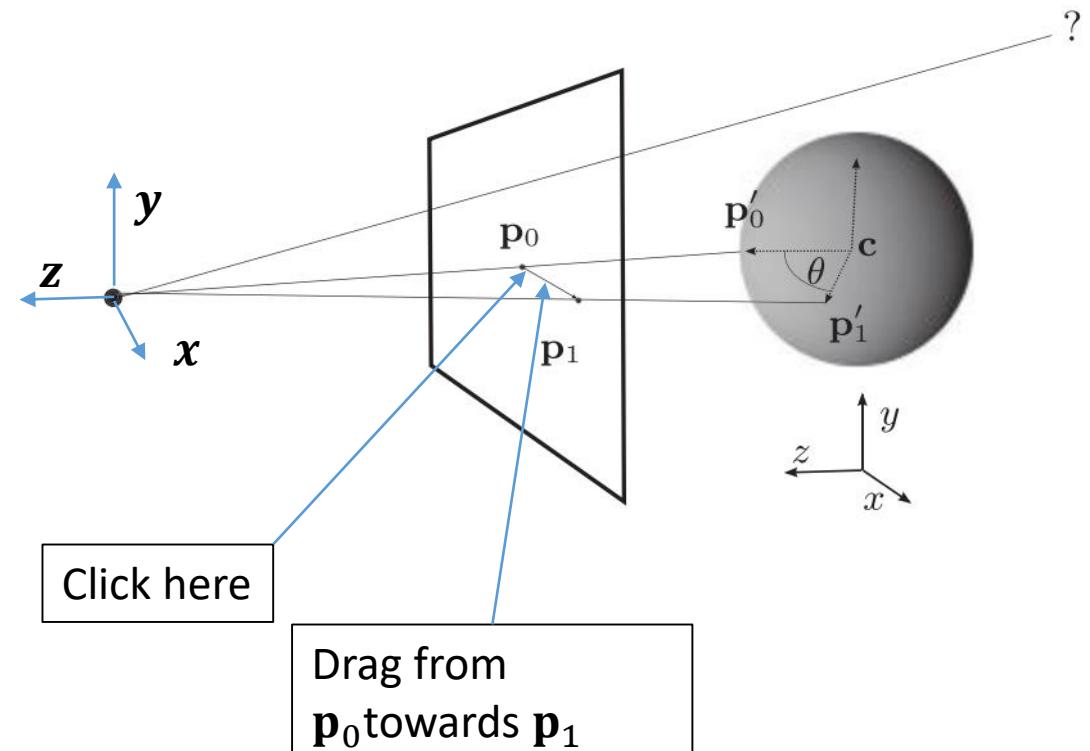
World-in-hand: the Virtual Trackball

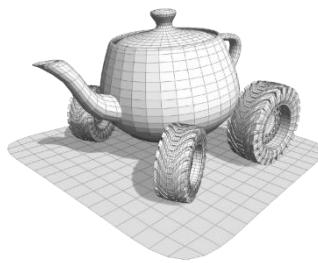
- Steps:

 1. Find the intersection between the ray from the point of view through clicked points on the viewport and the sphere
 2. Two points \mathbf{p}'_0 and \mathbf{p}'_1 on the sphere define a rotation

$$\mathbf{r} = \frac{(\mathbf{p}'_0 - \mathbf{c}) \times (\mathbf{p}'_1 - \mathbf{c})}{\|(\mathbf{p}'_0 - \mathbf{c}) \times (\mathbf{p}'_1 - \mathbf{c})\|}$$

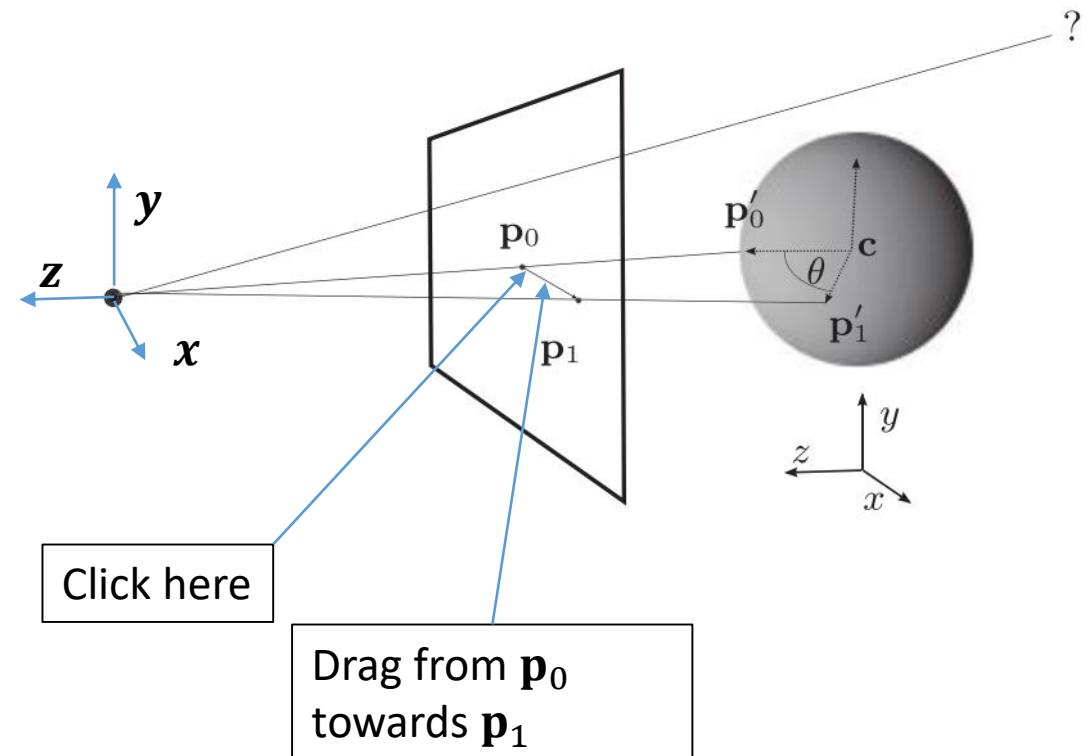
$$\theta = \arcsin \left(\frac{\|(\mathbf{p}'_0 - \mathbf{c}) \times (\mathbf{p}'_1 - \mathbf{c})\|}{\|(\mathbf{p}'_0 - \mathbf{c})\| \|(\mathbf{p}'_1 - \mathbf{c})\|} \right)$$

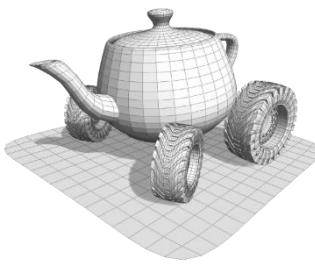




World-in-hand: the Virtual Trackball

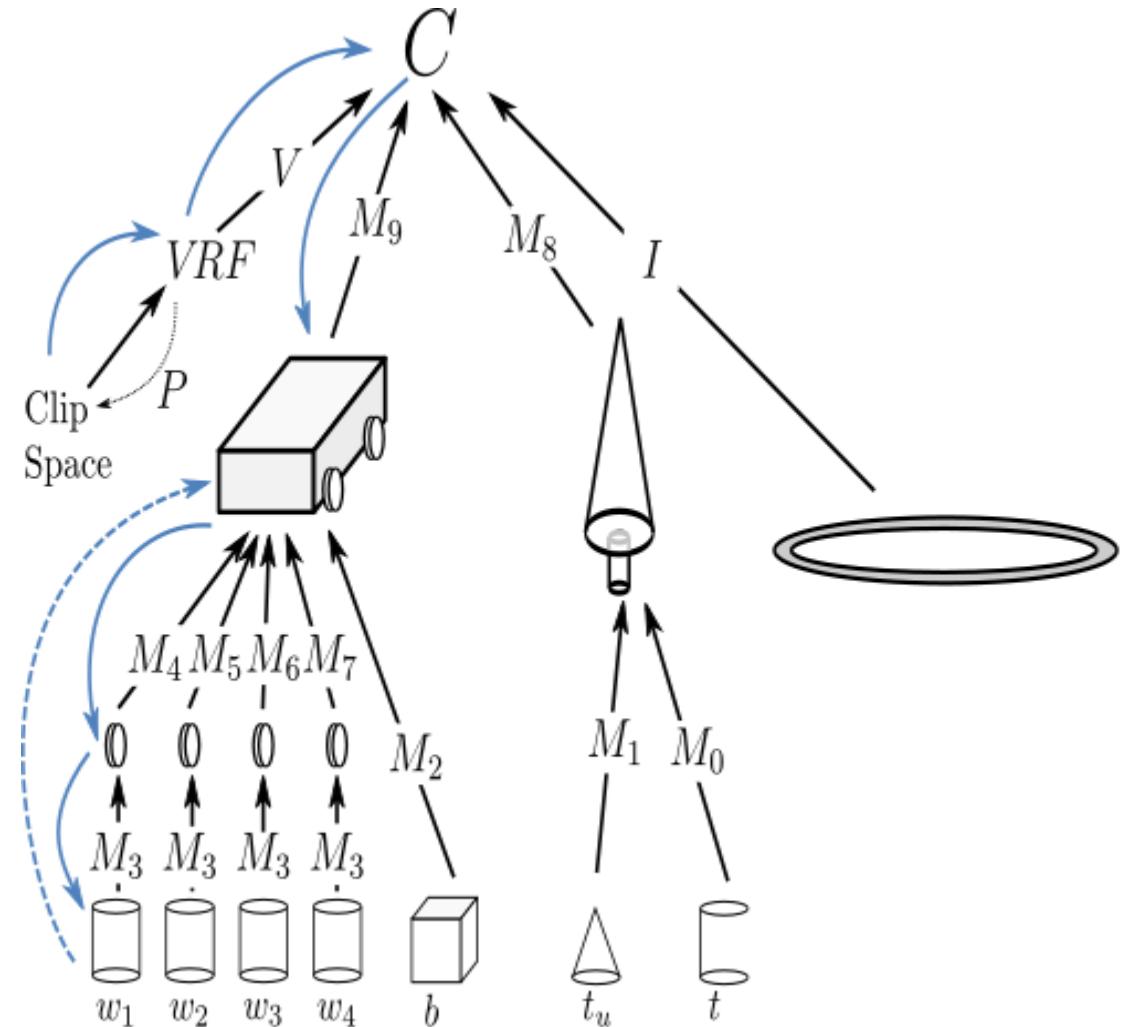
- Steps:
 1. Find the intersection between the ray from the point of view through clicked points on the viewport and the sphere
 2. Two points p'_0 and p'_1 on the sphere define a rotation
 3. Apply the rotation
 - Apply where?

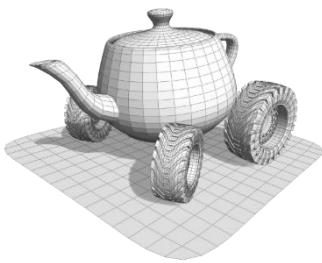




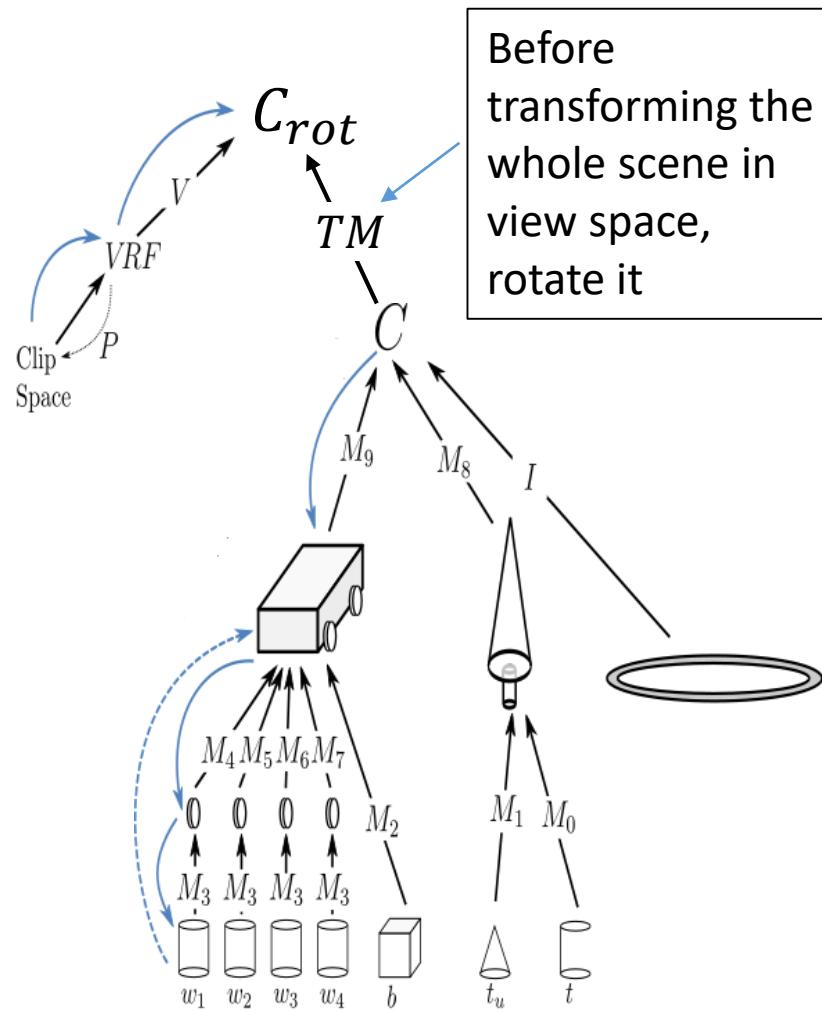
Apply virtual trackball transform

- Where in our hierarchy of transformations, should we put the trackball transformation matrix?

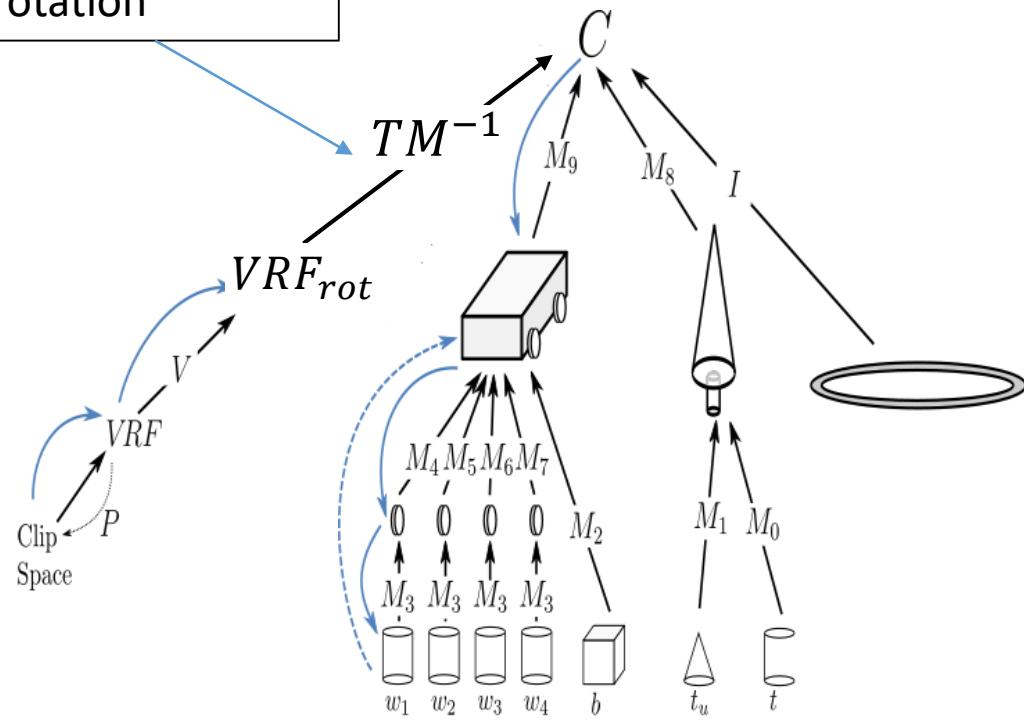


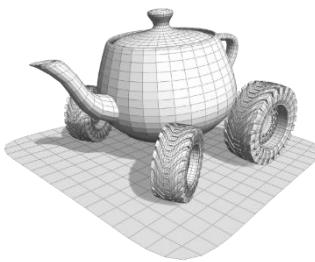


Two equivalent choices



Rotate the view reference frame by the inverse rotation

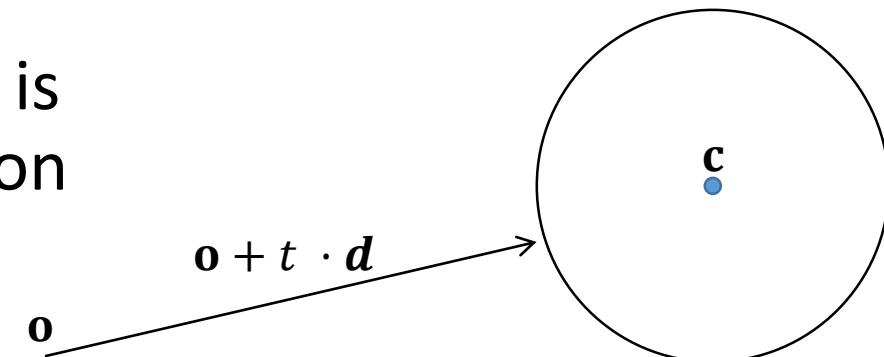




Ingredients: Ray-Sphere intersection

- A sphere centered at \mathbf{c} with radius r is described with the implicit formulation

$$S = \{\mathbf{p} \in \mathbb{R}^3 \mid (\mathbf{p} - \mathbf{c})(\mathbf{p} - \mathbf{c}) = r^2\}$$



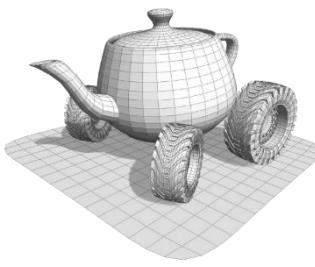
- Find if for some t , the ray $\mathbf{o} + t \cdot \mathbf{d}$ belongs to S

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c})(\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2$$

$$At^2 + Bt + C = 0$$

$$((\mathbf{o} - \mathbf{c}) + t\mathbf{d})((\mathbf{o} - \mathbf{c}) + t\mathbf{d}) = r^2$$

$$\underbrace{(\mathbf{o} - \mathbf{c})(\mathbf{o} - \mathbf{c}) - r^2}_{C} + t \underbrace{2\mathbf{d}(\mathbf{o} - \mathbf{c})}_{B} + t^2 \mathbf{d}^2 = 0 \quad \Rightarrow \quad t = \frac{-B \pm \sqrt{(B^2 - 4AC)}}{2A}$$



Ingredients: event handling

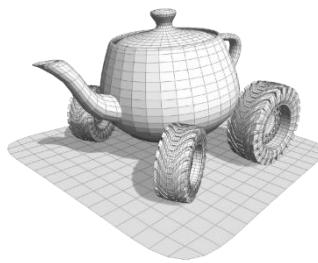
Your functions to handle the events

```
void cursor_position_callback(GLFWwindow* window, double xpos, double ypos){...}  
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods){...}
```

```
void main() {  
    /* ... */  
    if (glfwRawMouseMotionSupported())  
        glfwSetInputMode(window, GLFW_RAW_MOUSE_MOTION, GLFW_TRUE);  
  
    glfwSetCursorPosCallback(window, cursor_position_callback);  
    glfwSetMouseButtonCallback(window, mouse_button_callback);  
  
    while (...) {  
        glfwPollEvents();  
    }  
}
```

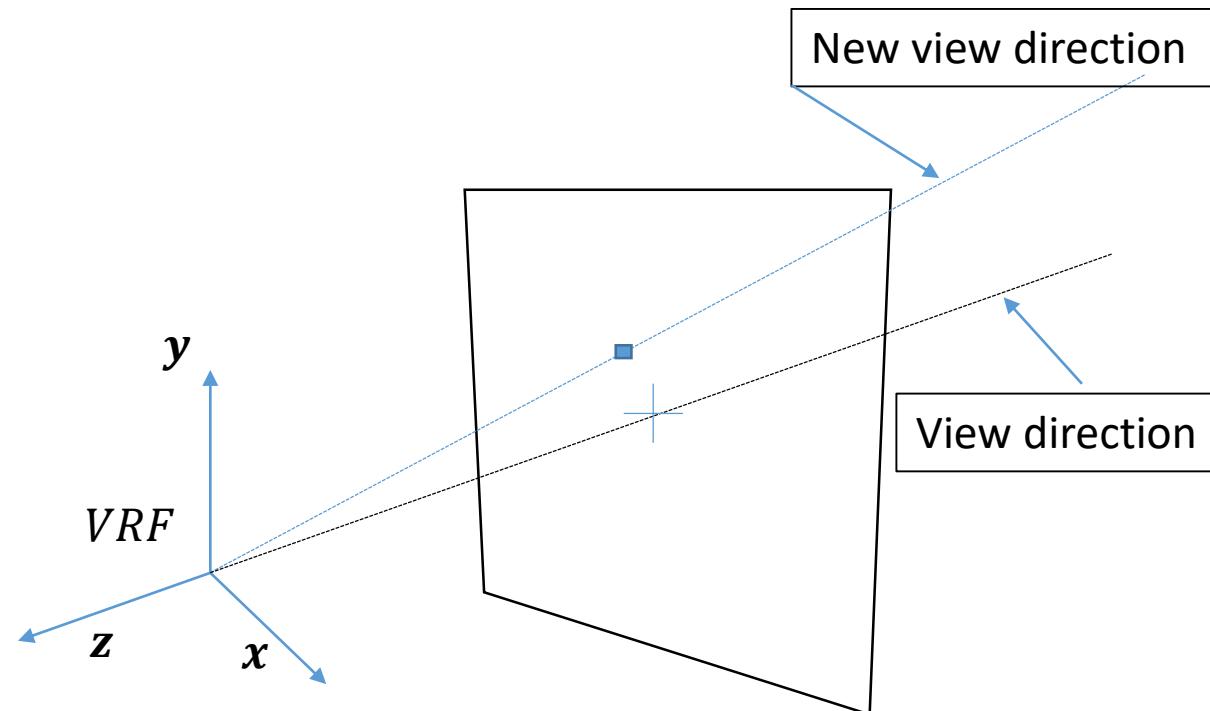
Tell GLFW which are the function handling mouse events

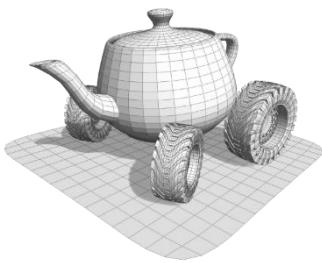
Remember to «Poll» the events



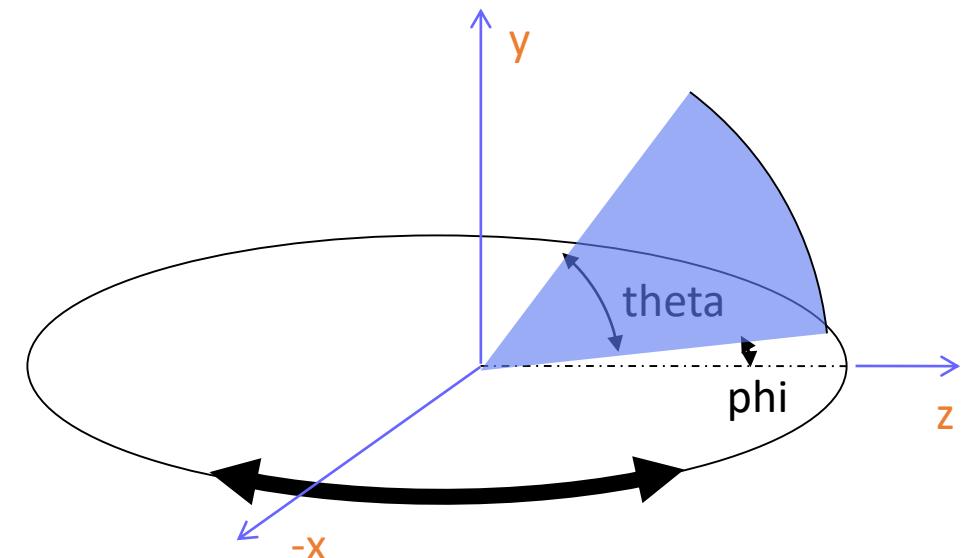
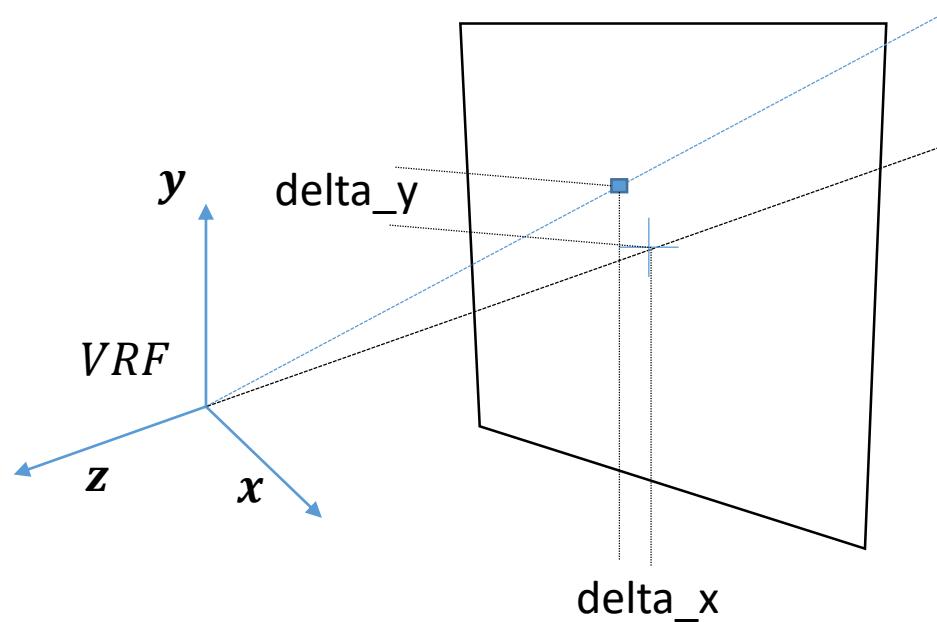
Camera-in-hand

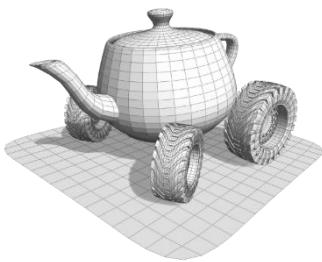
- WASD & mouse: very typical of First Person Shooter games
 - Maps keyboard keys to translation of the view reference frame (change origin, axis unchanged)
 - w: forward
 - s : backward
 - a: left
 - d: right
 - Map mouse movements (while left button is pressed) to view direction (changes axis, origin unchanged)



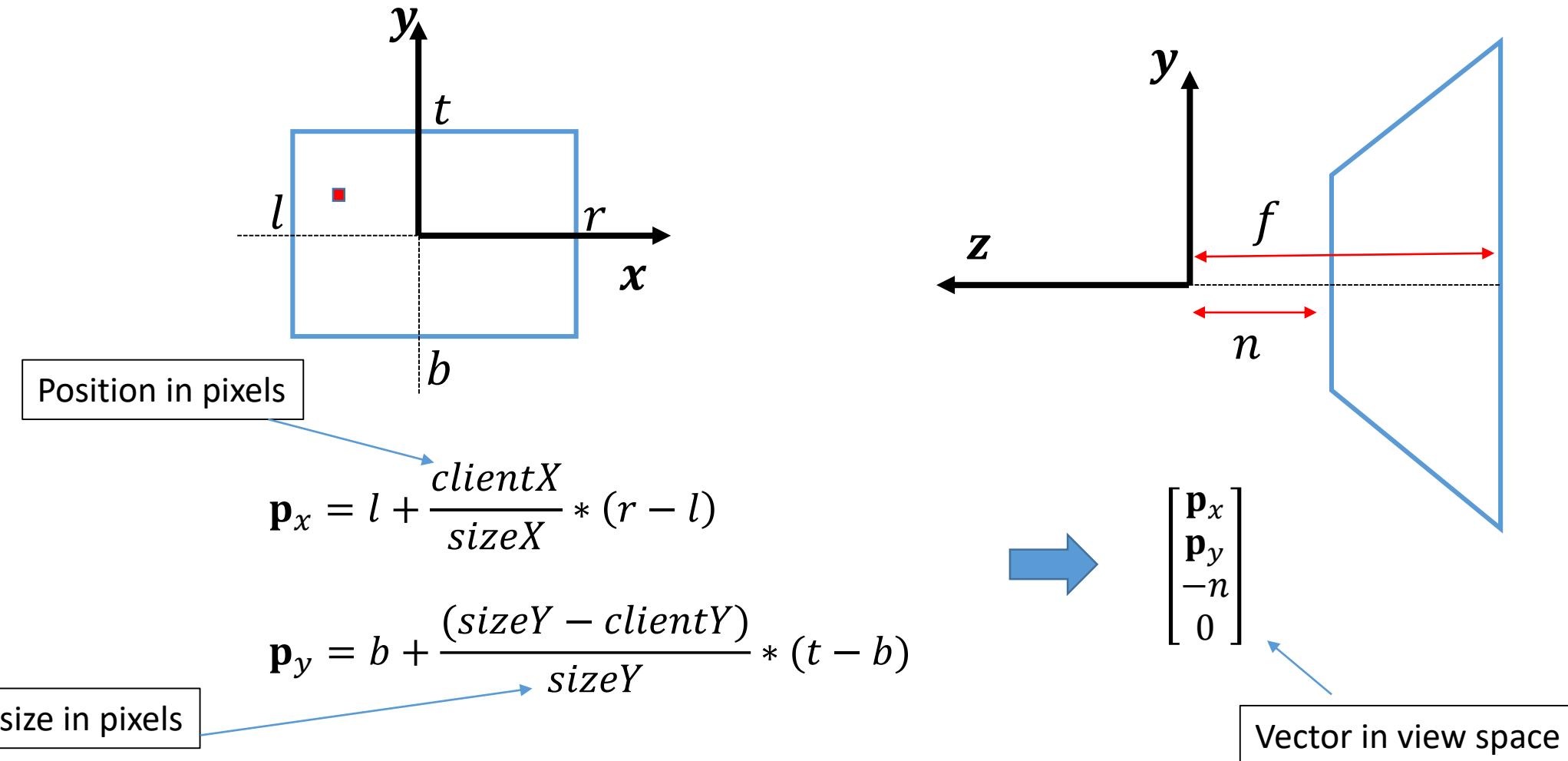


Camera-in-hand: looking at

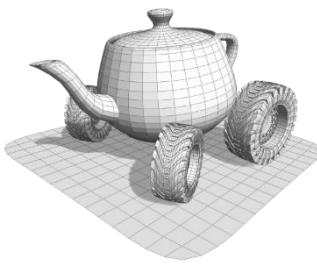




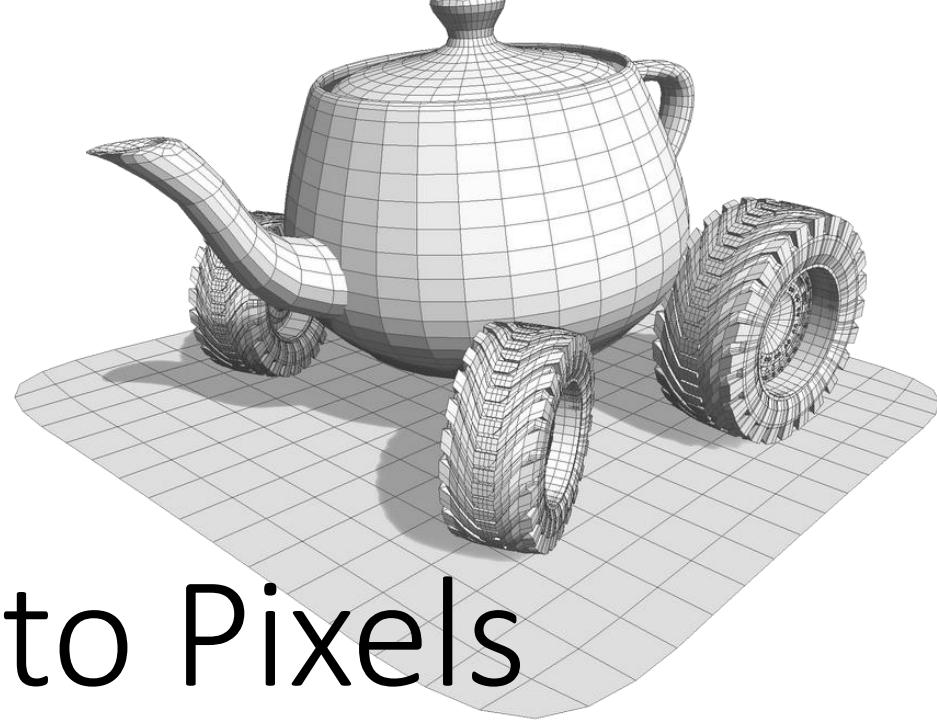
From screen coordinates to view space



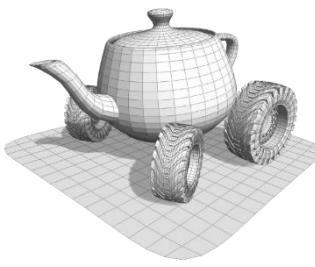
Exercise



1. Implement a virtual trackball in your version of “code_6_trackball”
2. Implement the camera-in-hand movement

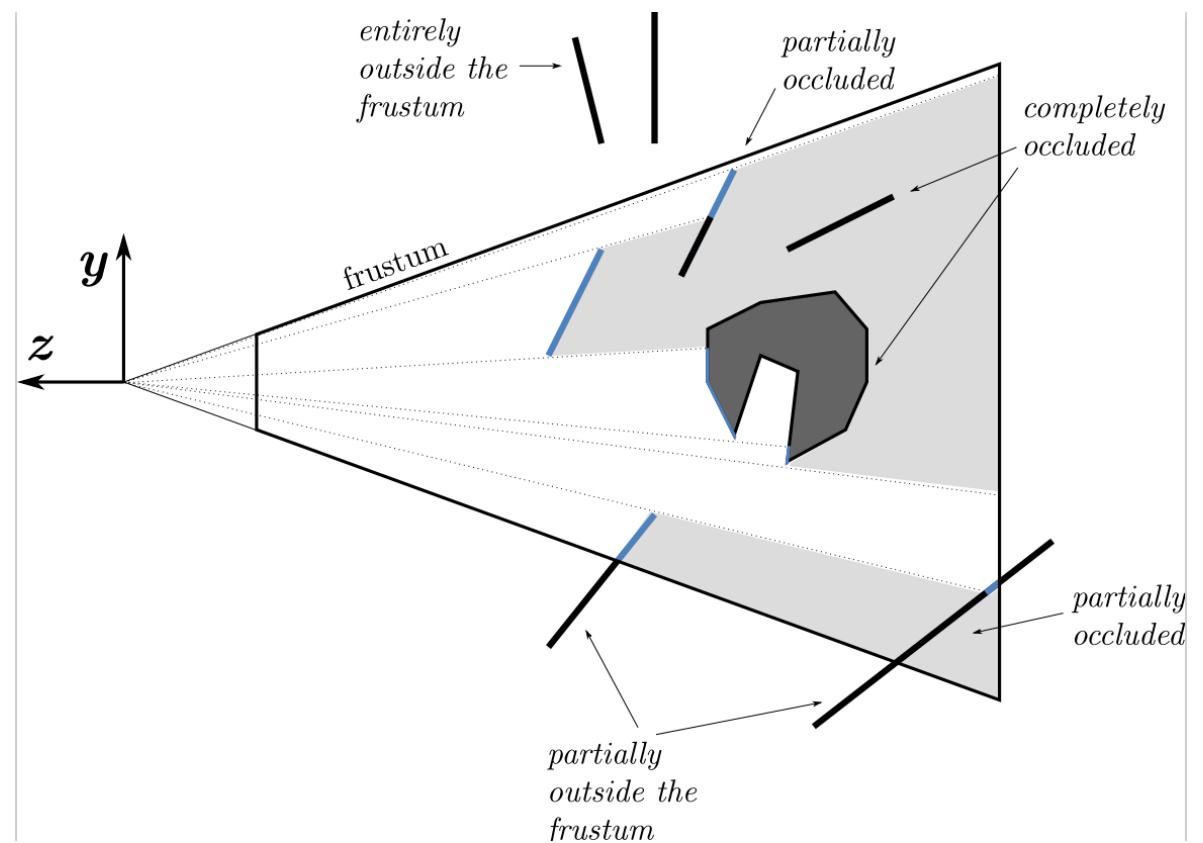


Turning Vertices into Pixels

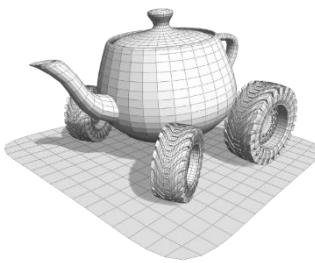


Hidden Surface Removal

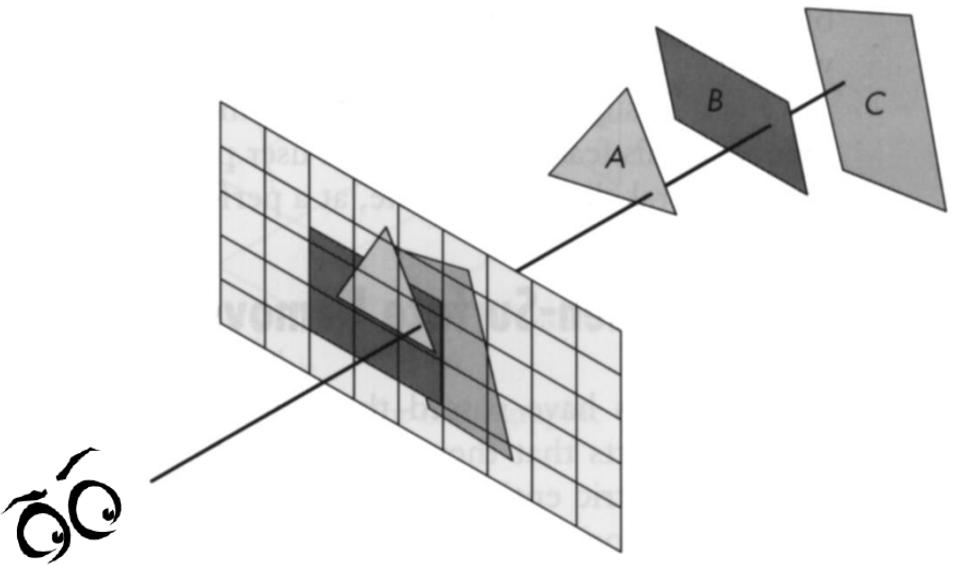
- **Hidden Surface Removal (HRS)** refers to the problem and solutions to avoid drawing things that are not currently visible because:
 - Partially or entirely occluded (occlusion culling)
 - Entirely outside the frustum (frustum culling)
 - Partially outside the frustum (clipping)

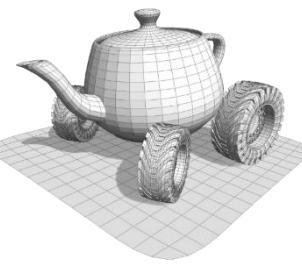


Occlusion Culling



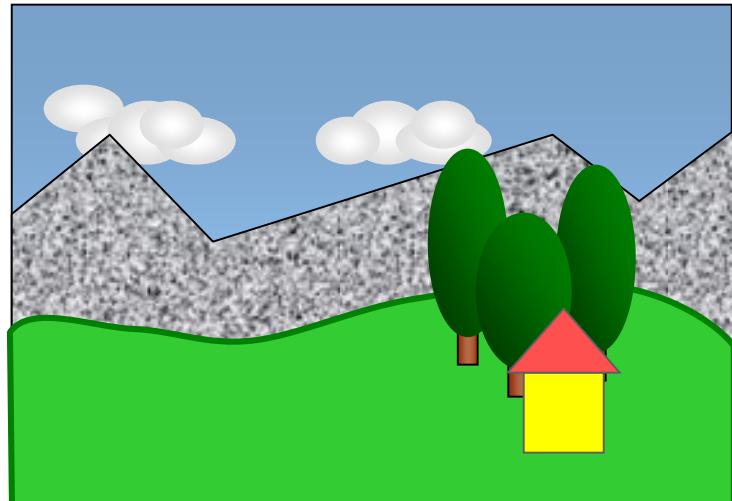
- Let us assume that the objects in the scene are opaque
- For each pixel, we need to find the closest primitive along the **view ray** (that is, the ray leaving the eye and passing through the pixel's center)
- Only that primitive will contribute to the color of the pixel

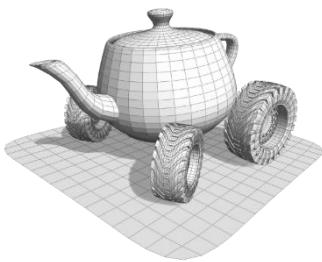




Depth Sort or The Painter's Algorithm

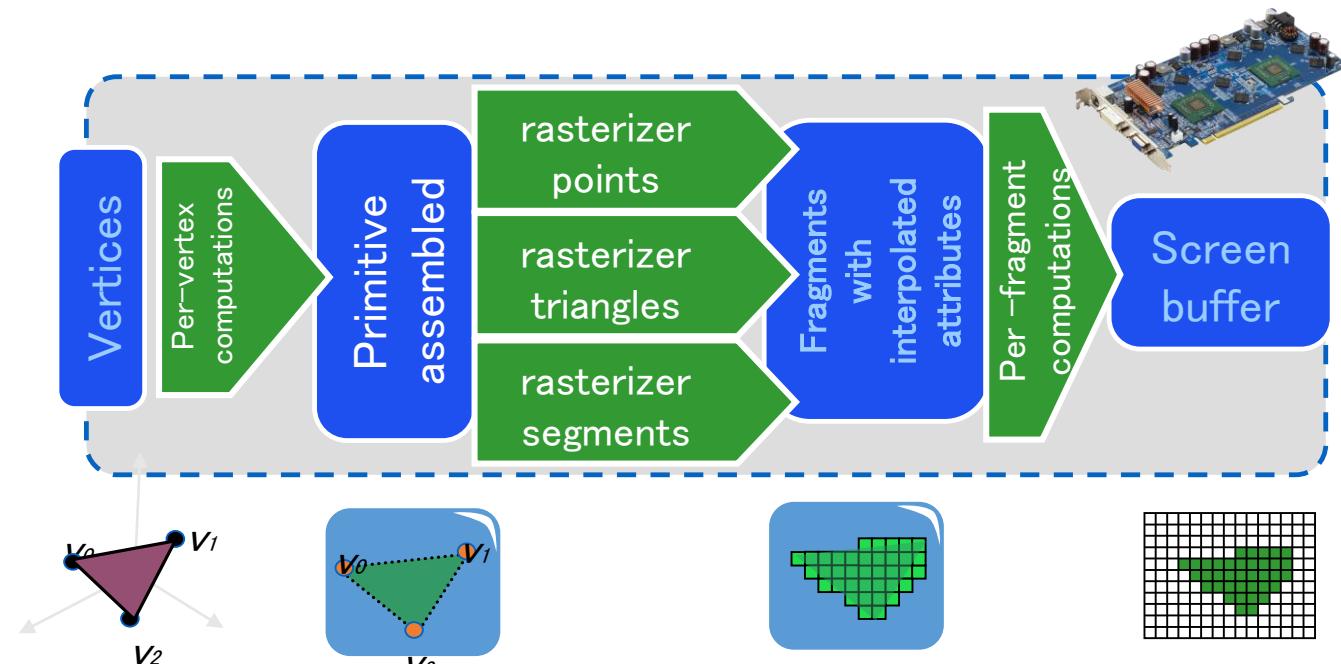
- Draw things from the farthest to closest
 - Every primitive will write into the frame buffer
 - The occluded one will be overdrawn by closest primitives
- We only need a way to sort them properly “back-to-front”
 - Back-to-front: from the farthest to the closest
 - Front-to-back : from the closest to the farthest

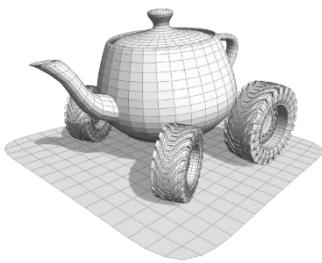




Where in the pipeline?

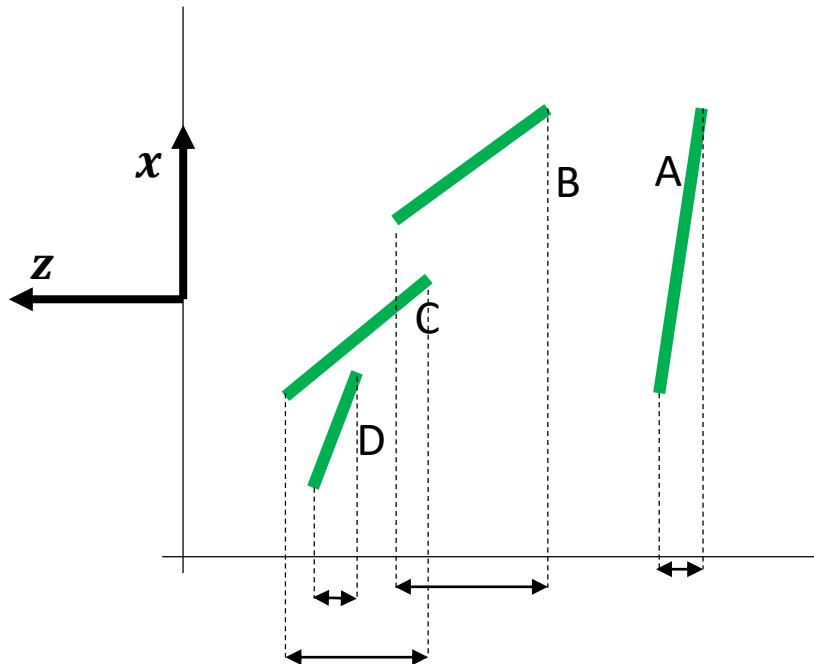
- Unfortunately it has to be done before sending primitives down the pipeline, which means in CPU
- However, *can* it be done at all?

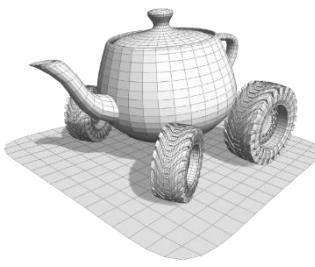




Depth Sort

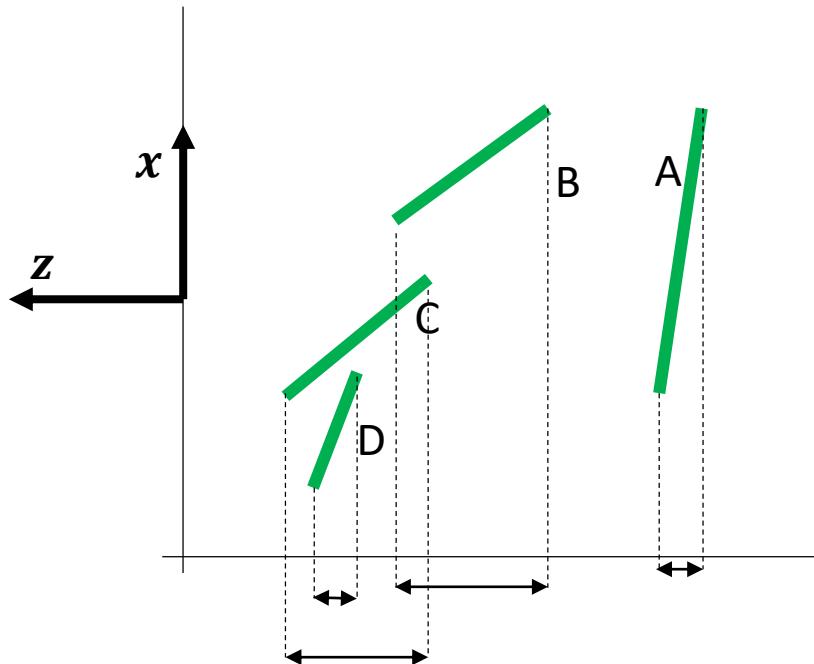
- **Depth Sort:** sorting primitives along their depth ($=-z$ axis)
- In the example:
 - A is the first to draw
 - B is the second
 - Which one between C and D?

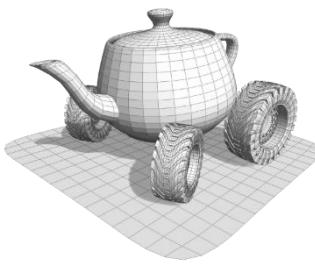




Depth Sort

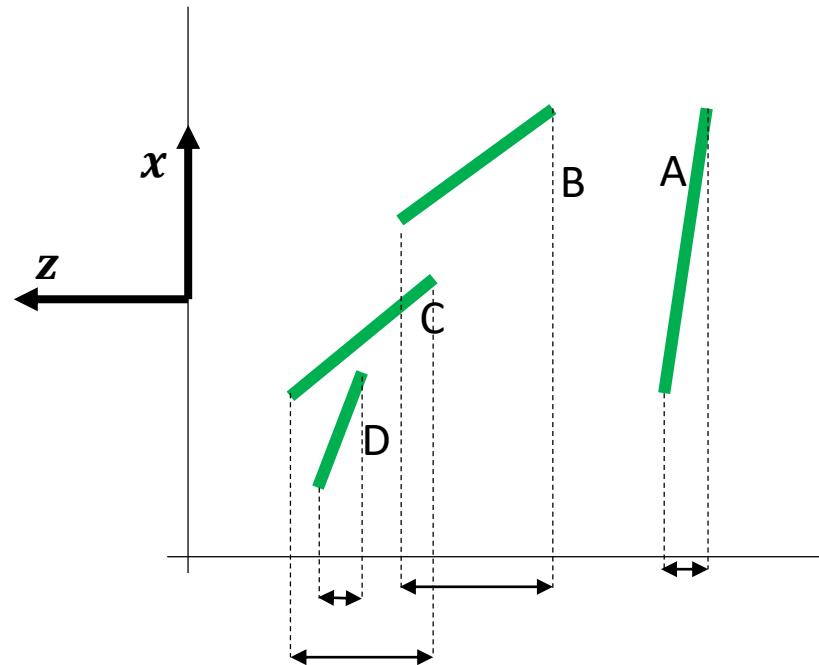
- The core of Depth Sort is to find an efficient test to decide the order of two primitives
 1. Do their projection along z overlap? If not, the ordering is trivial, else..
 2. Do their projections on the x axis overlap? If not, no need to sort..
 3. Do their projections on the y axis overlap? If not, no need to sort..
 4. Hang on....

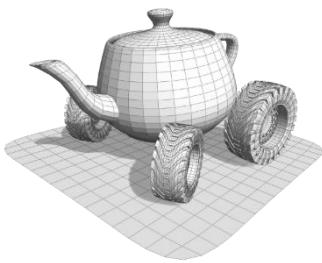




Depth Sort: the separating plane

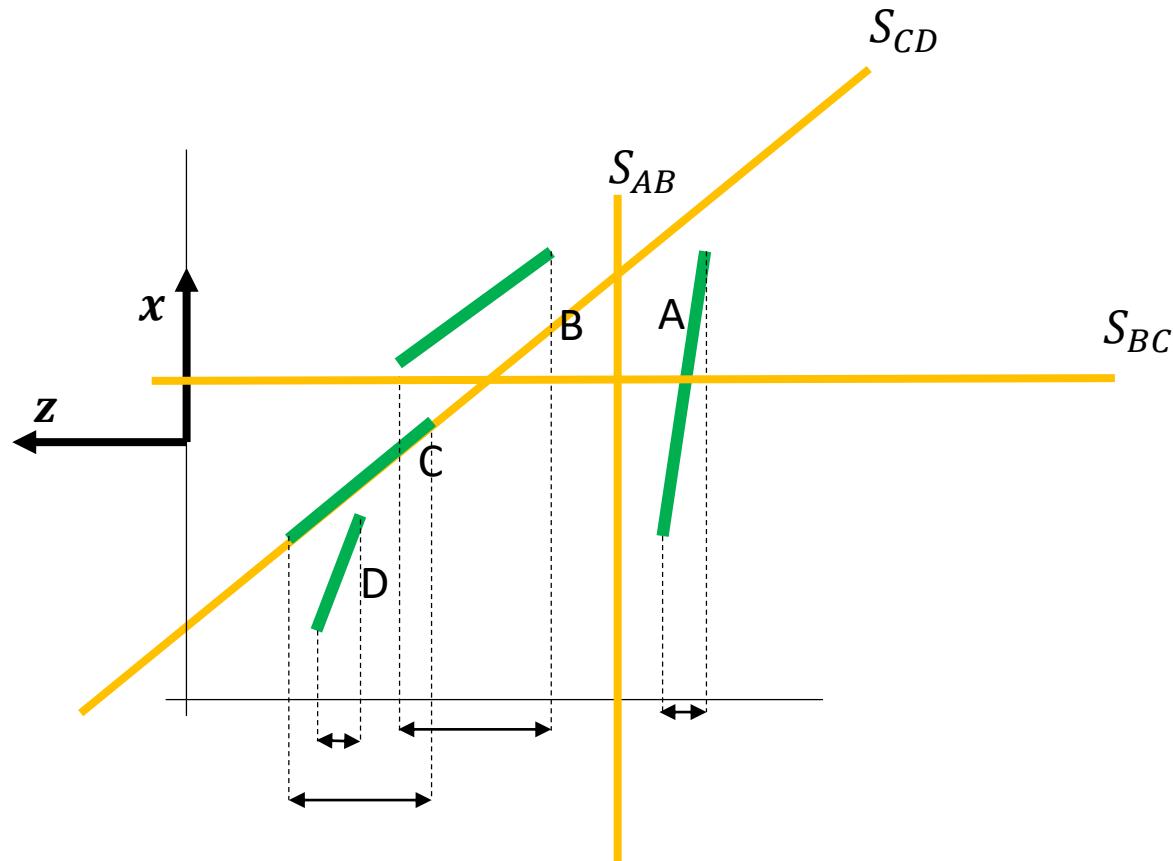
- A **separating plane** between two geometric entities is a plane that separates them in its two halfspaces
- If we find a separating plane such that A and the point of view are in one side and B in the other, then B can be drawn first
- If there exists a SP between two polygons, than at least one of the planes containing one of the polygons is a SP

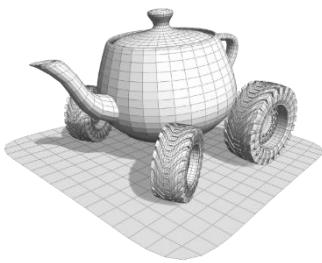




Depth Sort

- The core of Depth Sort is to find an efficient test to decide the order of two primitives P and Q
 - 1. Do their projection along z overlap? If not, the ordering is trivial, else..
 - 2. Do their projections on the x axis overlap? If not, no need to sort..
 - 3. Do their projections on the y axis overlap? If not, no need to sort..
 - 4. Is the viewpoint and P on the same side of the plane passing through P? P is in front
 - 5. The other way around (Q instead of P)

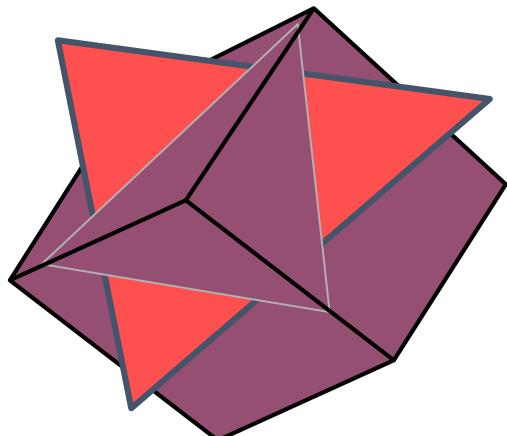




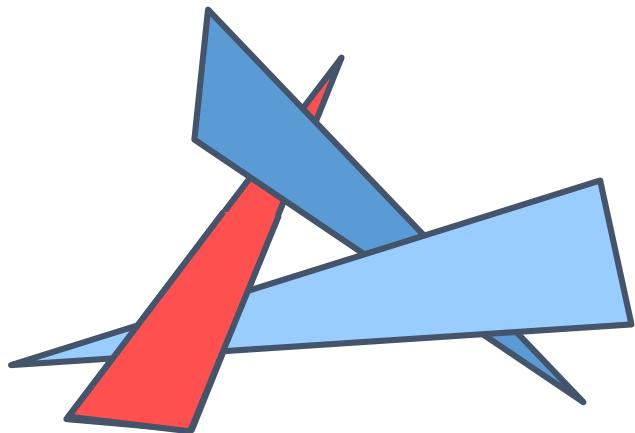
Depth Sort: problems

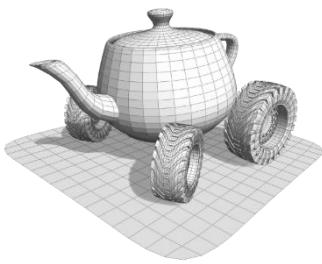
- A correct order may not exist
- In this case the primitives need be split (clipped) and draw in pieces

Intersecting primitives



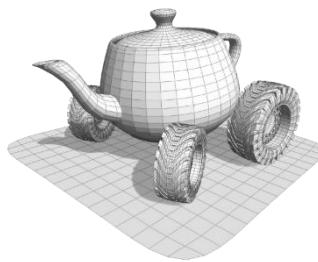
Cycles in the ordering





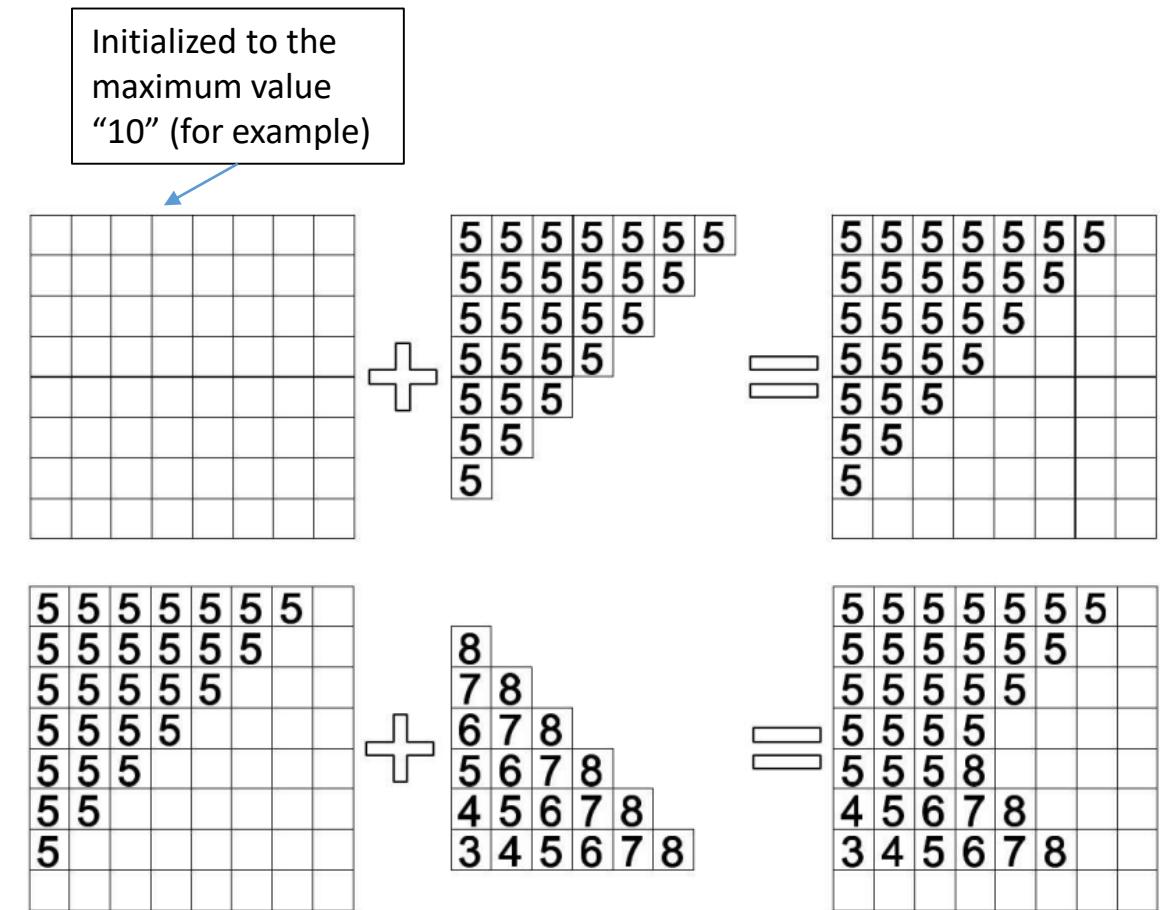
Depth Sort: recap

- It requires sorting the primitive in CPU but..
 - The coordinates of the vertices in *view space* are only known after their transformation (which we like to do it in GPU)
- It's not always possible
- The order is view dependent, any data structure should be updated/rebuilt while the point of view changes

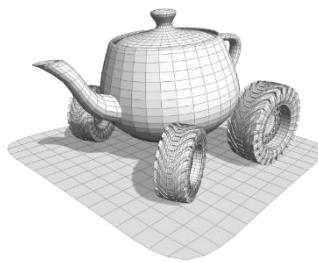


Rasterization acceleration: Z-Buffer

- The rasterization algorithm computes, for each pixel, the *distance* of the corresponding surface from the point of view
- Such distance is written into a memory buffer the same size of the screen, called **Depth (or Z) Buffer**
- Unless the Z-Buffer already contains a smaller value for that pixel



Z-buffer: esample



1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

+

.5	.5	.5	.5	.5	.5	.5	.5	
.5	.5	.5	.5	.5	.5	.5		
.5	.5	.5	.5	.5	.5			
.5	.5	.5	.5	.5				
.5	.5	.5	.5					
.5	.5	.5						
.5	.5							
.5								

-

.5	.5	.5	.5	.5	.5	.5	.5	1.0
.5	.5	.5	.5	.5	.5	.5		1.0
.5	.5	.5	.5	.5	.5			1.0
.5	.5	.5	.5	.5				1.0
.5	.5	.5	.5					1.0
.5	.5	.5						1.0
.5	.5							1.0
.5								1.0

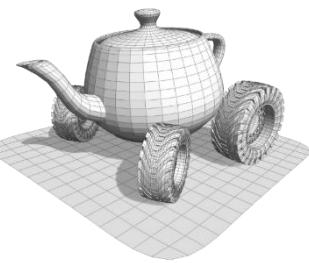
.5	.5	.5	.5	.5	.5	.5	.5	1.0
.5	.5	.5	.5	.5	.5	.5	1.0	1.0
.5	.5	.5	.5	.5	.5	1.0	1.0	1.0
.5	.5	.5	.5	.5	1.0	1.0	1.0	1.0
.5	.5	.5	.5	1.0	1.0	1.0	1.0	1.0
.5	.5	.5	1.0	1.0	1.0	1.0	1.0	1.0
.5	.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

+

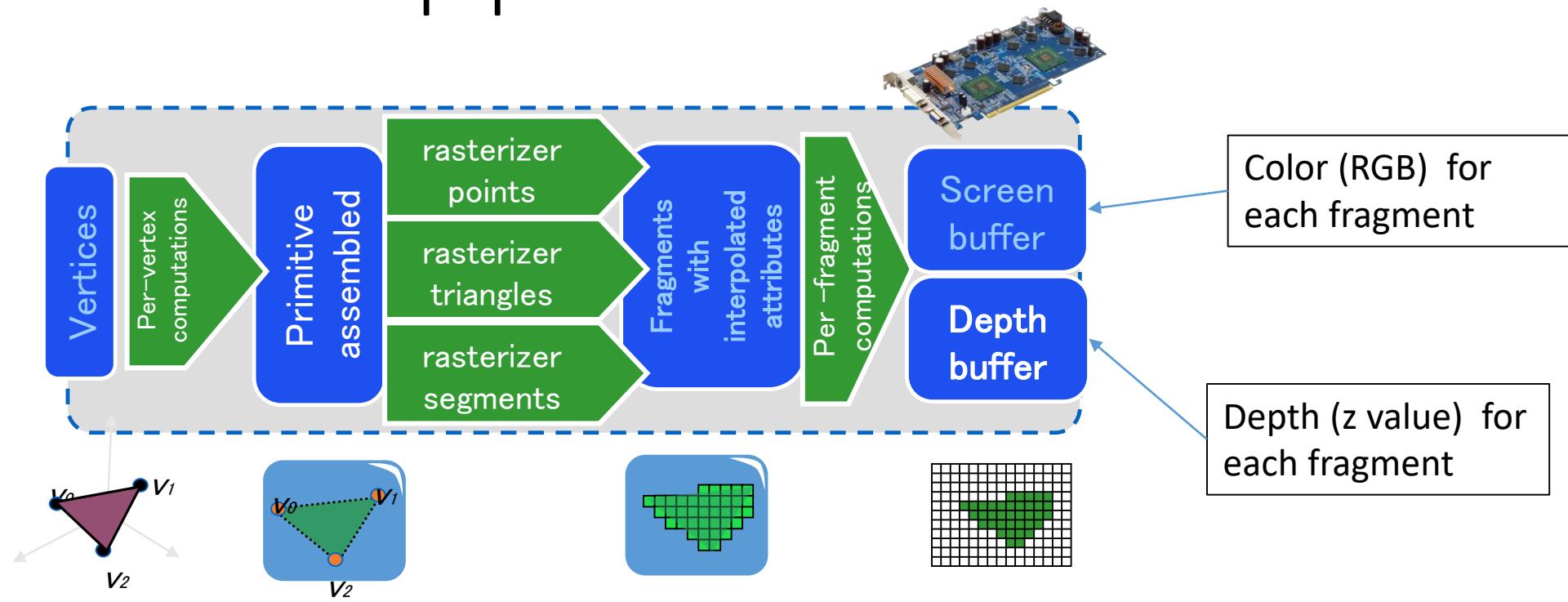
.7								
.6	.7							
.5	.6	.7						
.4	.5	.6	.7					
.3	.4	.5	.6	.7				
.2	.3	.4	.5	.6	.7			

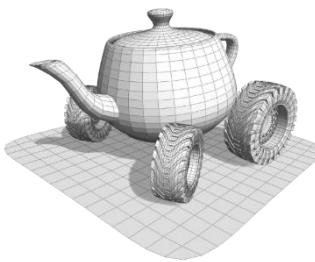
-

.5	.5	.5	.5	.5	.5	.5	.5	1.0
.5	.5	.5	.5	.5	.5	.5		1.0
.5	.5	.5	.5	.5	.5			1.0
.5	.5	.5	.5	.5				1.0
.4	.5	.5	.7	1.0	1.0	1.0	1.0	1.0
.3	.4	.5	.6	.7				1.0
.2	.3	.4	.5	.6	.7			1.0

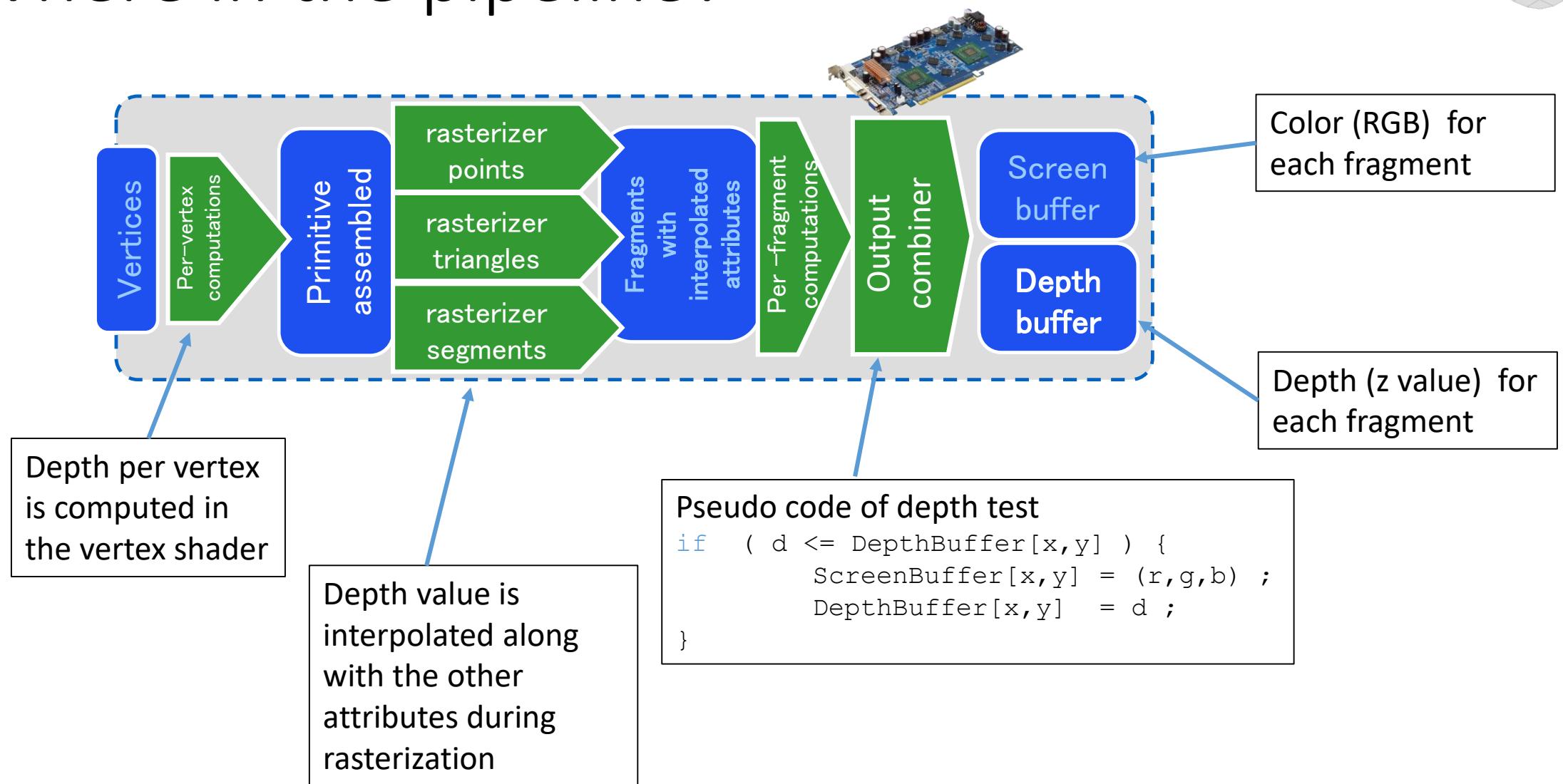


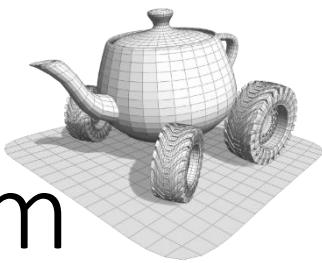
Where in the pipeline?



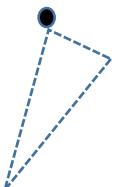
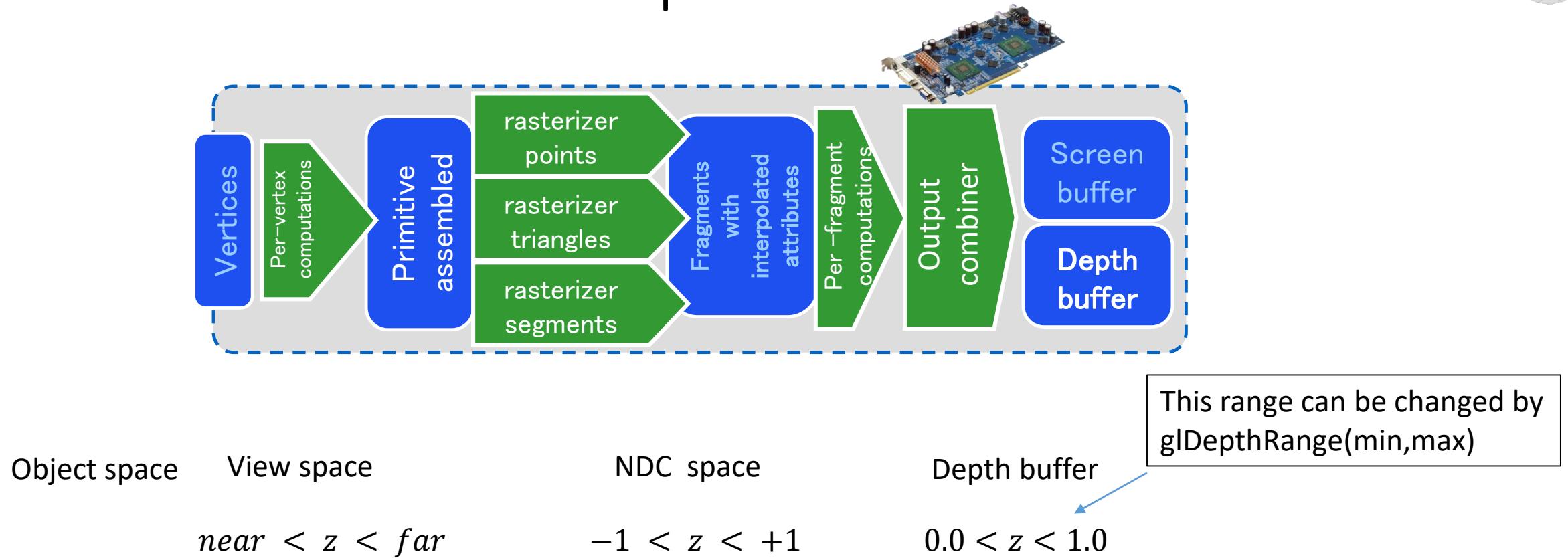


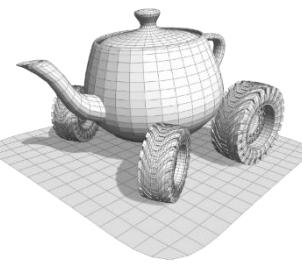
Where in the pipeline?





Z Coordinates of a point in the View Frustum

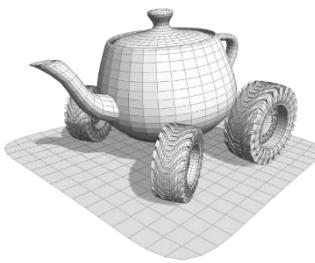




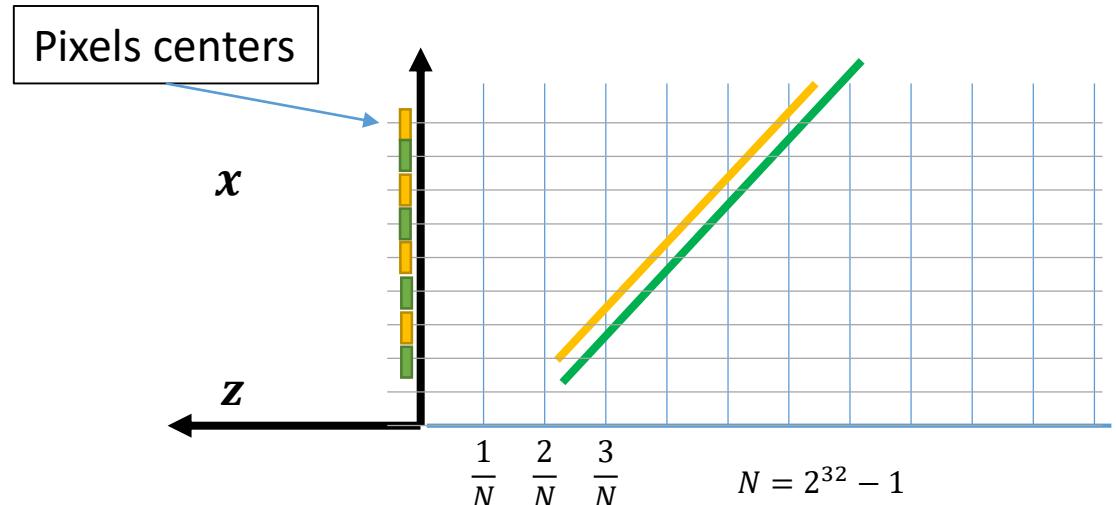
Depth buffer precision

- Z values are stored in *fixed precision*
 - Used to be 8 bits, now 24 or 32 bits in modern architectures
- e.g. with 32 bits, we store the value z between 0,.., $2^{32}-1$ to represent $\frac{z}{2^{32}-1}$ between 0.0 and 1.0
 - Therefore the represented value are equispaced between 0.0 and 1.0
- What happens if 32 bits are not enough to discriminates among all the depth between the near and far plane (in view space)?

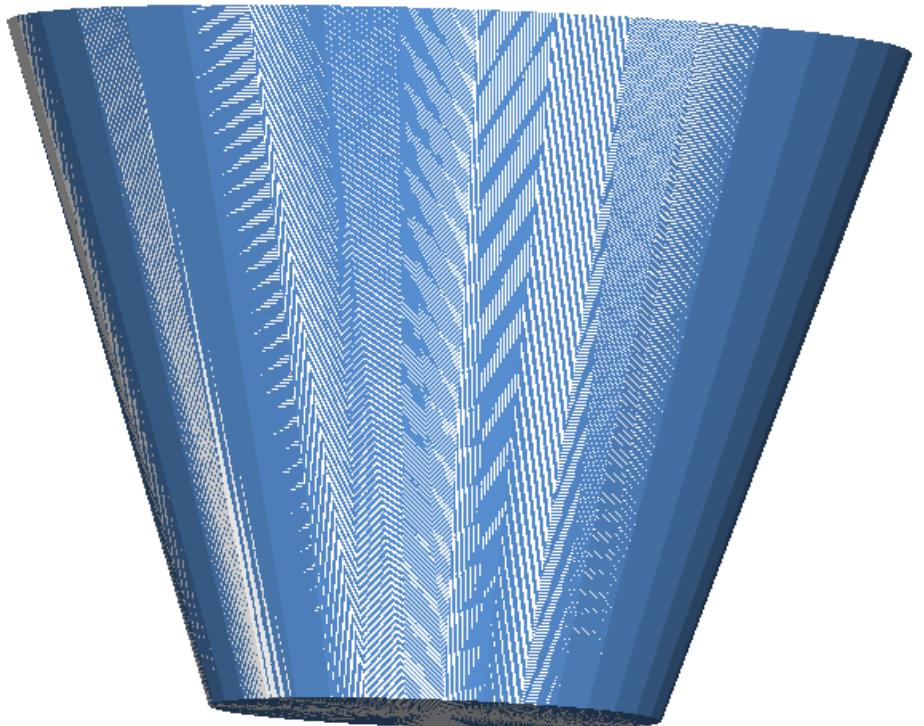
Z-fighting

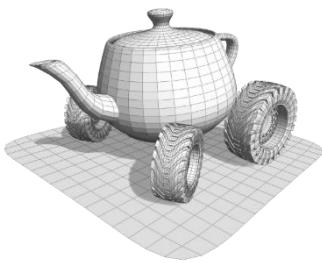


- **Z-fighting** happens when two polygons produce fragments with depth values very close to each other



One white and one blue trunks of cone
almost in the same exact position



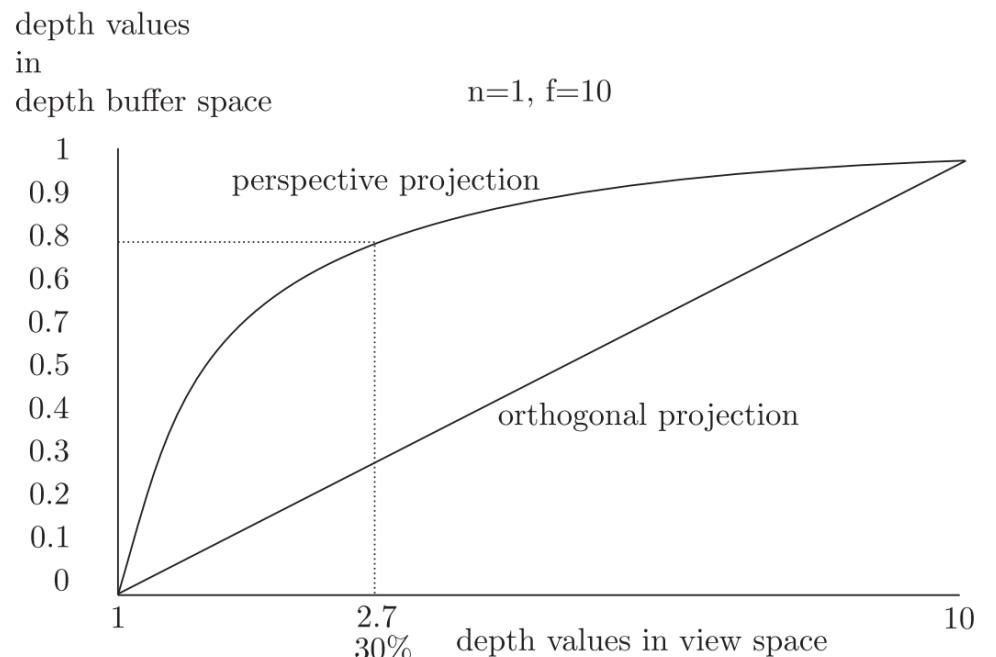


Precision on z

- With *perspective projection* the depth is computed as:

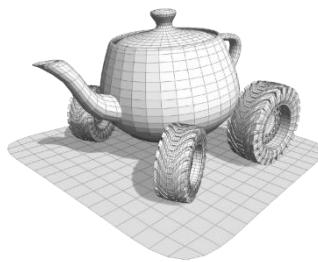
$$f_p(z) = \frac{1}{2} \left(\frac{f+n}{f-n} - \frac{1}{z} \frac{2fn}{f-n} + 1 \right) = \left(\frac{1}{2} \frac{f+n}{f-n} + \frac{1}{2} \right) - \frac{fn}{z(f-n)}$$

- so the depth value goes with the inverse of z
- Note: the 80% of the depth values are taken to represent the 30% of the depth
 - That is, less precision towards the “far” value

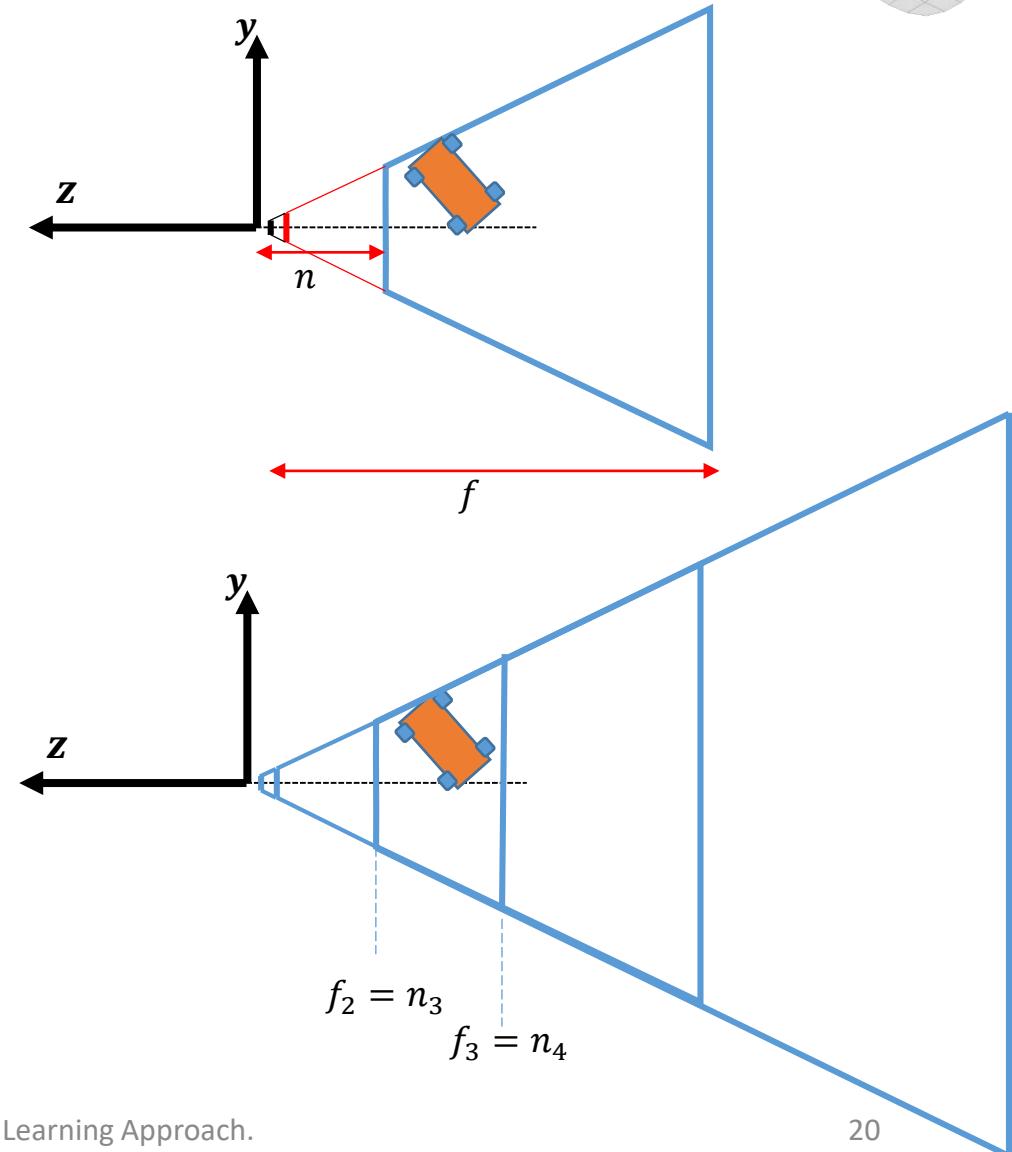


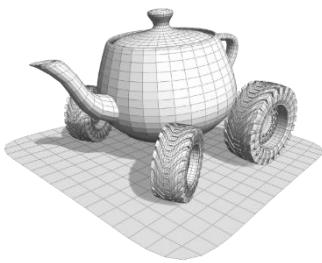
$$f_p(z) = \frac{10}{9} - \frac{1}{z} \frac{10}{9}$$

Fighting the z-fighting



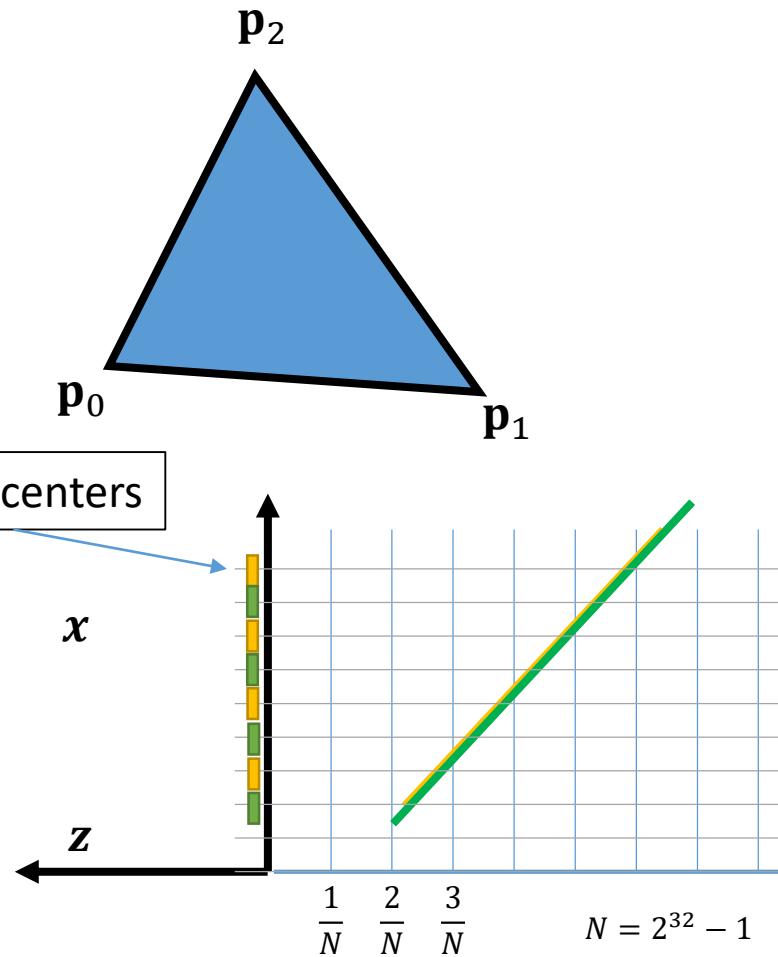
- Set the *near* and *far* value so to fit with the scene
 - Especially not too small *near* values
- How can we render from 1 centimeter to the stars?
 - There are other technique (later on...) but should you need it just render in clusters separated along the z axis and use different near-far values
 - Be careful that near/far planes need to be separating planes

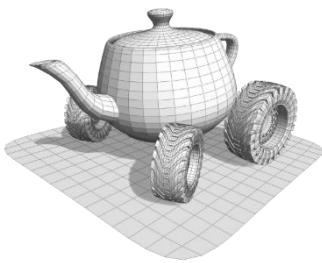




Polygon offset (1/2)

- Suppose you want triangles and their borders as `GL_LINES`
- Alert! The 3 lines for a triangle have exactly the same coordinates as the triangle: z-fighting worst case scenario
- OpenGL implements the polygon offset, an hardwired function to give a little bit of offset to the primitives you want to “win” the fight in close calls





Polygon offset (2/2)

`glPolygonOffset(GLfloat factor, GLfloat units);`

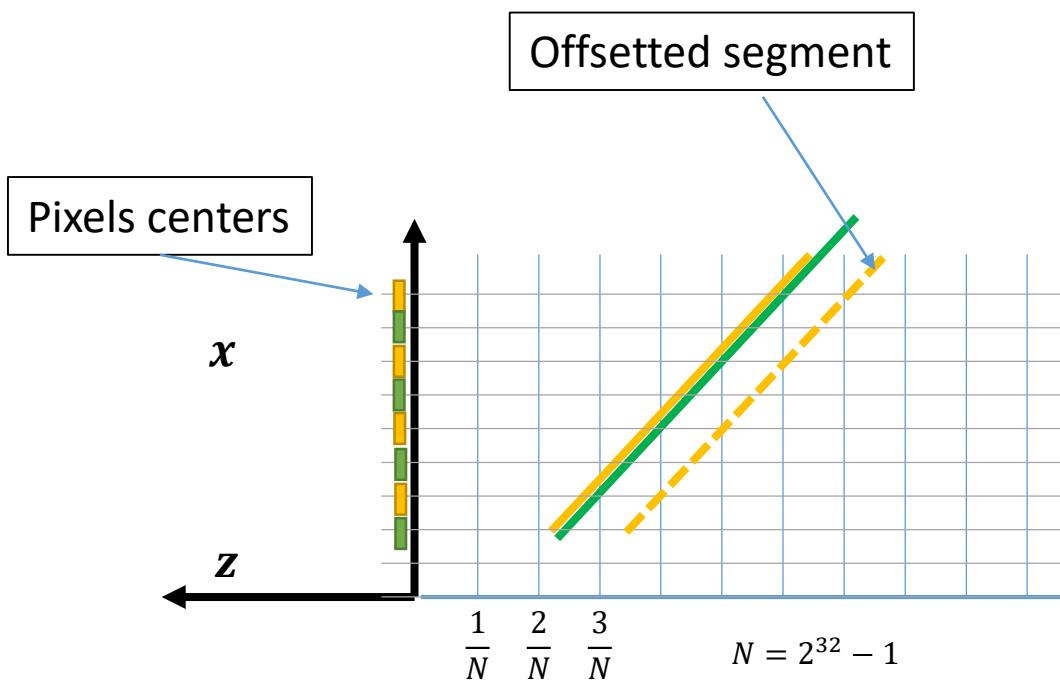
The interpolated depth value for the fragment is changed to:

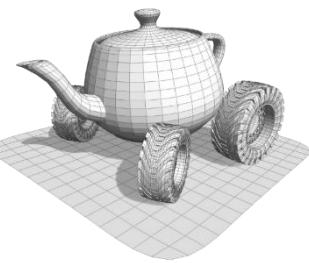
$$factor \cdot \frac{dz}{dxdy} + units \cdot r$$

Slope of the primitive at
the fragment position

Minimum value to tell
two depth values apart

`glPolygonOffset(1.0,1.0);` works well in most situations

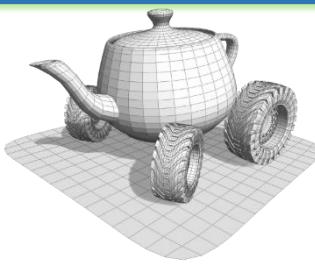




Depth Buffer as a by product

- The Depth Buffer is only used to produce the correct RGB image on the screen buffer
- However, it can be read back and many algorithms use for a number of reasons (shadows, out of focus, deferred shading...later on)





Using the Depth Buffering with OpenGL

- The Depth Test is carried out by the output combiner.
- It is hardwired, nothing to program, just a few options to give

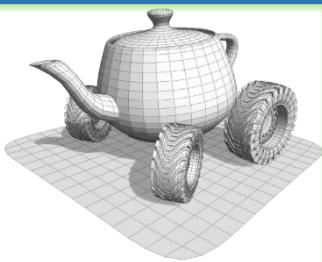
```
glEnable(GL_DEPTH_TEST);

/* set the range to which the clip
coordinates -1 and 1 will be mapped
to (default 0.0 ,1.0 )
*/
glDepthRange(minR,maxR);

/* set the value of all the depth
buffer to maxR*/
glClear(gl_DEPTH_BUFFER_BIT);

// render...

glDisable(GL_DEPTH_TEST);
```

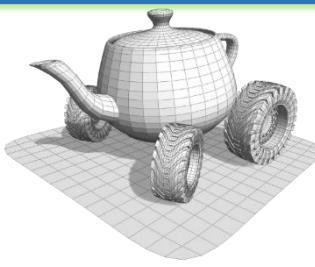


Using the Depth Buffering with WebGL

- The Depth Test is carried out by the output combiner.
- It is hardwired, nothing to program, just a few options to give
- E.g. we can specify what test do we want the output combiner to do

```
glDepthFunc(GL_LESS );  
GL_EQUAL  
GL_GREATER  
GL_LEQUAL  
GL_NOTEQUAL  
GL_GEQUAL  
GL_ALWAYS
```

Say when the tested value must be written in place of the one already present

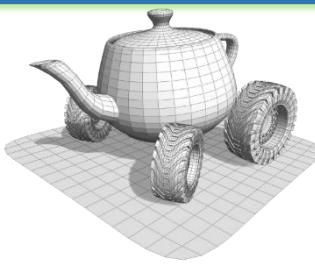


Writing the depth value

- The fragment shader receives the interpolated depth value
- We can change it output a new value for the depth
- If we do not do anything the default is to pass the value to the combiner as is
 - Which 99% of the times is what you want

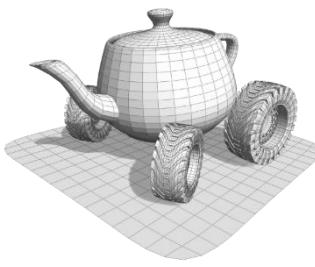
```
//fragment shader
out vec4 color;
void main(){
    // gl.fragCoord is an input value
    float depth = gl_FragCoord.z;
    // do something

    // output values
    color = vec4(...);
    gl_FragDepth = ...; // the output value
}
```



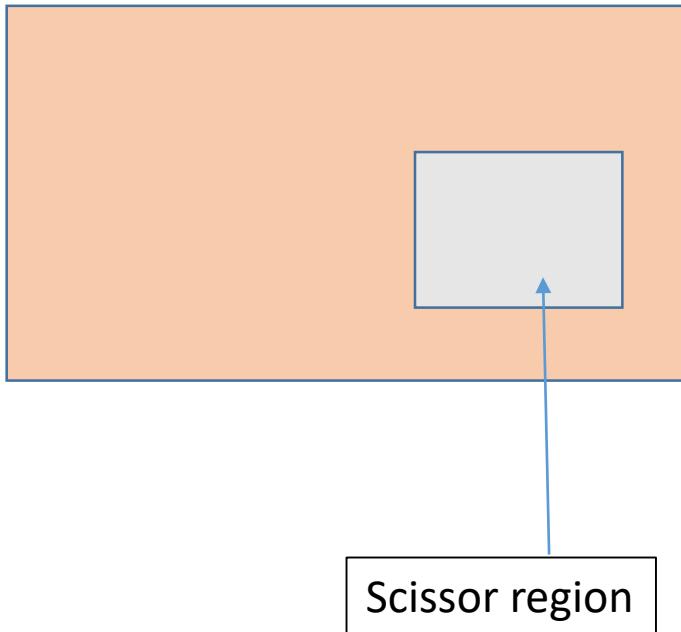
Optimizations

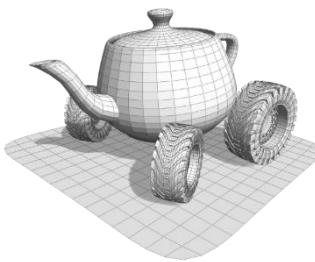
- Rule of thumb: discard what is to be discarded as early as possible
- If the fragment shader **does not** write the depth value, the fragment can be tested and discarded even *before* executing the fragment
 - So, writing the depth value in the fragment shader may significantly hinder the performances
- Changing the current depth value is more expensive than keeping it, so rendering front-to-back (when possible) is preferable
 - You won't appreciate it for small scenes...



Other tests: Scissor Test

- **Scissor test:** discard fragments outside a specified rectangle window (**the scissor region**) in screen space
 - That is: drawing in the rest of the screen is disabled
 - Useful to render GUI elements and leave the rest as it is
- E.g. `glClear` will only affect the scissor region





Other tests: Stencil Test

- The more advanced version of the scissor test
- Instead of a rectangular region, we can define a mask, called the **Stencil buffer**, that is, the exact set of pixels which will not be affected by any drawing command (or more complex things)

a mask for the inside of a car
as seen from inside (the non
white is the mask)

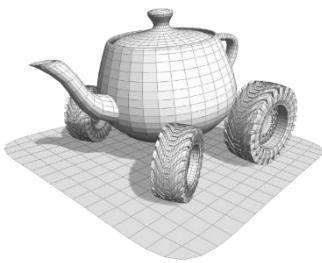


Rendering only affects
the rest of the screen,
the inside never needs to
be redrawn



Result



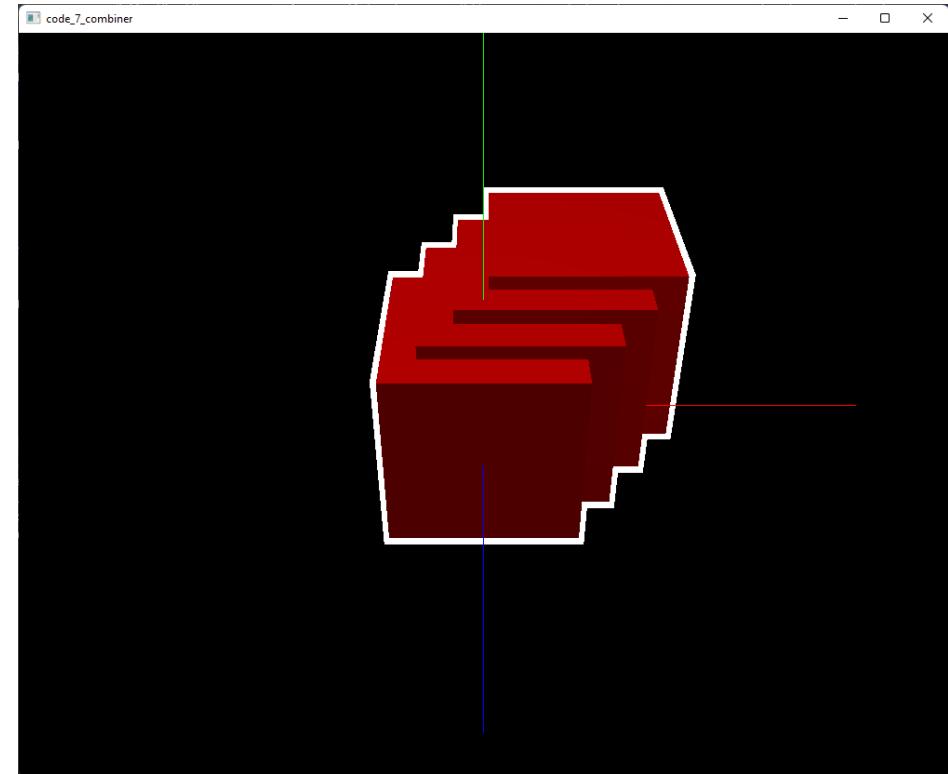


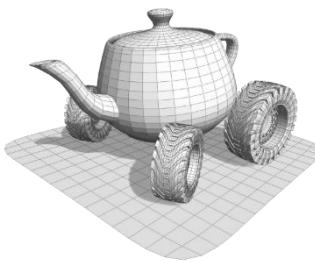
Stencil Test: uses

Drawing portals in a scene

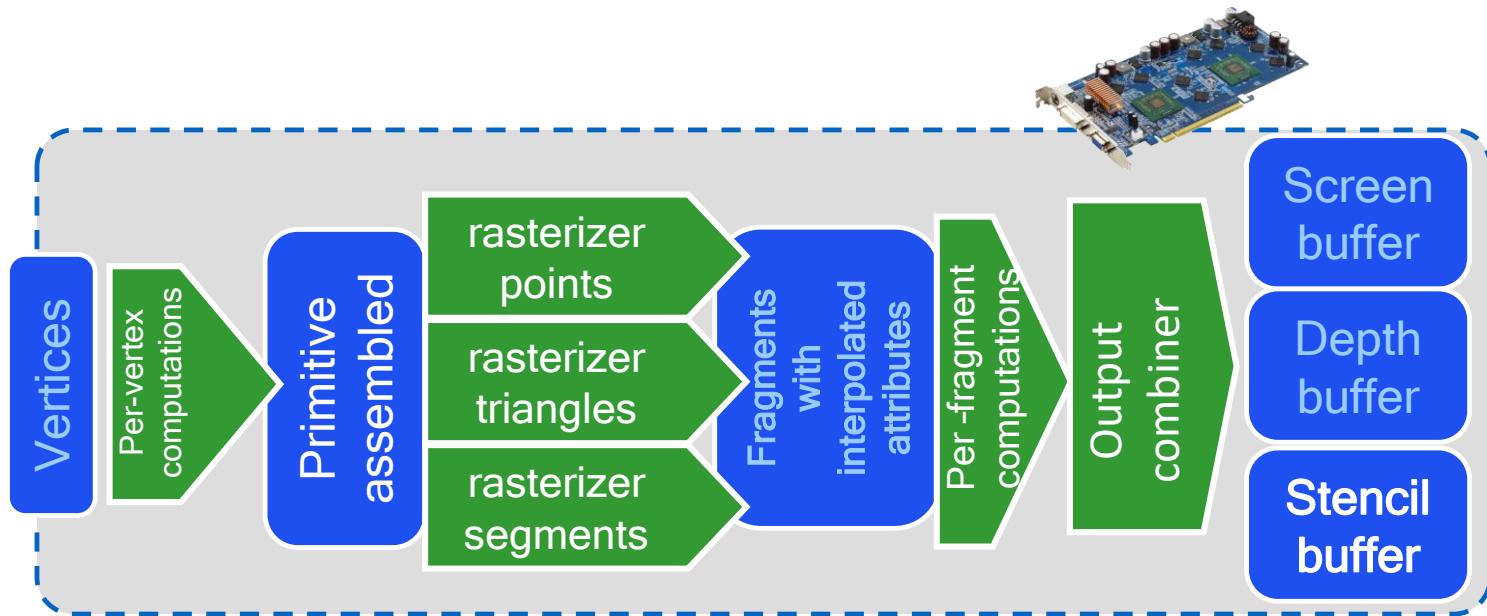


Emphasizing contours of objects (for example in a “selection mode”)

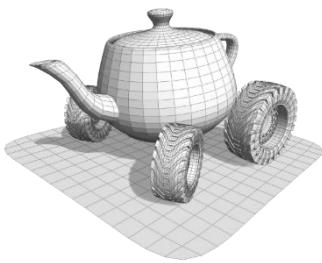




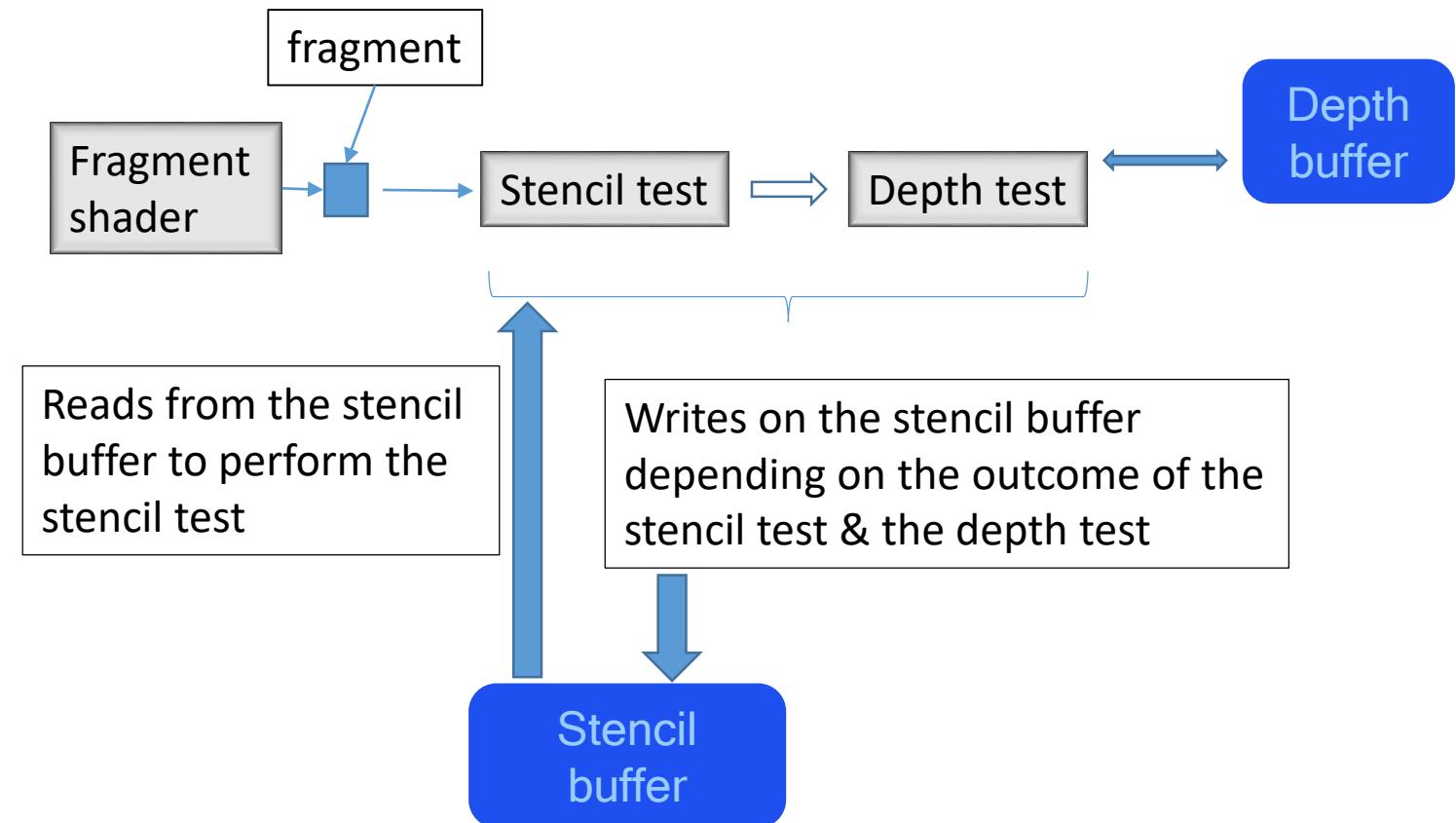
Other tests: Stencil Buffer

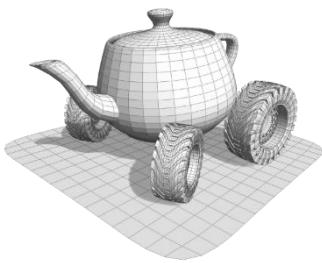


- The stencil buffer is a 8 bit buffer and it works similarly to the depth buffer
- each fragment is tested and the stencil buffer can be updated on the base of the outcome (more next slide..)

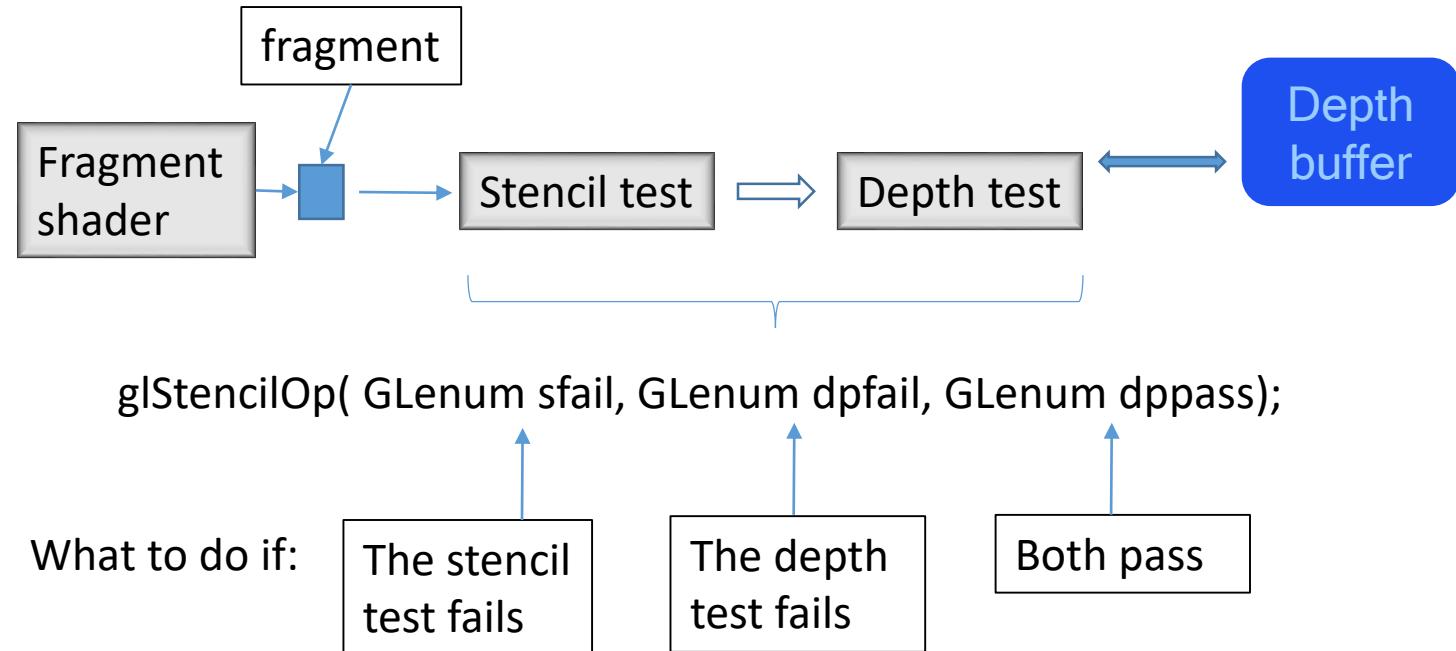


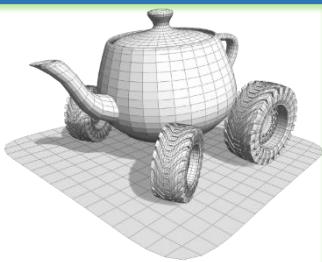
Other tests: Stencil Test





Stencil Test: what if..





Other tests: Stencil Test

```
setup_stencil (){  
    glEnable(GL_STENCIL_TEST);  
    glClear(GL_STENCIL_BUFFER_BIT);  
    glStencilMask(0xFF);  
    glStencilFunc(GL_ALWAYS, 1, 0xFF);  
    glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);  
    draw_the_mask();  
    glStencilMask(0x00);  
}
```

```
draw_scene = function(){  
    glStencilFunc(gl.EQUAL, 1, 0xFF);  
    draw_the_scene();  
}
```

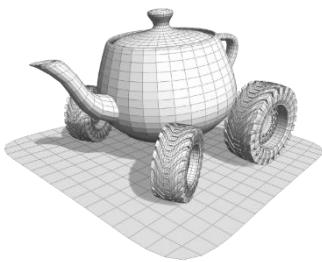
Enable writing to the stencil buffer on all its bits

Allow all fragments to pass the stencil test

replace the value of the stencil buffer with the reference value specified in **stencilFunc**

Do not write on the stencil buffer anymore

A fragment pass only if the corresponding stencil value is 0



Blending

- If a fragment passes the stencil and depth tests, its associated color value “contributes” to the final color written on the screen buffer
- “contributes” means that it’s no necessarily written “as is” in the framebuffer. In fact, it can be linearly combined what the value currently written on it
- This happens if **blending** is enabled

s : source color computed by the fragment shader

d : destination color stored in the framebuffer



Blending: the α value

- The α value is the fourth component of the RGBa color and its says how opaque the color is:
 - $\alpha = 0 \rightarrow$ fully transparent
 - $\alpha = 1 \rightarrow$ fully opaque
- s_F and d_F are defined as function of s and d in a *predefined* set of modes
- \oplus default is $+$ but it may be other things

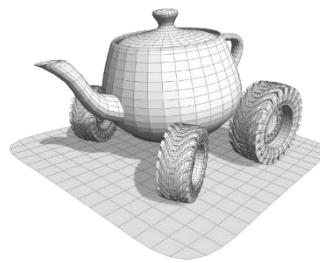
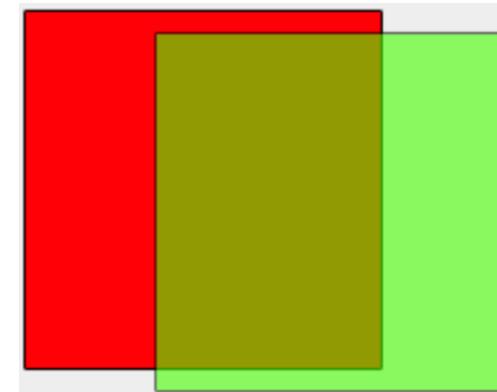
Component-wise multiplication

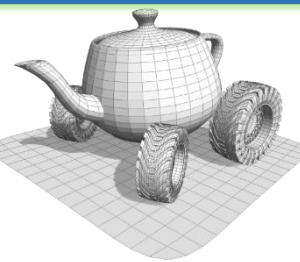
$$d' = s * F_s \oplus d * F_d$$

$$\begin{bmatrix} d'_R \\ d'_G \\ d'_B \\ d'_\alpha \end{bmatrix} = \begin{bmatrix} s_R \\ s_G \\ s_B \\ s_\alpha \end{bmatrix} \begin{bmatrix} F_{sR} \\ F_{sG} \\ F_{sB} \\ F_{s_\alpha} \end{bmatrix} \oplus \begin{bmatrix} d_R \\ d_G \\ d_B \\ d_\alpha \end{bmatrix} \begin{bmatrix} F_{dR} \\ F_{dG} \\ F_{dB} \\ F_{d_\alpha} \end{bmatrix}$$

Example

$$d' = \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} * 0.4 + \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{bmatrix} * 0.6 = \begin{bmatrix} 0.4 \\ 0.6 \\ 0.0 \\ 0.76 \end{bmatrix}$$

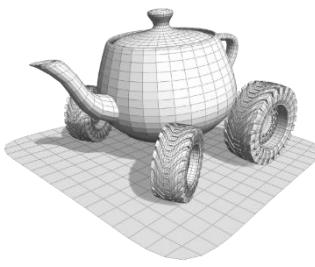




Blending: blending factors

```
gl.enable(gl.BLEND);  
gl.blendFunc(GLenum sfactor, GLenum dfactor)
```

GL_ZERO	Factor is equal to 0.
GL_ONE	Factor is equal to 1.
GL_SRC_COLOR	Factor is equal to the source color vector
GL_ONE_MINUS_SRC_COLOR	Factor is equal to 1 minus the source color vector
GL_DST_COLOR	Factor is equal to the destination color vector
GL_ONE_MINUS_DST_COLOR	Factor is equal to 1 minus the destination color vector
GL_SRC_ALPHA	Factor is equal to the alpha component of the source color vector
GL_ONE_MINUS_SRC_ALPHA	Factor is equal to 1-alpha of the source color vector.



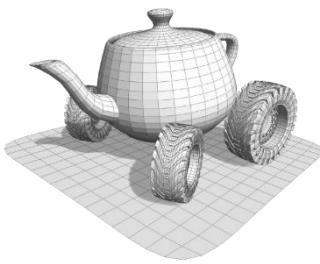
Blending for transparent surfaces

- Blending is typically used for showing transparent surfaces such as windows, windshields etc..
- How to:
 1. Draw all opaque objects (those with color alpha =1)
 2. Enable blending and draw all transparent objects in **back-to-front** order using blend function:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

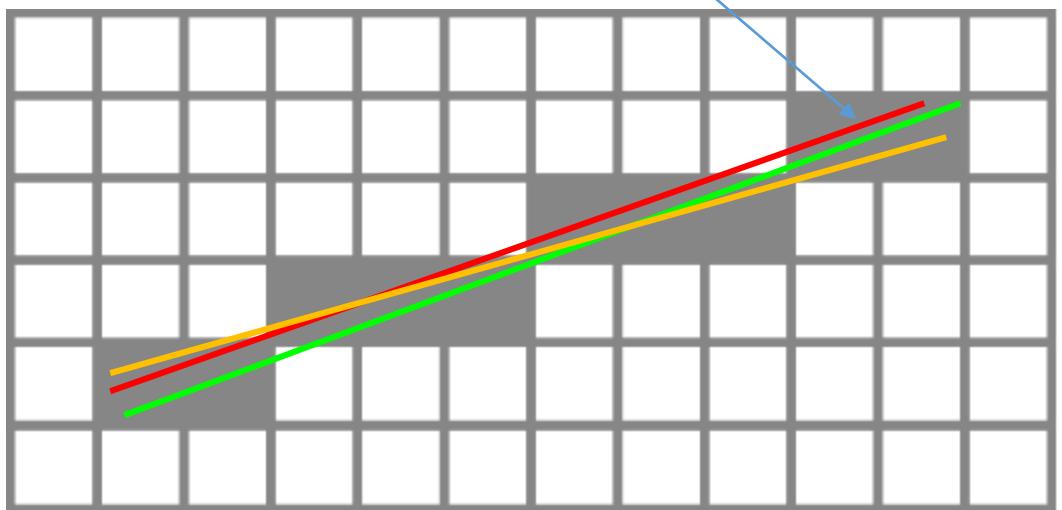
- Ratio:
 - the transparent object contributes to the color proportionally to its opacity
 - The resulting color is a point between the two (barycentric coordinates...)

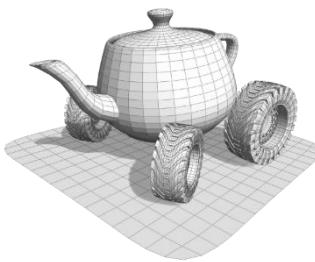
Aliasing



- **Aliasing** refers to the fact that a set of pixels may represent different things
- in CG it is a consequence of the discretization in pixels
- The set of pixels that represent a segment may in fact represent infinite other segments
- We see aliasing when with can spot the corner of pixels, we see *jagged edges*

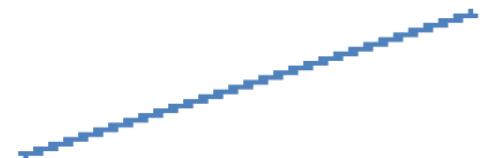
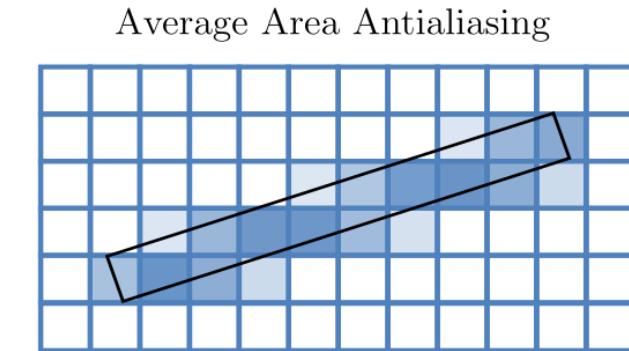
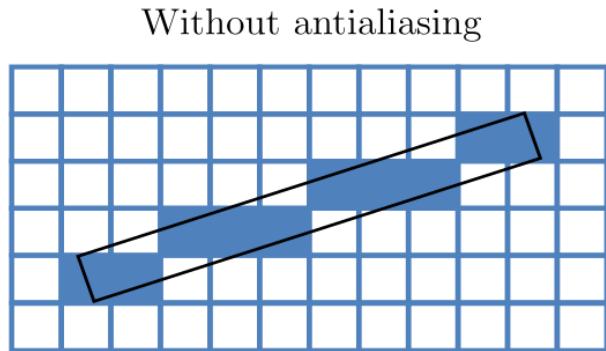
These 3 segments, and infinite others, rasterize to the same set of pixels



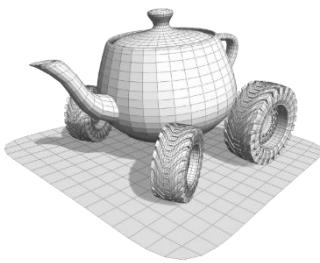


Antialiasing

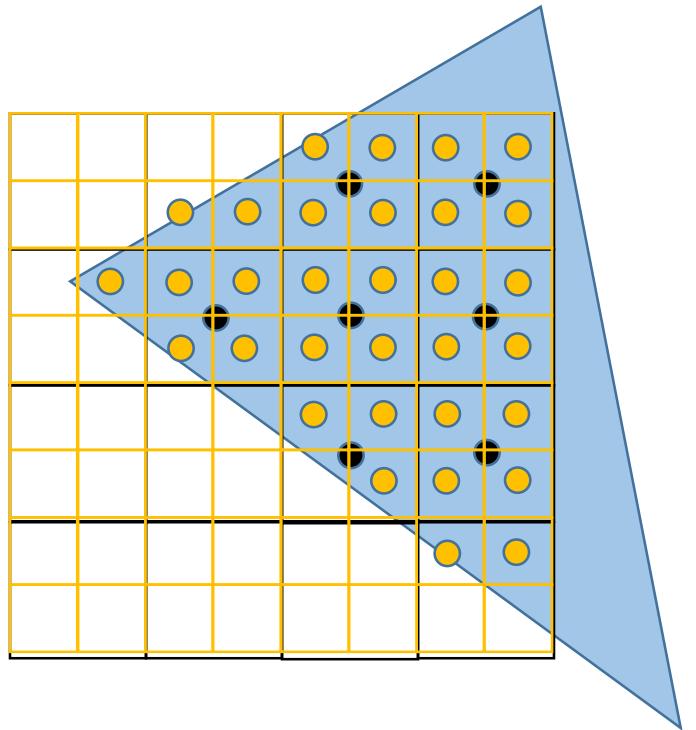
- Key idea: if a pixel is only partially covered by a primitive than its color «less saturated» (recall HSV?)
so, assign a color with an alpha proportional to the area of the pixel occupied



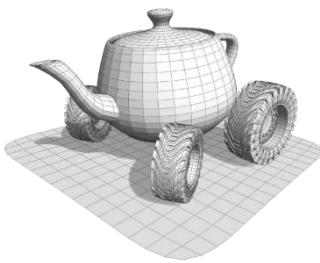
Antialiasing: how to in WebGL



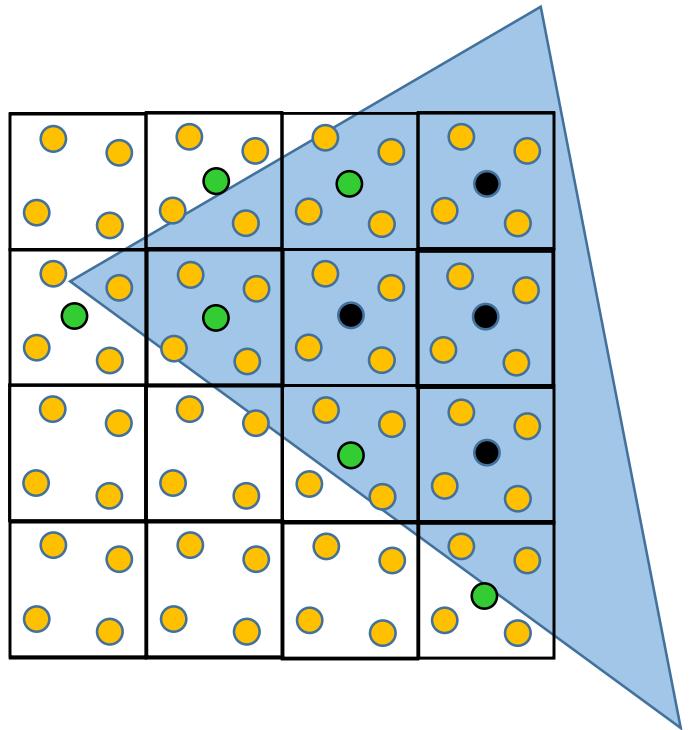
- **Super Sampling Anti Aliasing (SSAA)** it just renders the scene on a larger framebuffer (and depth/stencil buffer) and the average down the result
- It works but it costs
 - Memory, because of the buffers
 - Time: because of the multiple fragment shaders which must run
- Example: 2x2 size-> 4 times memory and 4 time fragment shaders runs

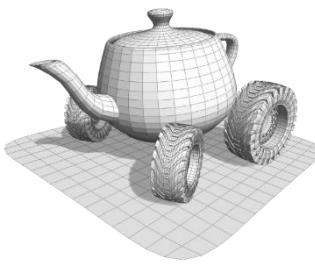


Antialiasing: how to in WebGL



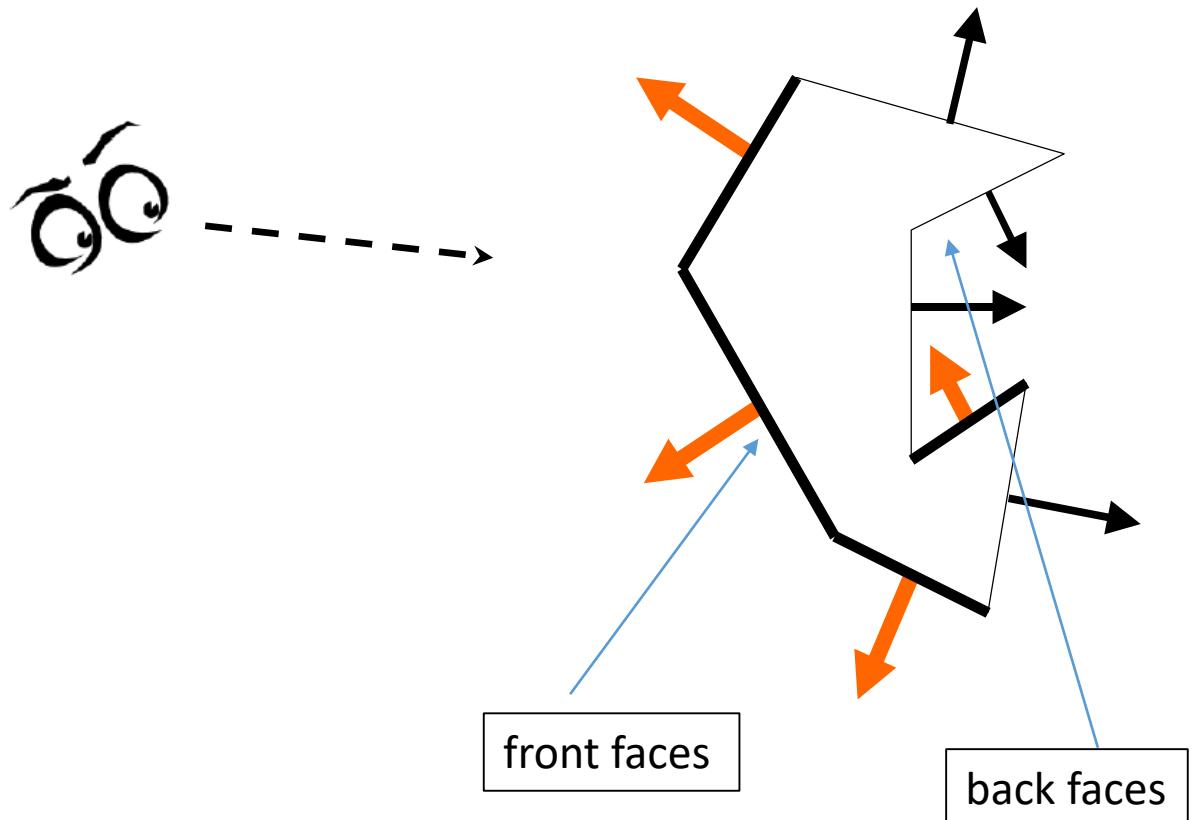
- **Multi Sampling Anti Aliasing (MSAA)**
 - More samples per pixels (e.g. 4)
 - If at least one is covered the fragment is activated with the color interpolated as for the pixel's center
 - Output of the fragment weighted by the number of covered samples
- Only one fragment shader runs (for each fragment)
- Only depth/stencil buffer need to be bigger (to resolve HSR)

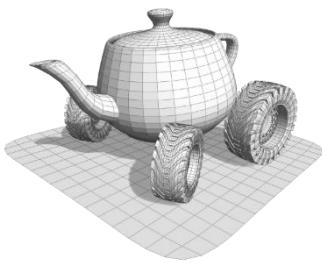




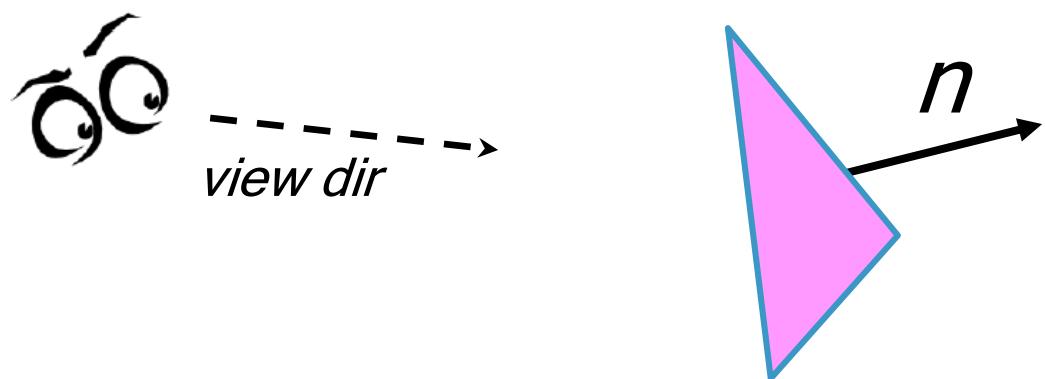
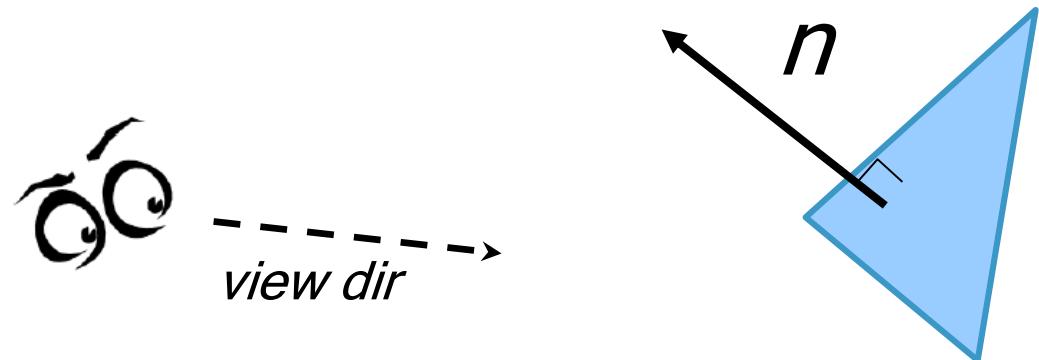
Backface culling

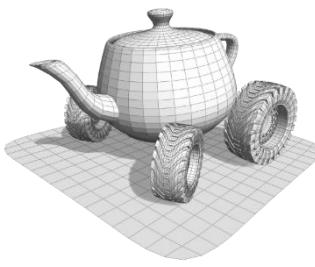
- If we know our scene is made of closed meshes we can avoid rendering **back faces**
 - Unless we can also go “inside” the closed objects
- **Back face:** a face whose normal points “away” from the observer
- In a closed mesh, back faces are occluded by front faces





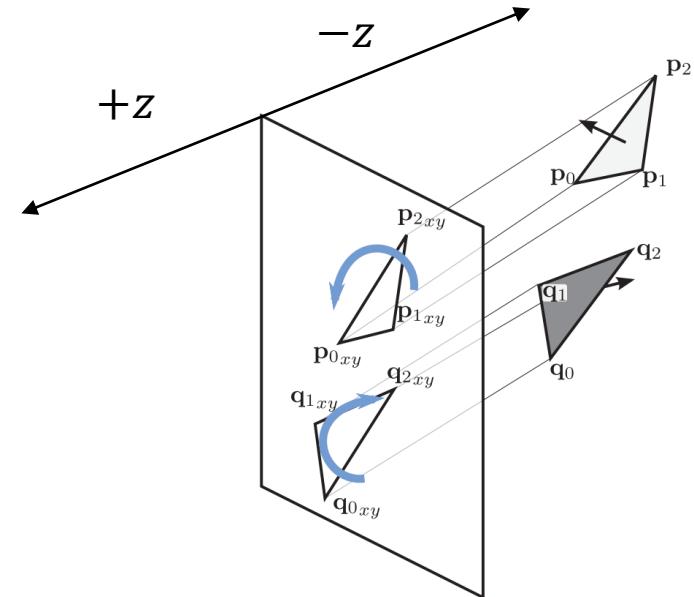
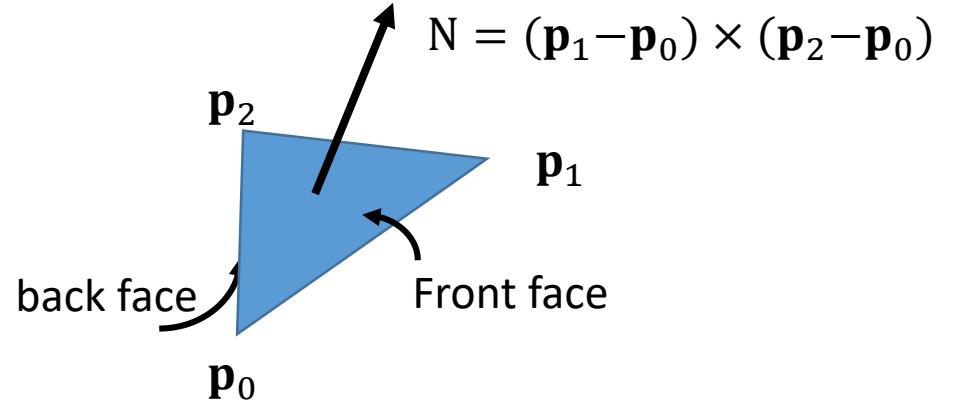
Front facing – back facing

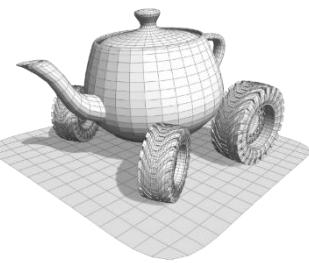




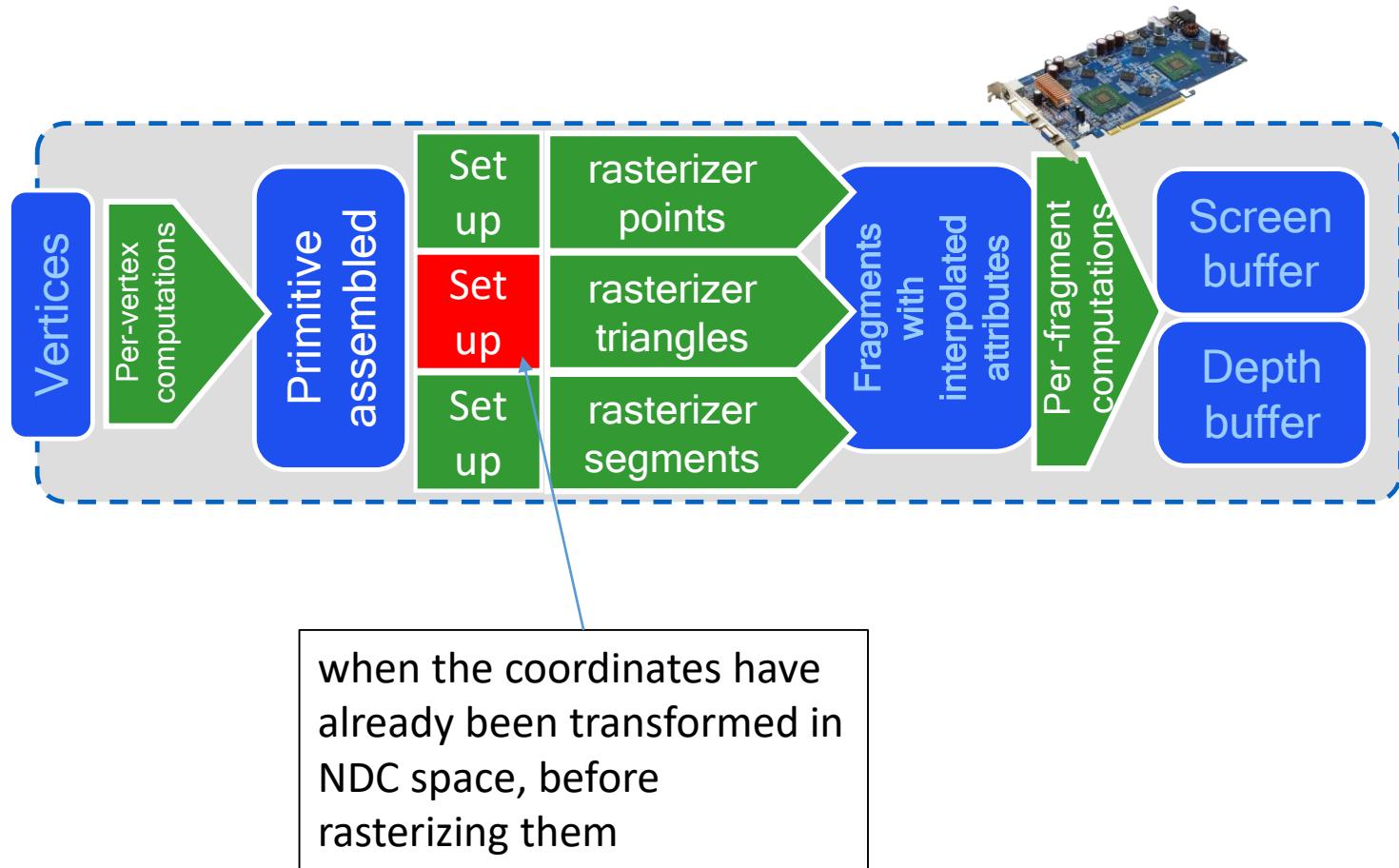
Detecting Back Faces

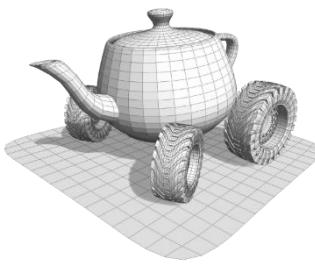
- The front of a face is the side towards its normal
- Consider the projection of a triangle $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ onto the near plane
- Compute the normal with the 2D projected points $\mathbf{p}_{0xy}, \mathbf{p}_{1xy}, \mathbf{p}_{2xy}$:
 - If positive -> front facing
 - If negative -> back facing, discard





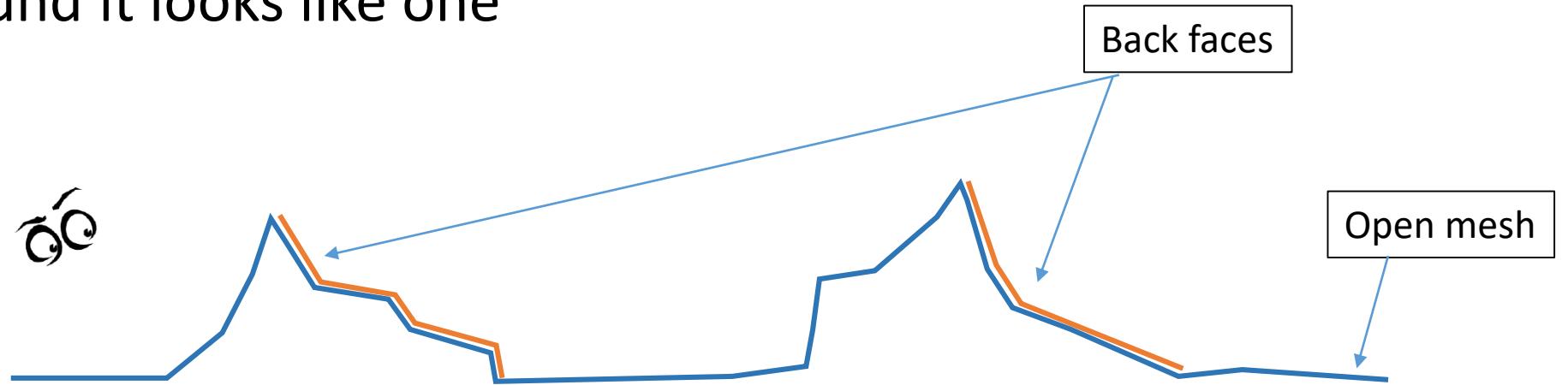
Where it's done





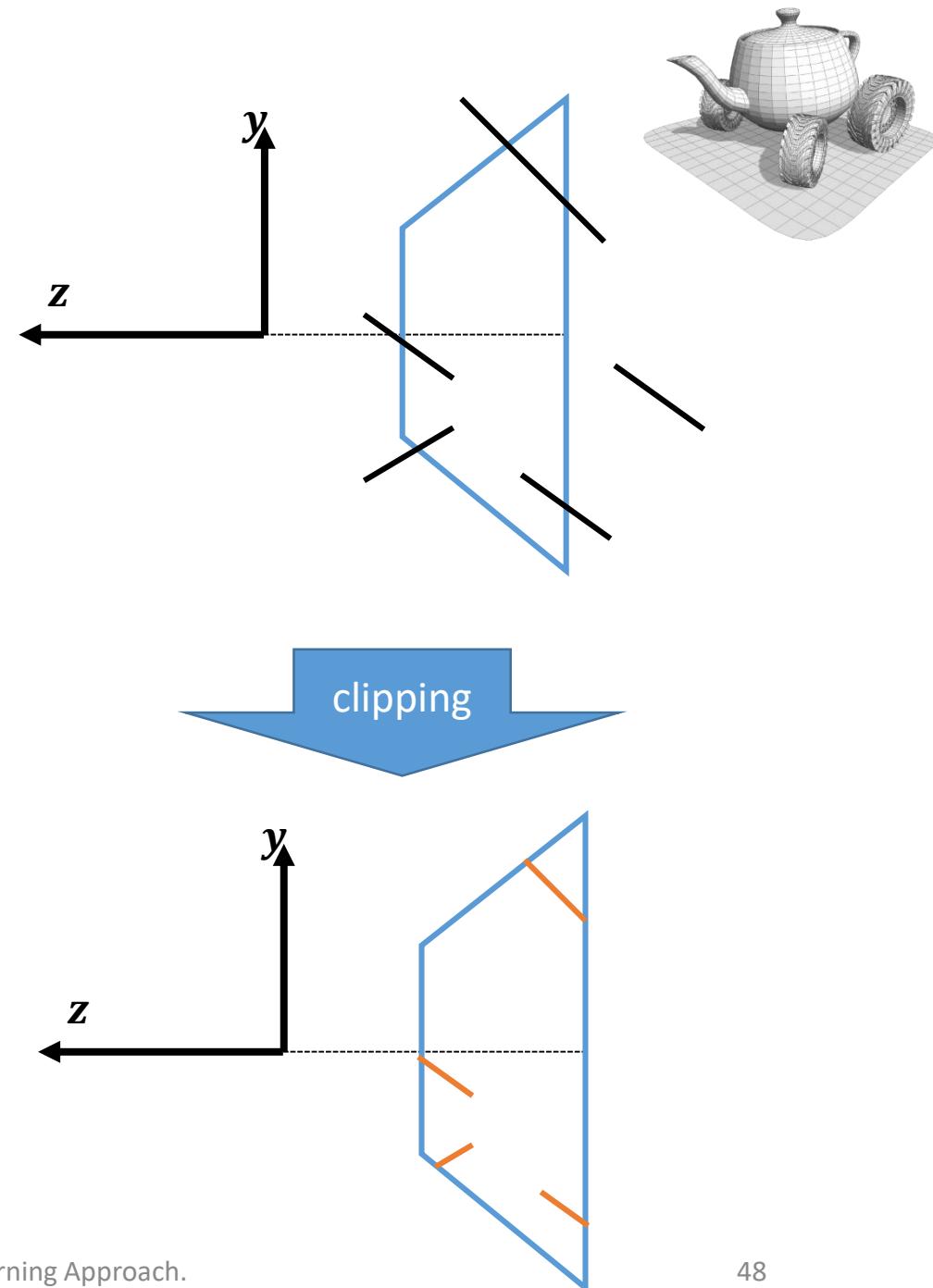
Backface culling

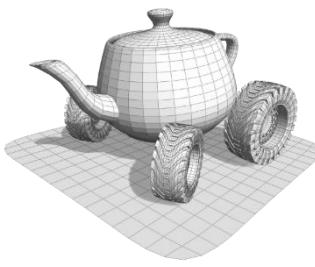
- note: in order to apply back face culling, the surface needs to be «appear closed» just from the where the admitted points of view in the scene
- Example: a terrain is not a closed object but if we cannot go underground it looks like one



Clipping

- Not all geometric primitives intersect the view volume.
 - There is no point in producing fragment outside the viewport
- **Clipping** is the action of transforming a primitive into the part of it that intersects the view volume





Clipping segments (1/4)

- **Cohen-Sutherland (2D case)**

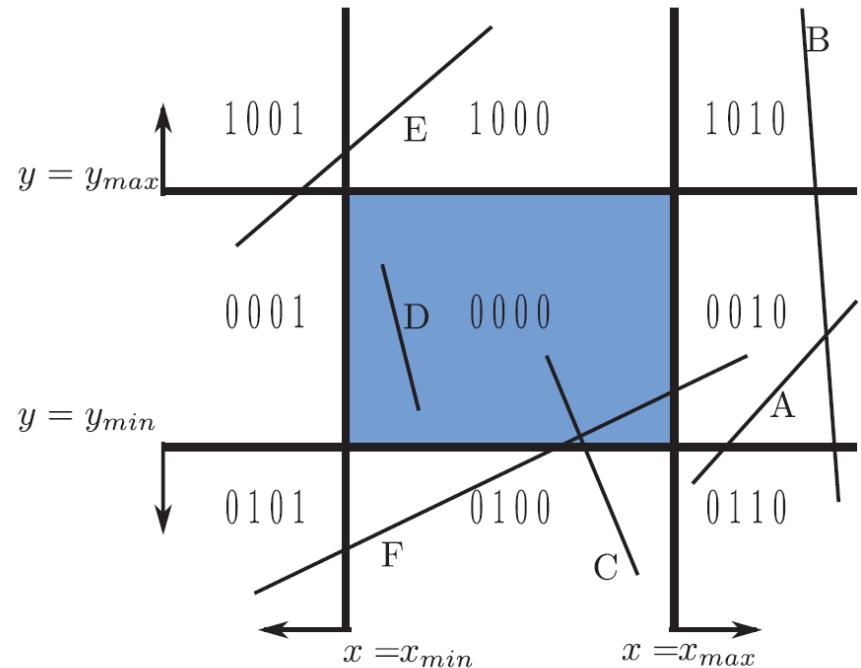
- The clipping rectangle is defined as the intersection of four axis aligned planes
- The planes partition the space in 9 quadrants
- A simple $< >$ test on each coordinates of the segment end points creates a 4 digit bit-code

$$b_{+x}(\mathbf{p}) = \begin{cases} 1 & \mathbf{p}_x > x_{max} \\ 0 & \mathbf{p}_x \leq x_{max} \end{cases}$$

$$b_{+y}(\mathbf{p}) = \begin{cases} 1 & \mathbf{p}_y > y_{max} \\ 0 & \mathbf{p}_y \leq y_{max} \end{cases}$$

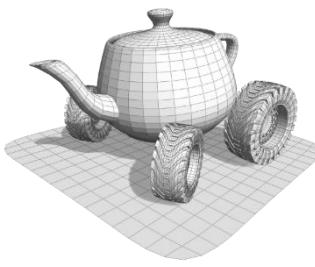
$$b_{-x}(\mathbf{p}) = \begin{cases} 1 & \mathbf{p}_x < x_{min} \\ 0 & \mathbf{p}_x \geq x_{min} \end{cases}$$

$$b_{-y}(\mathbf{p}) = \begin{cases} 1 & \mathbf{p}_y < y_{min} \\ 0 & \mathbf{p}_y \geq y_{min} \end{cases}$$



Bit code

$$R(p) = b_{+y}(p) \ b_{-y}(p) \ b_{+x}(p) \ b_{-x}(p)$$



Clipping segments (2/4)

- **Cohen-Sutherland (2D case)**

- The bitcodes of two endpoints gives two tests for discarding the segment as “non intersecting”

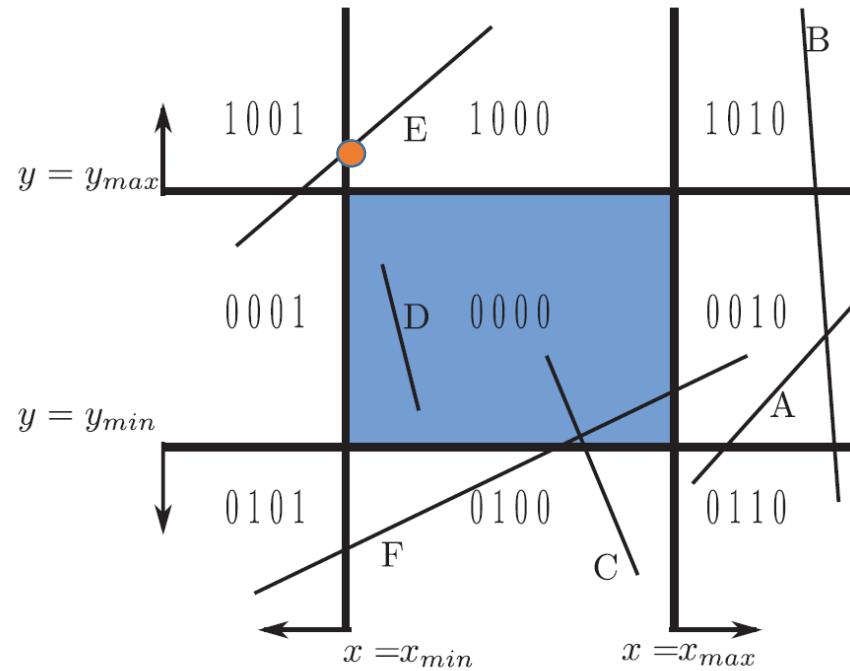
$R(\mathbf{p}_0) \& R(\mathbf{p}_1) \neq 0 \Rightarrow \text{no clip, outside}$

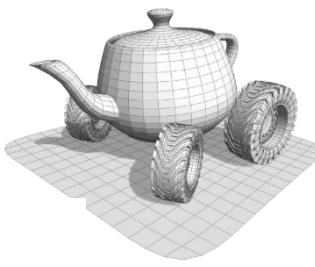
\mathbf{p}_0 and \mathbf{p}_1 are on the same side of one of the planes, e.g. A and B

$R(\mathbf{p}_0) \vee R(\mathbf{p}_1) = 0 \Rightarrow \text{no clip, inside}$

\mathbf{p}_0 and \mathbf{p}_1 are on the negative side of all of the planes, e.g. D

- If both fail compute the intersection with one plane and repeat





Clipping segments (3/4)

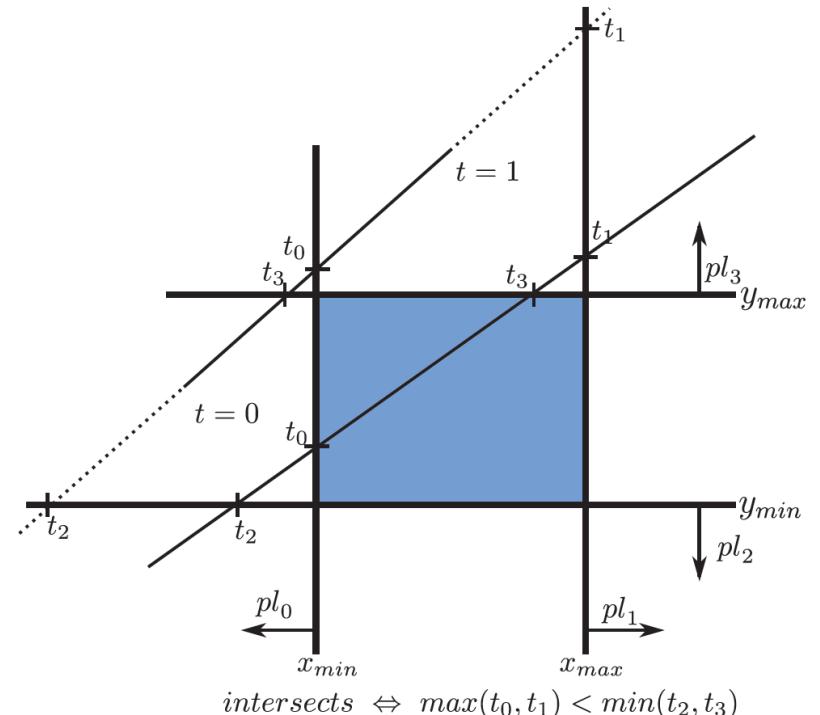
- Liang-Barsky use the parametric equation of the line

$$s(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0) = \mathbf{p}_0 + t\mathbf{v}$$

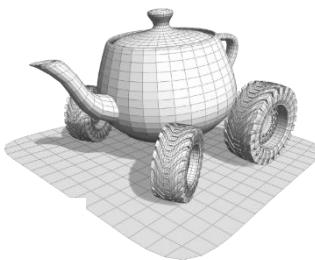
and find the crossing point between the line and the planes, e.g.

$$(\mathbf{p}_0 + \mathbf{v} t)_x = x_{min} \Rightarrow t = \frac{x_{min} - \mathbf{p}_0_x}{\mathbf{v}_x}$$

- the 4 intersections define «entry» and exit «points»
 - entry: from positive to negative halfspace
 - exit : from negative to positive halfspace



t_0 is an entry if $\mathbf{v}_x > 0$, exit otherwise
 t_1 is an entry if $\mathbf{v}_x < 0$, exit otherwise
 t_2 is an entry if $\mathbf{v}_y > 0$, exit otherwise
 t_3 is an entry if $\mathbf{v}_y < 0$, exit otherwise



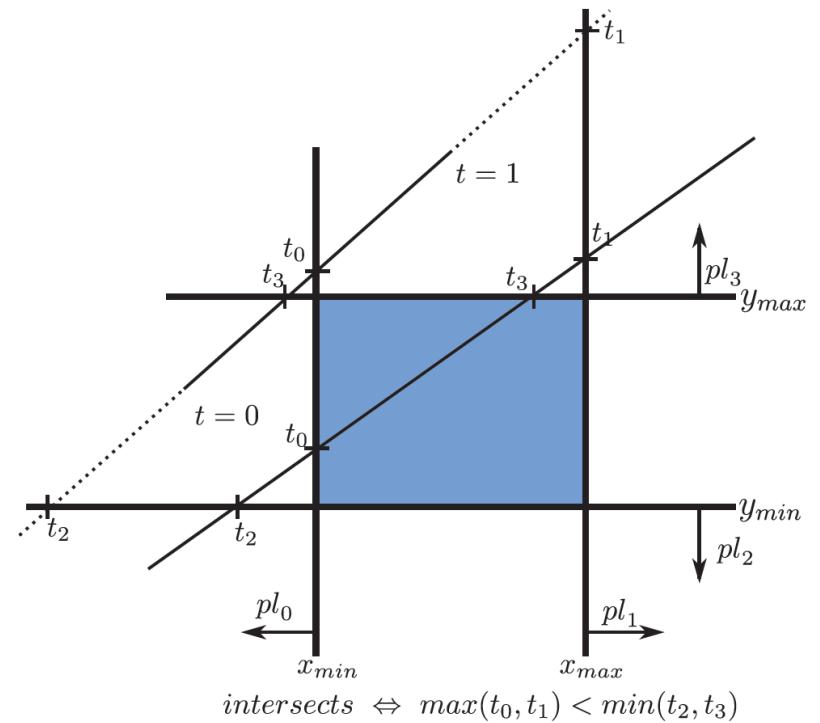
Clipping segments (3/ 4)

- If the first exit point is past the last entry point there is no intersection

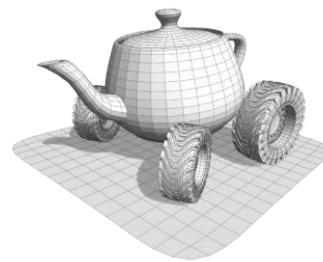
$$t_{min} = \max(0, \text{largest entry value})$$

$$t_{max} = \min(1, \text{smallest exit value})$$

- Otherwise $p'_0 = s(t_{min})$, $p'_1 = s(t_{max})$ is the clipped segment



t_0 is an entry if $v_x > 0$, exit otherwise
 t_1 is an entry if $v_x < 0$, exit otherwise
 t_2 is an entry if $v_y > 0$, exit otherwise
 t_3 is an entry if $v_y < 0$, exit otherwise



Vector images and raster images

- **Vector Images:**

a set of 2D geometric primitives, e.g.:

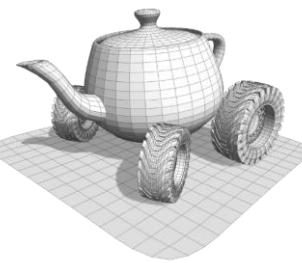
- Triangles,
polygons, circles, parametric
curves, text...

- **Raster Images:**

a regular 2D grid of tiles called pixel (**picture x element**)

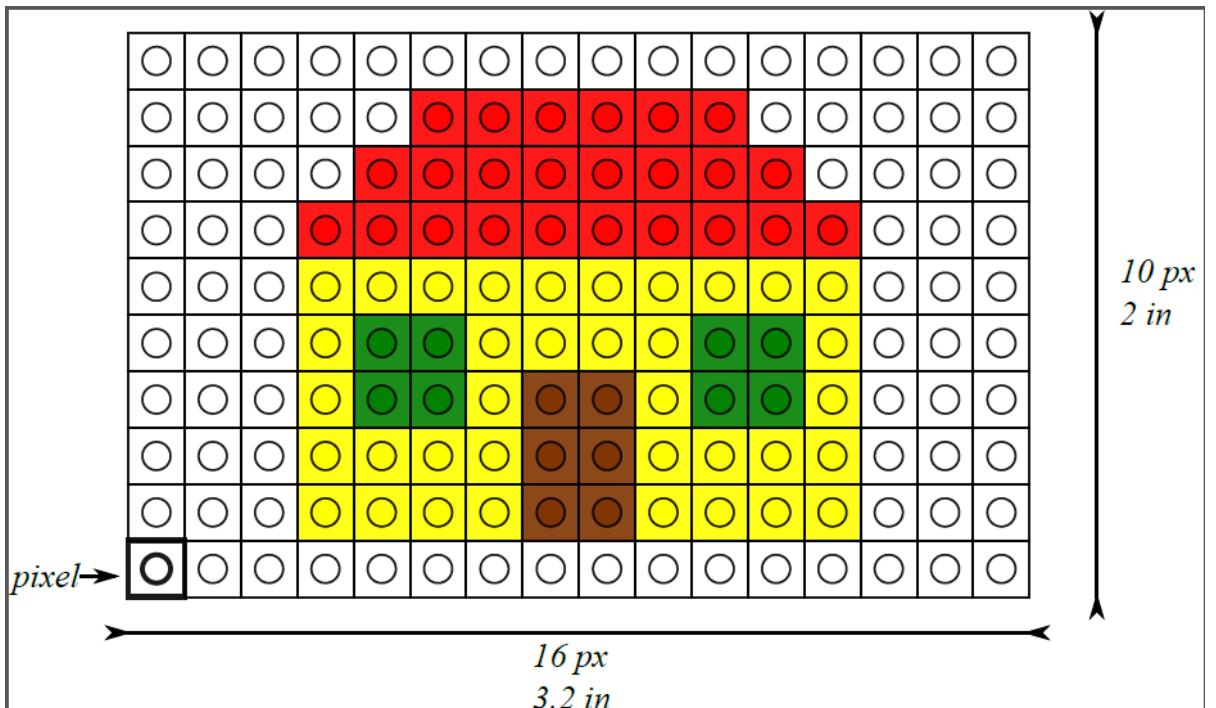


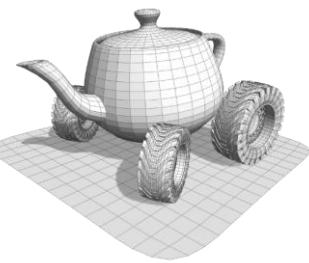
190	187	189	192	192	189	183	172	164	154	139	124	122	122	129	122	111	110	118	112
191	189	191	189	191	190	183	169	158	144	138	136	129	126	136	120	90	81	88	73
194	194	193	190	192	187	180	162	153	138	137	132	103	110	97	71	59	57	55	51
198	198	198	193	193	183	170	151	138	115	100	97	68	70	57	49	50	55	53	56
203	199	197	190	185	173	159	132	118	87	58	62	53	51	54	51	50	52	51	51
203	202	194	188	178	158	128	91	78	59	56	60	49	55	58	55	49	57	62	58
207	203	199	186	156	118	75	61	56	53	59	53	51	52	62	51	52	71	86	91
208	202	189	159	101	56	54	58	51	47	50	55	54	50	57	63	51	86	127	124
207	197	167	112	65	49	50	51	45	46	42	49	59	58	81	113	92	75	157	150
204	181	122	81	54	50	49	44	43	49	44	44	55	56	91	148	128	69	161	164
193	143	92	67	50	50	53	60	52	48	43	45	61	57	77	143	137	76	150	176
158	111	86	69	56	50	65	67	63	54	45	40	60	60	68	99	106	70	143	179
129	101	78	79	78	51	60	84	80	63	42	46	72	85	71	83	76	69	149	178
124	90	85	103	97	61	61	94	100	89	75	67	87	105	85	75	58	87	162	176
110	93	105	120	120	93	58	85	97	100	93	86	88	89	87	70	64	123	174	173
104	105	108	128	134	118	81	65	85	103	100	96	98	82	69	71	111	161	179	173
99	111	114	129	144	148	111	90	77	75	89	88	86	75	76	115	158	180	182	177
100	97	107	125	137	150	139	114	105	88	79	79	86	100	123	156	183	181	177	172



Raster images: resolution

- Resolution indicates the number of pixel per unity of space (e.g. Pixels per inch)
- It is popular to indicate the size in pixel as resolution

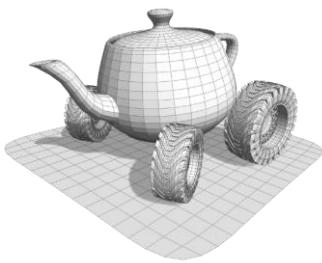




Raster images: channels

- The **channels** of a raster image indicates what information is written on each pixel, e.g.:
 - Gray: 1 channel encoding the grey level
 - RGB: three channels, Red Green and Blue
 - RGBa: RGB plus a channel indicating the transparency





Raster images: Image Depth (Color Depth)

- **Image Depth**

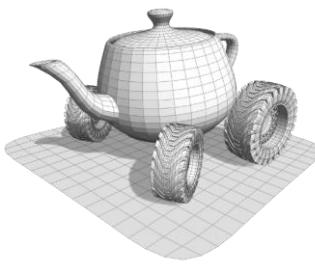
- e.g.: RGB
- e.g.: B & W



Introduction to Computer Graphics: a Practical Learning Approach

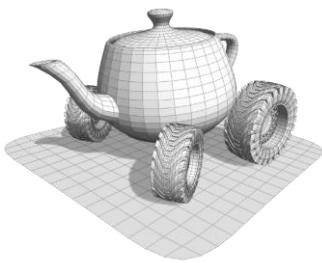
By Thegreenj - Own work, CC BY-SA 3.0,

<https://commons.wikimedia.org/w/index.php?curid=2048094>



Raster Images: memory occupation

- Memory: $\text{resX} \cdot \text{resY} \cdot \text{imageDepth}$
 - a 240×480 “true color” image
 $\text{memory occupation} = 240 \cdot 480 \cdot 3 \text{ bytes} = 345.600 \text{ bytes}$
 - 4000×2000 (8 MegaPixel) “true color”: $4000 \cdot 2000 \cdot 3 = 24 \text{ MB}$
- Often compressed:
 - Lossless compression: preserve the original data
 - Lossy compression: more aggressive, loose some of the original information
 - Texture compression: a lossy compression used in graphics at rendering time and implemented on GPU (more later on..)

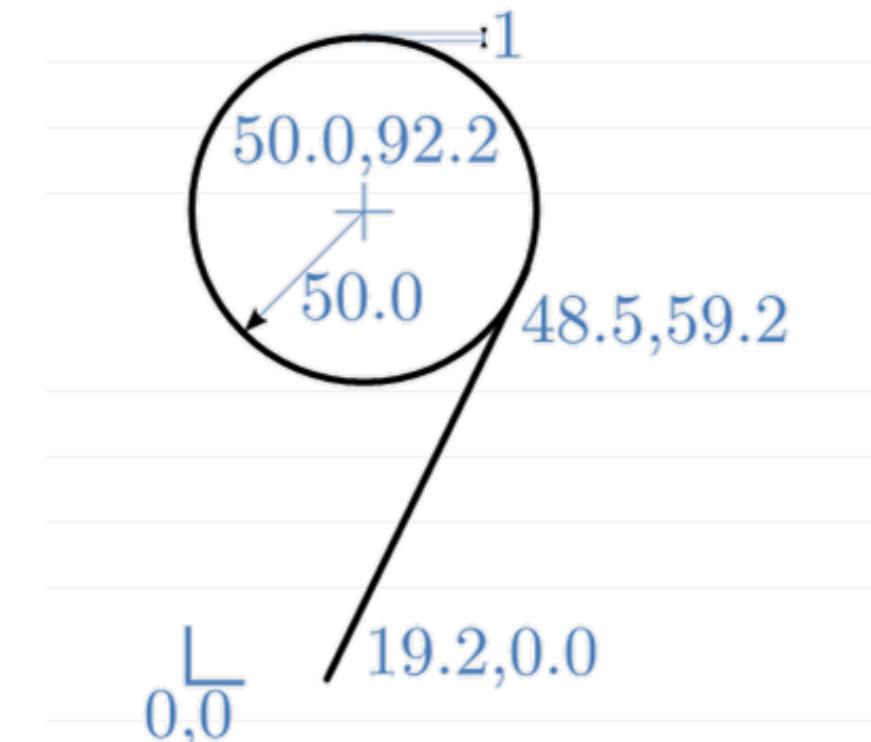


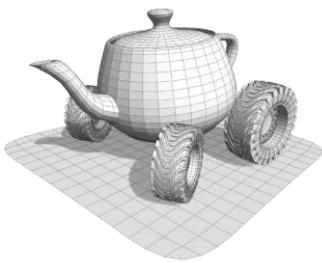
Vector image

```
NUMBER_OF_PRIMITIVES 2

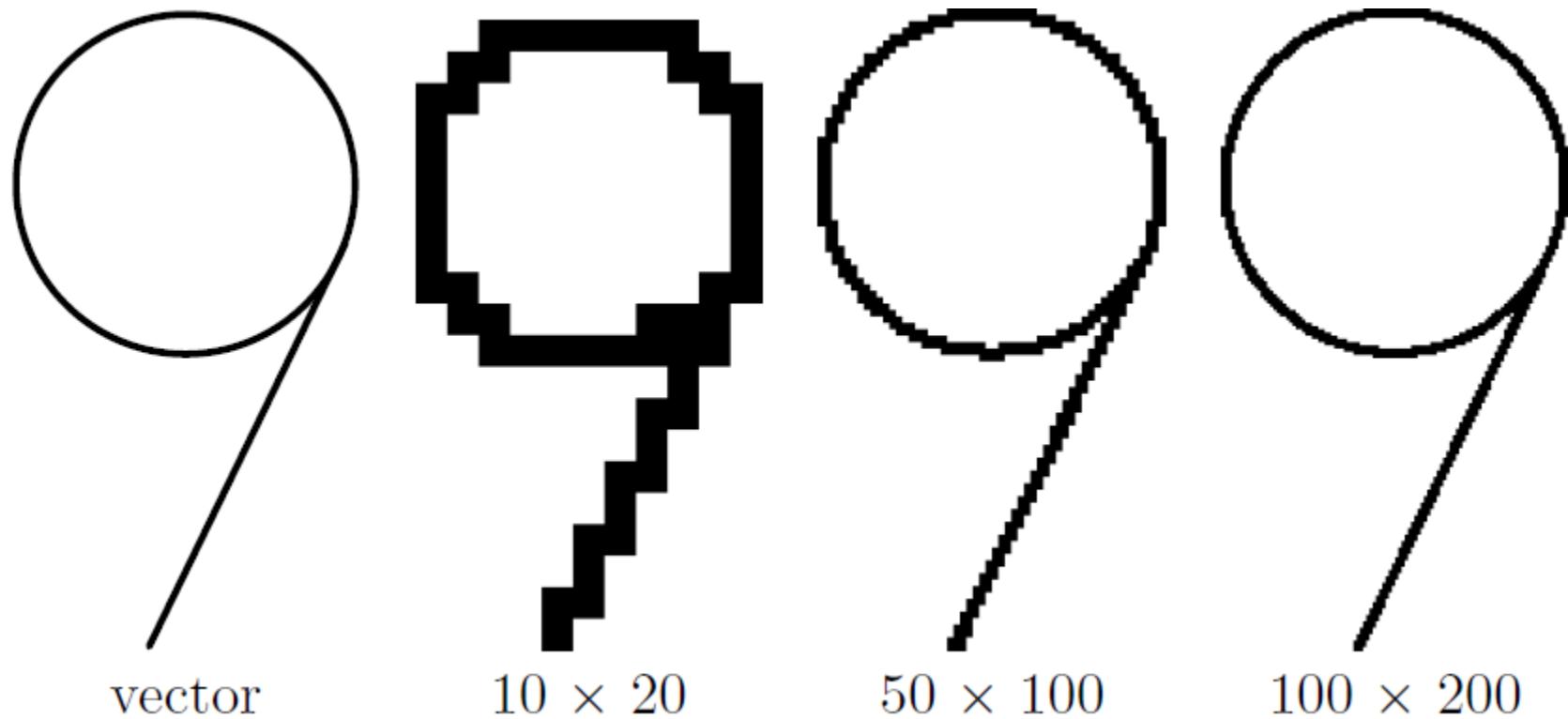
CIRCLE
center 50.0, 92.2
radius 50.0
fill_color 1.0, 1.0, 1.0
line_color 0.0, 0.0, 0.0
line_thickness 1pt

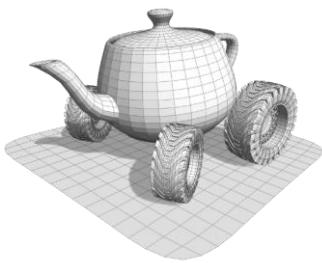
SEGMENT
endpoint_one 19.2, 0.0
endpoint_two 48.5, 59.2
line_color 0.0, 0.0, 0.0
line_thickness 1pt
```





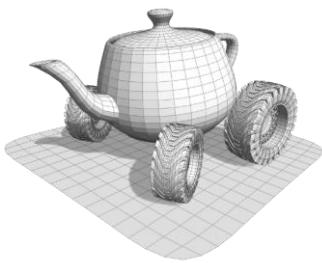
Rasterizing vector images





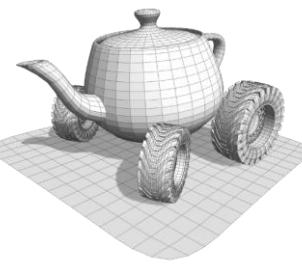
Pros and Cons

- Vector images:
 - independent from the device resolution!
 - well suited for: logos, diagrams, stylized drawings ...
 - can be very compact in space
 - interpretation: difficult (a rendering required)
 - resolution (number of primitive, of control points...) is adaptive
 - when zoomed in:
there may be little detail to look at, but it still looks good
- Raster images
 - well suited for: natural images (e.g. photographs)
 - quality depends on image resolution
 - interpretation: direct (monitors can display them directly)
 - when zoomed in: artifacts



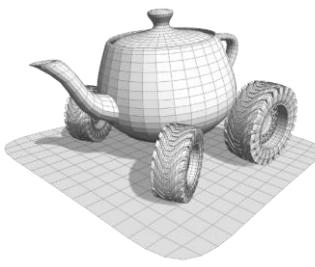
Raster Images: common formats

- **PNG (Portable Network Graphics):**
 - lossless compression (based on same algorithm as Zipped files)
 - many formats, including with 3 or 4 channels
 - good for synthetic images
- **JPEG (Joint Photographic Experts Group):**
 - (typically) lossy compression (based on “DCT”: discrete cosine transform)
 - 3 channels of 8 bits each
 - good for natural images (digital photography)
- **GIF (compuserve)**
 - strange *quirk* of the used image format
 - used for tiny animations over two decades ('90s, 2000's)
 - use declining



Raster Images: not so common formats

- TIFF
 - can be lossless or lossy
 - hi-dynamic range data (more than 8 bits per channel)
 - used for hi-quality digital photography
- RAW
 - a direct capture of a sensor (es camera CCD)
 - non compressed
 - non processed
- PNM (portable any map)
 - no compression
 - pixel values stored as ASCII (or binary)
 - not very used
 - but, useful: simple format, human readable values (ASCII numbers)
 - and, trivial to parse (an importer / exporter can be written in any language in a few lines of code)

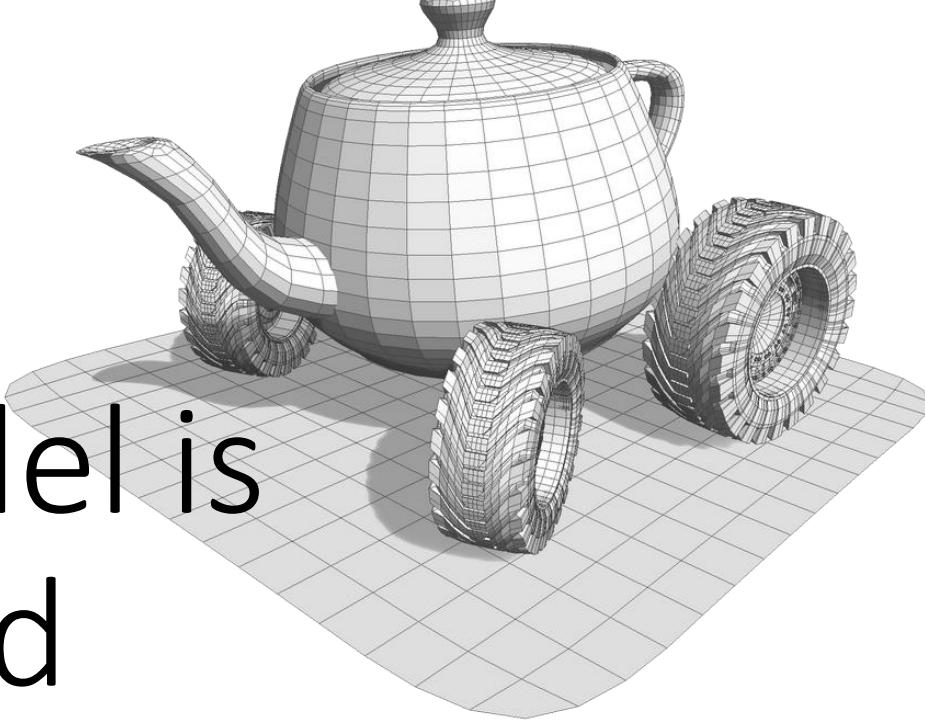


Vector Images: common formats

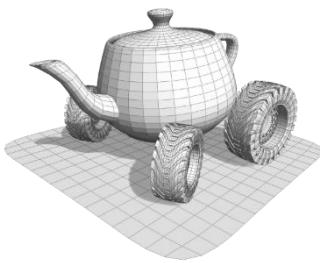
- **SVG:**
 - developed by W3C
 - a XML file describing the primitives
 - based on Bèzier curves (including, filled ones)
 - plus: basic shapes, text, colors, patterns ...
 - directly embeddable in Web pages (understood by all browsers)
 - example of Opensource editor / authoring tool: inkscape
- **PostScript (PS):**
 - designed for printers
 - set of instructions to send to a printer to print the image
 - includes ink control
- **Portable Document Format (PDF)**
 - by Adobe
 - includes subsets of PS

How a 3D Model is Represented

Geometry



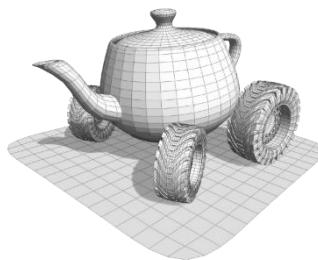
What do we mean by «3D Model»?



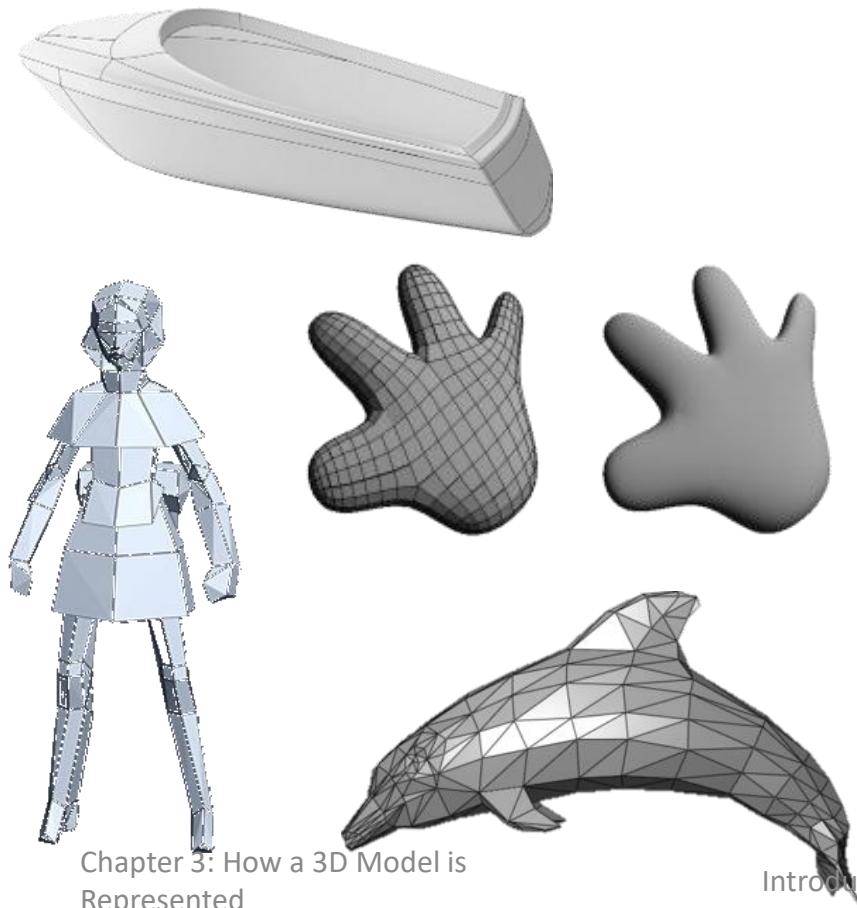
- 3D model: a mathematical representation of a physical entity



Types of 3D digital models

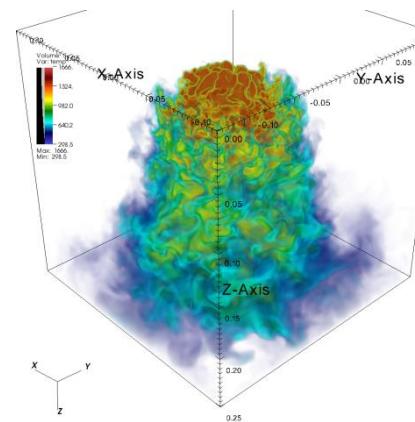
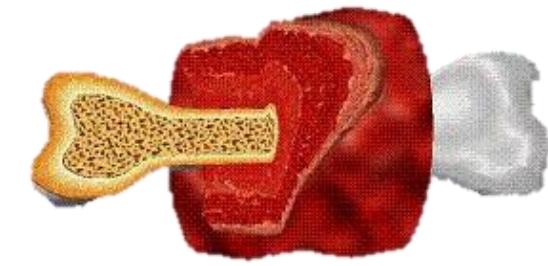


Surfaces (boundary-based)

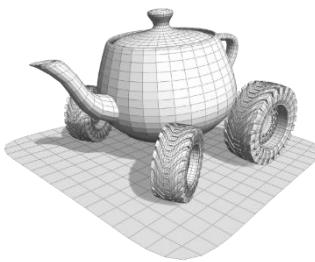


Chapter 3: How a 3D Model is Represented

Volumetric Models

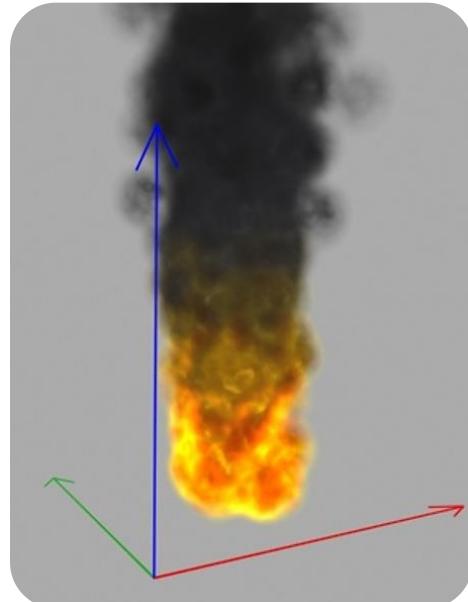


Introduction to Computer Graphics: a Practical Learning Approach.

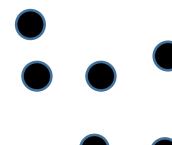


Types of 3D digital models

Point based

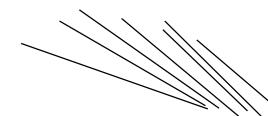


segments



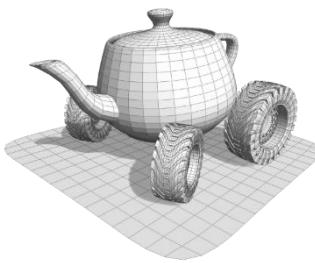
Chapter 3: How a 3D Model is Represented

Hey! Those are not 3 dimensional !1!!



Introduction to Computer Graphics: a Practical Learning Approach.

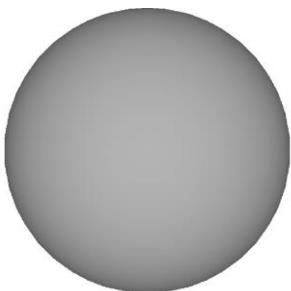
surfaces



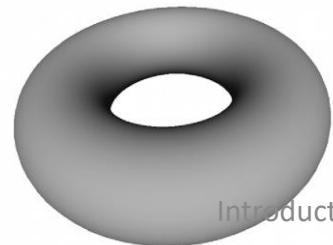
Implicit surfaces: the surface is the **zero set** of some function

$$S = \{(x, y, z) : f(x, y, z) = 0\}$$

$$S = \{(x, y, z) : x^2 + y^2 + z^2 - r^2 = 0\}$$

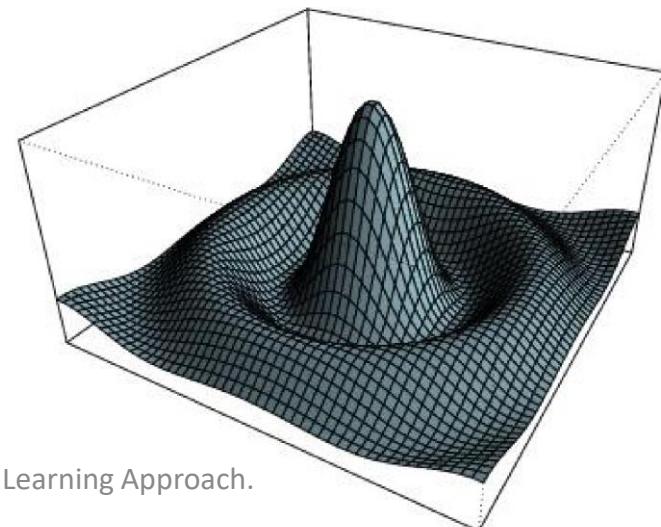


$$S = \{(x, y, z) : (x^2 + y^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2) = 0\}$$

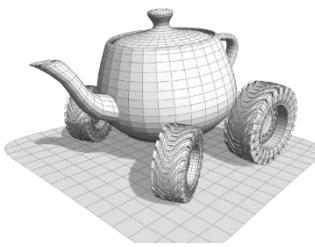


Parametric surfaces: the **codomain** of some function
 $S: \mathbb{R}^2 \rightarrow \mathbb{R}^3$

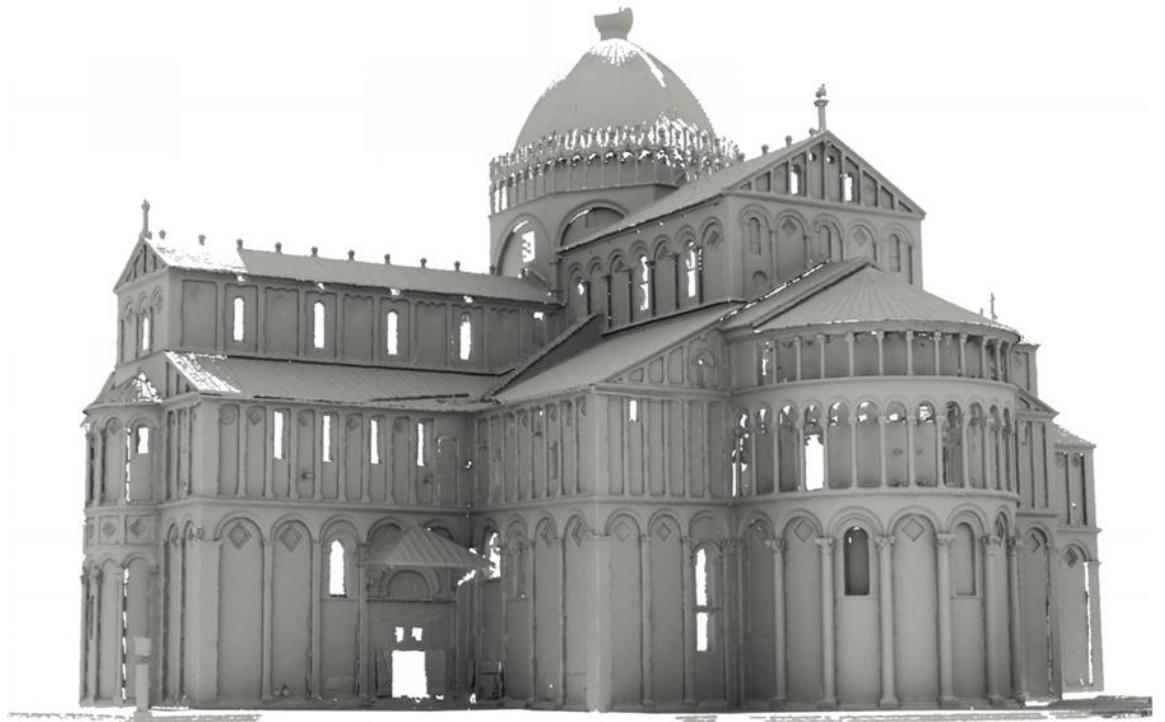
$$S(x, y) = (x, y, \frac{\sin(\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}})$$



surfaces

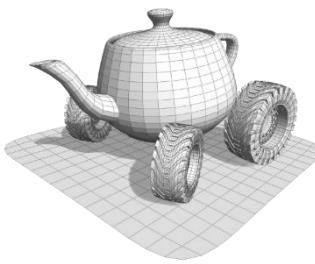


- **Polygon meshes** (or tessellated surfaces)
 - Real-world objects are difficult to define analytically
 - Idea: Complex shapes can be represented as collections of basic *building blocks*

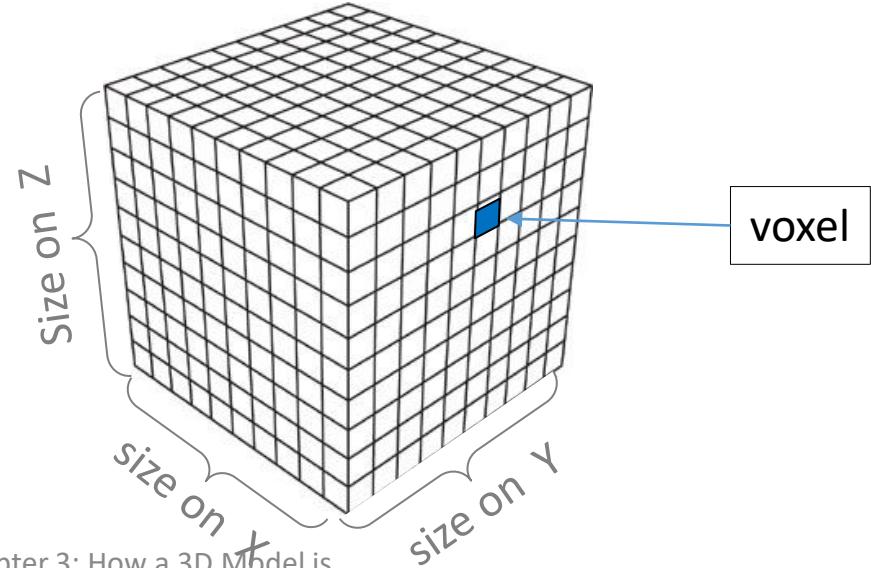


$$S(x, y, z) = \dots ??$$

Volumetric models

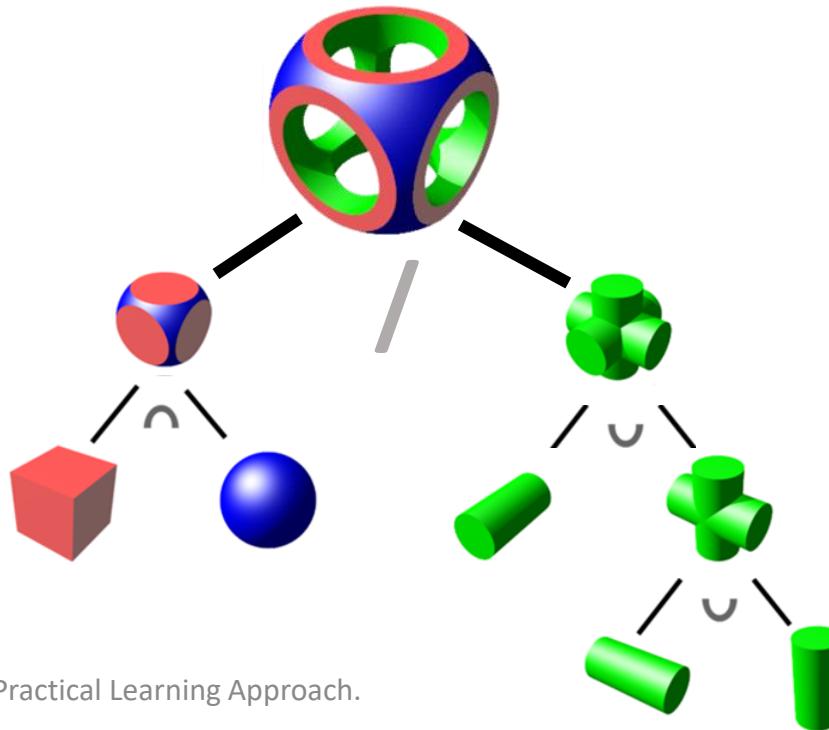


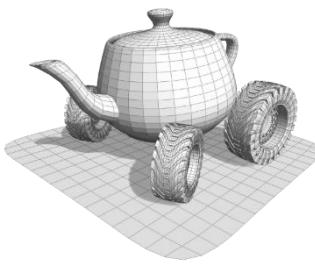
- Voxel based: the 3D version of the raster images
 - **Pix-el** = picture element
 - **Vox-el** = volume element



Chapter 3: How a 3D Model is Represented

- **Constructive Solid Geometry**
 - Set operations between simple volume primitives

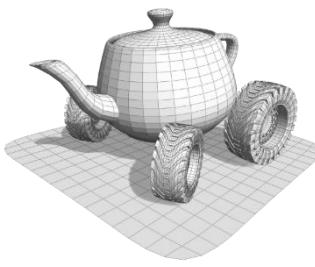




Types of 3D digital models

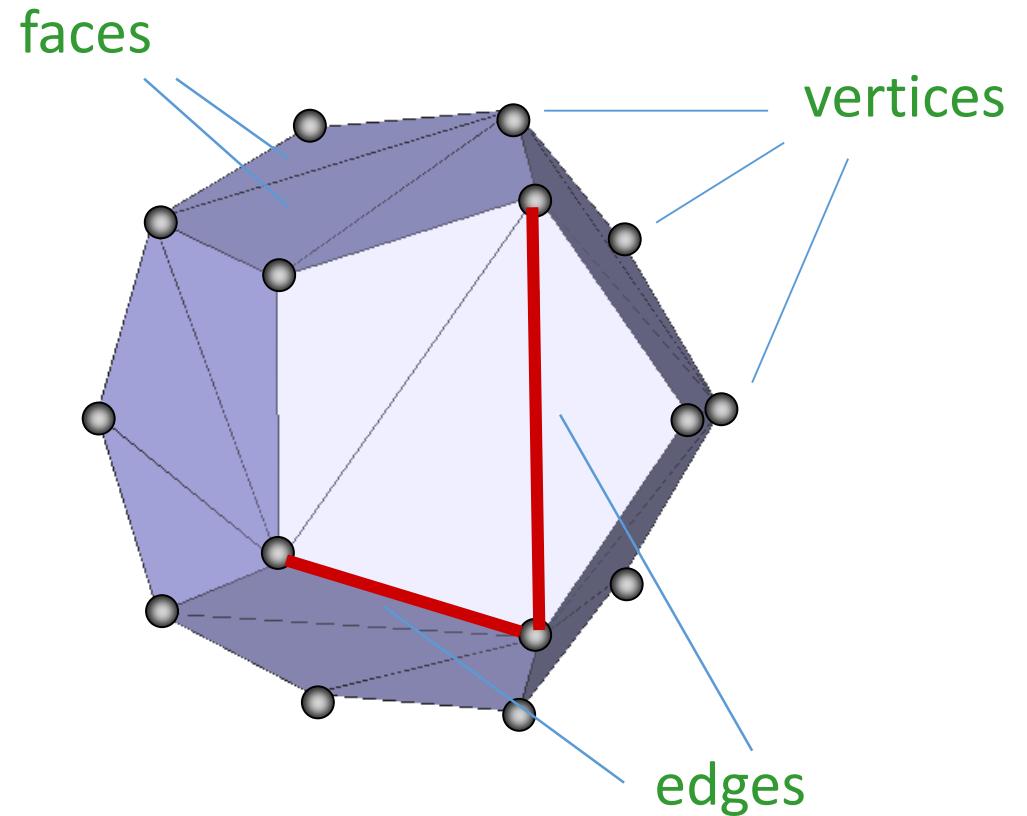
- Surfaces
 - Implicit surfaces
 - Parametric surfaces
 - Polygon meshes
- Volumetric
 - Voxellization
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes
- Points

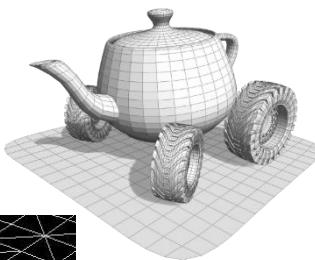
These are not the **only ways** to represent object.
They the way objects are commonly represented in CG



Polygon meshes(informal)

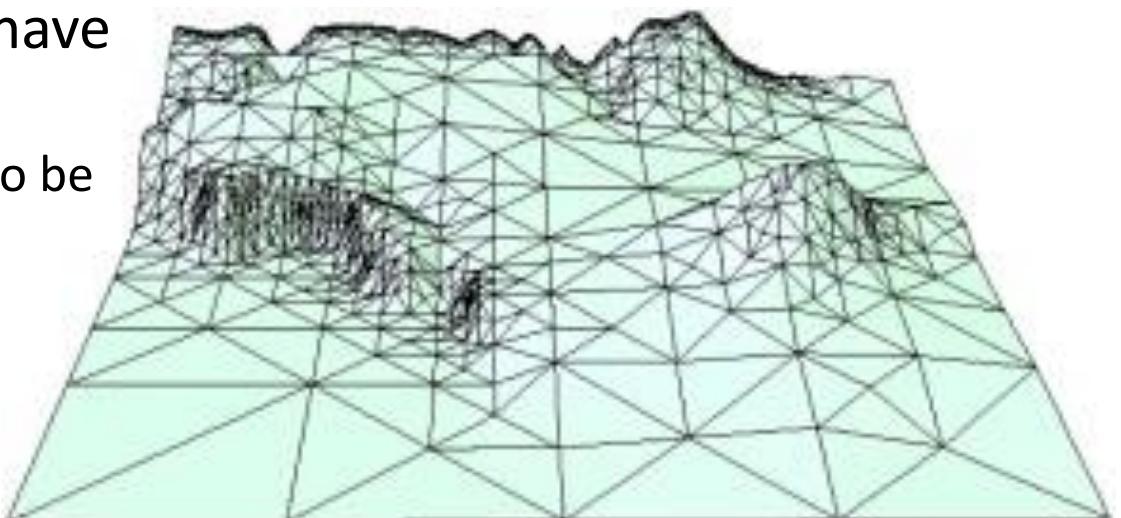
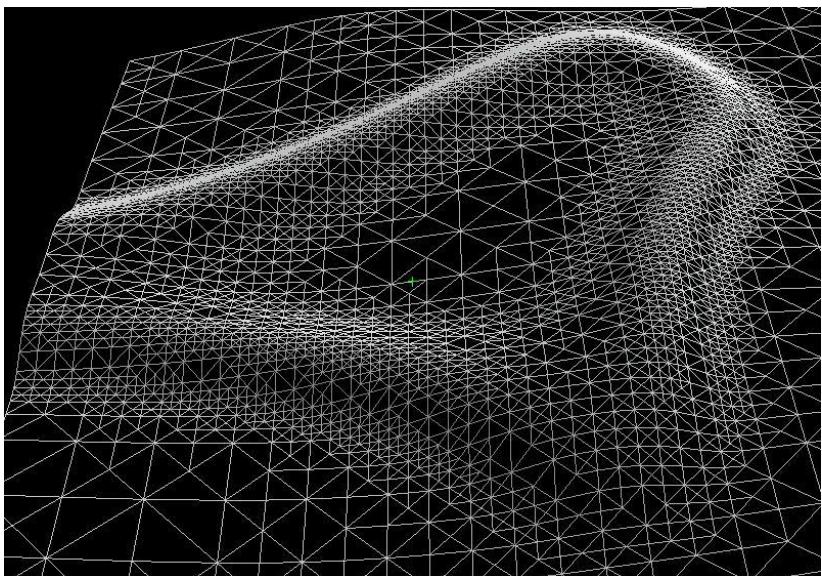
- **Polygon mesh:** a collection of adjacent polygons
- **Adjacent** polygons that share an edge

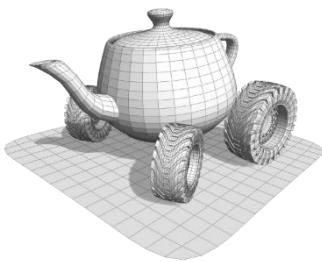




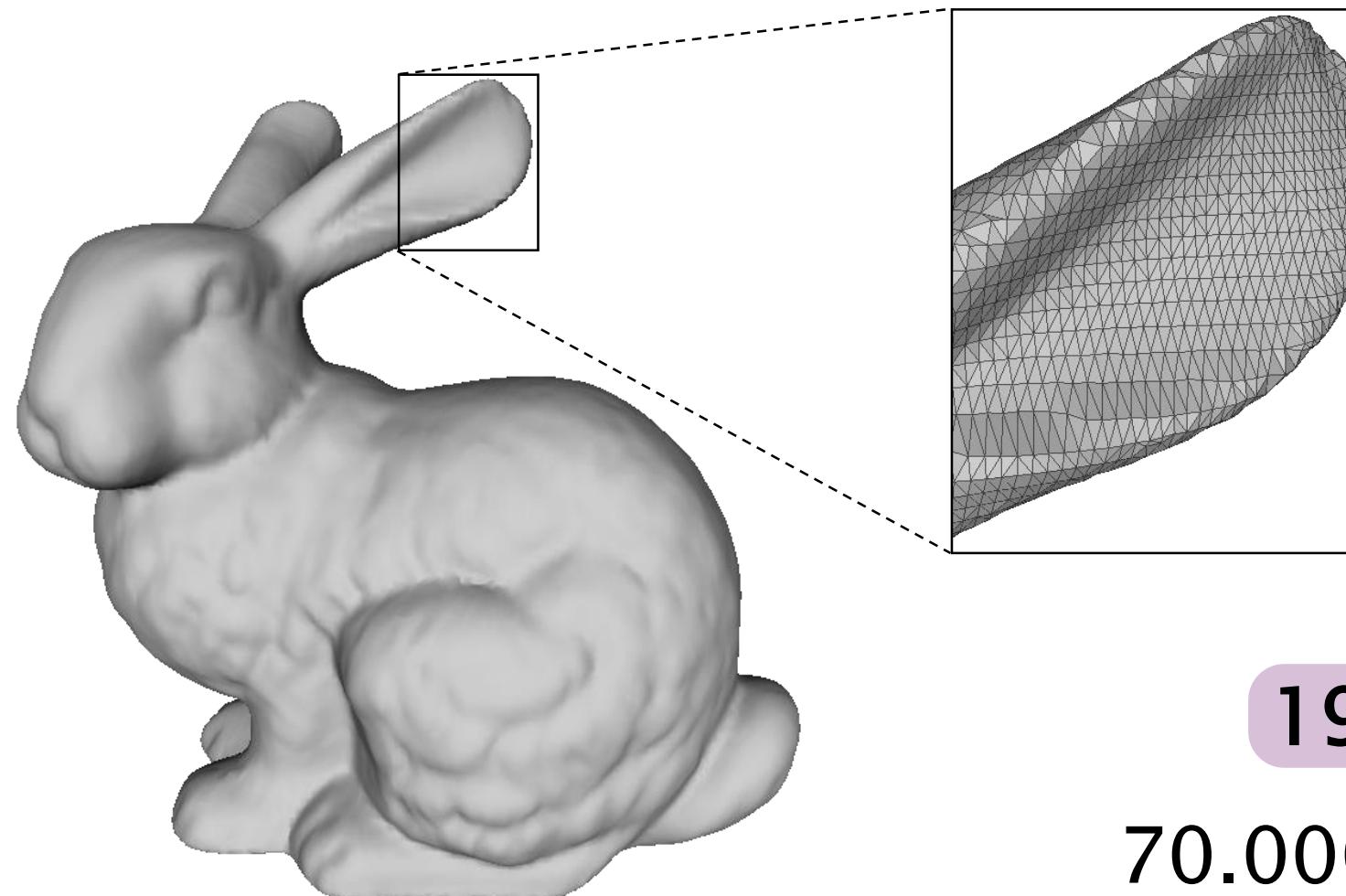
Mesh resolution

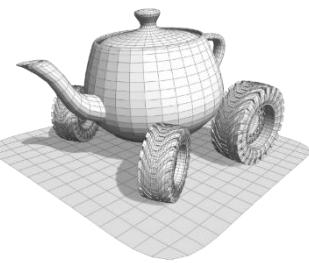
- Number of faces (or vertices) composing the polygon mesh
 - **Hi-res**: more accurate
 - **Low-res** (or low poly) : less accurate but more efficient to render/process
 - Resolution can be **adaptive**, that is, have more polygon where *necessary*
 - Where there is more geometric detail to be represented.



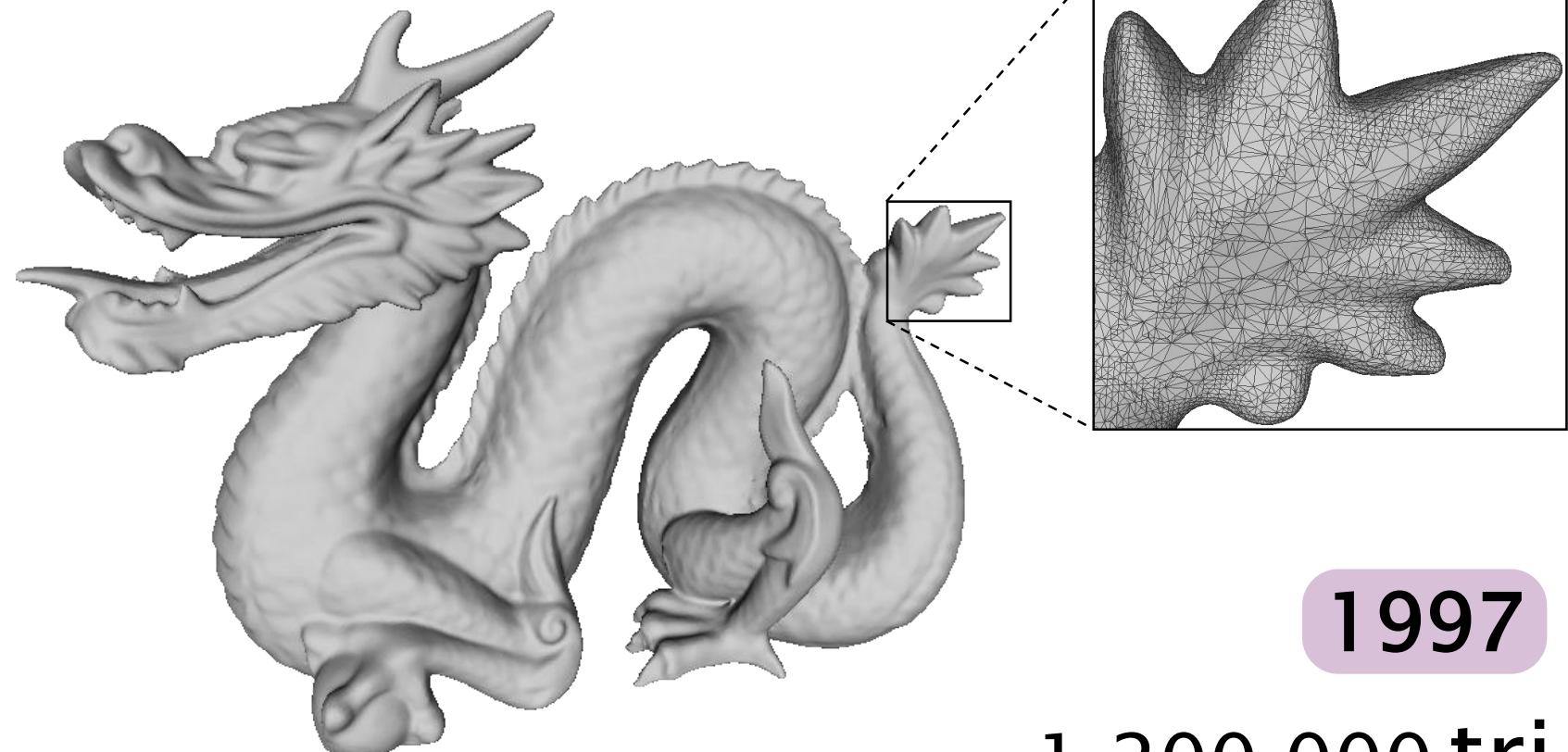


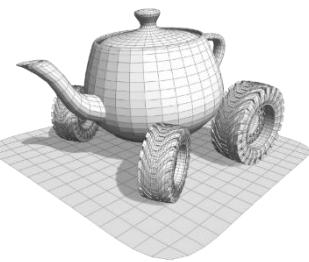
Manageable resolution along the years



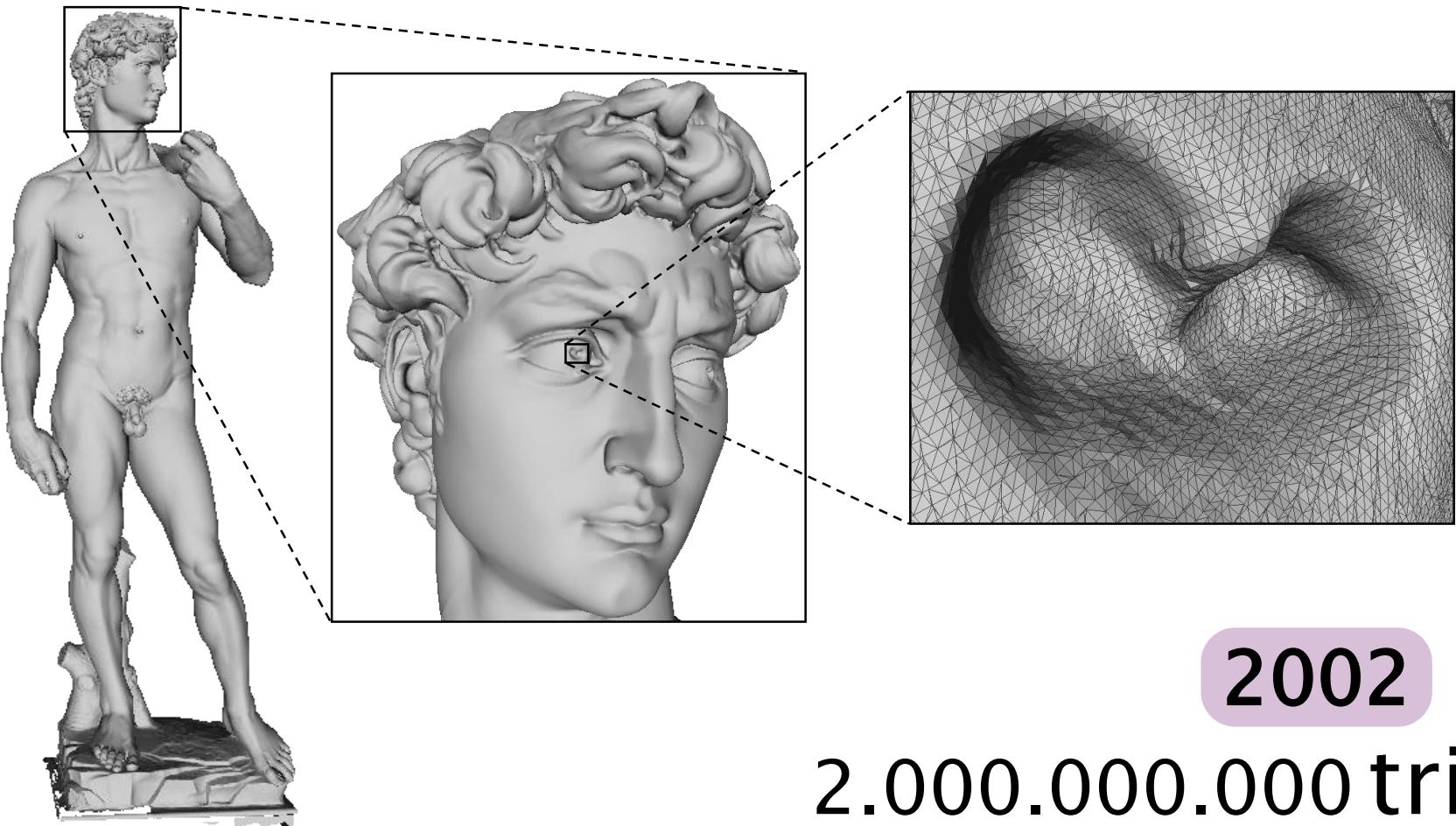


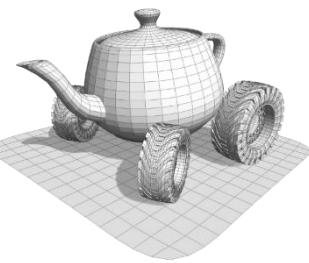
Manageable resolution along the years





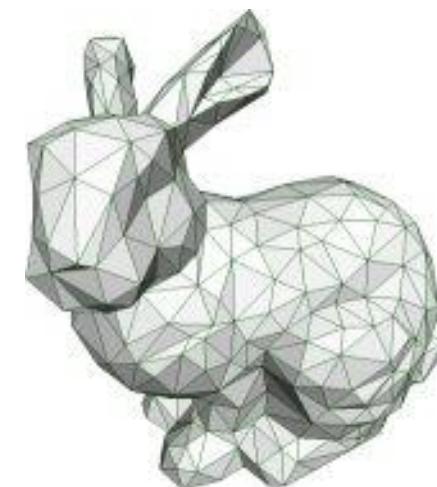
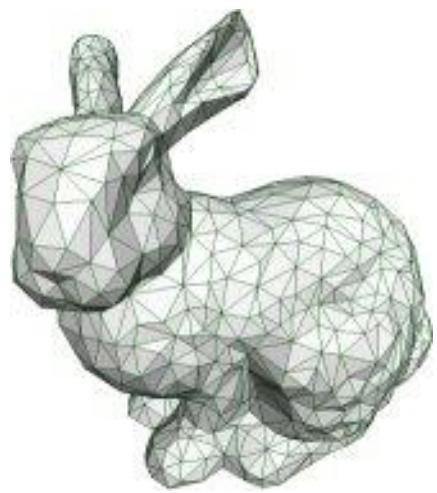
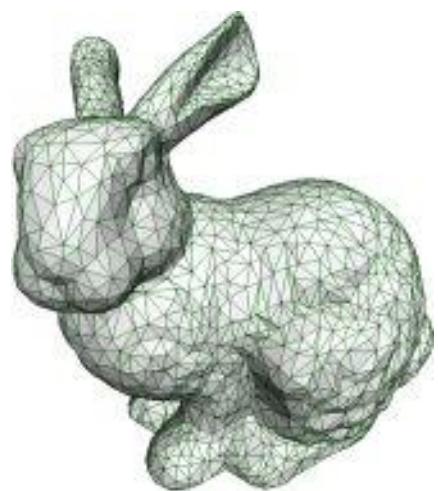
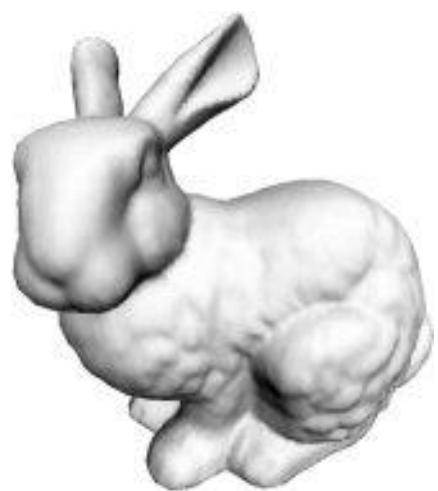
Manageable resolution along the years



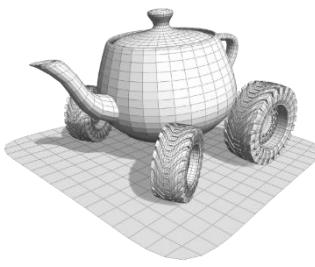


Performance vs quality

performance



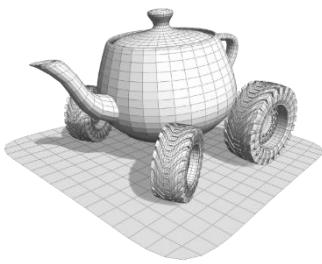
quality



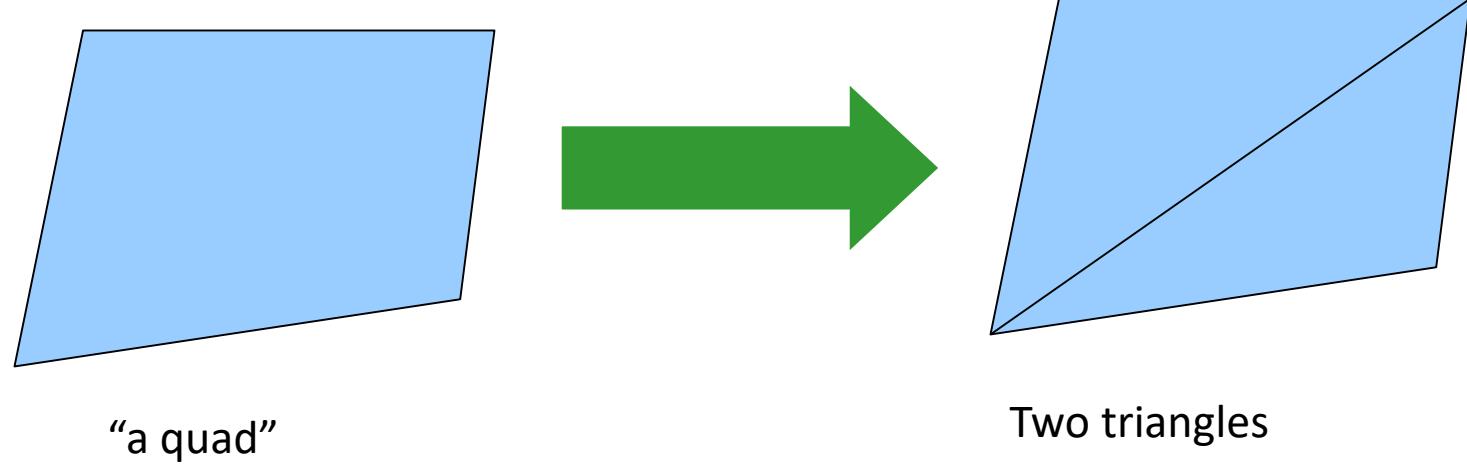
Triangle meshes

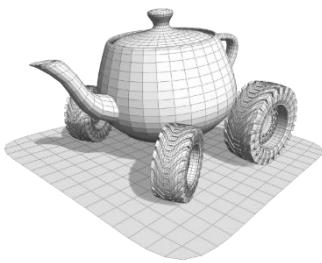
- faces are planar by definition
 - Easy to compute normals to the surface inside a face
 - On the edge and vertices is another story
- Most suitable to render with GPU
 - Actually, the only ones, other polygons are first split into triangles
- Formal name: “Maximal Simplicial Complex of order 2”
 - **Simplex in N dimensions:** the convex hull of $N+1$ points
 - **0-simplex:** point, **1-simplex** segment, **2-simplex** triangle
 - **Complex:** a set of simplices

There is much more, see
the course on Geometry
Processing

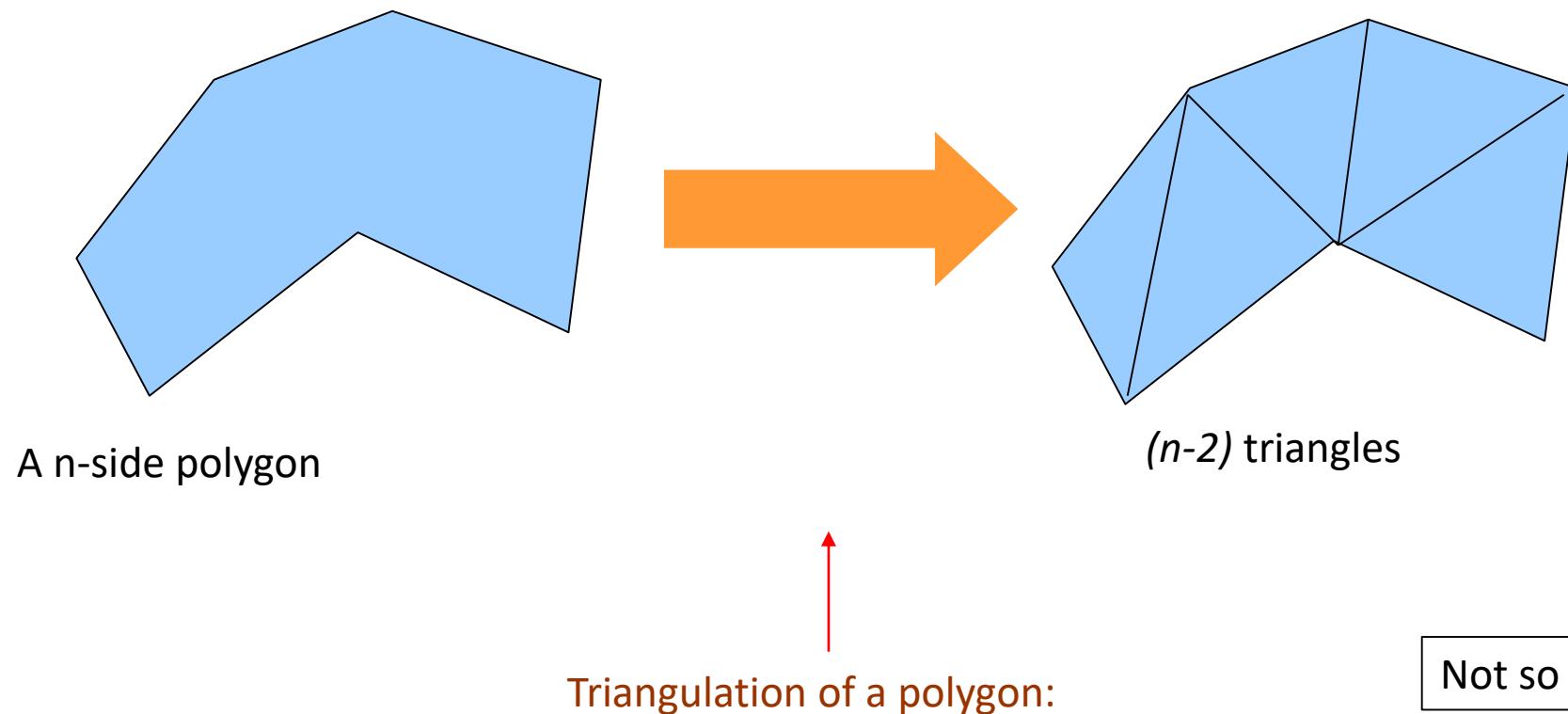


Quad Mesh or Triangles Mesh ?

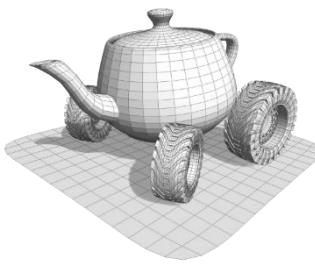




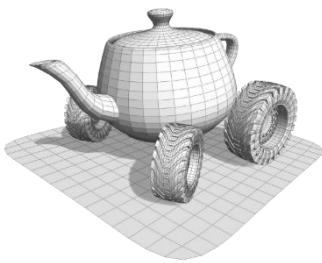
Polygonal meshes or triangle meshes?



Two-manifoldness

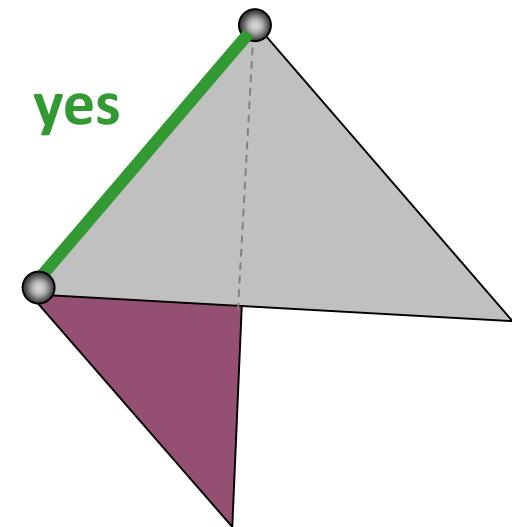
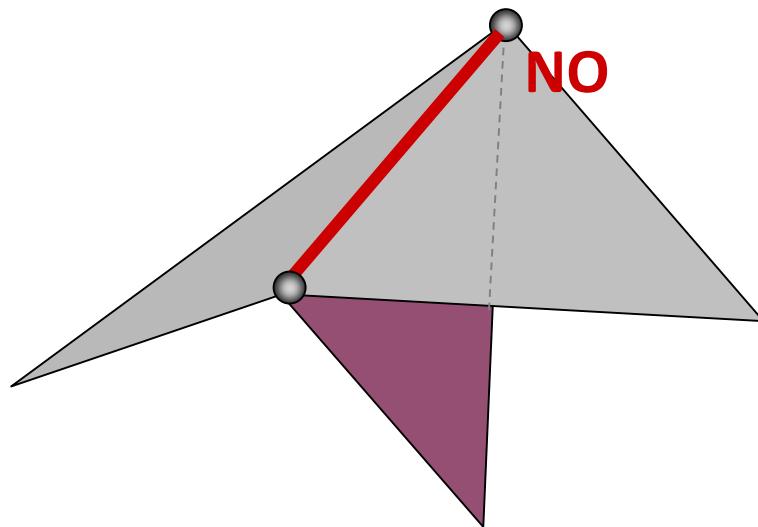


- (informal): a mesh is said to be two-manifold if it *actually* represents a surface
 - Many geometry processing algorithms make this assumption
- (still informal but more precise): a mesh is two-manifold if we can stick a rubber disk on any point on the surface
- Not every mesh is two-manifold
 - Faces intrinsically represent a piece of surface
 - On edges and vertices it may be different
 - How?

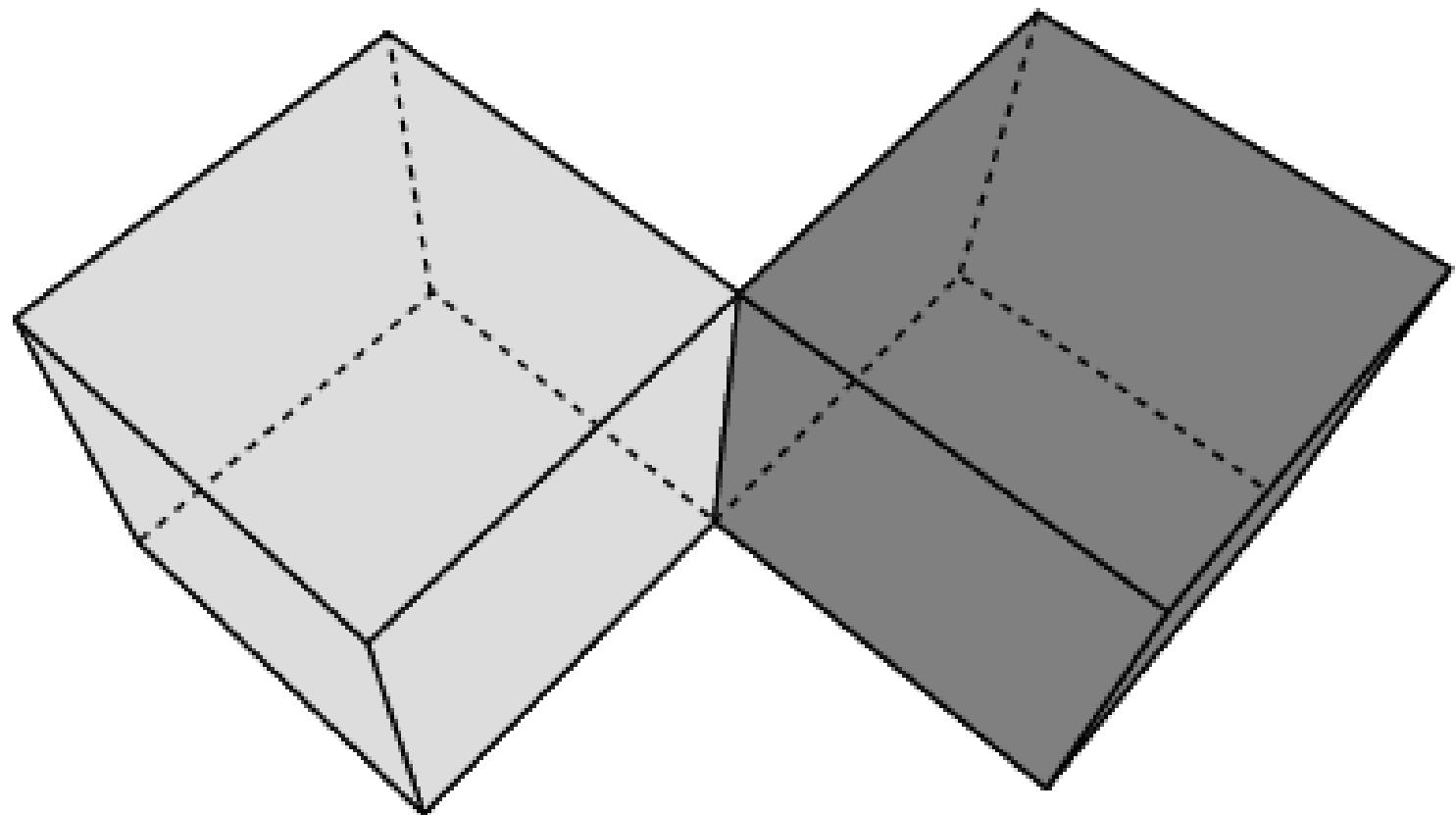
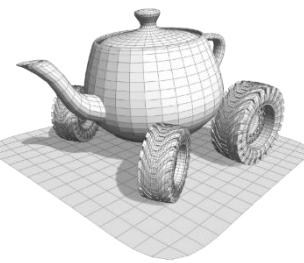


Edge two manifold

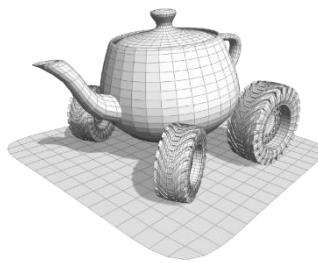
- An internal edge is two-manifold **iff** is shared by two faces



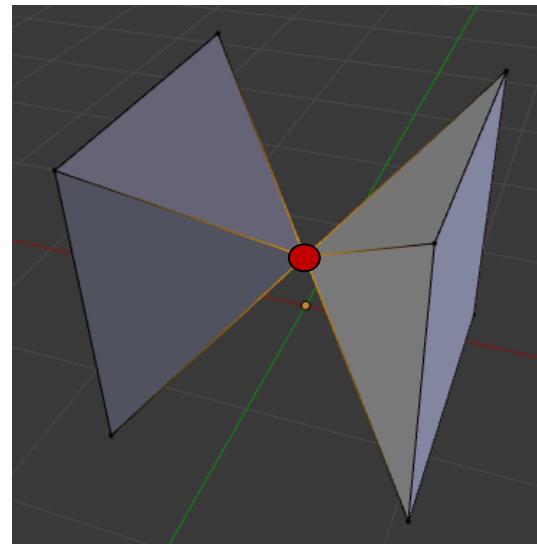
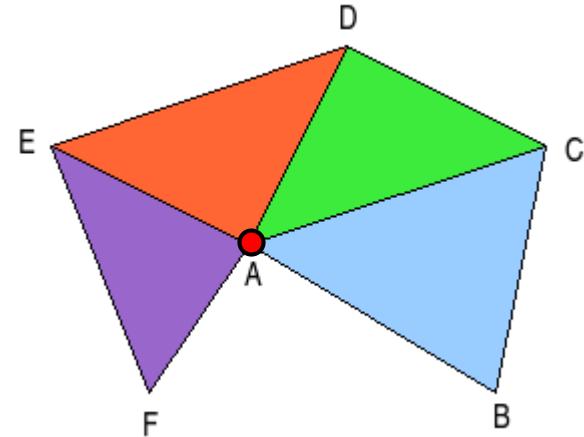
non 2-manifolds edge



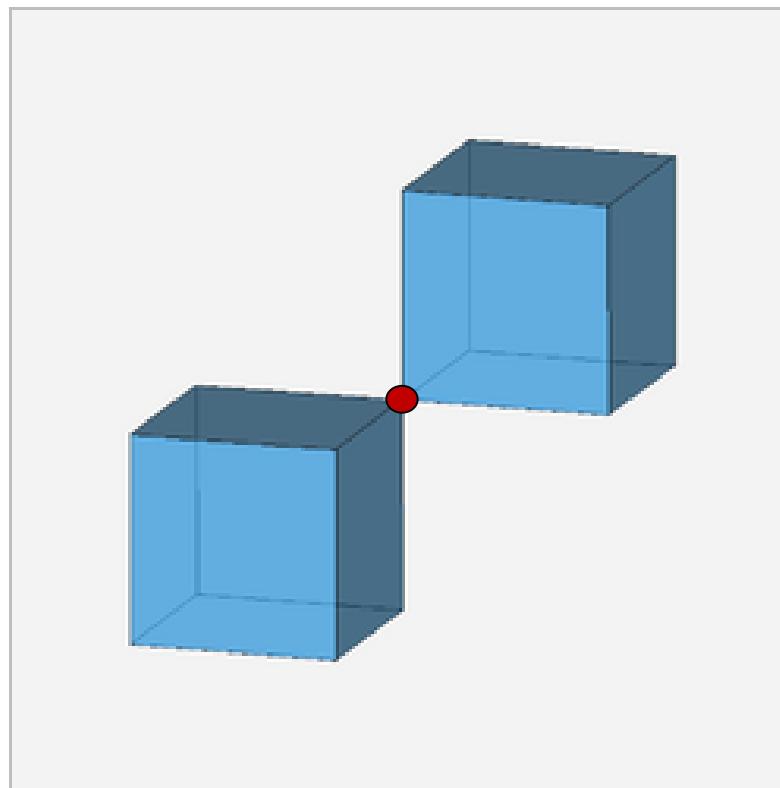
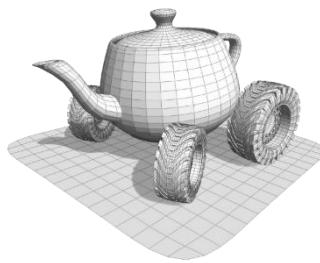
Two-manifold vertices

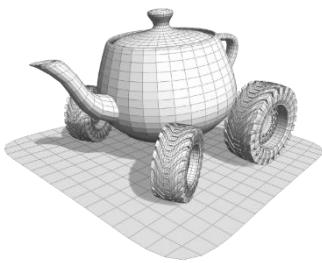


- A **fan** is a set of adjacent faces sharing a vertex
- A two-fold vertex may only be in one fan



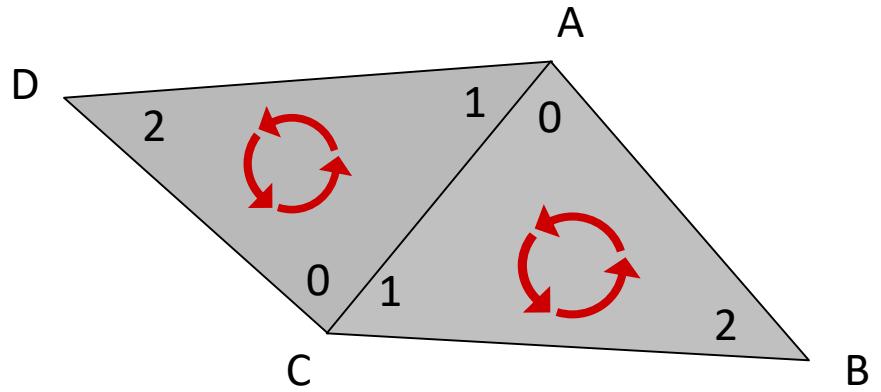
Non 2-manifolds vertex

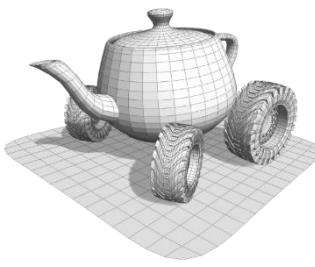




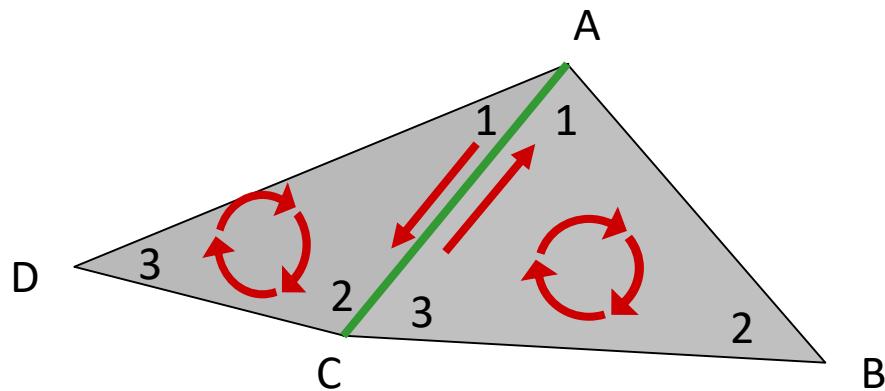
Orientation of a face

- The order on which the vertices are given define the **orientation** of a face
 - On the triangular face: clockwise or counter-clockwise
- a 2-manifold mesh where all adjacent faces have the same orientation is said **oriented**
- if 2-manifold mesh can be oriented by “flipping” a subset of its faces it is said **orientable**

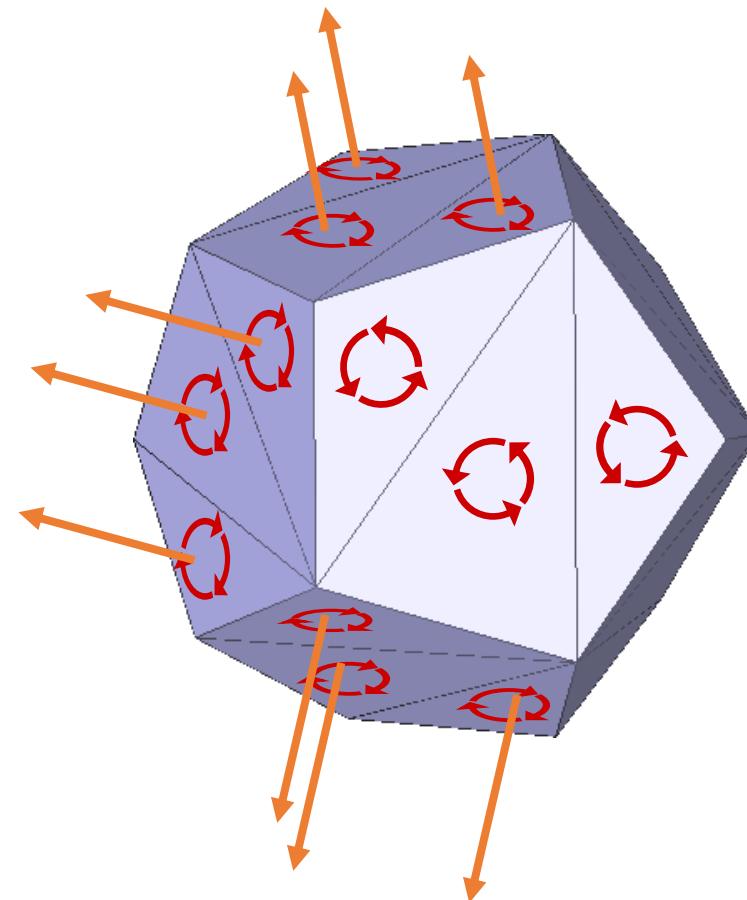




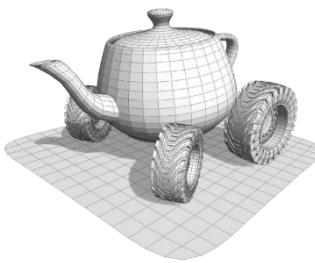
Orientation of a mesh



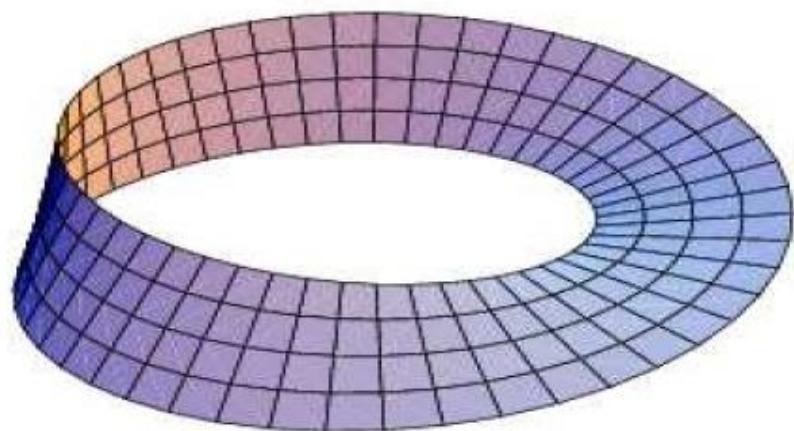
The edge shared by two faces with the same orientation has indices with opposite verse within the faces



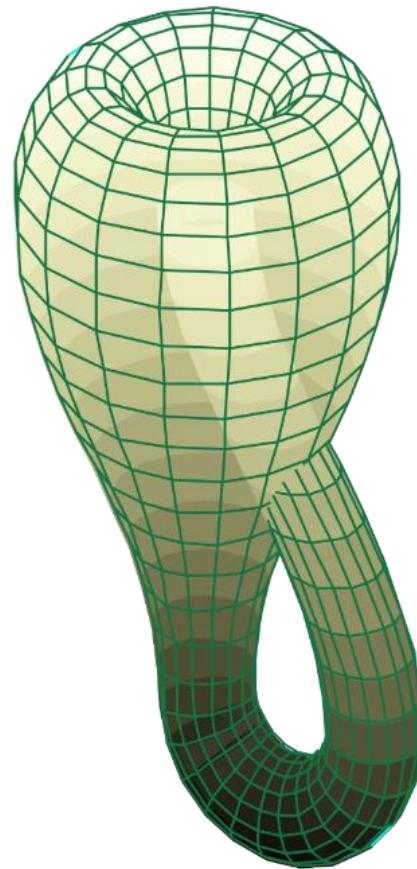
Orientation of a mesh



- Not all meshes can be oriented!
- Moebius strip and Klein Bottle are famous examples
 - They are one-sided surfaces.

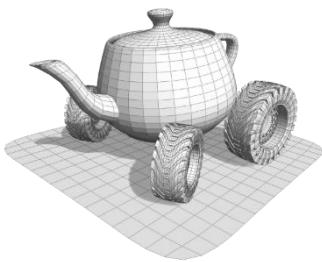


Moebius strip
(non orientable, open)



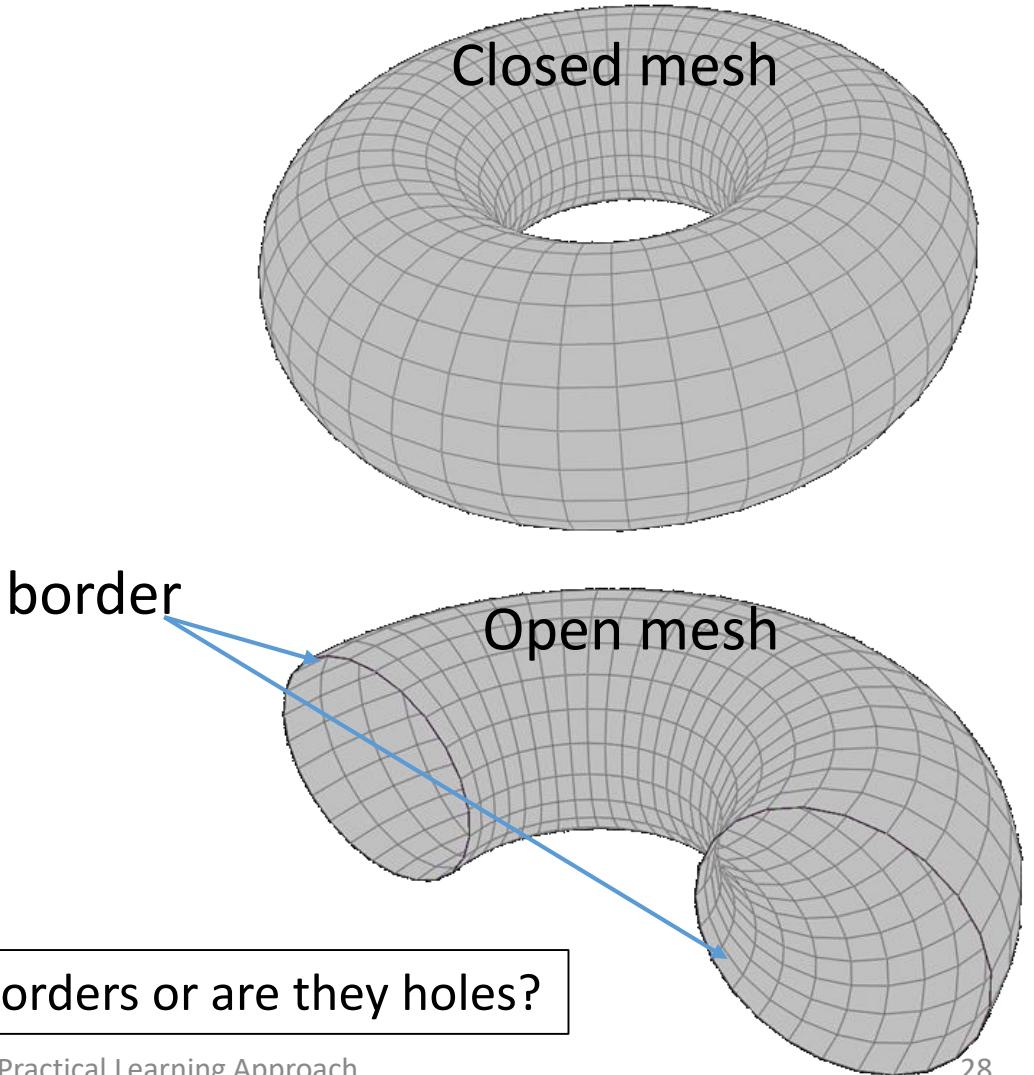
Klein Bottle
(non orientable, closed)

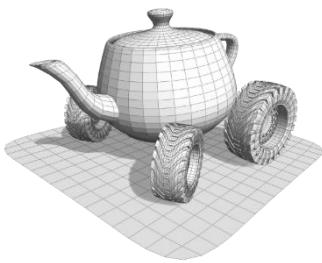
wikipedia



Open Meshes, closed meshes

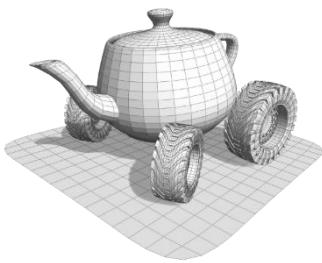
- **Border edge:** an edge not shared by two faces (otherwise is internal)
- **Border vertex:** a vertex of a border edge
- **Closed mesh:** a mesh with **no** border edges
- **Open mesh:** a mesh with border edges





Data structure for meshes (1/3)

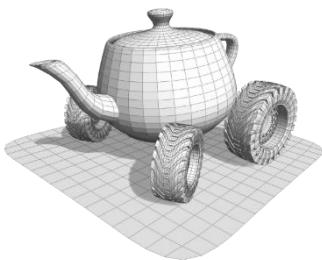
- «Polygon soup»:
an array of N polygons
each polygon is an array of positions (+ other attributes)
- Very simple but not very efficient
every vertex shared by different polygons is duplicated
- Hard to update
changing the position of a vertex implies updating all its copies, which is not doable without further data structure



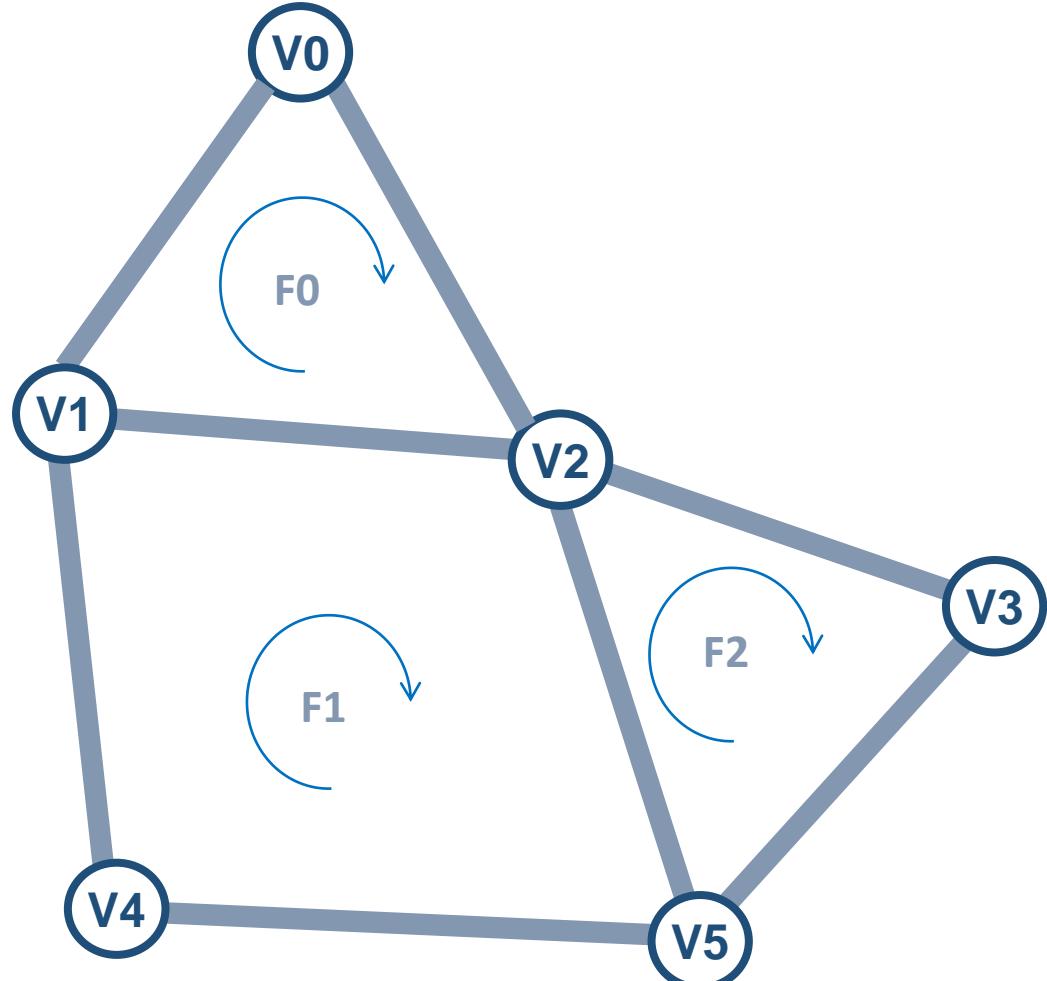
Data structure for meshes (2/3)

- **Indexed mesh:**

- Geometry: array of vertices (position+other attributes)
- **Connectivity** (how the vertices are connected to form primitives)
 - Array of polygons
 - Each polygon is a sequence of reference to a vertex in the geometry array



Data structure for meshes (3/3)



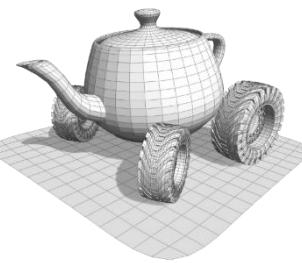
$V_0 \rightarrow$	x_0, y_0, z_0	r_0, g_0, b_0
$V_1 \rightarrow$	x_1, y_1, z_1	r_1, g_1, b_1
$V_2 \rightarrow$	x_2, y_2, z_2	r_2, g_2, b_2
$V_3 \rightarrow$	x_3, y_3, z_3	r_3, g_3, b_3
$V_4 \rightarrow$	x_4, y_4, z_4	r_4, g_4, b_4
$V_5 \rightarrow$	x_5, y_5, z_5	r_5, g_5, b_5

Position and attributes

$F_0 \rightarrow$	0, 2, 1
$F_1 \rightarrow$	1, 2, 5, 4
$F_2 \rightarrow$	2, 3, 5

connectivity

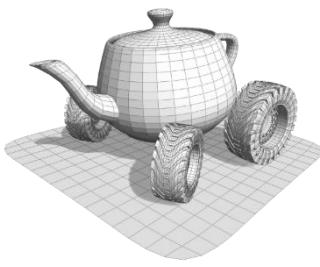
Attributes on meshes



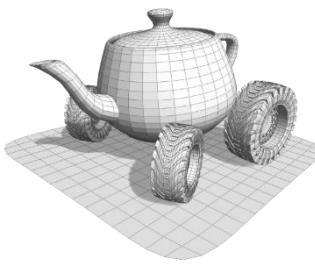
Not a news..
remember our first
triangle?

- Vertex position is not the only useful quantity
- Often we want some “value” (or signal) to be associated with each point of the mesh
 - Color
 - Normal vector to the surface
 - Some quality measure
 - Some function mapping the surface locations to some other space
 - ...

Attributes on meshes: storing



- How are they encoded?
- Per-vertex attribute
 - Most common case
 - How to define the color over **all** the surface?
- Per-face attribute
 - Less used
 - It means they are constant over the face..
 - Discontinuities (not C0) between faces



Barycentric Coordinates (edge)

- **Barycentric Coordinates:** a system to refer a point inside a simplex as a weighted combination of its defining points

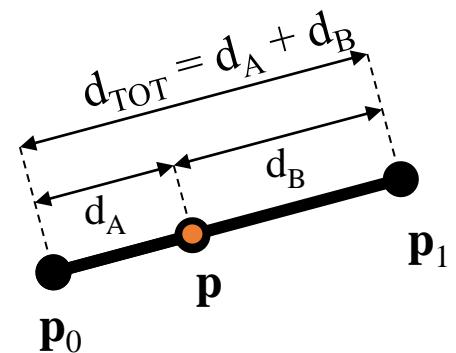
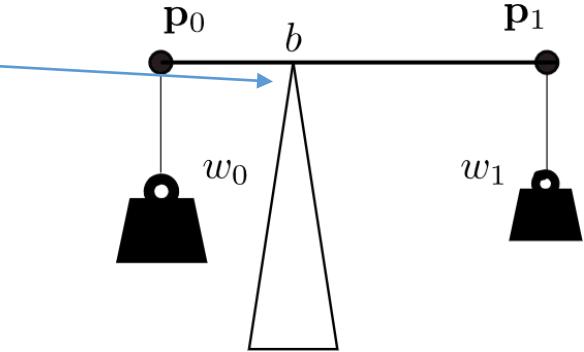
- For the 1-simplex (the edge):

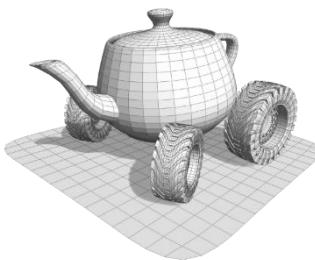
$$\mathbf{p} = w_0 \mathbf{p}_0 + w_1 \mathbf{p}_1$$

$$w_0 + w_1 = 1$$

- How to compute the barycentric coordinates of a given point?

Where is the name from?





Barycentric Coordinates (triangle)

- For the 2-simplex (the face):

$$\mathbf{p} = w_0 \mathbf{p}_0 + w_1 \mathbf{p}_1 + w_2 \mathbf{p}_2$$

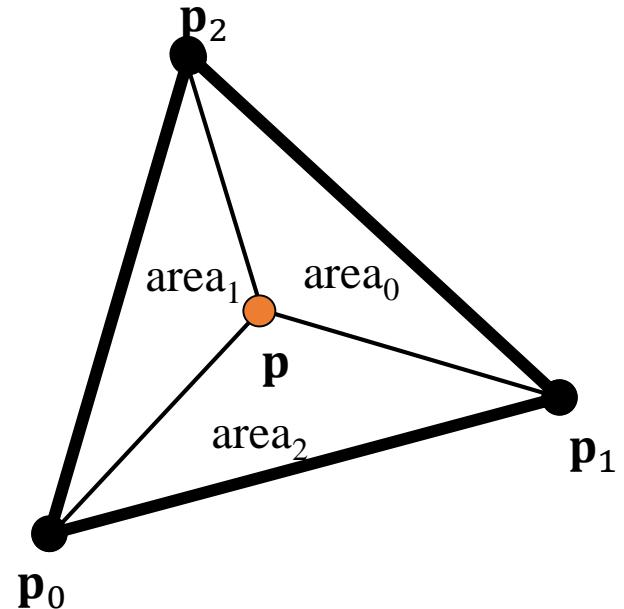
$$w_0 + w_1 + w_2 = 1$$

- How to compute the barycentric coordinates of a given point?

$$w_0 = \frac{\text{area}(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p})}{\text{area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)}$$

$$w_1 = \frac{\text{area}(\mathbf{p}_0, \mathbf{p}, \mathbf{p}_2)}{\text{area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)}$$

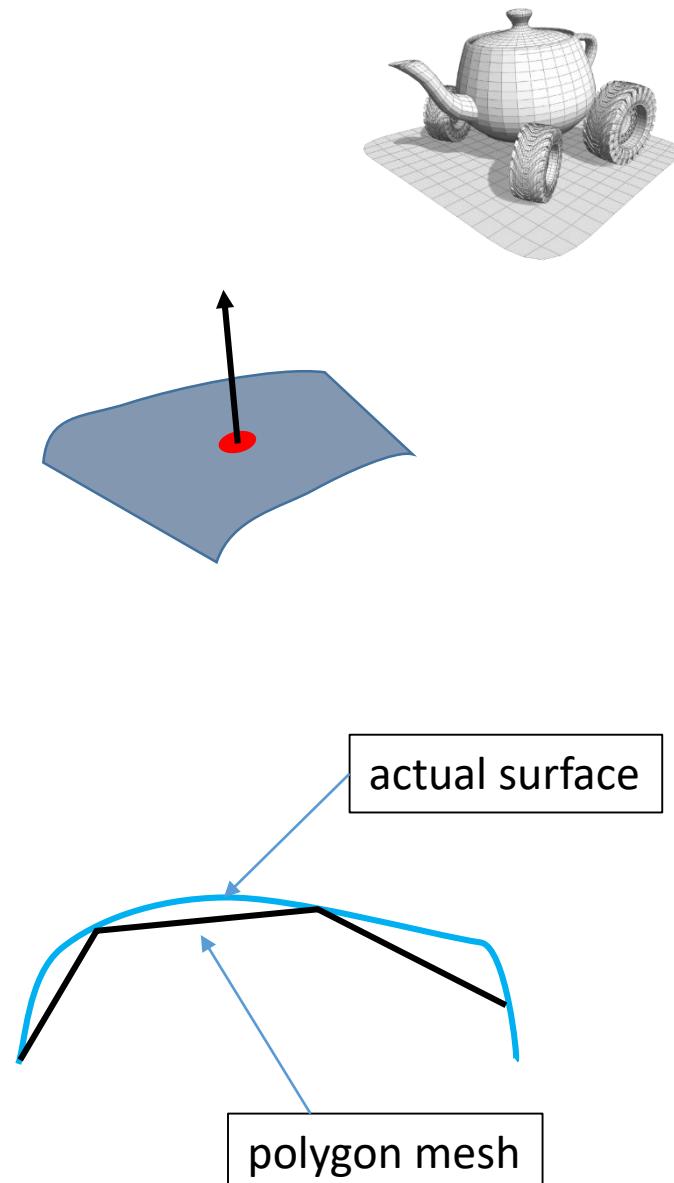
$$w_2 = \frac{\text{area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p})}{\text{area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)}$$



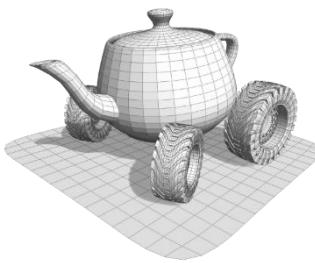
How to compute the
area of a triangle?

Normals of a triangle mesh

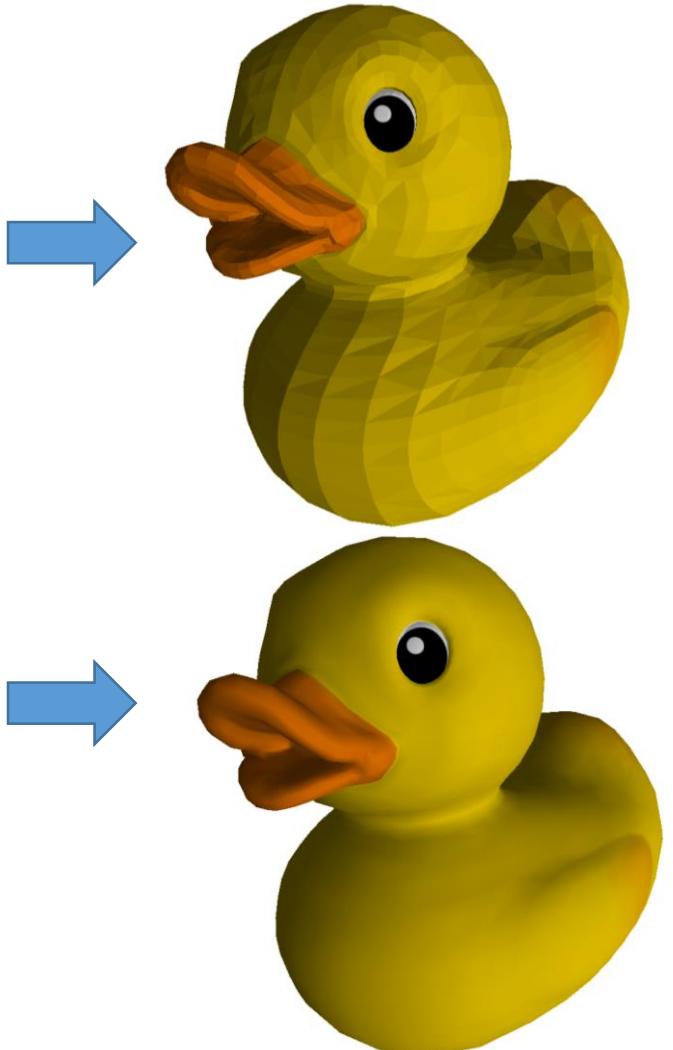
- The normal vector at point \mathbf{p} tells the orientation of the surface in the neighborhood of \mathbf{p}
- Normals to the surface play a fundamental role on the way the surface will look
- What is the normal of a polygon mesh, really?
It's the normal to the actual *continuous* surface we are approximating with the mesh
...Only, most of the time, we only have the tessellation

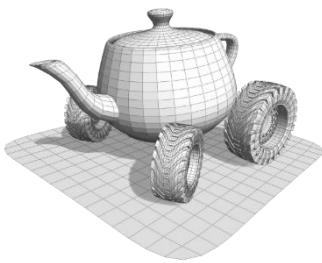


Computing the normals of a triangle mesh



- Triangle faces are flat, with constant normal.
We can compute the normal per-face and use it as normal of all the point in the face
 - This will emphasize the tessellation, and most of the times it's not what we want
- If we evaluate the normal per-vertex and then interpolate its value inside the face, the tessellation will be less visible and the look will be more smooth
 - Is it always what we want?



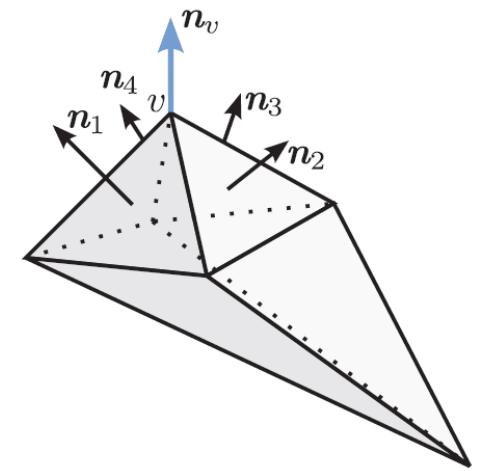


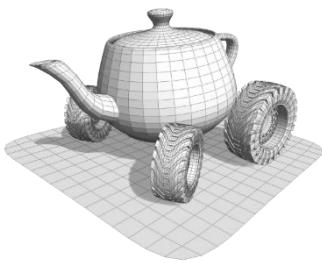
Normal per-vertex

- Many possible definitions, e.g.
 - The average among all the normals of the faces sharing the vertex

$$n_v = \frac{1}{|S^*(v)|} \sum_{i \in S^*(v)} n_{f_i}$$

Faces sharing vertex i





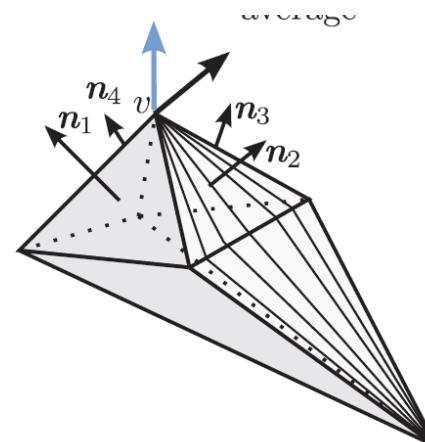
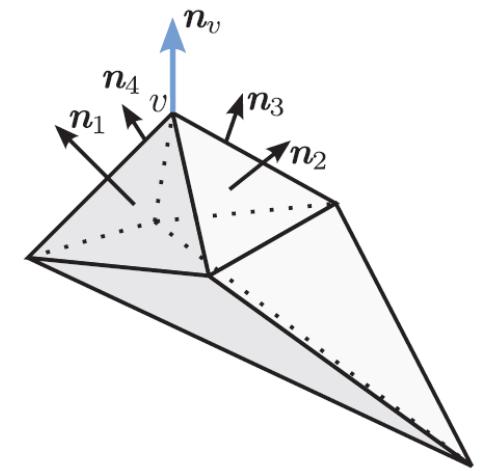
Normal per-vertex

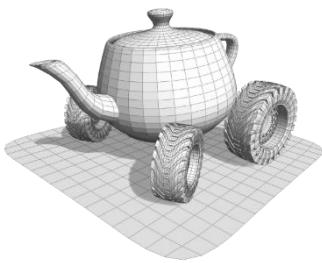
- Many possible definition, e.g.
 - The average among all the normal of the faces sharing the vertex

$$n_v = \frac{1}{|S^*(v)|} \sum_{i \in S^*(v)} n_{f_i}$$

widely used, it assumes that the tessellation is fairly regular.

Many thin faces contribute more than one large face..





Normal per-vertex

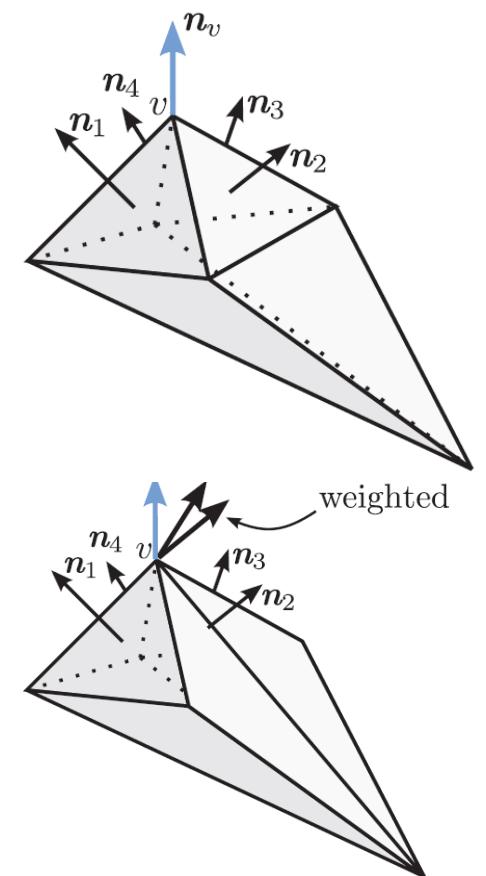
- Many possible definition, e.g.
 - The average among all the normal of the faces sharing the vertex

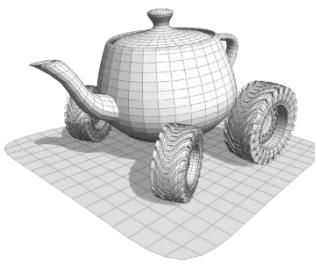
$$\mathbf{n}_v = \frac{1}{|S^*(v)|} \sum_{i \in S^*(v)} \mathbf{n}_{f_i}$$

- Angle weighted average among all the normal of the faces sharing the vertex

$$\mathbf{n}_v = \frac{1}{\sum_{i \in S^*(v)} \alpha(f_i, v)} \sum_{i \in S^*(v)} \alpha(f_i, v) \mathbf{n}_{f_i}$$

↑
 Faces sharing vertex i
 ↑
 Angle of face f_i on vertex v



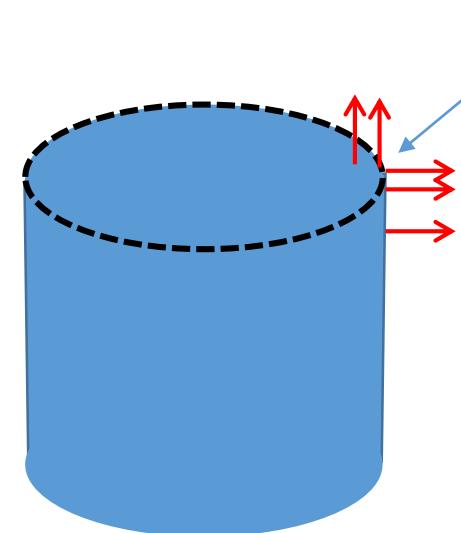


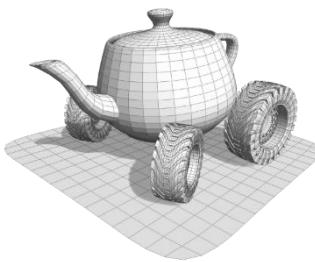
Normal per-vertex: always a good idea?

- If the *actual* shape we are trying to represent is not smooth, per-vertex normal it's a problem
- Assigning the normal per-vertex assumes the surfaces is approximated by a plane around that vertex.

This is not true in the «corners»

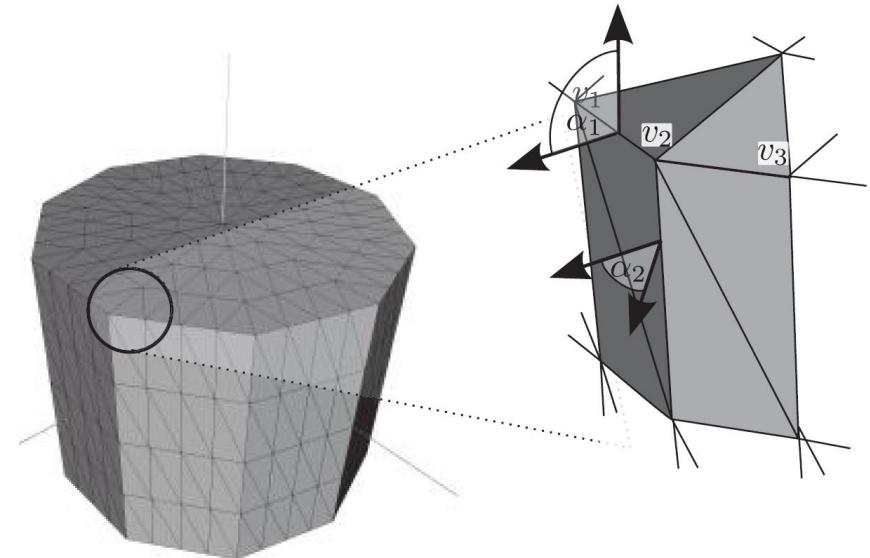
Here the normal changes discontinuously





Normal per-vertex: crease angle

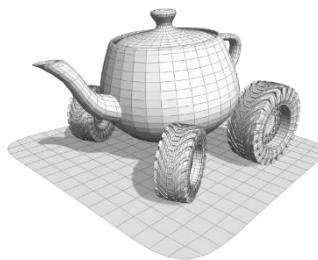
- Decide when we need more than one normal for a vertex
- when the faces sharing a vertex form an angle greater than a predefined **crease angle** we decide that the surface is not smooth in that vertex. So the vertex needs more than one normal...



$$\alpha_{cr} = \frac{\pi}{2}$$

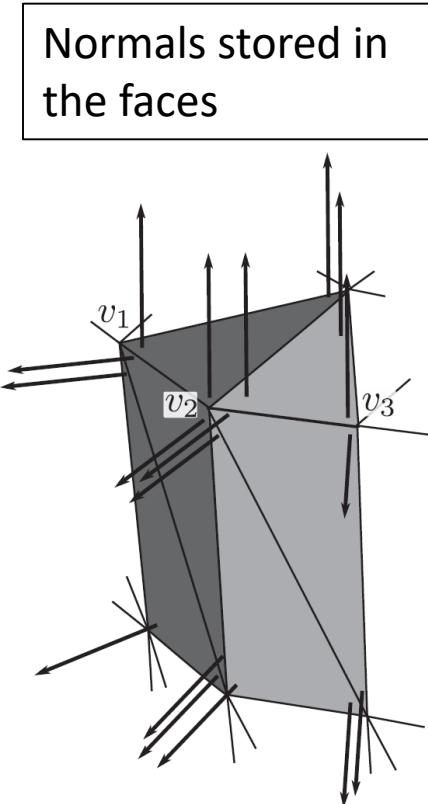
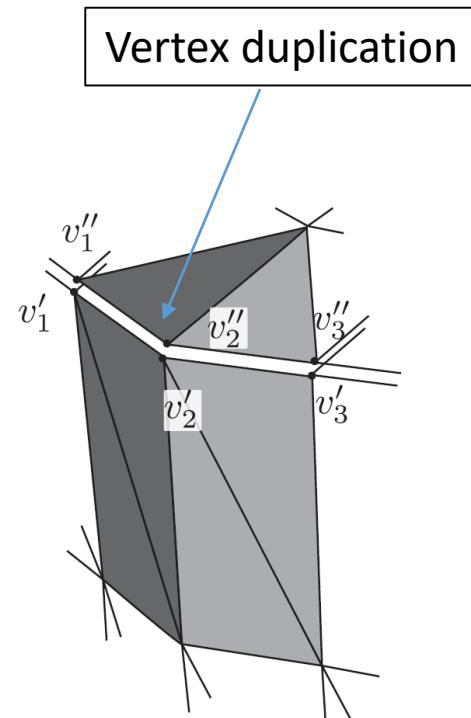
$$\alpha_1 = \frac{\pi}{2} > \alpha_{cr} \rightarrow \text{crease}$$

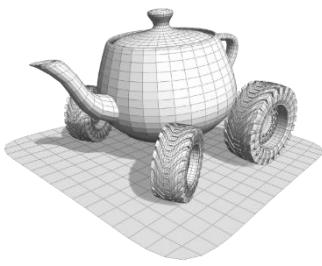
$$\alpha_2 = \frac{\pi}{5} < \alpha_{cr} \rightarrow \text{smooth}$$



Normal per-vertex: crease angle

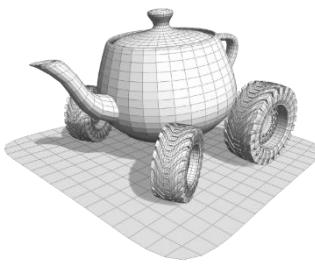
- Decide when we need more than one normal for a vertex
- when the faces sharing a vertex form an angle greater than a predefined **crease angle** we decide that the surface is not smooth in that vertex.
So the vertex needs more than one normal...
- We can either duplicate the vertex or store the normal attribute on the faces





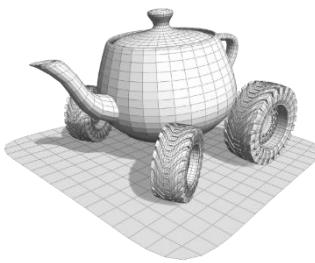
Wrapping up

- If mesh is two-manifold, oriented and closed then it separates the inside and outside space
 - *That is, it's the boundary of a volume*
- Hence:
 - It represents a **solid** object
 - We can say if a point is external or internal
 - We can compute if volume and area
 - Its barycenter (assuming constant density)
 - We can convert it to a 3D volumetric model



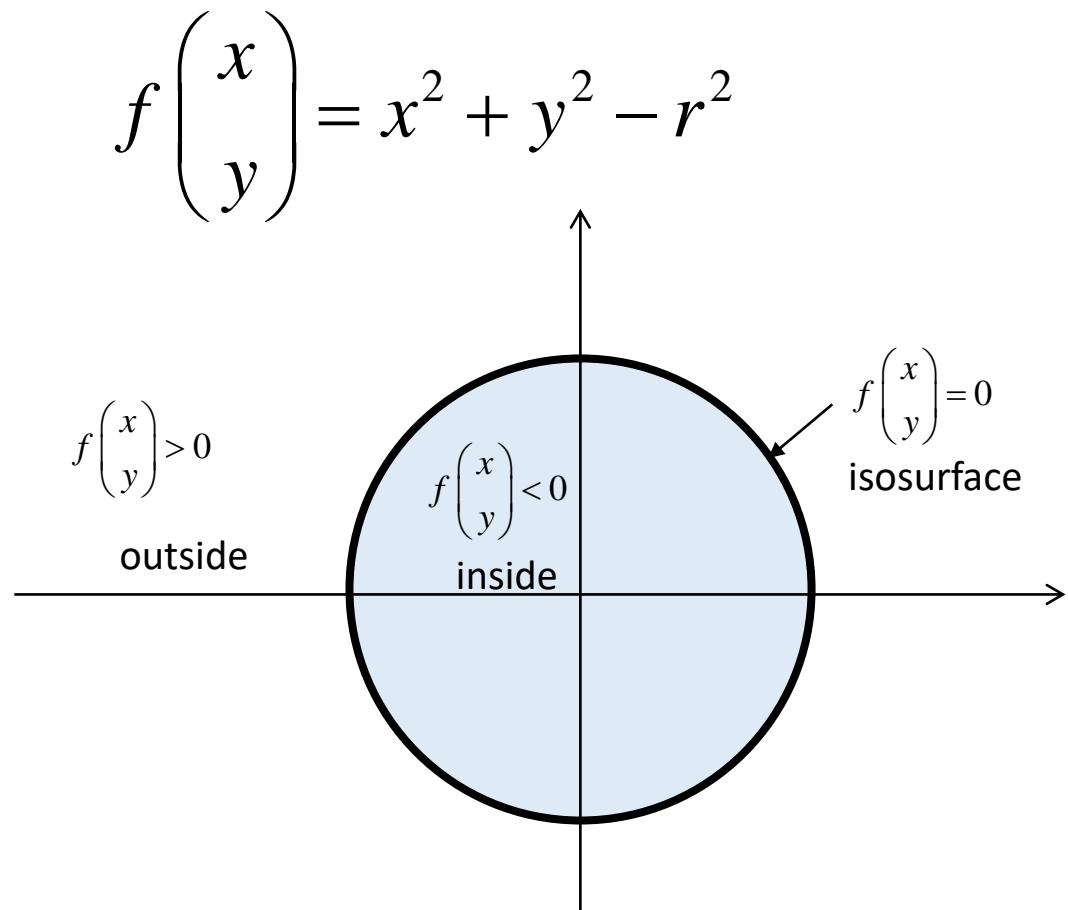
Types of 3D digital models

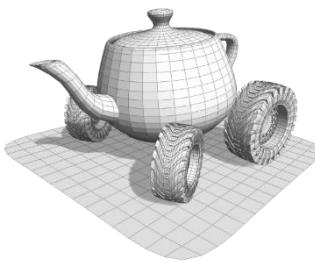
- Surfaces
 - Implicit surfaces
 - Parametric surfaces
 - Polygon meshes
- Volumetric
 - Voxellization
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes
- Points



Implicit Surfaces

- Is the zero set of a function $f: R^3 \rightarrow R$
- That is: $S = \{(x, y, z): f(x, y, z) = 0\}$
- “Implicitly” separates *inside* from *outside*:
 - $f(\mathbf{p}) < 0 \iff \mathbf{p}$ inside
 - $f(\mathbf{p}) > 0 \iff \mathbf{p}$ outside
 - $f(\mathbf{p}) = 0 \iff \mathbf{p}$ on the surface

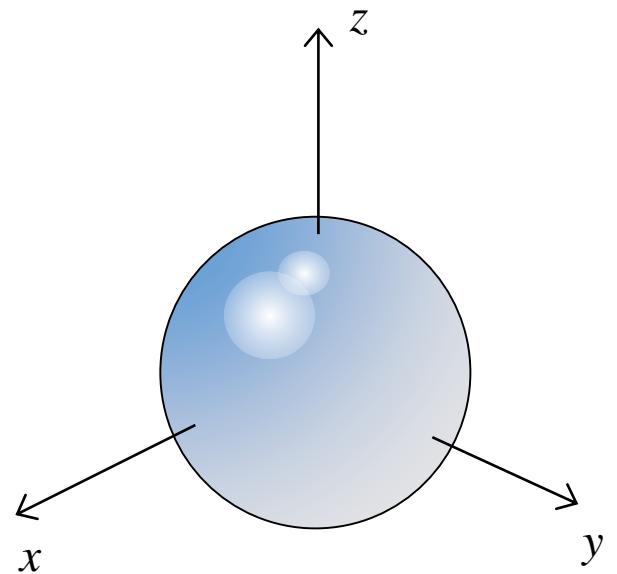


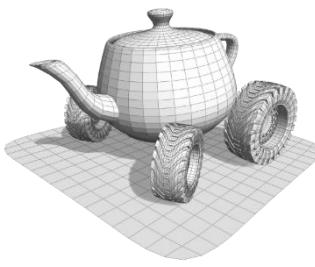


Implicit surface: a 3D example

- It is also a **volumetric representation**
- Easy to know if a point is inside or outside
- It is a compact representation
- It is easy to **combine**, hence we can create more complex surface/volumes

$$f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = x^2 + y^2 + z^2 - r^2$$

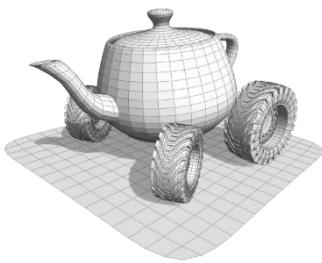




Normal of an implicit model

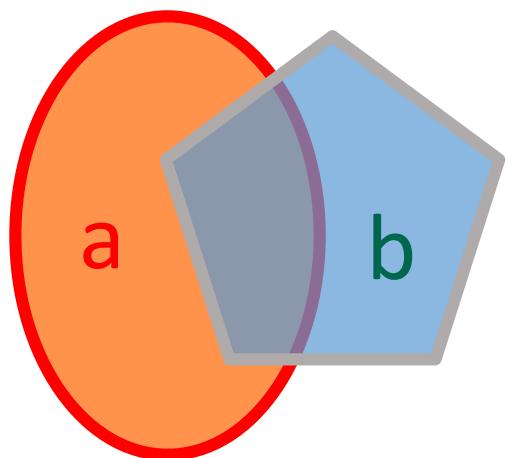
- Let \mathbf{p} be a point on the surface: $f(\mathbf{p}) = 0$
- What is the normal of the surface at \mathbf{p} ?
 - That is: toward which direction \mathbf{d} do we have to move away from \mathbf{p} to maximize the value of $f(\mathbf{p} + k \mathbf{d})$
- Answer: take the gradient of $f(\mathbf{p})$

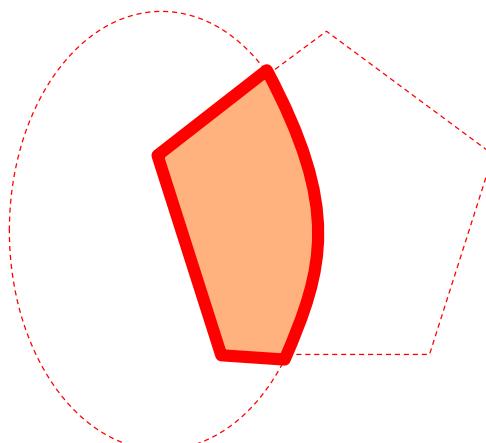
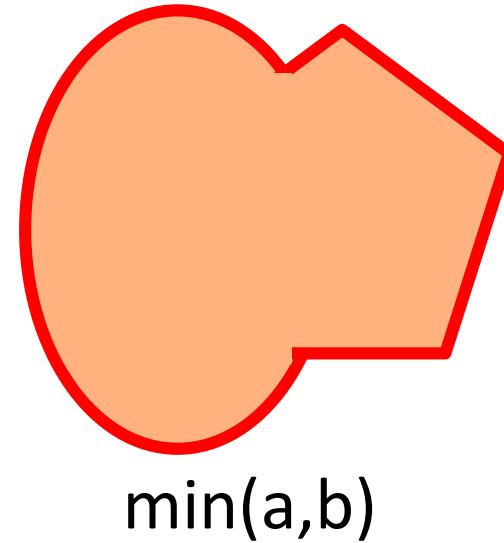
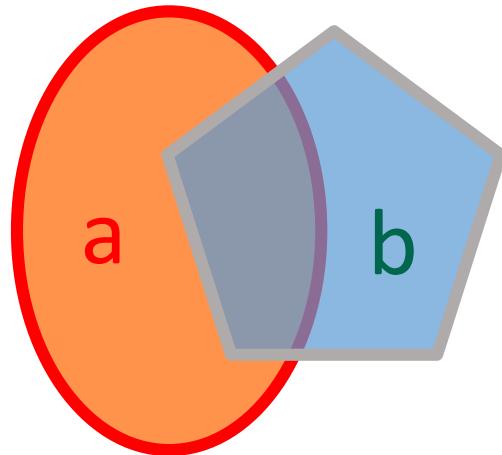
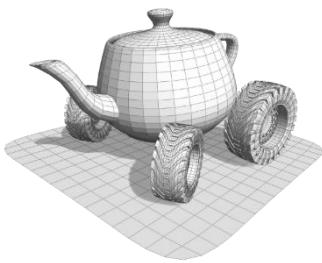
$$normal(f) = \frac{\nabla(f(\mathbf{p}))}{\|\nabla(f(\mathbf{p}))\|} = \begin{bmatrix} \frac{d}{dx}f(\mathbf{p}) \\ \frac{d}{dy}f(\mathbf{p}) \\ \frac{d}{dz}f(\mathbf{p}) \end{bmatrix}$$



Boolean operations

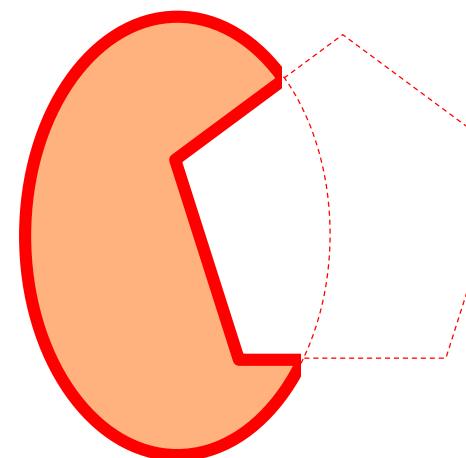
- Let A and B be two solid objects described as the volume enclosed by implicit surfaces f_A and f_B
- We can define the following operators
 - complement: $-f_A$
 - intersection: $\max(f_A, f_B)$
 - union: $\min(f_A, f_B)$
 - difference: $\max(f_A, -f_B)$



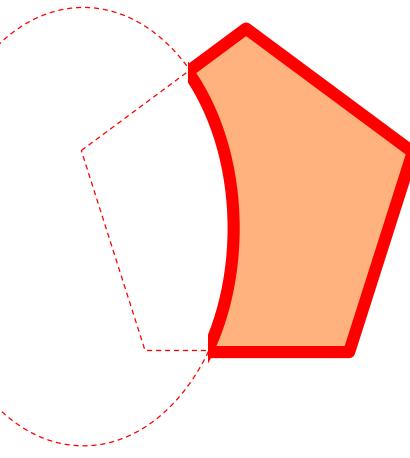


Chapter 3: How a 3D Model is Represented

$\max(a, b)$

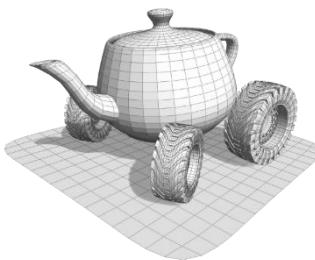


$\max(a, -b)$

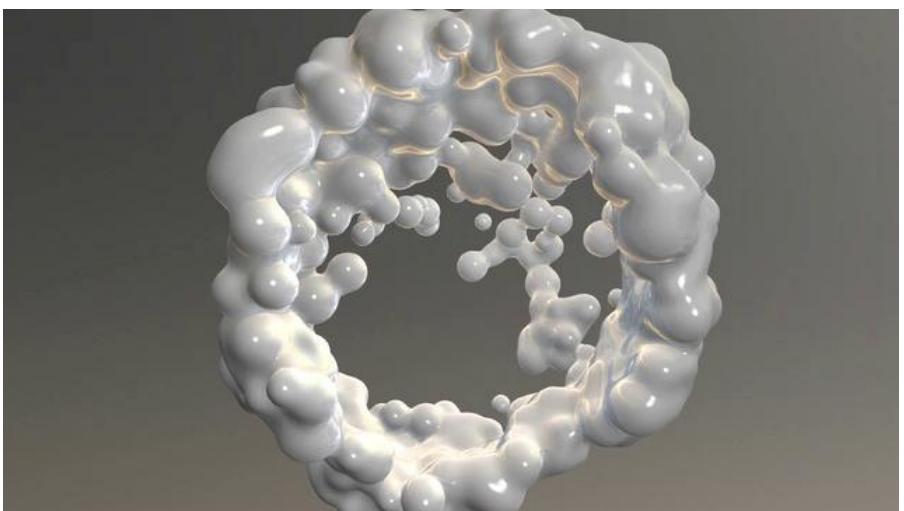
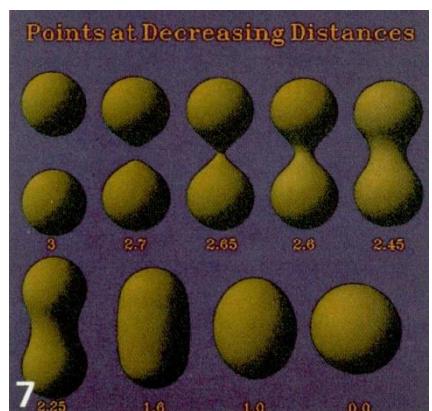


$\max(-a, b)$

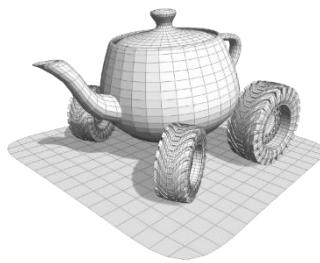
Implicit Surfaces: metaballs



- Idea: the surface is a union of *blobby* spheres
 - When the spheres are well apart, they are just sphere
 - When they get closer they tend to join and form a single surface



Implicit Surfaces: metaballs

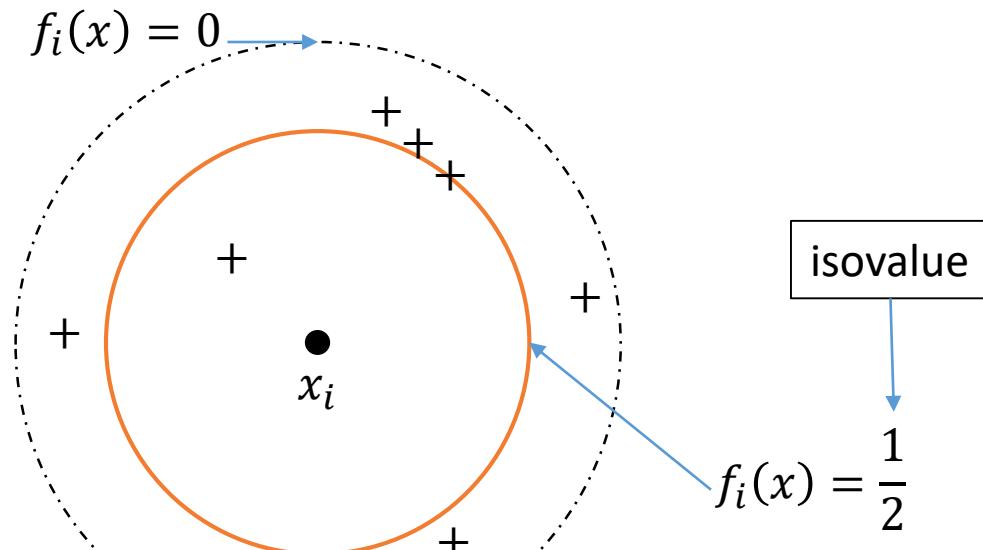


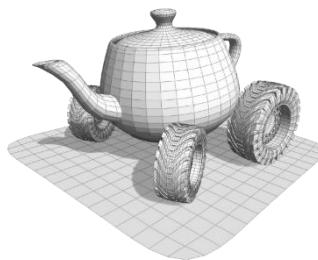
- Idea: define $f(x)$ as a sum of simple functions:

$$f(x) = \sum_i f_i(x)$$

where each function $f_i(x)$ depends only on the distance from a 3D point x_i

$$f_i(x) = \begin{cases} 1 - \frac{r^2}{R^2} & r < R \\ 0 & r \geq R \end{cases}$$





Implicit Surfaces: metaballs

x_i : center

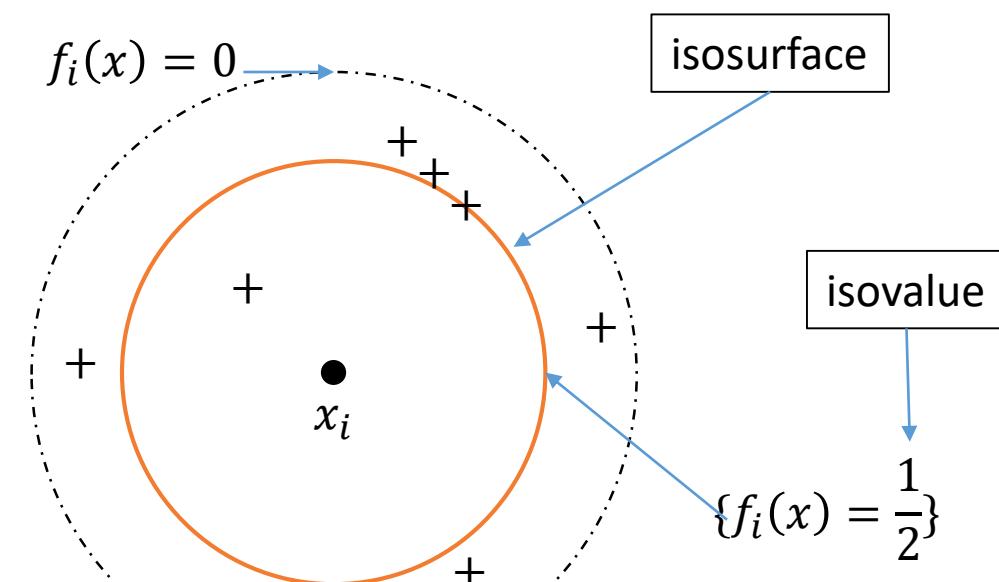
R : Support radius. The influence of $f_i(x)$ is 0 outside

Isovalue = some value between 0 and 1 that defines the isosurface. Often referred to as α

$$f_i(x) = \begin{cases} 1 - \frac{r^2}{R^2} & r < R \\ 0 & r \geq R \end{cases}$$

support

$$r = \|x - x_i\|$$



Functions to use

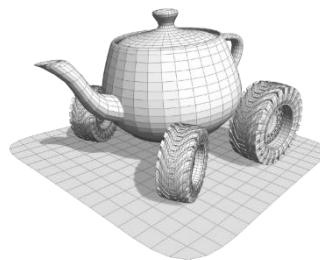
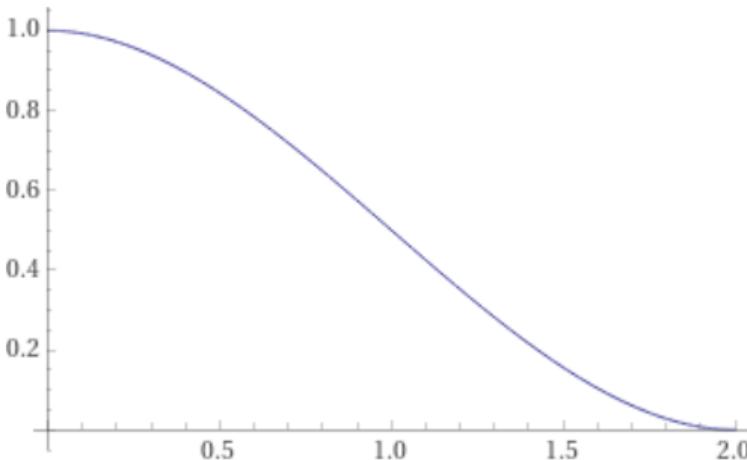
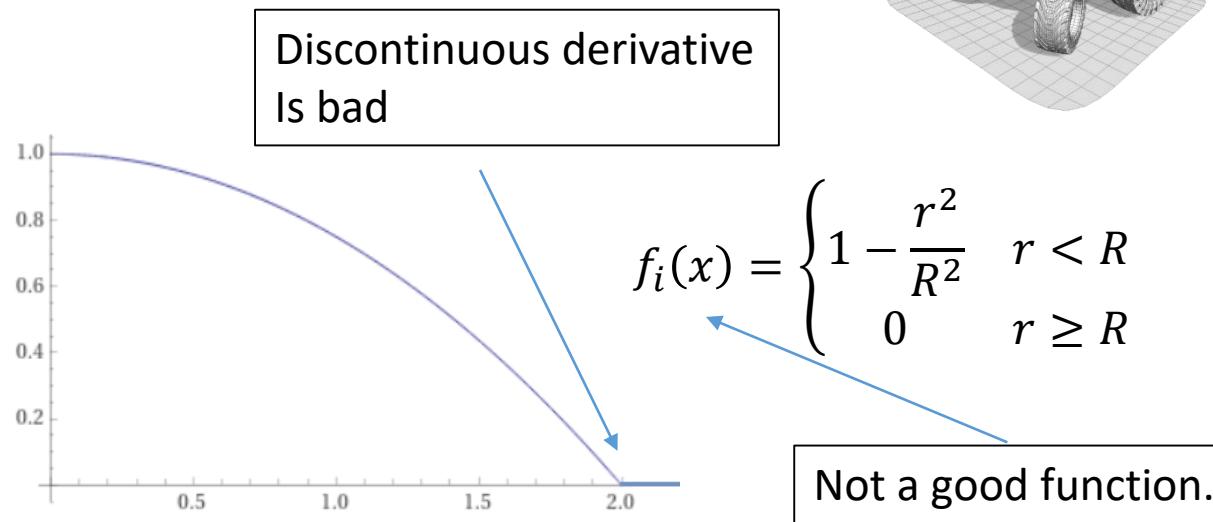
- Function f should
 - Gently decay
 - Approach 0 with null derivative

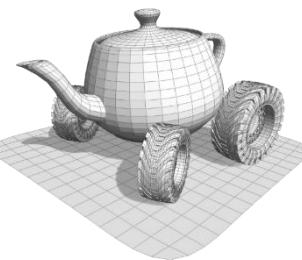
$$f(x_i) = 1 \quad f(R) = 0$$

$$f'(x_i) = 0 \quad f'(R) = 0$$

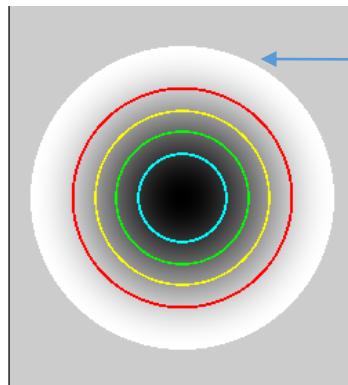
- For example:

$$f_i(x) = \begin{cases} 2\frac{r^3}{R^3} - 3\frac{r^2}{R^2} + 1 & r < R \\ 0 & r \geq R \end{cases}$$





Implicit Surfaces: metaballs



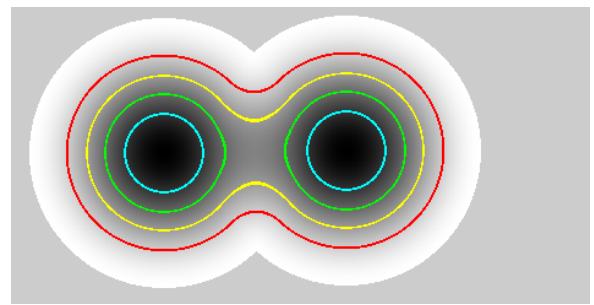
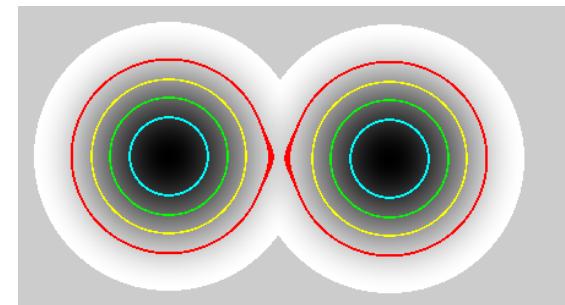
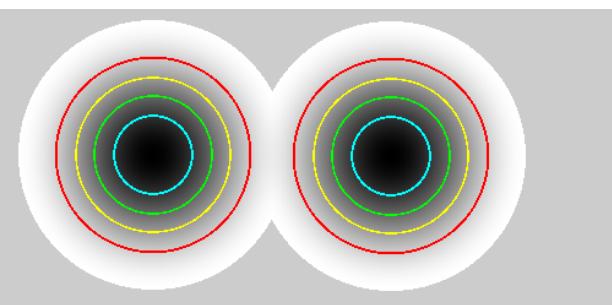
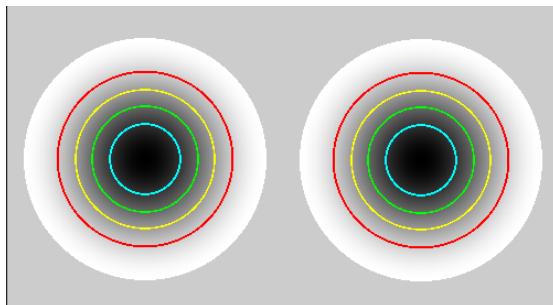
- $\alpha = 0.2$
- $\alpha = 0.4$
- $\alpha = 0.6$
- $\alpha = 0.8$

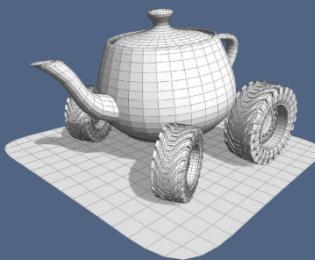
Far part, reciprocally
out of their
respective support
radius

Closer than the sum
of radii, but isolines
not affected

Near enough,
isolines for 0.2 start
to bulge

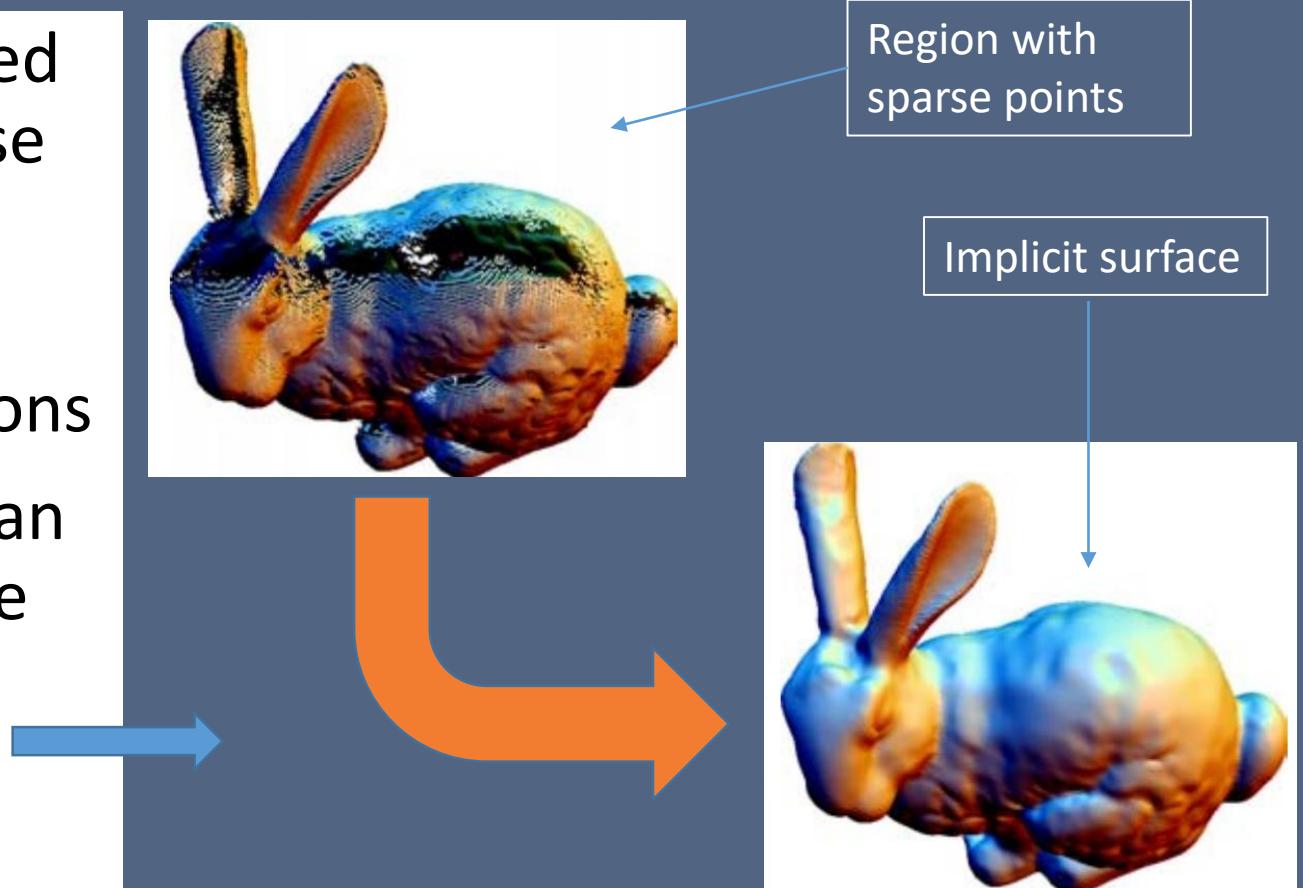
The two metaballs
become one for
isovalue 0.2

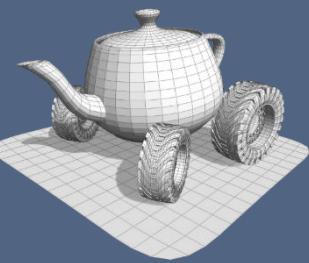




Implicit Surfaces from point clouds

- Similar formulations can be used to define isosurfaces over dense point clouds
- $f(x)$ is typically defined as the sum of weighted simple functions
- Their parameters (e.g. $\alpha, R..$) can be locally adjusted, for example for compensate for irregular density of the point cloud



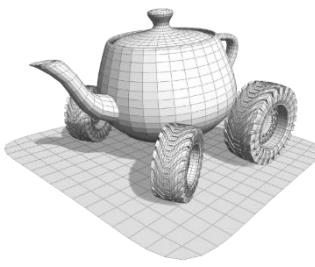


Implicit Surfaces

- Note that $f(x): \mathbb{R}^3 \rightarrow \mathbb{R}$ define a **scalar field**, that is, a mapping from \mathbb{R}^3 to \mathbb{R}
- What happens if we sum two scalar field obtained by two point clouds?

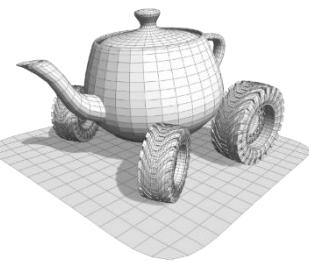


$(1 - \alpha) f_{max}(x) + \alpha f_{min}(x)$



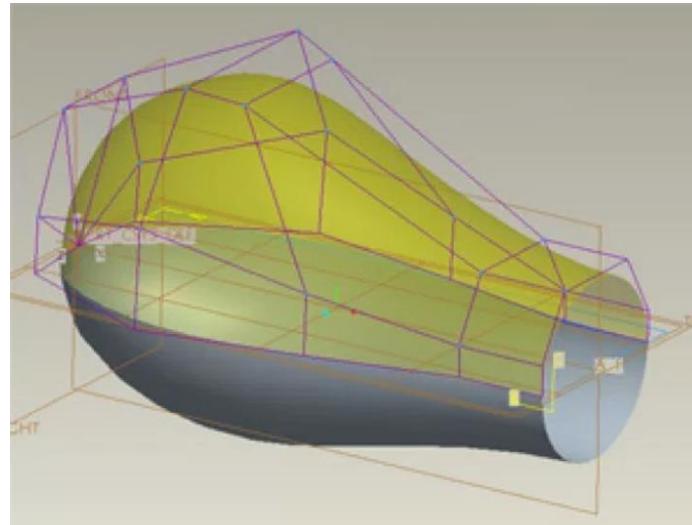
Types of 3D digital models

- Surfaces
 - Implicit surfaces
 - Parametric surfaces
 - Polygon meshes
- Volumetric
 - Voxellization
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes
- Points

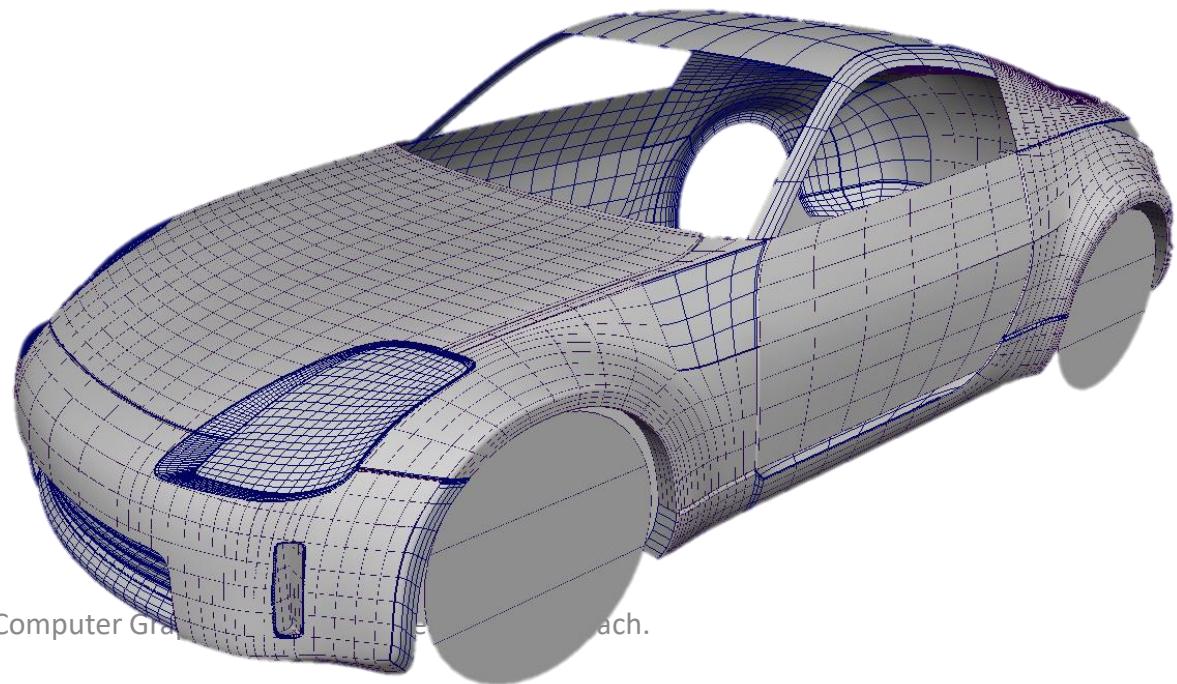


Parametric Surfaces

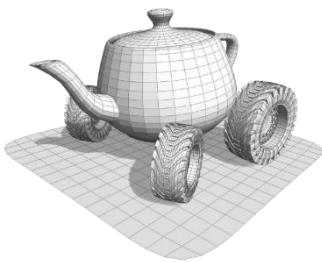
- The choice for representing complex curved surfaces as a combination of simple ones
- The basis of all CAD/CAM softwares



Chapter 3: How a 3D Model is Represented



Introduction to Computer Graphics



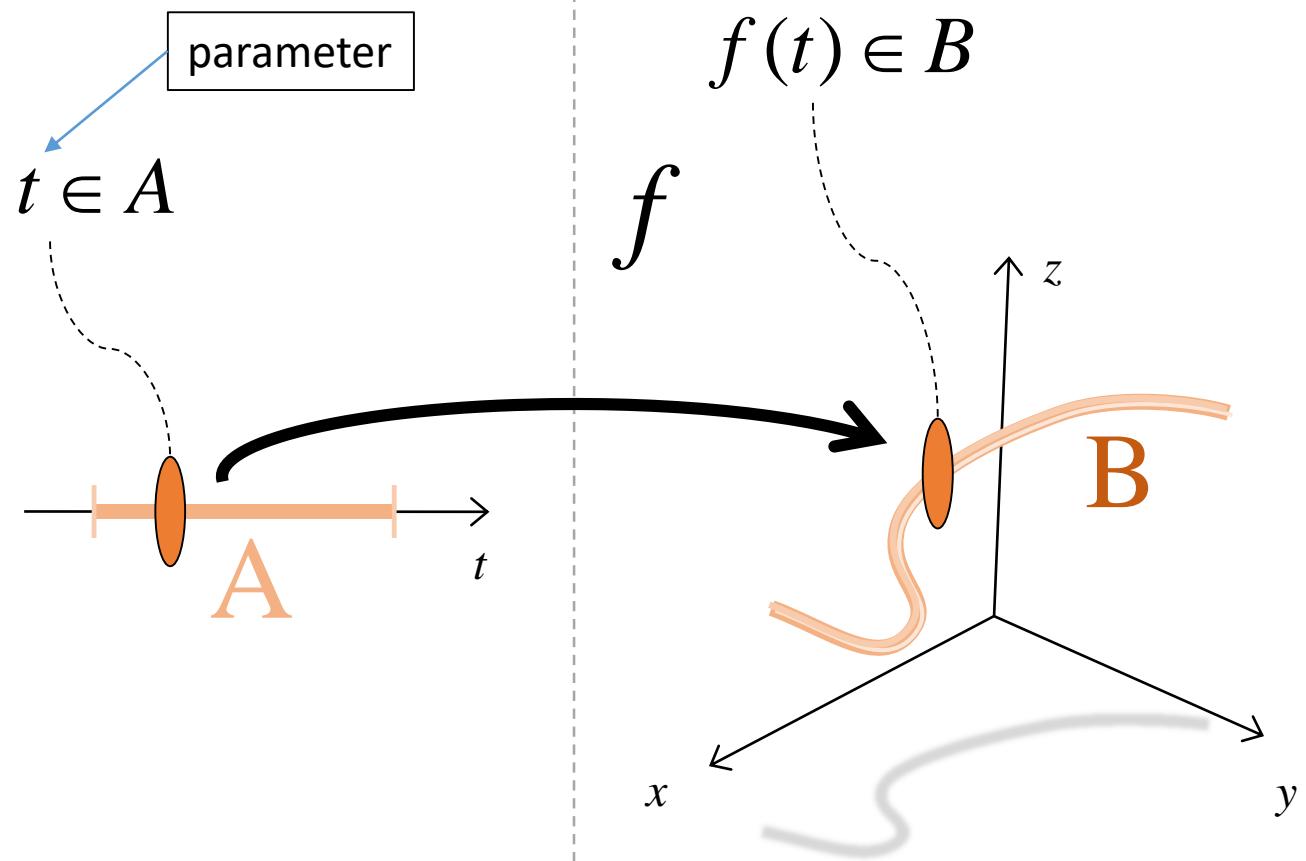
Parametric curve

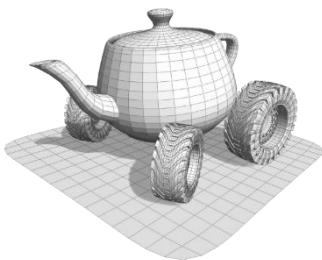
$$f: A \rightarrow B$$

$$A \subseteq \mathbb{R}$$

$$B \subseteq \mathbb{R}^3$$

$$f(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$



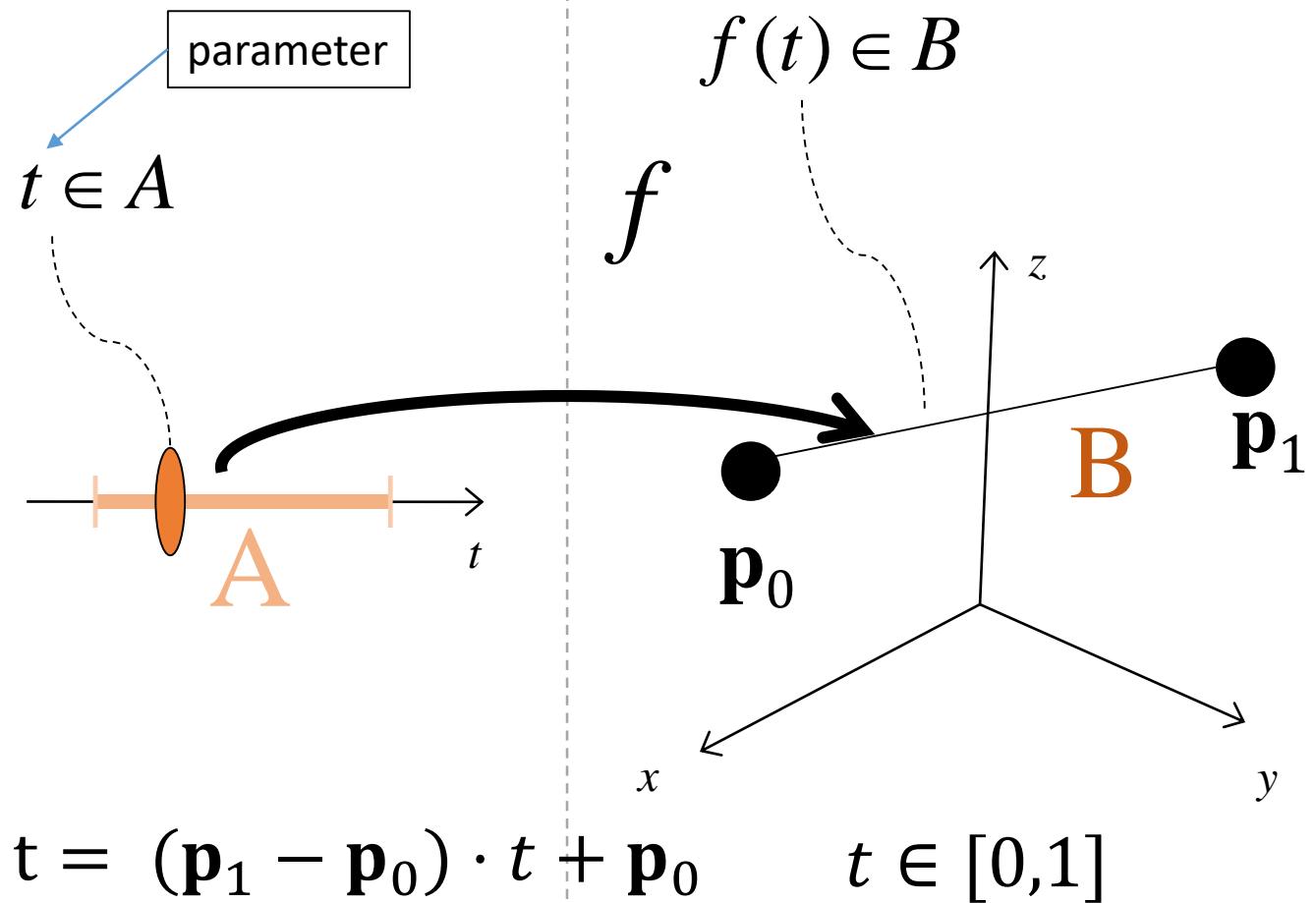


Parametric segment

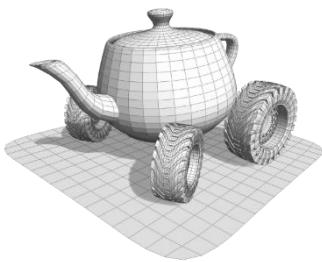
$$f: A \rightarrow B$$

$$A \subseteq \mathbb{R}$$

$$B \subseteq \mathbb{R}^2$$



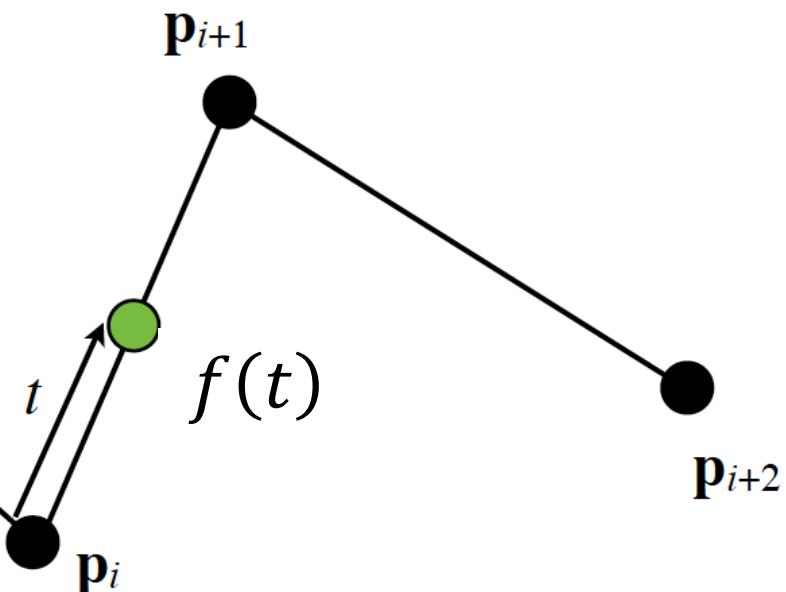
$$f(t) = \mathbf{p}_0 \cdot (1 - t) + \mathbf{p}_1 \cdot t = (\mathbf{p}_1 - \mathbf{p}_0) \cdot t + \mathbf{p}_0$$



Parametric piecewise line

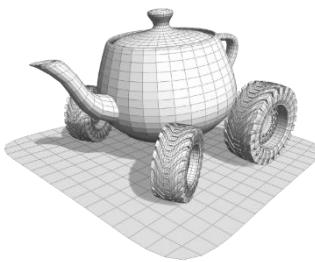
- As a concatenation of segments
- For example:

$$f(t) = \begin{cases} \mathbf{p}_{i-1} \cdot (1 - 3t) + \mathbf{p}_i \cdot 3t & t \in [0, \frac{1}{3}] \\ \mathbf{p}_i \cdot \left(1 - 3\left(t - \frac{1}{3}\right)\right) + \mathbf{p}_{i+1} \cdot 3\left(t - \frac{1}{3}\right) & t \in [\frac{1}{3}, \frac{2}{3}] \\ \mathbf{p}_{i+1} \cdot \left(1 - 3\left(t - \frac{2}{3}\right)\right) + \mathbf{p}_{i+2} \cdot 3\left(t - \frac{2}{3}\right) & t \in [\frac{2}{3}, 1] \end{cases}$$



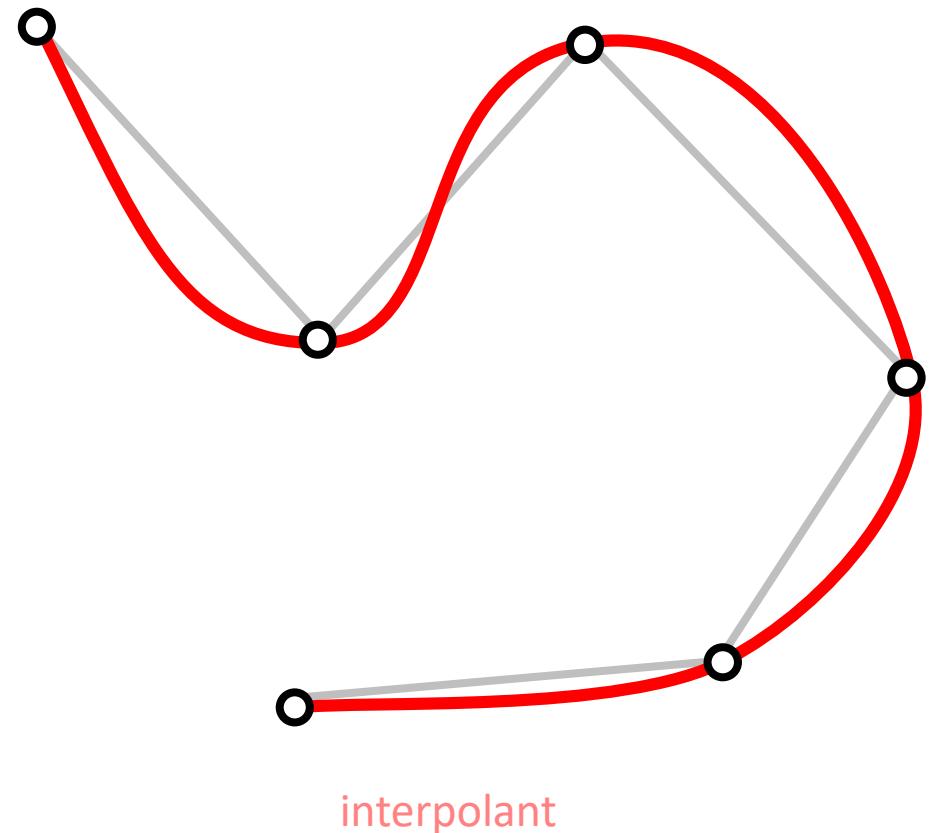
Not the only way. Considering
the three pieces as one third
each is completely arbitrary

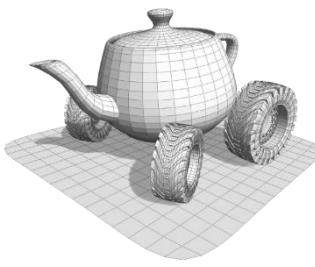
Spline



Polynomial? More in the next slide...

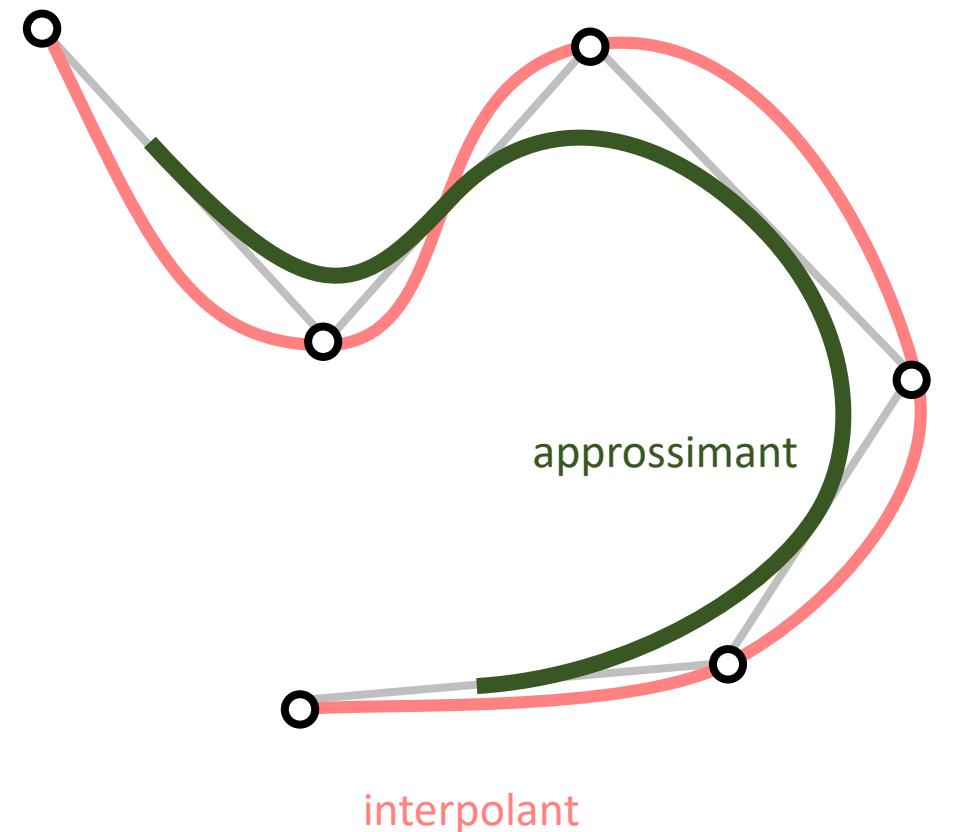
- Spline: piecewise **polynomial** curve with degree > 1
- It is defined on a sequence of **control points**
- It is said to be **interpolant** if it passes through the control points



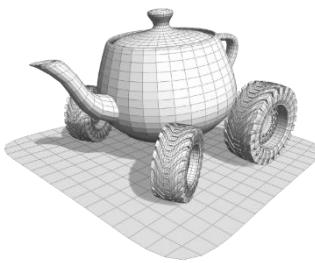


Spline

- Spline: piecewise **polynomial curve** with degree > 1
- It is defined on a sequence of **control points**
- It is said to be **interpolant** if it passes through the control points
- It is said to be **approximant** if it passes nearby



Spline formulation



$$f(x) = \sum_{i=0}^n \mathbf{p}_i B_i(t) \quad 0 \leq t \leq 1$$

Blending functions

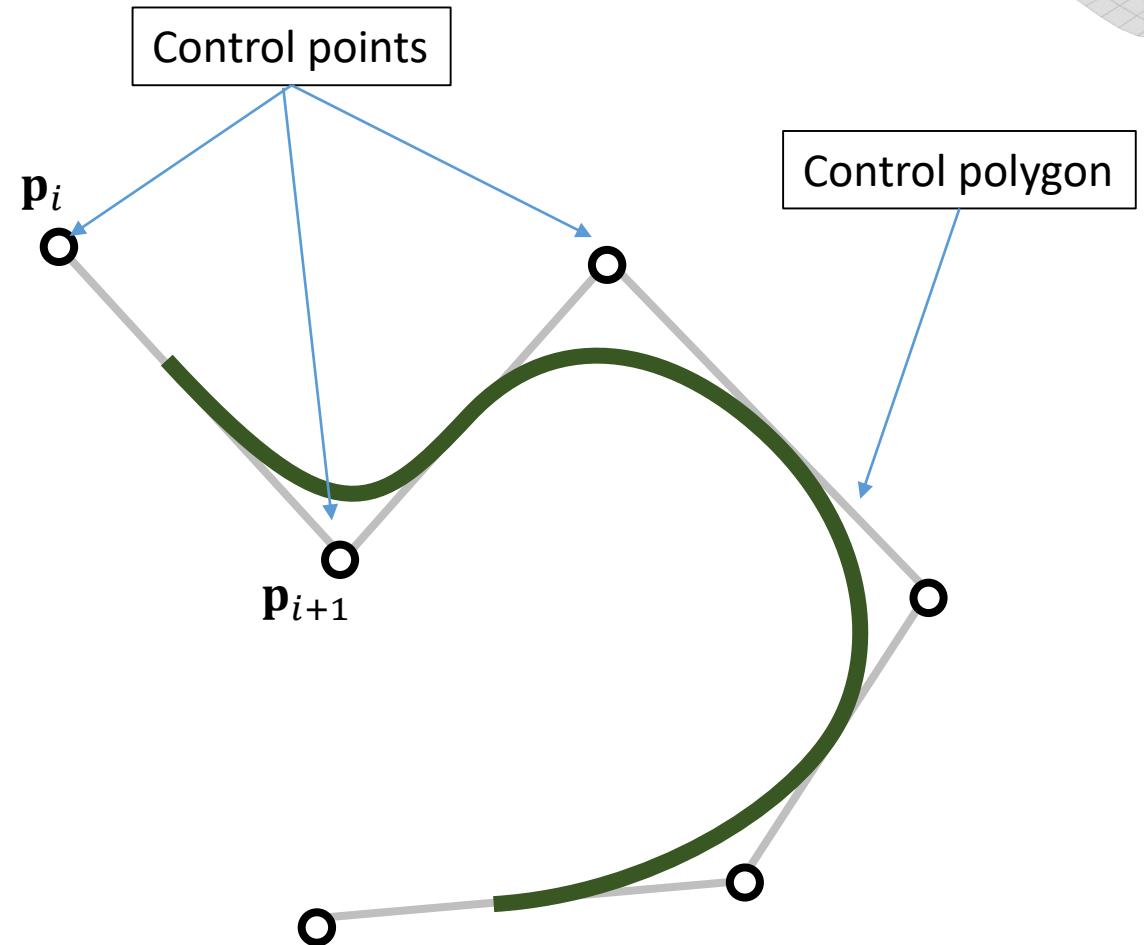
Polynomials on t

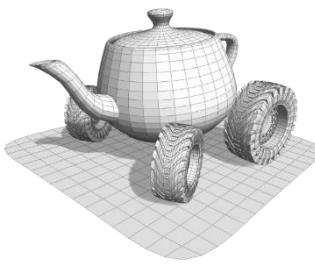
Remember the segment?

$$f_{seg}(t) = \mathbf{p}_0 \cdot (1 - t) + \mathbf{p}_1 \cdot t$$

$$B_0(t) = 1 - t$$

$$B_1(t) = t$$





Bèzier curves

- A widely used type of curve which uses **Bernstein polynomials** as blending functions

$$f(t) = \sum_{i=0}^n \mathbf{p}_i B_{i,n}(t) \quad 0 \leq t \leq 1$$

Degree n, control points n+1

Binomial coefficient $\rightarrow \binom{n}{i} = \frac{n!}{i!(n-i)!}$

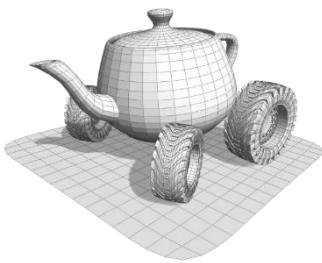
$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Bernstein polynome

$$B_{i,n}(t) = 0, i < 0 \text{ or } i > n$$

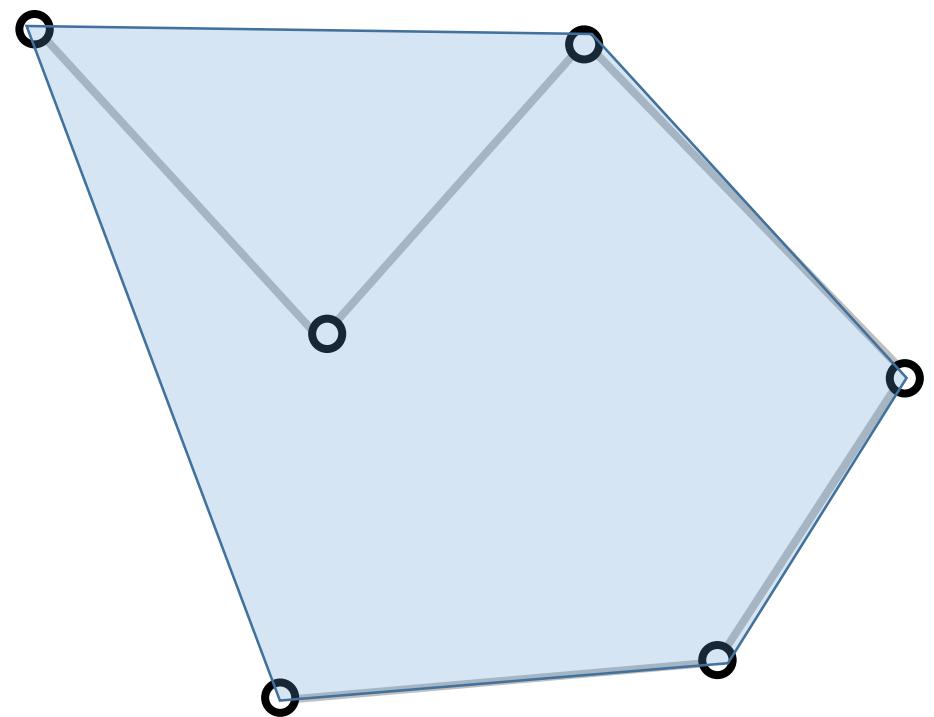
Remember the segment?
 $f_{seg}(t) = \mathbf{p}_0 \cdot (1-t) + \mathbf{p}_1 \cdot t$

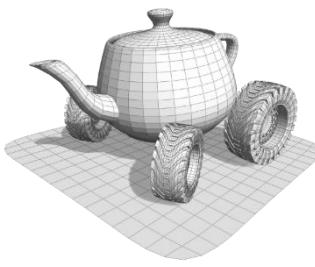
$$B_{0,1}(t) = 1 - t$$
$$B_{1,1}(t) = t$$



Bernstein polynomials

- Se set of Bernstein Polynomial of degree n form the **Bernstein Basis** $\mathcal{B}_n = \{B_{0,n}(t), B_{1,n}(t), \dots, B_{n,n}(t)\}$
- The sum of all terms of a Bernstein basis is 1
 - Therefore, the Bezièr curve always lies in the convex hull of the control points



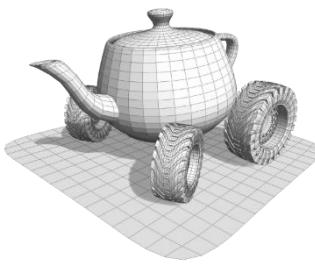


Bernstein polynomials

- Se set of Bernstein Polynomial of degree n form the **Bernstein Basis** $\mathcal{B}_n = \{B_{0,n}(t), B_{1,n}(t), \dots, B_{n,n}(t)\}$
- The sum of all terms of a Bernstein basis is 1
 - Therefore, the Bezièr curve always lies in the convex hull of the control points
- The basis of degree $n+1$ is expressed as a linear combination of the terms of basis n :

$$B_{i,n}(t) = B_{i-1,n-1}(t) t + B_{i,n-1}(t)(1 - t)$$

Provides a simple incremental algorithm for evaluating the curve...



Evaluating a Bèzier curve

$$B_{i,n}(t) = B_{i-1,n-1}(t) t + B_{i,n-1}(t)(1 - t)$$

$$B_{i,n}(t) = 0, i < 0 \text{ or } i > n$$

$$B_{0,1}(t) = 1 - t, \quad B_{1,1}(t) = t$$

$$f(\mathbf{t}) = \mathbf{p}_0 B_{0,2}(t) + \mathbf{p}_1 B_{1,2}(t) + \mathbf{p}_2 B_{2,2}(t)$$

$$\cancel{B_{-1,1}(t)t} + B_{0,1}(t) (1 - t)$$

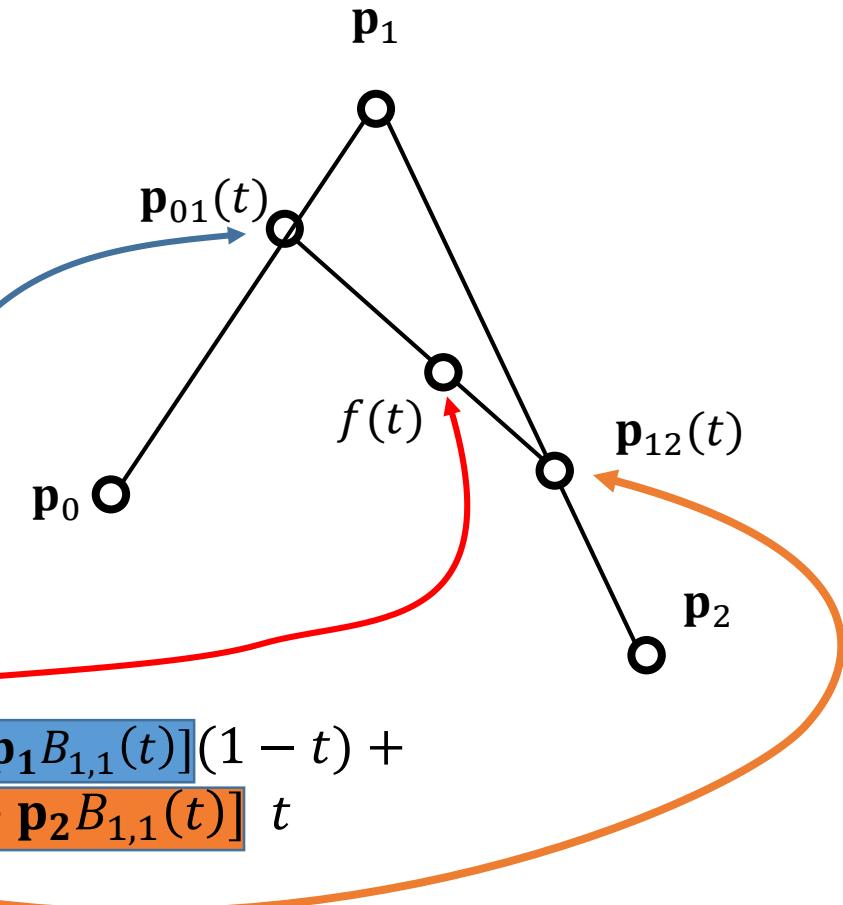
$$B_{0,1}(t)t + \cancel{B_{1,1}(t)} (1 - t)$$

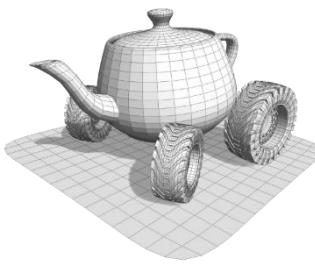
$$\cancel{B_{1,1}(t)t} + B_{2,1}(t) (1 - t)$$

$$f(\mathbf{t}) = \mathbf{p}_0 B_{0,1}(t)(1 - t) + \mathbf{p}_1 B_{1,1}(t)(1 - t) + \\ \mathbf{p}_1 B_{0,1}(t) t + \mathbf{p}_2 B_{1,1}(t) t$$



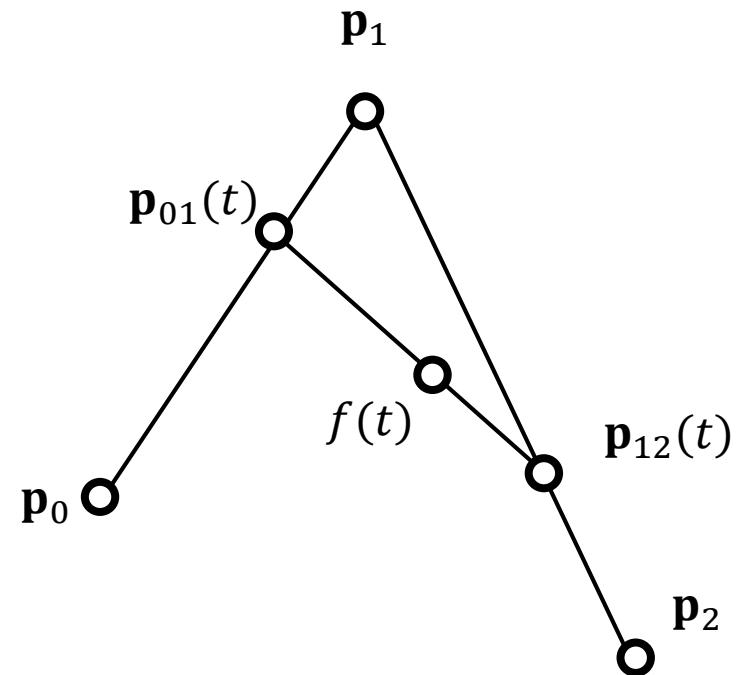
$$f(\mathbf{t}) = [\mathbf{p}_0 B_{0,1}(t) + \mathbf{p}_1 B_{1,1}(t)](1 - t) + \\ [\mathbf{p}_1 B_{0,1}(t) + \mathbf{p}_2 B_{1,1}(t)] t$$





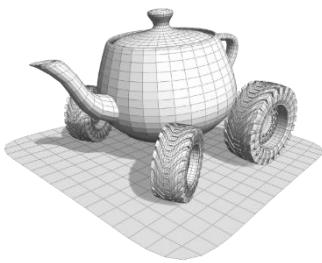
Bèzier curves of degree 2

- $n=2 \rightarrow 3$ control points p_0, p_1, p_2
- *De Casteljau* algorithm to find the point $f(t)$
 - Find the point $\mathbf{p}_{01}(t)$ of the linear Bèzier curve between p_0 and p_1
 - Find the point $\mathbf{p}_{12}(t)$ of the linear Bèzier curve between p_1 and p_2
 - Find the point $f(t)$ of the linear Bèzier curve between $\mathbf{p}_{01}(t)$ and $\mathbf{p}_{12}(t)$



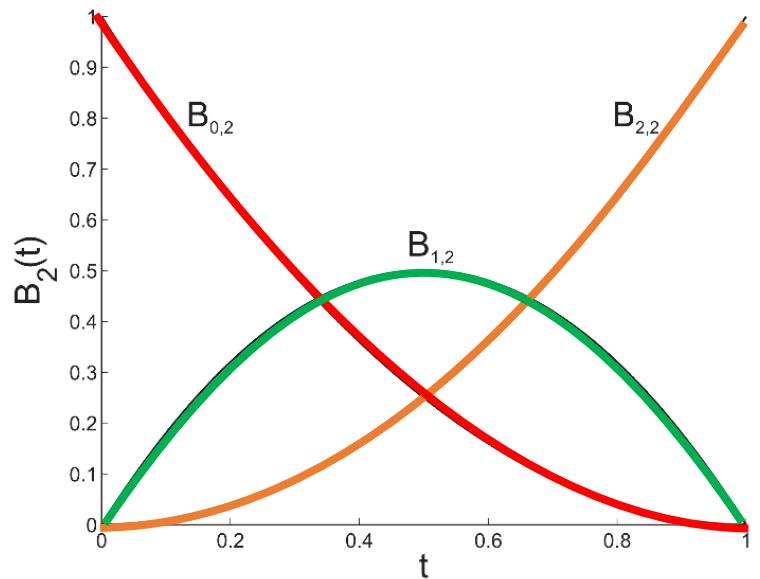
[demo](#)

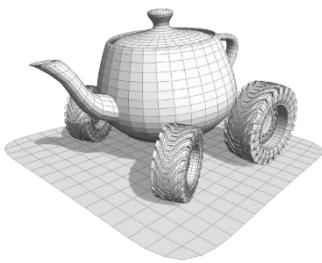
[De Casteljau's Algorithm and Bézier Curves \(malinc.se\)](#)



Quadratic Bézier curves: weights

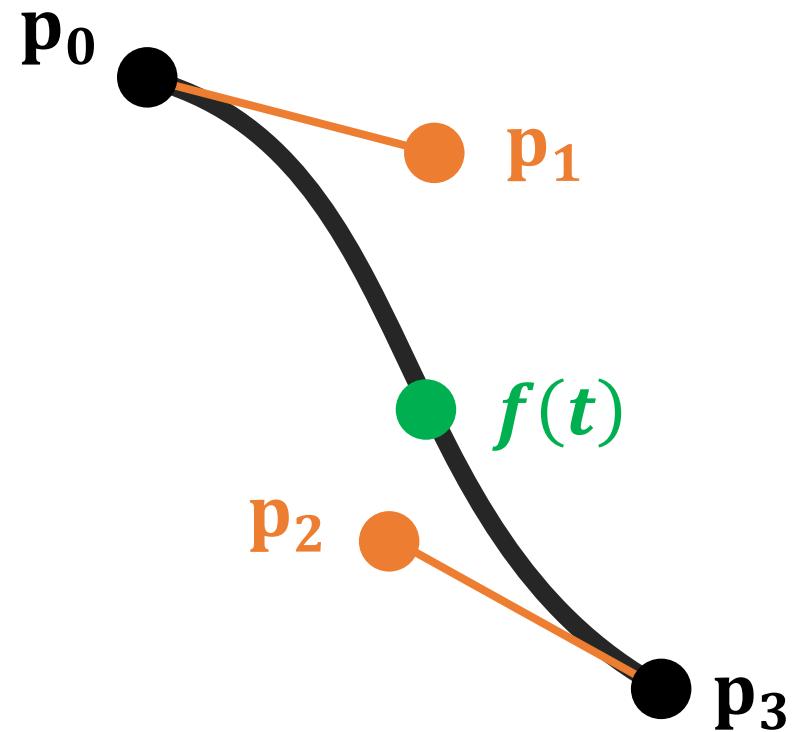
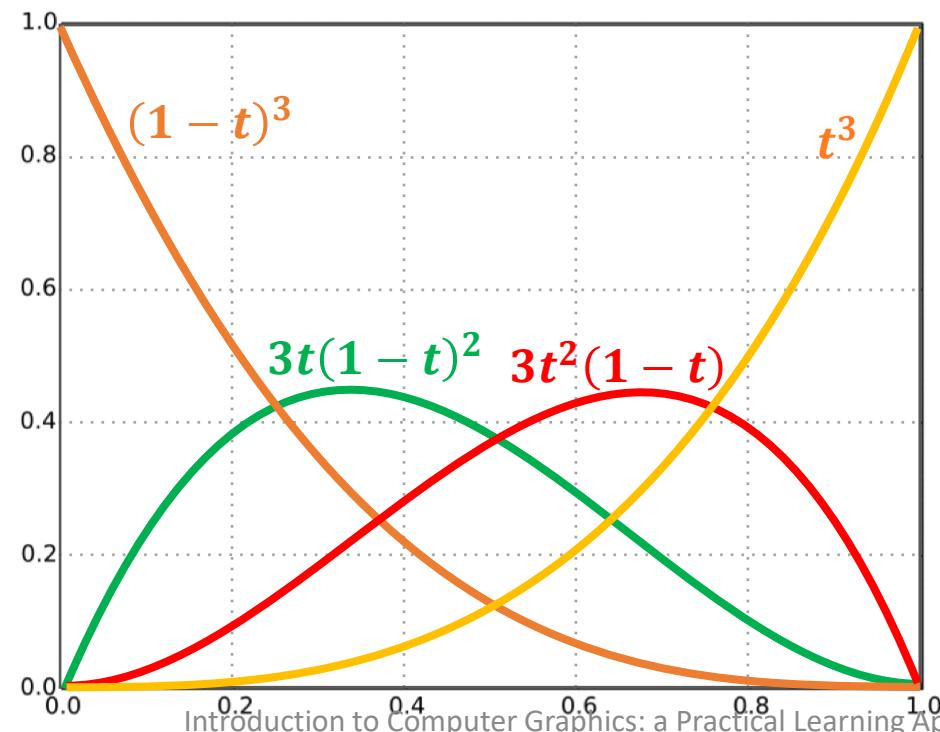
$$f(t) = \frac{(1-t)^2}{2t(1-t)} \mathbf{p}_0 + \frac{t^2}{t^2} \mathbf{p}_1 +$$

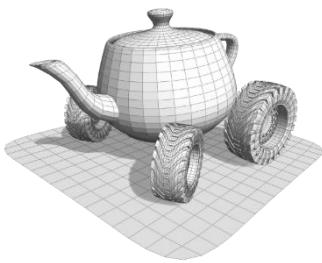




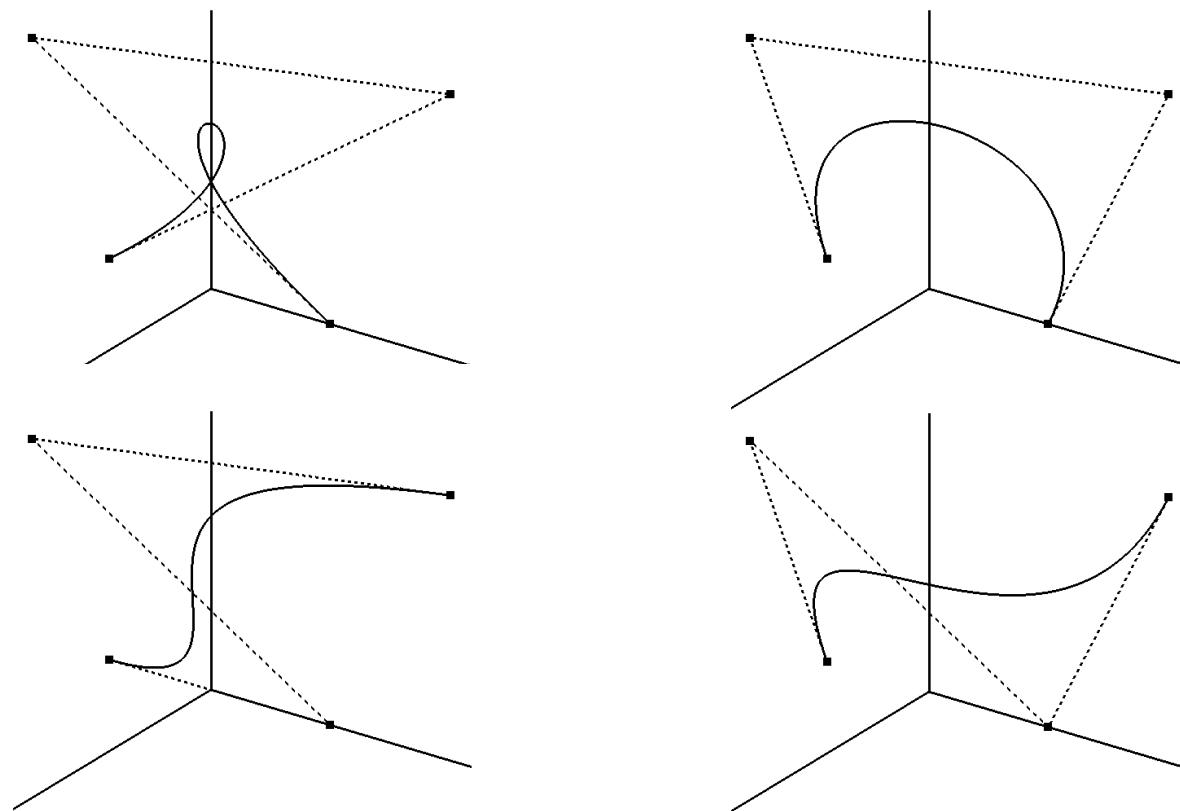
Cubic Bézier curves: weights

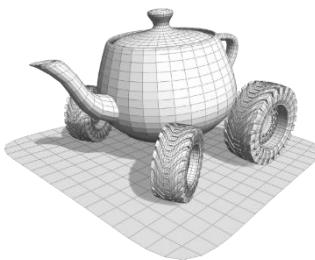
$$\begin{aligned}f(t) &= (1-t)^3 p_0 \\&\quad + 3t(1-t)^2 p_1 \\&\quad + 3t^2(1-t) p_2 \\&\quad + t^3 p_3\end{aligned}$$





Cubic Bézier curves: examples



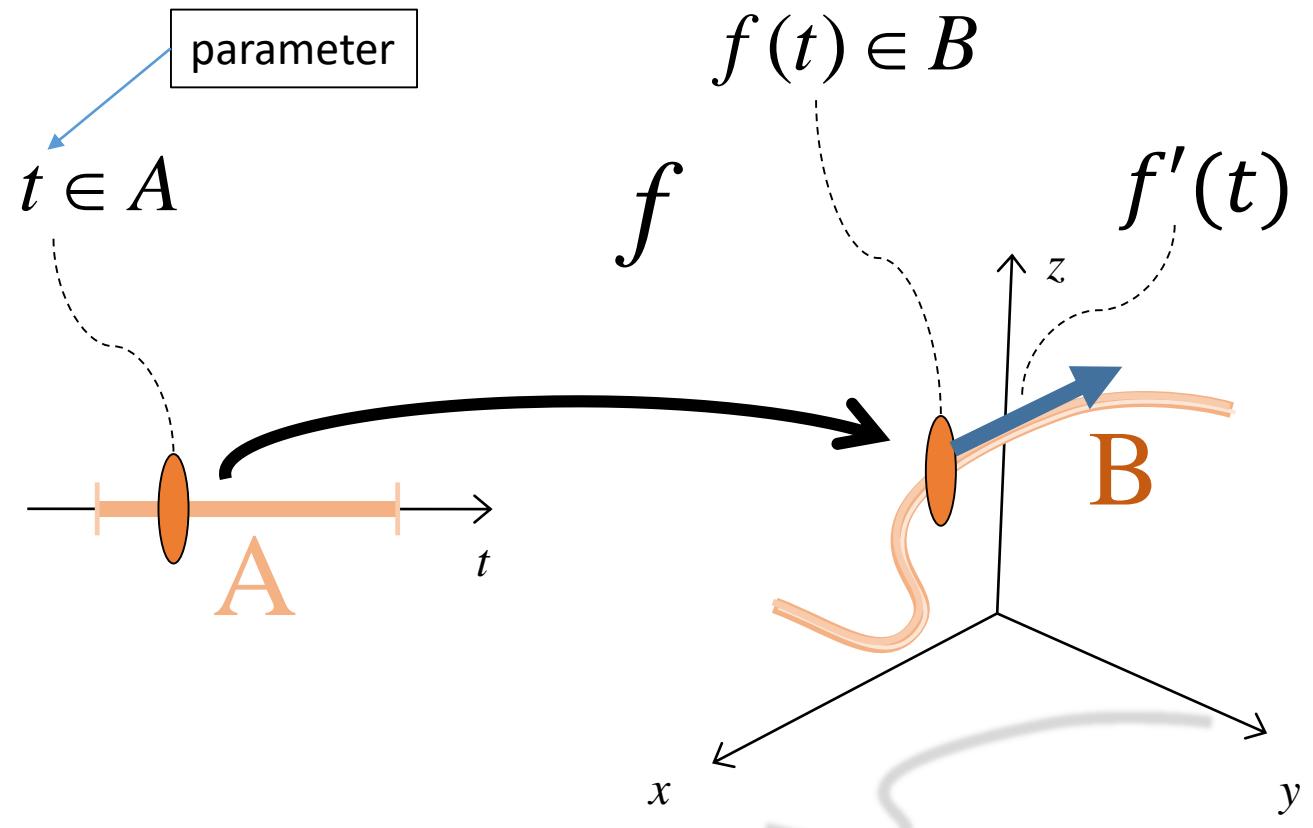


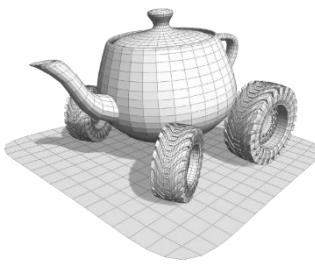
Tangent to a parametric curve

- The tangent vector to a curve is found as:

$$\lim_{dt \rightarrow 0} \frac{f(t + dt) - f(t)}{dt}$$

- That is, $f'(t)$ the derivative of $f(t)$





Tangent to a parametric curve

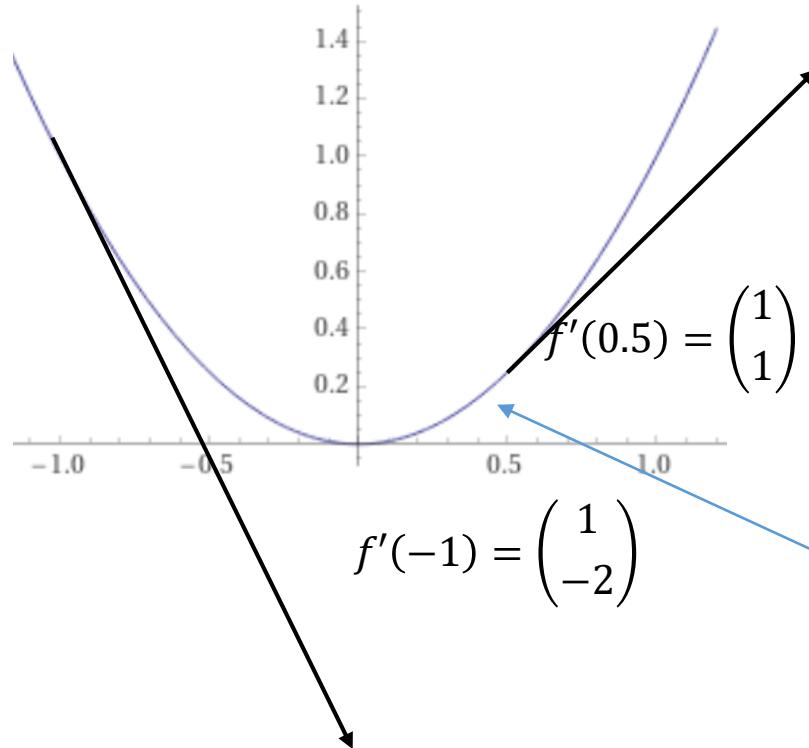
- Example: $y = x^2$

$$f(t) = \begin{pmatrix} t \\ t^2 \end{pmatrix}$$

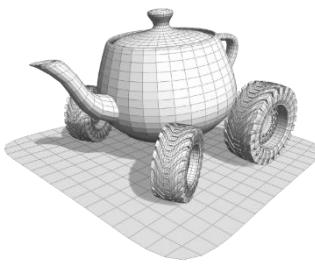
$$f'(t) = \begin{pmatrix} 1 \\ 2t \end{pmatrix}$$

- unit tangent vector:

$$T(f(t)) = \frac{f'(t)}{\|f'(t)\|}$$



Can we make this
parabola with a
Bézier curve?



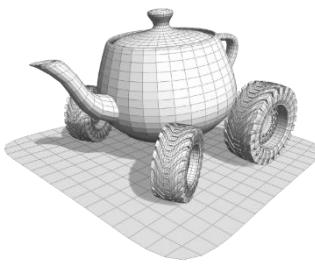
Tangent to a Bezier curve

- The derivative of a Bezier curve of degree n is the derivative of a sum of n terms
- The derivative of a Bernstein polynome is found as:
- Which ultimately gives:

$$\frac{d}{dt} f(t) = \sum_{i=0}^n \mathbf{p}_i \frac{d}{dt} B_{i,n}(t)$$

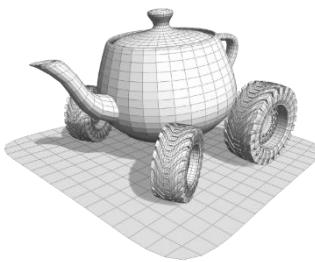
$$\frac{d}{dt} B_{i,n}(t) = n \left(B_{i-1,n-1}(t) - B_{i,n-1}(t) \right)$$

$$\frac{d}{dt} f(t) = n \sum_{i=0}^{n-1} \mathbf{p}_{i+1} B_{i-1,n-1}(t) - \mathbf{p}_i B_{i,n-1}(t)$$



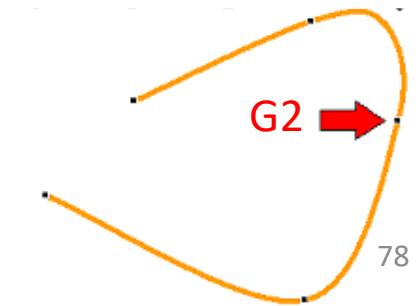
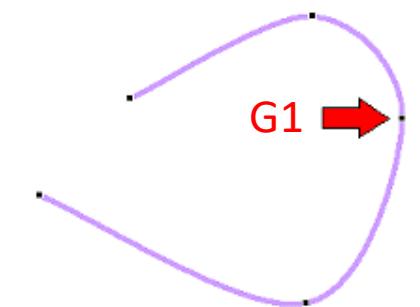
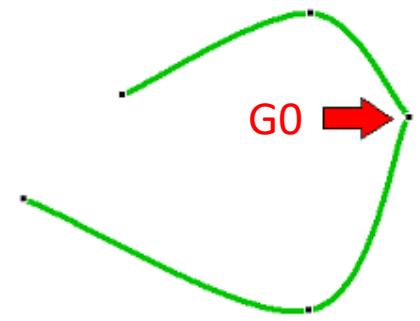
Bézier curves properties

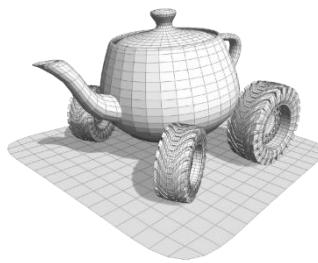
- The first and the last control points of the control polygon are interpolated
- The curve at the first and last control points is tangent the first and last segment of the control polygon, respectively
- If a straight line crosses the control polygon n times, it crosses the curve *at most* n times



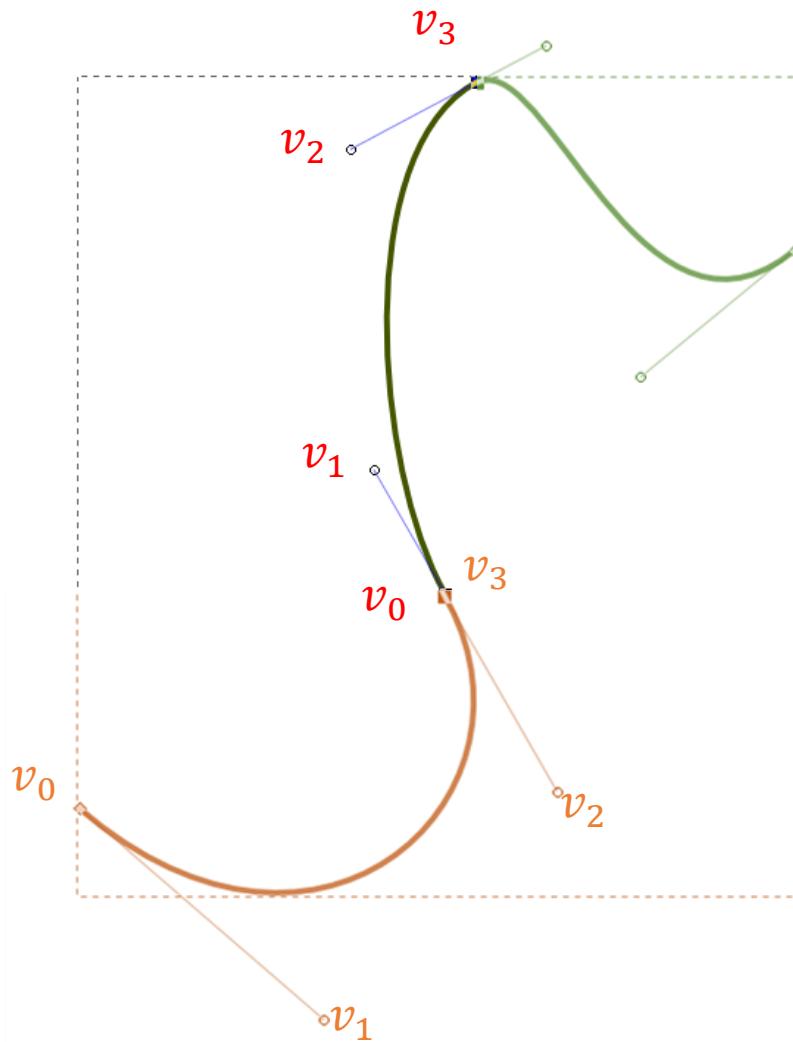
Spline: concatenating polynomial curves

- We can make more complex curves by concatenating multiple polynomial curves
- Each curve is C^∞
- In the joints few things can happen:
 - The curves touch: G0
 - They also have the same unit tangent G1 («smooth»)
 - If also their second derivative is the same: G2 («perfectly smooth»)





Example on *Bézier path*



iff the end points are the same:

$$(v_3 = v_0)$$

then the curve is C0

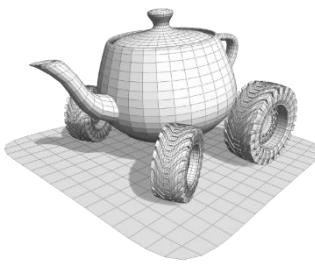
Iff

($v_2, v_3 = v_0, v_1$) are collinear
then the curve is G1

Iff is G1 and

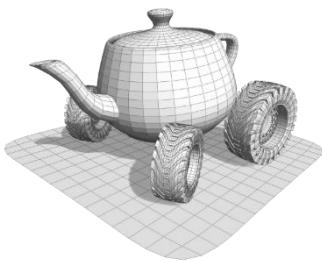
$$\|v_2 - v_3\| = \|v_0 - v_1\|$$

then the curve is G2



Bézier curves and Bézier paths

- Very used in:
 - 2D design (most applications)
 - SVG (Adobe Illustrator, Inkscape) for vector images
 - Font specification
 - Ideal for animation paths where “something” moves along the curve



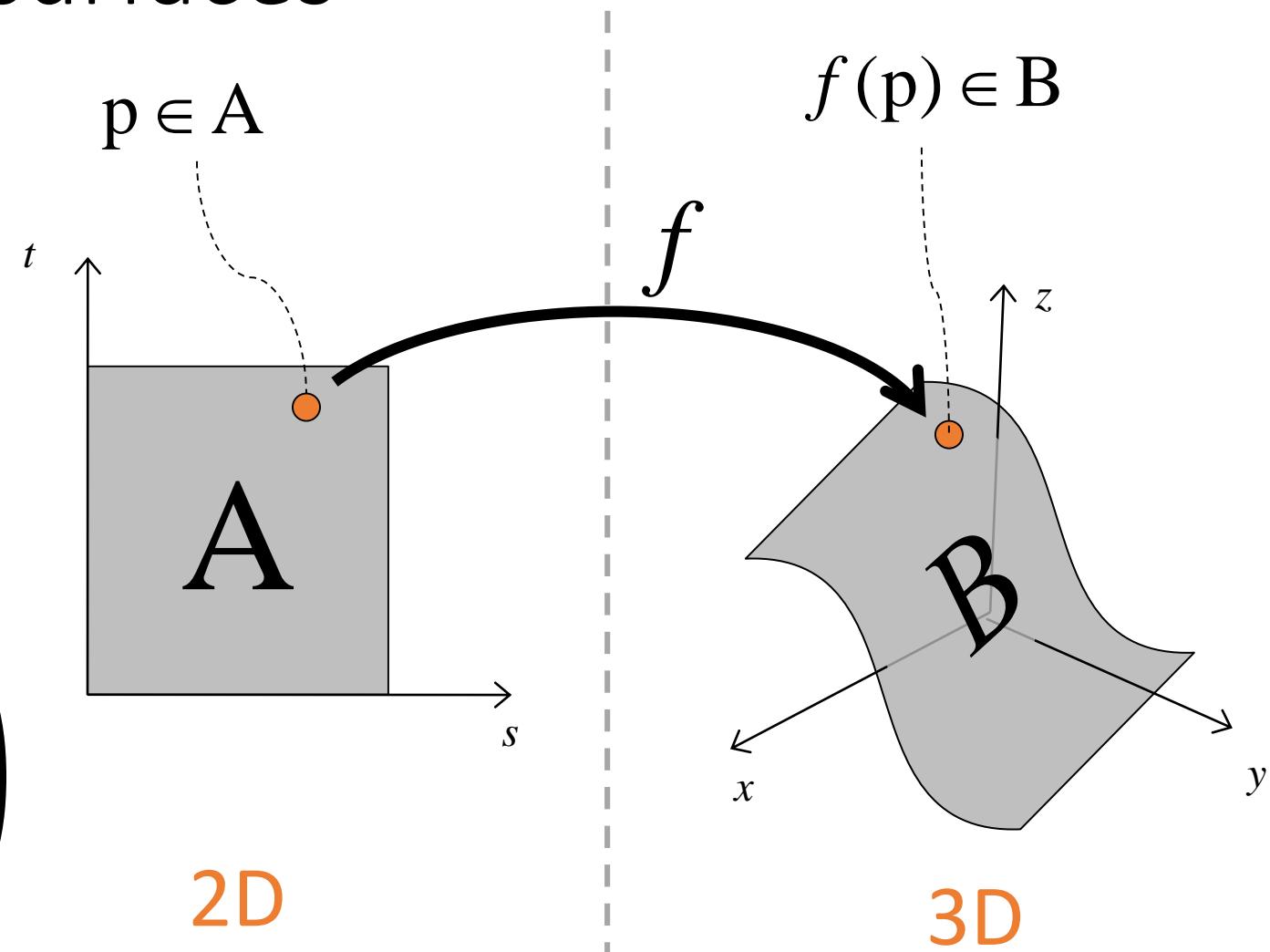
Parametric surfaces

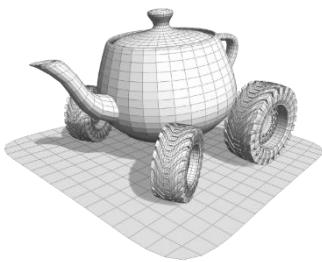
$$f: A \rightarrow B$$

$$A \subseteq \mathbb{R}^2$$

$$B \subseteq \mathbb{R}^3$$

$$f \begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$



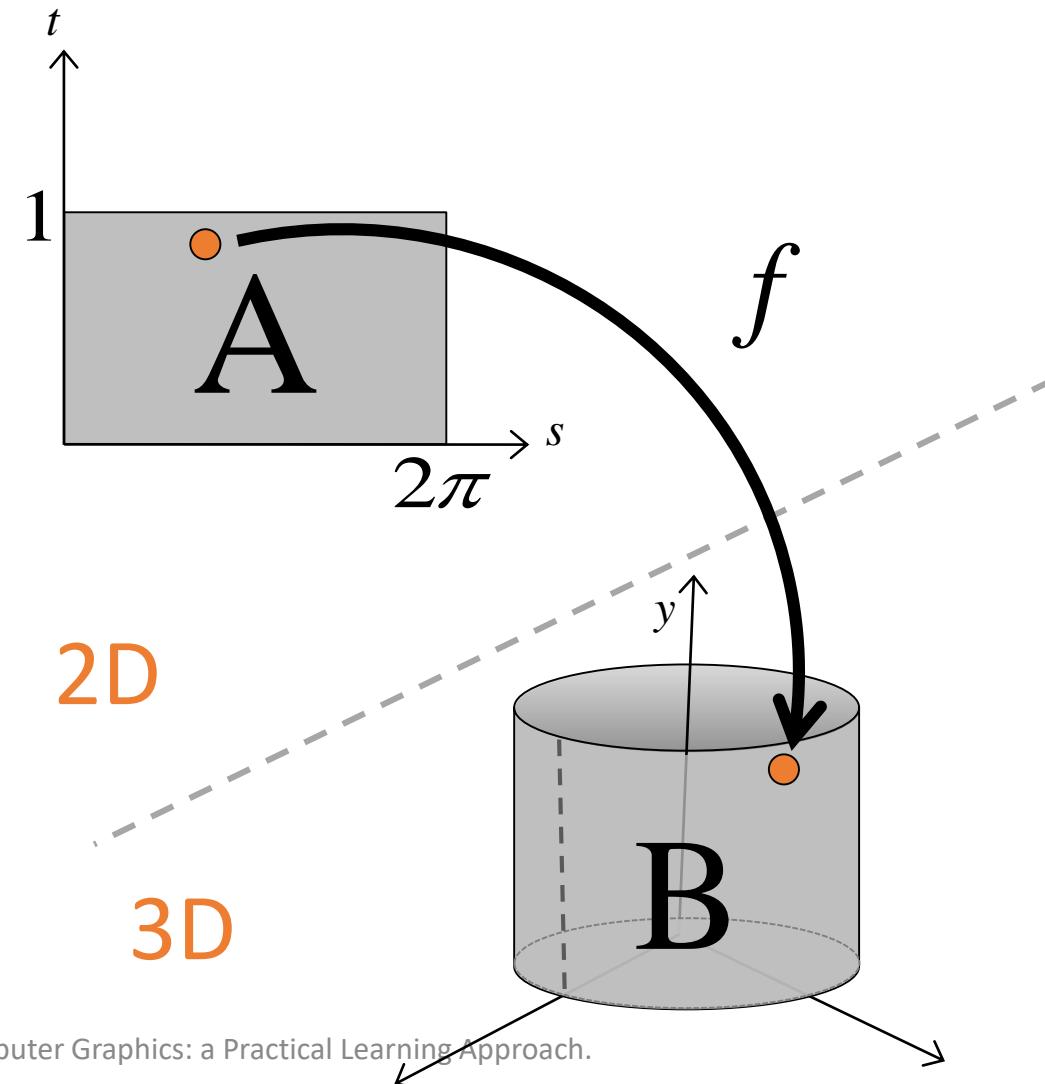


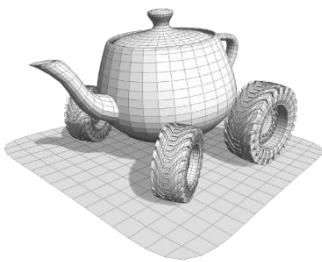
Parametric surfaces: example

$$f : A \rightarrow B$$

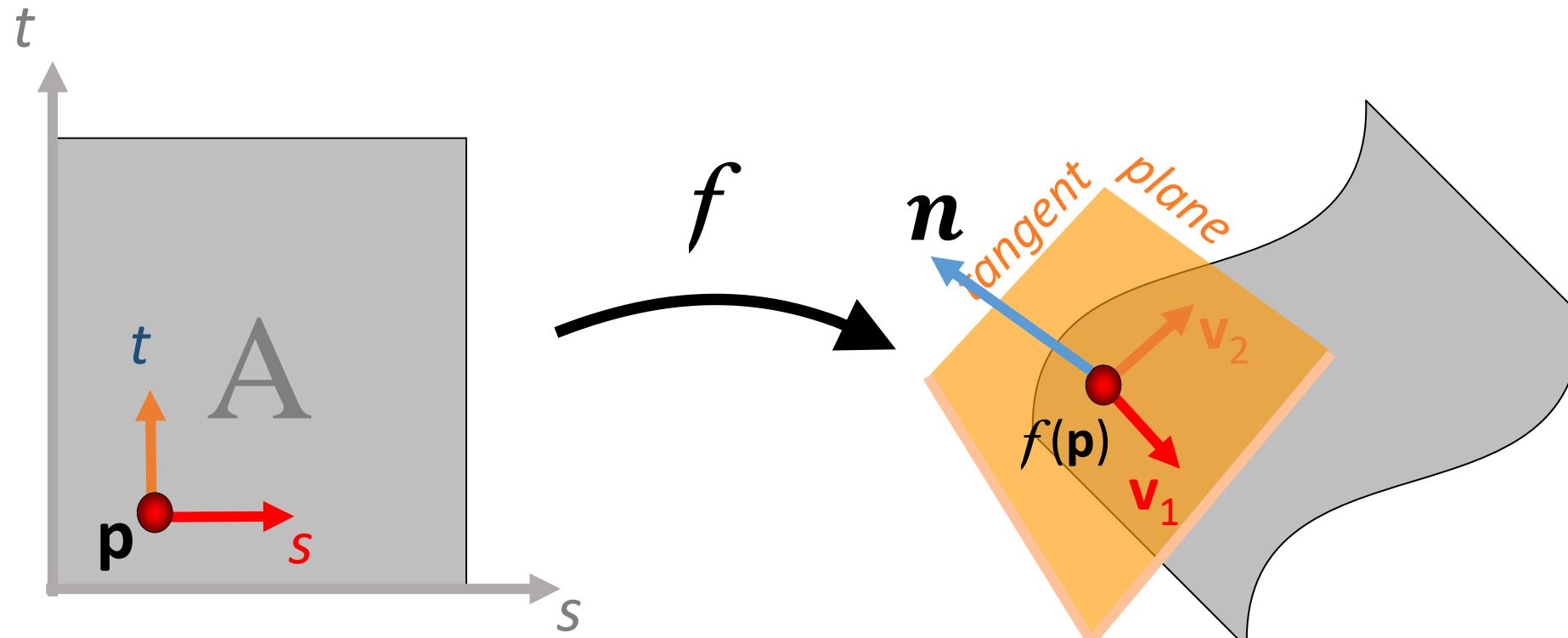
$$A = [0, 2\pi] \times [0, 1]$$

$$f \begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} r \cos s \\ h \cdot t \\ r \sin s \end{pmatrix}$$





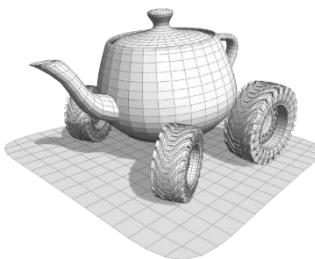
Normal of a parametric surface



$$v_2 = \frac{\partial f}{\partial t}(p)$$

$$v_1 = \frac{\partial f}{\partial s}(p)$$

$$n = \frac{v_1 \times v_2}{\|v_1 \times v_2\|}$$



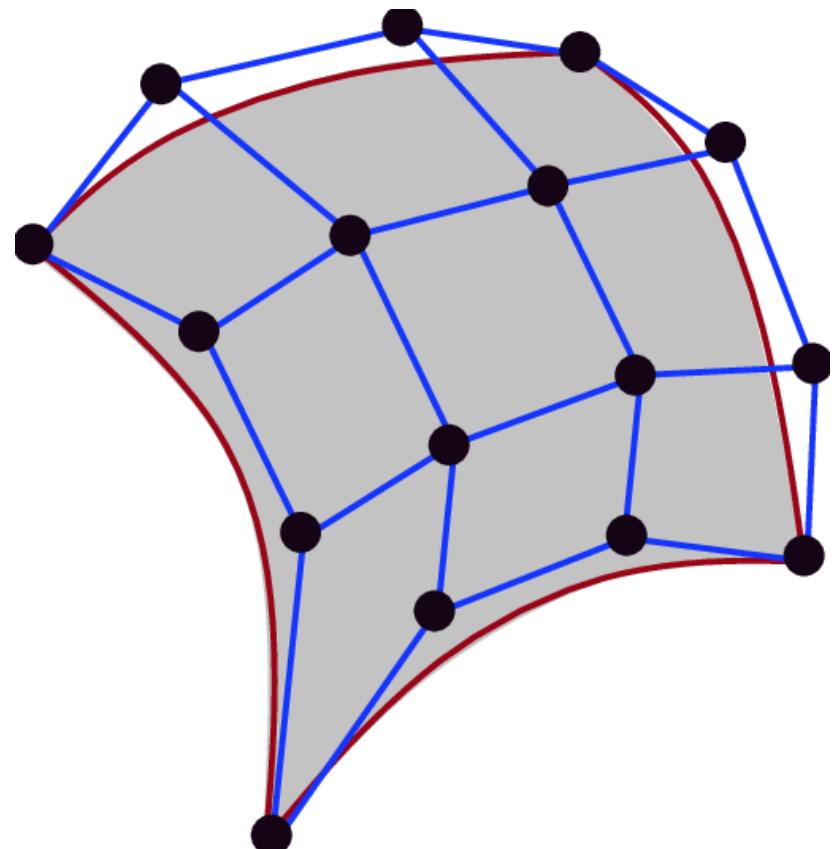
Bezier patch

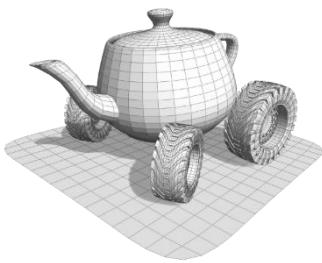
- A **Bezier patch** is defined as

$$f(s, t) = \sum_{i=0}^n \sum_{j=0}^n \mathbf{p}_{i,j} B_{i,n}(t) B_{j,n}(t) \quad 0 \leq t \leq 1$$

where $\mathbf{p}_{i,j}$ $i, j = 0 \dots n$ is a grid of $n+1$ control points

Example: cubic Bezier patch
 $n=3$
 $(3+1)*(3+1)=16$ control points





Bezier patch

- Note that:

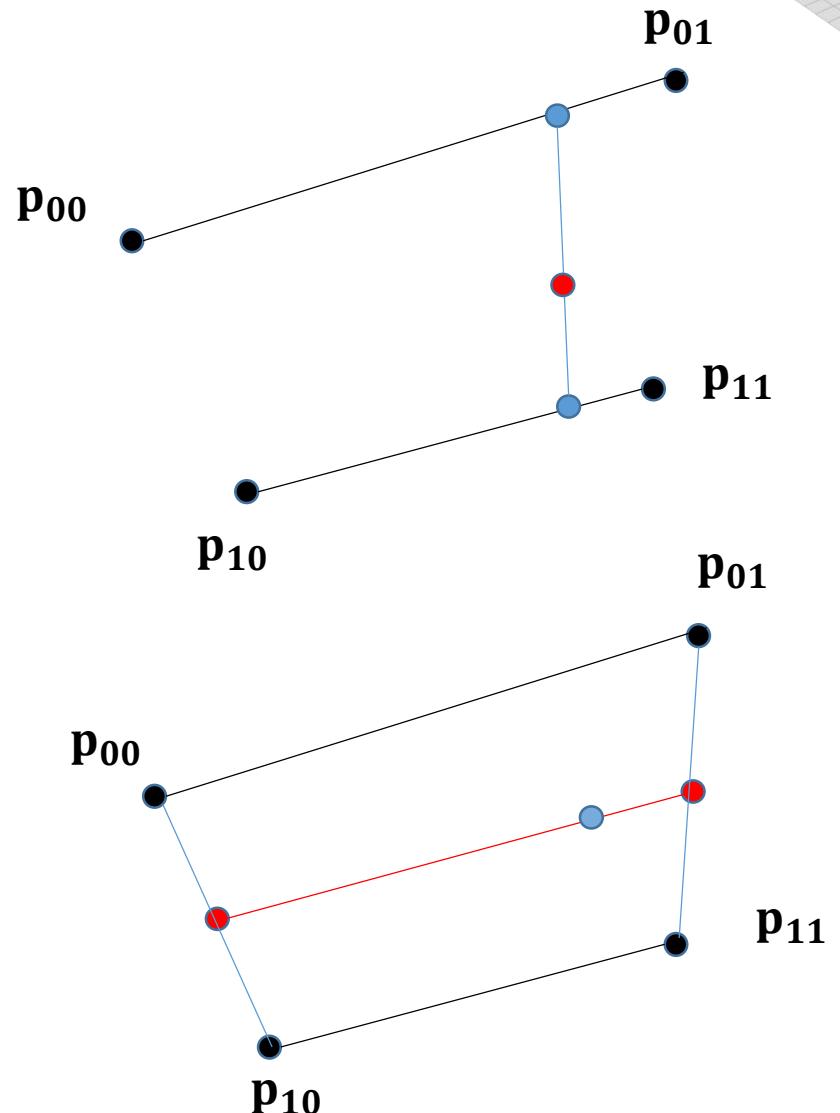
$$f(s, t) = \sum_{i=0}^n \sum_{j=0}^n \mathbf{p}_{i,j} B_{i,n}(s) B_{j,n}(t) \quad 0 \leq t \leq 1$$

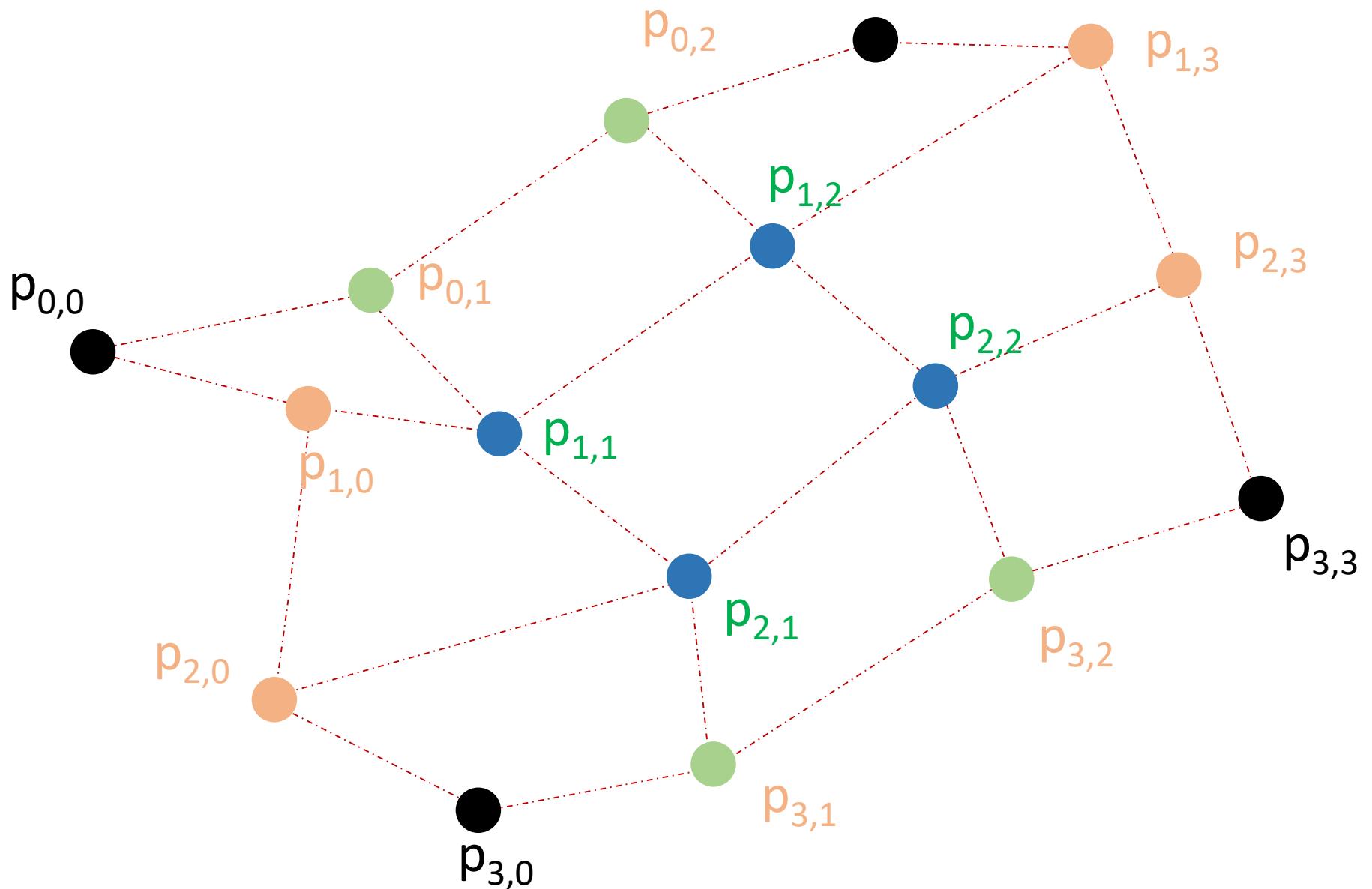
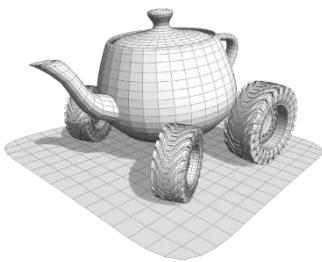
=

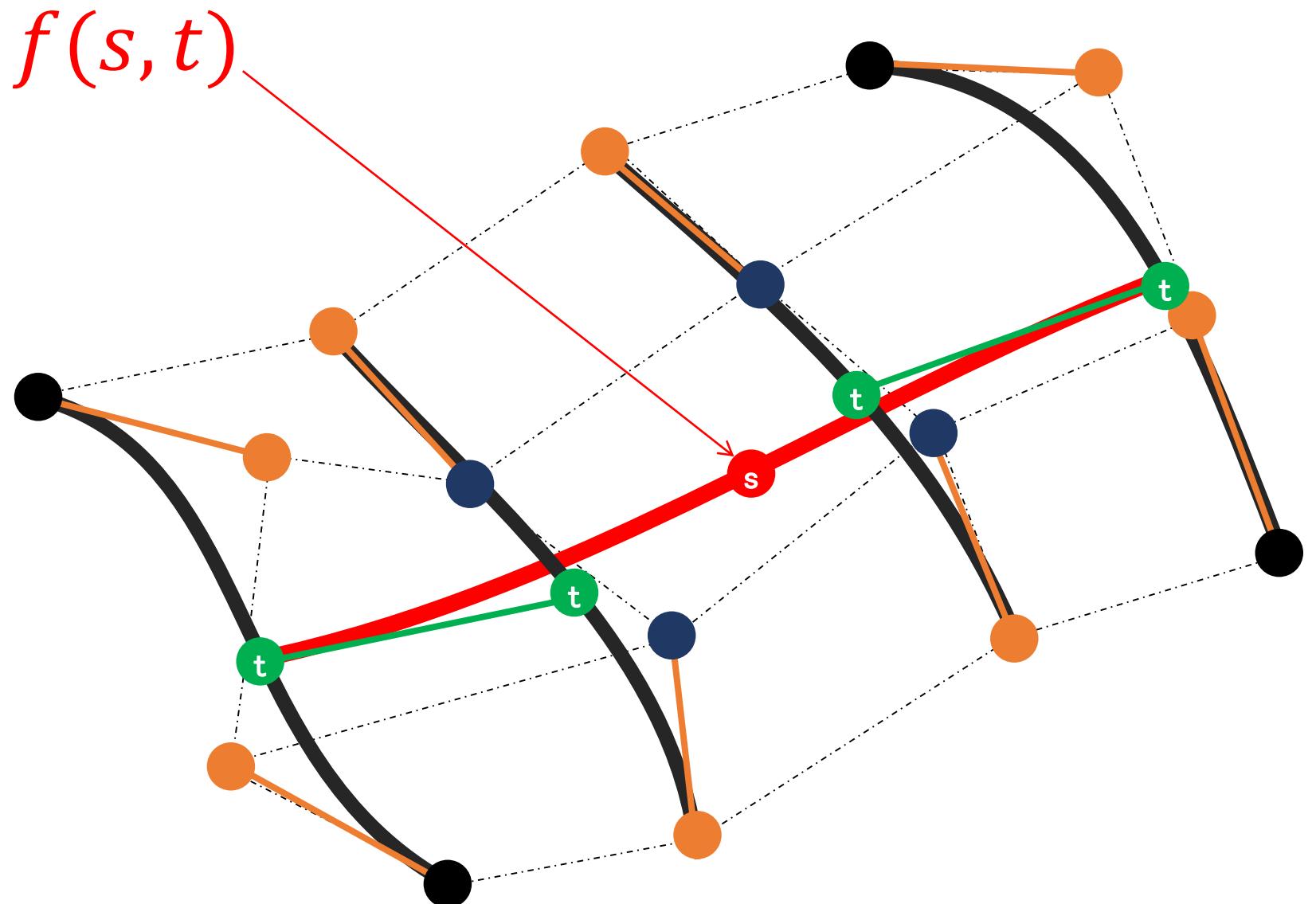
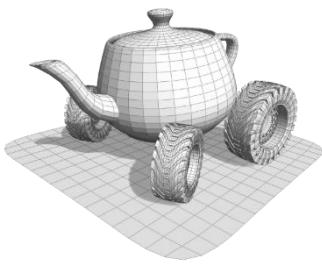
$$f(s, t) = \sum_{i=0}^n B_{i,n}(s) \sum_{j=0}^n \mathbf{p}_{i,j} B_{j,n}(t) \quad 0 \leq t \leq 1$$

=

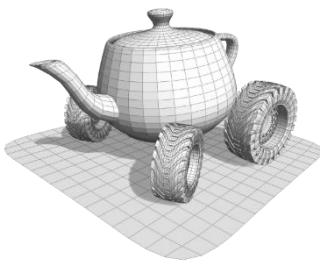
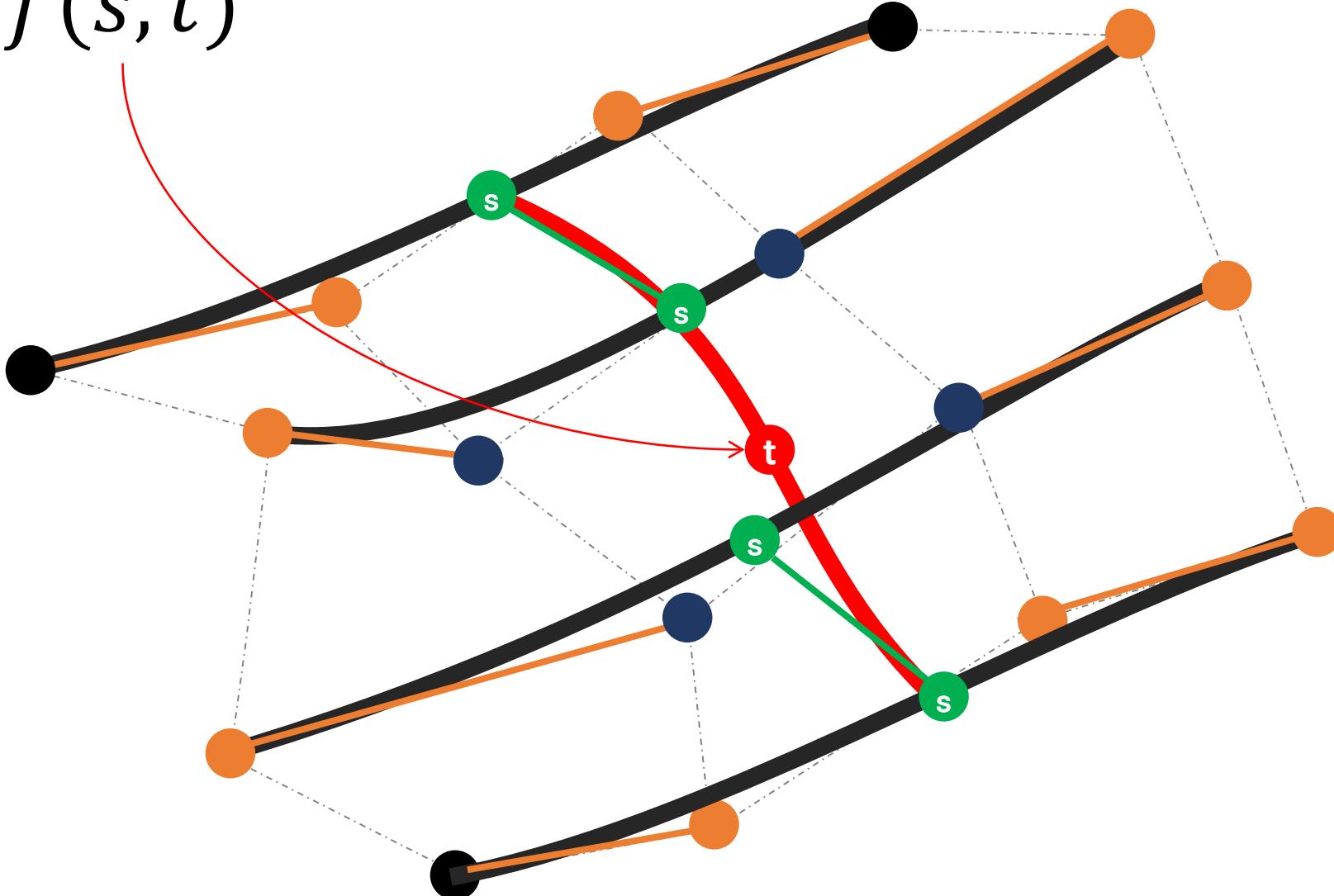
$$f(s, t) = \sum_{j=0}^n B_{j,n}(t) \sum_{i=0}^n \mathbf{p}_{i,j} B_{i,n}(s) \quad 0 \leq t \leq 1$$

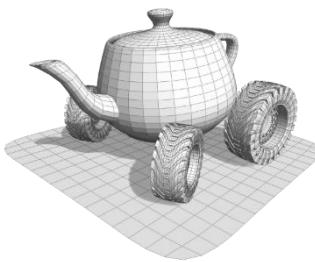






$$f(s, t)$$



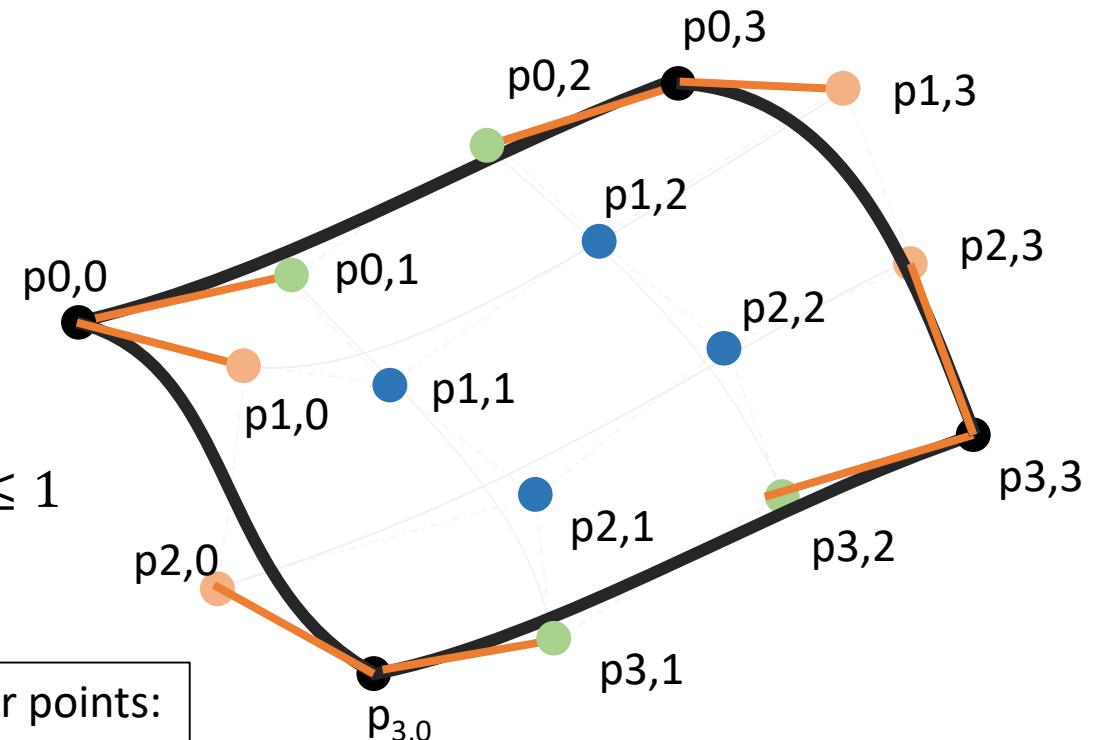


Interpolation and derivatives

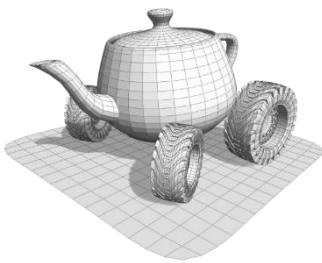
- Bezier patches interpolate on the end points $(0,0)(0,n)(n,0)(n,n)$

- The derivatives $\frac{d}{ds} f(s, t)$ is:

$$\frac{d}{ds} f(s, t) = \sum_{j=0}^n B_{j,n}(t) \underbrace{\frac{d}{ds} \sum_{i=0}^n \mathbf{p}_{i,j} B_{i,n}(s)}_{\text{Derivative of the Bezier curve for points: } \mathbf{p}_{i,0}, \mathbf{p}_{2,0}, \dots, \mathbf{p}_{n,0}} \quad 0 \leq s, t \leq 1$$



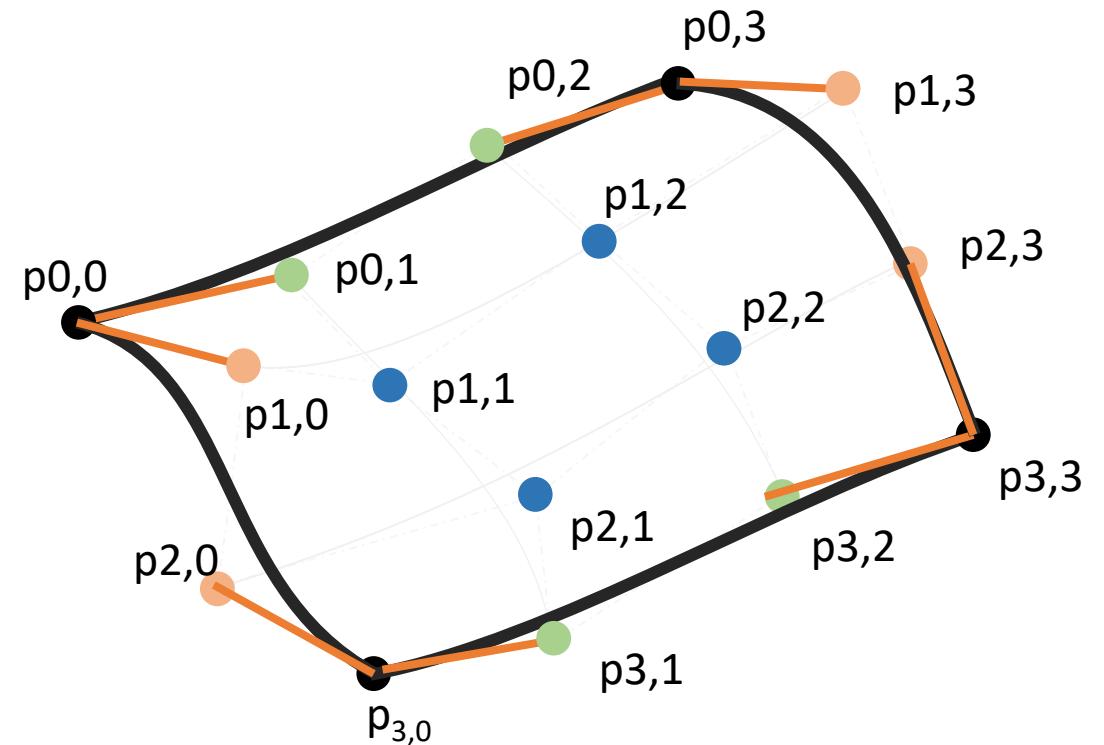
- Same for $\frac{d}{dt} f(s, t)$

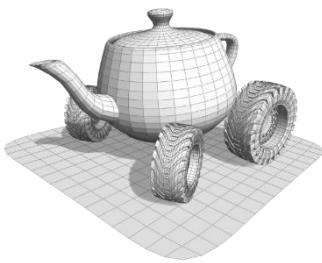


Normal

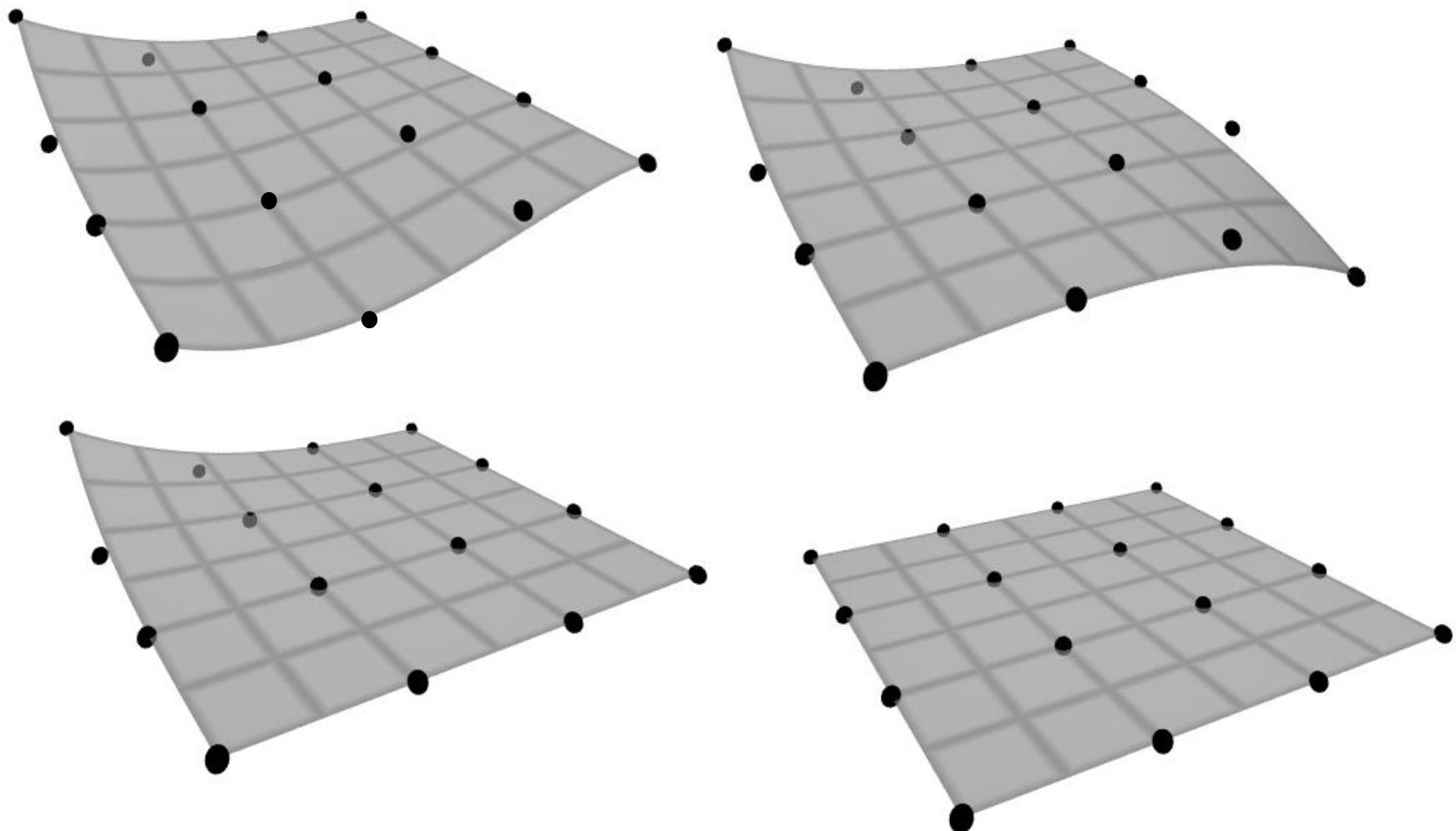
- Knowing the derivatives along s and t, the normal is found as their cross product

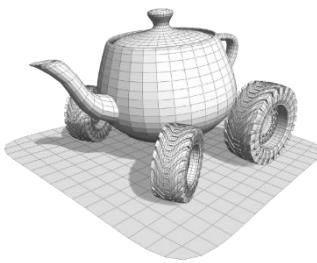
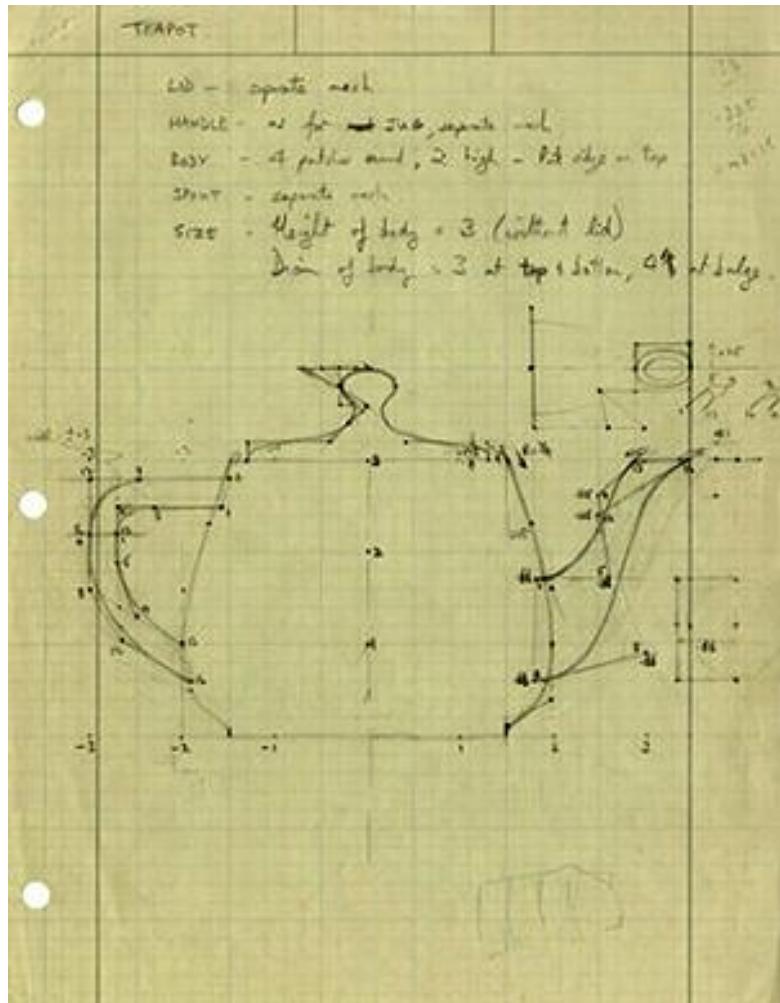
$$N(s, t) = \text{normalize} \left(\frac{d}{ds} f(s, t) \times \frac{d}{dt} f(s, t) \right)$$



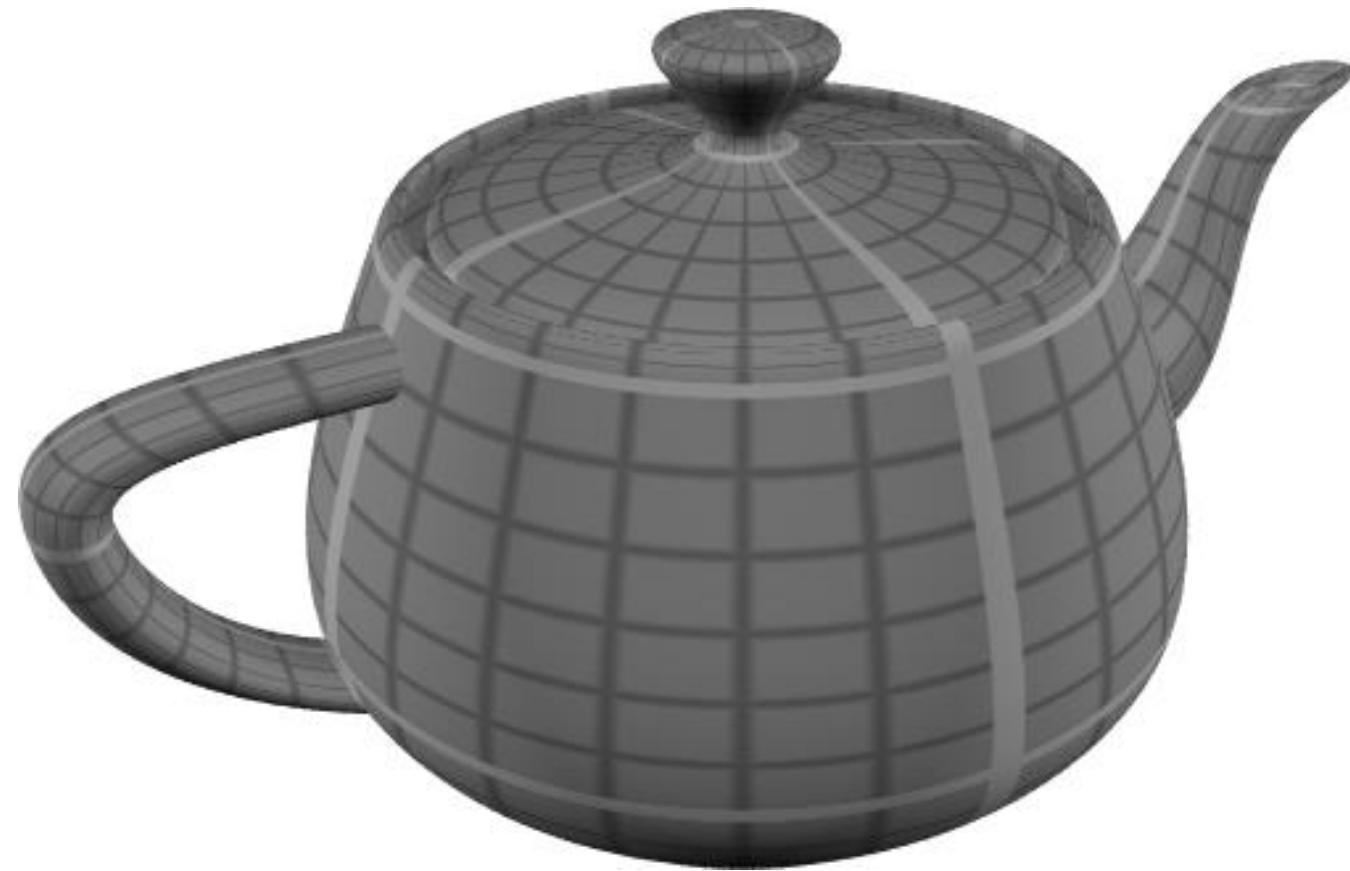
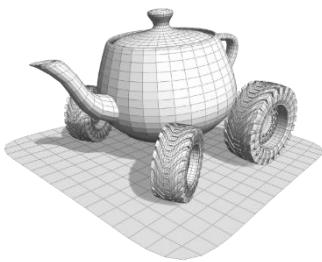


Bicubic Bezièr patches: examples





The Utah Teapot, by Martin Newell (1975)

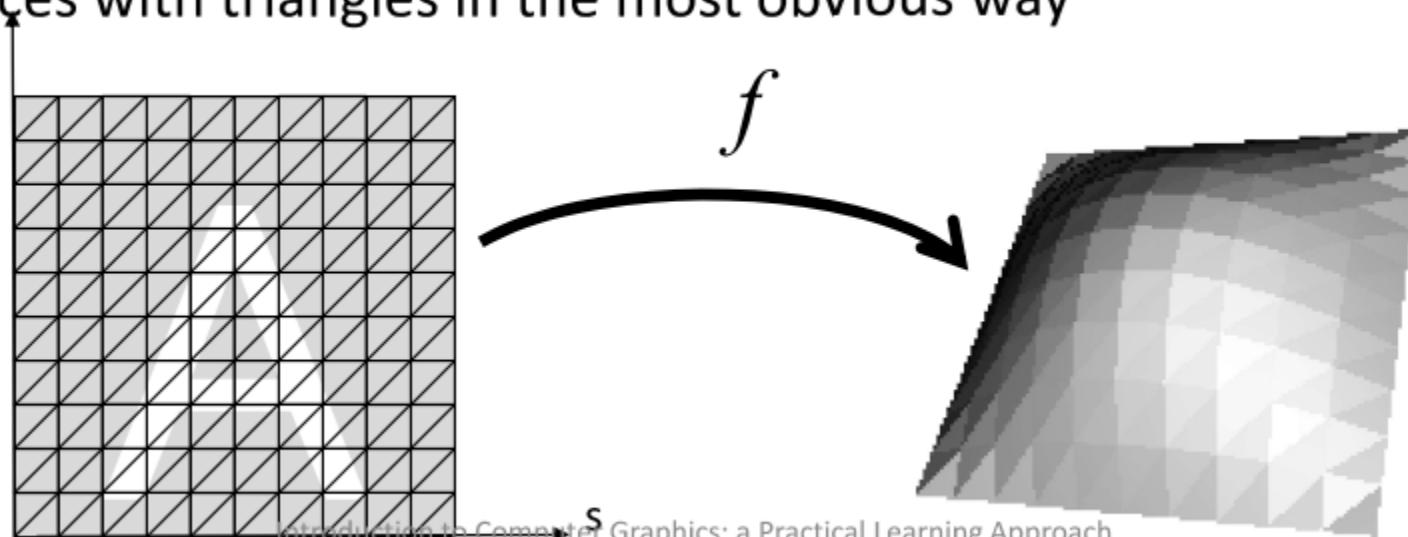


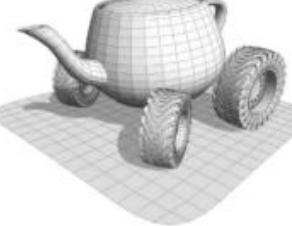
The Utah Teapot, by Martin Newell (1975)



From parametric surfaces to meshes

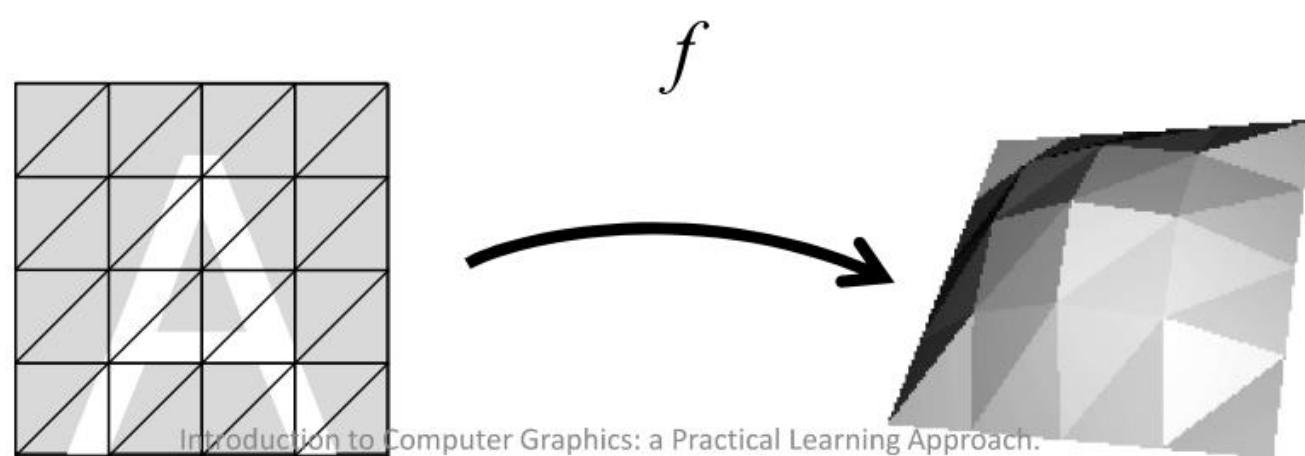
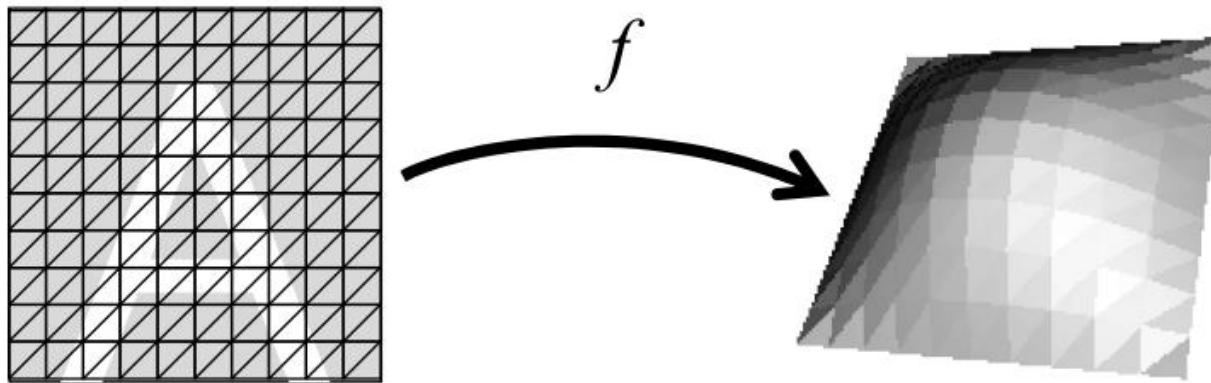
- We can sample in the parameter domain (eg: 10x10)
 - $(s,t) = (0.0,0.0), (0.1,0.0), \dots (0.0,0.1), (0.1,0.1) \dots, (1.0, 1.0)$
- For each sample (s,t)
 - Create a vertex of the mesh in $f(s, t)$
 - Create a normal vector for each vertex by deriving f (along s and t and then cross product)
- Connect vertices with triangles in the most obvious way

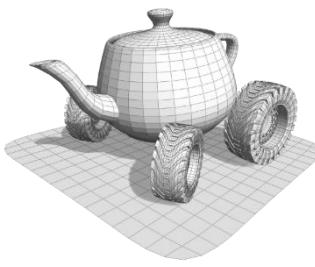




From parametric surfaces to meshes

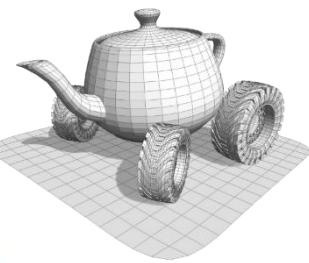
- Note: we can decide the fineness of the tessellation





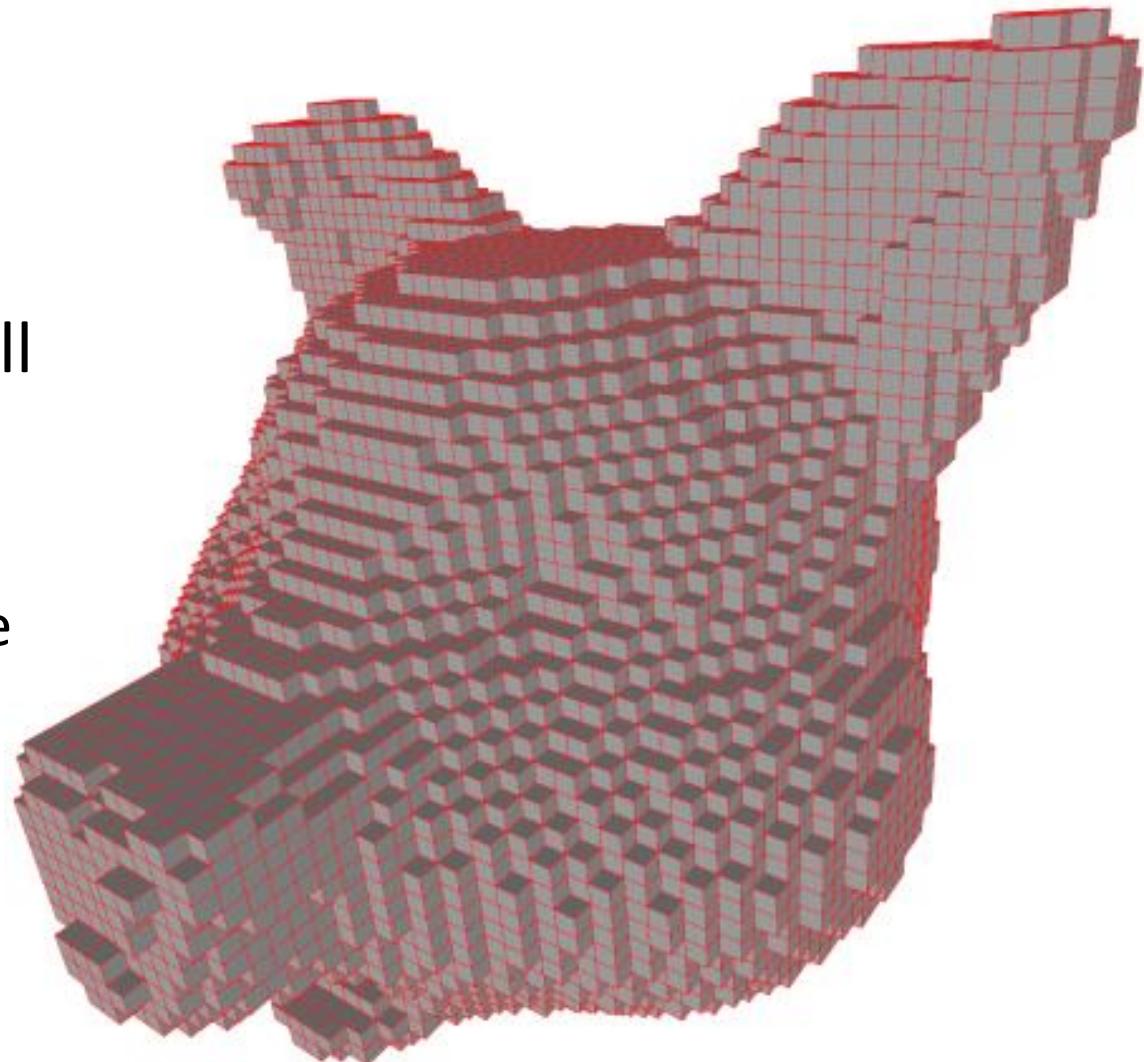
Types of 3D digital models

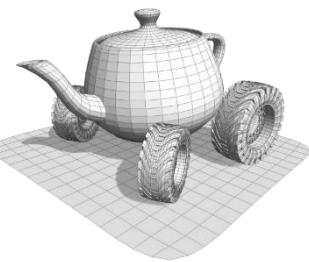
- Surfaces
 - Implicit surfaces
 - Parametric surfaces
 - Polygon meshes
- Volumetric
 - Voxelization
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes
- Points



Representing 3D objects with voxels

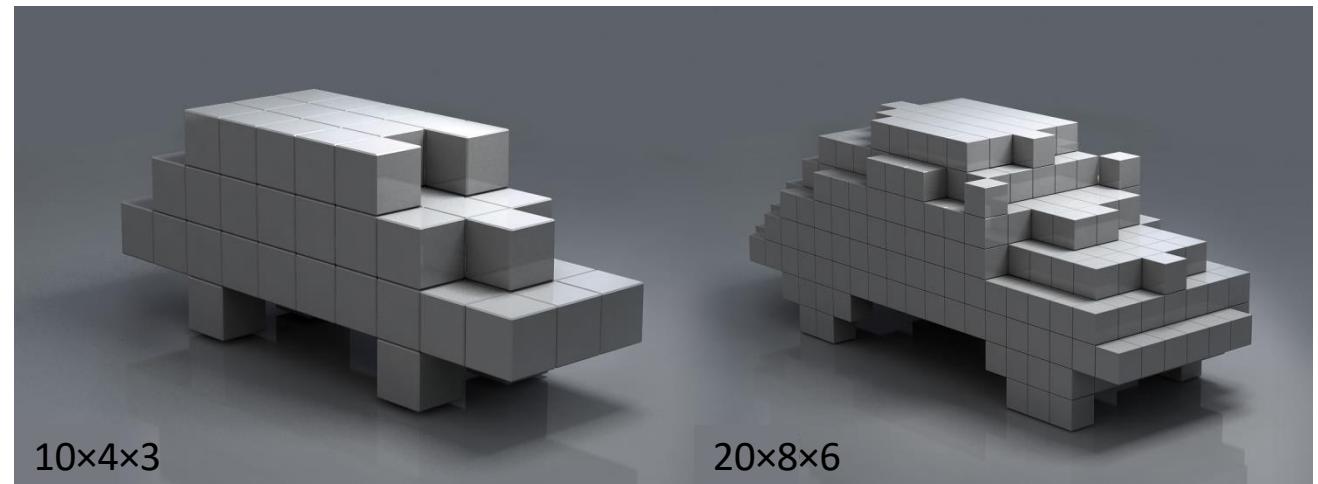
- **Boolean:** each voxel can be either full (1) or empty (0)
- The 3D object is the set of full voxels
- Cost: $\text{size}_X \cdot \text{size}_Y \cdot \text{size}_Z$
 - It grows cubically with the size

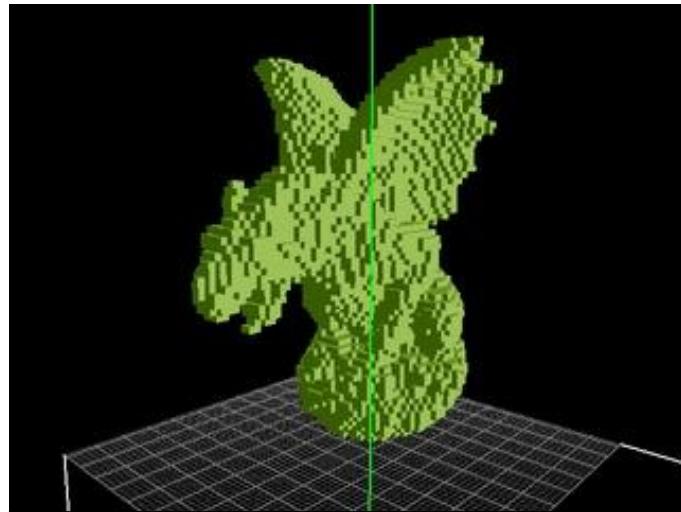
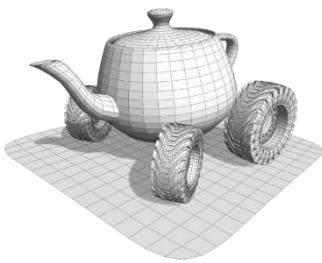




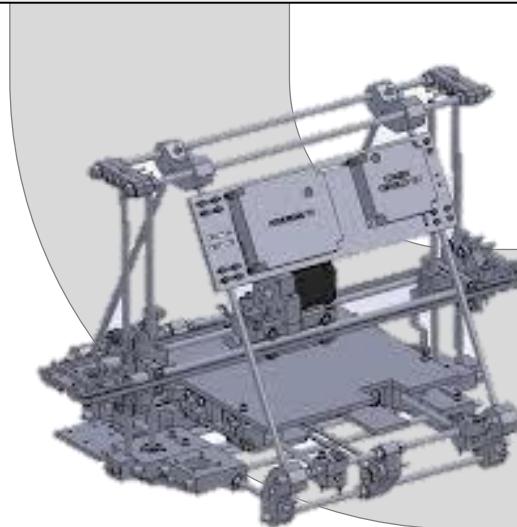
Models at varying resolution

- The higher the resolution, the more detailed the model
- ..the higher the memory occupation





Voxelized dataset
Boolean Voxels



Chapter 3: How a 3D Model is
Represented

3D printer

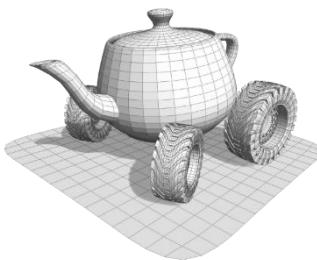
Introduction to Computer Graphics: a Practical Learning Approach.

Input taken by some
additive 3D printers



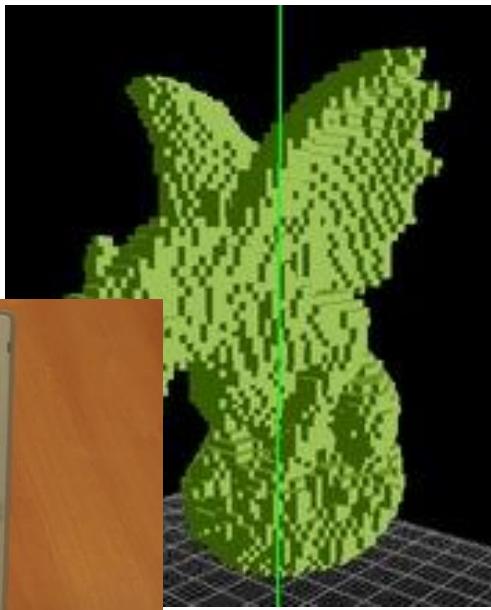
Printed objects

Creating Voxelized Volumes



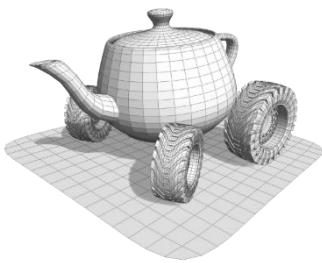
Triangle
2-manifold
and cl

Voxelization



zed volume
bit \times voxel

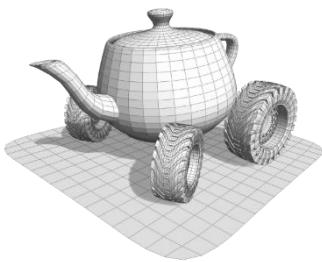
to printer



Voxelized Volumes:

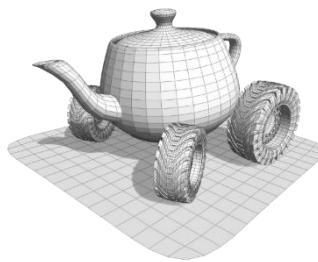
- Input naturale dei **3D printing devices**





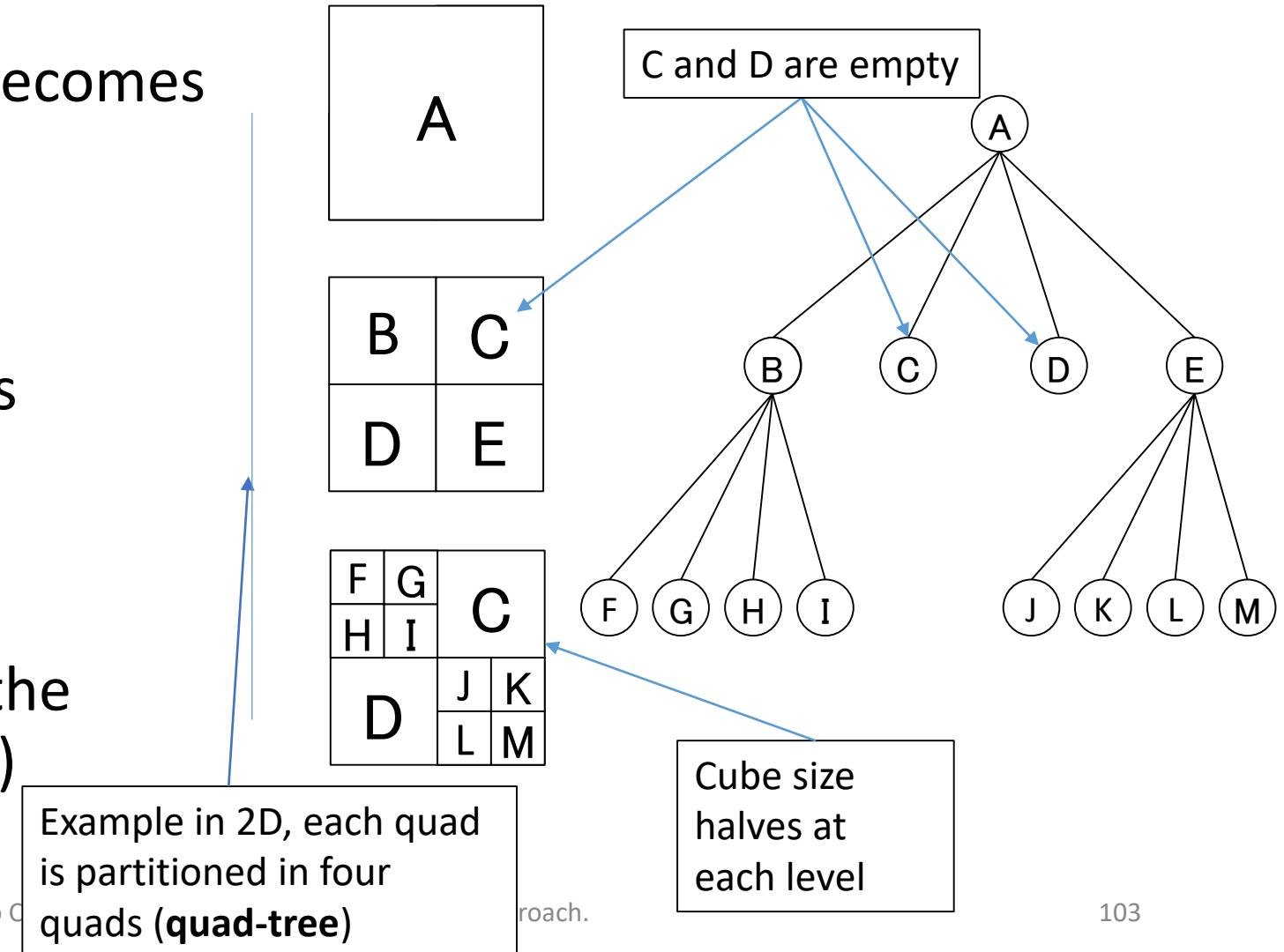
Probably know example: Minecraft™

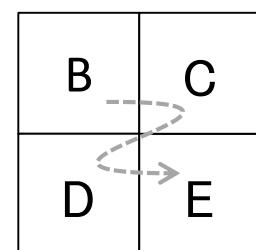
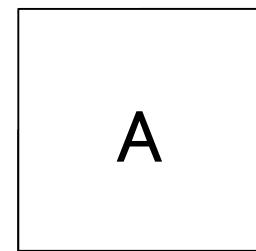
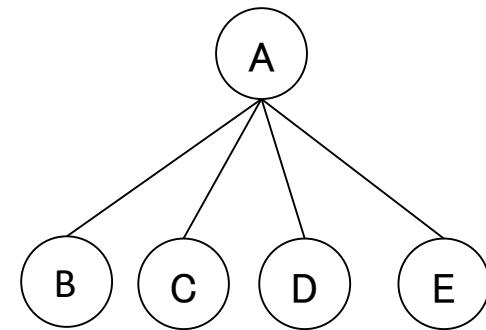
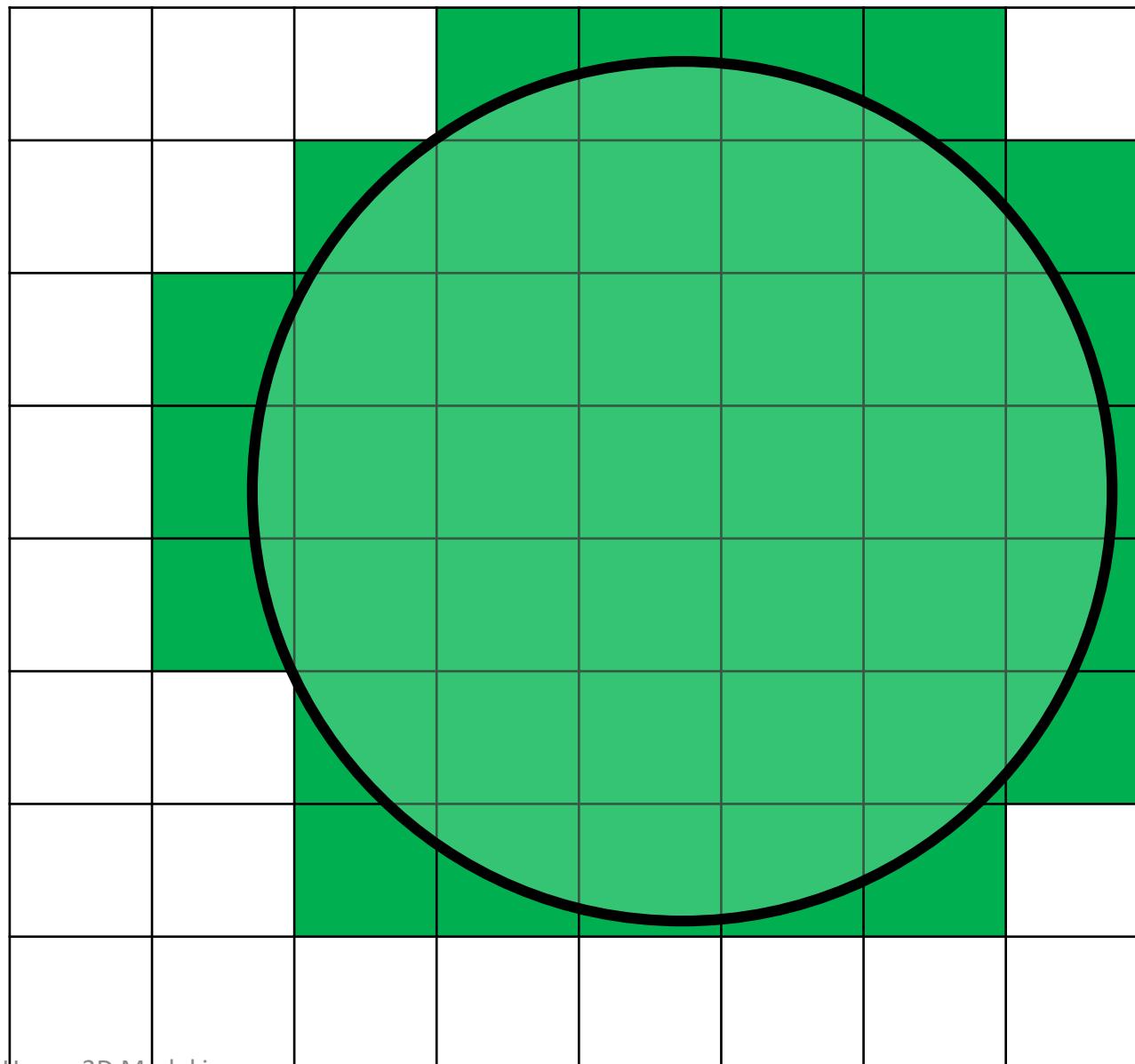
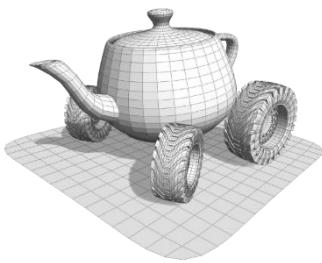


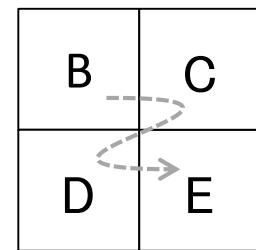
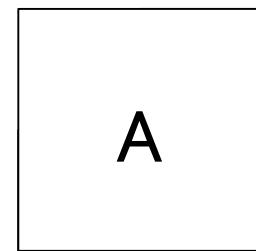
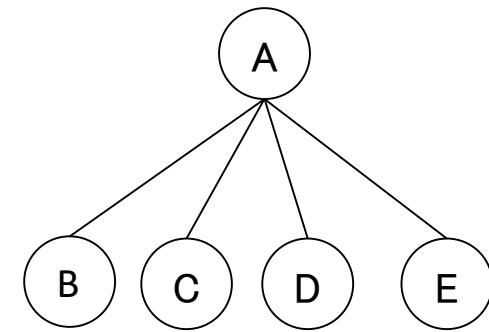
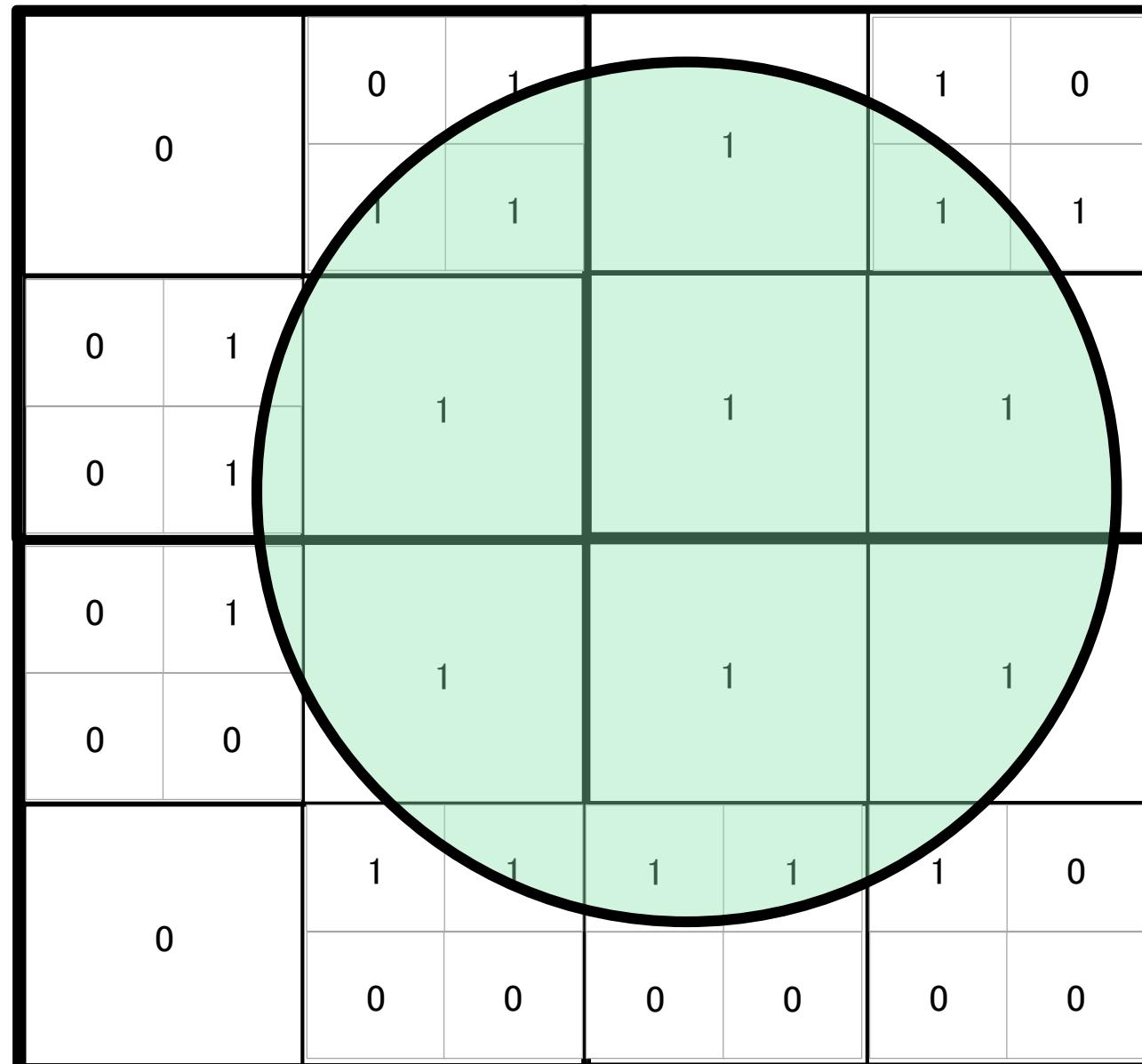
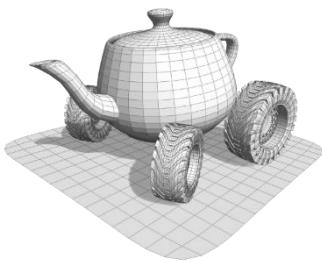


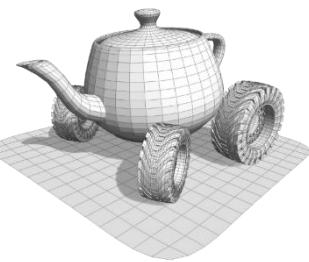
Efficient encoding of voxel models

- Flat-brute storage of data becomes easily unmanageable
- Hierarchical representation the volume is recursively partitioned in 8 subvolumes
 - Recursion stops on empty subvolumes
 - Predefined maximal depth
- The cost is quadratic with the resolution (instead of cubic)

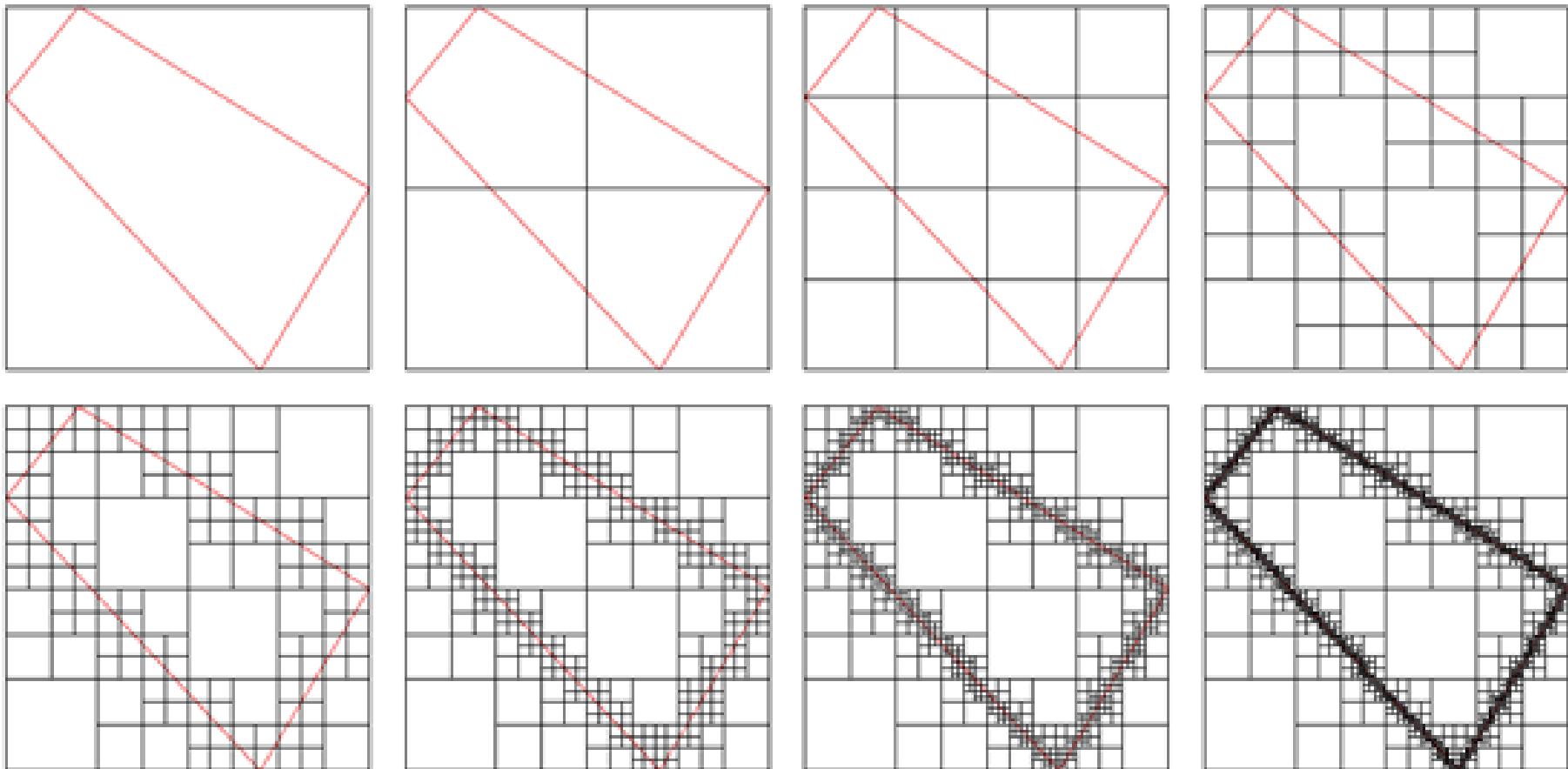


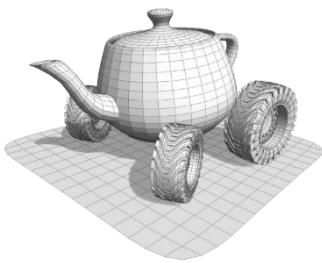




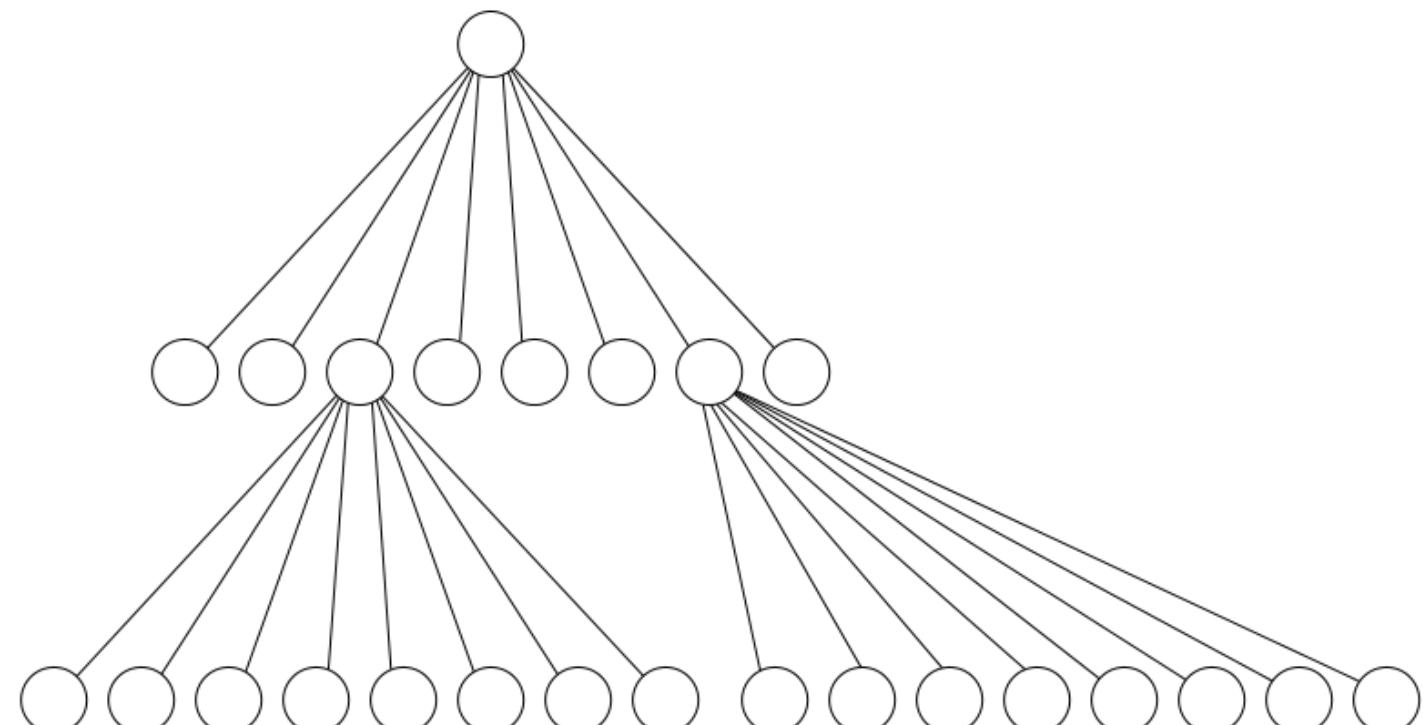
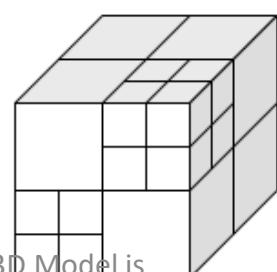
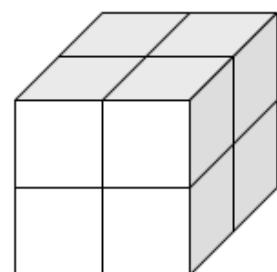
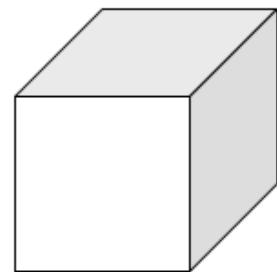


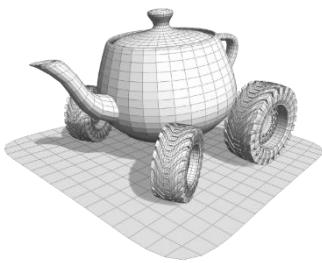
Another example



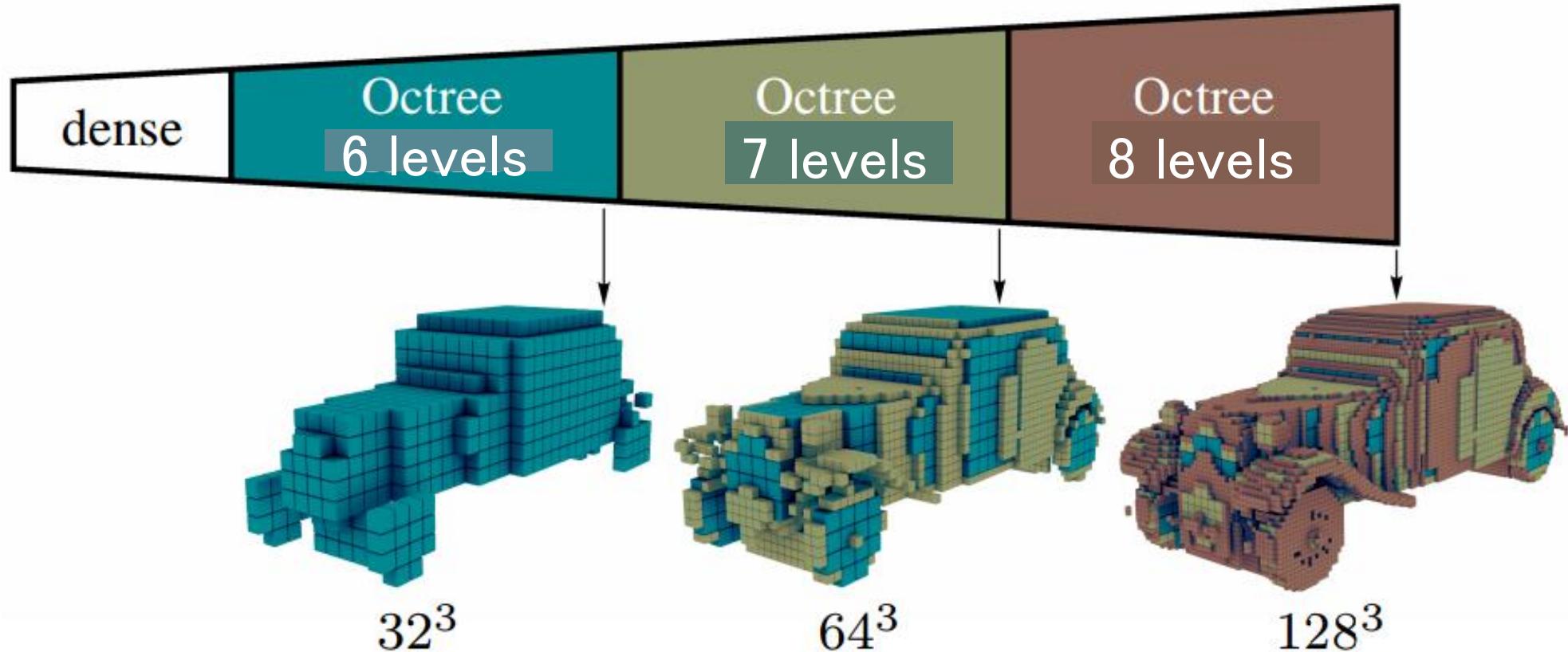


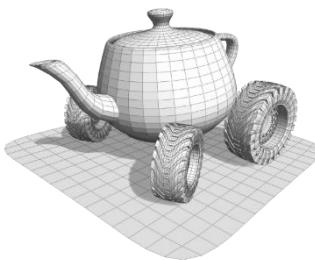
Oct-tree





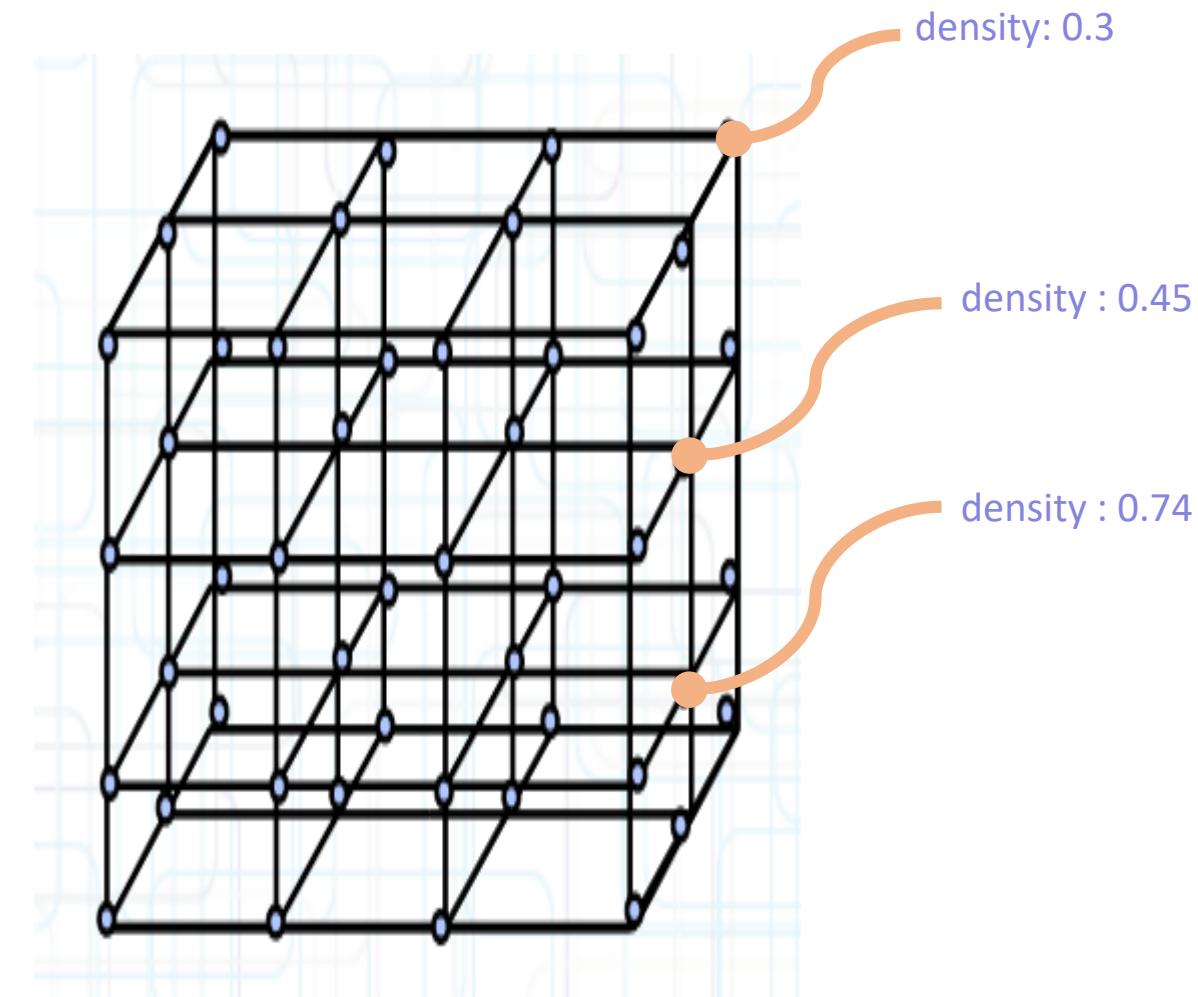
Levels of detail with octree

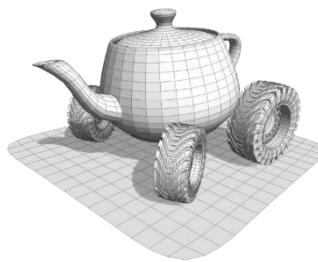




Representing 3D objects with voxels

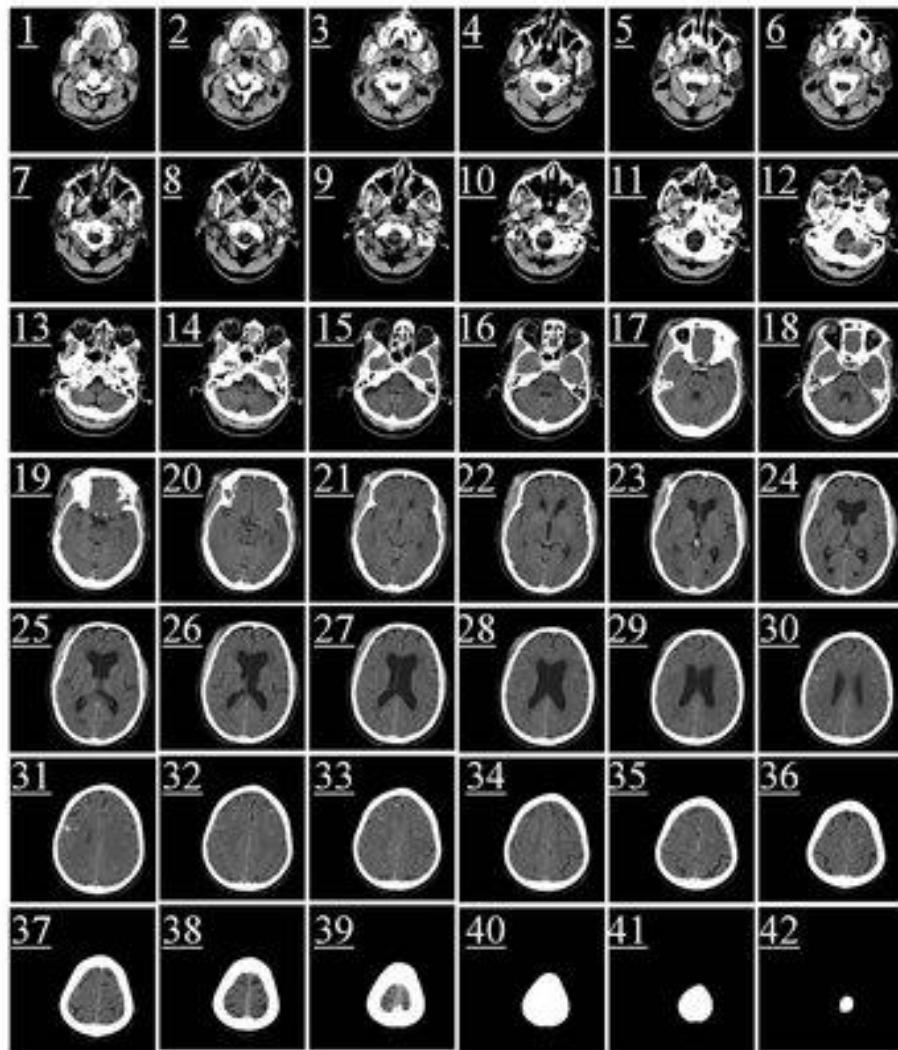
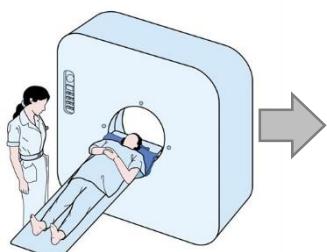
- **float:** each voxel stores a float, which can be, for example:
 - A density value
 - A temperature
- Note: values are associates with the “center” of each cube or (dually) at the corners

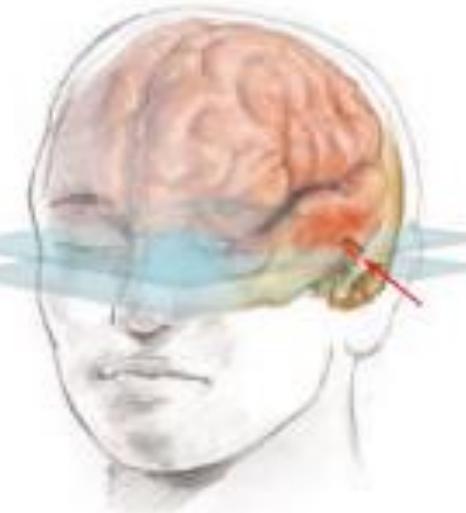
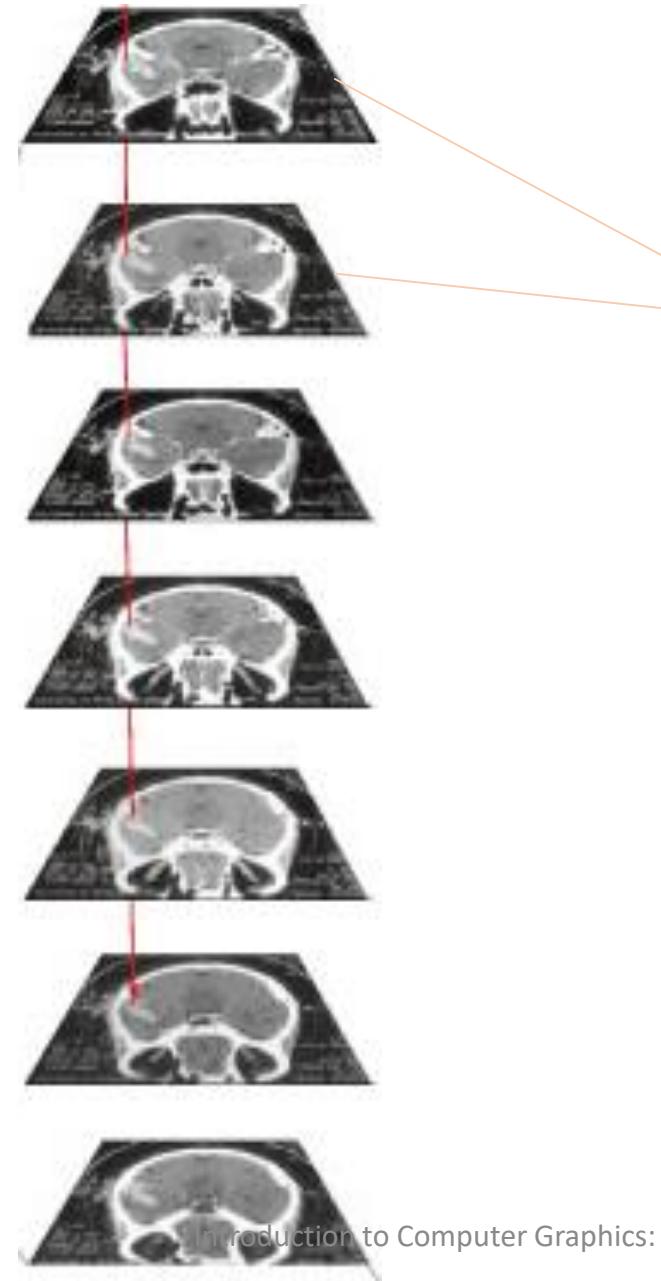
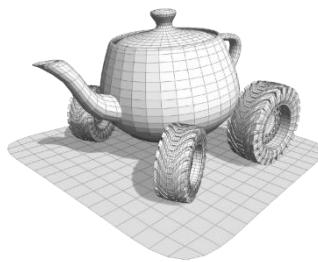




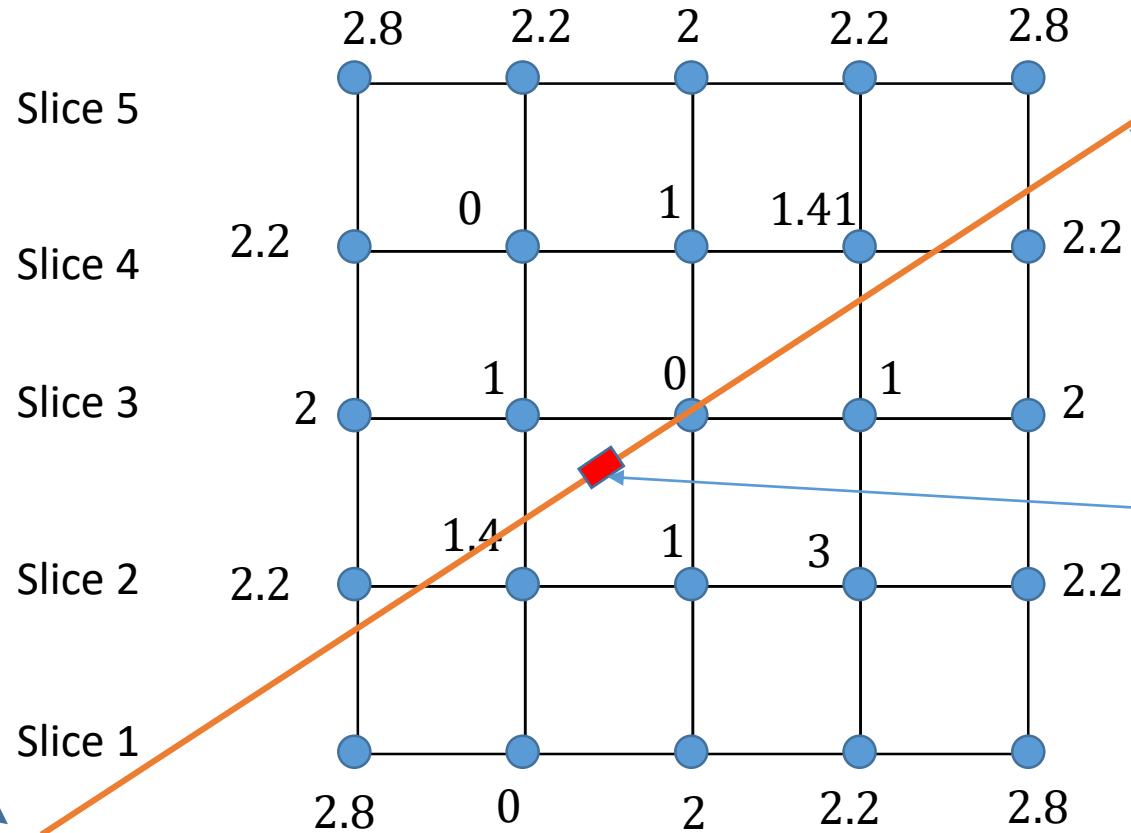
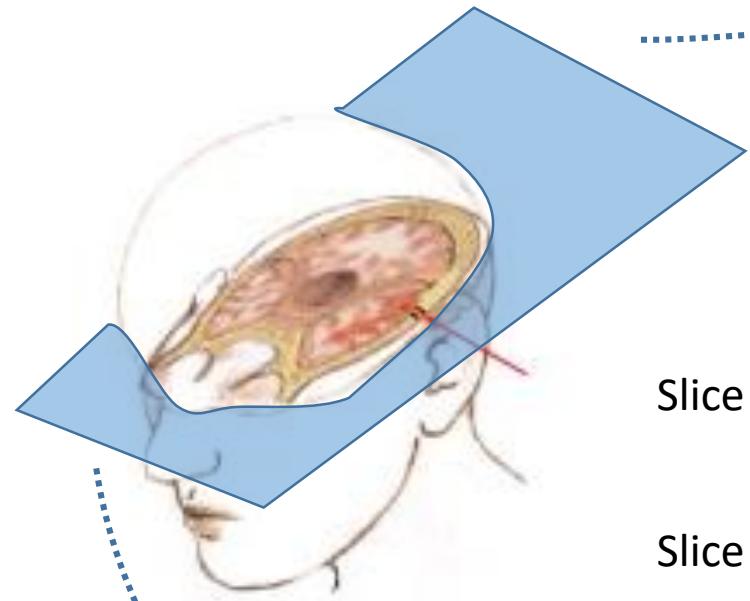
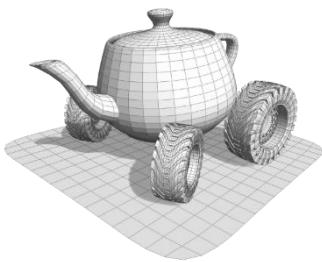
CT scans output

- Computerized Tomography output a series of 2D images
- Each 2D images is a slice of the scanned volume
- Stacking the images we have a float voxel model





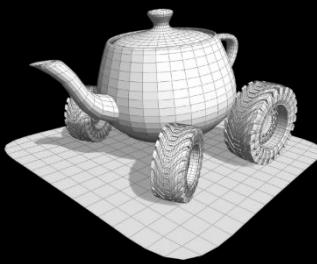
This slice is not
one of the
originals!



A raster image made with values of the intersected voxels



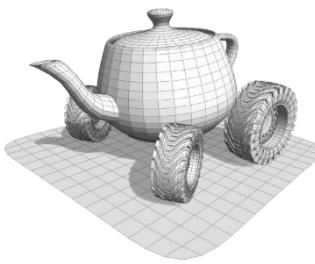
Volume ray tracing



arbitrary slice

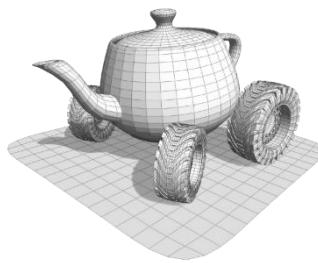
direct volume
rendering

Types of 3D digital models: recap

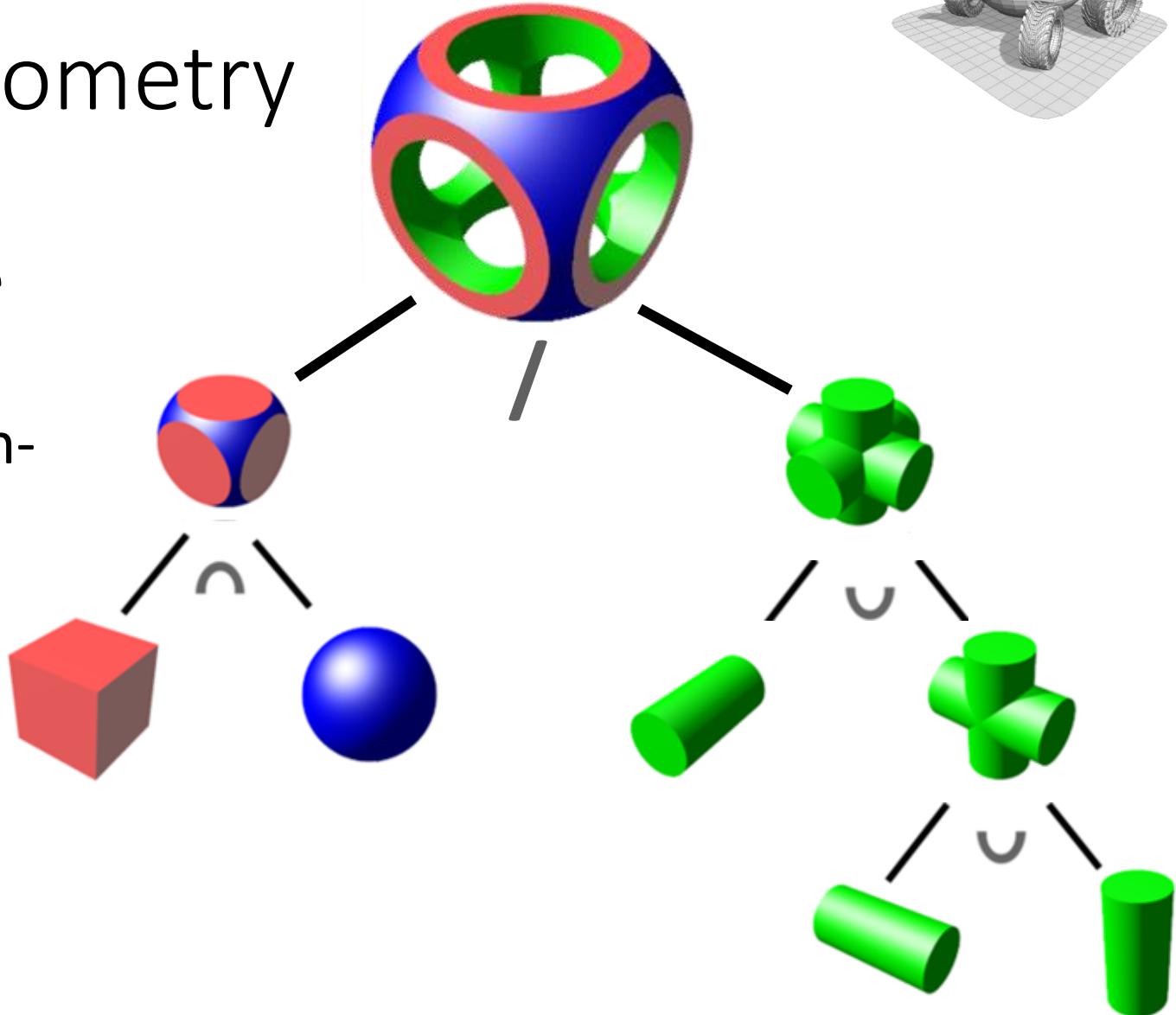


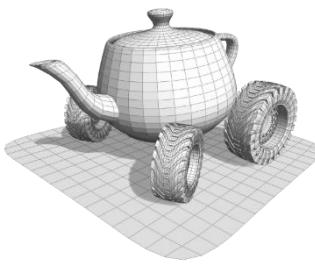
- Surfaces
 - Implicit surfaces
 - Parametric surfaces
 - Polygon meshes
- Volumetric
 - Voxelization
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes
- Points

Constructive Solid Geometry



- CSG defines a 3D model as a combination of simple primitive 3D objects
- Naturally represented with a bin-tree structure
 - root = the whole object
 - Internal nodes = object resulting from a boolean operation of its 2 childs
 - leaves = basic primitives
- Demo: OpenSCAD
(<https://openscad.org/>)





Types of 3D digital models: recap

- Surfaces
 - Implicit surfaces
 - Parametric surfaces
 - Polygon meshes
- Volumetric
 - Voxellization
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes
- Points

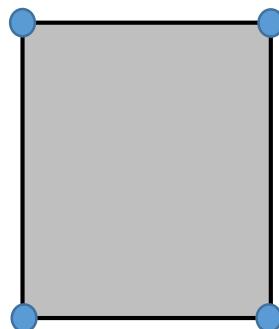
These are not the **only ways** to represent object.
They the way objects are commonly represented in CG

Polyhedral meshes

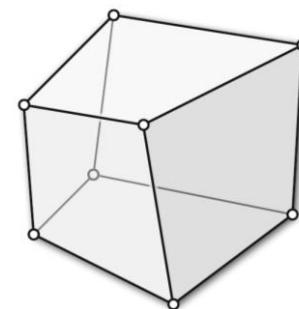
- The natural extension of surface meshes
- Typically made of two types of primitives:
 - **Hexahedra**, i.e. the 3D extension of the quadrilateral
 - **Tetrahedra**, i.e. the 3D extension of the triangle (also 3-simplex, the convex hull of 4 points)
- Very similar data structures

Check this out: www.hexalab.net

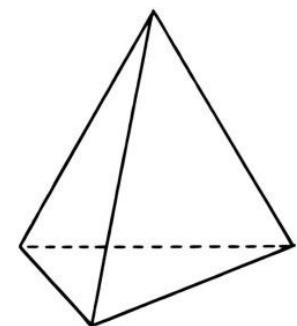
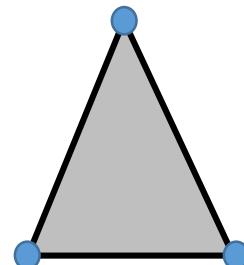
$$\mathbb{R}^2$$



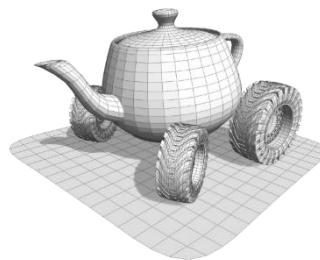
$$\mathbb{R}^3$$



hexahedra

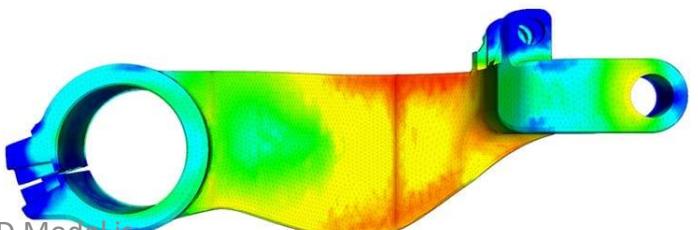


tetrahedra



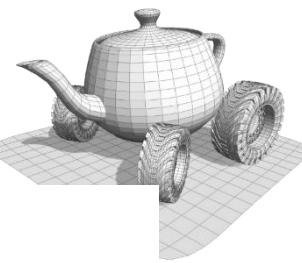
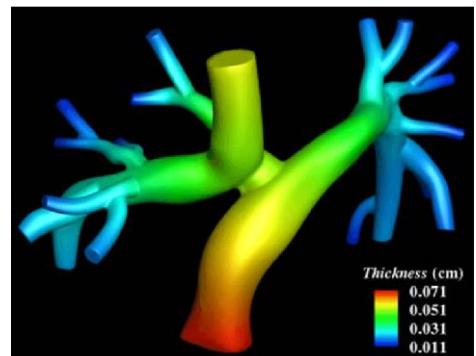
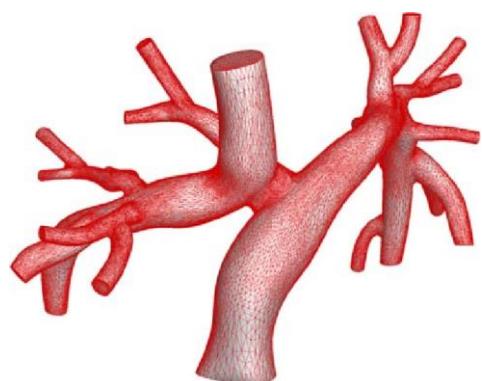
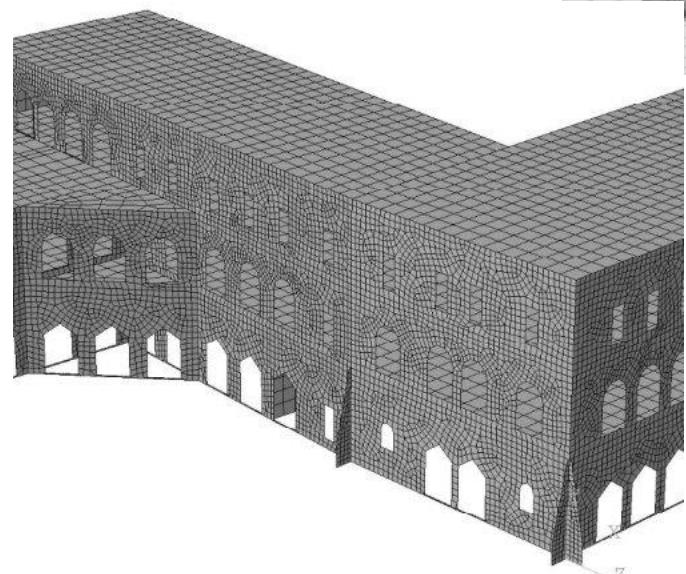
What are PM used for?

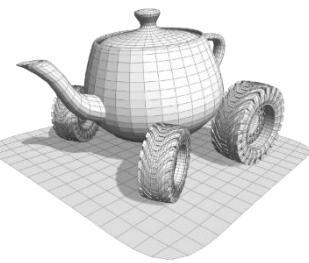
- Finite Element Method / Finite Element Analysis
 - Virtual verification of structural properties of solid objects
 - load simulation for buildings, bridges
 - Stress simulation for mechanical pieces
 - Dynamic simulation, e.g. fluid dynamics
 - virtual surgery, e.g. virtual cutting



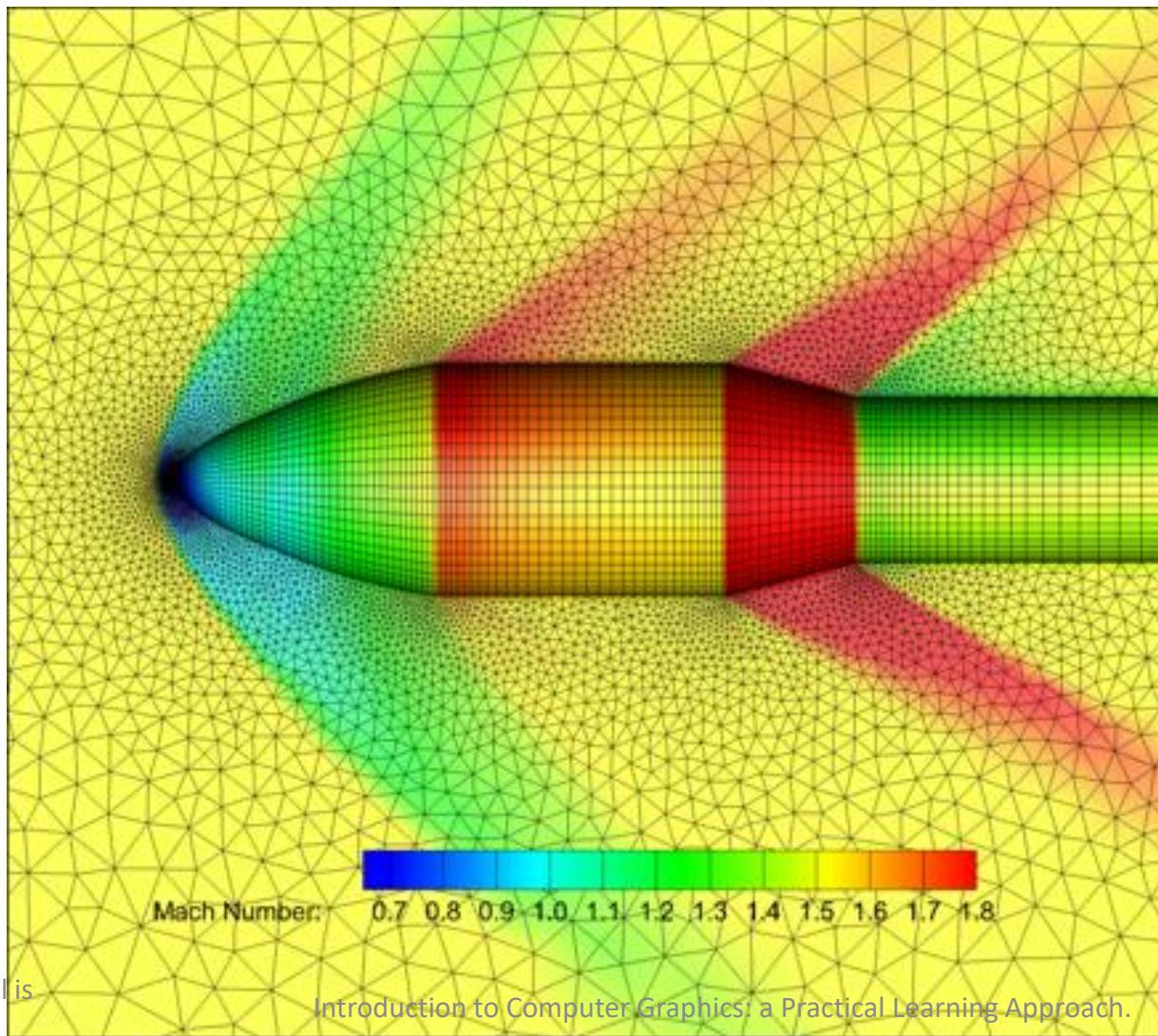
Chapter 3: How a 3D Model is Represented

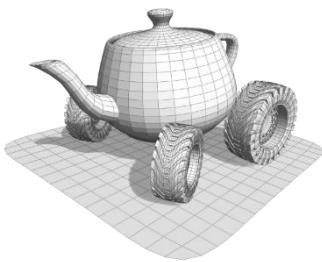
Introduction to Computer Graphics: a Practical Learning Approach.



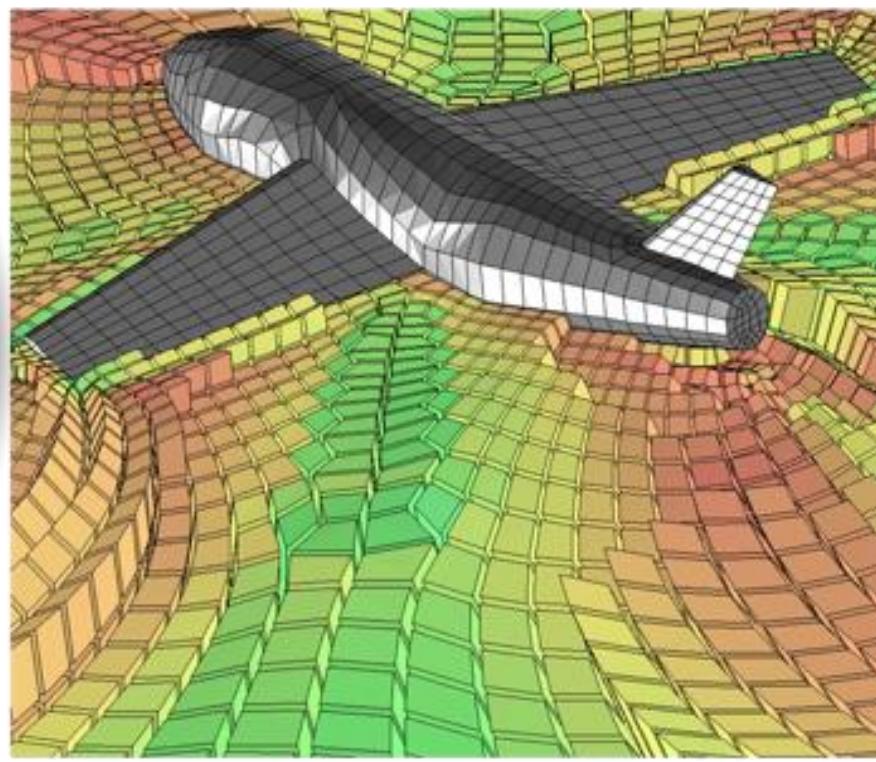
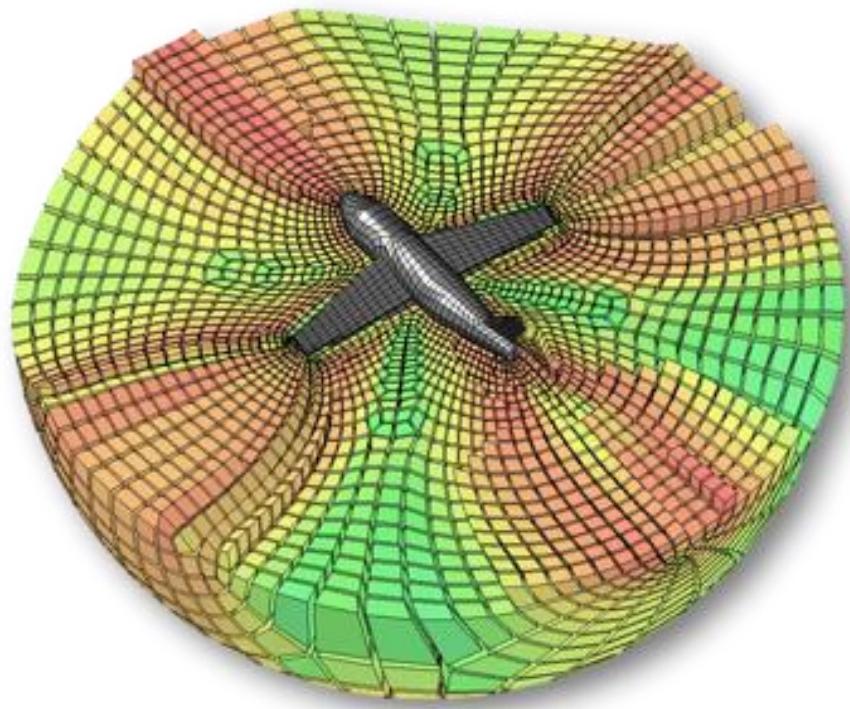


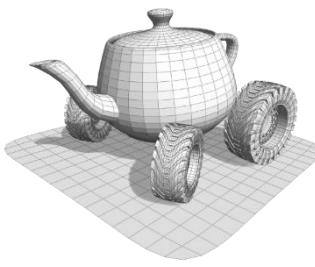
Example: Computational Fluid Dynamics





Example: Computational Fluid Dynamics

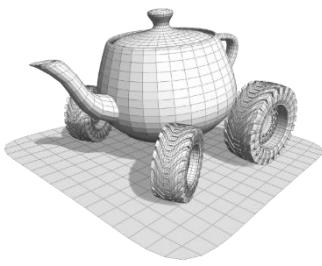




Types of 3D digital models: recap

- Surfaces
 - Implicit surfaces
 - Parametric surfaces
 - Polygon meshes
- Volumetric
 - Voxellization
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes
- Points

These are not the **only ways** to represent object.
They the way objects are commonly represented in CG



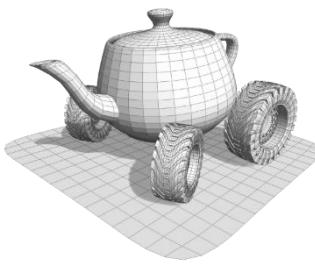
Point Clouds

- Collection of 3D surface samples
- Completely unstructured (that is, no adjacency information)
- For each sample:
 - position \mathbf{p} (x,y,z)
 - “normal” \mathbf{n} (x,y,z), that is the local surface orientation
 - other attributes such as color
- Characteristic depends also on how they are created
 - Maybe be very very large set
 - May have **noise** or **outliers**

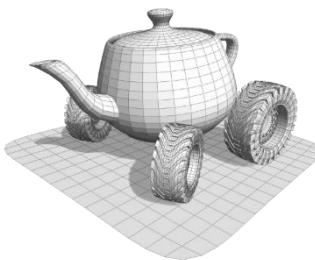
Point Clouds



Sources of point cloud



- 3D **active** scanners (informal): like a color camera but return distances instead of color
 - Laser scanners
 - Triangulation / Time of flight
 - Structured light scanners
 - Kinect (tm) like
- Sampling is regular and dense, similar to a photographs but without color
- Structure from Motion (SfM): a point cloud can be produced just from a set of normal photographs (**Computer Vision**)
 - Points also have color (from the images)



Point clouds from real world

Material properties may not be ideal for active scanning



(a) Original shape



(b) Nonuniform sampling



(c) Noisy data



(d) Outliers

3D active scanning:
A single «take» usually does not capture the whole surface. They must be re-aligned



(e) Misaligned scans

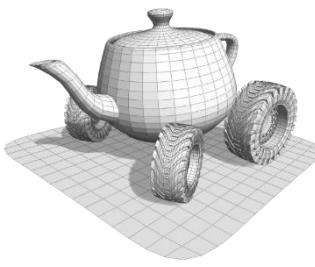


(f) Missing data

Very typical of SfM.
Only points with distinctive color features are created

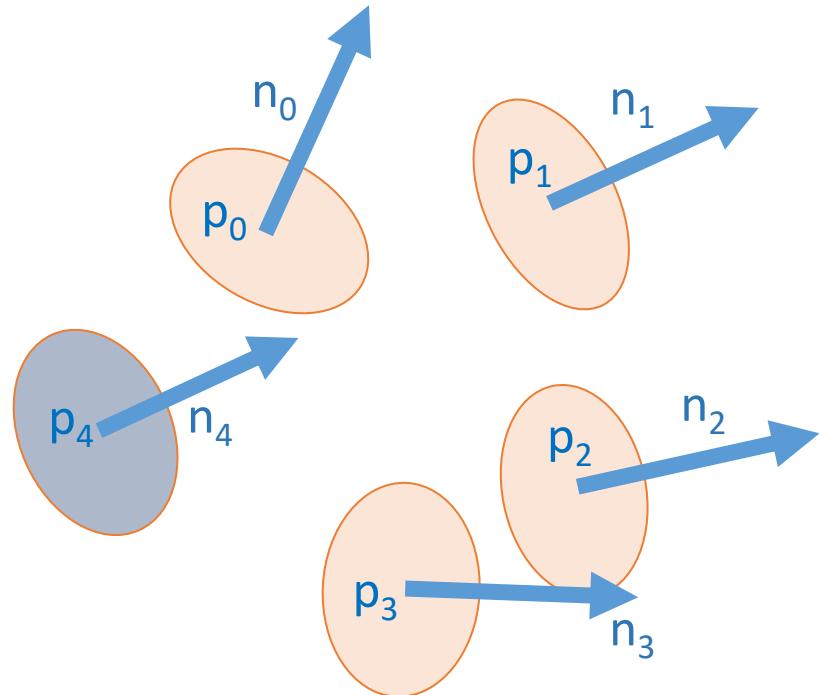
Very typical of SfM.

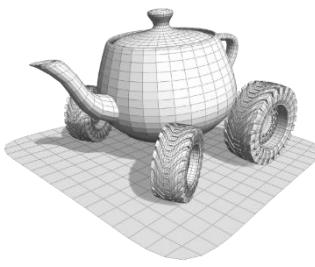
Berger et al. / A Survey of Surface Reconstruction from Point Clouds



Rendering point cloud: surfel

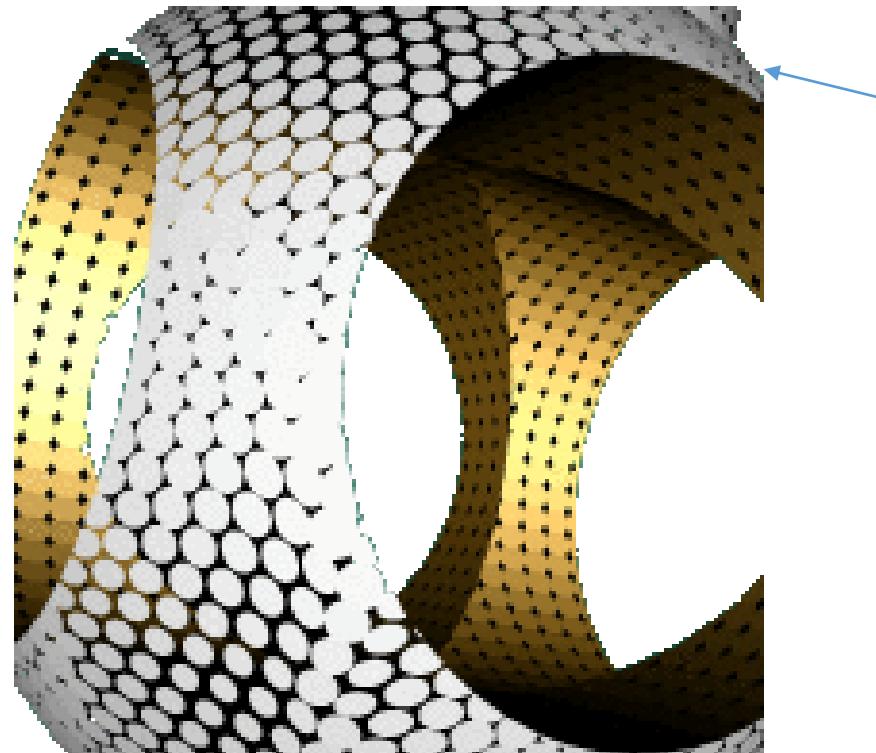
- **Surfel (surface element):** description of a small piece of surface as:
 - Position
 - Normal
 - Color
 - Disk size
- In other words, an enhanced point



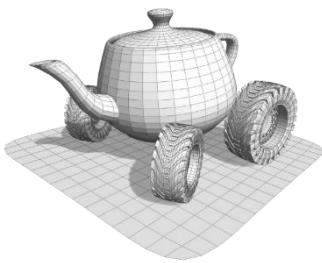


Rendering point clouds

- Disk size varies with the local point density, (possibly) the distance from the observer, the local curvature



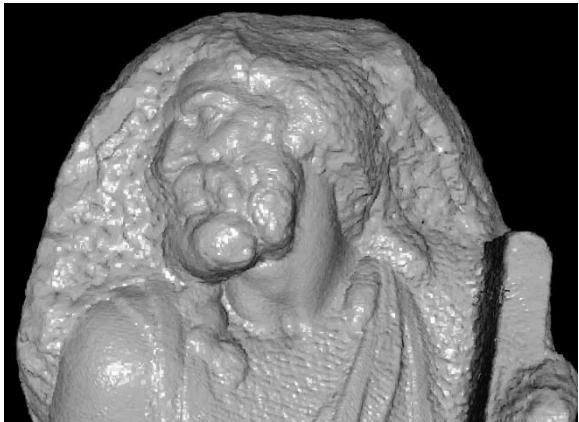
This specific model is not obtained as a point cloud from the real world. It is a sampling of a surface mesh



Example: point based rendering



131,712 splats



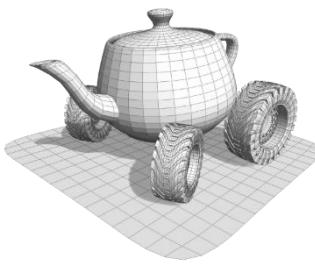
259,975 splats



1,017,149 splats

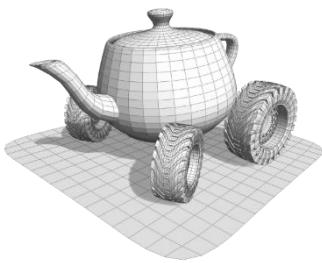


14,835,967 splats



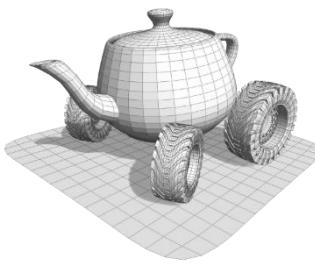
Point Clouds: pros and cons

- Cheap & easy to capture (like with your smartphone)
- Fairly easy to render on the GPU
 - Just as point primitives
 - As small disk centered at each point and covering a small area
- from a mathematic point of view:
do **not** define a surface
 - e.g. not easy to determine volume, or inside/outside status of a given position, etc.
- from an algorithmic point of view:
do **not** define a graph
 - e.g. not easy to define neighbors of a sample
(which is needed by most algorithms on surfaces)
- Not all models are easy to render
 - Noise, irregular sampling



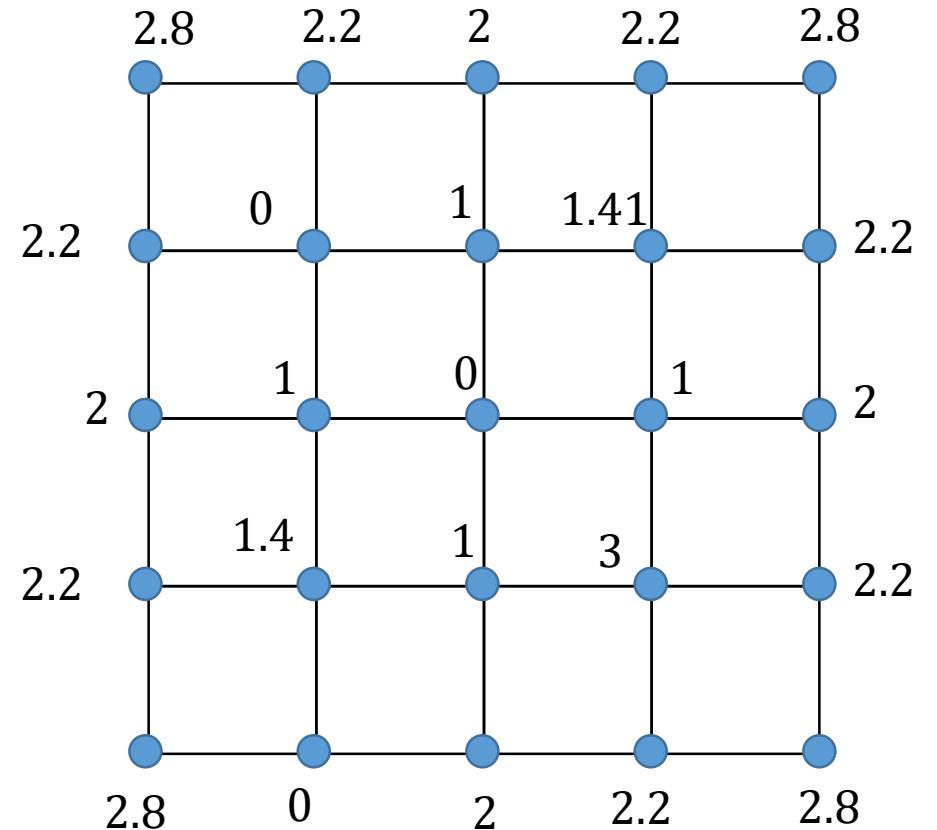
Rendering 3D models

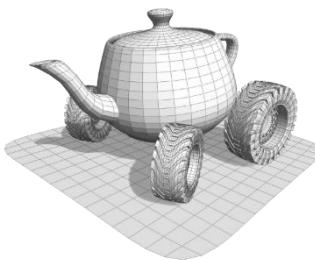
- Surfaces
 - Implicit surfaces
 - Parametric surfaces: sample in parameters space, create triangle (or quads)
 - Polygon meshes: just draw them!
- Volumetric
 - Voxel based:
 - Boolean: they are just cubes
 - Float: volume ray tracing
 - Constructive Solid Geometry
 - Hexahedral/Tetrahedral Meshes: their surface is by definition a polygon mesh
- Points
 - Point clouds: point based rendering (splats/surfel)



Turning scalar fields into polygon meshes

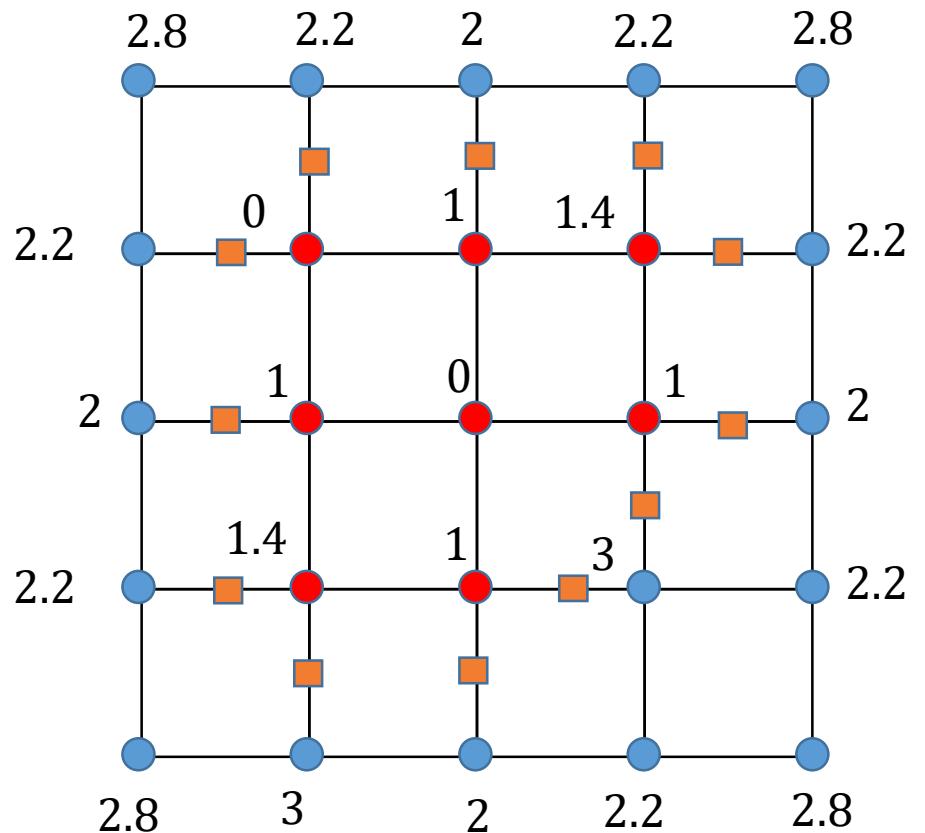
- Consider a regular sampling of a scalar field $f(x, y)$
- Connect the sampled (x, y) points to form a grid
- Assume that the field varies **linearly** along the edge between neighbors points
- Which edges will be crossed by the isosurface with $\alpha = 1.5$?

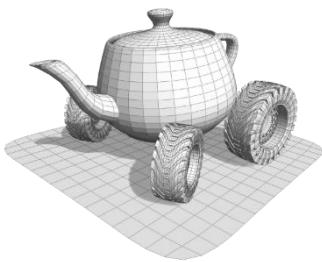




Turning scalar fields into polygon meshes

- Consider a regular sampling of a scalar field $f(x, y)$
- Connect the sampled (x, y) points to form a grid
- Assume that the field varies **linearly** along the edge between neighbors points
- Which edges will be crossed by the isosurface with $\alpha = 1.5$?
- Mark the nodes as:
 - Blue **iff** $f(x, y) > \alpha$
 - Red **iff** $f(x, y) < \alpha$





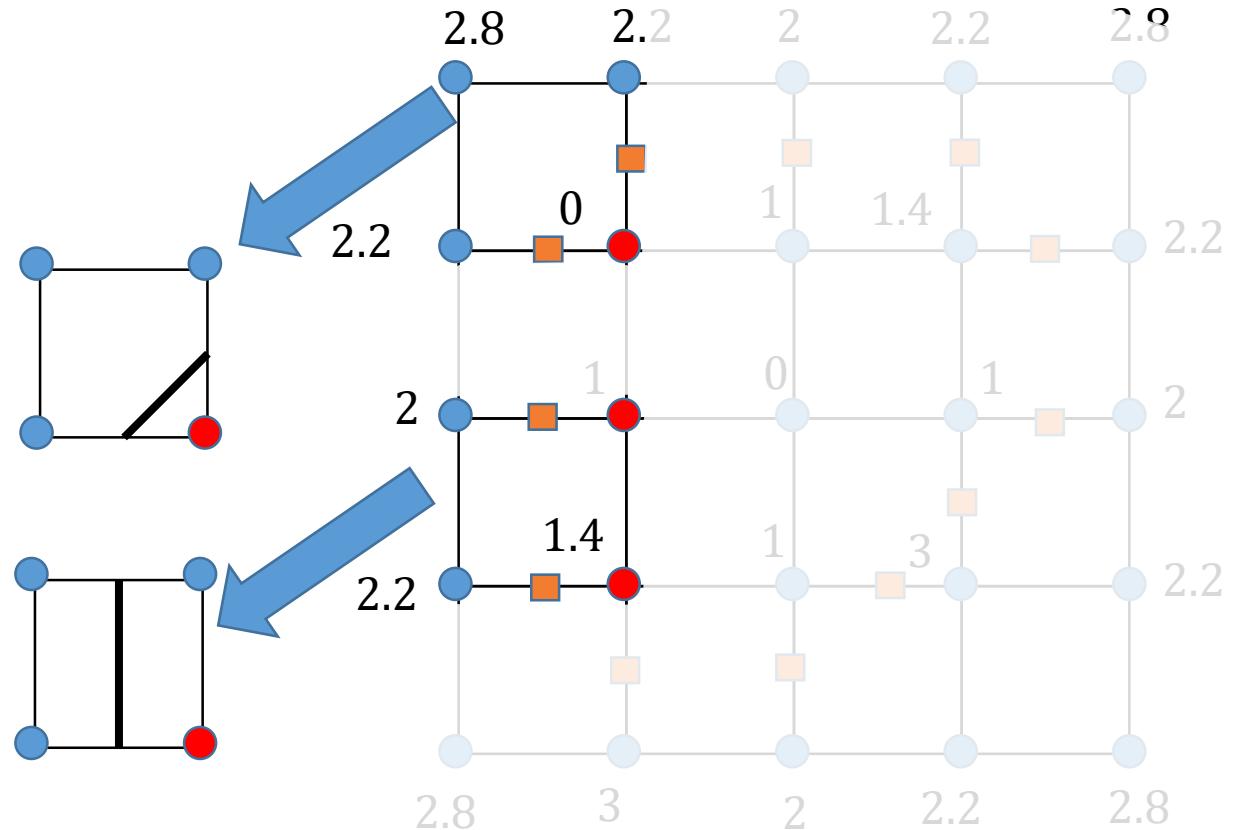
Marching square

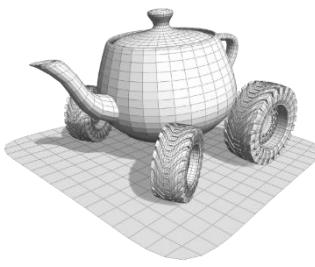
- **Marching square algorithm:**

for each cell of the grid

1. find the intersection of the isosurface with the edges

2. define a set of segments for each **permutation** of red/blue nodes





Intersections with the grid edges

- The exact position of the vertices (that is, the intersection points) are found by linear interpolation

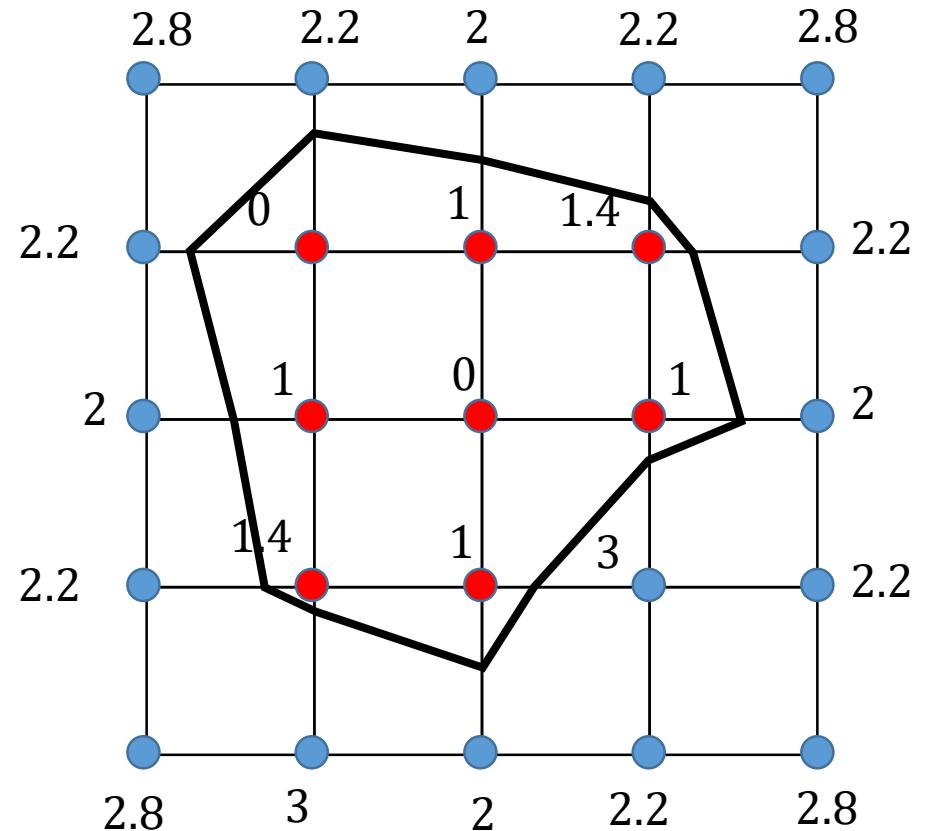
$$\exists t: f(\text{O})(1 - t) + f(\text{O})t = \alpha$$

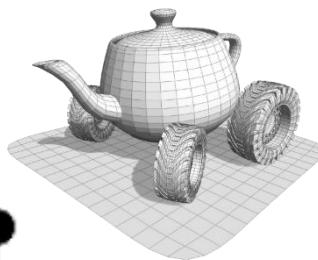
$$\Rightarrow t = \frac{\alpha - f(\text{O})}{f(\text{O}) - f(\text{O})}$$

$$\mathbf{p}(\text{O})(1 - t) + \mathbf{p}(\text{O})t = \mathbf{p}$$

Position of
the red node

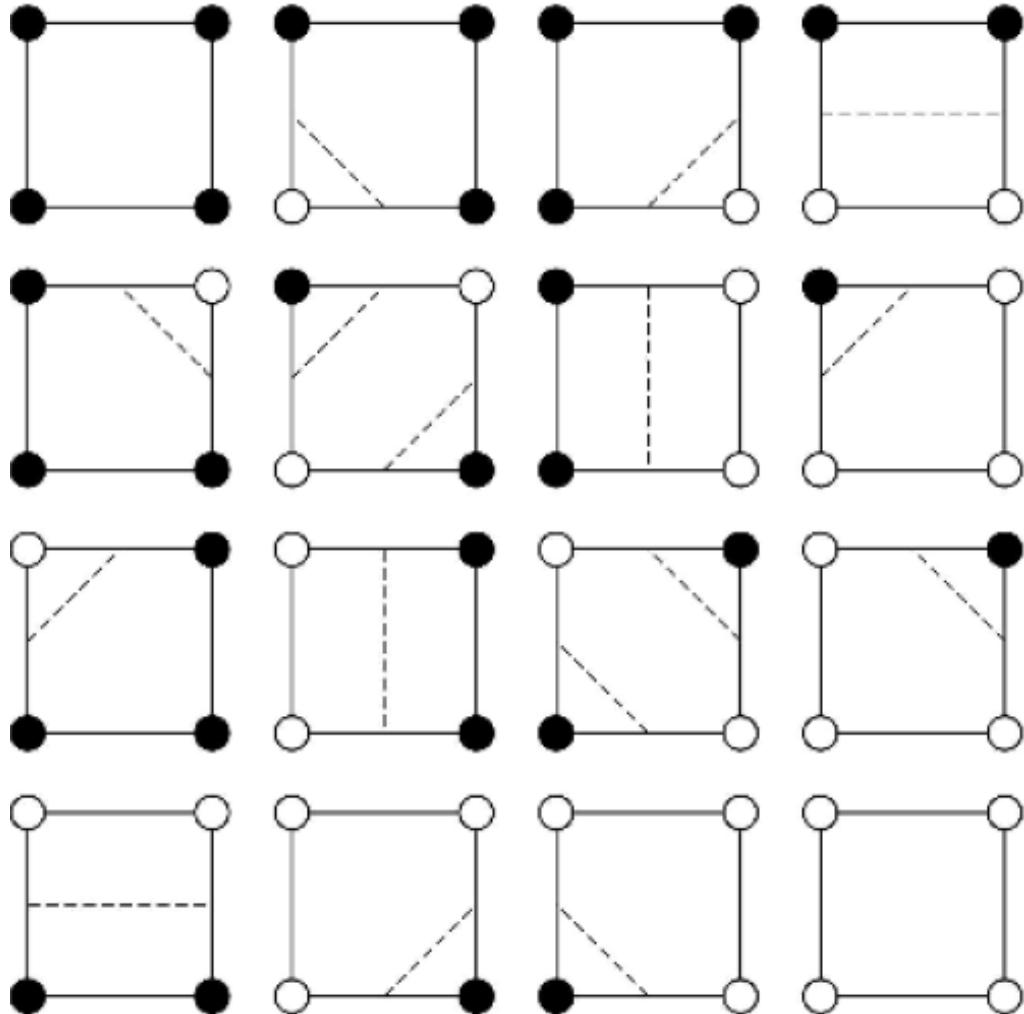
Position of
the blue node

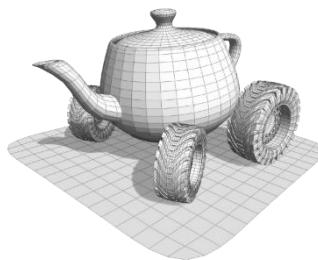




Look-up table

- Each node can be red or blue, so we have $2^4 = 16$ permutations
- A **Look-up table** is built once for all to encode, for each permutation, the set of segment to use





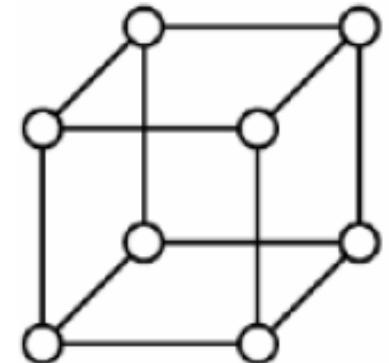
Marching cube (the real thing)

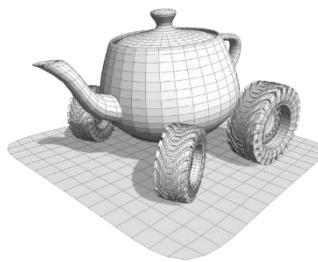
Marching square

- two dimensional grid
- Quadrilater cells (4 nodes)
- $2^4=16$ permutations of red/blue value
- Segments are created for each cell

Marching cube

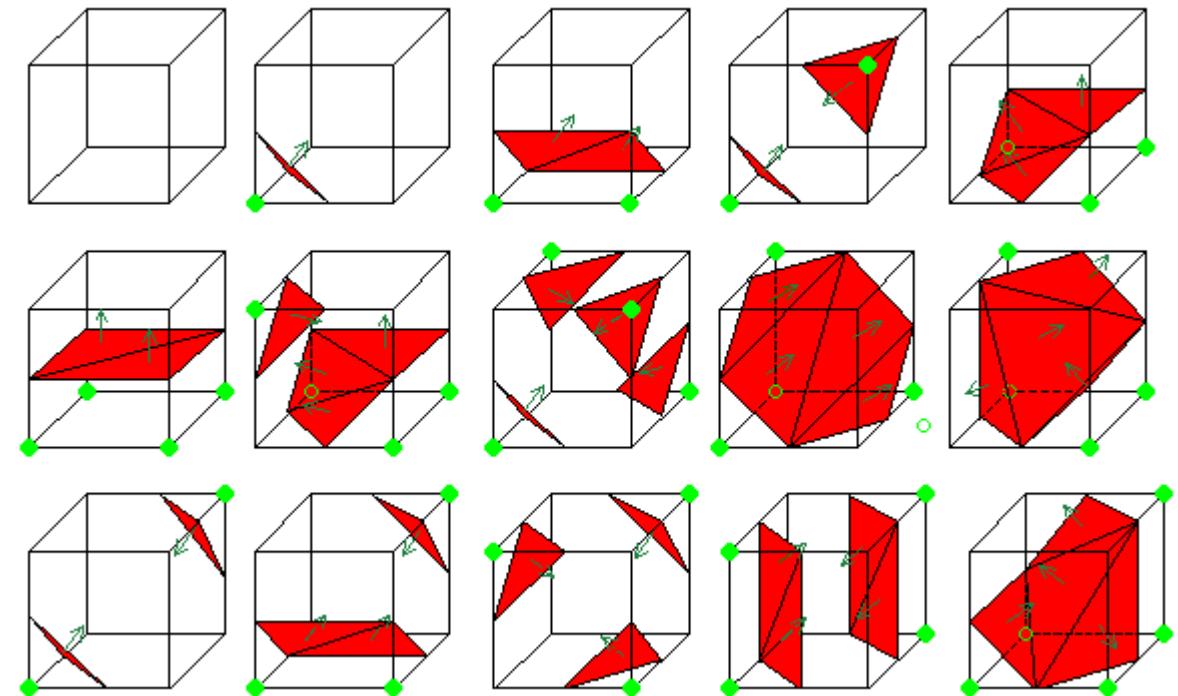
- Three dimensional grid
- Cubic cells (8 nodes)
- $2^8=256$ permutations of red/blue value
- Triangles are created for each cell

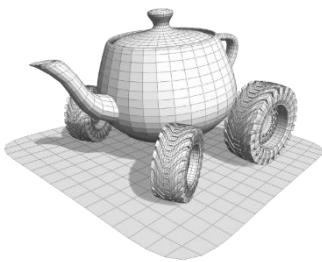




Marching cubes: Look-up table

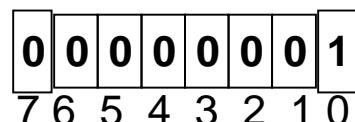
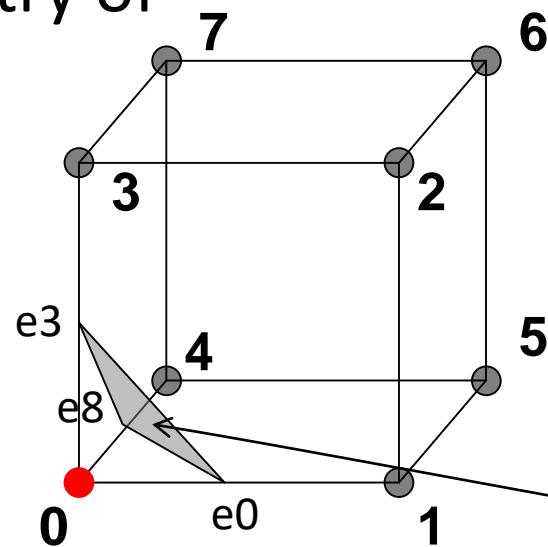
- The 256 permutations reduce to 15 different cases by means of mirroring and rotation



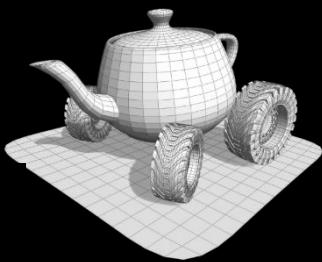


Marching cubes: Look-up table

- The 256 permutations reduce to 15 by means of mirrorings and rotations
- Each permutation corresponds to an entry of the look-up table



LookUpTable	
0	: nil
1	: {e0,e3,e8}
2	:
3	:
...
255	: nil



arbitrary
slice

Chapter 3: How a 3D Model is
Represented

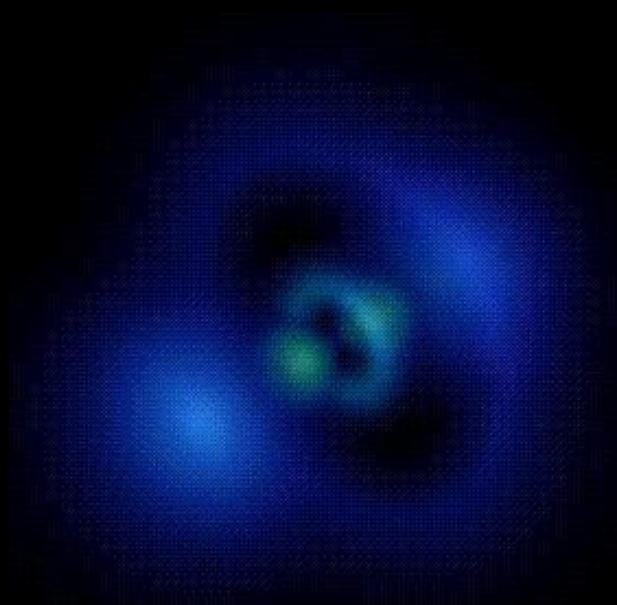
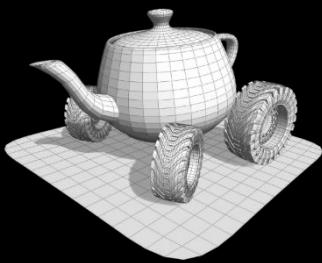


direct volume
rendering

Introduction to Computer Graphics: A Practical Learning Approach.



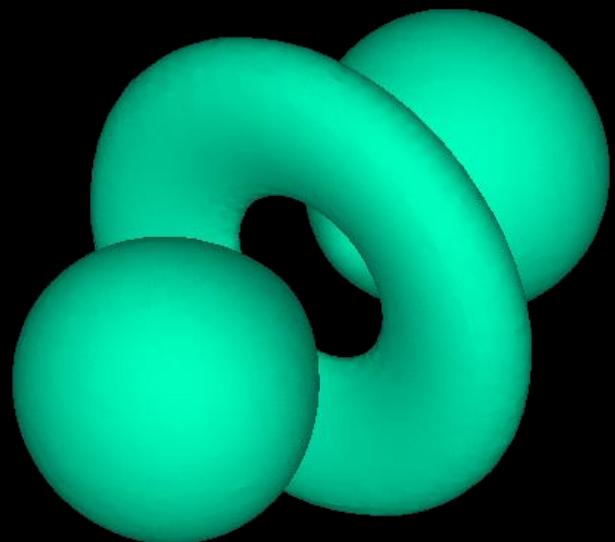
Marching cube meshes



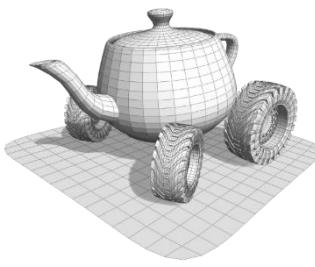
direct volume
rendering

Chapter 3: How a 3D Model is
Represented

Introduction to Computer Graphics: a Practical Learning Approach.



Marching cube



Marching Cubes: pros/issues

- The original paper (1987) is one of the most cited in Computer Graphics
- Easy to implement
- Robust and amenable to parallel processing
- It solved a difficult problem

Marching cubes: A high resolution 3D surface construction algorithm

[WE Lorensen, HE Cline - ACM siggraph computer graphics, 1987 - dl.acm.org](#)

... We present a new algorithm, called **marching cubes**, that creates triangle models of constant density surfaces from 3D medical data. Using a ... and functionality of **marching cubes**. We also discuss improvements that decrease processing time and add solid modeling capabilities. ...

[☆ Save](#) [99 Cite](#) [Cited by 17716](#) [Related articles](#) [All 11 versions](#)

..then why from '87 zillions papers on Marching cubes where published ?

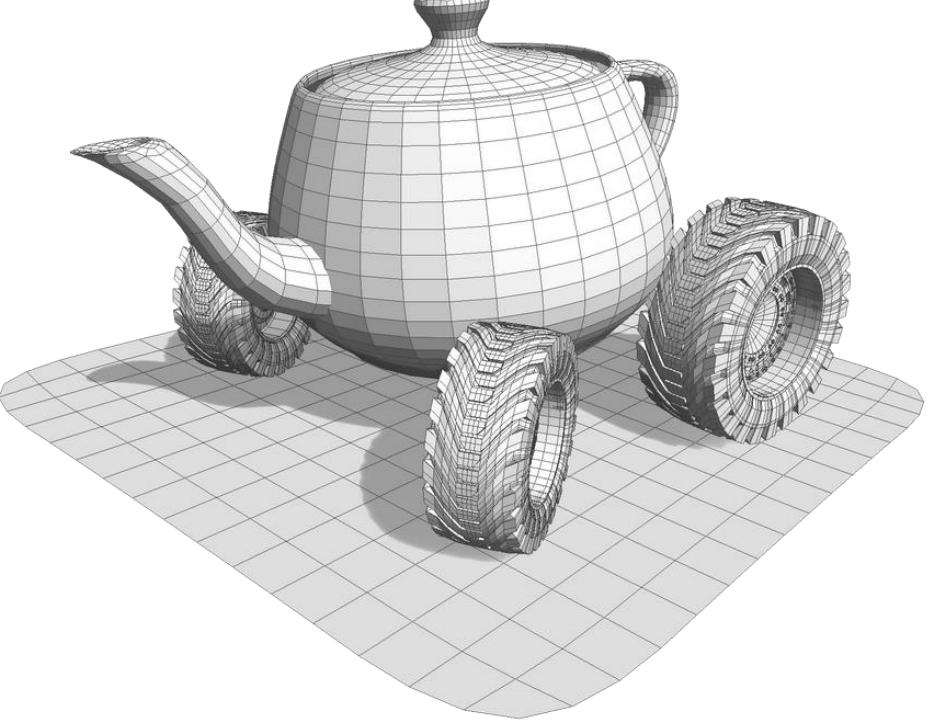
Issues:

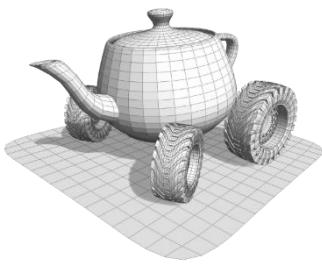
- **Consistency.** How to guarantee a C0 and two-manifold result: ambiguous cases
- **Correctness:** return a good approximation of the “real” surface
- **Mesh complexity:** the number of triangles does not depend on the shape of the isosurface
- **Mesh quality:** arbitrarily thin triangles

Cliffhanger!!

Chapter 3: How a 3D Model is
Represented

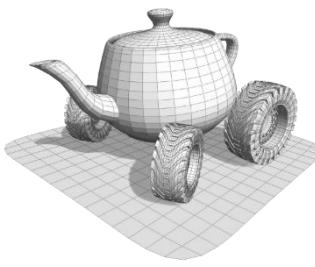
Picking





Picking

- By picking we mean: mouse-click on the x,y window coordinates and knowing which point on the visible scene projects on x,y. There are two cases:
 - Return the coordinates of 3D point (that is, no knowledge of the specific primitive)
 - Read back the depth value from the depth buffer and transform x,y,depth in world space
 - Select an object (to know which object was «picked»). With the term «object» we may refer to a triangle mesh, or a triangle, an edge, a point...
 - How to?
 1. Encode the primitive «name» (number) in the rendering pass
 2. Read back from color/depth/stencil buffer the value at x,y



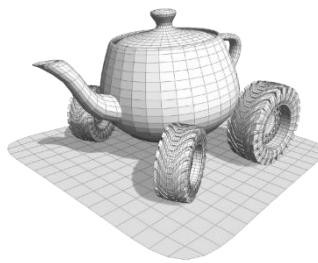
Picking «just» the 3D point

1. The mouse callback provides x,y pos
2. Read back from the depth buffer the depth value d

```
float depthvalue;  
glReadPixels(x, width - y, 1, 1, GL_RGBA, GL_DEPTH_COMPONENT, &depthvalue);
```

3. Reproject (x,y,d) in world space

```
glm::vec3 hit = glm::unProject(glm::vec3(x, width - y, depthvalue), view,  
proj, glm::vec4(0, 0, width, height));
```



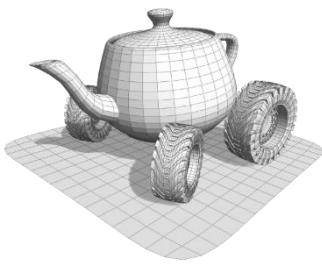
Selecting the object

1. Associate an integer «**id**» to each object
2. Encode **id** as the RGB color:
3. Pass r,g,b to the fragment shader as a uniform variable while rendering the object
4. Render
5. Readback the color
6. Recompute the index

```
int r = i & 0x000000FF;  
int g = (i & 0x0000FF00) >> 8;  
int b = (i & 0x00FF0000) >> 16;
```

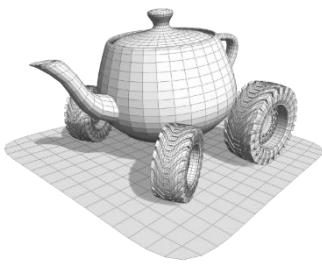
If GL_FLOAT is used to read the color value,
If GL_UNSIGNED_BYTE then do not multiply

```
int id = r*255 +  
        ((g*255) << 8) +  
        ((b*255)<<16)
```



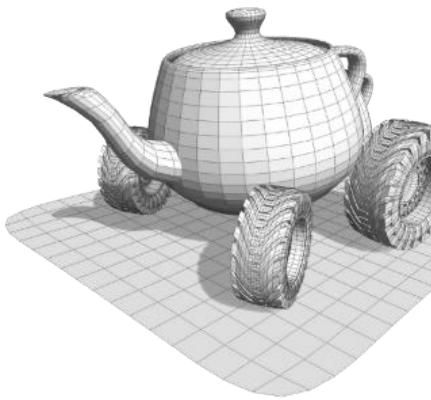
Exercise (for real now) (1/2)

- Starting from `code_8_picking`:
 - Implement picking for all the objects of the scene
 - Pick if CTRL is pressed (or another modifier)
 - When an object is selected, **translate** the view frame so that the object is in the center of the scene
 - Add a trackball such that its transformation is applied to the object



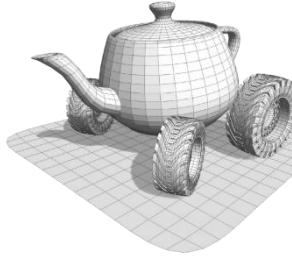
Exercise (for real now) (2/2)

- Note: you will need to render the selectable objects with the color coding. This is not something you want to show on screen, therefore you will want to:
 1. Draw the objects with color coding
 2. Find out the ID of the clicked object
 3. Clear (`glClear(GL_...)`)
 4. Draw the scene normally



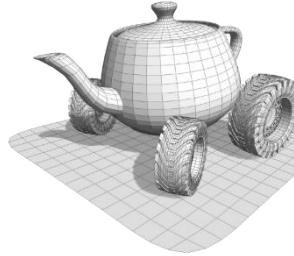
Lighting

Lighting: intro



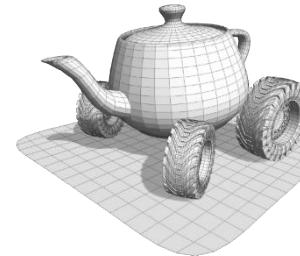
- Computing light
 - How much light
 - Color of the light
- that arrives
 - From a point in the scene
 - To the eye of the observer (to the camera)

Lighting: intro



- Output:
 - Perceived color for any given point
- Input:
 - Characteristics of the object lit (aka “material”)
 - (eg: wet paper, rough plastic, …)
 - (or just «a color»)
 - Characteristics of the light
 - (what are the light sources, what is their intensity and color)
 - shape of the objects in the scene

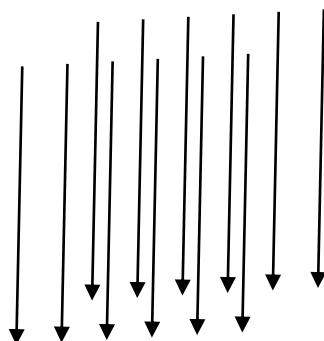
Some quantities: radiant flux



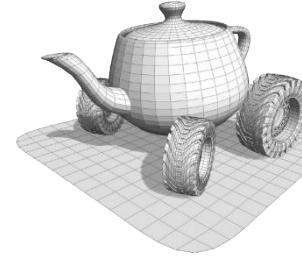
symbol	name	unit
Φ	Radiant flux	Watt (J/s)

Radiant flux: total amount of light passing through an area or a volume.

Higher flux → brighter light



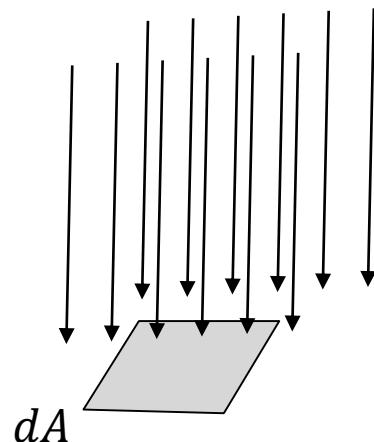
Some quantities: irradiance



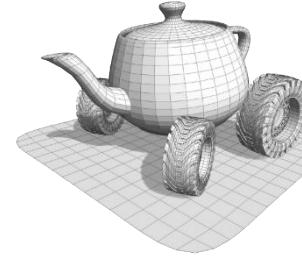
symbol	name	unit
Φ	Radiant flux	Watt (J/s)
$E = \frac{d\Phi}{dA}$	Irradiance/exitance	$Watt/m^2$

Irradiance/exitance: amount of flux (arriving or leaving) per unity of area.

For a surface with constant radiant flux: $E = \frac{\Phi}{A}$

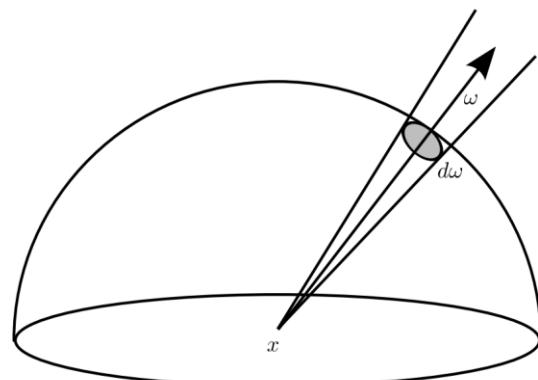


Some quantities: intensity (2/2)

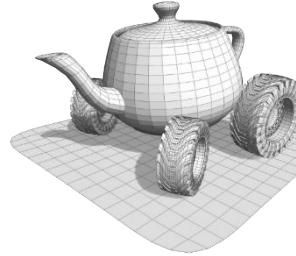


symbol	name	unit
Φ	Radiant flux	Watt (J/s)
$E = \frac{d\Phi}{dA}$	Irradiance/exitance	$Watt/m^2$
$I = \frac{d\Phi}{d\omega}$	Directional intensity	$Watt/sr$

Directional Intensity: amount of flux per unit **solid angle**
leaving from a point towards a direction ω
The solid angle is expressed in **steradians**



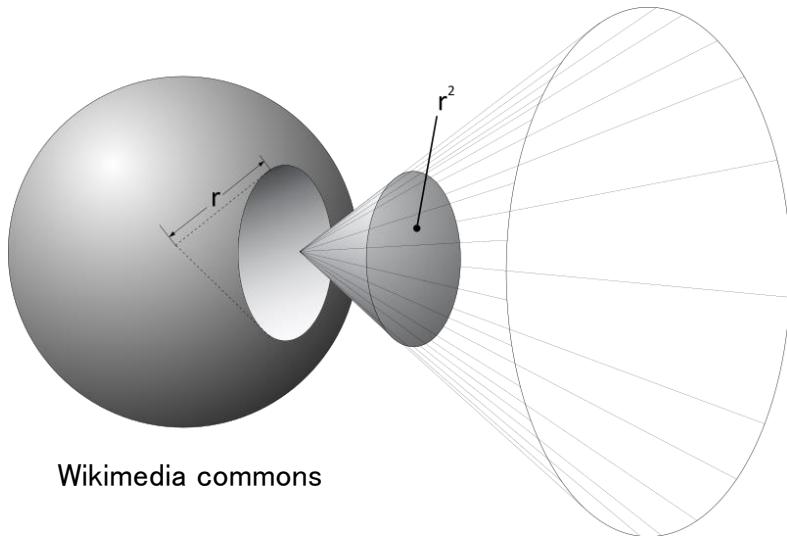
Some quantities: steradian



Steradian: is the 3D version of the radian

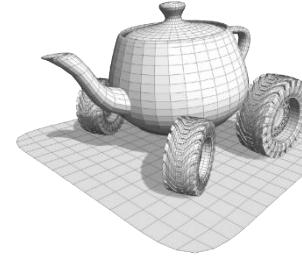
Given a sphere with radius r , one steradian is the solid angle that subtends a region of the sphere's surface of area r^2

The total area of the sphere is $S = 4\pi r^2$, which means that the total solid angle is $S/r^2 = 4\pi$ steradians



Wikimedia commons

Some quantities: intensity (2/2)



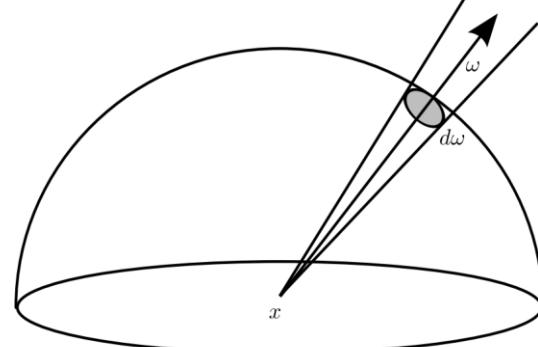
symbol	name	unit
Φ	Radiant flux	Watt (J/s)
$E = \frac{d\Phi}{dA}$	Irradiance/exitance	$Watt/m^2$
$I = \frac{d\Phi}{d\omega}$	Directional intensity	$Watt/sr$

Directional Intensity: amount of flux per unit **solid angle**

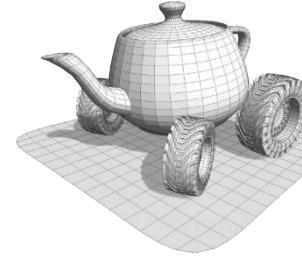
leaving from a point towards a direction ω

The solid angle is expressed in **steradians**

E.g. the intensity Φ of a light emitted uniformly along all directions by a point light source is $\Phi/4\pi$

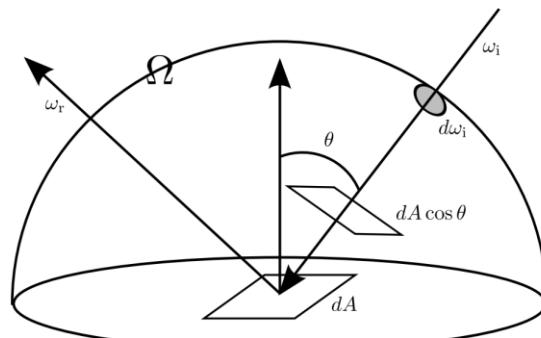


Some quantities: radiance

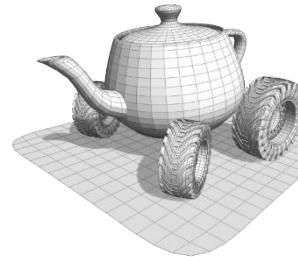
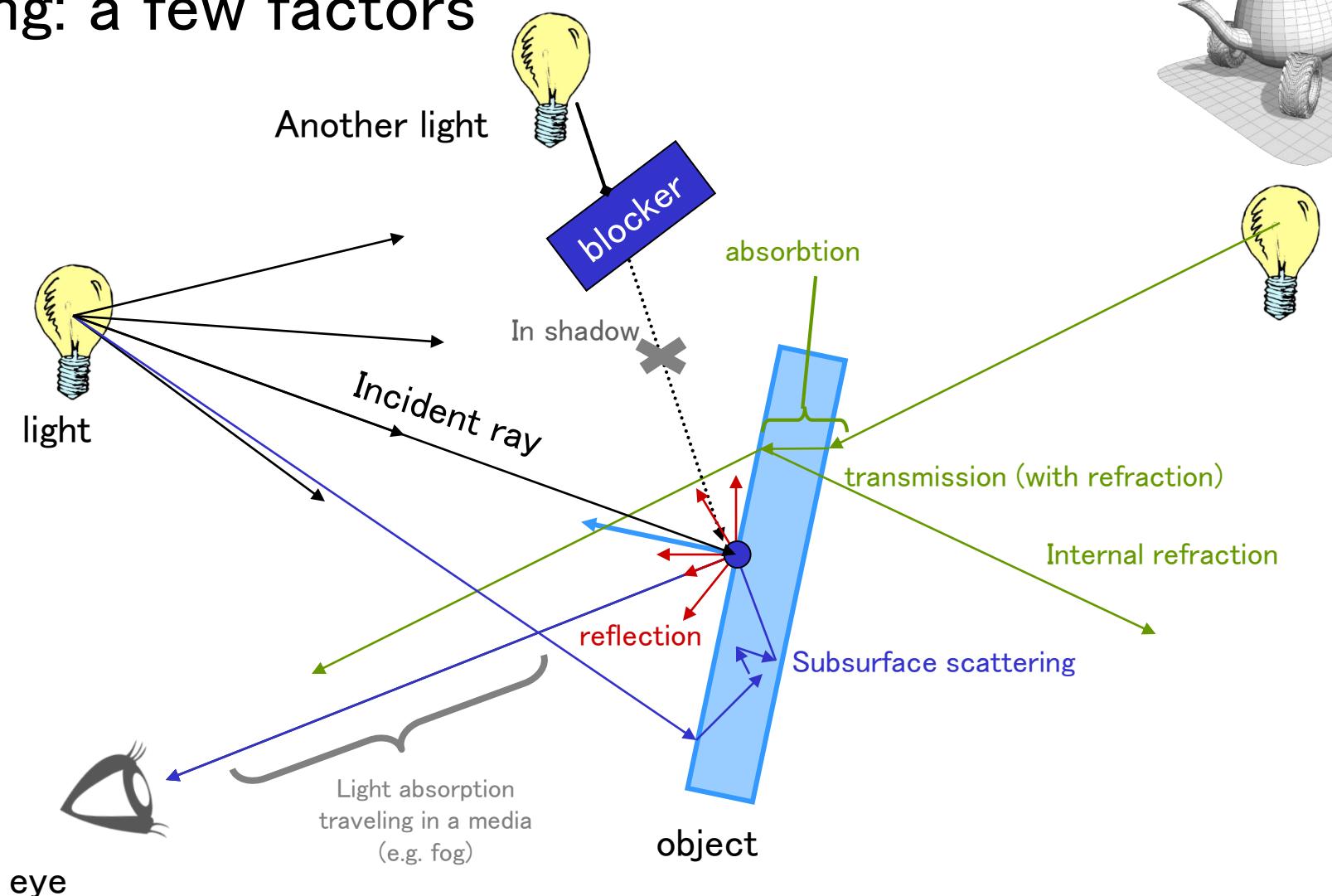


symbol	Name	unit
Φ	Radiant flux	Watt (J/s)
$E = \frac{d\Phi}{dA}$	Irradiance/exitance	Watt m^{-2}
$I = \frac{d\Phi}{d\omega}$	Directional intensity	Watt s_r^{-1}
$L = \frac{d^2\Phi}{dA \cos \theta d\omega}$	Radiance	Watt $s_r^{-1}m^{-2}$

Radiance: flow of radiation per unity of area and per unity of solid angle

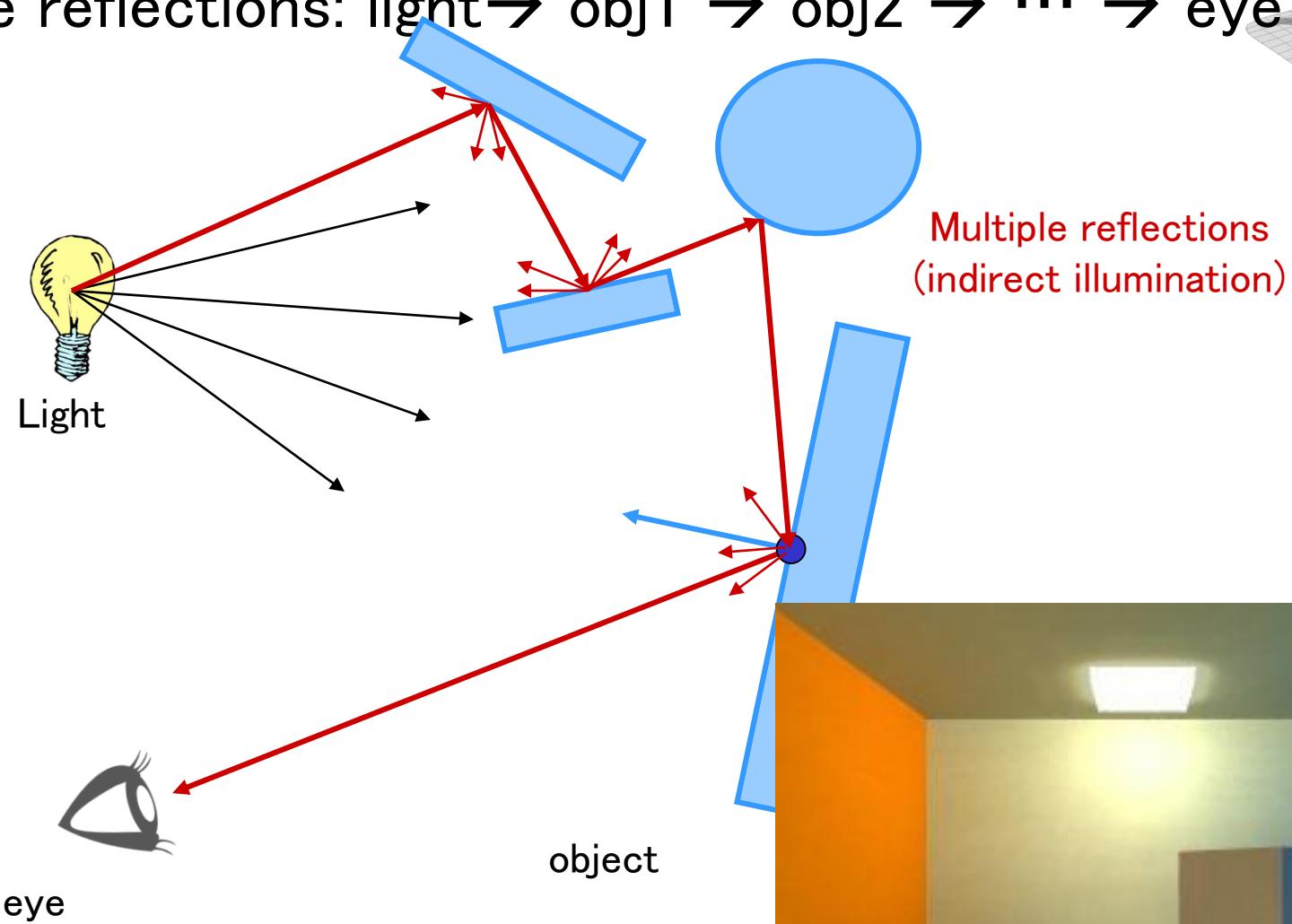
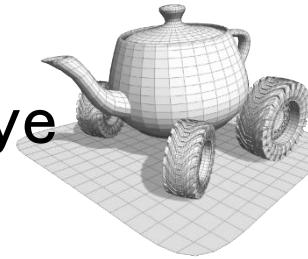


Lighting: a few factors

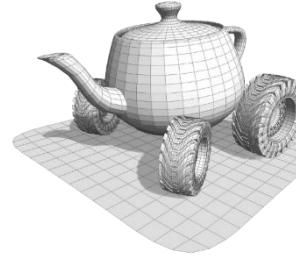


Lighting: some factors

multiple reflections: light \rightarrow obj1 \rightarrow obj2 $\rightarrow \dots \rightarrow$ eye



Lighting: global VS local



Local lighting

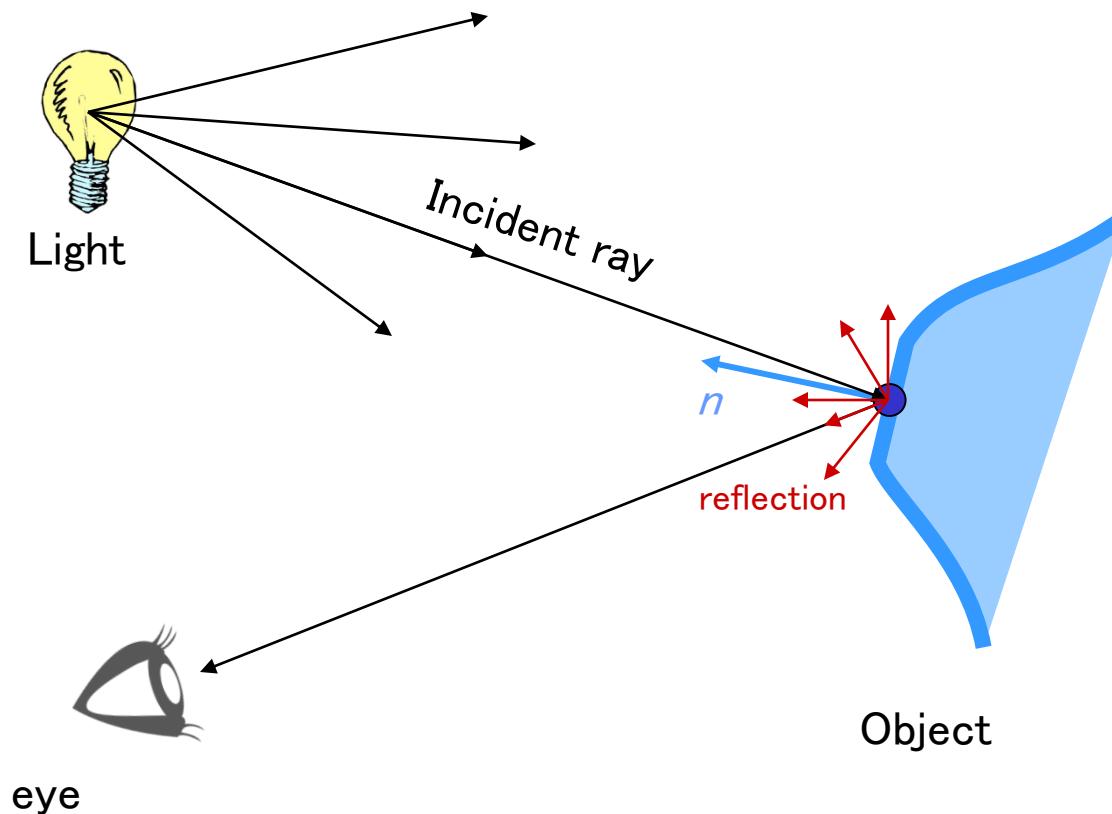
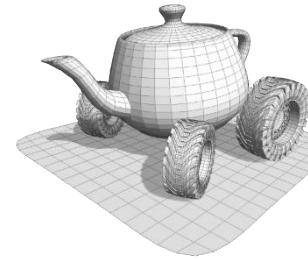
- Only takes into account:
 - Lighting conditions
 - Number of lights
 - Their position
 - Their color
 - Point on the surface to be lit
 - Its orientation (normal)
 - Its characteristics
 - E.g. color
 - Position of the observer (POV)
- The rest of the world is ignored

Global lighting

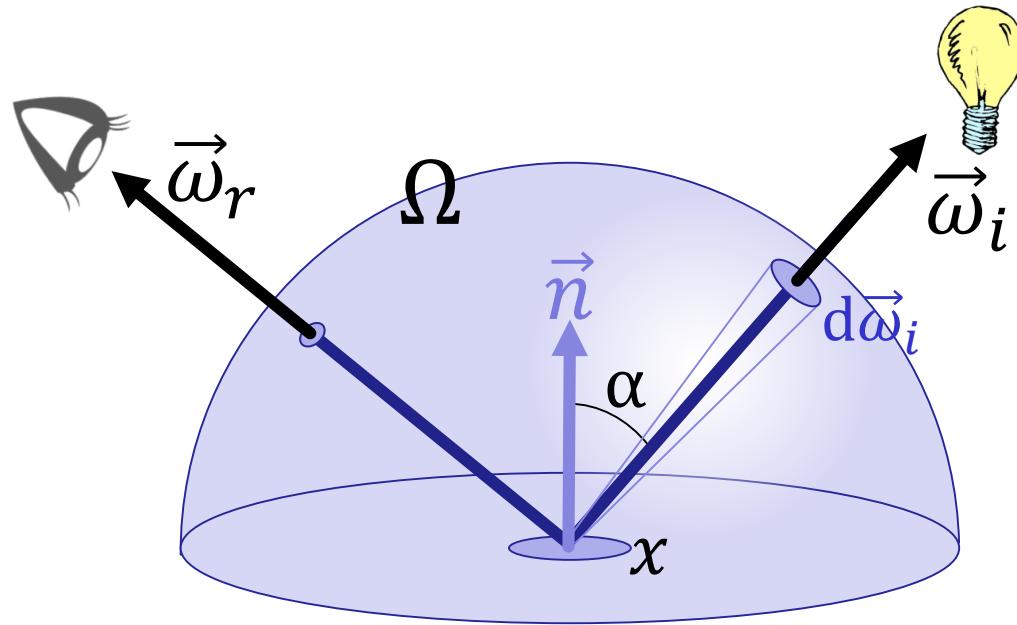
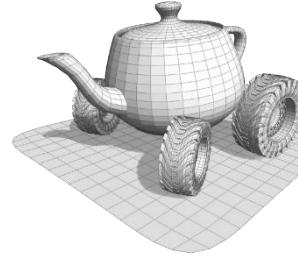
- Multiple reflections
- shadow
- Subsurface scattering
- refraction
- ...

Why easier with the
rasterization based
pipeline

Local lighting :
emitter → object → eye (and nothing else)



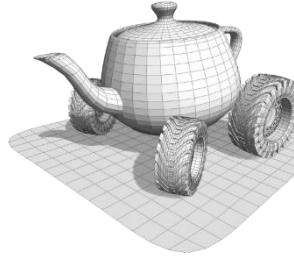
The radiance equation for local lighting



$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) \cos(\alpha) d\vec{\omega}_i$$

The radiance equation for local lighting



x point on the lit surface;

$\vec{\omega}_r$ Direction from x to the observer

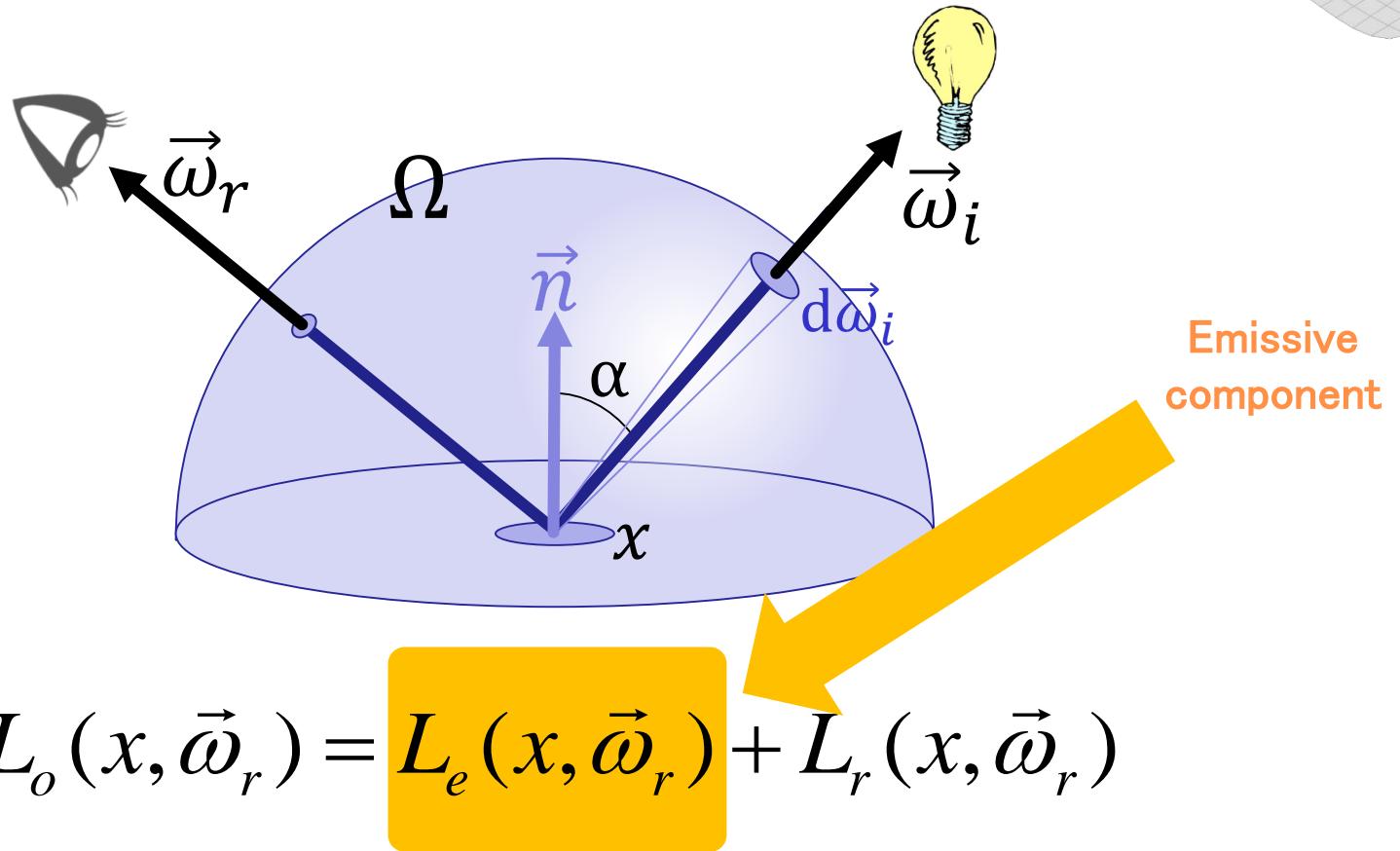
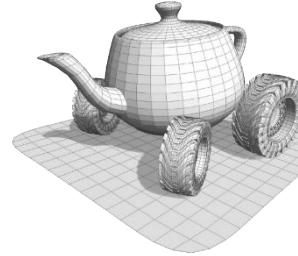
$\vec{\omega}_i$ Direction from which the incident ray arrives

$L_i(x, \vec{\omega}_i)$ amount of incident light that reaches x from dir $\vec{\omega}_i$

$(\vec{\omega}_i \cdot \vec{n})$ cosine of the incident angle w.r.t. To the surface normal

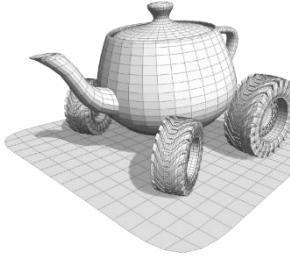
Ω All possible directions (that is, unit vectors in the hemisphere)

The radiance equation for local lighting



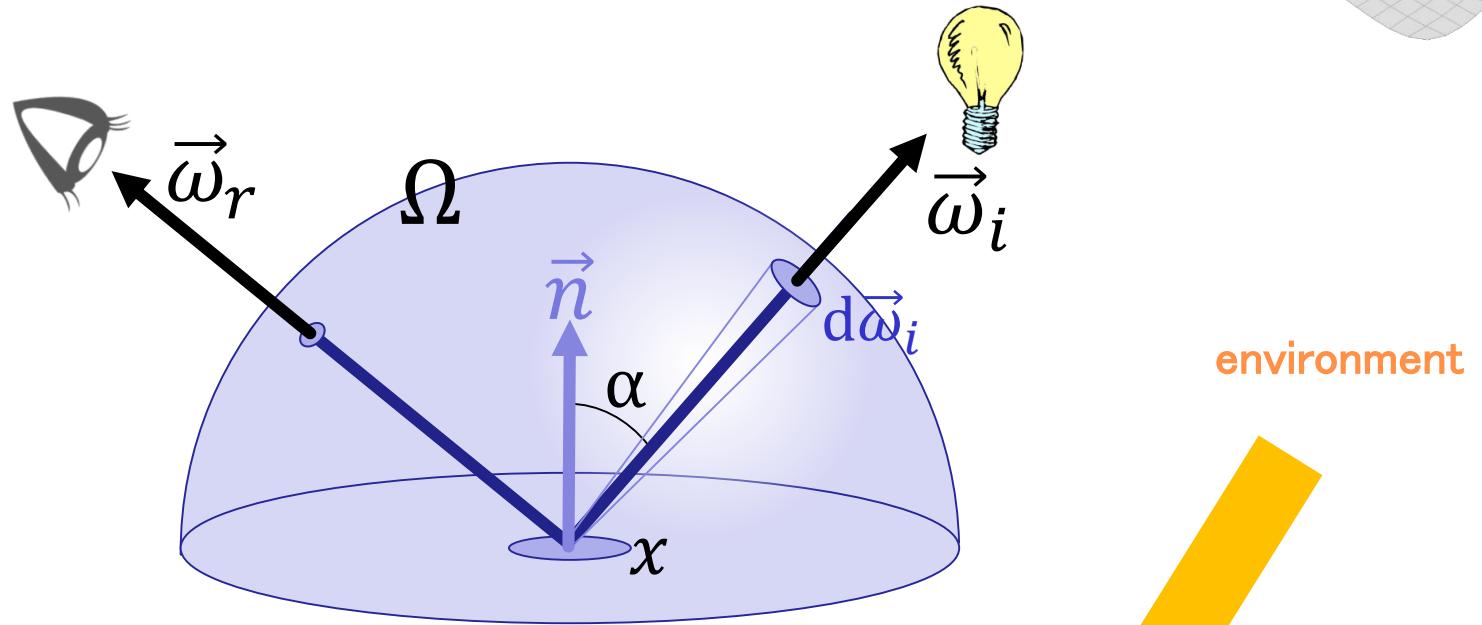
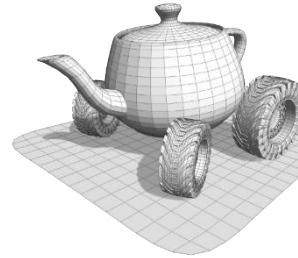
$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

Emissive component



- Only used for the few material which emit light
 - It models only the path: emitter→eye
(not emitter→object→eye)
 - That is the effect of emitted light in the scene is not consider by this term

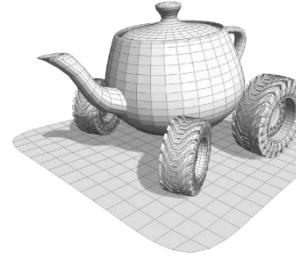
The radiance equation for local lighting



$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

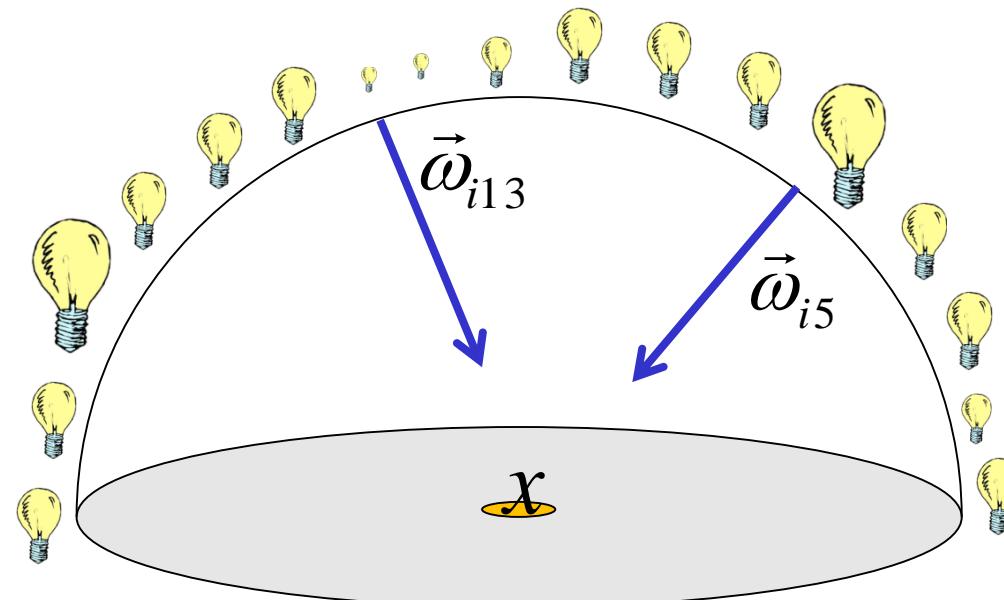
Incident light



- For each position x
 $L_i(x, \dots)$ represents
the distribution of incident light

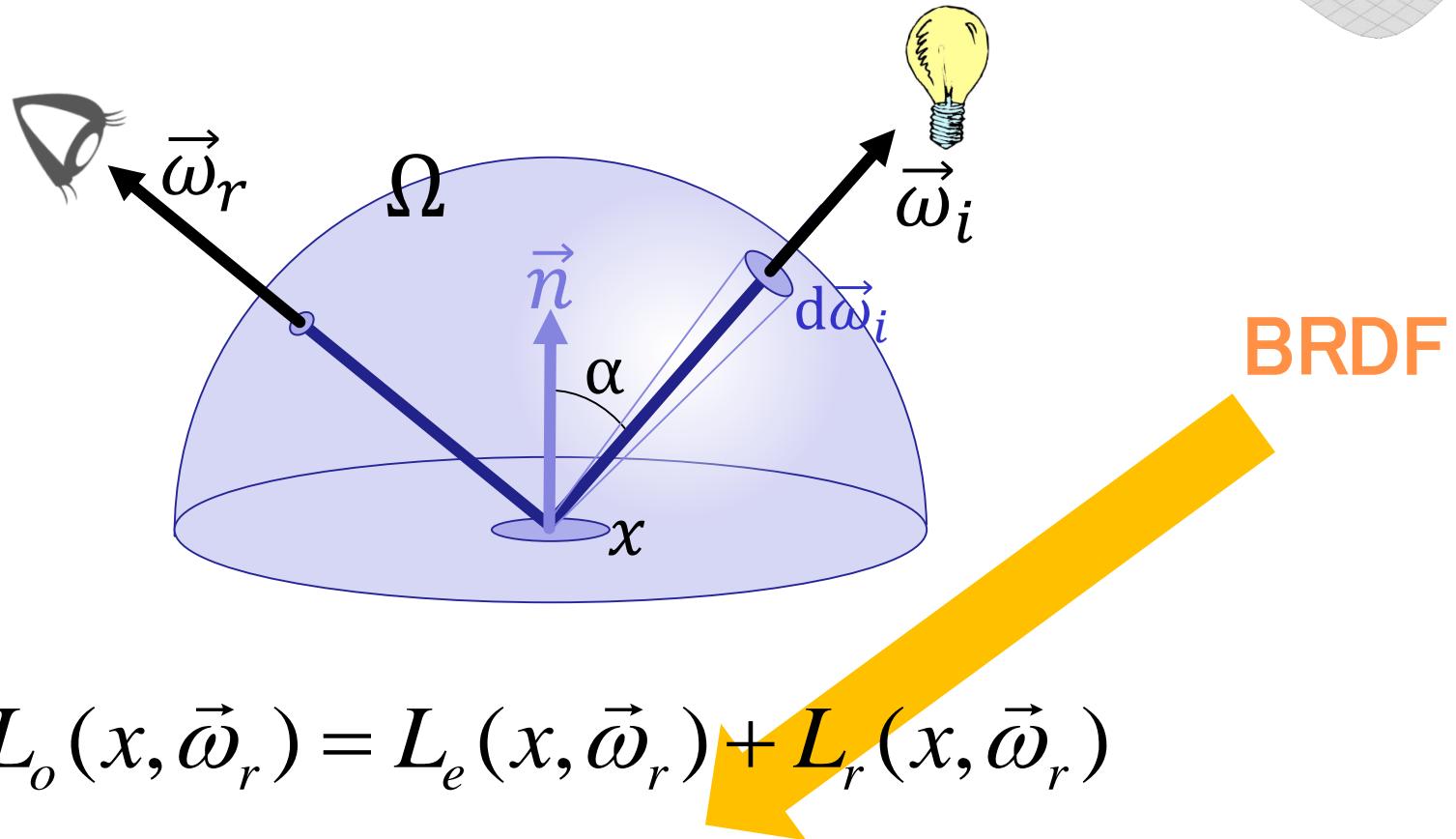
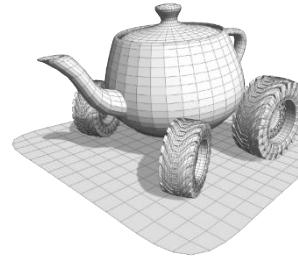
$$L_i(x, \vec{\omega}_i) = \text{how much light arrives to } x \text{ from direction } \vec{\omega}_i$$

It models the illumination ambient Eg:
• Room with an open window
• sunny day
• cloudy day
• discoteque



How many photons
from each direction

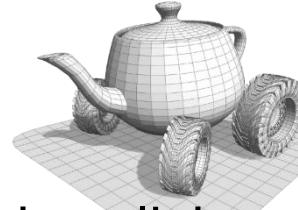
The radiance equation for local lighting



$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

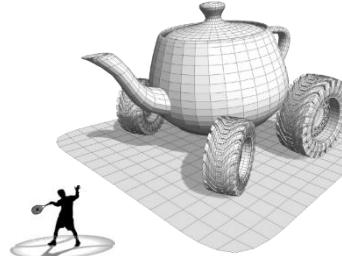
The radiance equation for local lighting: BRDF



$f_r(x, \vec{\omega}_i, \vec{\omega}_r)$ Function representing the fraction of incident light from ω_i which is reflected towards

- It's the function that describes the material of the object
- If it's constant w.r.t. to $x \Rightarrow$ uniform material:
 - same material on every point
 - In this case we talk about **BRDF** of the material
 - **BRDF** = Bidirectional Reflectance Distribution Function
 - Describes the properties of the material (e.g: it's glossy, if it has purple reflexes, metallic...)

BRDF of a material (no « x » parameter)



$$f_r = \text{tennis player}$$

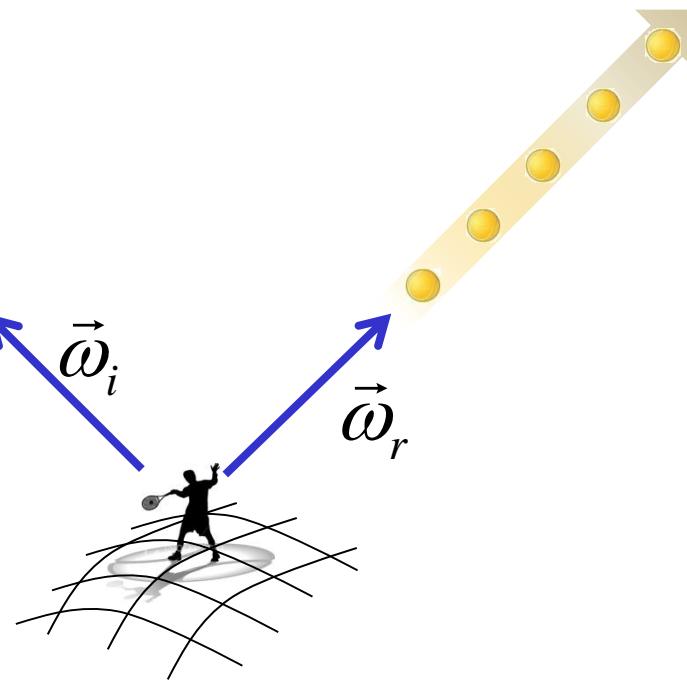
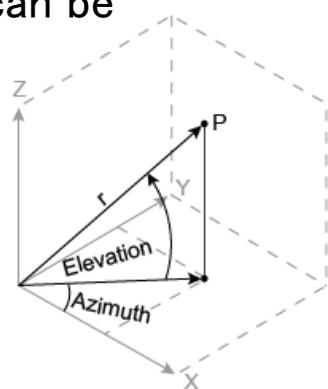
$$f_r(\vec{\omega}_i, \vec{\omega}_r) =$$

On 100 arriving at from direction $\vec{\omega}_i$, how many will bounce along $\vec{\omega}_r$?

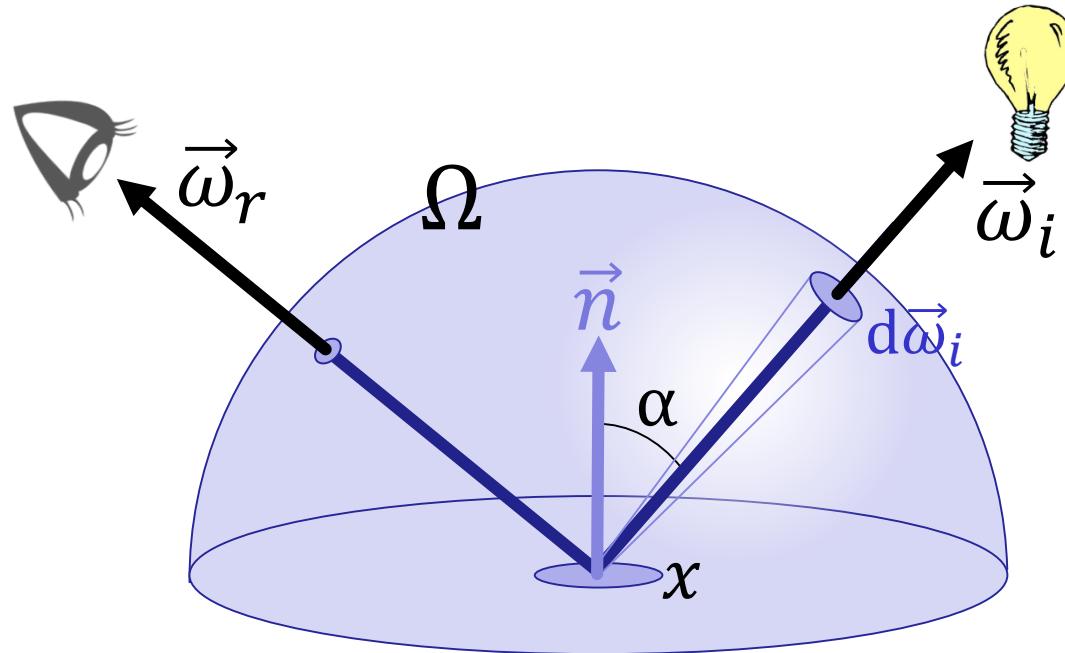
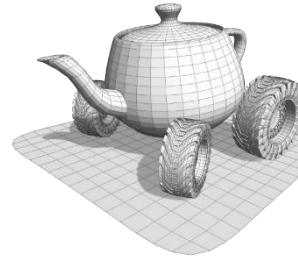
photon =

$$f_r: \mathbb{R}^4 \rightarrow \mathbb{R}$$

Each direction can be specified with 2 parameters



The radiance equation for local lighting

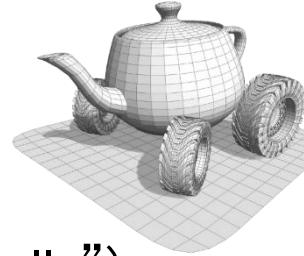


Cosine law

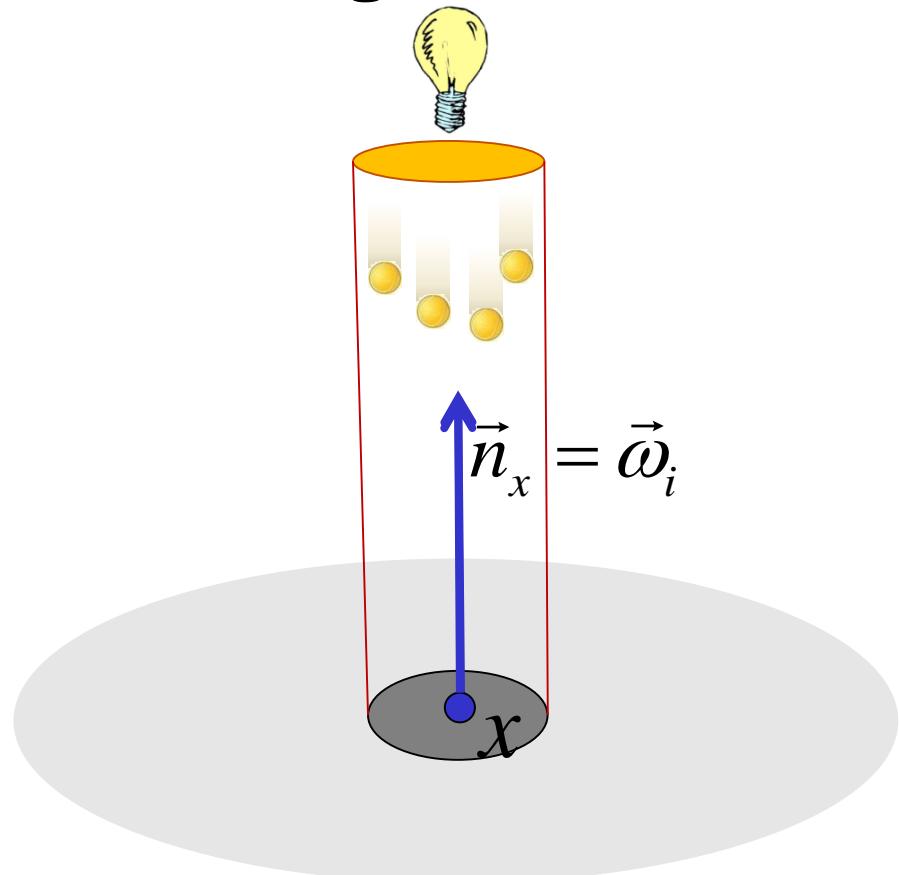
$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

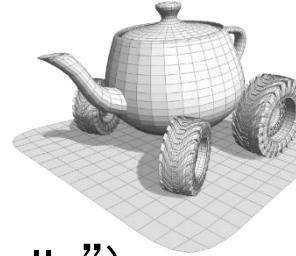
Cosine law



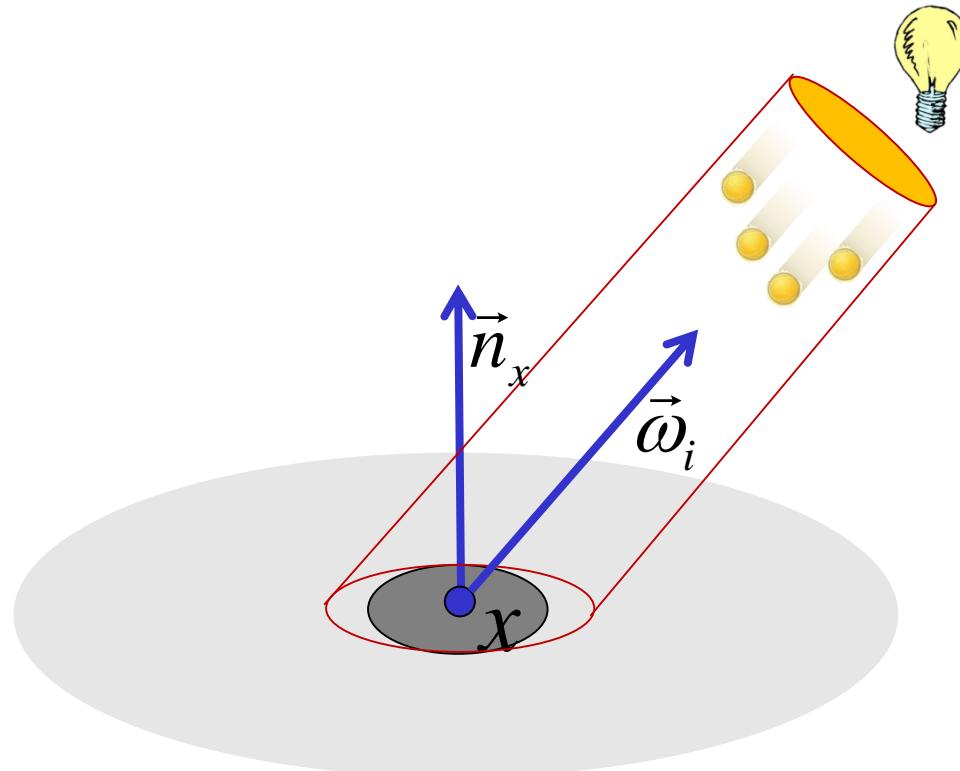
- From direction ω_i arrive L lumens (N “tennis balls”), how many hit the neighborhood of x ?



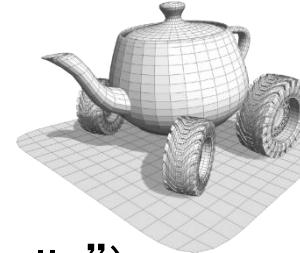
Cosine law



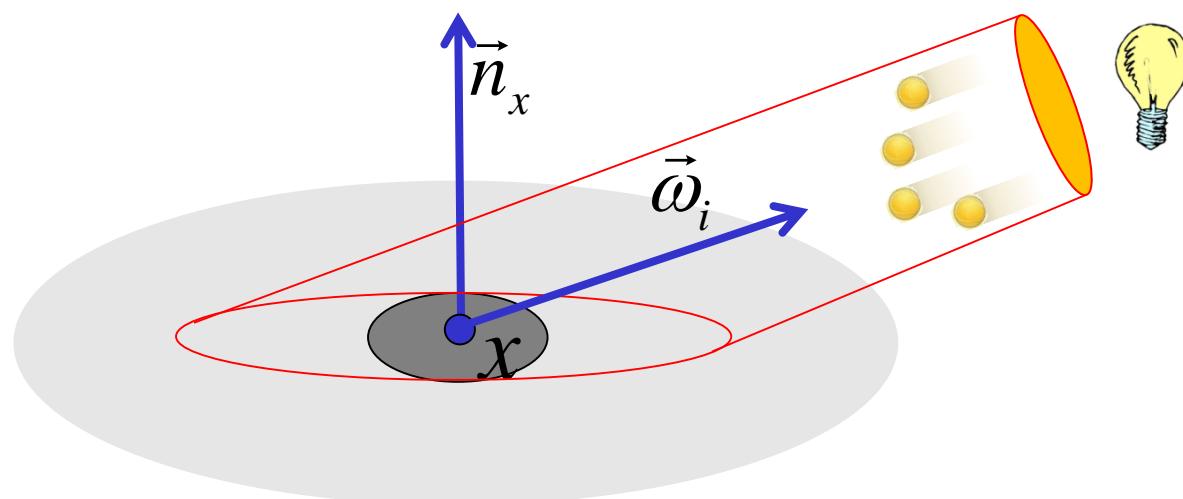
- From direction ω_i arrive L lumens (N “tennis balls”), how many hit the neighborhood of x ?



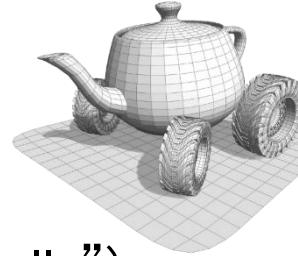
Cosine law



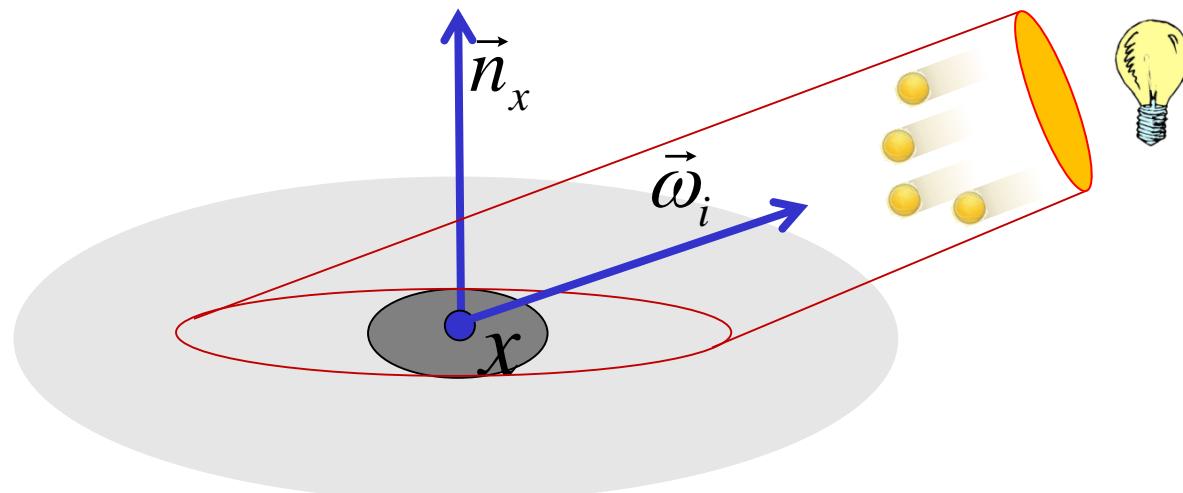
- From direction ω_i arrive L lumens (N “tennis balls”), how many hit the neighborhood of x ?



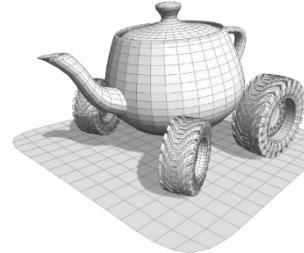
Cosine law



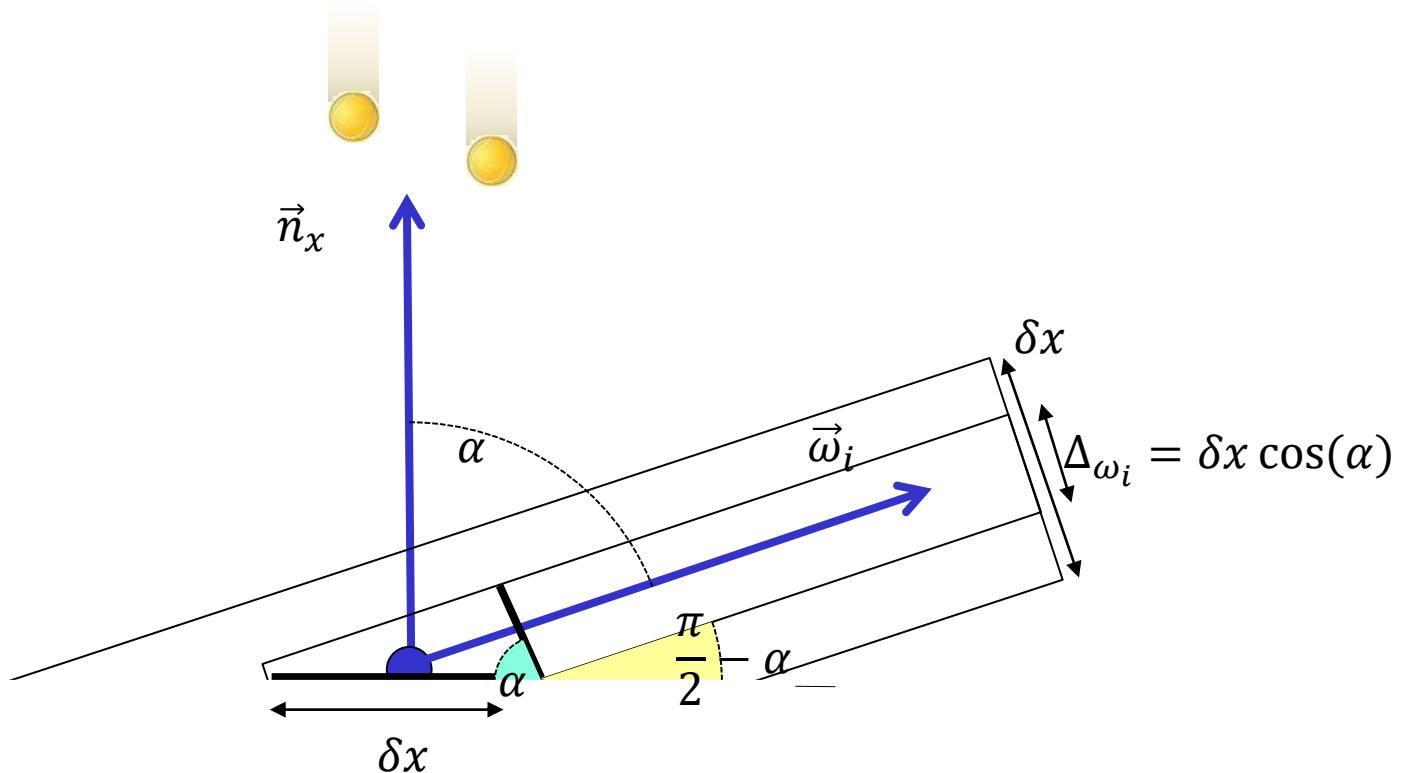
- From direction ω_i arrive L lumens (N “tennis balls”), how many hit the neighborhood of x ?
- The cosine law: $\cos(\alpha)L = (\vec{\omega}_i \cdot \vec{n})L$



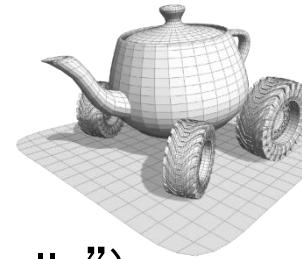
Cosine law



- Consider the 2D case:



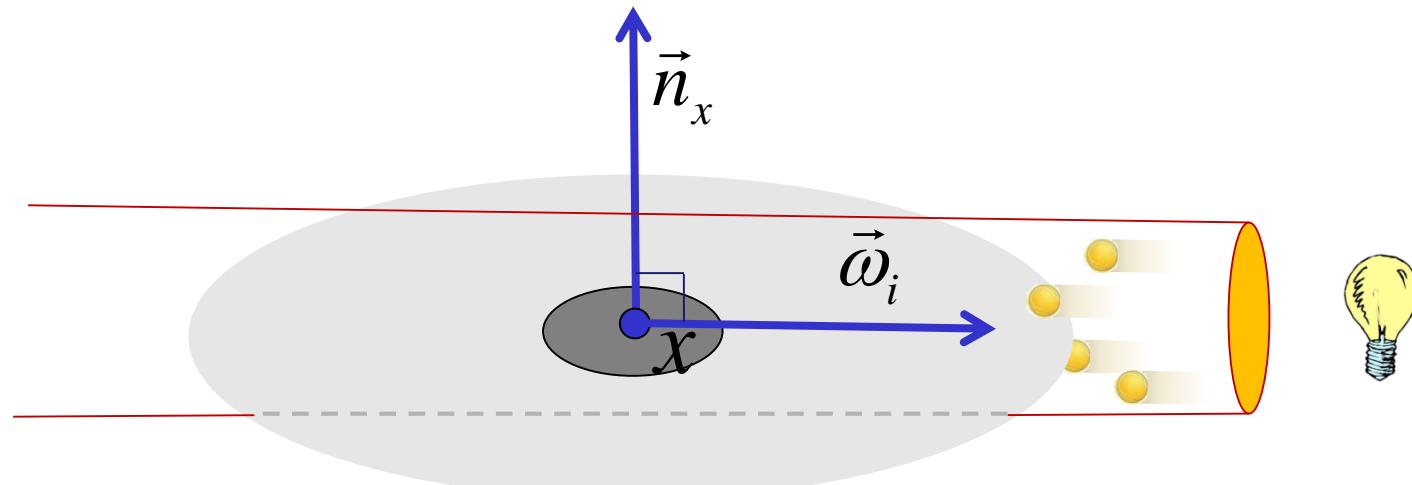
Cosine law



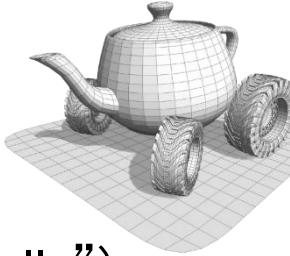
- From direction ω_i arrive L lumens (N “tennis balls”), how many hit the neighborhood of x ?

Grazing light:

$$(\vec{\omega}_i \cdot \vec{n}) = 0$$



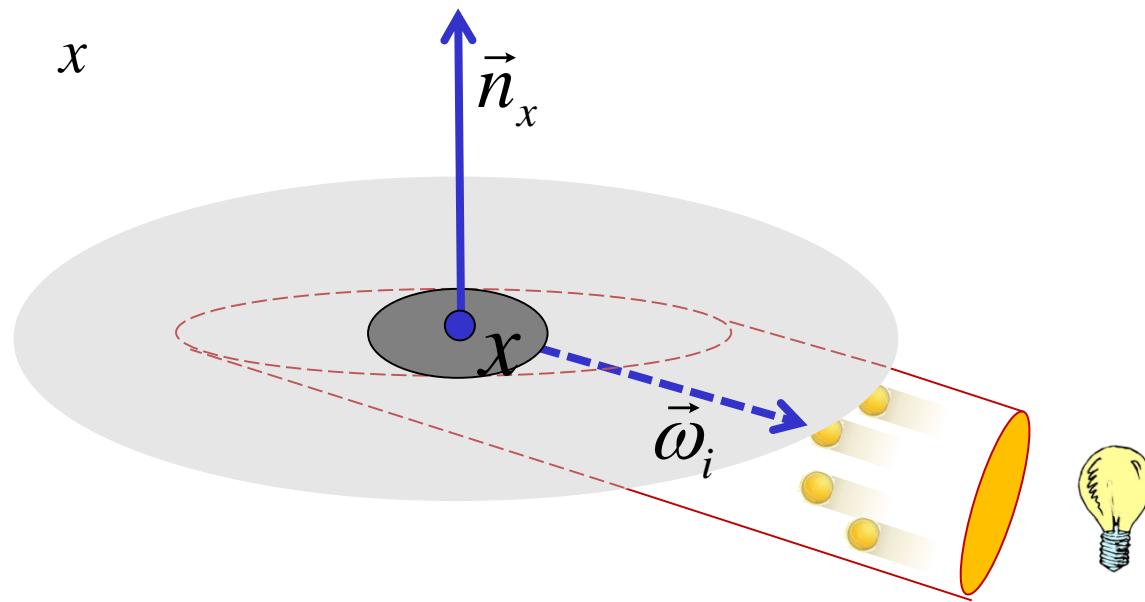
Cosine law



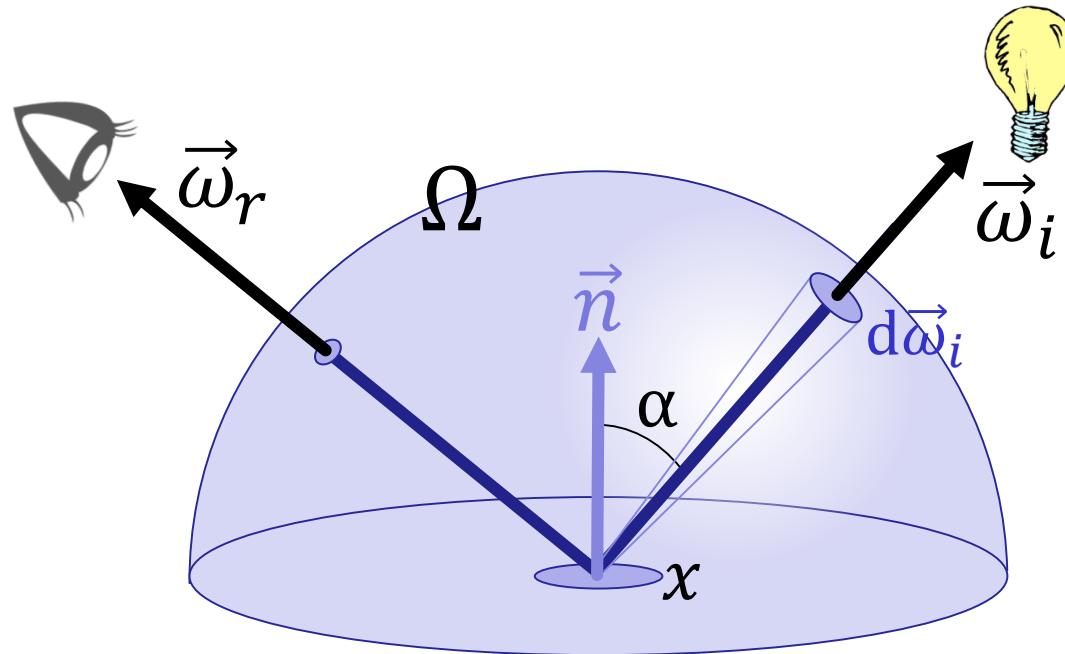
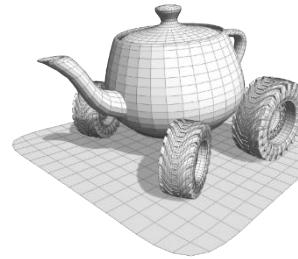
- From direction ω_i arrive L lumens (N “tennis balls”), how many hit the neighborhood of x ?

Light from behind: $(\vec{\omega}_i \cdot \vec{n}) < 0$

answer: 0 because x casts a shadow on itself



The radiance equation for local lighting

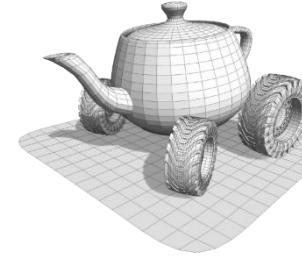


$$\max(\cos \alpha, 0)$$

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

The radiance equation for local lighting: recap



$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

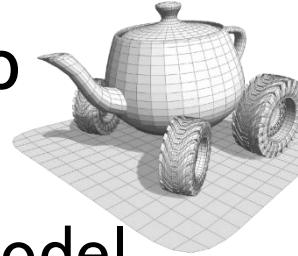
“the light we see from a point x is given by the sum of light emitted from x and light reflected from x ”

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} \overbrace{f_r(x, \vec{\omega}_i, \vec{\omega}_r)}^{(C)} \overbrace{L_i(x, \vec{\omega}_i)}^{(A)} \overbrace{(\vec{\omega}_i \cdot \vec{n})}^{(B)} d\vec{\omega}_i \quad (D)$$

reflected light is the integral of the contribution of all light sources in the scene. For each direction the contribution is given by the product of:

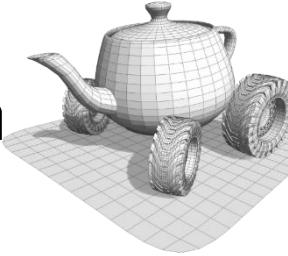
- (A) · (D) how much light from that direction (ambient illumination)
 - (B) what fraction of this light gets to x ([cosine law](#))
 - (C) what fraction of the remaining is reflected towards the observer (from x)

The radiance equation for local lighting: recap



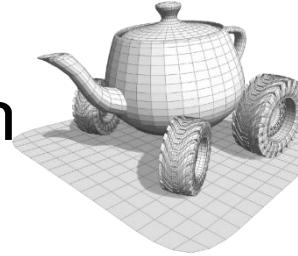
- The radiance equation is a **physically accurate** model
- The material is represented with a **BRDF**
 - the BRDF of actual material can be acquired with the proper setup, or simulated
- It can produce very accurate renderings.
 - But it's still too costly for real time rendering
- We need strong/drastic simplification/approximations

Approximation 1: simplified and discrete ambient of illumination



- Instead of $L_i(x, \vec{\omega}_i)$
- say:
 - Every point in the model receives the same light
 - (x disappear)
 - Light arrives only from a small number of direction N
 $\vec{\omega}_0, \vec{\omega}_1, \dots, \vec{\omega}_{N-1}$
 - Even just one (cioè $N=1$)
 - No light from all other directions
 - The integral becomes a small sum over N
 - One term for each light

Approximation 1: simplified and discrete ambient of illumination



- from: **integral** (over all incident direction)

$$\int_{\vec{\omega}_i \in \Omega} \dots \cdot L_i(x, \vec{\omega}_i) \cdot \dots d\vec{\omega}_i$$

Here i stands for “incident”

Here i stands for i^{th}

$$\sum_{i=0}^{N-1} \dots \cdot L_i \cdot \dots$$

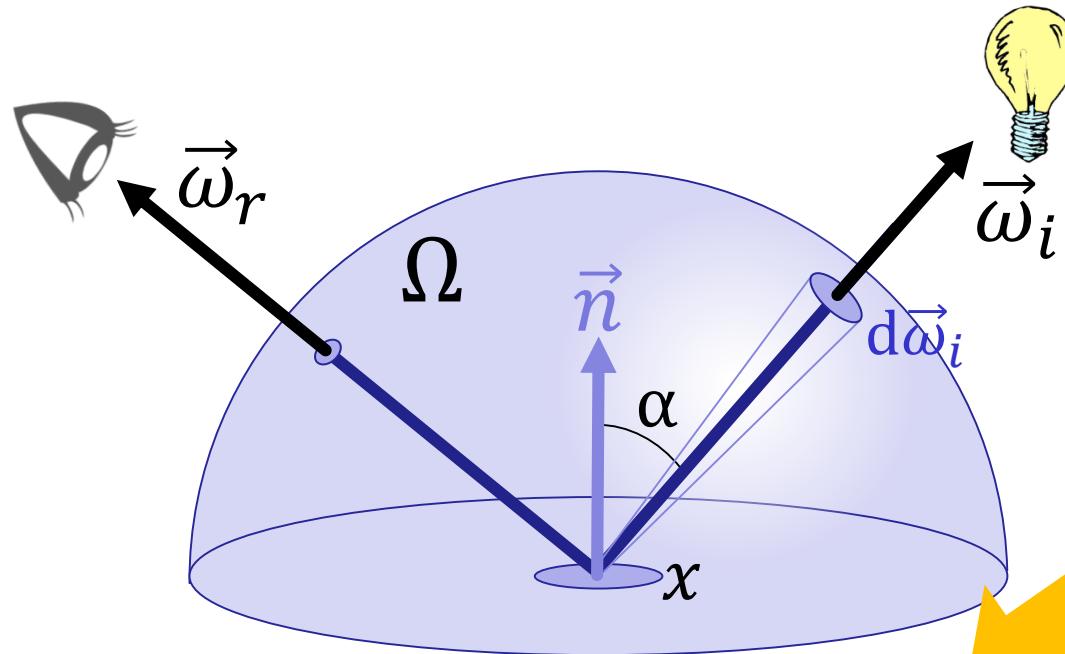
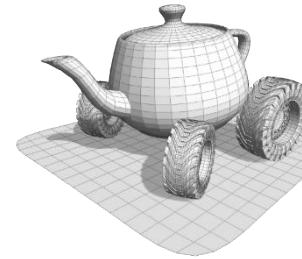
Constant light from light i

- For colored light:

$$\sum_{i=0}^{N-1} \dots \cdot \begin{pmatrix} L_i^R \\ L_i^G \\ L_i^B \end{pmatrix} \cdot \dots$$

Color and intensity of light i

Approximating radiance equation 1



Constant color
and intensity

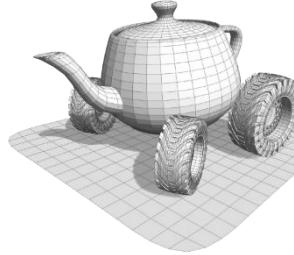
Incoming
direction of
light i

$$L_r(x, \vec{\omega}_r) = \sum_{i=0}^{N-1} f_r(x, \vec{\omega}_i, \vec{\omega}_r)$$

$$\begin{pmatrix} L_i^R \\ L_i^G \\ L_i^B \end{pmatrix}$$

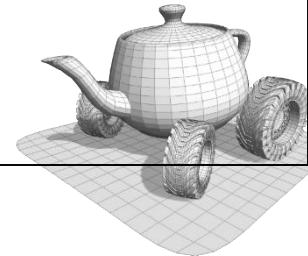
$$(\vec{\omega}_i \cdot \vec{n})$$

Approximation 2 : simplifying material

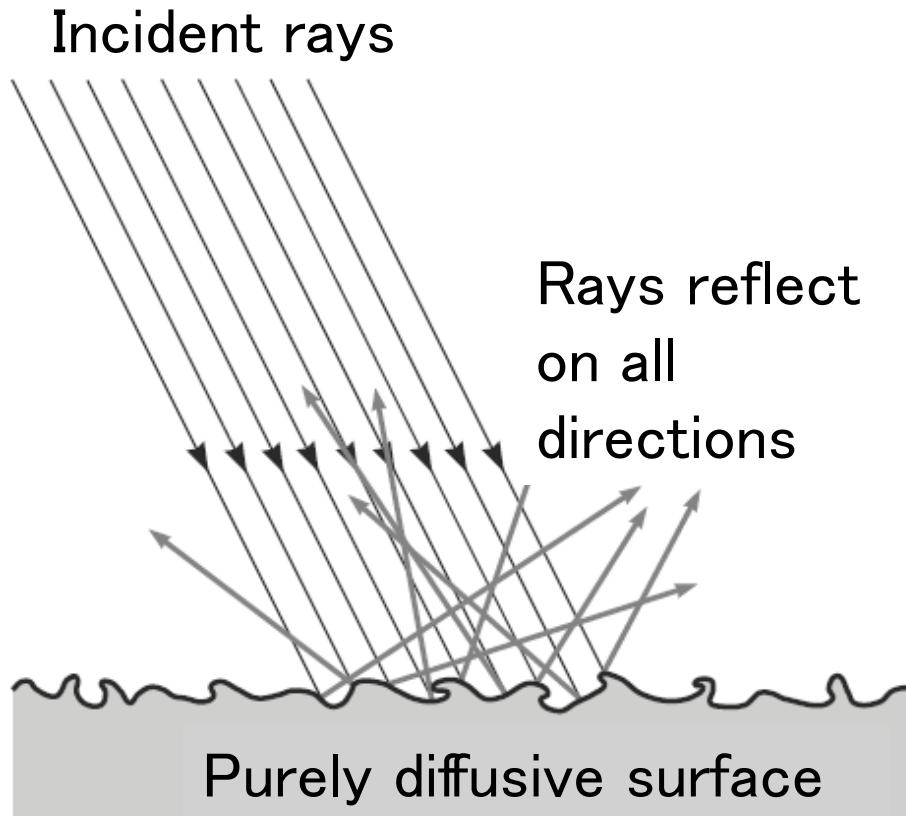


- Constant BRDF
 - Independent from direction of incoming light
 - Constant on outgoing direction, that is, independent from the point of view
- Are there real material with constant BRDF?
 - Yes: sand, chalk,
 - Opaque, dull (not glossy) material (no reflexes)
 - They are also called «purely diffusive» or «Lambertian»

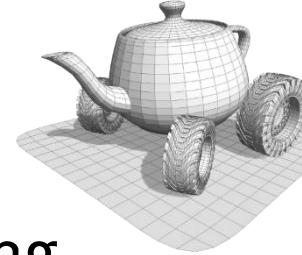
Approximation 2 : purely diffusive material



- It happens when
 - At microscopic level...
The surface show micro-facet randomly oriented



Approximation 2 : purely diffusive material



- From a function varying over incident and outgoing direction

$$\sum_{i=0}^{N-1} \dots \cdot f_r(x, \vec{\omega}_i, \vec{\omega}_r) \cdot \dots$$

- To a constant, called «albedo»

$$\sum_{i=0}^{N-1} \dots \cdot B_x \cdot \dots$$

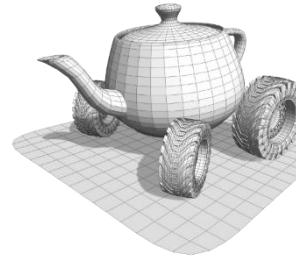
albedo (of point x):
ratio between incident and
outgoing light (to and from x).

- For colored material, 3 constants:

$$\sum_{i=0}^{N-1} \dots \cdot \begin{pmatrix} D_x^R \\ D_x^G \\ D_x^B \end{pmatrix} \cdot \dots$$

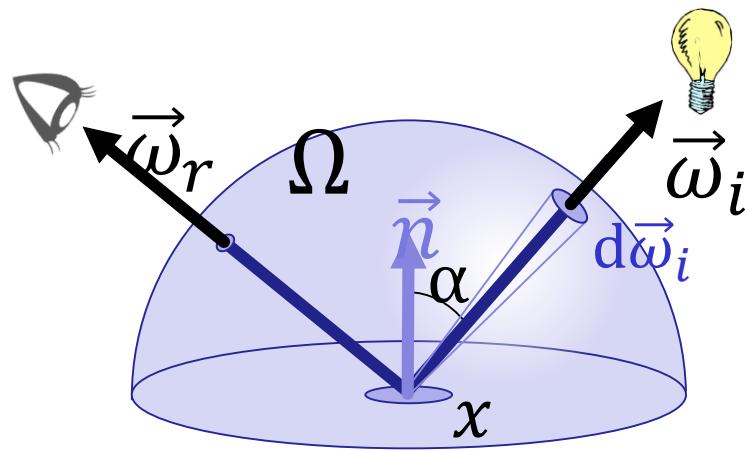
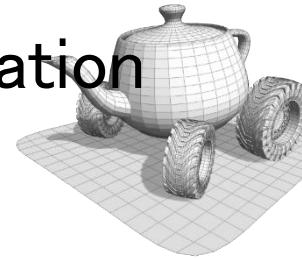
diffusive color (or “base
color”) (of point x)

Approximation 2 : recap



- A purely diffusive material (or Lambertian material) is described with a single parameter
 - **Albedo** – if we are not interested in colors
 - **Base Color** (or **diffuse color**) – for rendering colors
- the **Diffuse Color** is the intuitive answer to the question «what's the color of this object»
- Like all the properties of the materials, **albedo** / **base color** can be:
 - constant for the object (GLSL «uniform» variable)
 - Varying over the object:
 - per vertex GLSL «attribute»
 - per fragment (see texturing later on..)

Recap: the purely diffuse approximated lighting equation

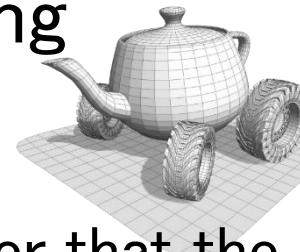


$$L_r = \sum_{i=0}^{N-1} \begin{pmatrix} D_x^R \\ D_x^G \\ D_x^B \end{pmatrix} \cdot \begin{pmatrix} L_i^R \\ L_i^G \\ L_i^B \end{pmatrix} \cdot (\vec{\omega}_i \cdot \vec{n})$$

Light color/diffuse color

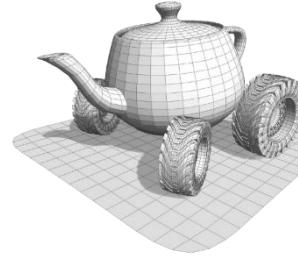
base color/diffuse color

Recap: the purely diffuse approximated lighting equation



- This approximated lighting equation is considerably simpler than the original one
 - Diffusive materials
 - Material described by constant BDRF (1 or 3 parameters)
 - But it is still **physically based**, and energy is conserved
 - That is, reflection does not increase or decrease the total amount of incoming light
 - ..although only for the intensity
 - That is, no frequency changing reflections
- However, it is not expressive enough to model interesting materials.
- Next, Phong illumination model..

Phong illumination model



Final light

=

One RGB component for each term

ambient

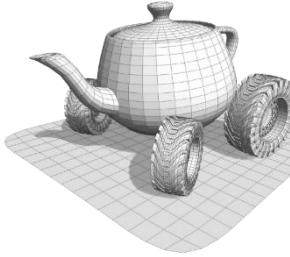
+

reflection

+

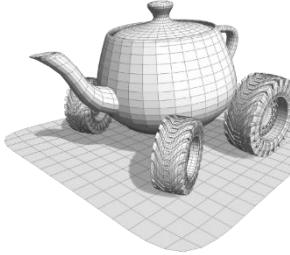
emission

Emissive term



- LEDs, light bulbs, a stub...
- Does not depend on lights, does not lit objects, it only lit our eye
- Just an additive component
 - Constant for R, G e B
- Only models light from the emitter to the eye
 - not: light-emitter → object → eye

Phong illumination model



Final light

=

ambient

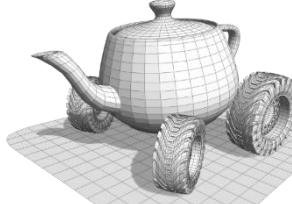
+

reflection

+

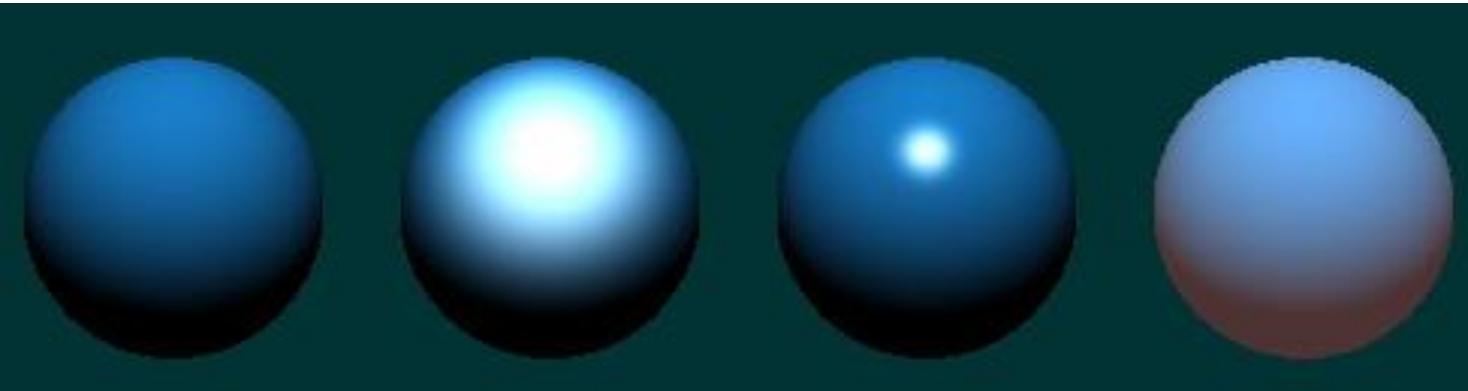
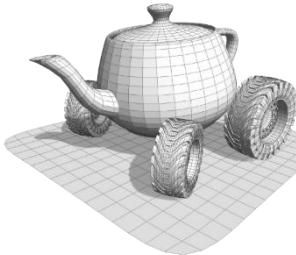
emission

Ambient component

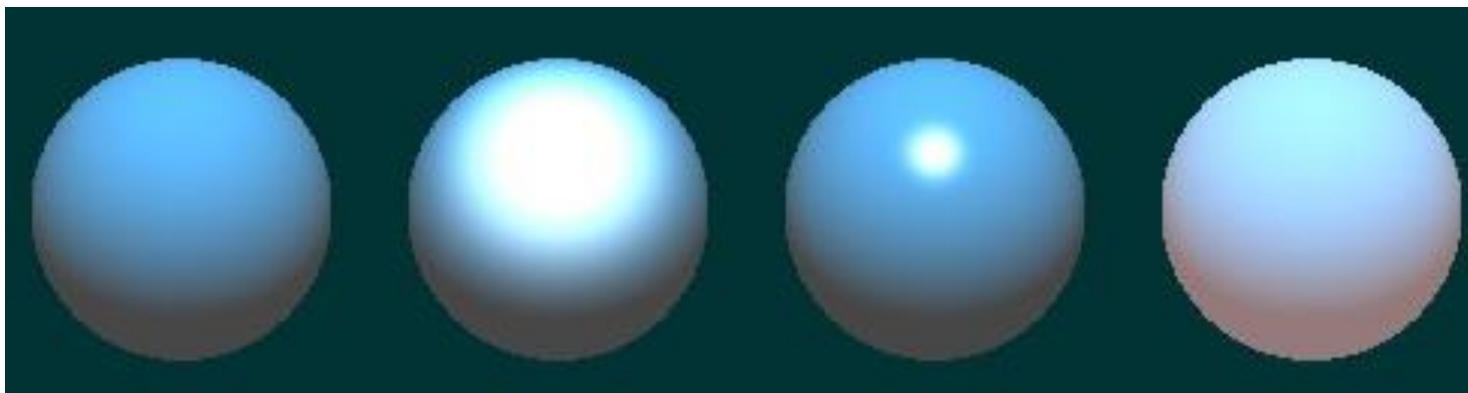


- A brutal approximation of the global illumination, that is, incident light from multiple reflections
- It's a (usually) small additive constant
 - Note: independent from any other parameter
- It means: every point get some light somehow
 - Even points in shadow
 - Even inside a closes box
 - No matter what
- It's defined as the product between:
 - “ambient color” of material ($R_M \ G_M \ B_M$)
 - color of the light ($R_L \ G_L \ B_L$)

Ambient component

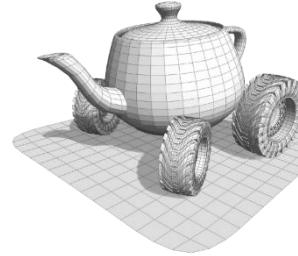


without



with

Phong illumination model



Final light

=

ambient

+

reflection

+

emission

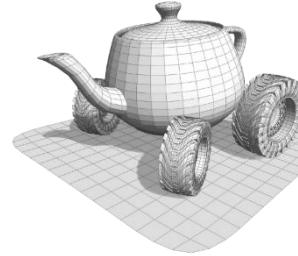


diffuse reflection

+

specular reflection

Phong illumination model



Final light

=

ambient

+



Diffusive reflection

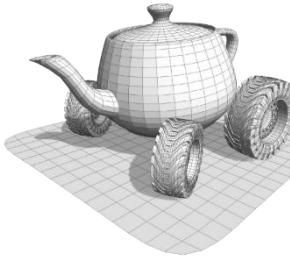
+

Specular reflection

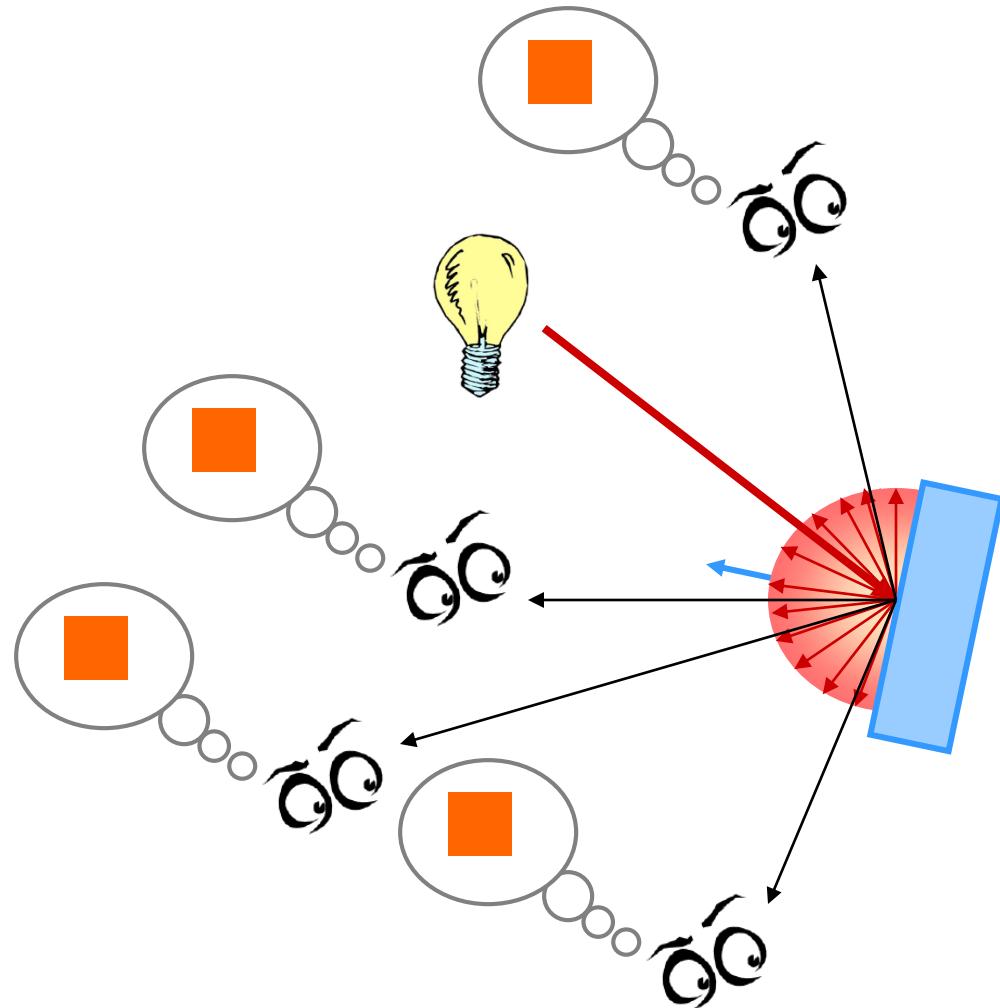
+

emission

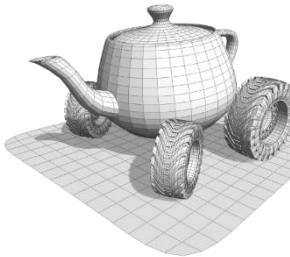
Diffuse reflection term



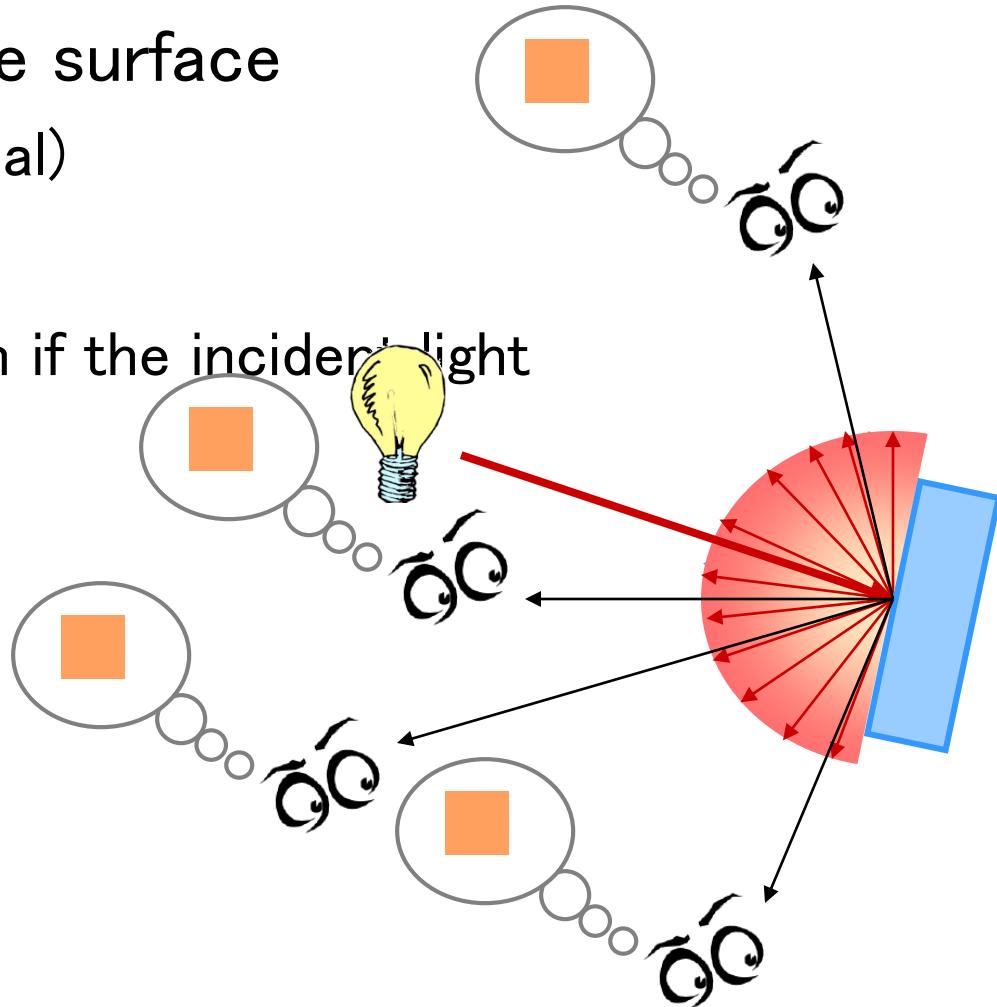
- Light hitting a Lambertian surface reflect equally on all directions



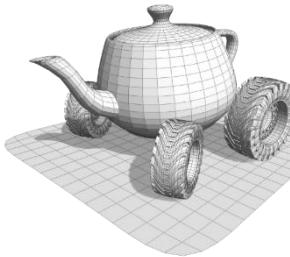
Diffuse reflection term



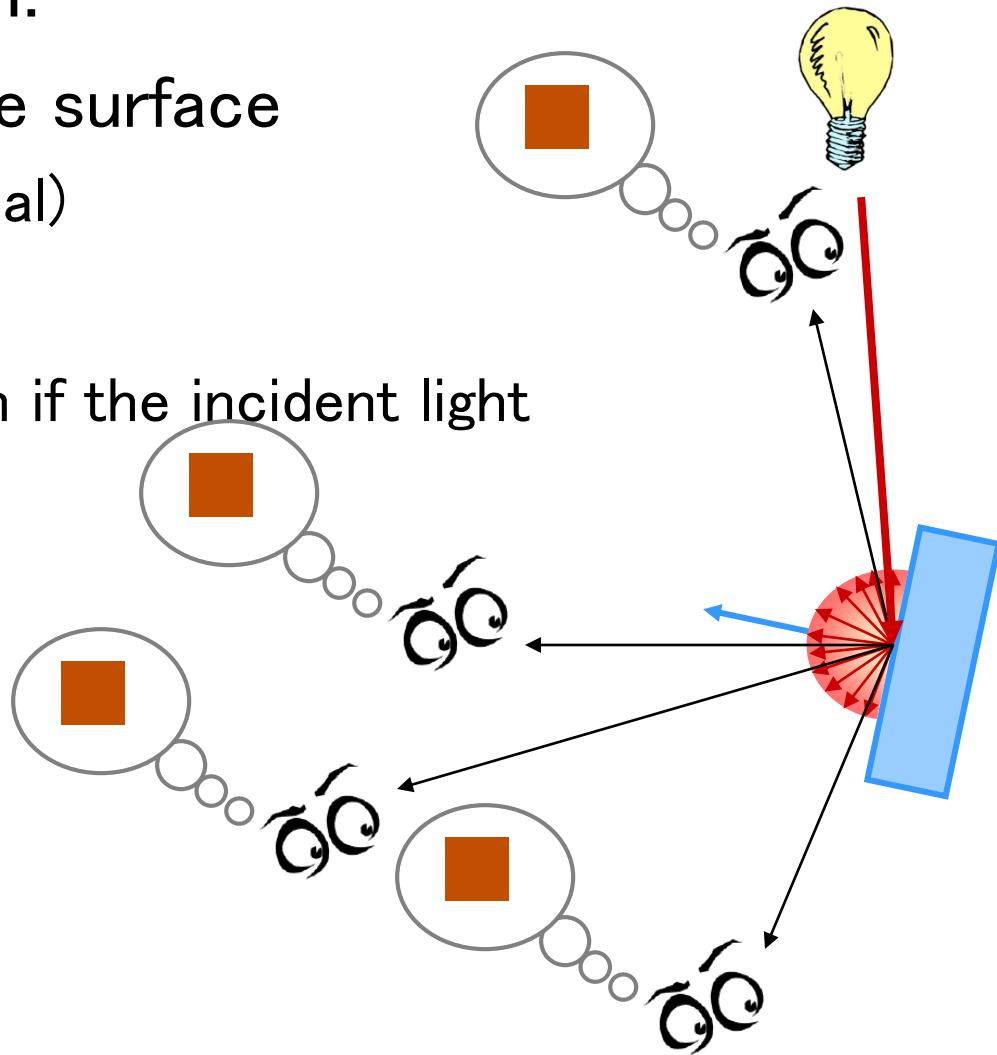
- It only depends on:
 - orientation of the surface
 - (that is, its normal)
 - light direction
 - That is, direction if the incident light (cosine law)



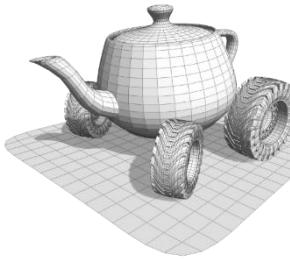
Diffuse reflection term



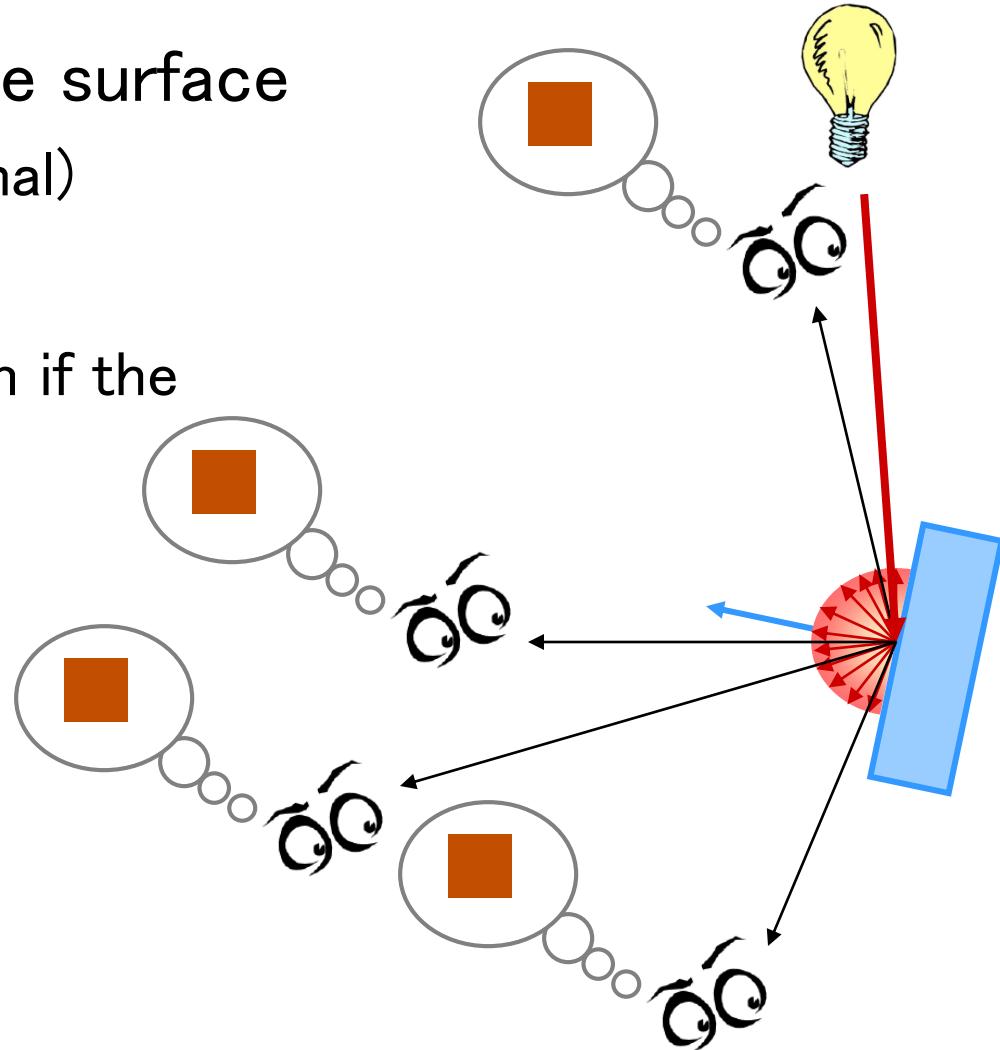
- It only depends on:
 - orientation of the surface
 - (that is, its normal)
 - light direction
 - That is, direction if the incident light (cosine law)



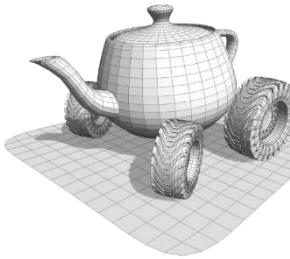
Diffuse reflection term



- It only depends on:
 - orientation of the surface
 - (that is, its normal)
 - light direction
 - That is, direction if the incident light
(cosine law)



Diffuse reflection term



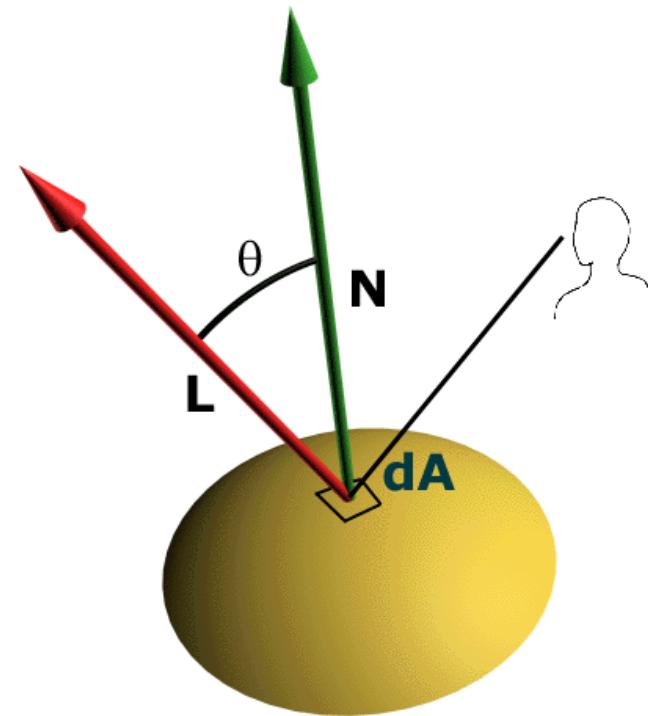
- It only depends on:
 - orientation of the surface
 - (that is, its normal)
 - light direction
 - That is, direction of the incident light
(cosine law)

$$I_{diff} = I_{lucediff} \cdot k_{diff} \cdot \cos \theta$$

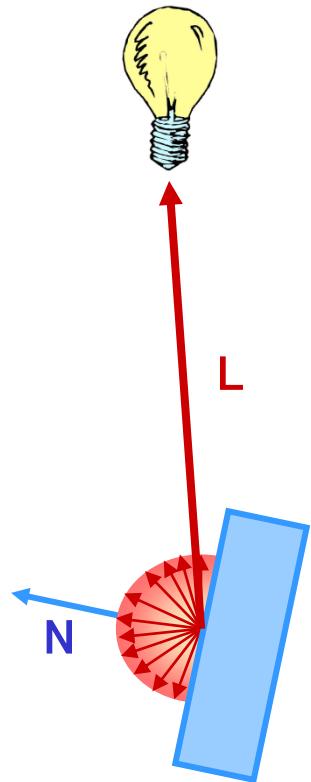
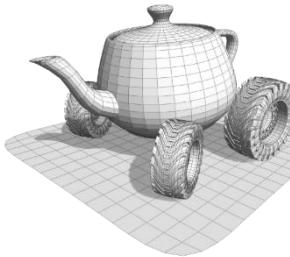
R, G, B

R, G, B
(diffuse color)

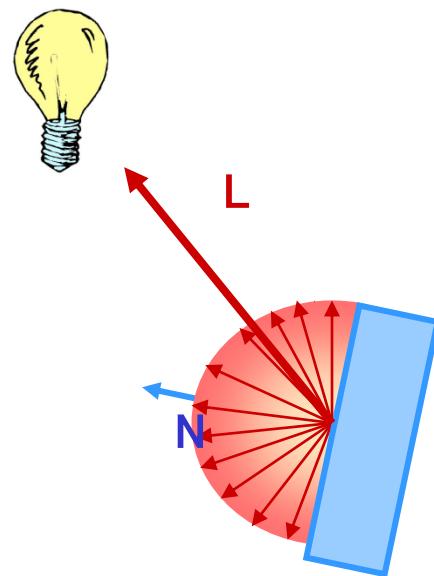
Component-wise multiplication



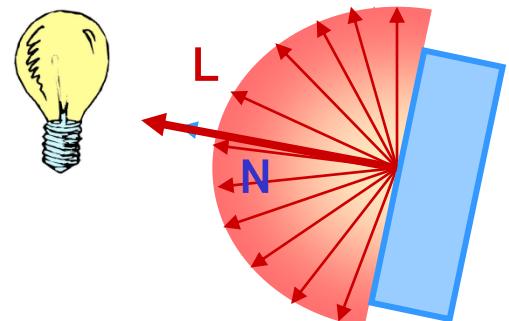
Diffuse reflection term



Small diffuse term
 $\theta=70^\circ$

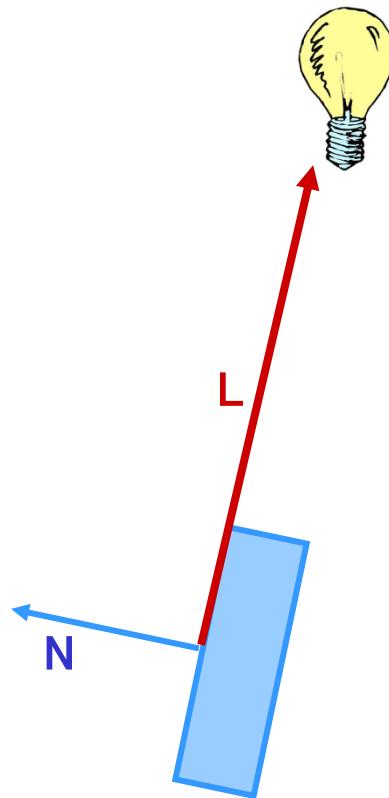
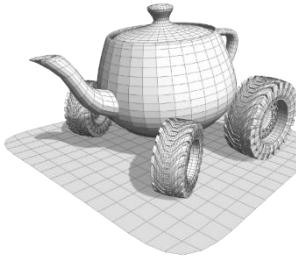


Big diffuse term
 $\theta=35^\circ$

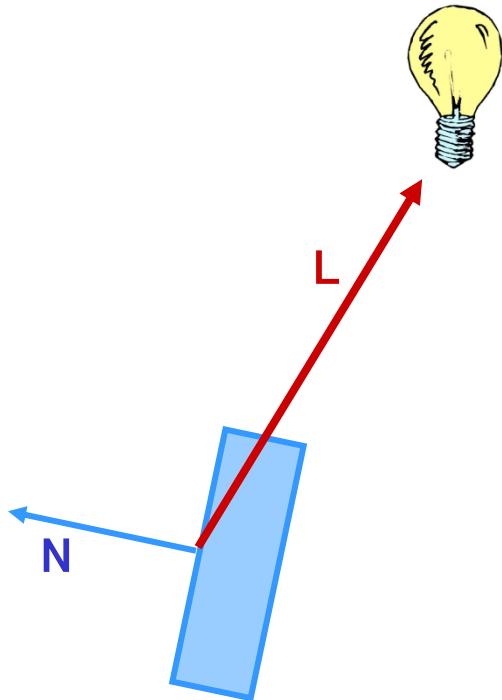


Maximal diffuse term
 $\theta=0^\circ$

Diffuse reflection term



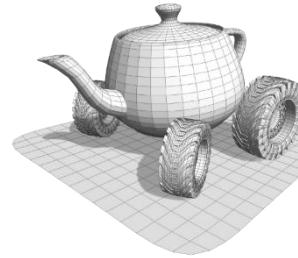
ZERO
diffuse term
 $\theta=90^\circ$



ZERO
diffuse term
 $\theta>90^\circ$

(surface is in its own shadow)

Phong illumination model



Final light

=

ambient

+

Diffuse reflection

+

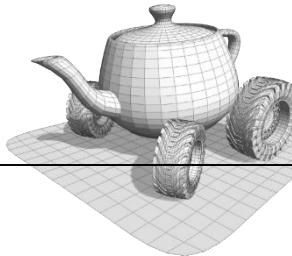
Specular reflection

+

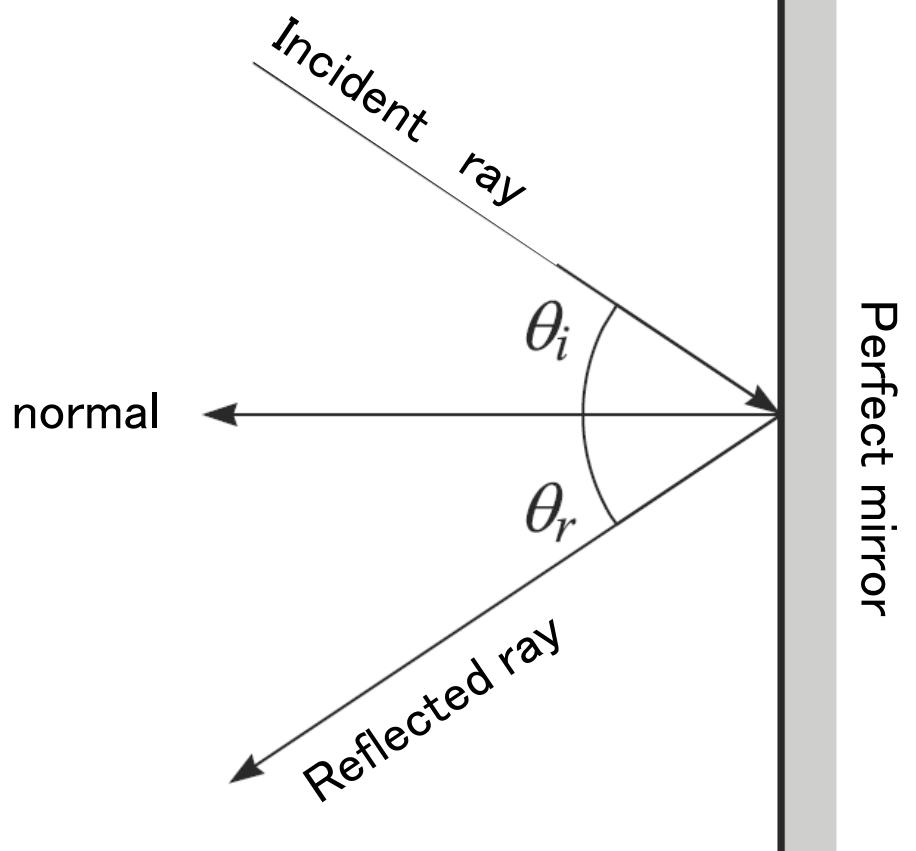
emission



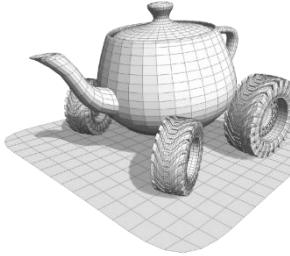
Specular term



- Conceptually simple phenomenon
 - Photons bounces just like balls
 - With the same angle w.r.t. to the normal as they arrive
 - A perfect mirror
- **Microscopically:** normals of the microfacets are coherent with the macroscopic normal



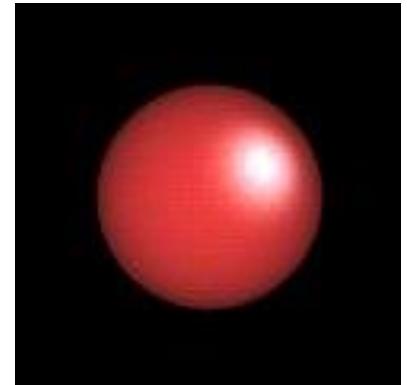
Specular term



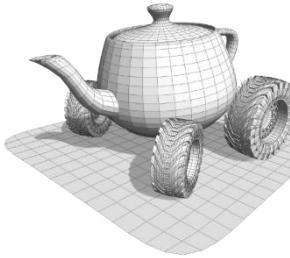
- “Specular” reflection
 - Or «Phong» reflection
- Shiny materials
 - With highlights



without

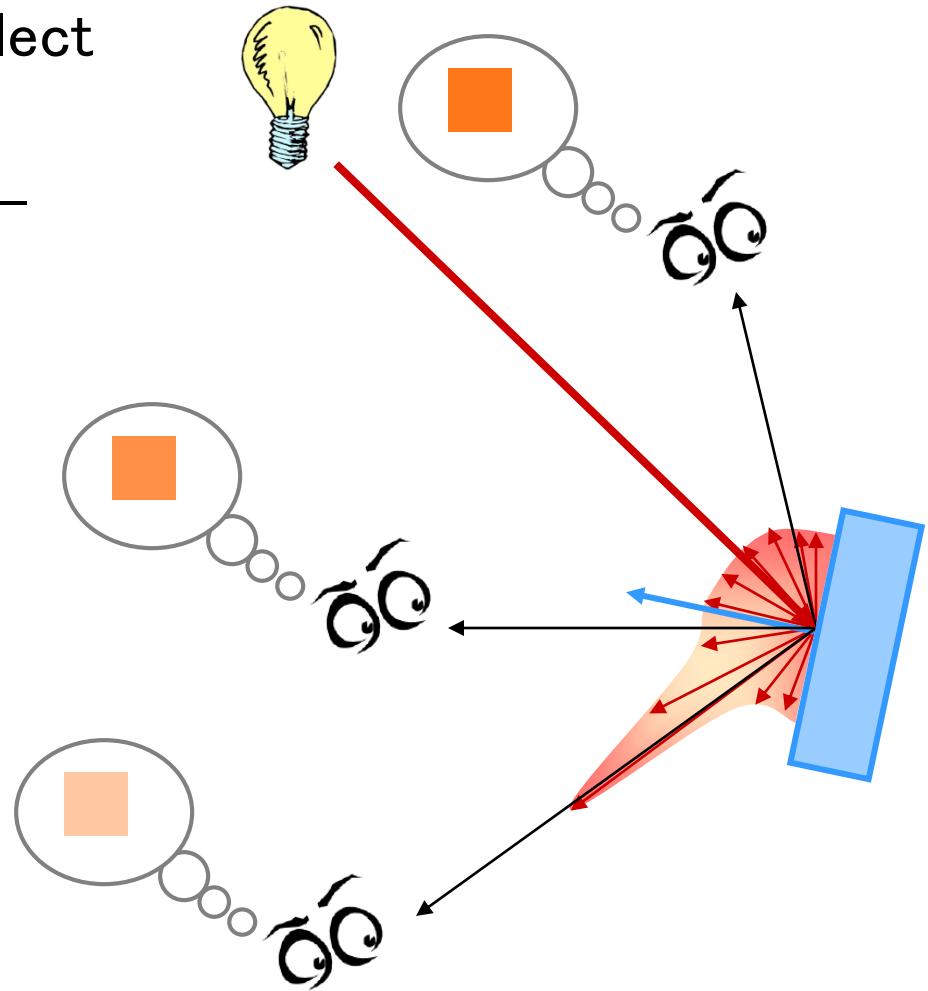


with

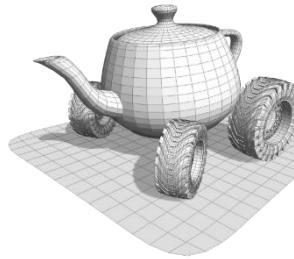


Specular term

- Idea:
glossy material do not reflect light equally in all directions
- Specular reflection is view-dependent



Specular term

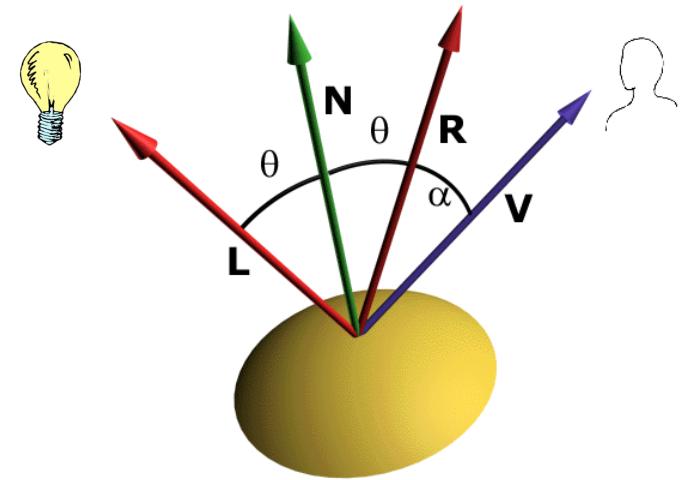
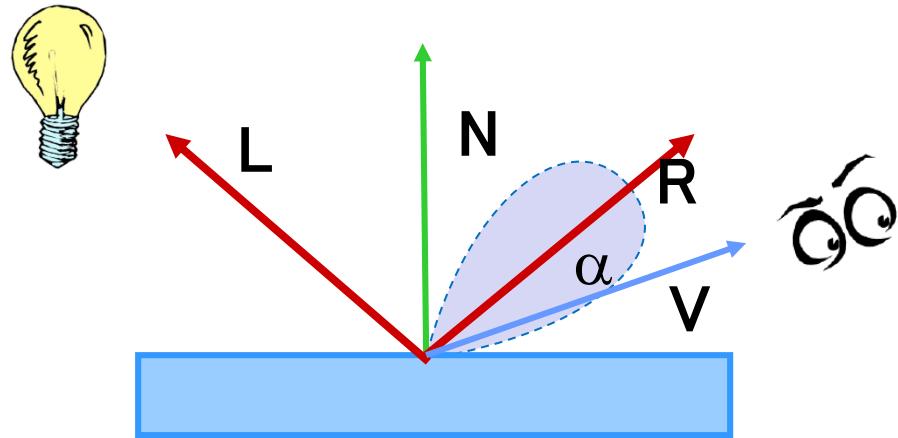


L: incident ray

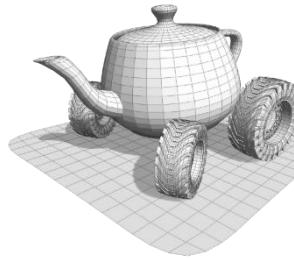
N: normal

R: reflected ray

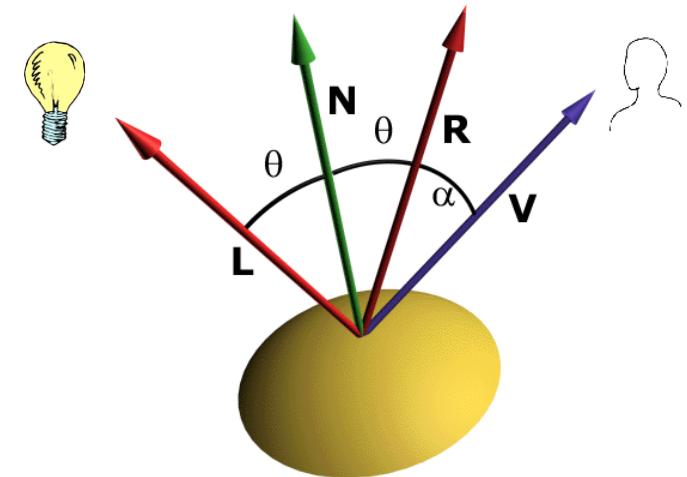
V: view direction



Specular reflection term

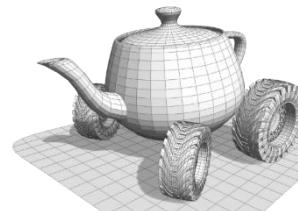


- Phong light model
 - by Bui-Tuong Phong, 1975
- Amount of light toward view direction is proportional to the cosine of the angle w.r.t. to the specular direction

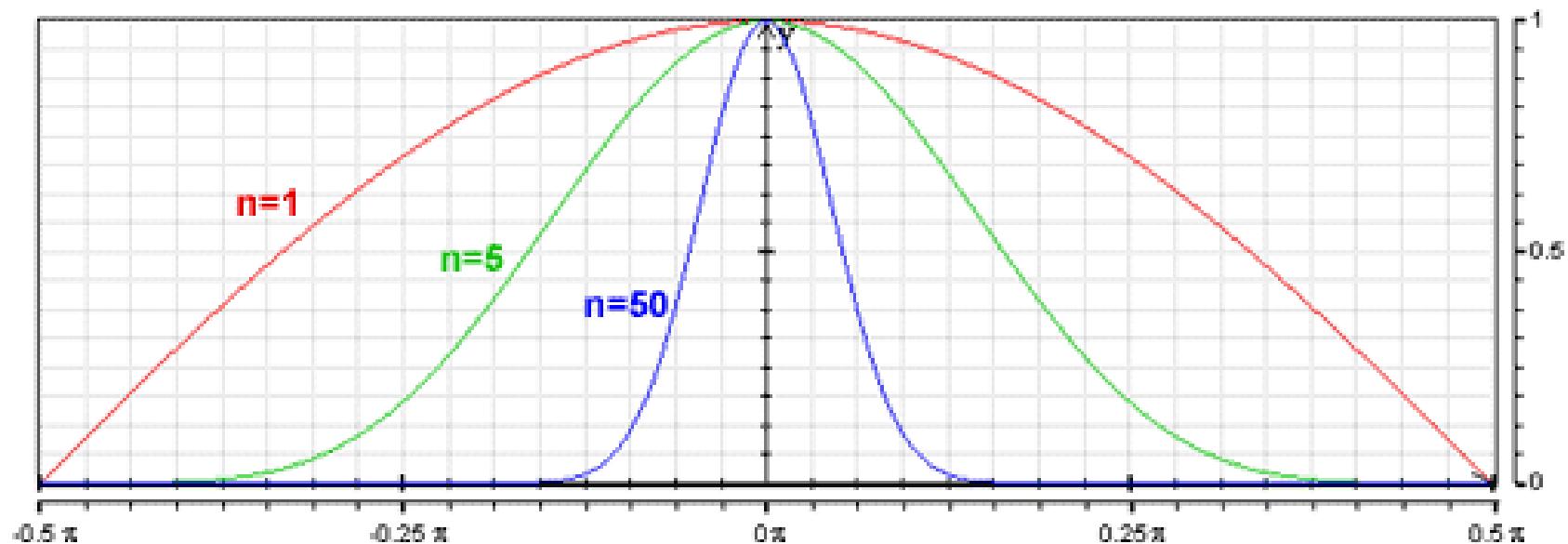


$$I_{spec} = I_{luce\ spec} \cdot k_{materiale\ spec} \cdot \cos \alpha$$

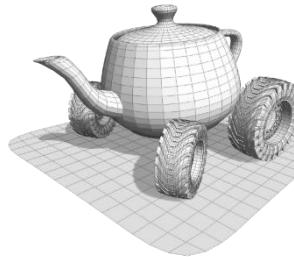
Specular reflection term



- Raising the cosine to a power > 1 we have smaller and shinier specular reflections



Specular reflection term



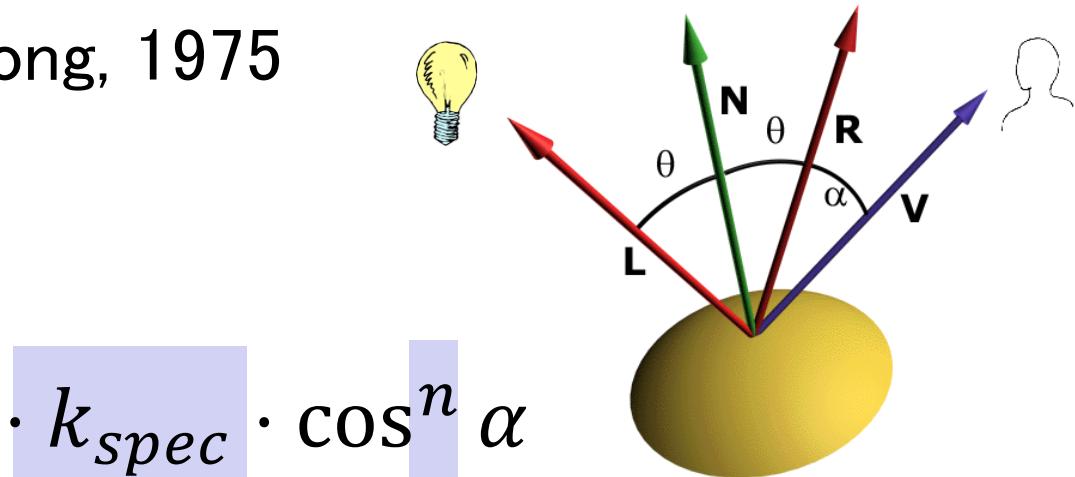
- Phong light model
 - by Bui-Tuong Phong, 1975

$$I_{spec} = I_{lucespec} \cdot k_{spec} \cdot \cos^n \alpha$$

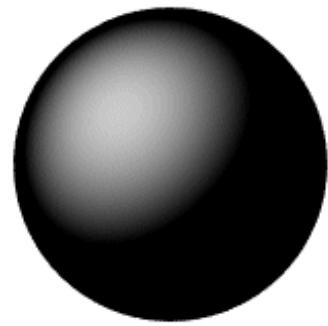
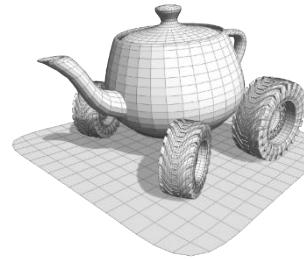
“shininess”
(intensity of the
reflexes)

“glossiness”
(how focused the
reflexes are)

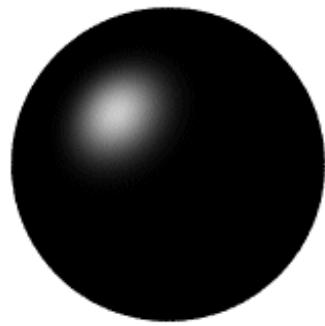
Part of the object material



Specular reflection term: glossiness



$n = 1$



$n = 5$

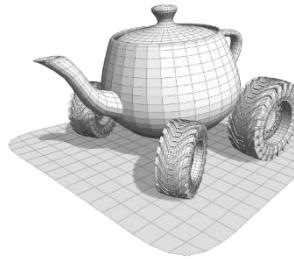


$n = 10$

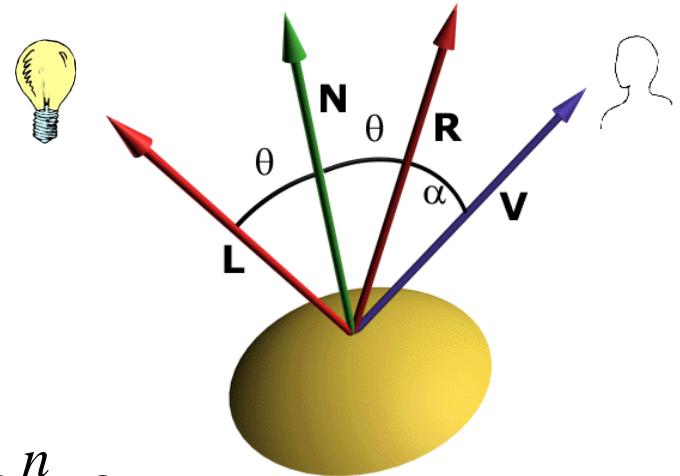


$n = 100$

Specular reflection term



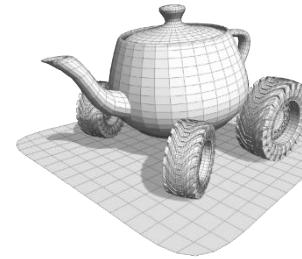
- Phong light model
 - by Bui-Tuong Phong, 1975



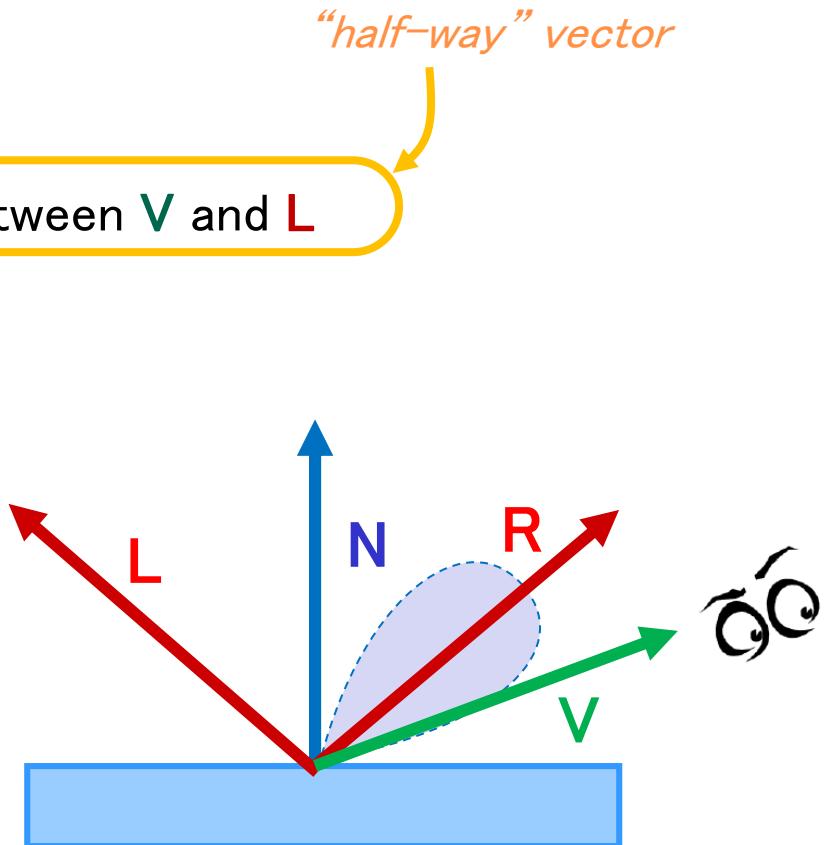
$$I_{spec} = I_{luce\ spec} \cdot k_{materiale\ spec} \cdot \cos^n \alpha$$

$$= I_{lucespec} \cdot k_{materialspec} \cdot (\hat{R} \cdot \hat{V})^n$$

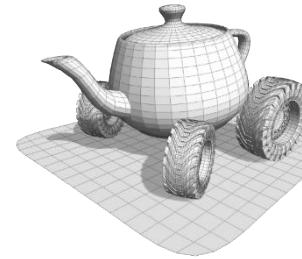
Specular reflection term: Blinn's variant



- Blinn–Phong light model:
 - Simplification of Phong light model
 - Similar results, slightly different formula
 - idea: V is almost the same as R
iff
 N is almost the average between V and L
 - (so we don't have to compute R)

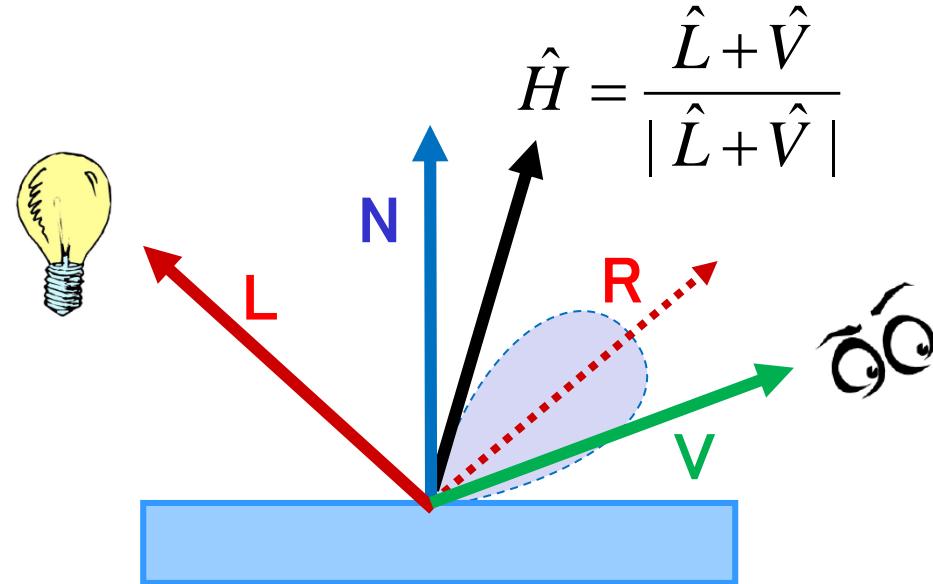


Specular reflection term: Blinn's variant



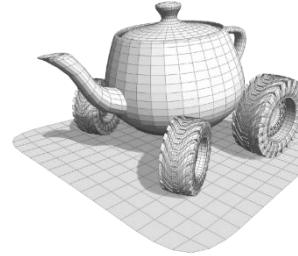
phong: $I_{spec} = I_{lucespec} \cdot k_{material\&spec} \cdot (\hat{R} \cdot \hat{V})^n$

blinn–phong: $I_{spec} = I_{lucespec} \cdot k_{material\&spec} \cdot (\hat{H} \cdot \hat{N})^n$



“half-way”
vector

I 4 fattori che consideriamo



Final light

=

ambient

+

Diffuse reflection

+

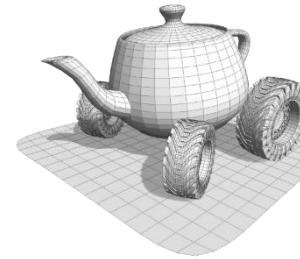
Specular reflection

+

emission



Classic *GL lighting equation

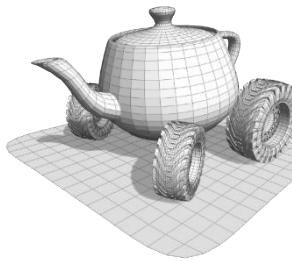


$$I_{tot} = I_{ambient} \cdot k_{ambientMat} + I_{diffuse} \cdot k_{diffuseMat} \cdot (\hat{L} \cdot \hat{N}) + I_{specular} \cdot k_{specularMat} \cdot (\hat{H} \cdot \hat{N})^n + k_{emissiveMat} \frac{(\hat{L} + \hat{V})}{|\hat{L} + \hat{V}|}$$

Properties of material

Properties of light

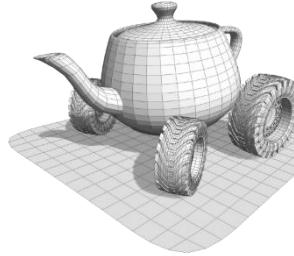
Materials...



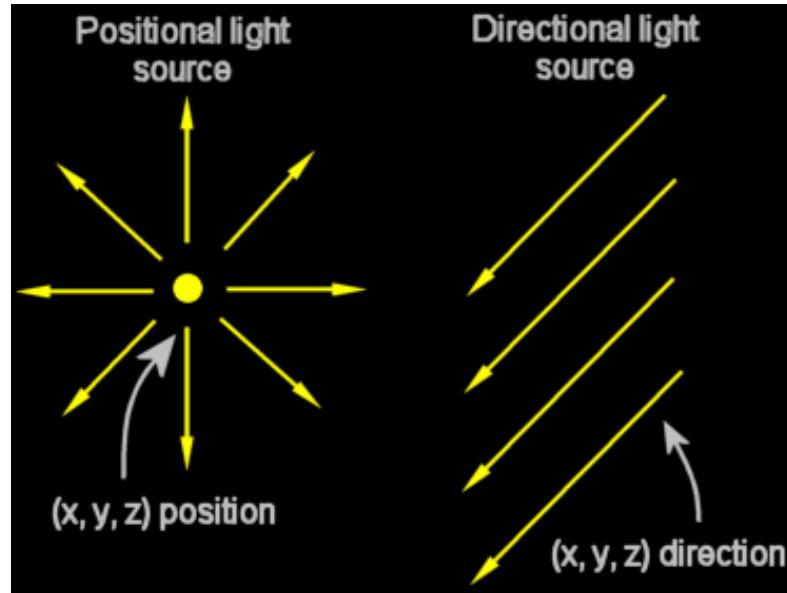
Material	GL_AMBIENT	GL_DIFFUSE	GL_SPECULAR	GL_SHININESS
Emerald	0.0215	0.07568	0.633	
	0.1745	0.61424	0.727811	
	0.0215	0.07568	0.633	
	0.55	0.55	0.55	76.8
Jade	0.135	0.54	0.316228	
	0.2225	0.89	0.316228	
	0.1575	0.63	0.316228	
	0.95	0.95	0.95	12.8
Obsidian	0.05375	0.18275	0.332741	
	0.05	0.17	0.328634	
	0.06625	0.22525	0.346435	
	0.82	0.82	0.82	38.4
Pearl	0.25	1.0	0.296648	
	0.20725	0.829	0.296648	
	0.20725	0.829	0.296648	
	0.922	0.922	0.922	11.264
Ruby	0.1745	0.61424	0.727811	
	0.04175	0.04136	0.626959	
	0.04175	0.04136	0.626959	
	0.55	0.55	0.55	76.8
Turquoise	0.1	0.396	0.297254	
	0.18725	0.74151	0.30829	
	0.1745	0.69102	0.306678	
	0.8	0.8	0.8	12.8
Black Plastic	0.0	0.01	0.50	
	0.0	0.01	0.50	
	0.0	0.01	0.50	
	1.0	1.0	1.0	32
Black Rubber	0.02	0.01	0.4	
	0.02	0.01	0.4	
	0.02	0.01	0.4	
	1.0	1.0	1.0	10
Brass	0.329412	0.780392	0.992157	
	0.223529	0.568627	0.941176	
	0.027451	0.113725	0.807843	
	1.0	1.0	1.0	27.8974
Bronze	0.2125	0.714	0.393548	
	0.1275	0.4284	0.271906	
	0.054	0.18144	0.166721	
	1.0	1.0	1.0	25.6
Polished Bronze	0.25	0.4	0.774597	
	0.148	0.2368	0.458561	
	0.06475	0.1036	0.200621	
	1.0	1.0	1.0	76.8
Chrome	0.25	0.4	0.774597	
	0.25	0.4	0.774597	
	0.25	0.4	0.774597	
	1.0	1.0	1.0	76.8



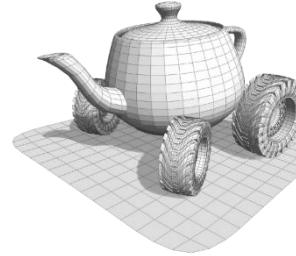
Types of light



- What can L be?
 - **directional light:** constant over the entire scene
 - It models very far away light, like the sun
 - **Positional light:** varies over the scene
 - It models nearby light, e.g. lightbulbs

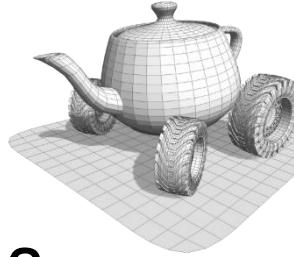


Directional and Positional Lights in our code



- Directional light:
 - The direction L is a uniform GLSL variable
- Positional light:
 - The position of the light \mathbf{p} is a uniform
 - The direction of light must be computed for each point \mathbf{x}
 - $$L = \frac{\mathbf{p}-\mathbf{x}}{\|\mathbf{p}-\mathbf{x}\|}$$

Positional light



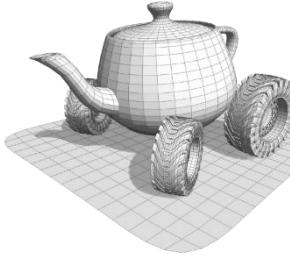
- We can introduce an **attenuation factor** as a function of the **distance**
- Physically that would be:

$$f_{attenuation} = \left(\frac{1}{c \cdot d_L^2} \right)$$

Arbitrary factor

distance m

Positional light

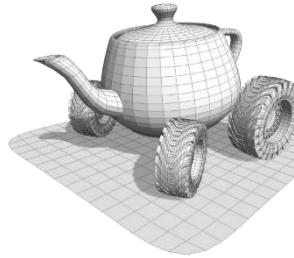


- In practical terms, that is too much
- Often replaced with a polynomial:

$$f_{attenuation} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right)$$

↑ ↑ →
Arbitrary factors

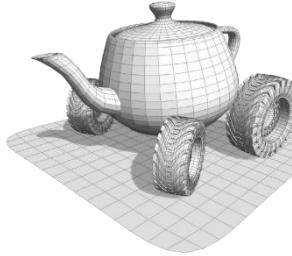
Modelling lights



$$I_{tot} = \left. \begin{aligned} I_{ambient} &\cdot k_{ambientMat} + \\ I_{diffuse} &\cdot k_{diffuseMat} \cdot (\hat{L} \cdot \hat{N}) + \\ I_{specular} &\cdot k_{specularMat} \cdot (\hat{H} \cdot \hat{N})^n + \\ k_{emissiveMat} \end{aligned} \right\} \cdot f_{attenuation}$$

$$f_{attenuation} = \min \left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1 \right)$$

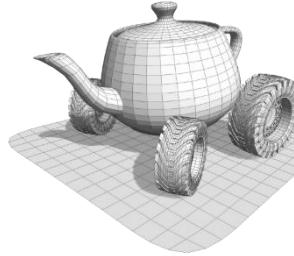
Types of light



- What can L be?
 - **directional light:** constant over the entire scene
 - It models very far away light, like the sun
 - **Positional light:** varies over the scene
 - It models nearby light, e.g. lightbulbs
 - **Spotlights**

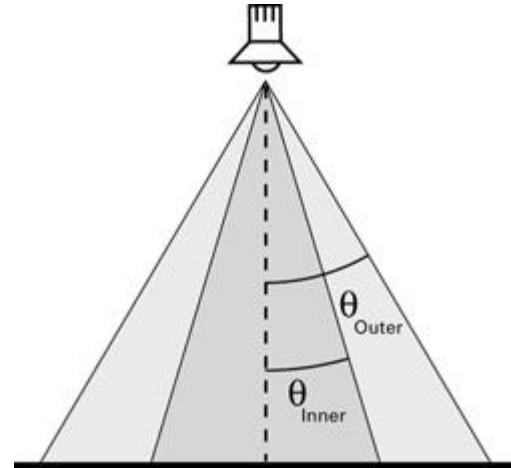


Spotlights



- Defined by:
 - Position of the spotlight
 - Direction of the spotlight
 - Opening angle (beam width)
 - Cutoff angle

nNidia CG tutorials

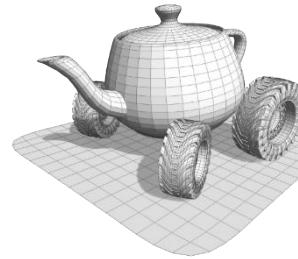


$$f_s = \begin{cases} 1 & \alpha < \theta_{inner} \\ \cos(\alpha - \theta_{inner})^n & \theta_{inner} < \alpha < \theta_{outer} \\ 0 & \alpha > \theta_{outer} \end{cases}$$

α : angle between ω and S_{dir}

$$f_s = f(S_{dir}, \theta_{inner}, \theta_{outer}, n)$$

Spotlights

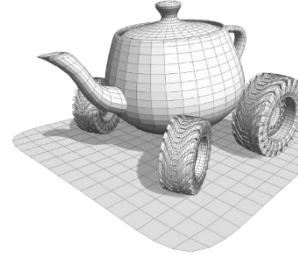


$$I_{tot} = \left[I_{ambient} \cdot k_{ambientMat} + \right. \\ \left. I_{diffuse} \cdot k_{diffuseMat} \cdot (\hat{L} \cdot \hat{N}) \cdot f_s + \right. \\ \left. I_{specular} \cdot k_{specularMat} \cdot (\hat{H} \cdot \hat{N})^n \cdot f_s + \right] \cdot f_{attenuation}$$
$$k_{emissiveMat} \cdot f_s$$

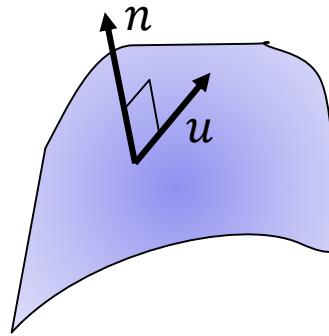
$$f_{attenuation} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right)$$

$$f_s = f(L, spot_{direction}, spot_{cutoffAngle}, spot_{beamwidth})$$

Transforming the normal (1/3)

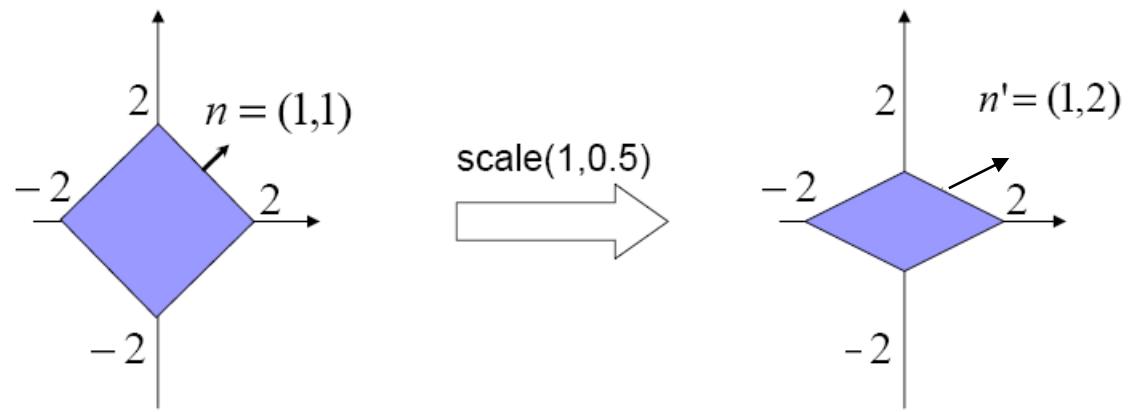
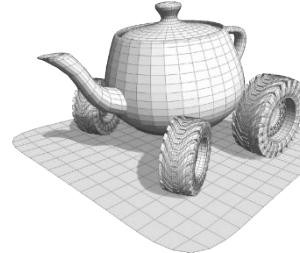


- We need the normal to compute the lighting equations
- The normal is defined in object space and hence need to be transformed
- Consider a vector \mathbf{u} tangent to the surface and the normal \mathbf{n} . Since they are orthogonal $\mathbf{n}^T \mathbf{u} = 0$



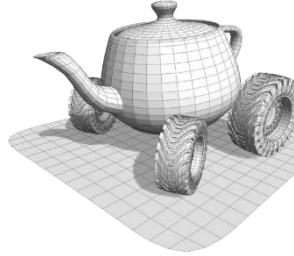
- If M is the transformation from object space to view (or world) space, can we say that the transformed n and u still orthogonal?
 $(M\mathbf{n})^T(M\mathbf{u}) = 0$?
- No

Transforming the normal (3/3)



$$n' = \begin{pmatrix} 1 & 0 \\ 0 & 0.5 \end{pmatrix} n = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}$$

Transforming the normal (2/3)

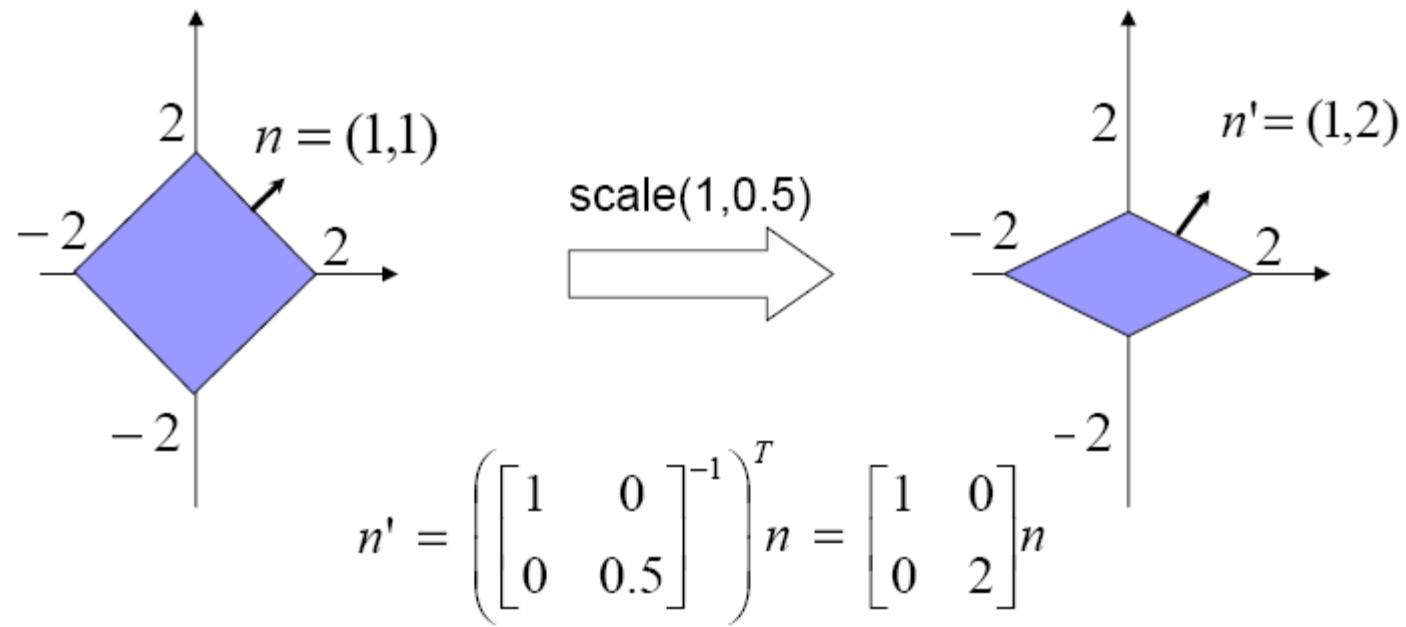
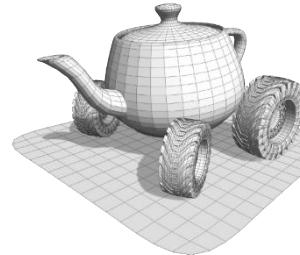


- $\mathbf{n}^T \mathbf{u} = 0 \Rightarrow \mathbf{n}^T \mathbf{M}^{-1} \mathbf{M} \mathbf{u} = 0 \Rightarrow (\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M} \mathbf{u}) = \mathbf{0}$
- Therefore the transformed normal \mathbf{n}' is:

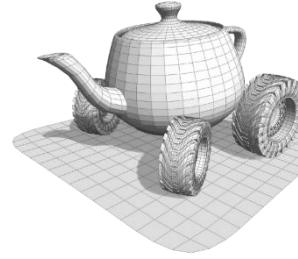
$$\begin{aligned}\mathbf{n}'^T &= (\mathbf{n}^T \mathbf{M}^{-1}) \Rightarrow \mathbf{n}' = (\mathbf{n}^T \mathbf{M}^{-1})^T \Rightarrow \\ \mathbf{n}' &= \mathbf{M}^{-1}{}^T \mathbf{n}\end{aligned}$$

- The normal vector must be transformed by the inverse transposed of the matrix transforming the points
- Is it *always* necessary?
 - No, if \mathbf{M} preserves lengths and angles is not necessary
 - if \mathbf{M} preserves the angles it is also not necessary, but if it does not preserve the lengths we need to normalize the result

Transforming the normal (3/3)

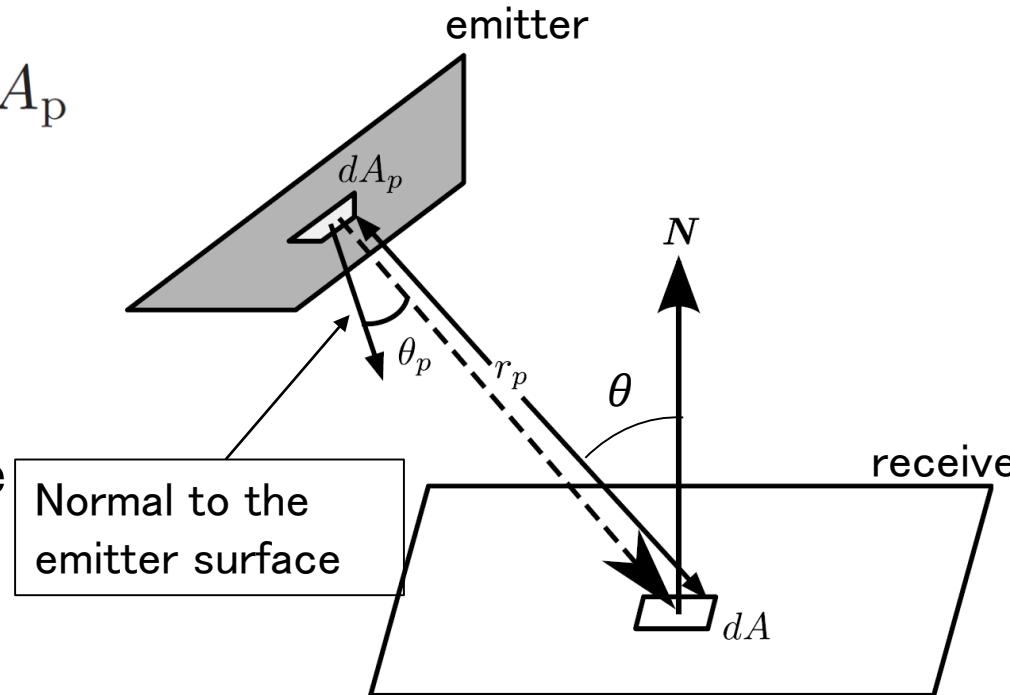


Area Lights

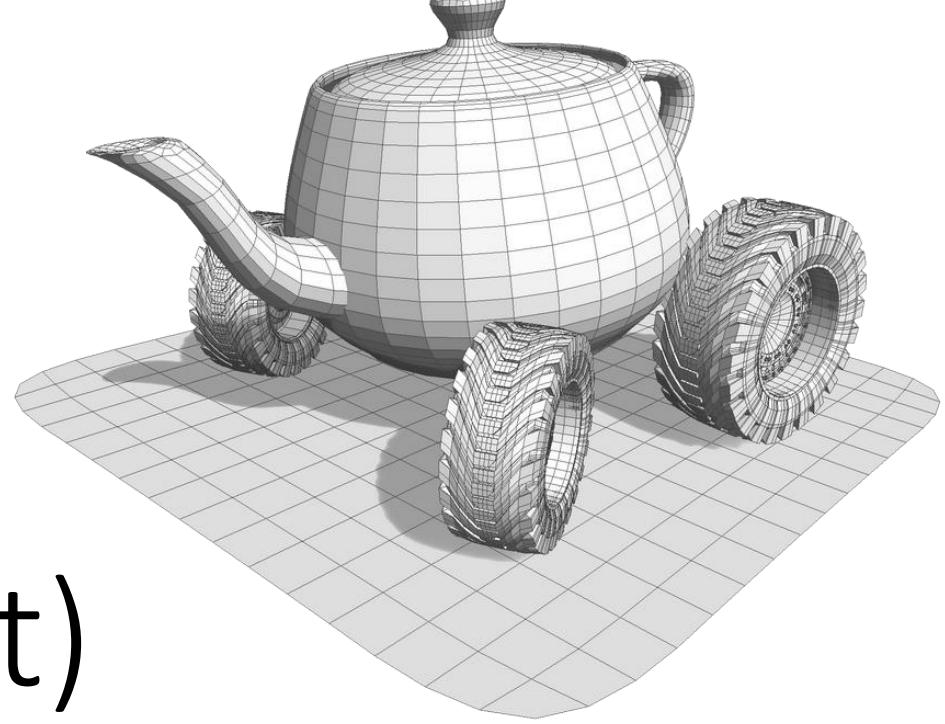


- The source is not a point or a direction but a whole area
- Even for constant radiance over the emitter and purely Lambertian surface, the equation for computing the reflected requires an integral

$$L_r = \frac{\rho}{\pi} L_p \int_{p \in A} \frac{\cos \theta \cos \theta_p}{r_p^2} dA_p$$

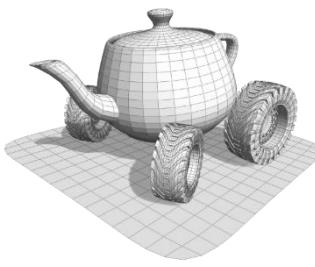


- It is commonly approximated with a series of point lights
- Beware: it may quickly become expensive



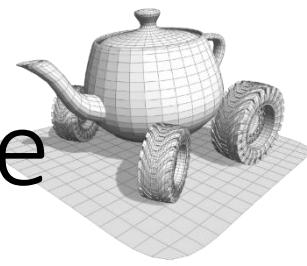
Lighting (cnt)

Advanced Reflection Models

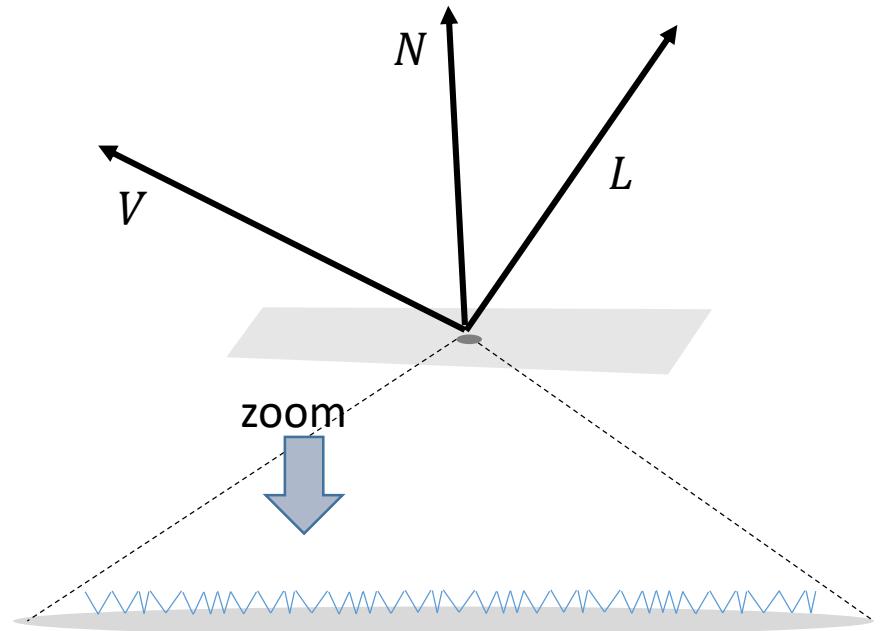


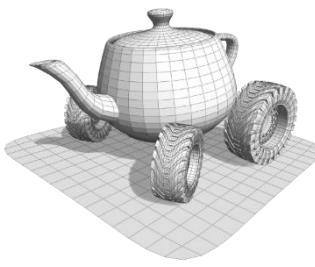
- Phong Lighting has been a longstanding standard *de facto* for many years
 - Hard coded in the fixed OpenGL pipeline
- However, programmable shaders paved the way to more expressive models
- •Next: two well-known shading models
 - Cook-Torrance
 - Oren-Nayar

Specular rough materials: the Cook-Torrance model



- Key idea: model the surface at a microscopic level as an assembly of many **microfacets** that act like mirrors
- The microfacets are isotropically distributed (in the neighborhood of the point lit)
- Each microfacet is a **perfect mirror**
- Unlike Phong specular component, Cook-Torrance is physically based

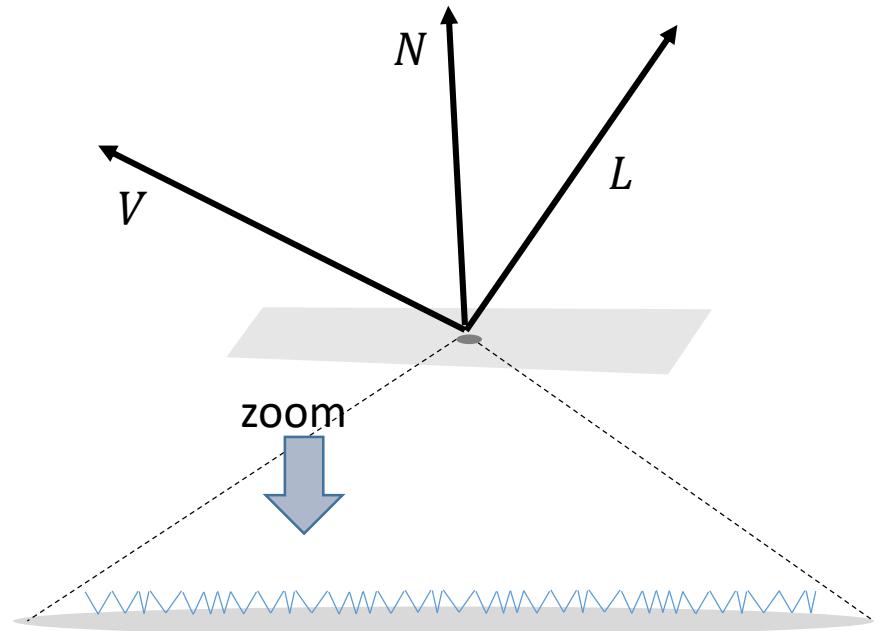


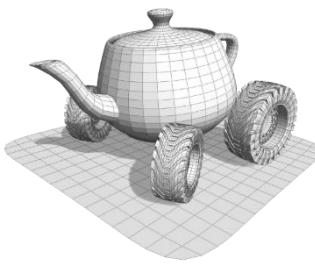


Cook-Torrance model

- The distribution of microfacets influence the amount of reflected light with three terms
 - D: the *roughness* term.
 - G: shadowing/masking effects among the microfacets
 - F: **Fresnel** reflection (more later..)

$$L_r = L_i \frac{D \cdot G \cdot F}{(N \cdot L)(N \cdot V)}$$





Cook-Torrance model

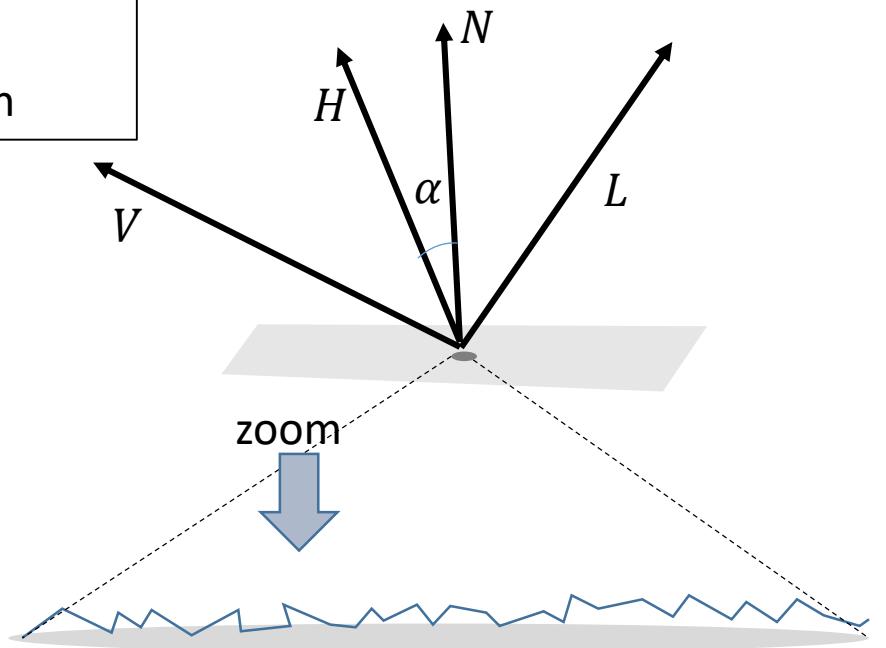
- The roughness term depends on the average slope of the microfacets m

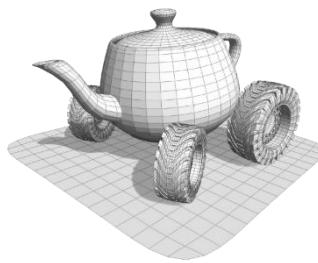
$$D = \frac{1}{m^2 \cos^4 \alpha} e^{-\frac{\tan^2 \alpha}{m^2}}$$

Spizzichino-
Beckmann
distribution

$$\alpha = \arccos(N \cdot H)$$

m : average slope of the
microfacets

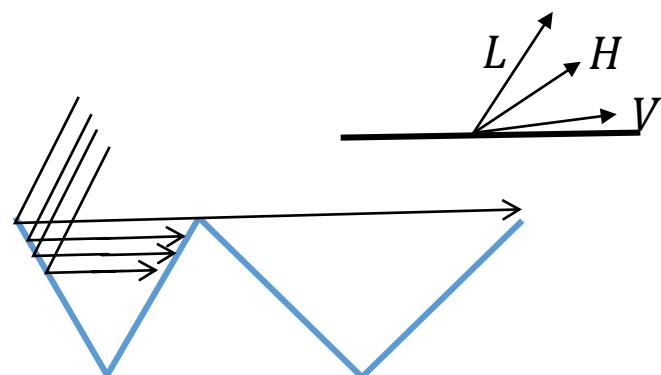
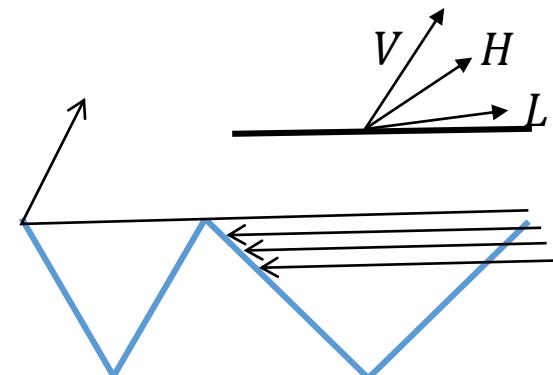




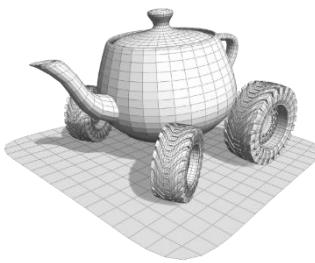
Cook-Torrance model

- G accounts for the self shadowing and masking of the microfaces
 - **Self shadowing:** a microfacet receive less light because it's occluded by another
 - **Masking:** a part of the light leaving a microfacet does not reach the eye because other microfacets along the path

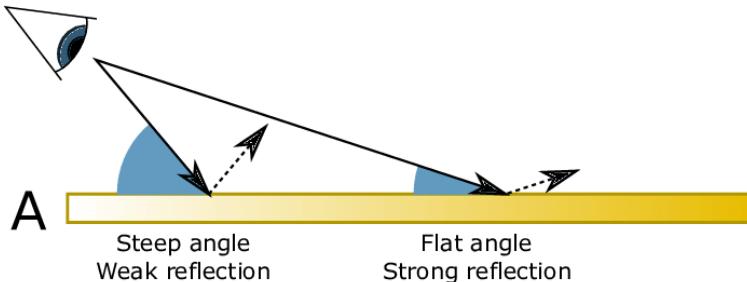
$$G_2 = \frac{2(N \cdot H)(N \cdot L)}{V \cdot H} \quad \text{Shadowing effect}$$
$$G_1 = \frac{2(N \cdot H)(N \cdot V)}{V \cdot H} \quad \text{Masking effect}$$
$$G = \min(1, G_1, G_2)$$



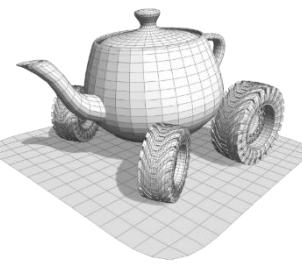
Cook-Torrance model



- **Fresnel effect:** light at a grazing angle reflects more than light at greater angles
 - It's a frequency dependent effect
 - note that this is ignored in Phong lighting
- Fresnel equations tell what fraction of light is reflected and what fraction is refracted when light hits the boundary between two different media



wikipedia

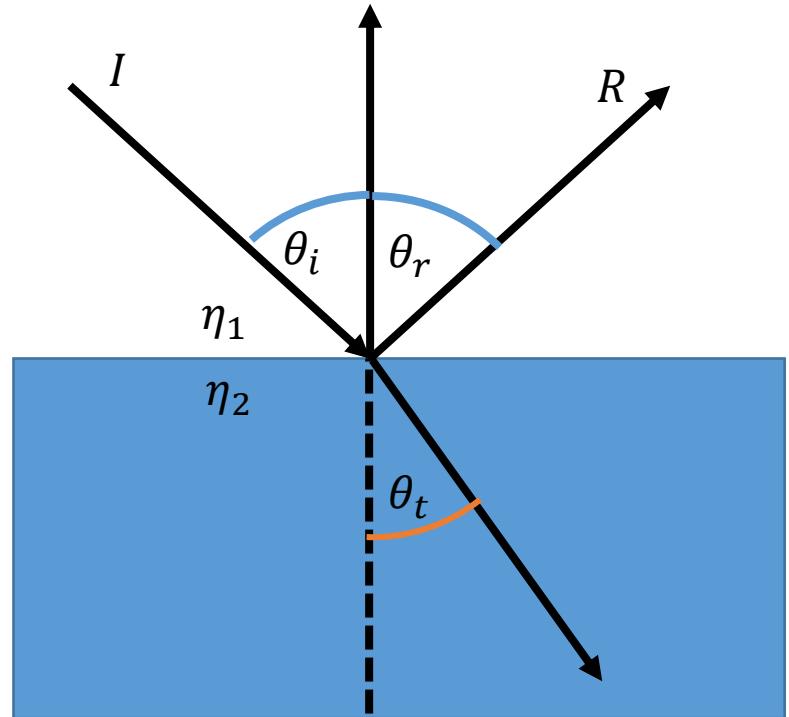


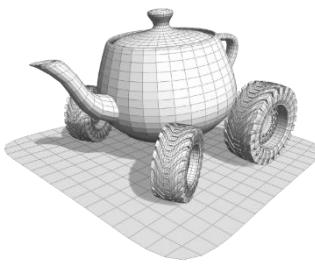
Cook-Torrance model

- For reflection: $\theta_i = \theta_r$
- The angle of refraction follows the **Snell's law**:

$$\eta_1 \sin \theta_i = \eta_2 \sin \theta_t$$

- Where η_1, η_2 are the **indices of refraction**
- What about the *amount* of reflected light?



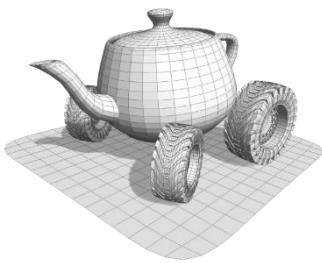


Cook-Torrance model

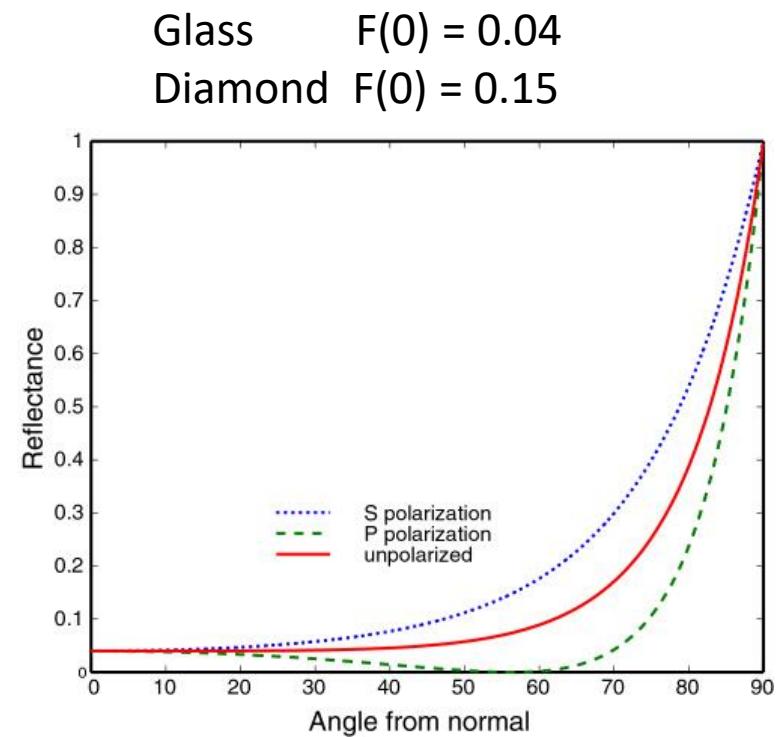
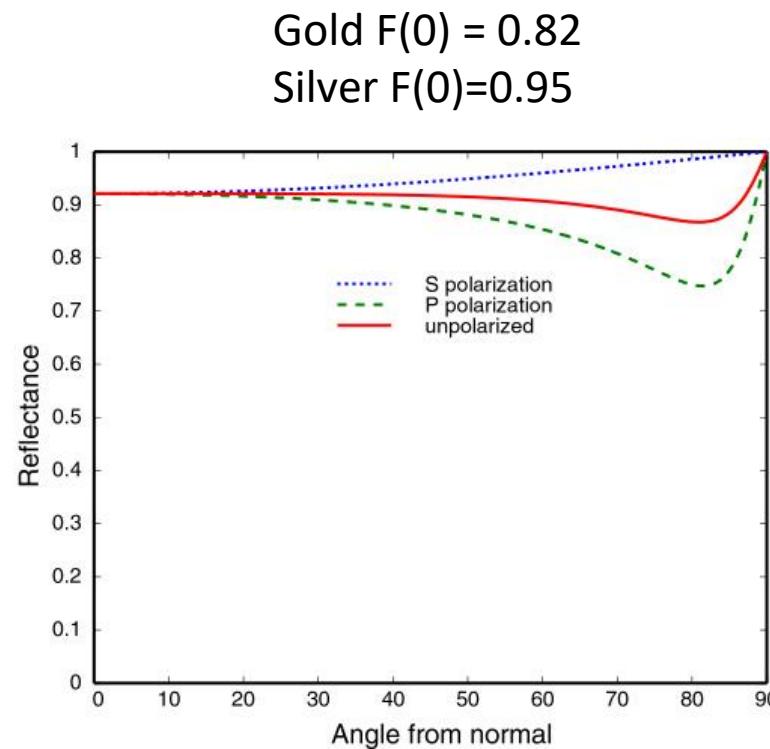
- Fresnel equations formulation are quite complex, even when neglecting the dependency on frequency
- **Schlick's approximation:** a widely used approximation of the actual Fresnel factor in specular reflection ⇒

$$F = \frac{1}{2} \frac{(g - c)^2}{(g + c)^2} \left[1 + \frac{(c(g + c) - 1)^2}{(c(g - c) + 1)^2} \right]$$
$$c = \sqrt{V \cdot H}$$
$$g = \sqrt{c^2 + \eta^2 - 1}$$

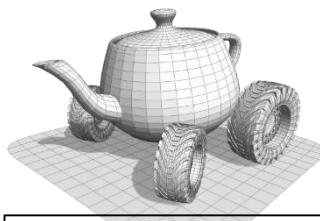
$$F = R_0 + (1 - R_0)(1 - \cos(\theta))^5$$
$$R_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2} \right)^2$$



Fresnel effect on materials



Diffusive rough materials: the Oren-Nayar model



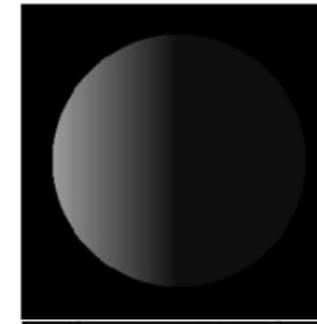
- So far, we modeled lighting diffusive (Lambertian) material as:

$$L_{diffuse} = \sum_{i=0}^n I_i k_i (N \cdot L)$$

- Does it capture well all diffusive materials?
- Does the reflected light only depends on the inclination of incident light ?

Sphere rendered with Lambertian lighting

Image of the moon

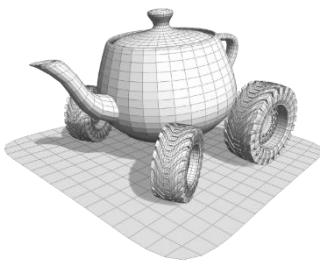


Shades varies greatly

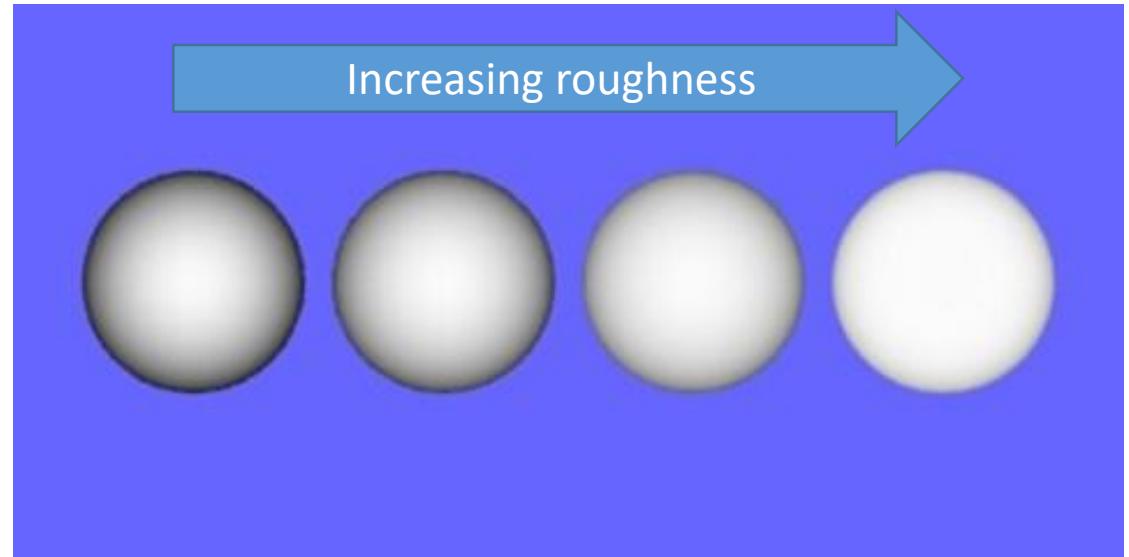
Flat appearance

Clear edges

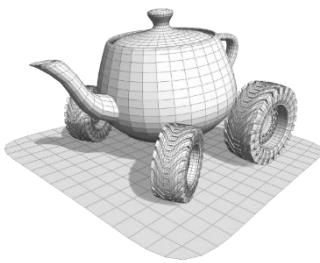
Diffusive rough materials: the Oren-Nayar model



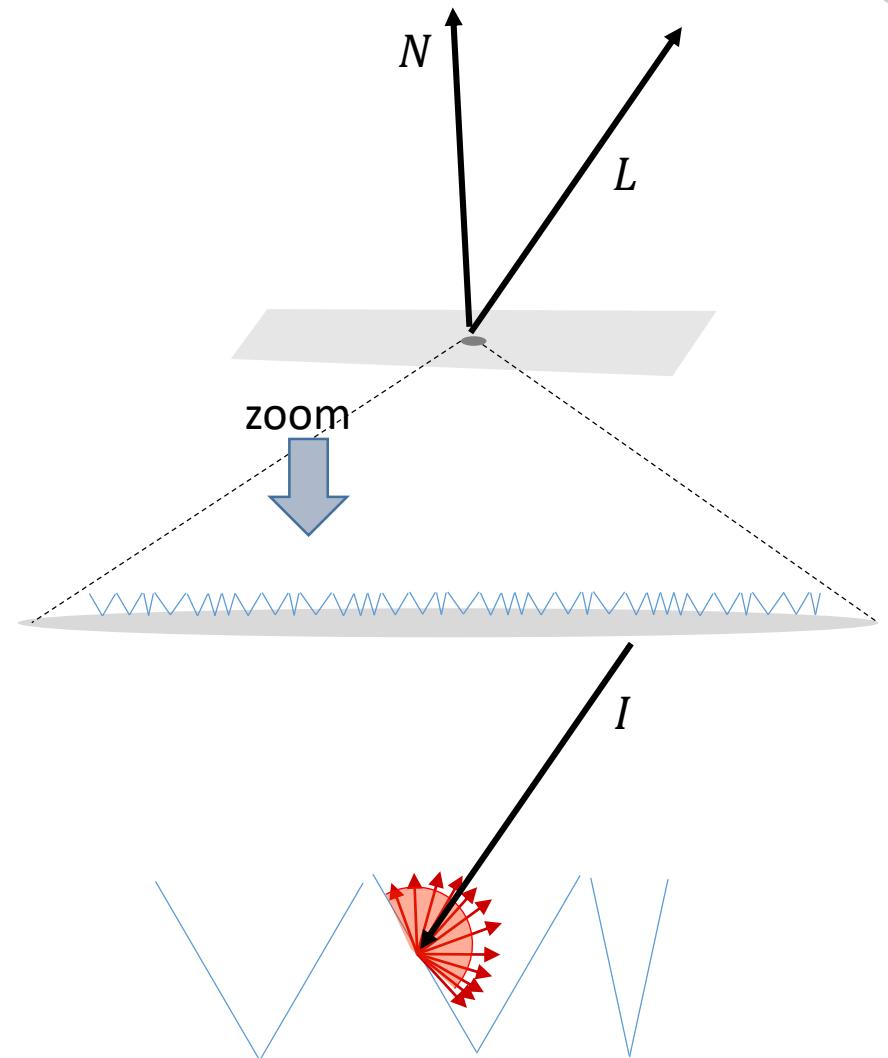
- The more a surface is «rough» the more it will appear flat
- **Retro-reflection:** a portion of light is reflected back towards the incoming light direction
- Lambertian model is good for «smooth» diffusive but bad for «rough» diffusive

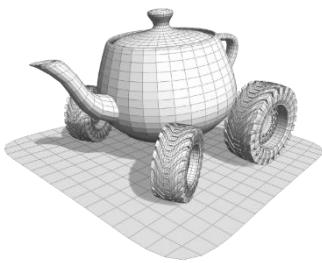


The Oren-Nayar model

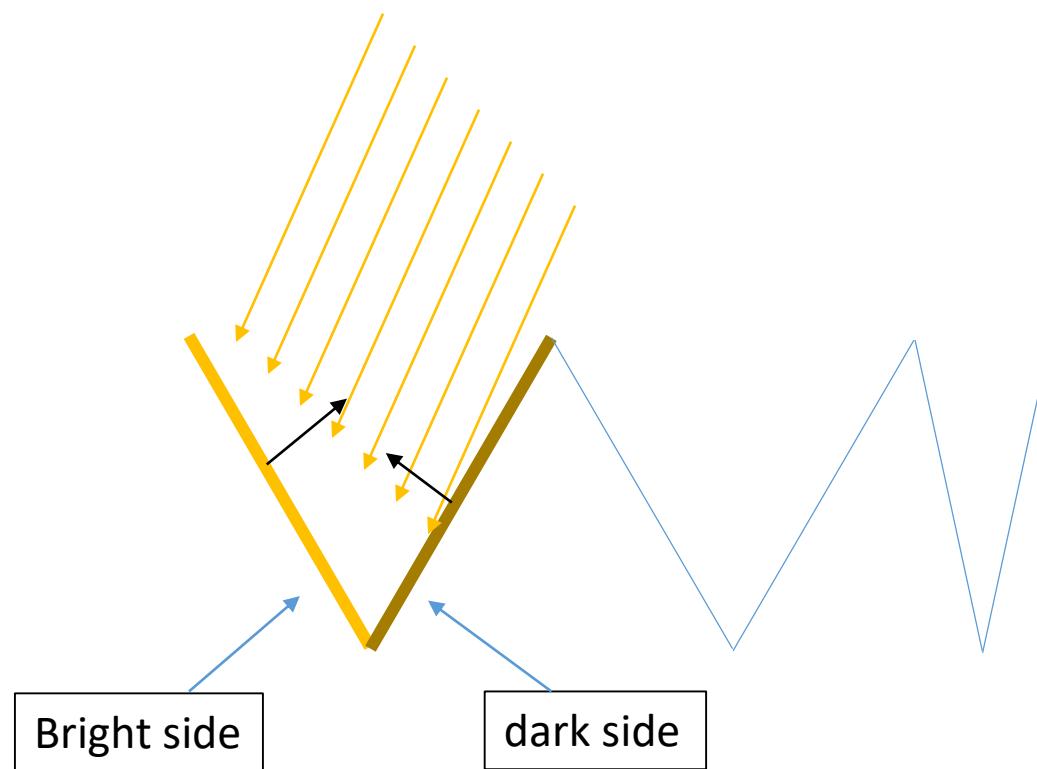


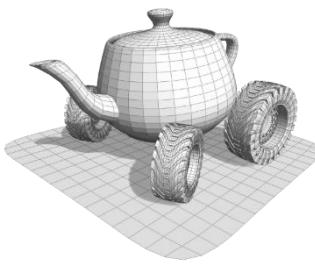
- Key idea: model the surface at a microscopic level as an assembly of many **microfacets** that act like mirrors
- The microfacets are isotropically distributed (in the neighborhood of the point lit)
- Each microfacet is a **perfect Lambertian surface**





Lambertian microfacets



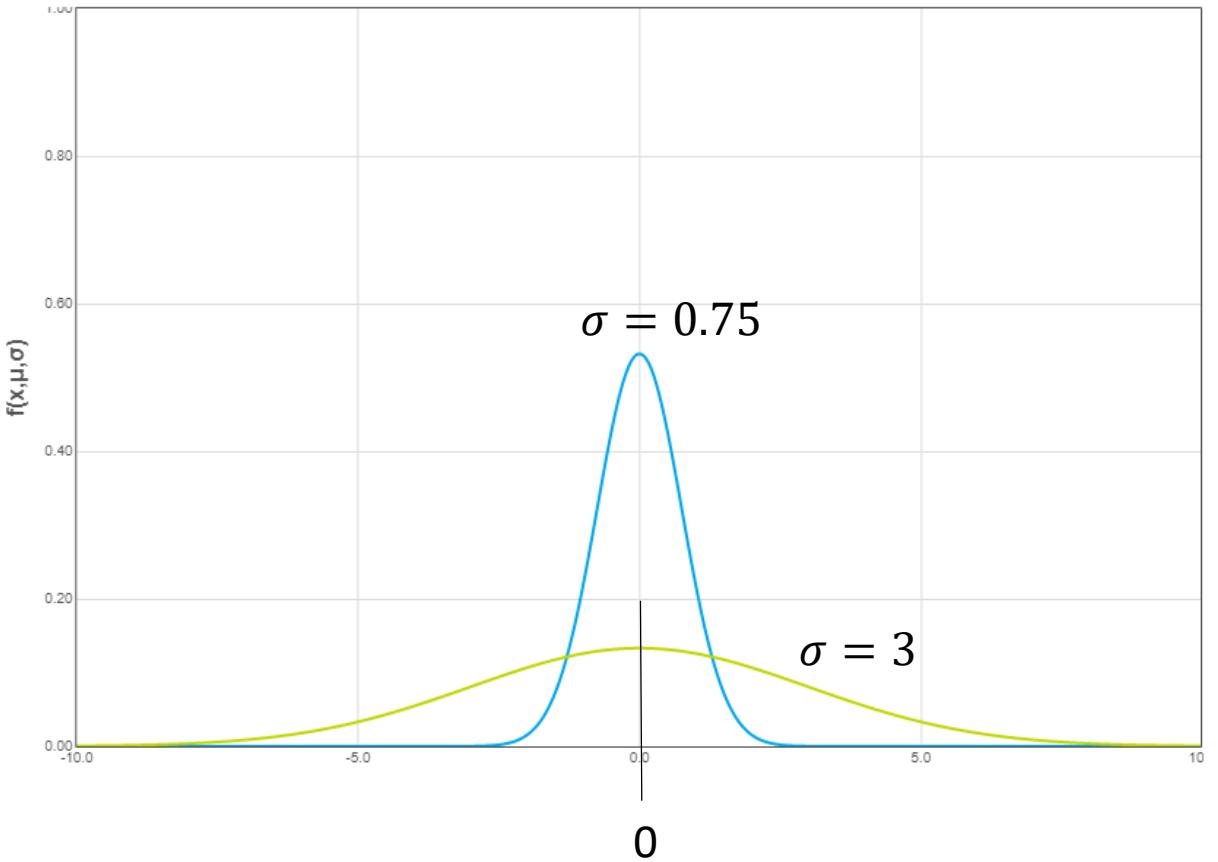


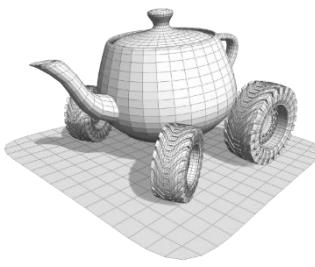
The Oren-Nayar model

- Distribution of microfacets is modeled with a 0 mean Gaussian distribution

$$\rho(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

- The higher the variance σ , the larger the spectrum of slopes, the higher the roughness





The Oren-Nayar model

$$L_r = \rho I_i (N \cdot L) (A + BC \sin \alpha \tan \beta)$$

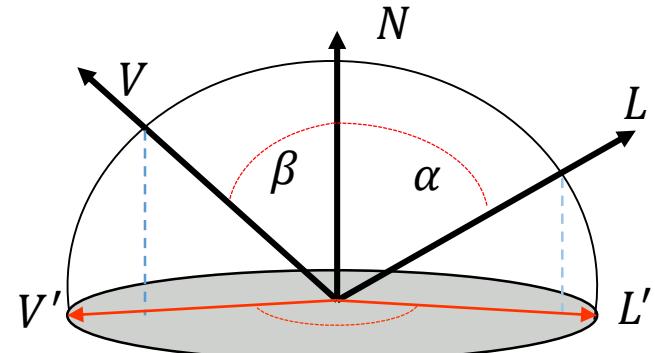
$$A = 1.0 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.009}$$

$$C = \max(0.0, L'V')$$

$$L' = L - (L \cdot N)N$$

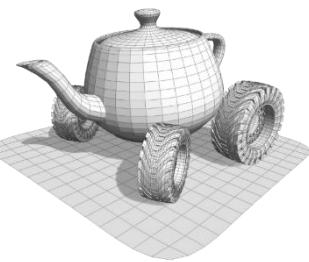
$$V' = V - (L \cdot N)N$$



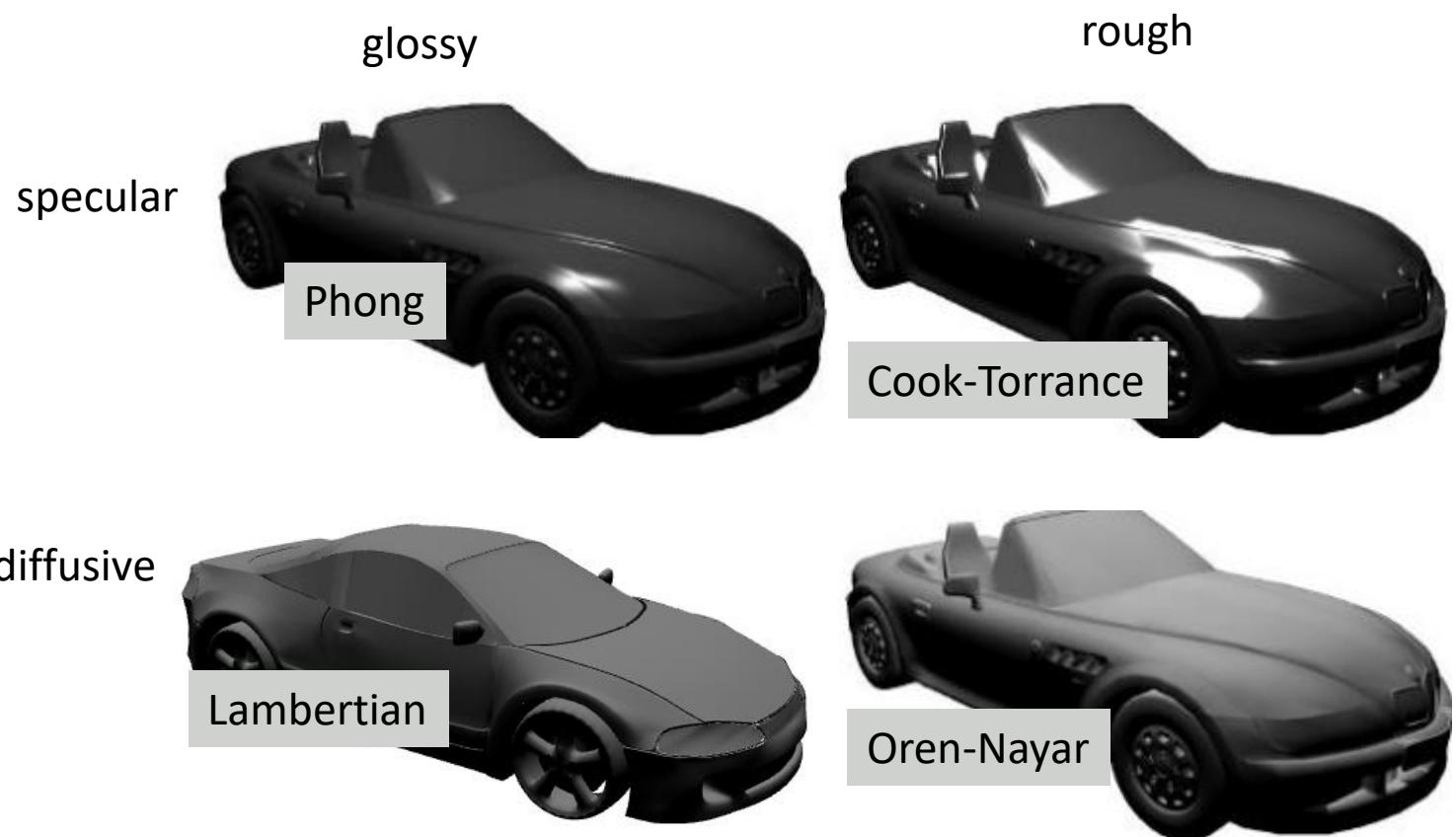
A and B depend on roughness

C depends on the **azimuth angle**
between the projections of L and V

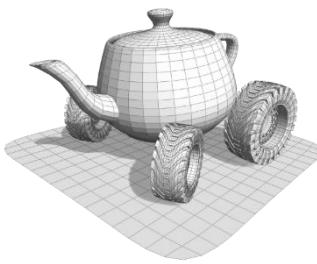
What is L_r for $\sigma = 0$?



Cook-Torrance ad Oren-Nayer: recap



Shading

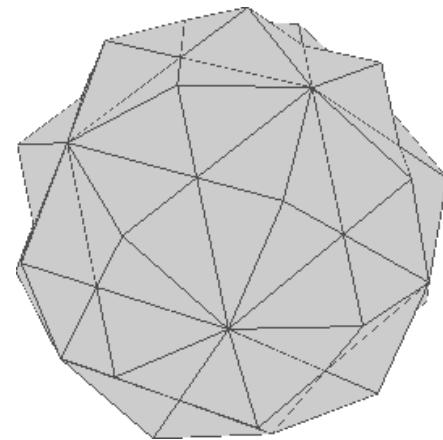
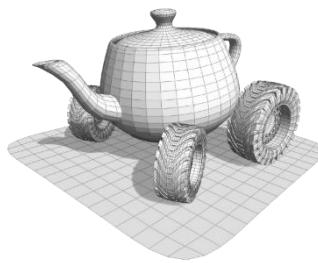


- Lighting refers to the computation of amount of light leaving the surface towards the observer direction
- A lighting model gives as a function:

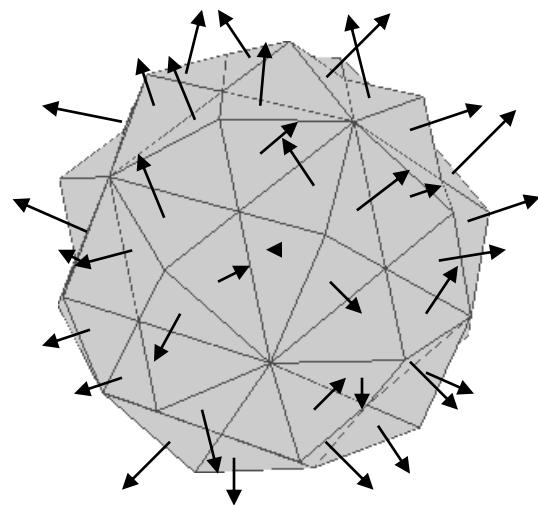
$$L(x, \text{lights}, \text{objects in the scene}) = \text{final_color}$$

- **Shading** is the process that computes L for each fragment.
 - Flat shading: lighting is computed per-face,hence the color is constant over each face
 - Gouraud shading: lighting is computed per-vertex and color is interpolated over the fragments
 - Phong shading: lighting is computed per fragment

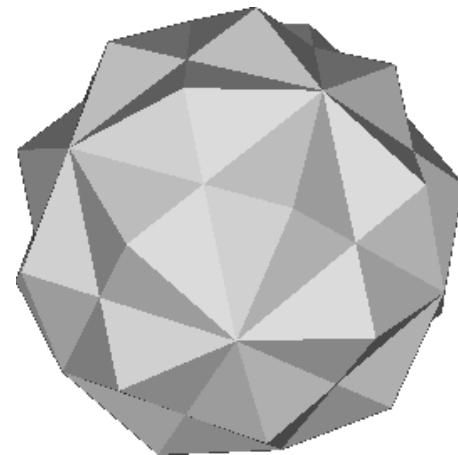
Flat Shading



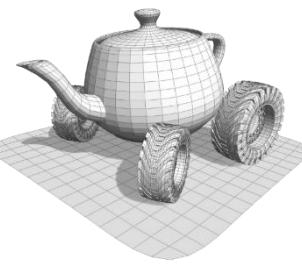
geometry



Compute the normal
for each face

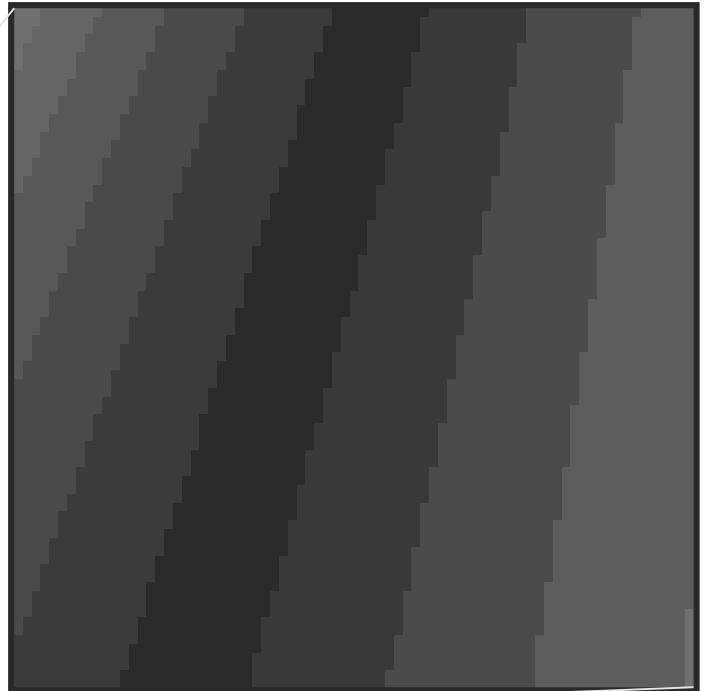
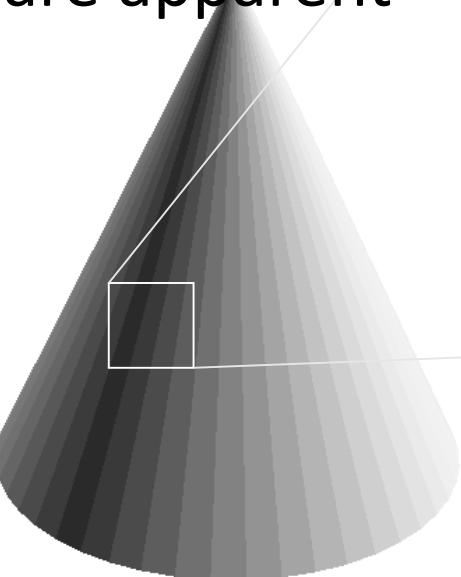
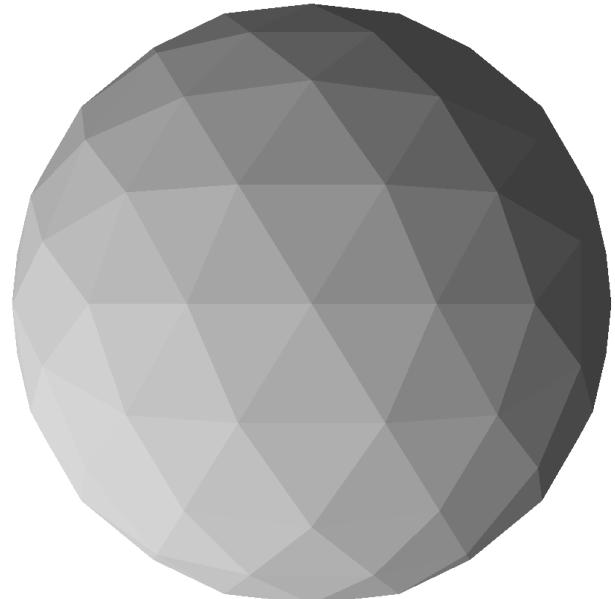


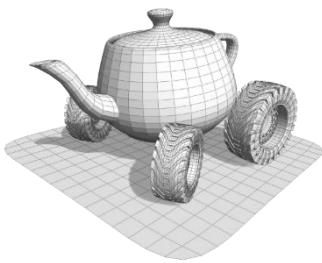
Compute the lighting equation for
each face and assign the resulting
color to all the fragments of that
face



Flat Shading: problem

- Color is constant over each face, therefore it changes discontinuously between neighbor faces
- therefore the triangle edges are apparent





Flat Shading: problem

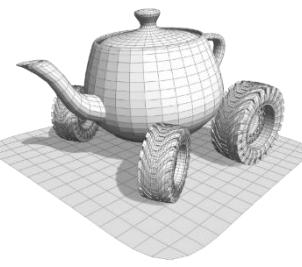
- **Mach band:** our visual system tends to enhance the contrast between slightly different luminances

6 shades of gray, one per band



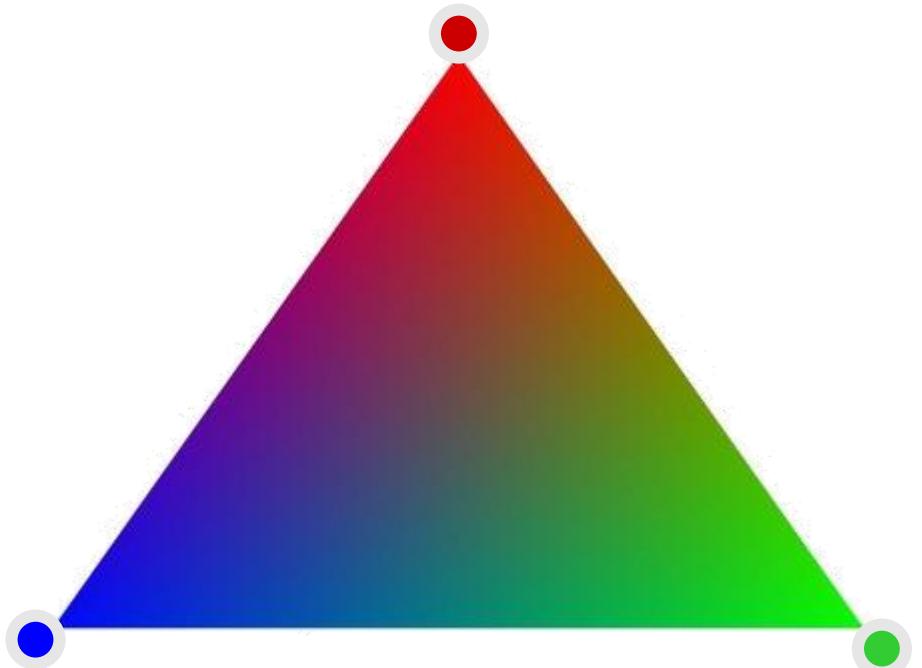
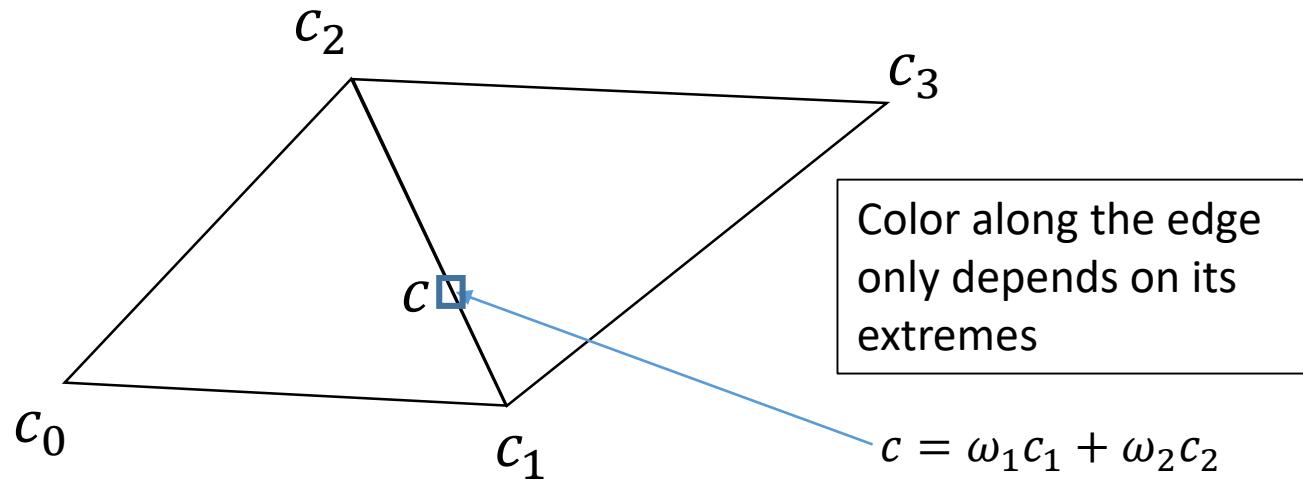
Inner squares have the same color

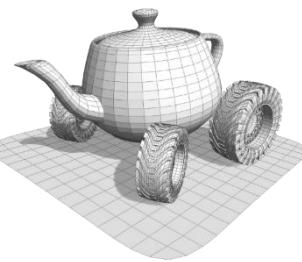




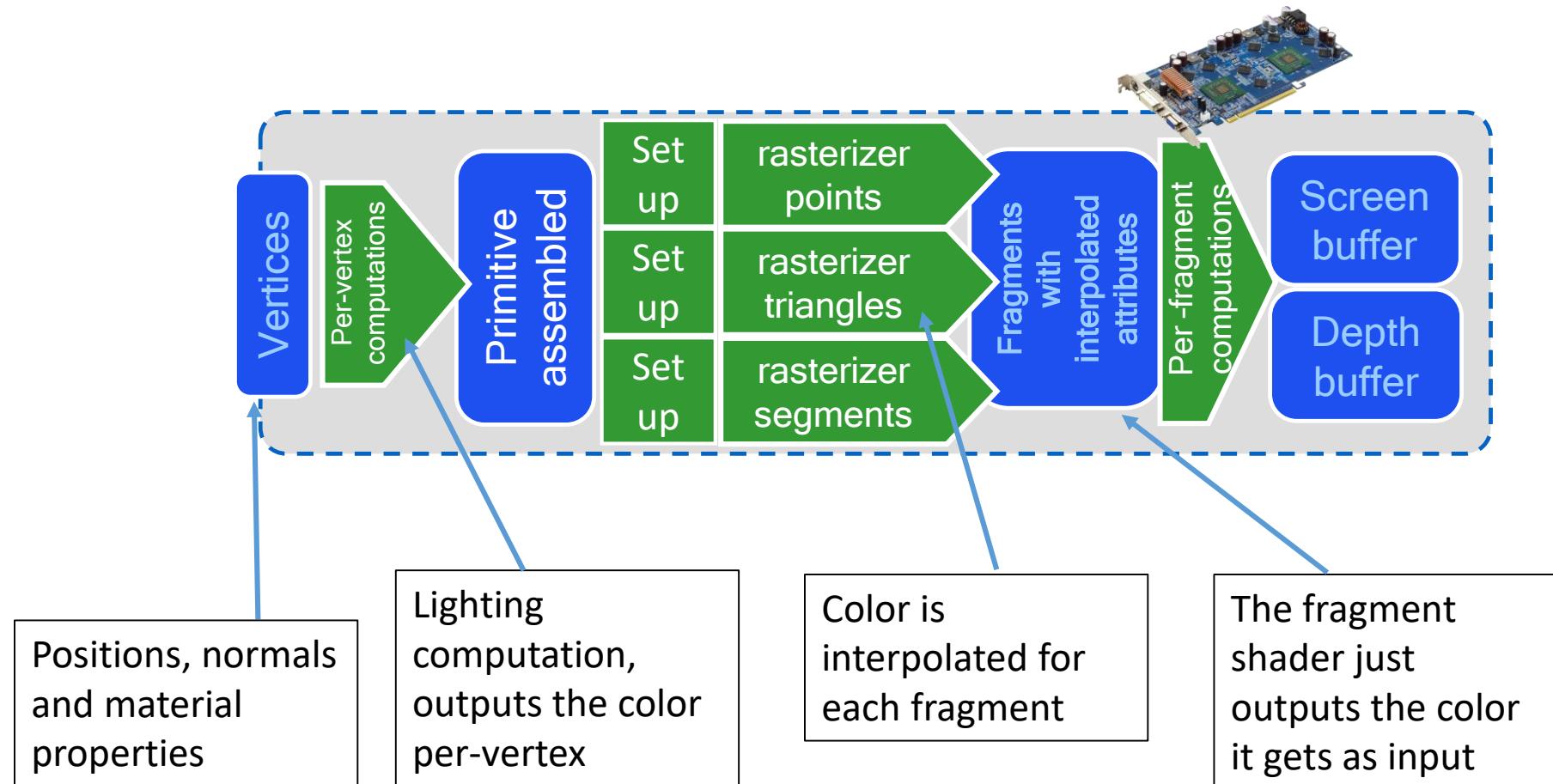
Gourad shading

- Lighting is computed per vertex, the result is interpolated inside the triangle
- Gouraud shading solves the hard discontinuity problem along the edges between triangles
 - Or does it?





Gourad Shading: where in the pipeline?



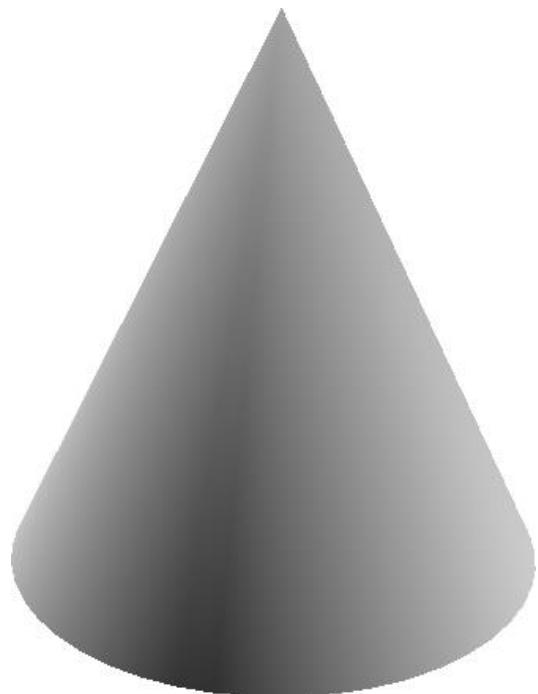
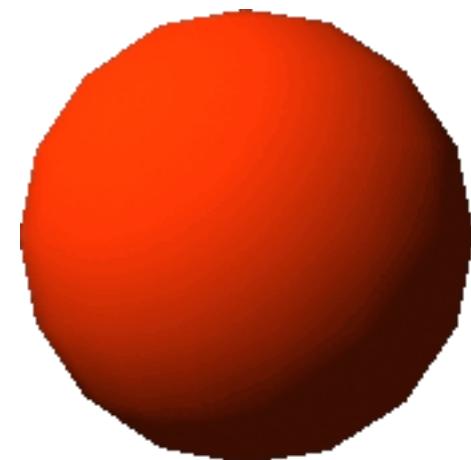
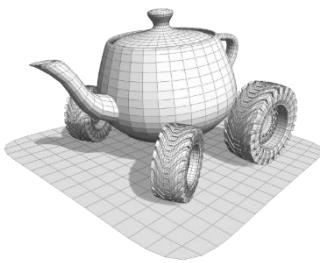
Positions, normals
and material
properties

Lighting
computation,
outputs the color
per-vertex

Color is
interpolated for
each fragment

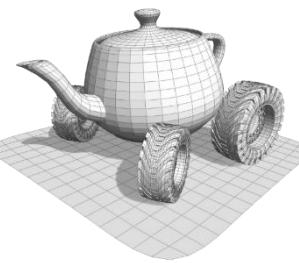
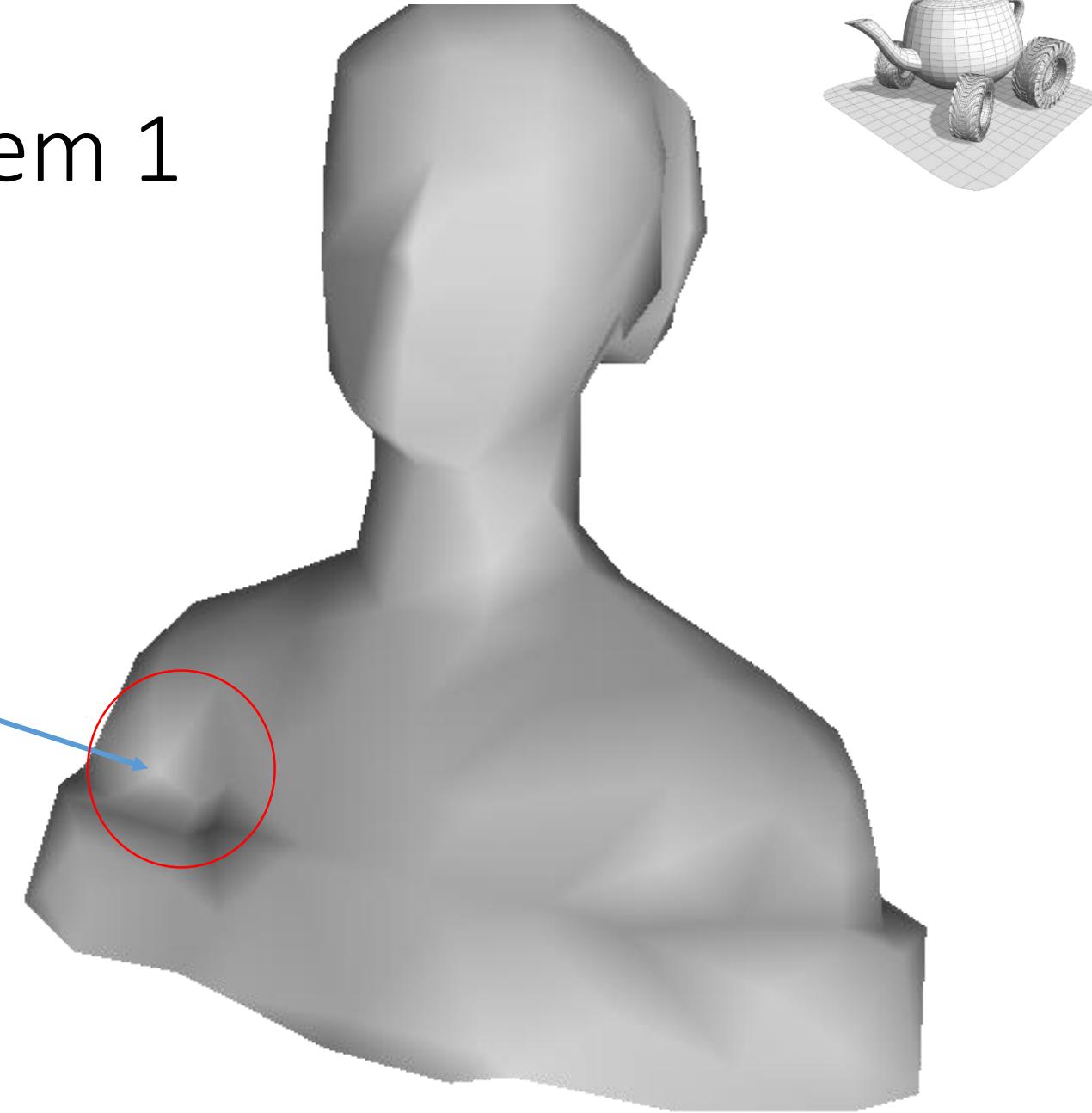
The fragment
shader just
outputs the color
it gets as input

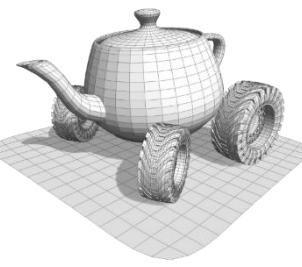
Gourad Shading: results



Gourad Shading: problem 1

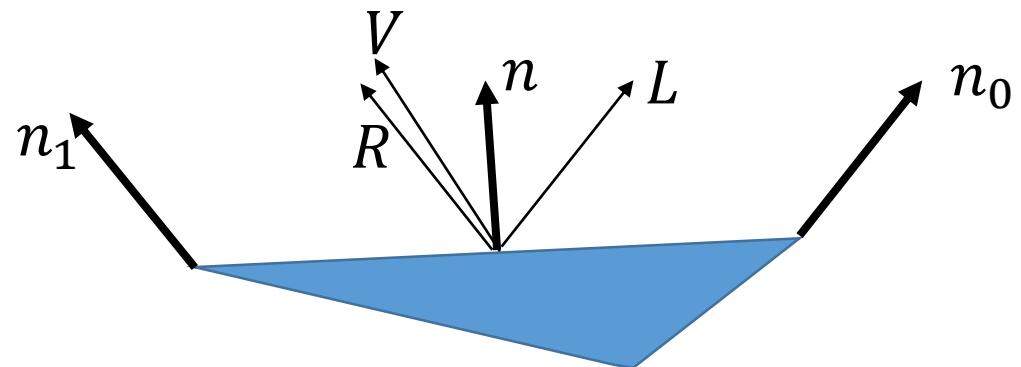
- The common barycentric coordinates on the edges do not completely solve the problem of visible tessellation

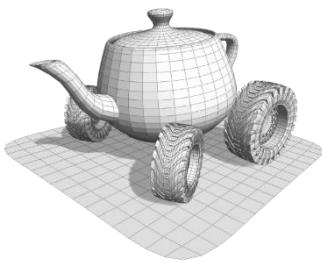




Gouraud Shading: problem 2

- Results of Gouraud shading are highly dependent on (fineness of) tessellation
- E.g. Specular highlights are likely to be missed or noticeably wrong

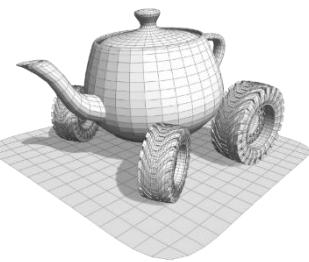




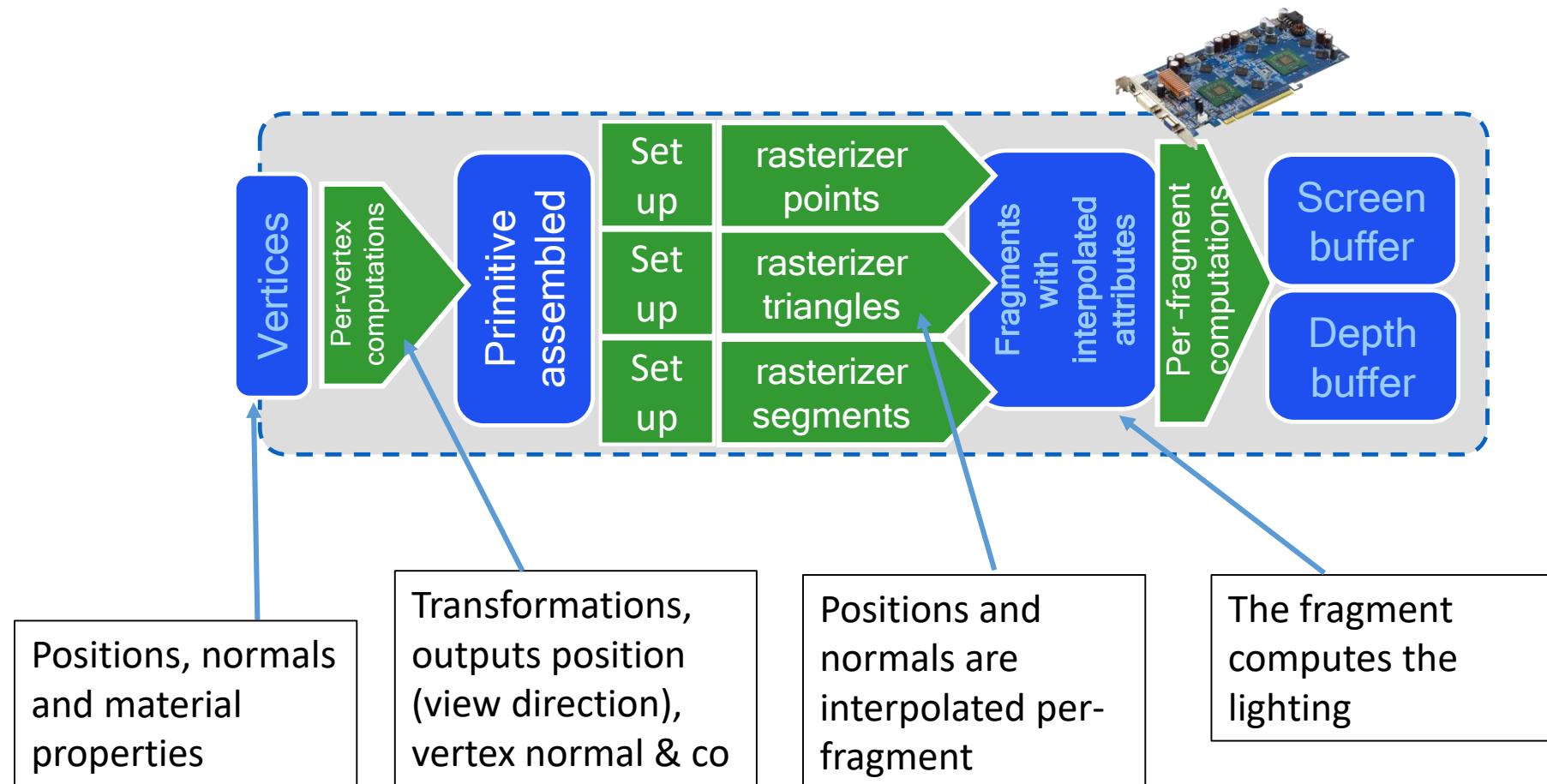
Phong shading

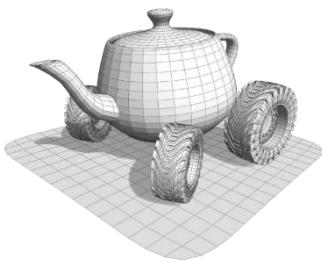
- Instead of computing lighting per vertex and then interpolate the color, interpolate the vertex normal (light if necessary) and compute the color in the fragment shader

	Vertex Shader	output	Fragment Shader
Gouraud Shading	Computes transformations and Lighting	color	Pass through
Phong Shading	Computes transformations	Transformed position, vertex normal, view direction & other attributes	Computes lighting



Phong Shading: where in the pipeline?

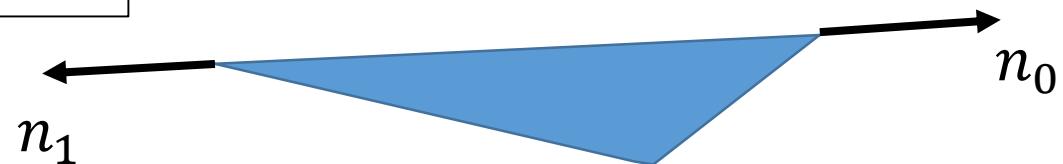




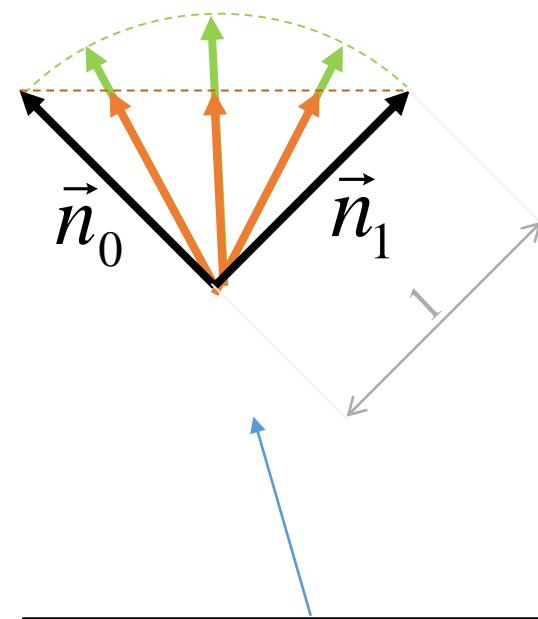
Phong Shading

- beware! Linear interpolation between vector does not preserve length
- We need to normalize the vector before using it
- Is it entirely correct? Not really...

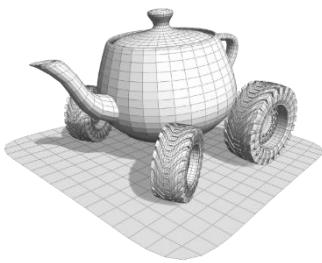
Interpolate this!



- Interpolation
- Normalized interpolation



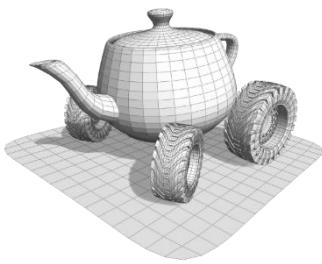
Where have we seen this before?



Comparing shading modes

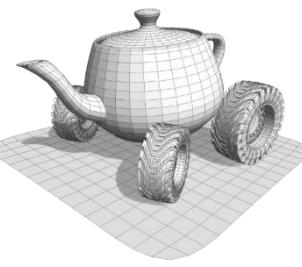
- Gouraud Shading: cheaper to compute, because vertices are normally much less than the fragments
- Phong Shading: more costly but more precise results
 - Especially with small highlights





Computing lighting

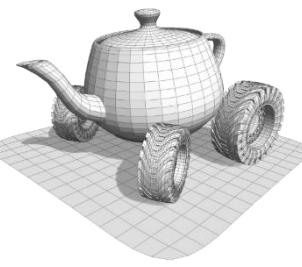
- Lighting can be computed in View Space, in World Space or even in Object space
 - The important thing is that the involved quantities are in the same coordinates system when combined
- Geometric entities involved in lighting (so far):
 - Light direction / light position
 - Surface normal
 - View direction



Examples: in View Space

Entity	transformation
Light defined in View Space, that is, that moves with the camera	ready
Light defined in World Space	View matrix to bring it in ViewSpace
Normal defined in object space	ModelView matrix to bring it in ViewSpace (inv transp)
View direction (only for orthogonal projection)	ready

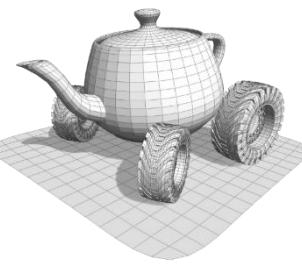
```
p_transf = Projection*View*Model*p  
ModelView = View*Model  
NormalMatrix = invTransp(ModelView)
```



Examples: in World Space

Entity	transformation
Light defined in View Space, that is, that moves with the camera	Inverse of ViewMatrix to bring it in WorldSpace
Light defined in World Space	ready
Normal defined in object space	Model matrix to bring it in WorldSpace (inv transp)
View direction (only for orthogonal projection)	Inverse of ViewMatrix to bring it in WorldSpace

```
p_transf = Projection*View*Model*p  
ModelView = View*Model  
NormalMatrix = invTransp(ModelView)
```

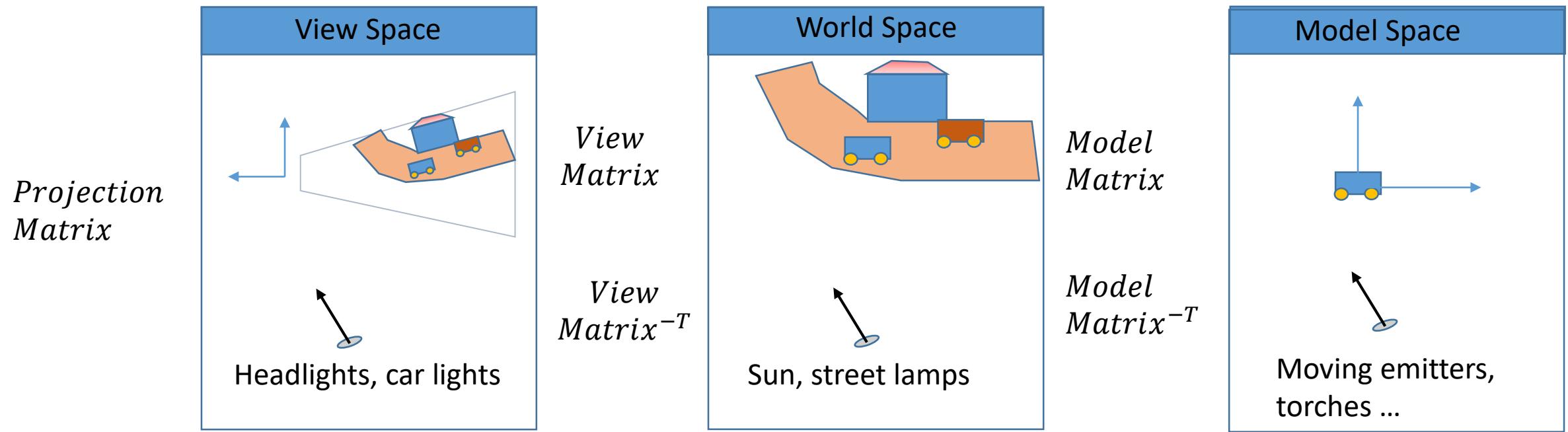
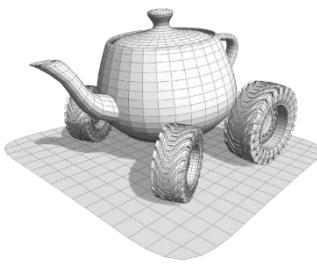


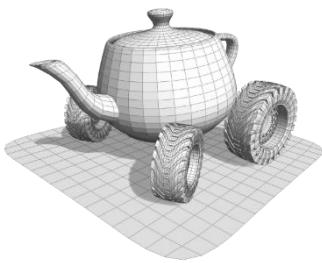
Examples: in Object Space

Entity	transformation
Light defined in View Space, that is, that moves with the camera	Inverse of ModelView
Light defined in World Space	Inverse of Model matrix to bring it in Object Space
Normal defined in object space	ready
View direction (only for orthogonal projection)	Inverse of ModelView

```
p_transf = Projection*View*Model*p  
ModelView = View*Model  
NormalMatrix = invTransp(ModelView)
```

Transformations: recap

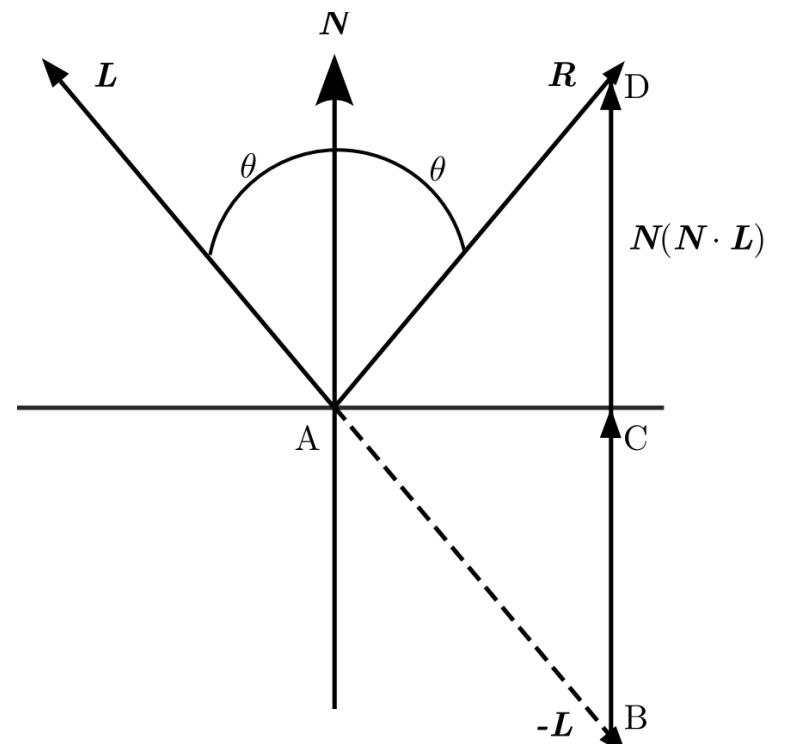


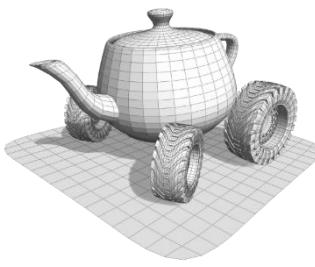


Specular Direction

- When computing lighting, the specular direction with respect to a vector may be needed
- Typically the specular direction of light with respect to the normal
 - Not in Blinn-Phong lighting
 - Luckily is a simple expression among vectors

$$R = -L + 2(N \cdot L)N$$



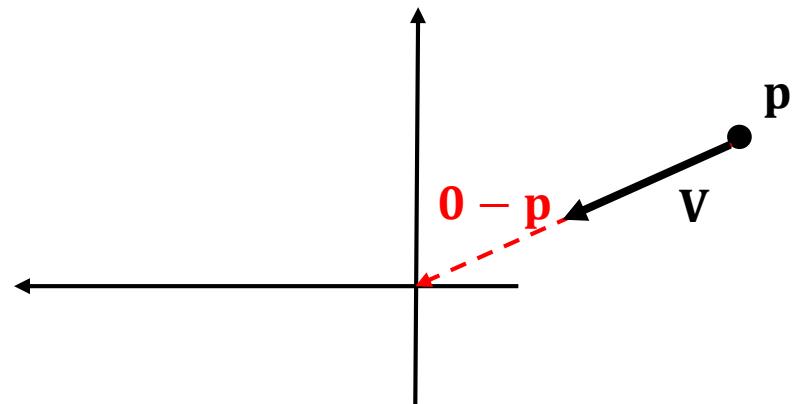


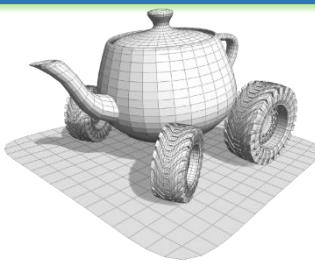
View Direction

- The view direction in View Space is simply the position transformed by the ModelView matrix and negated

This may be a sequence of matrices

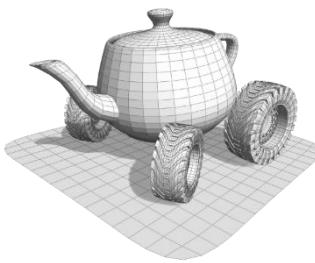
```
vec4 posVS = uViewMatrix * uModelMatrix * vec4(aPosition, 1.0) ;  
vec3 viewDir = normalize(posVS);
```





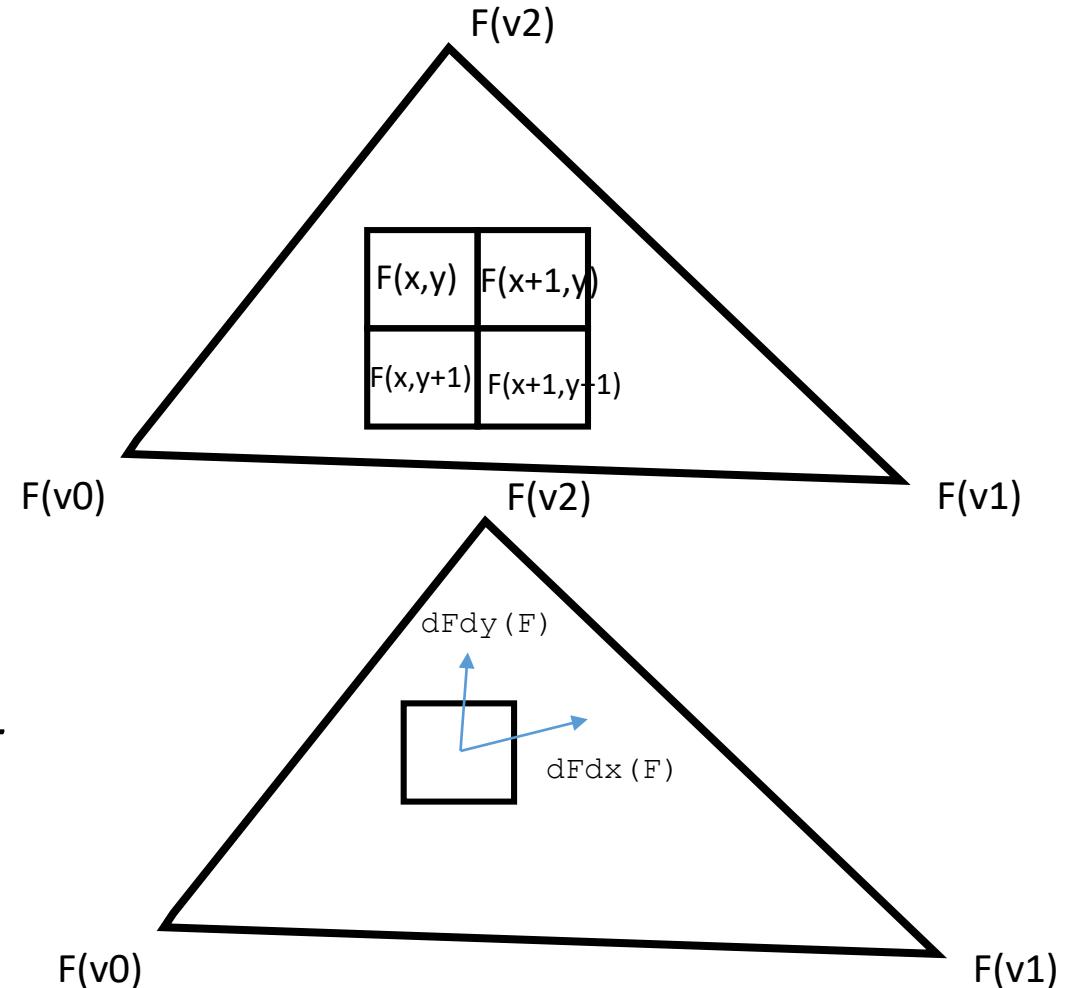
Implementing Flat Shading

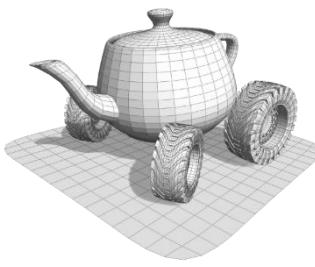
- Gourad shading requires to access the normal per-face. How to do it?
 1. `glShadeModel(GL_FLAT)` in the fixed pipeline....no more
 2. Interpolation qualifiers:
`out flat vec3 normal;` specifies that «normal» must not be interpolated but just copied
 3. Pass the same normal for the three vertices of the triangle
 - This implies one copy of the same vertex for each triangle using it, it works but it can be costly
 4. Compute the normal in the fragment shader.. next



Derivates in the fragment shader

- The fragment shader can provide the derivatives of interpolated values:
 $dFdx(F)$ and $dFdy(F)$
- If F the «position» of vertex, its value on the fragment will be the 3D position of the surface that produces that fragment during rasterization
- Then $dFdx$ and $dFdy$ give two *tangent vectors* of the triangle..

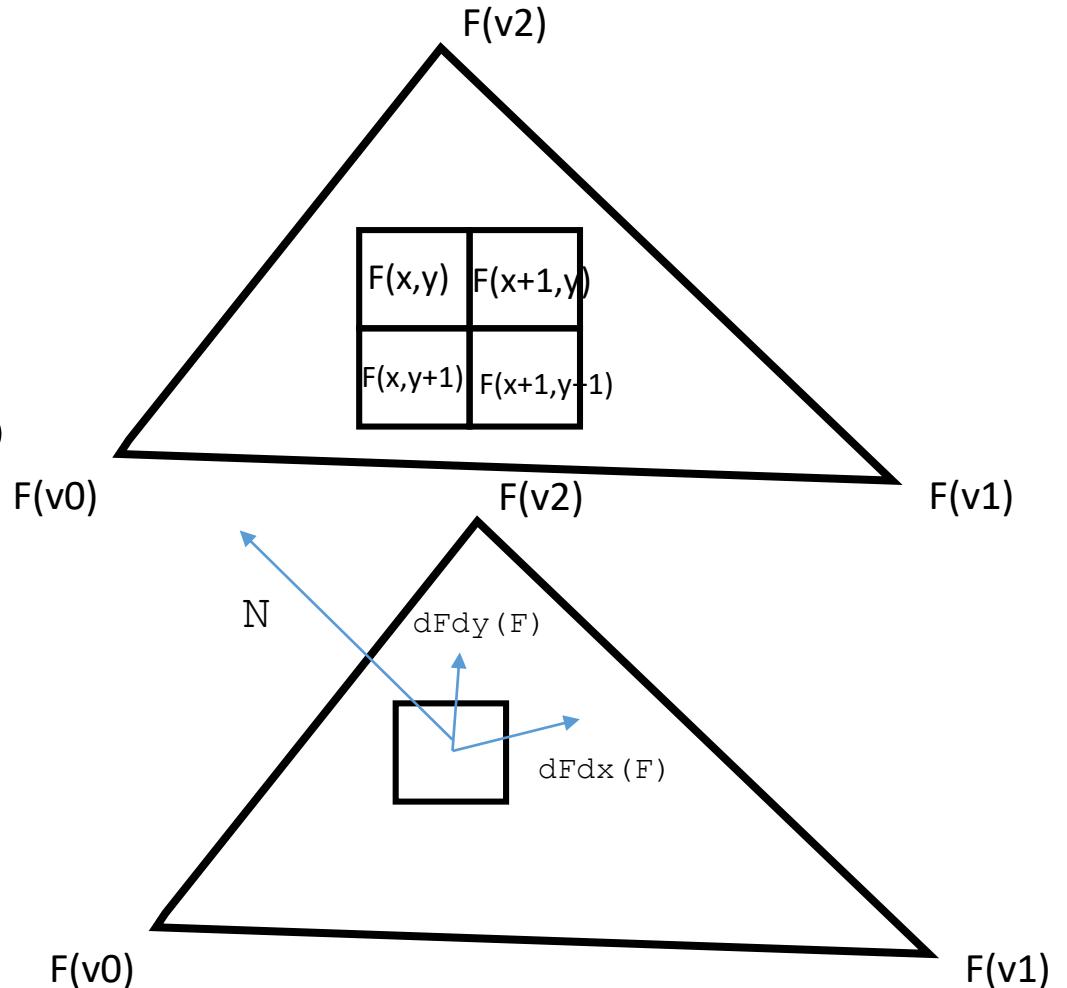




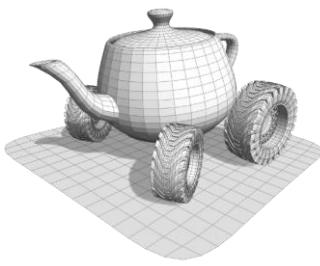
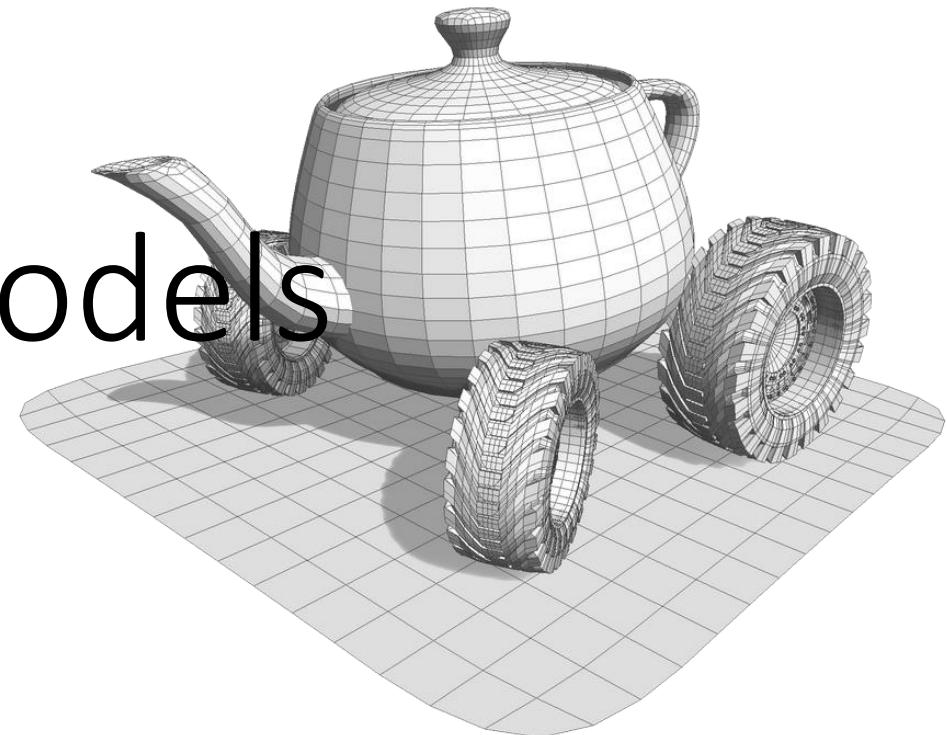
Normal in the fragment shader

- Since $dFdx(pos)$ and $dFdy(pos)$ are tangent to the surface, their cross product is orthogonal to it
- So we can compute the normal as:

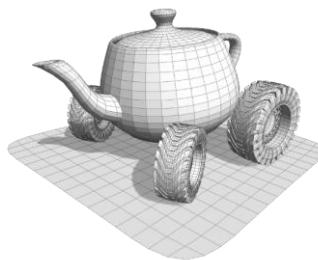
```
N = normalize(cross(dFdx(pos), dFdy(pos)))
```



Loading 3D Models



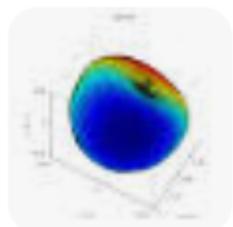
Lots of file formats for 3D scenes available



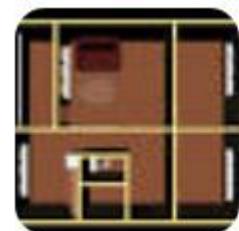
Wavefront .obj file



gITF



PLY



VRML



X3D



OFF



STL



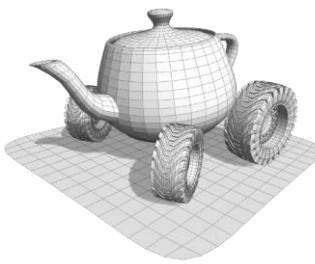
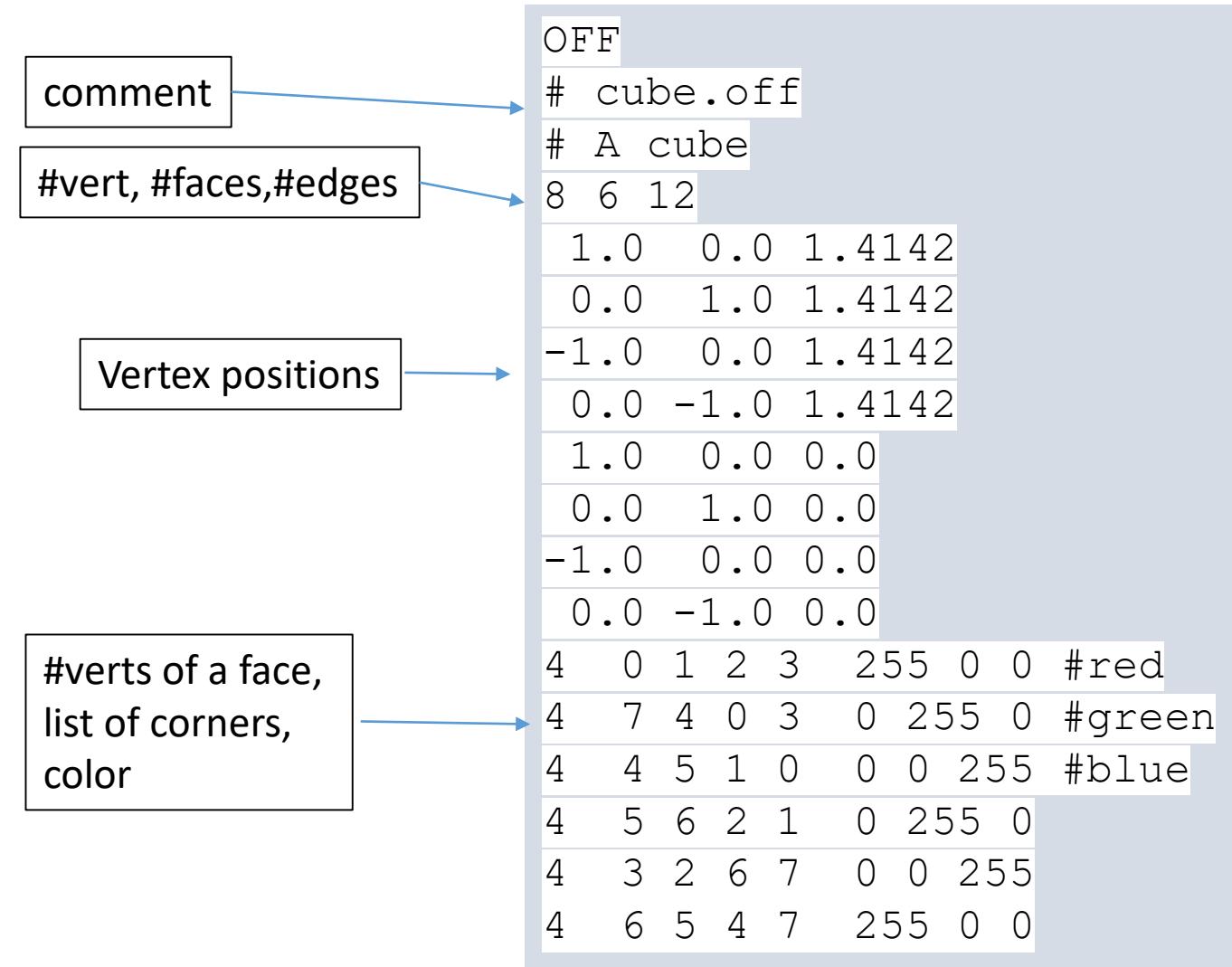
COLLADA

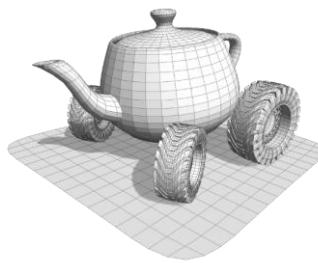


.3ds

.OFF

- Very simple ASCII format
- Only supports vertex positions, faces and color per face
- Created for geometry processing, not meant for rendering





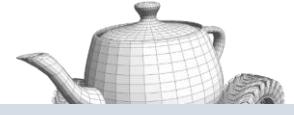
.STL (3DSystems)

- **STereoLitography**
- Only supports vertex positions and triangle faces
- No metadata
- Developed for printing, not for rendering

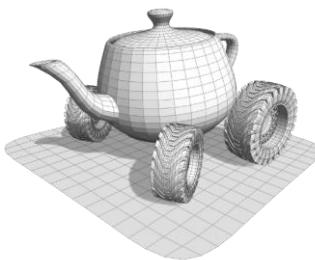
```
solid STL generated by MeshLab
facet normal 3.568221e-01 9.341723e-01 -0.000000e+00
  outer loop
    vertex 0.000000e+00 1.618034e+00 -1.000000e+00
    vertex 0.000000e+00 1.618034e+00 1.000000e+00
    vertex 1.618034e+00 1.000000e+00 0.000000e+00
  endloop
endfacet
facet normal -3.568221e-01 9.341723e-01 0.000000e+00
  outer loop
    vertex 0.000000e+00 1.618034e+00 1.000000e+00
    vertex 0.000000e+00 1.618034e+00 -1.000000e+00
    vertex -1.618034e+00 1.000000e+00 0.000000e+00
  endloop
endfacet
....
```

.PLY

- **PoLYgon file format (also Stanford File Format)**
- Supports per vertex attributes («properties»): color, normal, texture coordinates
- Supports user-defined properties
 - You could define the material for Phong lighting..
- Designed for scanned 3D data & visualization



```
ply
format ascii 1.0
comment VCGLIB generated
element vertex 12
property float x
property float y
property float z
property uchar red
property uchar green
property uchar blue
property uchar alpha
element face 20
property list uchar int vertex_indices
end_header
0 1.618034 1 255 1 0 255
.. other vertices ..
-1 0 -1.618034 255 0 255 255
3 1 0 4
3 0 1 6
3 2 3 5
... other faces ....
```



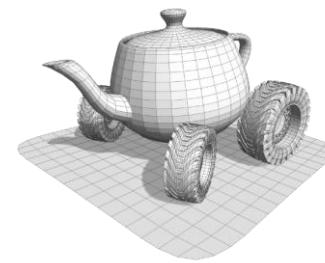
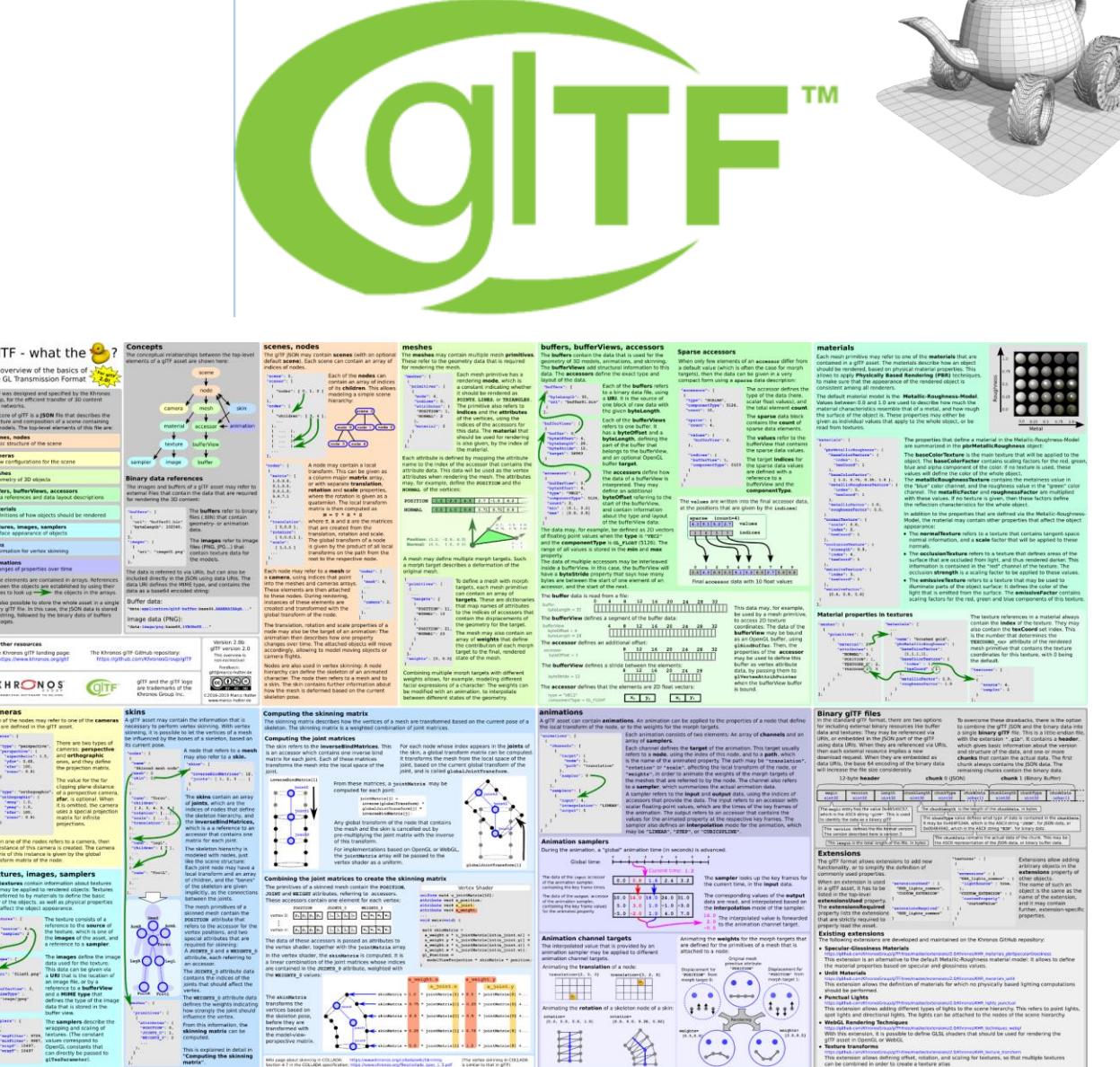
.VRML and .X3D

- Virtual Reality Modeling Language
- Developed for showing 3D objects in a Web Browser (in 1994)
- Supports basic rendering attribute and animation
- Include code nodes in Java
- Superseded by .X3D

```
#VRML V2.0 utf8  
Shape {  
    geometry IndexedFaceSet {  
        coordIndex [ 0, 1, 2 ]  
        coord Coordinate {  
            point [ 0, 0, 0, 1, 0, 0, 0.5, 1, 0 ]  
        }  
    }  
}
```

.gltf .glb

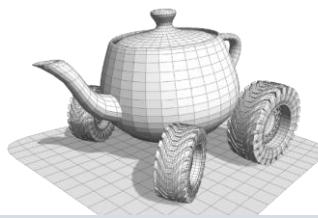
- *gl Transmission Format
- Developed for efficient streaming/loading/rendering of 3D scenes
 - The «jpeg of 3D objects»
- Describe a scene
 - Hierarchies
 - Animation
 - Physically Based Rendering



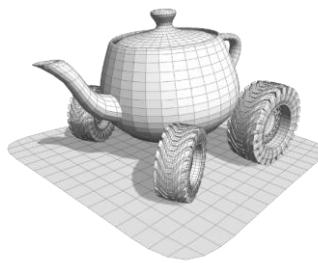
.obj

- Developed for modeling and rendering
- Define geometric objects alone (no scene hierarchy, no animation or deformation)
- Include **materials** for subparts of an object
- Include normal, color and texture coordinates for vertices

```
# List of geometric vertices, with (x, y, z, [w]) coordinates, w  
is optional and defaults to 1.0.  
v 0.123 0.234 0.345 1.0  
v ...  
...  
# List of texture coordinates, in (u, [v, w]) coordinates, these  
will vary between 0 and 1. v, w are optional and default to 0.  
vt 0.500 1 [0]  
vt ...  
...  
# List of vertex normals in (x,y,z) form; normals might not be  
unit vectors.  
vn 0.707 0.000 0.707  
vn ...  
...  
# Polygonal face element (see below)  
f 1 2 3  
f 3/1 4/2 5/3  
f 6/4/1 3/5/3 7/6/5  
f 7//1 8//2 9//3  
f ...  
...  
# Line element (see below)  
l 5 8 1 2 4 9
```



.obj (.mtl)



A face specifies, for each vertex, an index to its position, one to its tex coords, and one to its normal

positions



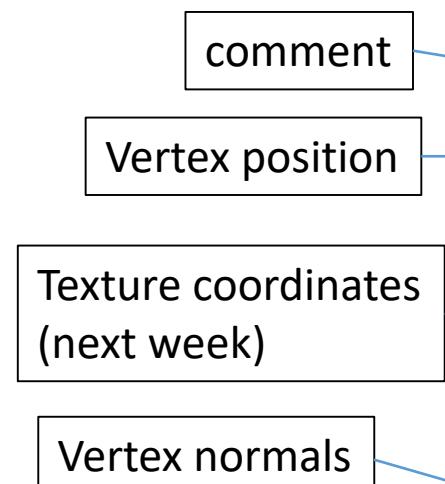
Tex coords

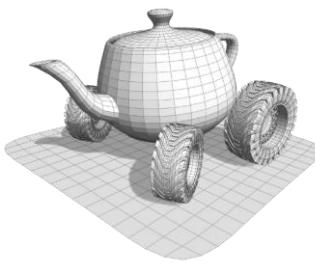


normals



f 6/4/1 3/5/3 7/6/5





.obj (.mtl)

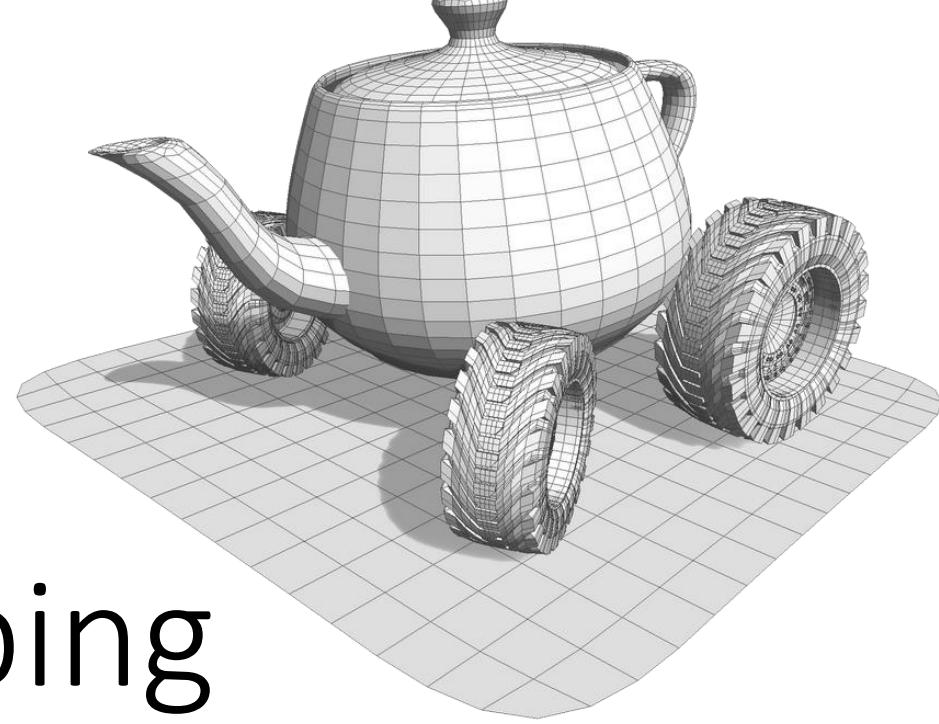
- Materials are specified in separated .mtl files
- newmtl 280z_Blackout specifies a material with name 280z_Blackout
- In the obj file, the keyword usemtl tells what material to use for the following element (a group, an object or even a face)
- .mtl also contains the name of the images to load as textures

Datsun_280Z.obj

```
# Blender v2.93.8 OBJ File:  
'Datsun_280Z.blend'  
# www.blender.org  
mtllib Datsun_280Z.mtl  
...
```

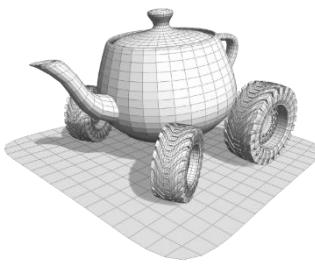
Datsun_280Z.mtl

```
# Blender MTL File: 'Datsun_280Z.blend'  
# Material Count: 15  
  
newmtl 280z_Blackout  
Ns 0.000000  
Ka 1.000000 1.000000 1.000000  
Kd 0.000000 0.000000 0.000000  
Ks 0.000000 0.000000 0.000000  
Ke 0.000000 0.000000 0.000000  
Ni 1.450000  
d 1.000000
```



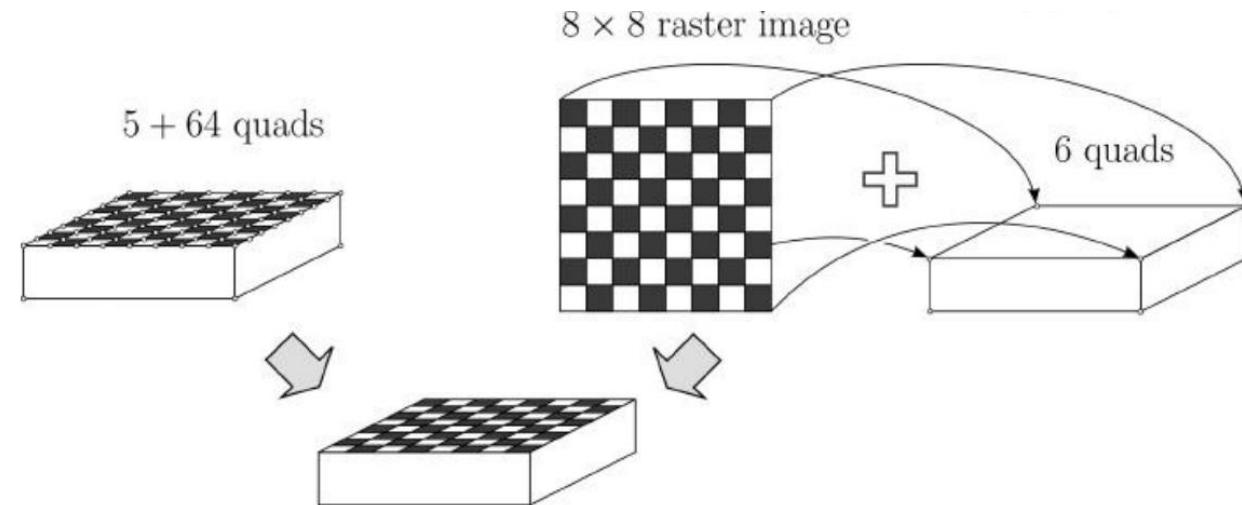
Texture Mapping



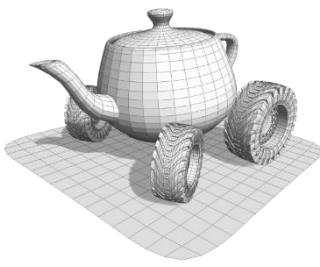


What is texturing?

- Roughly speaking **texture mapping** is stitching (=mapping) a raster image (texture) onto geometry (mostly polygons)
- Why? Compared to vertex attributes, raster data can convey much more information at a fraction of the cost

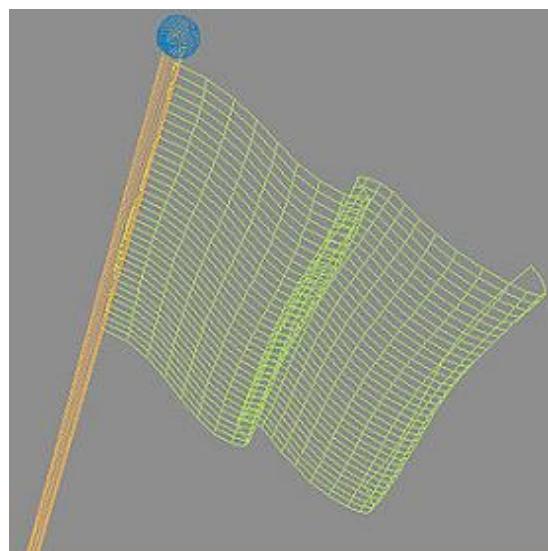
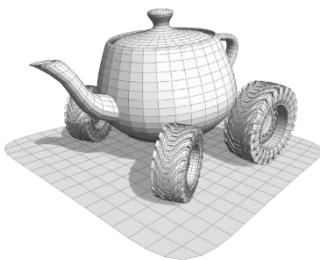


Texture mapping



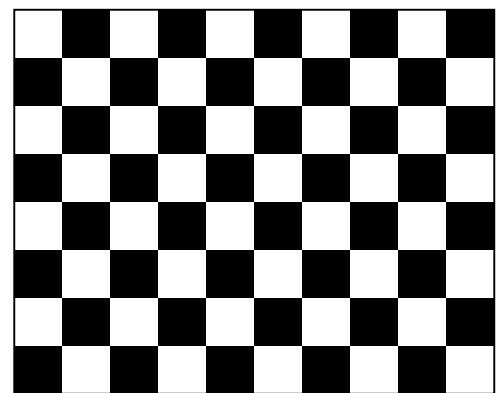
- A **texture** is an array of data, which can be 1, 2 or 3D dimensional
- The most common type of texture is just a simple raster image but a texture can encode any sort of information other than color
 - Normal, curvature and other geometric measures
 - Backed lighting, e.g. result of diffuse lighting
- A **texel (texture element)** is one element of the array (just like a pixel for a 2D image)

Examples



Polygon mesh

+



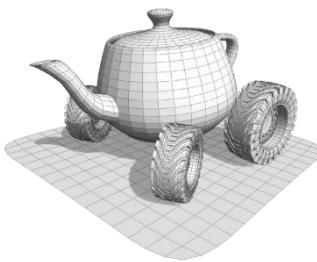
2D RGB texture
(color-map)

=



Textured mesh

examples



Polygon mesh

+



2D RGB texture
(color-map)

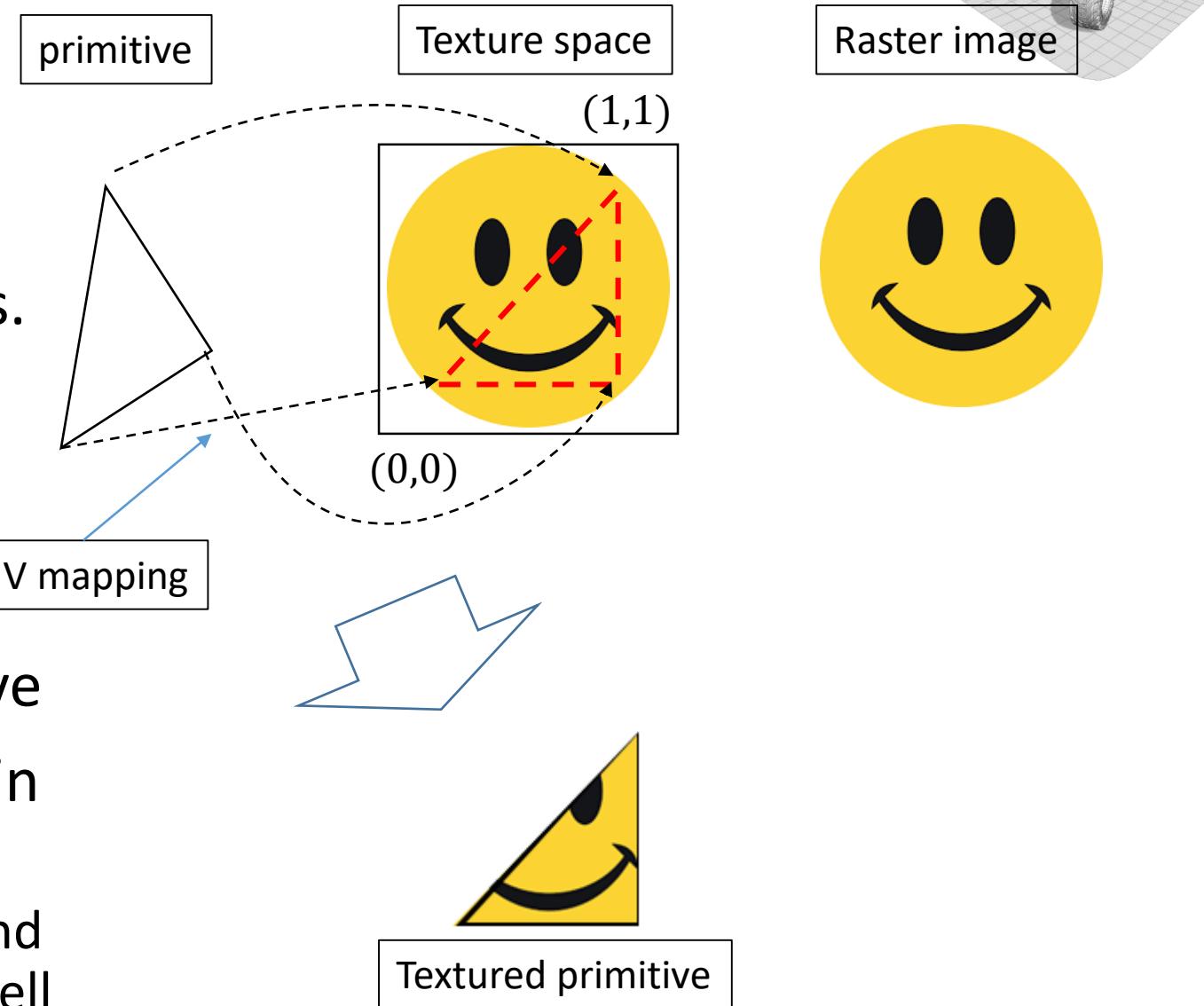
=

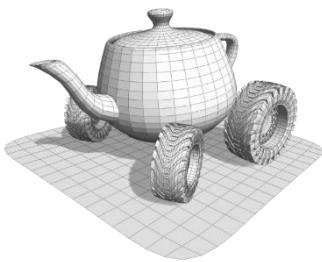


Textured mesh

Texture mapping

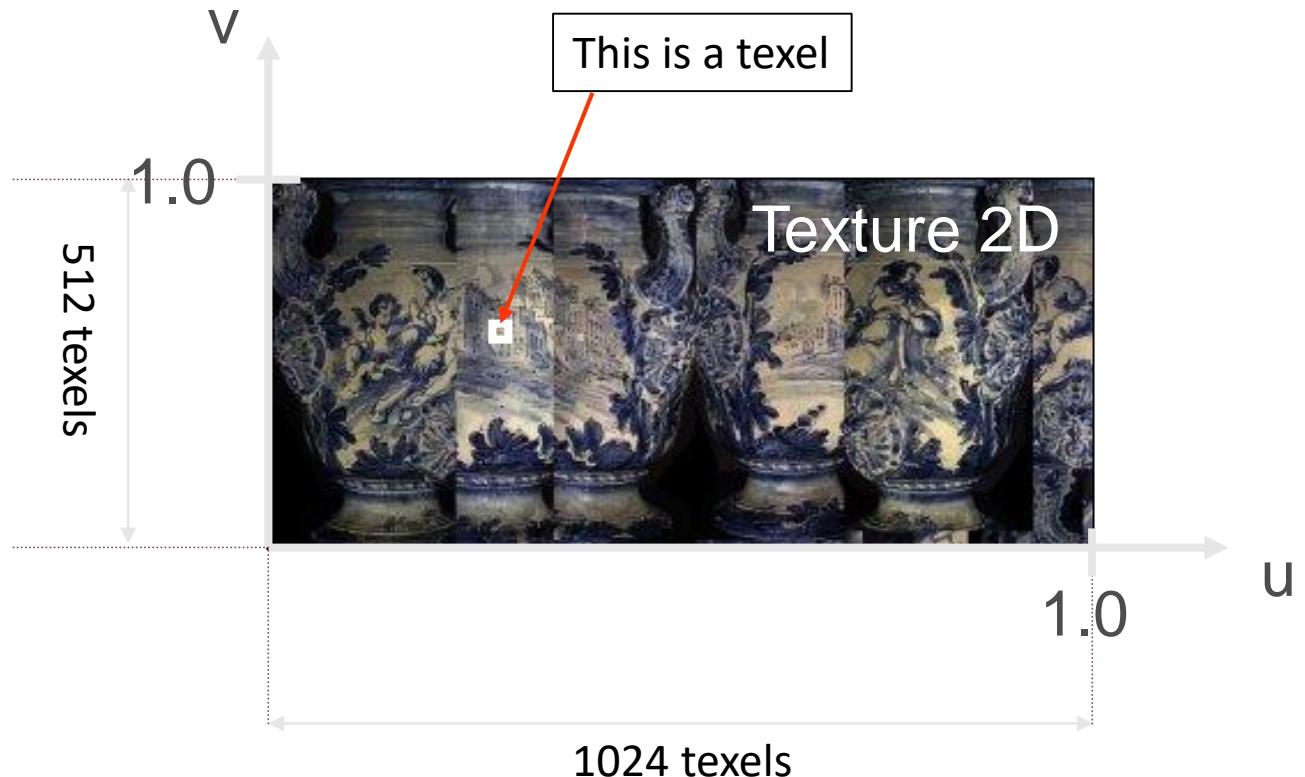
- We need to map points on geometry to points on textures. This mapping is called **UV-mapping**
- UV coordinates are typically assigned to vertices and interpolated inside the primitive
- UV coordinates are expressed in **texture space**
 - It's typically 2D between [0,0] and [1,1] but it can be 1D or 3D as well

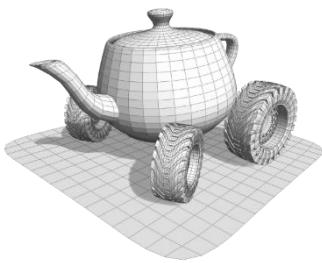




Texture Space

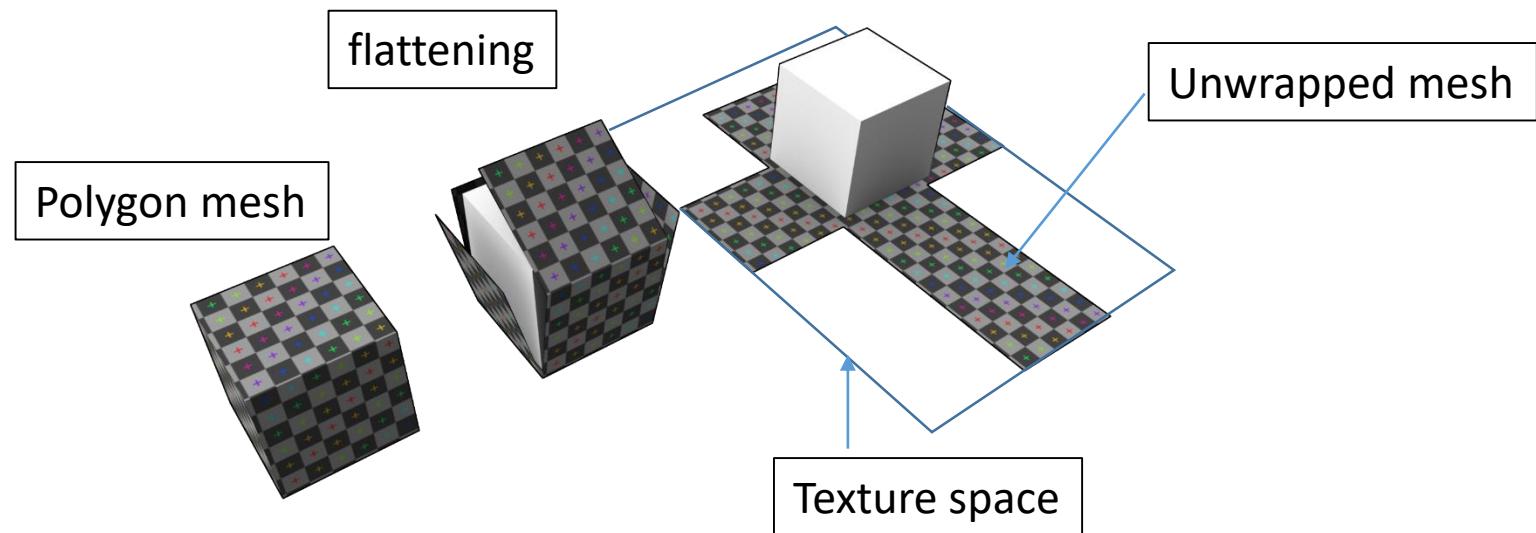
- Or **uv-space**, or **parametric space**
- A texture is defined in the region $[0,0]-[1,1]$ of the parametric space
- The texture does not have to be square, but its sides must be power of two



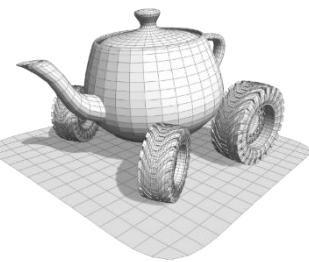


UV-unwrapping

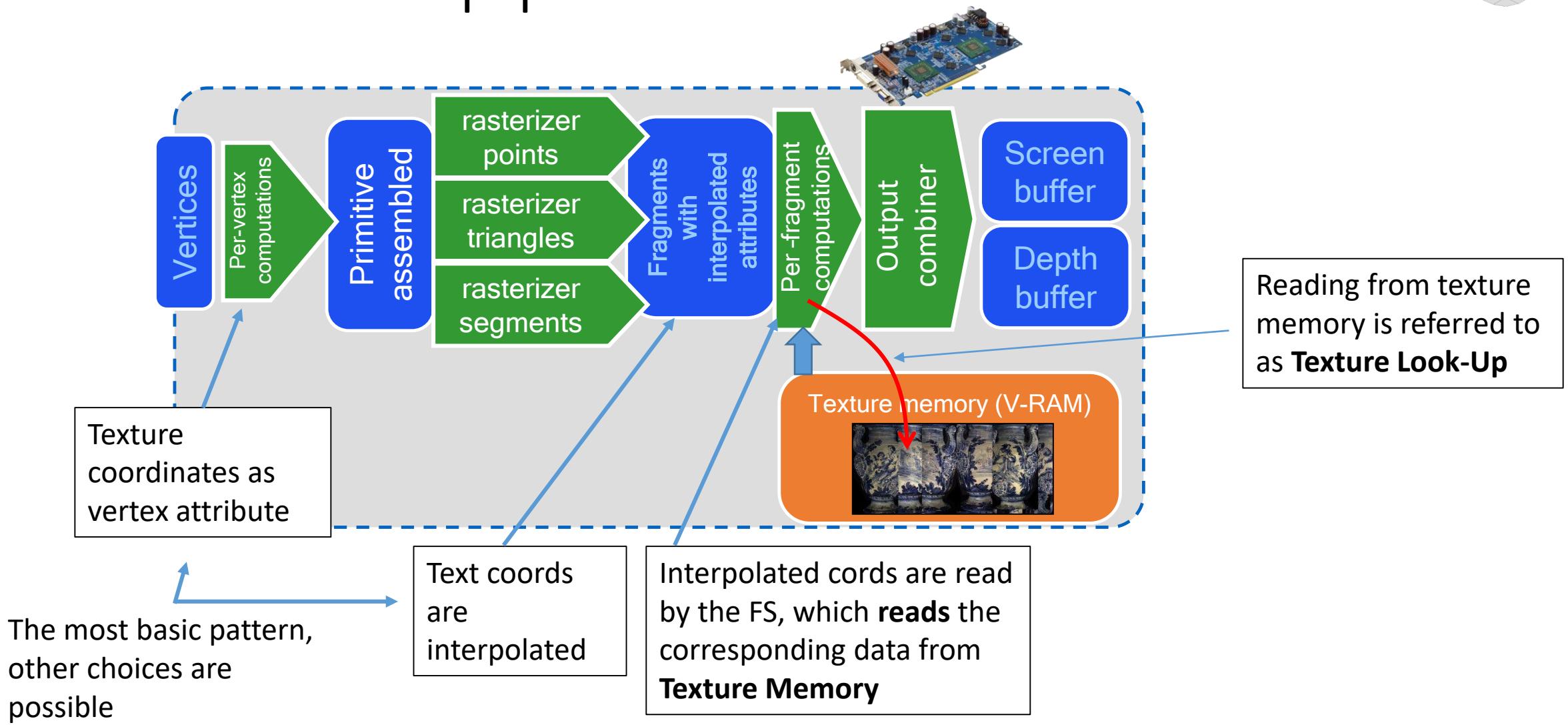
- What for polygon meshes more complex than a single polygon?
- **UV-unwrapping** is the process of «flattening» a surface from 3D to texture space



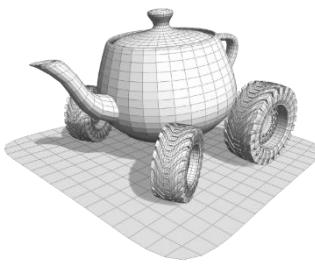
Zephyris at en.wikipedia, CC BY-SA 3.0,



Where in the pipeline?



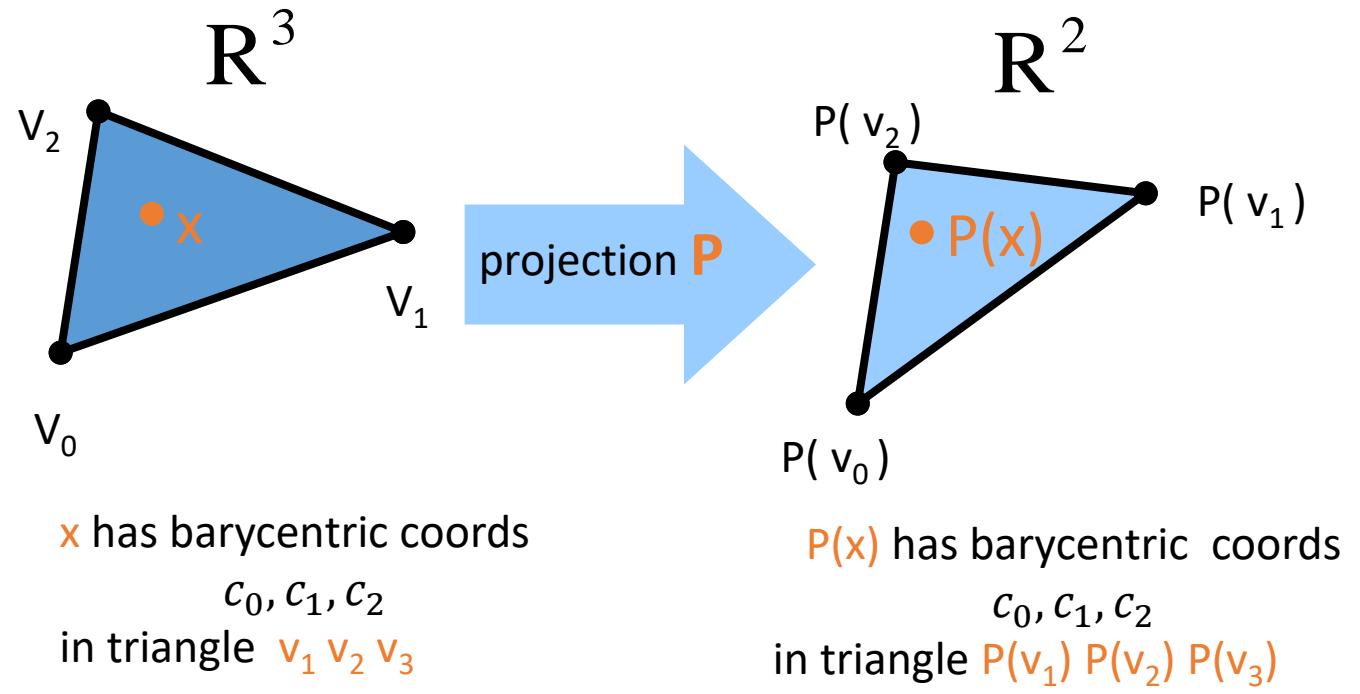
Attributes Interpolation with Perspective Projection



- We assumed that projection was a linear transformation, that is:

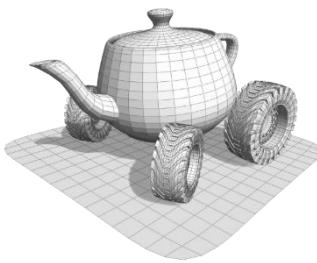
$$P(\lambda_a a + \lambda_b b) = \lambda_a P(a) + \lambda_b P(b)$$

- Orthogonal projection is an affine (and hence linear) transformation
- What about *perspective* projection? Is it a linear transformation?

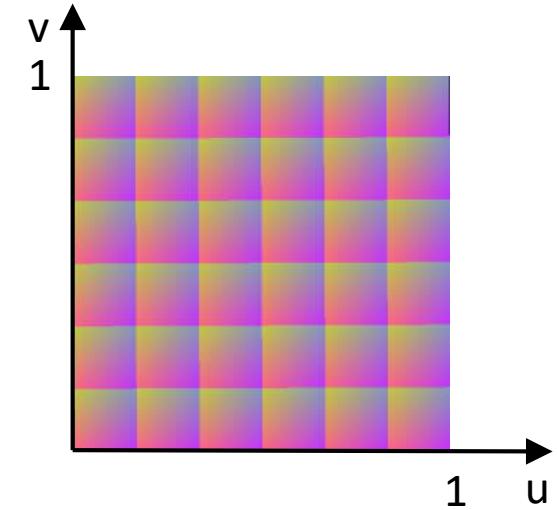
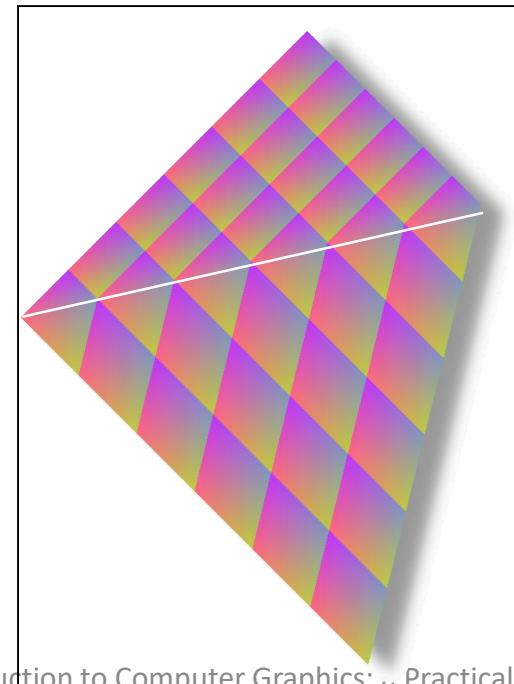
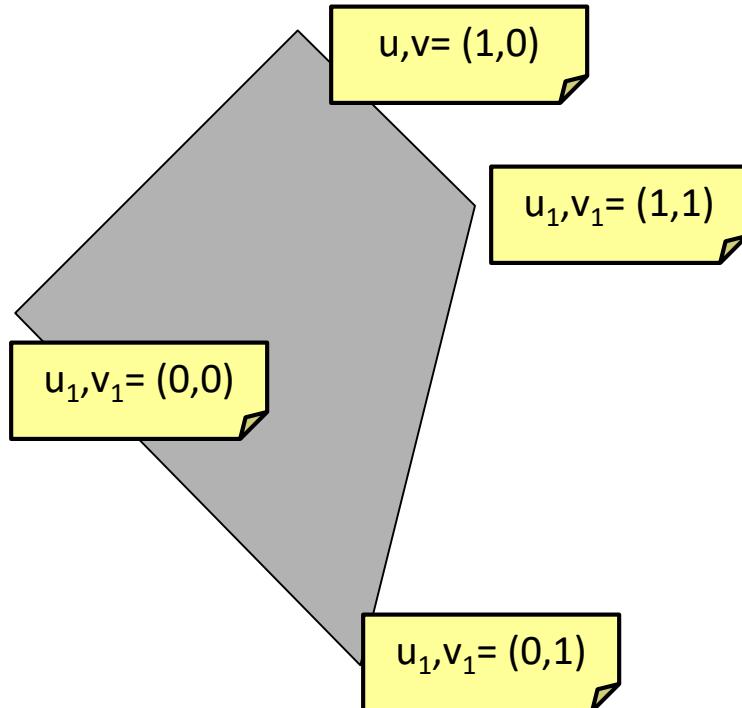


This is true for linear projections

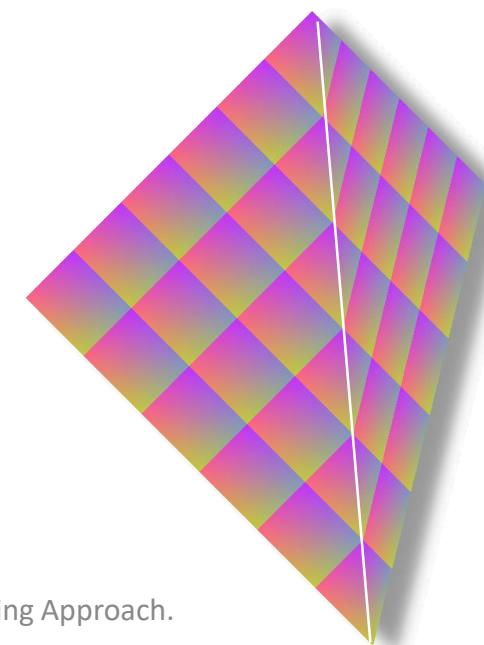
Attributes Interpolation with Perspective Projection



- Perspective projection **is not** affine -> is not linear

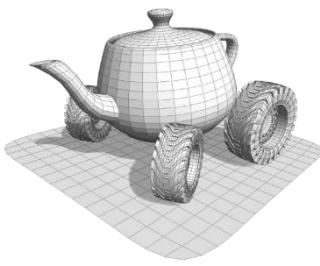


Texture space

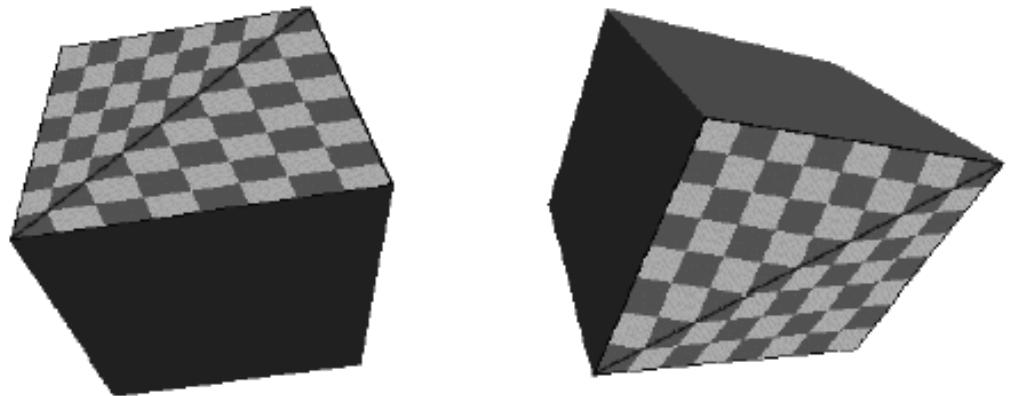


Textured polygons

Attributes Interpolation with Perspective Projection

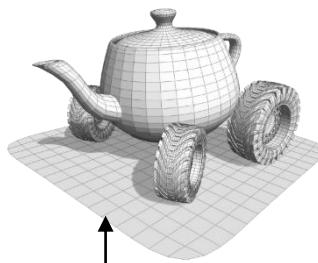


- With perspective projection, barycentric coordinates for the attribute are not those assigned to vertices
- Using them would result in a visible distortion
- What about the interpolated color and normal? Is wrong for them too?



Yes

Attributes Interpolation with Perspective Projection



$x_0, x_1 \in \mathbb{R}$: position of the segment vertices in view space

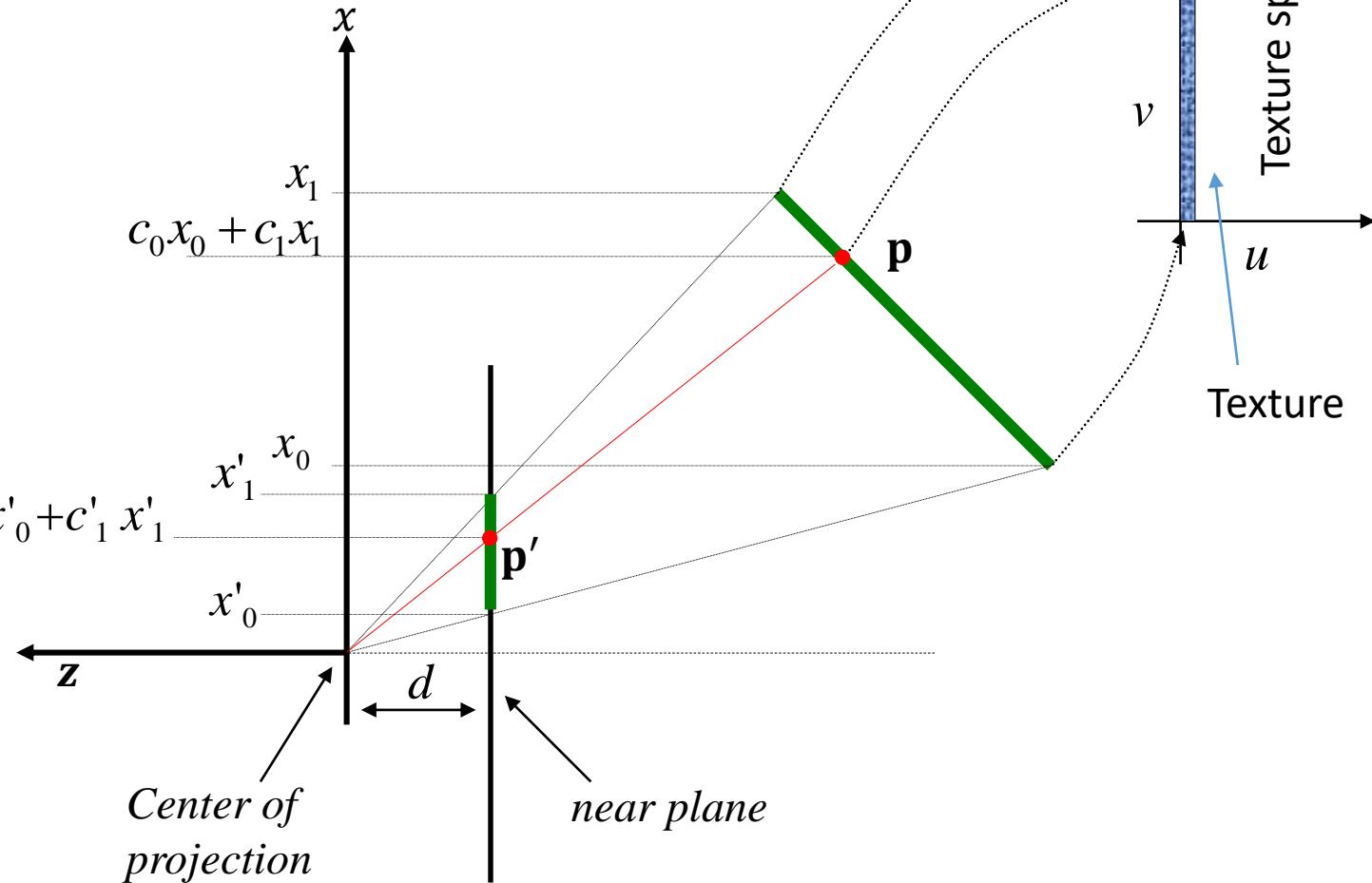
$a_0, a_1 \in \mathbb{R}^n$: vertex attributes to be interpolated

$c_0, c_1 \in \mathbb{R}$: barycentric coords of \mathbf{p}

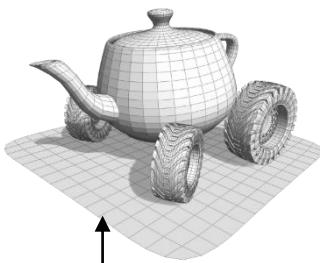
$x'_0, x'_1 \in \mathbb{R}$: position of the segment vertices in the perspective plane

$c'_0, c'_1 \in \mathbb{R}$: barycentric coords of \mathbf{p}'

$$c'_0, c'_1 \neq c_0, c_1$$



Attributes Interpolation with Perspective Projection

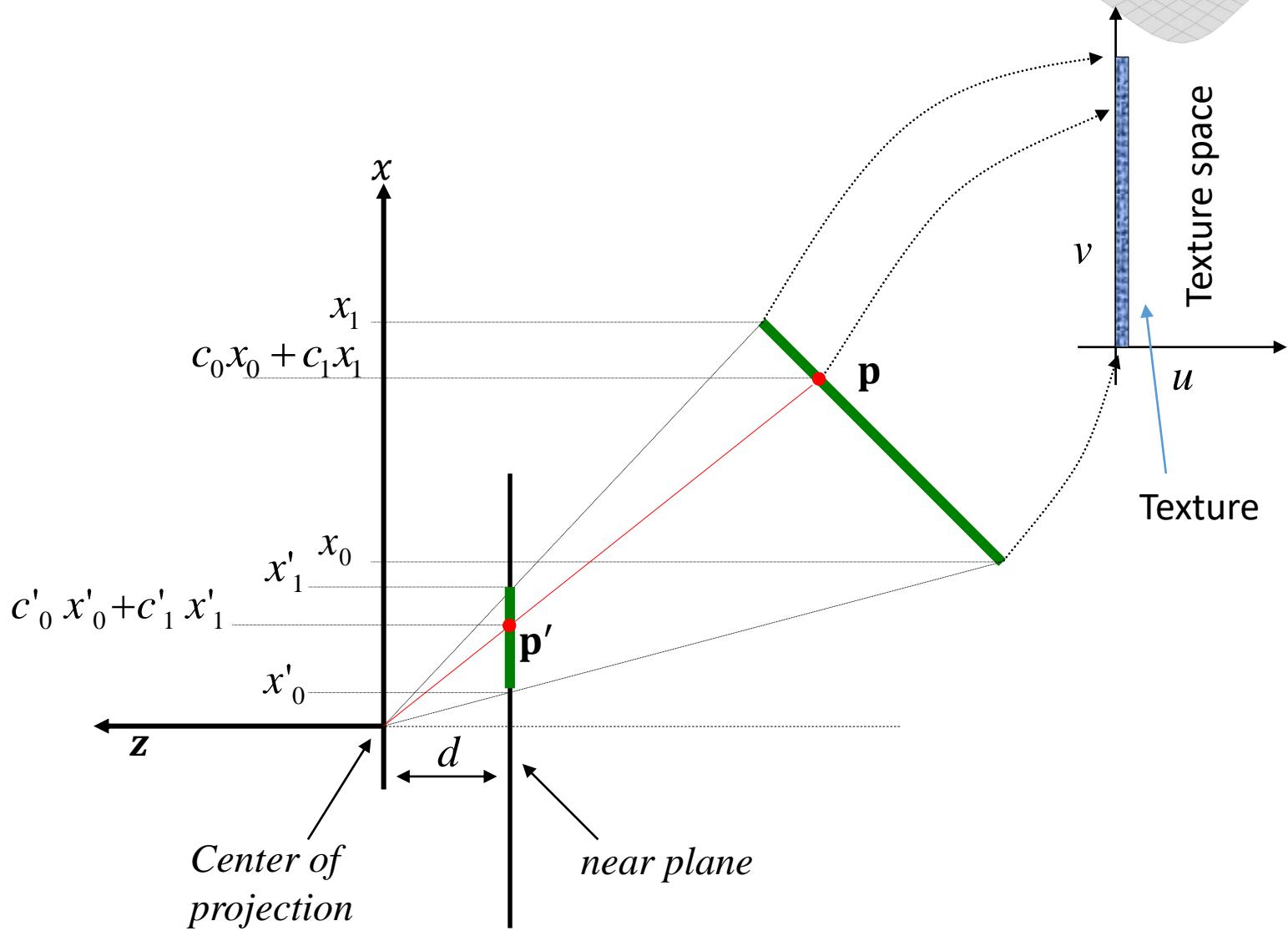


$$\mathbf{p}' = \begin{pmatrix} c'_0 x'_0 + c'_1 x'_1 \\ 1 \end{pmatrix}$$

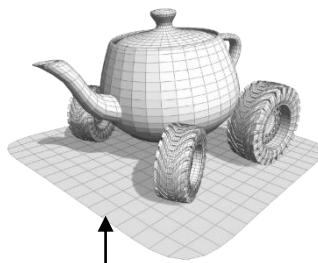
$$\begin{bmatrix} x'_0 \\ z'_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ -z_0/d \\ z_0 \\ -z_0/d \end{bmatrix} = \begin{bmatrix} \frac{x_0}{-z_0/d} \\ \frac{z_0}{-z_0/d} \\ 1 \end{bmatrix}$$

$$x'_0 = \frac{x_0}{-z_0/d} \quad x'_1 = \frac{x_1}{-z_1/d}$$

$$\mathbf{p}' = \begin{pmatrix} c'_0 \frac{x_0}{(-z_0/d)} + c'_1 \frac{x_1}{(-z_1/d)} \\ 1 \end{pmatrix}$$



Attributes Interpolation with Perspective Projection

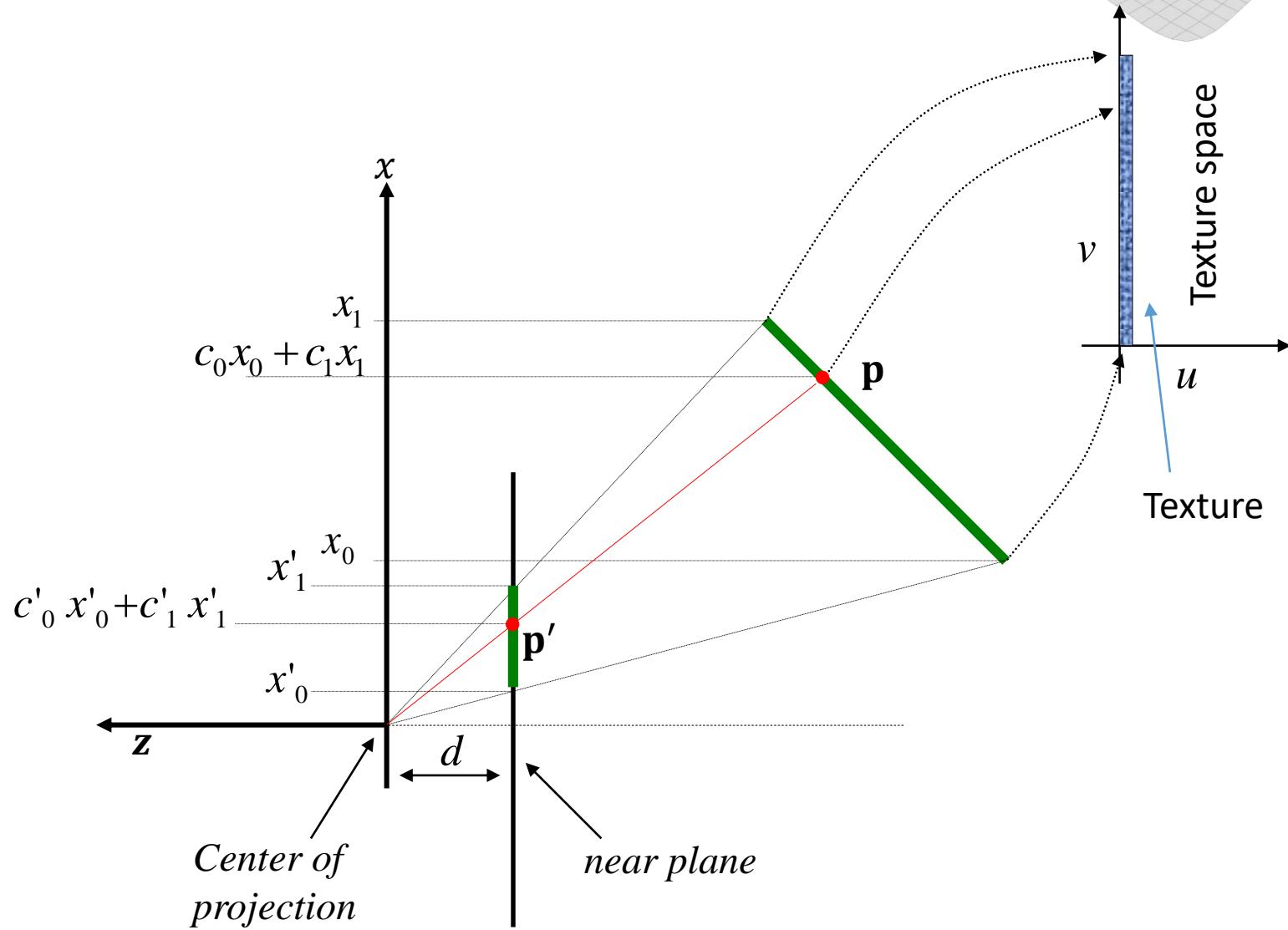


$$w_o = -\frac{z_0}{d} \quad w_1 = -\frac{z_1}{d}$$

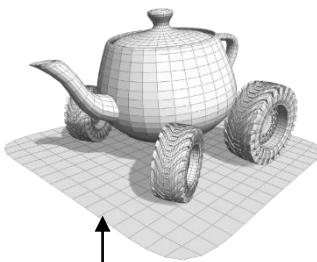
$$\mathbf{p}' = \begin{pmatrix} c'_0 \frac{x_0}{w_0} + c'_1 \frac{x_1}{w_1} \\ 1 \end{pmatrix}$$

Recall that in homogeneous coordinates:

$$\begin{bmatrix} a \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda a \\ \lambda \end{bmatrix}, \forall \lambda \neq 0$$



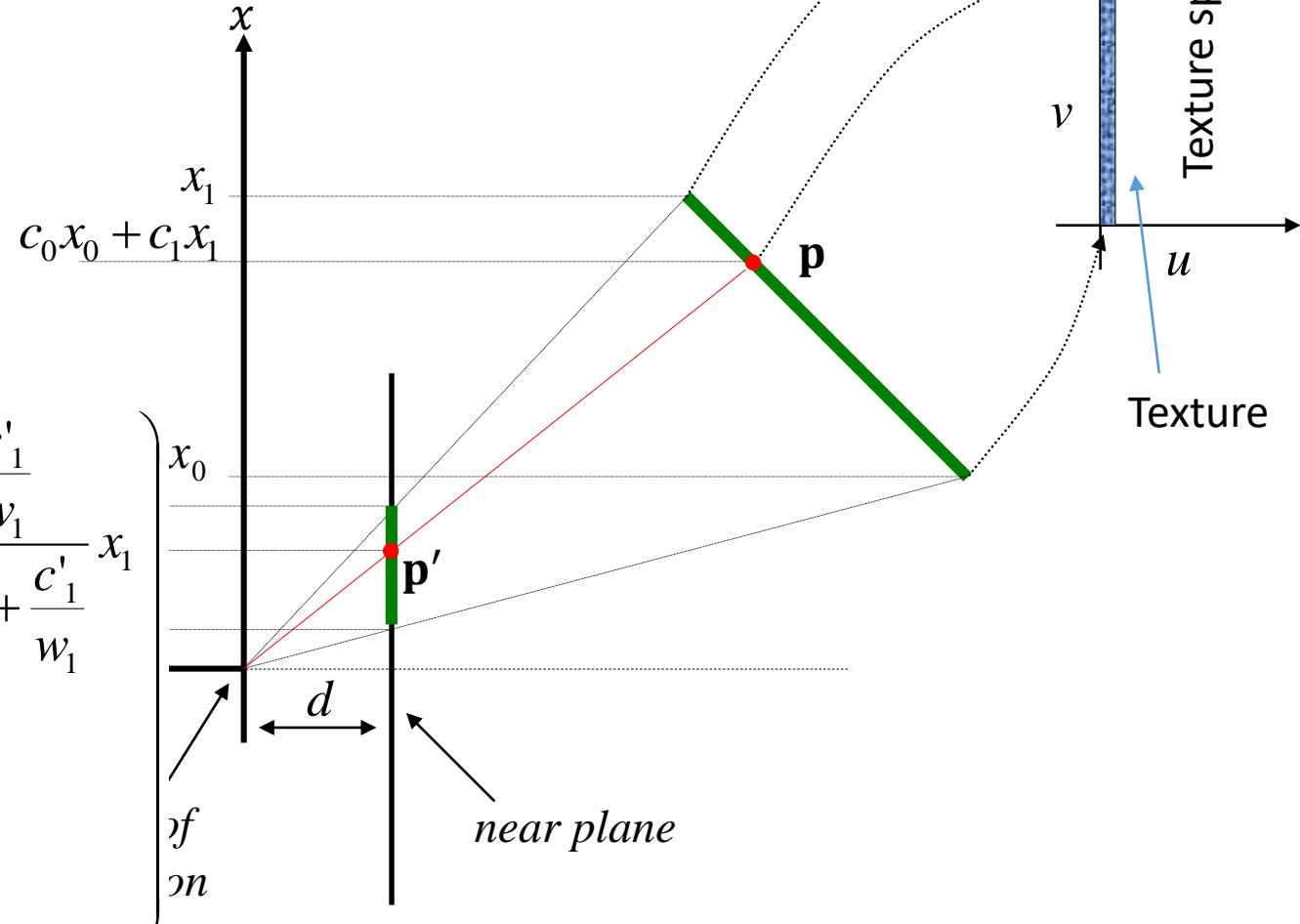
Attributes Interpolation with Perspective Projection



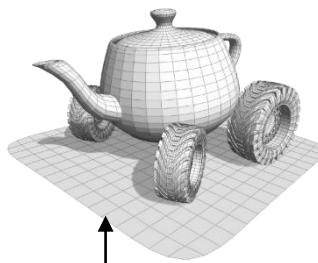
$$w_0 = -\frac{z_0}{d} \quad w_1 = -\frac{z_1}{d}$$

$$\mathbf{p}' = \begin{pmatrix} c'_0 \frac{x_0}{w_0} + c'_1 \frac{x_1}{w_1} \\ 1 \end{pmatrix}$$

$$\mathbf{p}' = \begin{pmatrix} \frac{c'_0}{w_0} x_0 + \frac{c'_1}{w_1} x_1 \\ \dots \\ 1 \end{pmatrix} \cdot \frac{1}{\frac{c'_0}{w_0} + \frac{c'_1}{w_1}} = \begin{pmatrix} \frac{c'_0}{w_0} & \frac{c'_1}{w_1} \\ \frac{\frac{c'_0}{w_0} + \frac{c'_1}{w_1}}{w_0} x_0 + \frac{\frac{c'_0}{w_0} + \frac{c'_1}{w_1}}{w_1} x_1 & \frac{\frac{c'_0}{w_0} + \frac{c'_1}{w_1}}{w_0} & \frac{\frac{c'_0}{w_0} + \frac{c'_1}{w_1}}{w_1} x_1 \\ \dots \\ \frac{1}{\frac{c'_0}{w_0} + \frac{c'_1}{w_1}} \end{pmatrix}$$



Attributes Interpolation with Perspective Projection



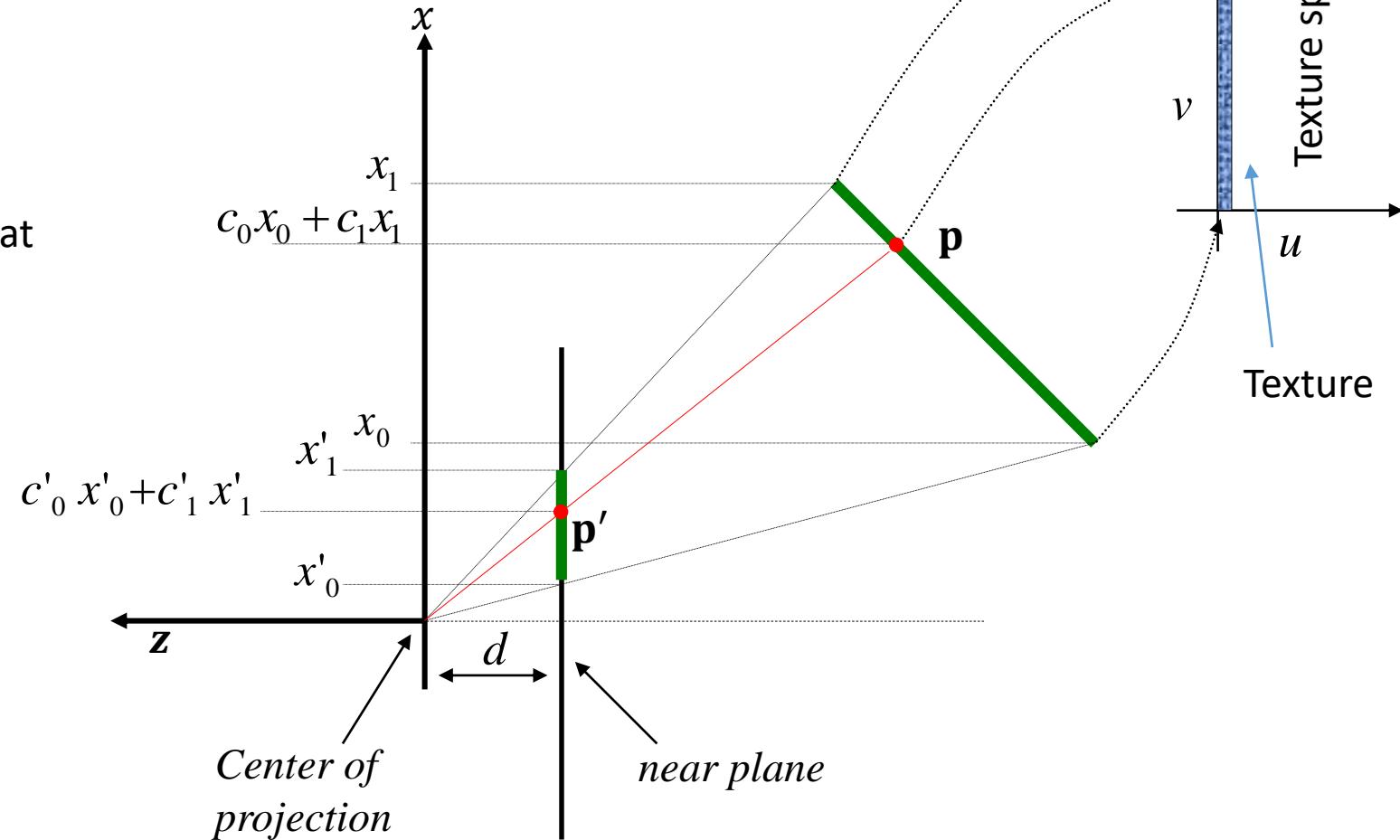
The coefficients of x_0 and x_1 sum to 1 hence the point is on the segment

Also, the point is along the projector

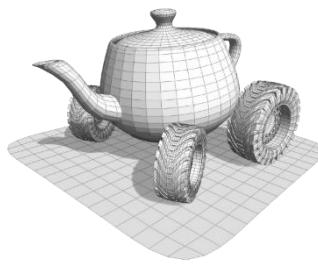
\Rightarrow

The point is \mathbf{p} , so we have found the barycentric coordinates for x_0 and x_1 that give \mathbf{p}

$$\mathbf{p}' = \left(\begin{array}{c} c_0 \\ \frac{c'_0}{w_0} x_0 + \frac{c'_1}{w_1} x_1 \\ \frac{c'_0 + c'_1}{w_0 + w_1} \\ \dots \\ \frac{1}{c'_0 + c'_1} \end{array} \right)$$



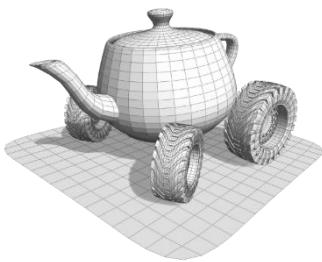
Attributes Interpolation with Perspective Projection



- While rasterizing, the attributes are interpolated using equation
- It is called **hyperbolic interpolation**
- Other old names:
 - perfect texture mapping
 - Method of the 3 magic vectors

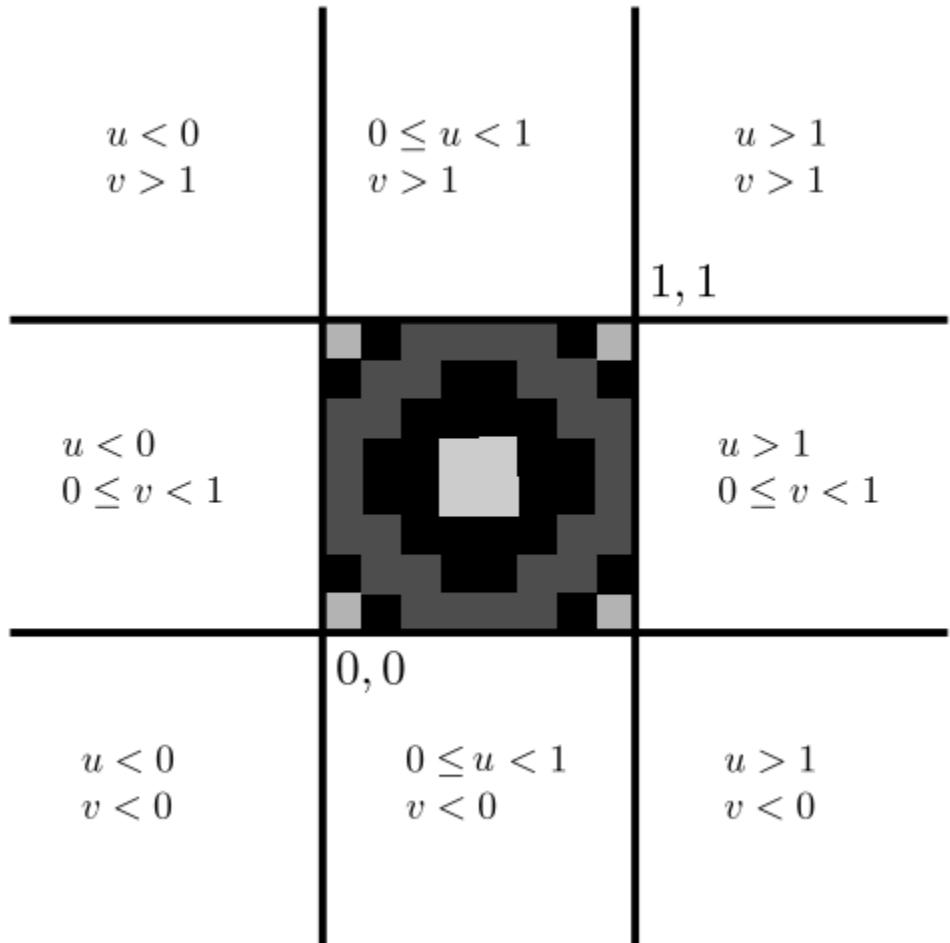
$$a' = \frac{\frac{c'_0}{w_0} a_0 + \frac{c'_1}{w_1} a_1}{\frac{c'_0}{w_0} + \frac{c'_1}{w_1}}$$

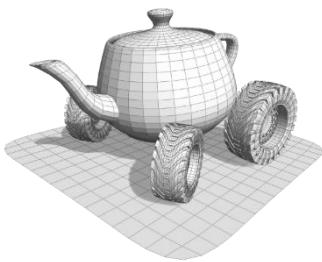
c' : screen space bar. coord.
 a' : perspectively correct
interpolated attribute a



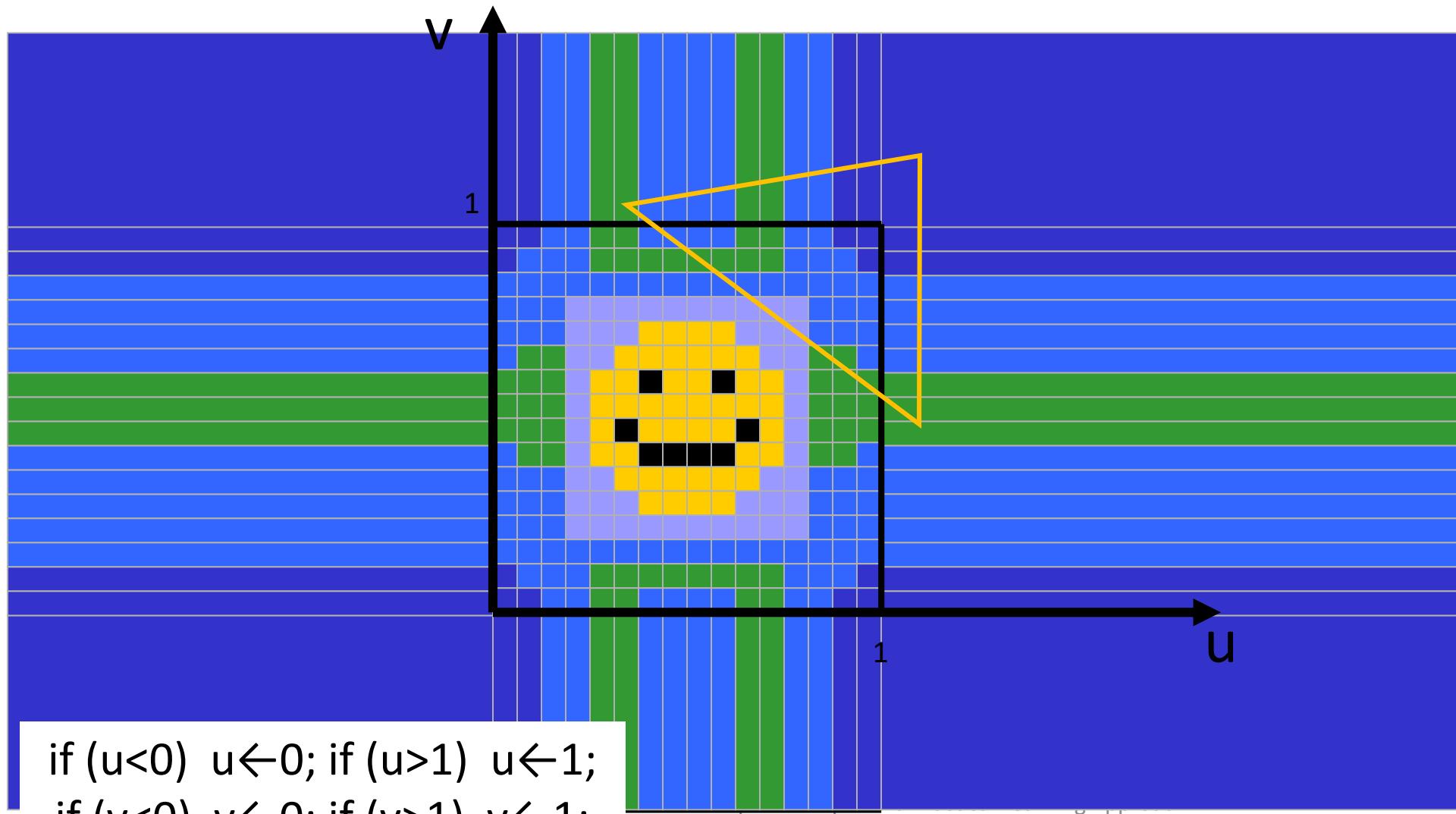
Handling UV-coordinates

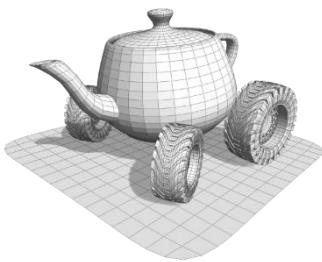
- It is useful to extend the domain of the UV-coordinates to $[-\infty, -\infty][+\infty, +\infty]$



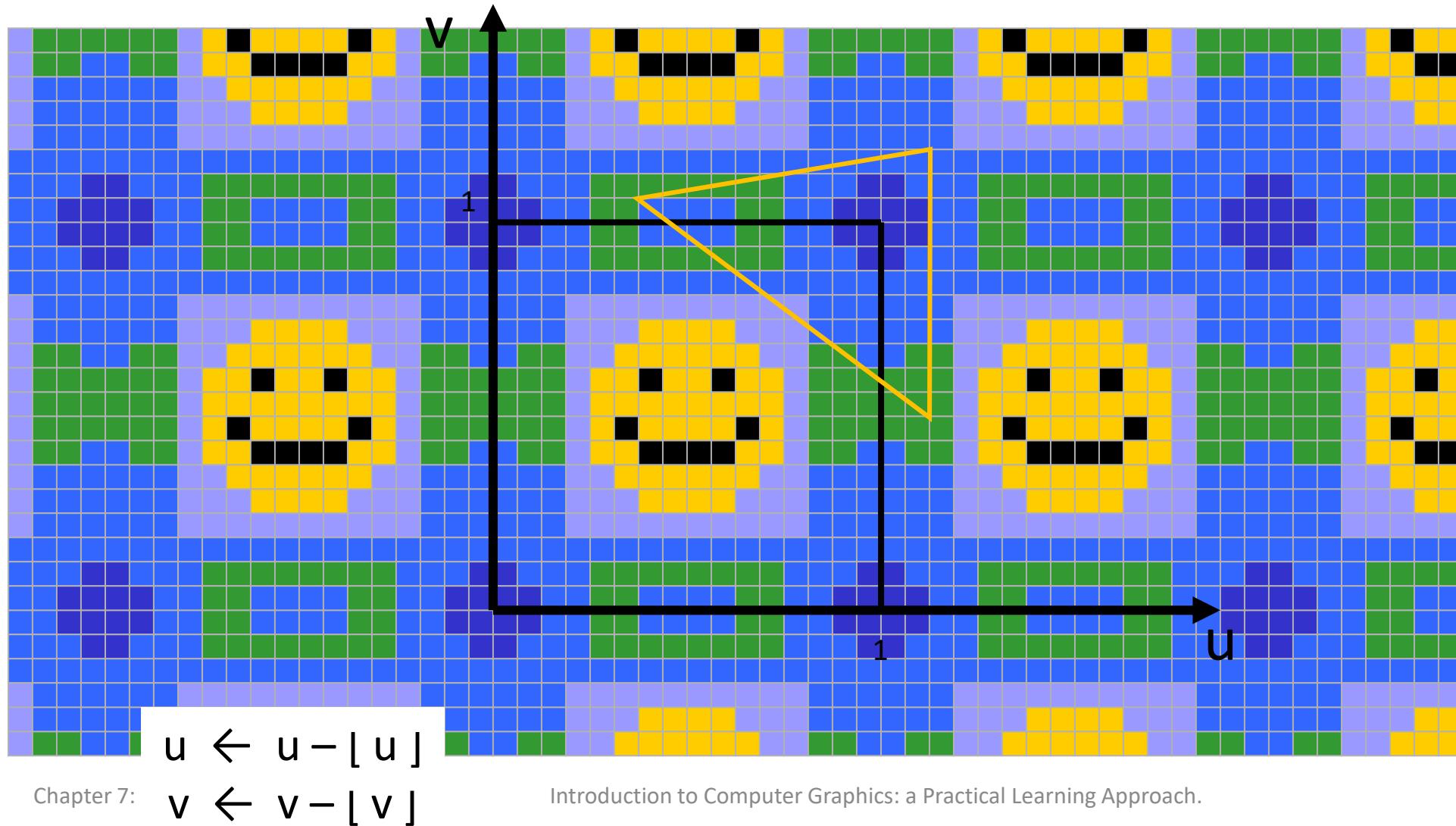


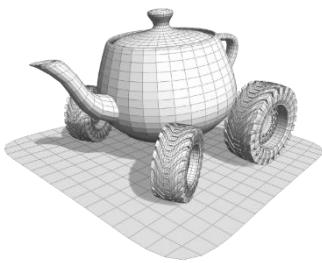
UV-coordinates wrapping: clamp



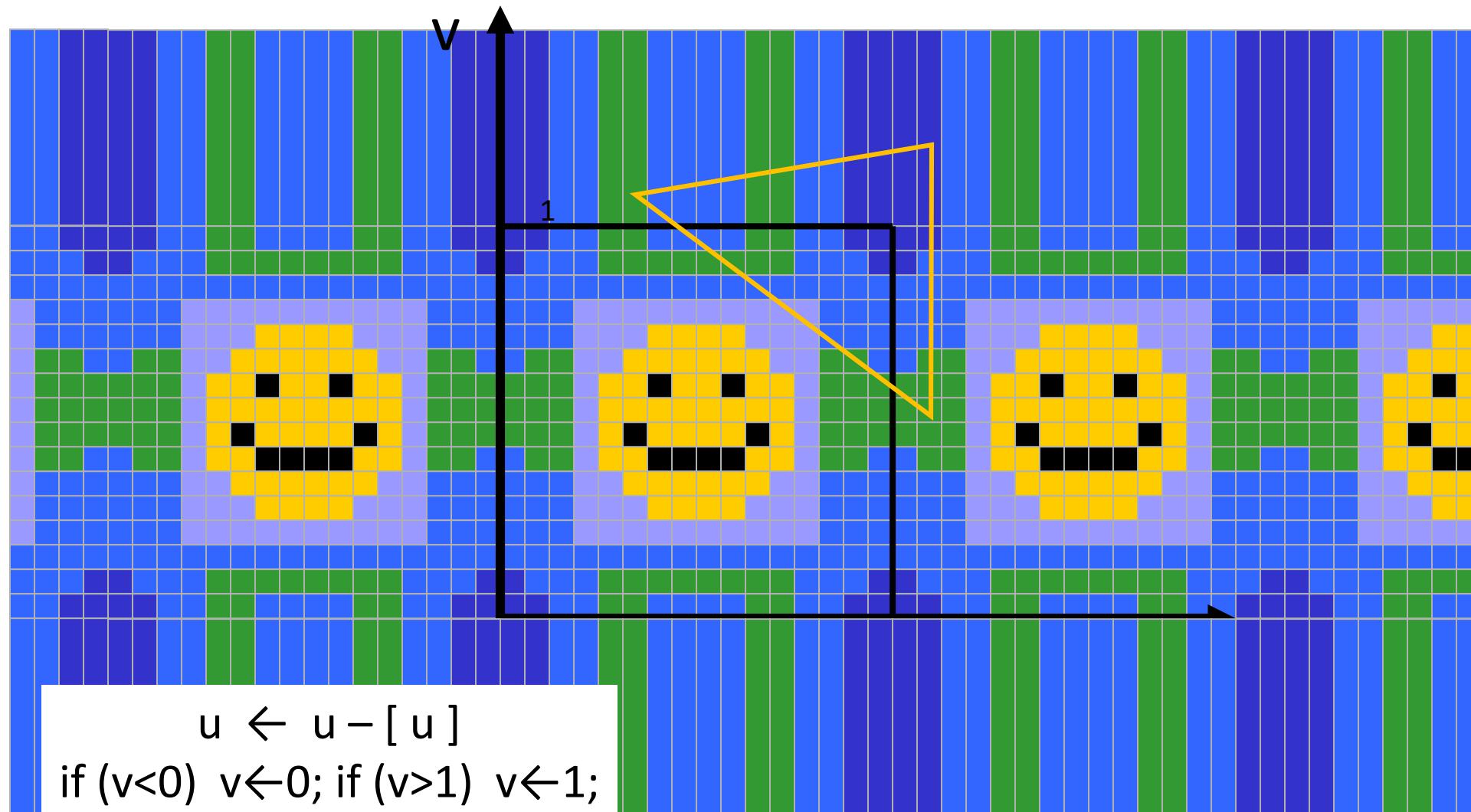


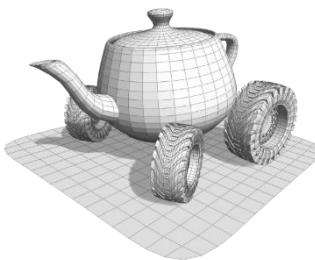
UV-coordinates wrapping: repeating





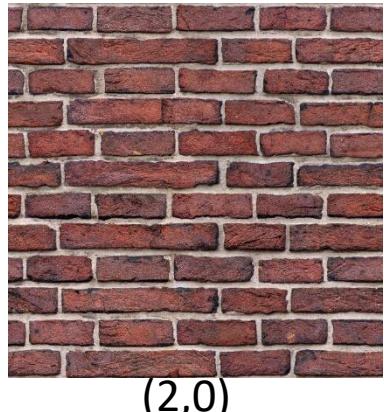
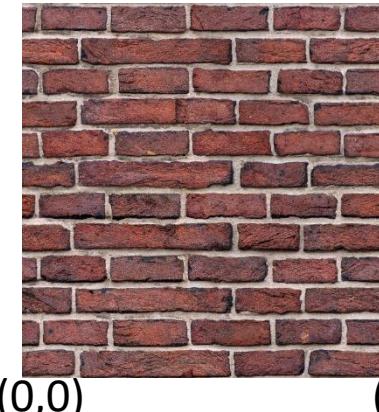
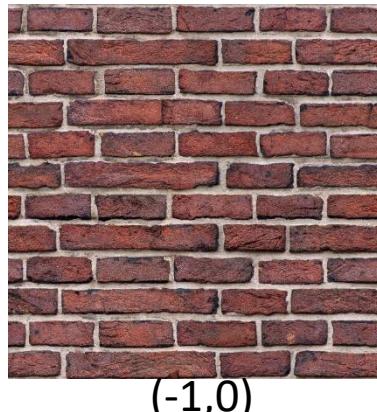
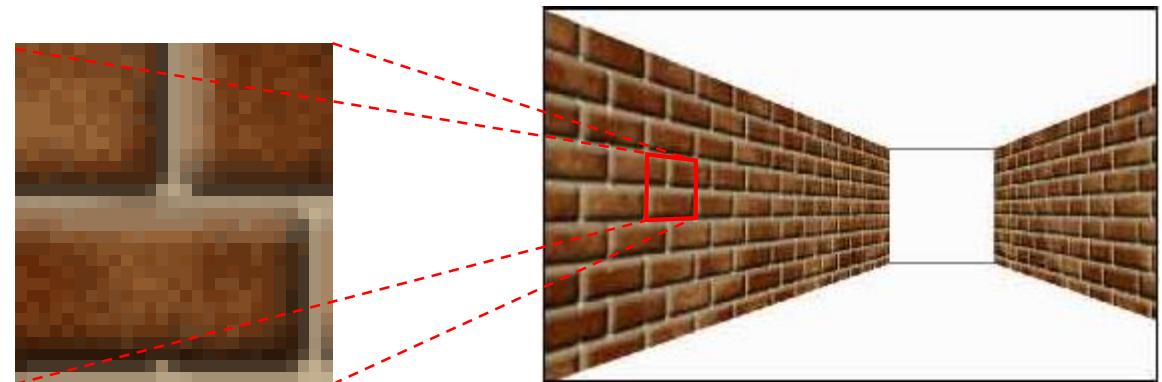
UV-coordinates wrapping: combinations



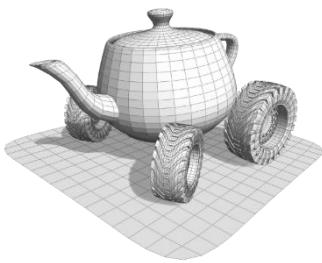


“Repeat” wrapping

- Wrapping UV coordinates is useful for showing *patterns*
- As long as the texture is tilable
- A **tileable** texture does not show discontinuities when placed side-by-side with a copy of itself

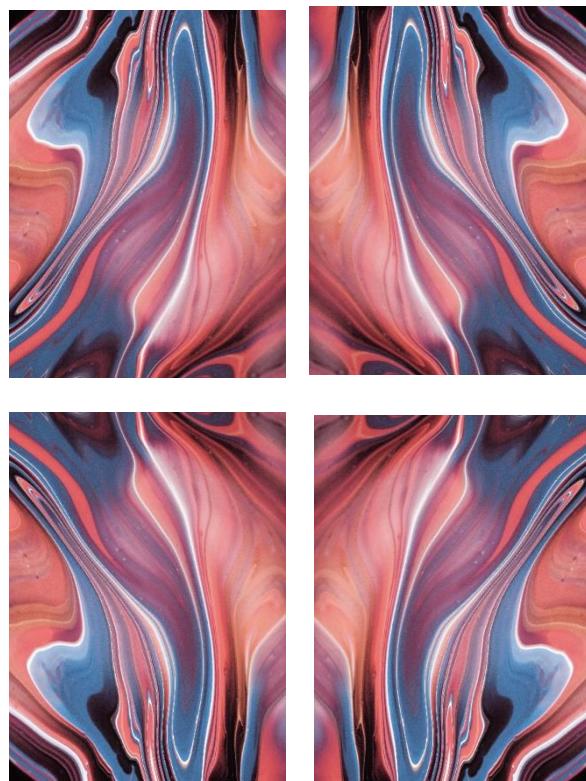
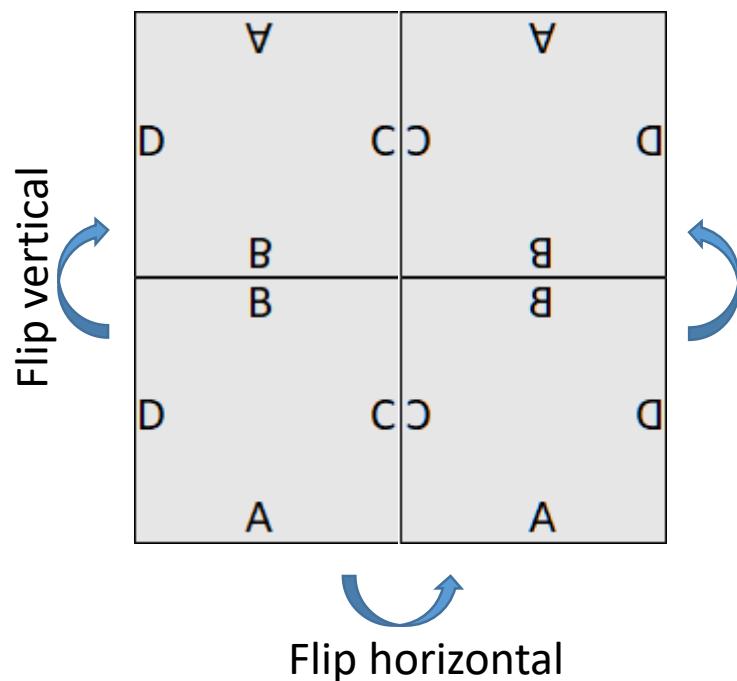


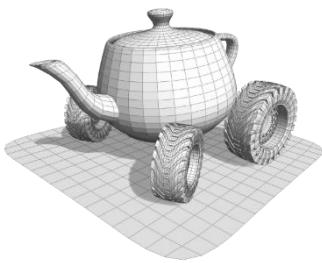
(2,0)



Making textures tileable

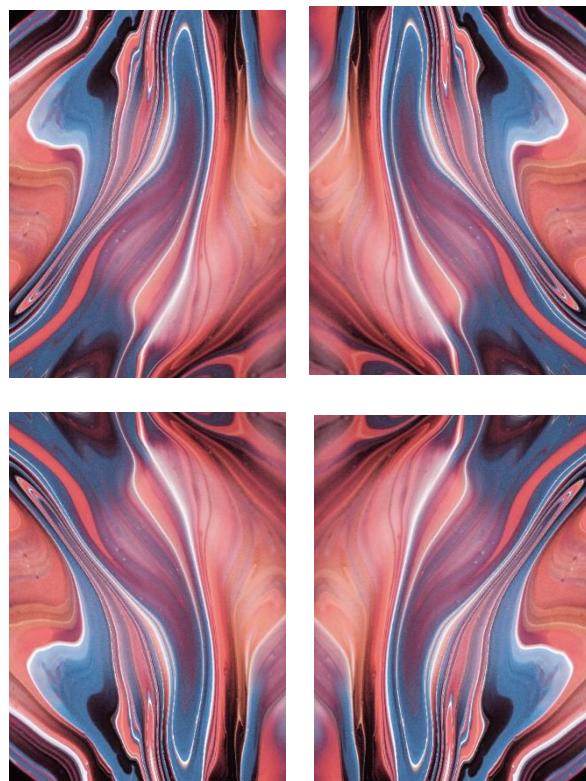
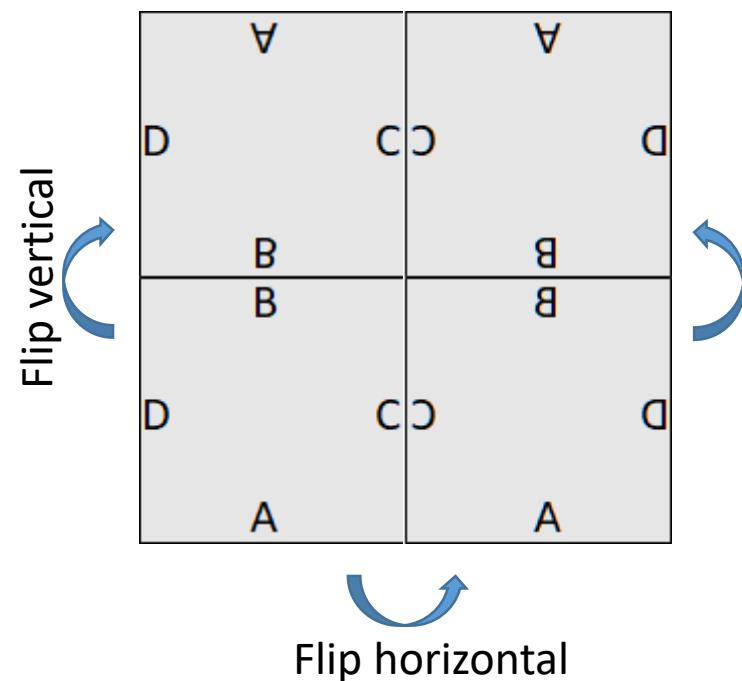
- We can make a texture tileable by duplication and mirroring along u and v

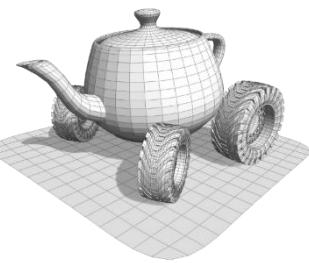




Making textures tileable

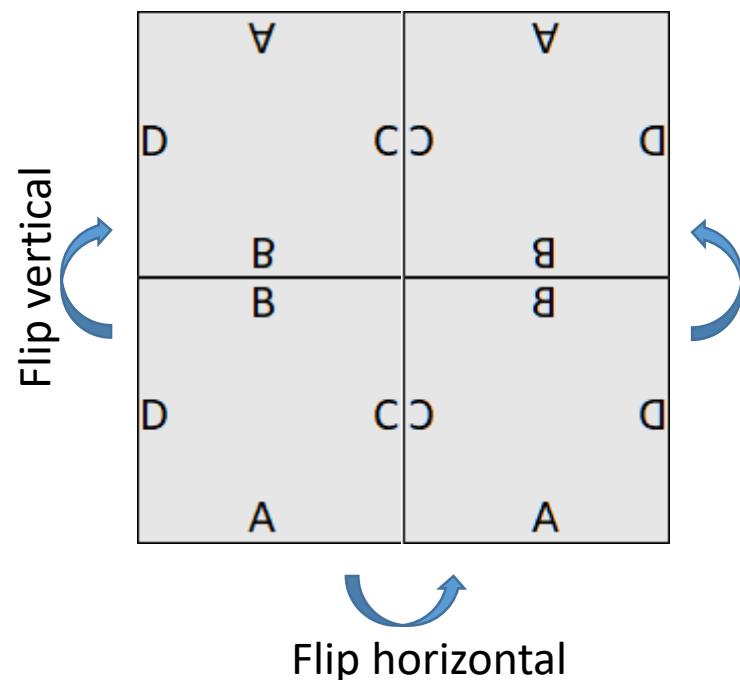
- We can make a texture tileable by duplication and mirroring along u and v

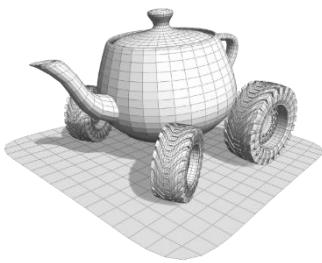




Making textures tileable

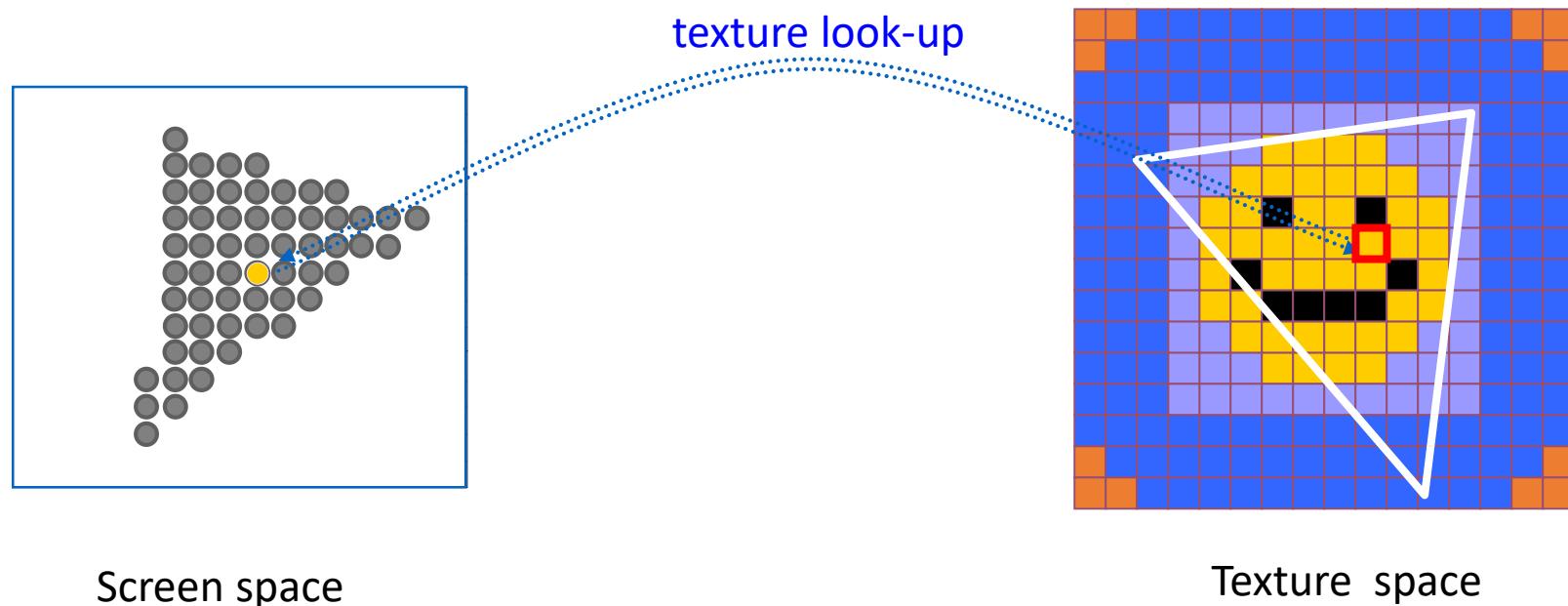
- We can make a texture tileable by duplication and mirroring along u and v



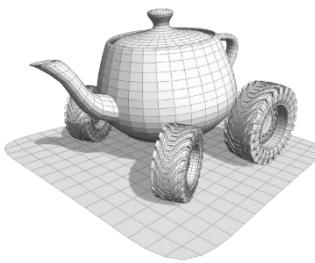


Texture Look-Up

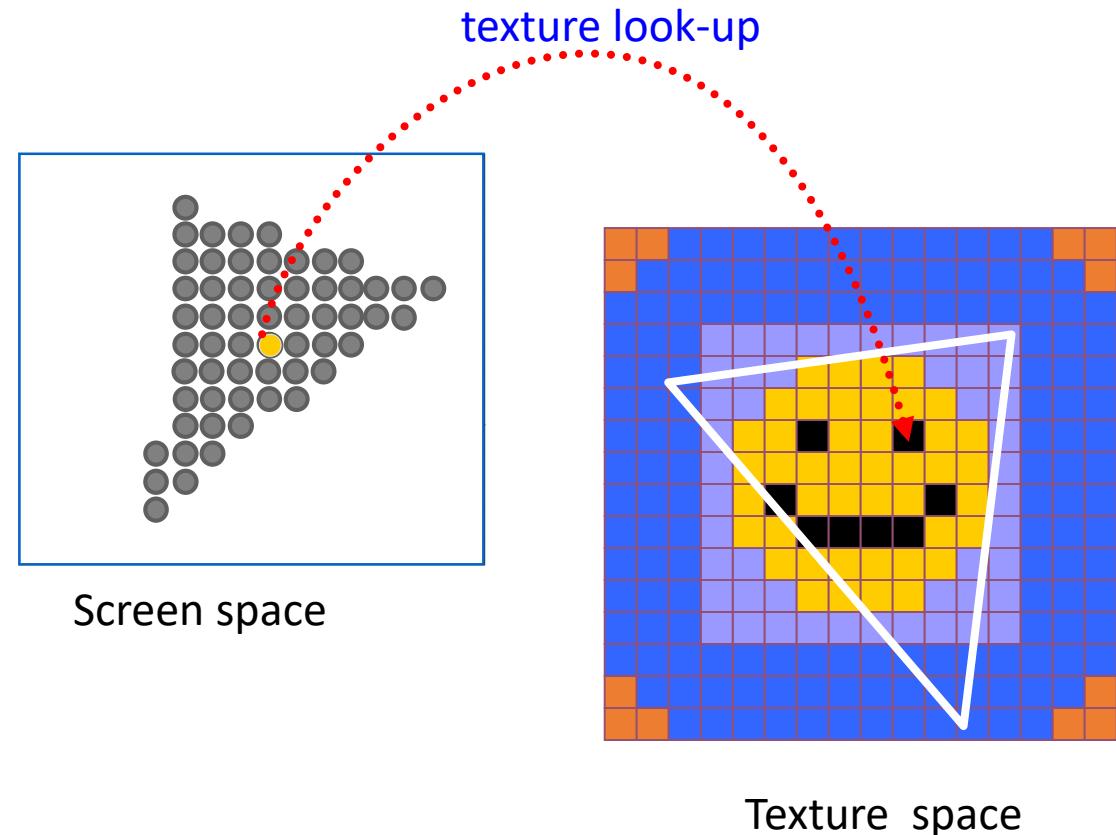
- Each fragment has its own texture coordinates in texture space and accesses to the texture



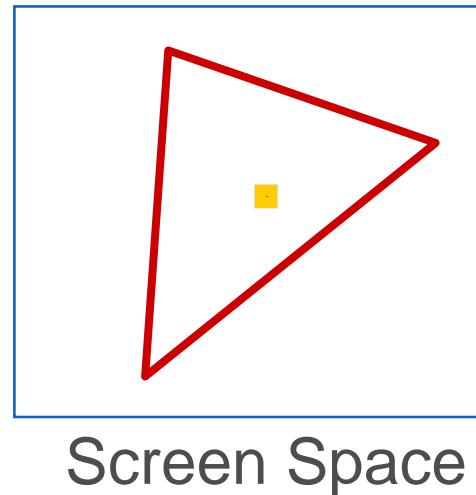
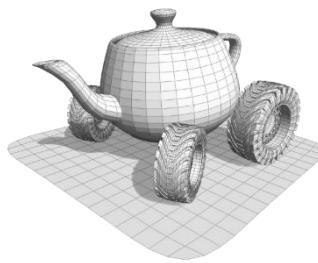
Texture Filtering: Magnification and Minification



- Texture coords address to a non-integer position in texture space
- In general, a fragment in screen space does not correspond to a texel in screen space
- What color should we read for the fragment?
- Two cases:
 - Magnification: a texel spans over multiple pixels
 - Minification: a pixel spans over multiple texels

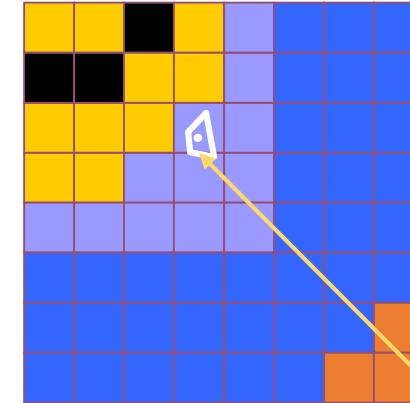


Texture Filtering: Magnification and Minification



magnification

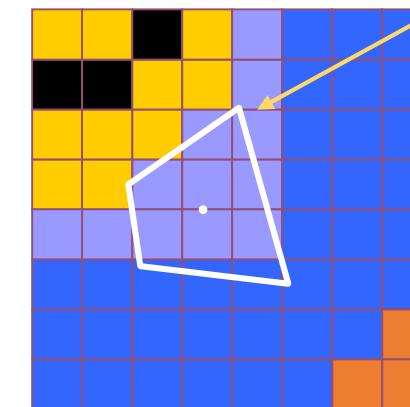
the pixel is smaller
than the texel



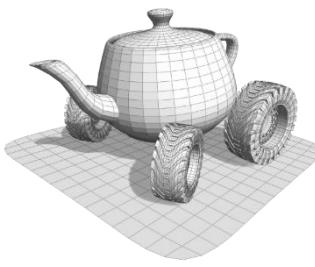
Pixel shape in
Texture Space

minification

the pixel is bigger
than the texel

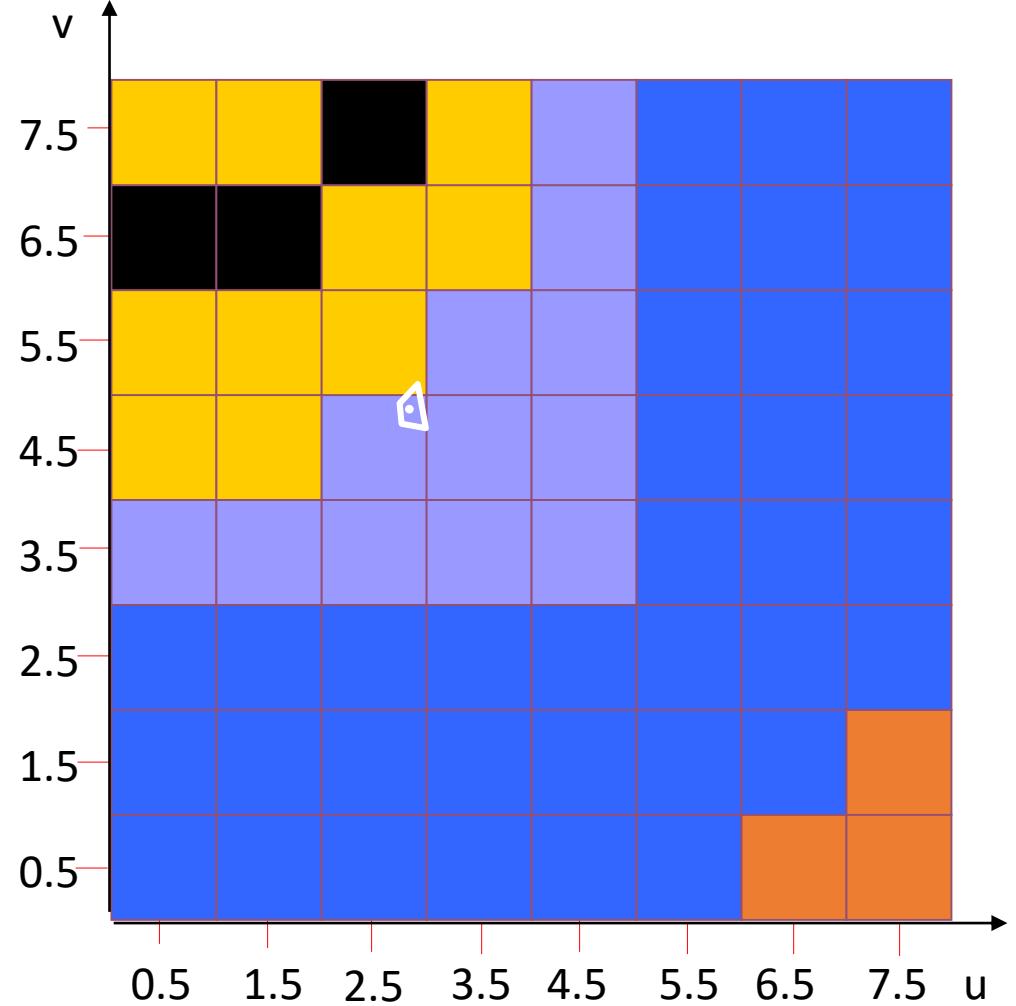


Texture Space

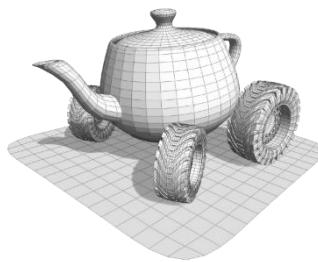


Texture Filtering: Magnification

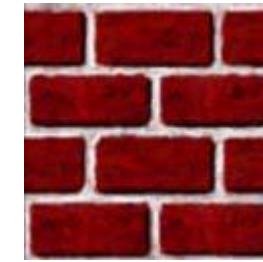
- **Nearest Filtering:** simply take the value (color) of the closest texels center
- In other words its round the UV coordinates to their closest integer coordinates



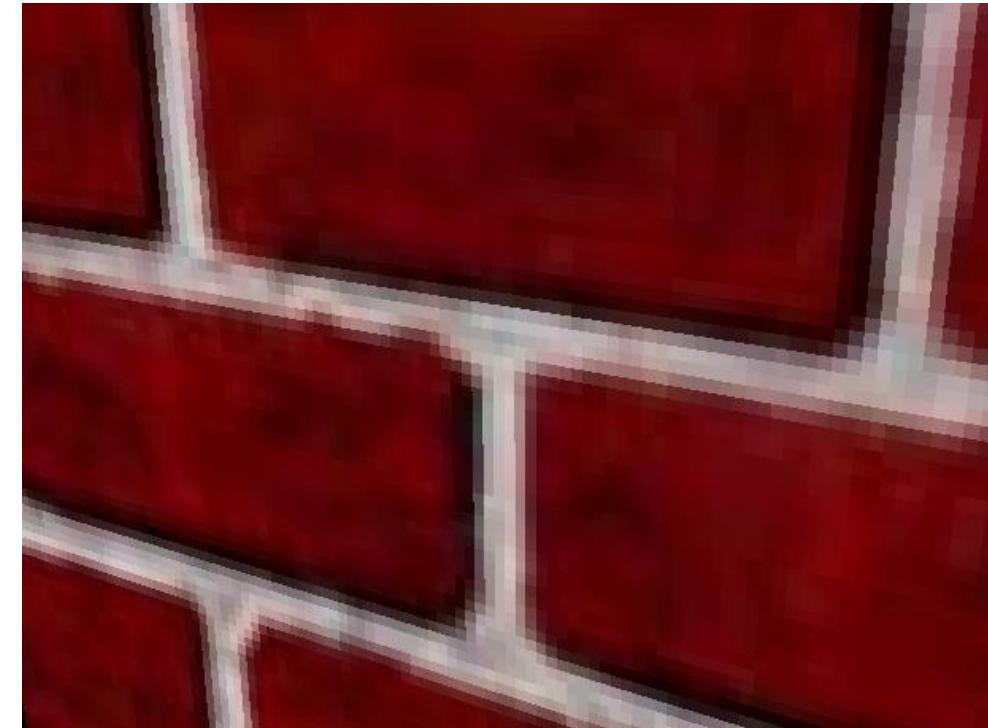
Magnification: Nearest Filtering

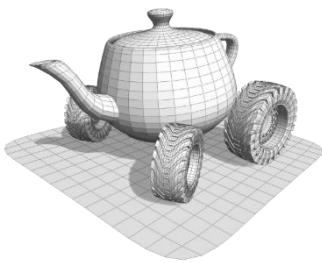


- With nearest filtering the shape of the pixels become clearly visible
- It is not what usually we want



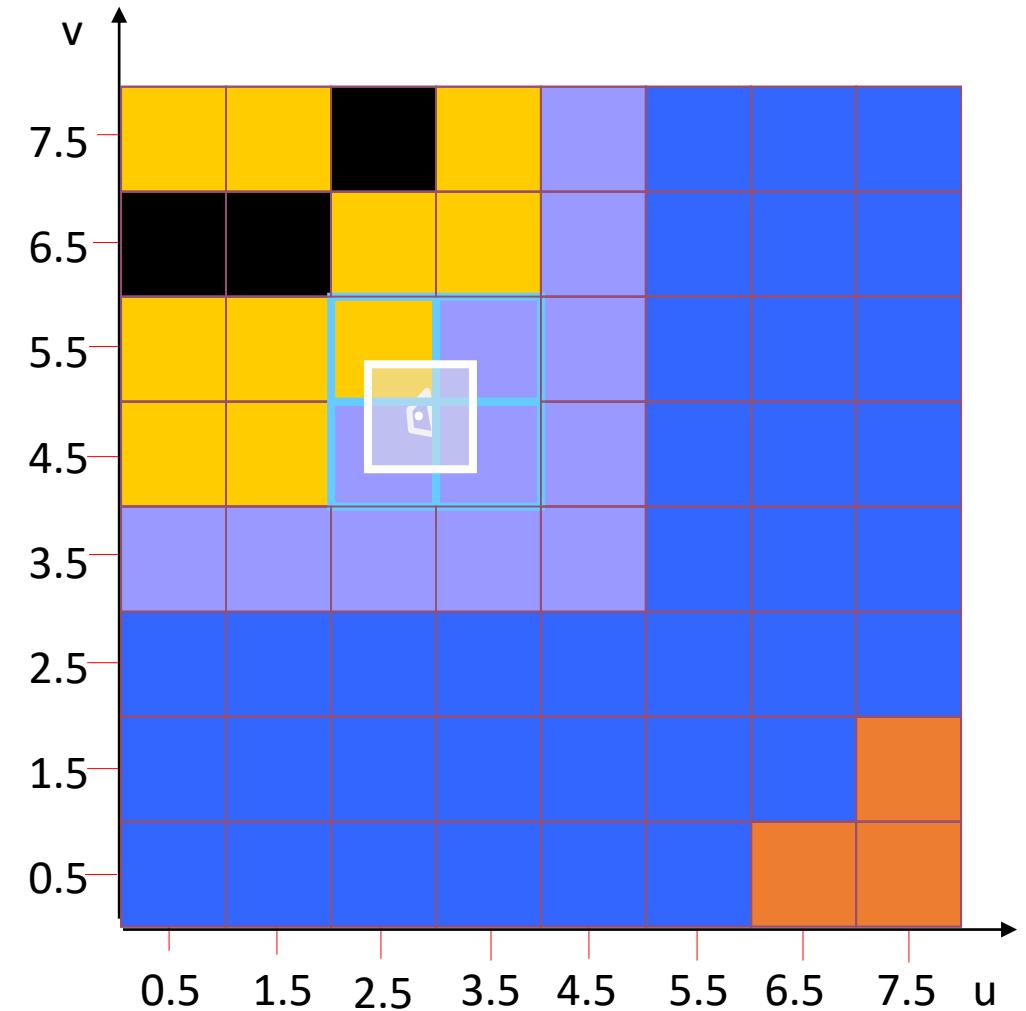
texture 128x128

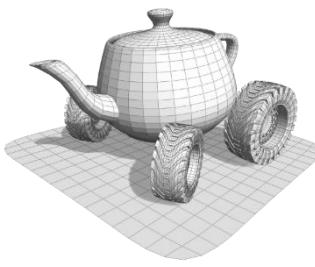




Magnification: Bilinear Interpolation

- Take a combination of the value of the four nearest texels
- Use bilinear interpolation





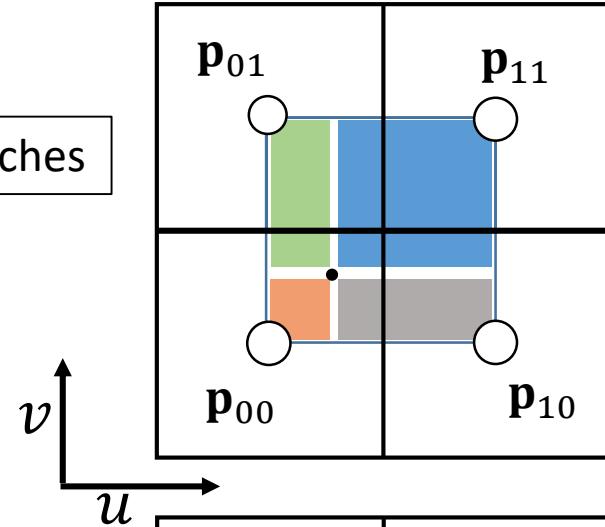
Magnification: Bilinear Interpolation

$c_{ij} == \text{value of texel at position } \mathbf{p}_{ij}$

Bilinear interpolation

$$c = [c_{00}(1 - u) + c_{10} u] (1 - v) + \\ [c_{01}(1 - u) + c_{11} u] v$$

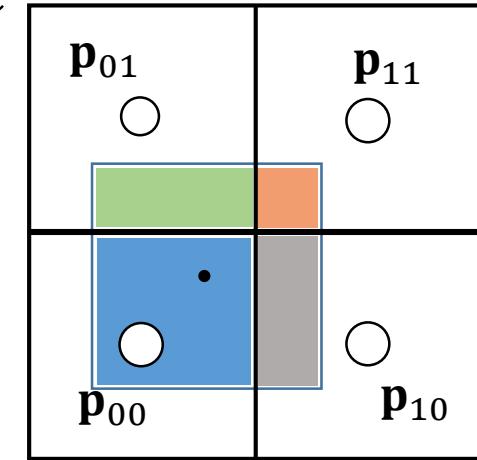
Just like degree 1 Bezier patches

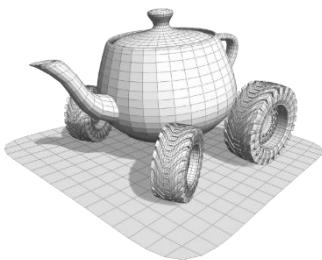


We can look at it as the portion of the square centered at p overlapping with the texels.

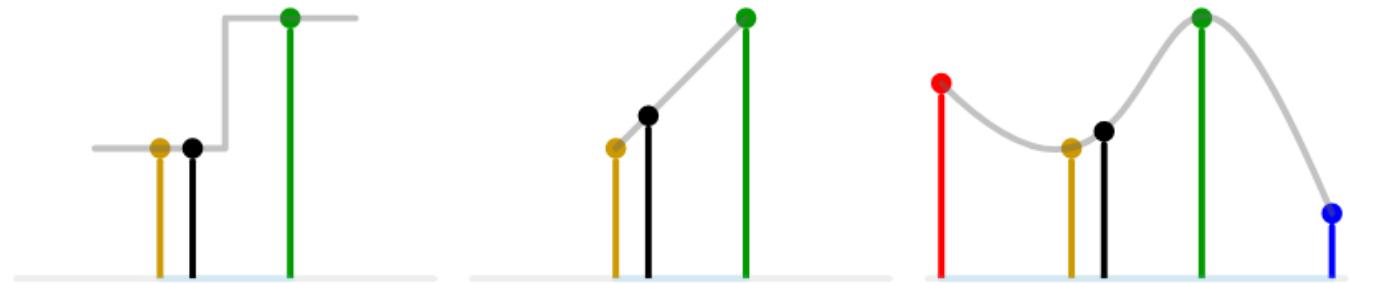
In both cases:

$$c = (1 - u)(1 - v)c_{00} + (1 - u)v c_{01} + (1 - v)u c_{10} + uv c_{11}$$





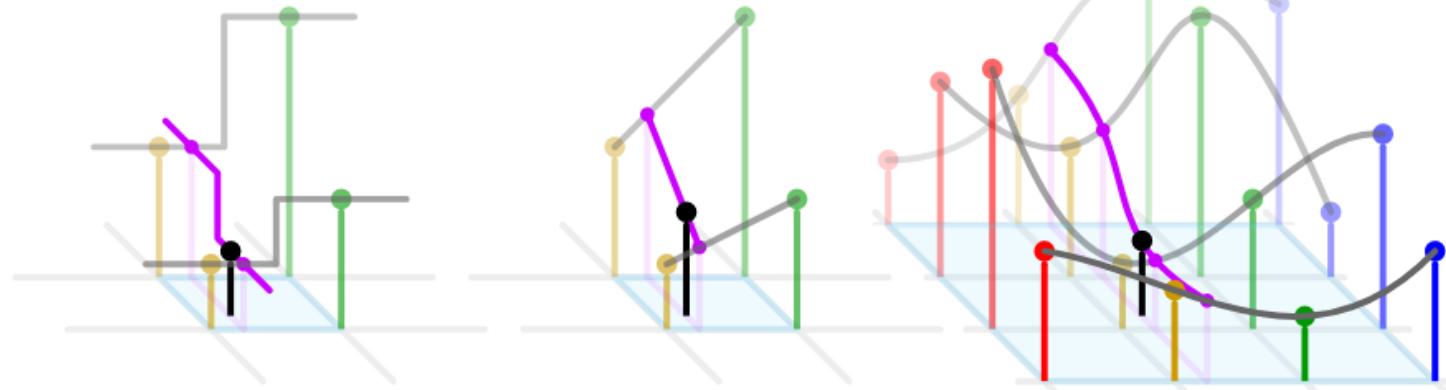
Magnification: Bilinear Interpolation



1D nearest-neighbour

Linear

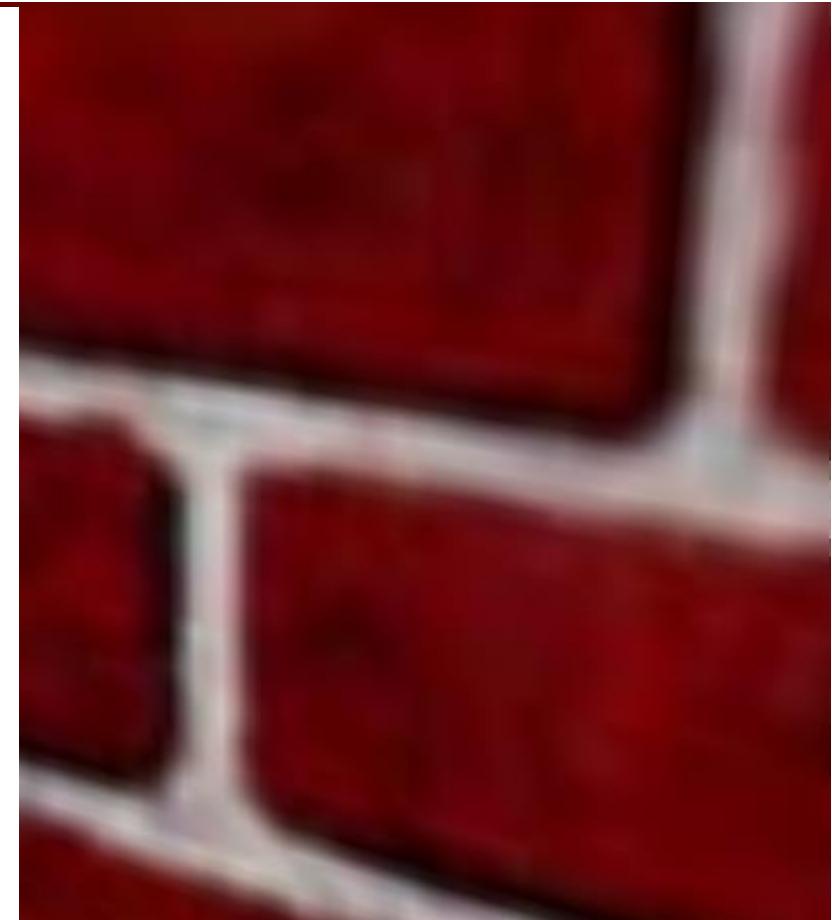
Cubic

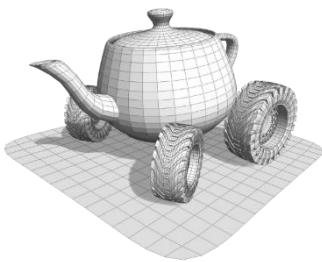


2D nearest-neighbour

Bilinear

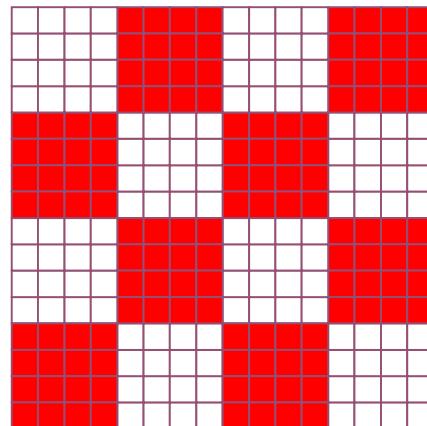
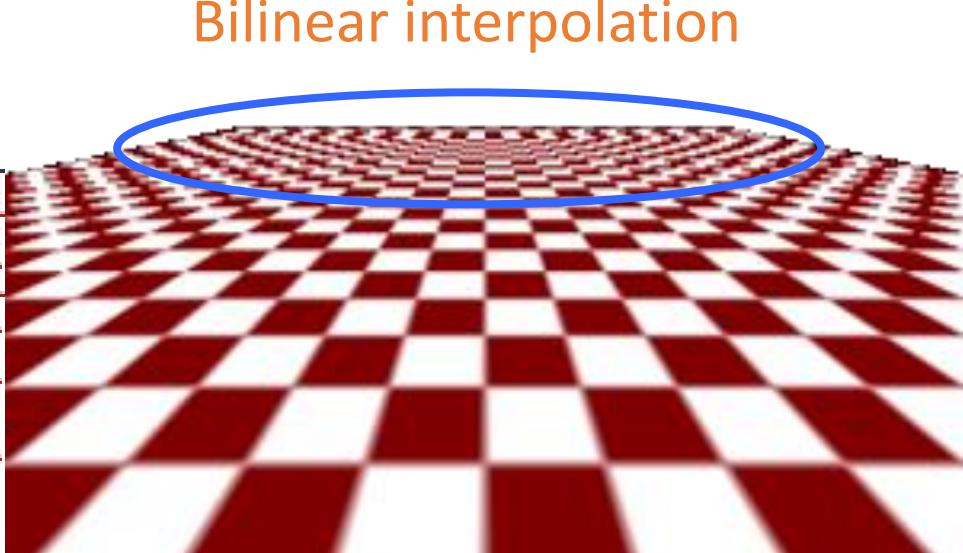
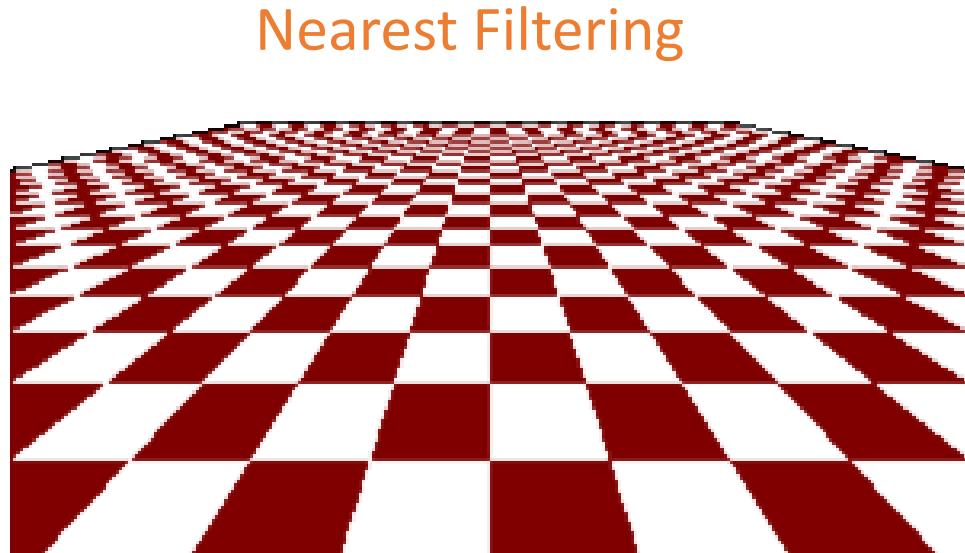
Bicubic





Texture Filtering: Minification

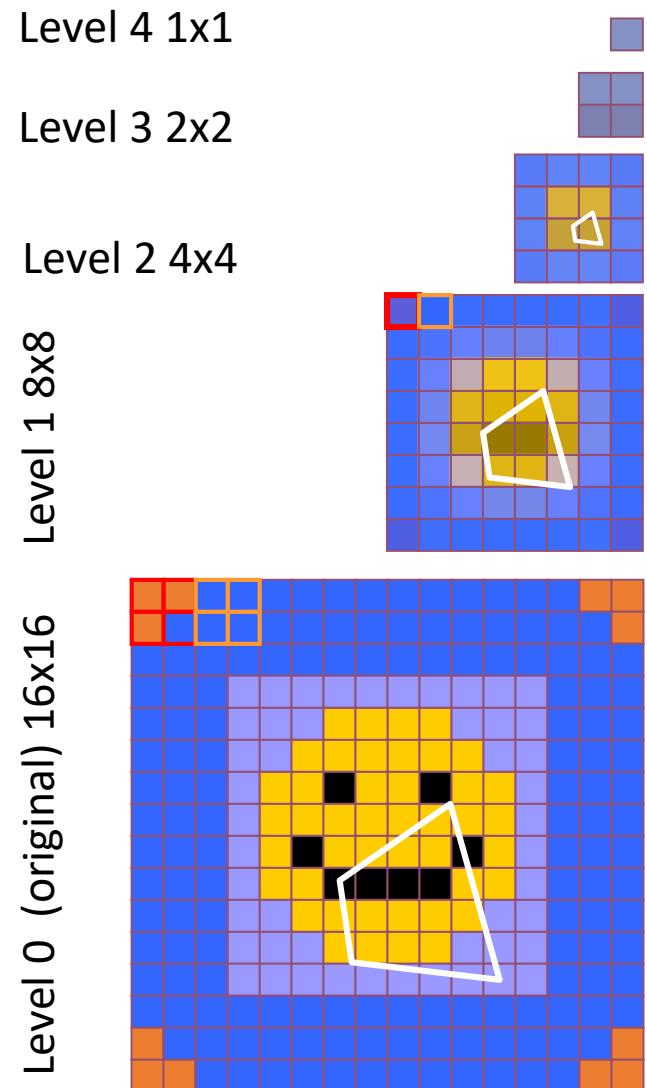
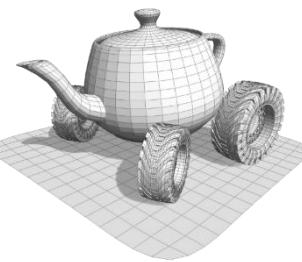
- With Nearest Filtering on minification, we are just sampling one texel there and there
- Bilinear filter does not solve the problem
- How many texels should we interpolate? All those covered by the pixel

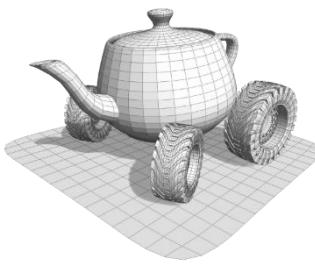


texture

MipMapping

- **Multum In Parvo (MIP)** mapping consists in precomputing a set of $\log_2 n$ versions of the original texture image at halved resolutions.
- When accessing the texture, the size of the pixel *in texels* is evaluated and the *proper level* is used
- The *proper level* means the level where one pixel covers one texel
 - How is it found?...next

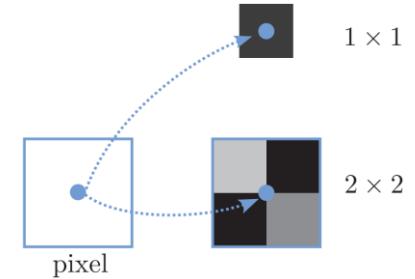




Mip-Mapping: choosing the proper level

- Take the gradient of the texture coordinates at pixel along x and y

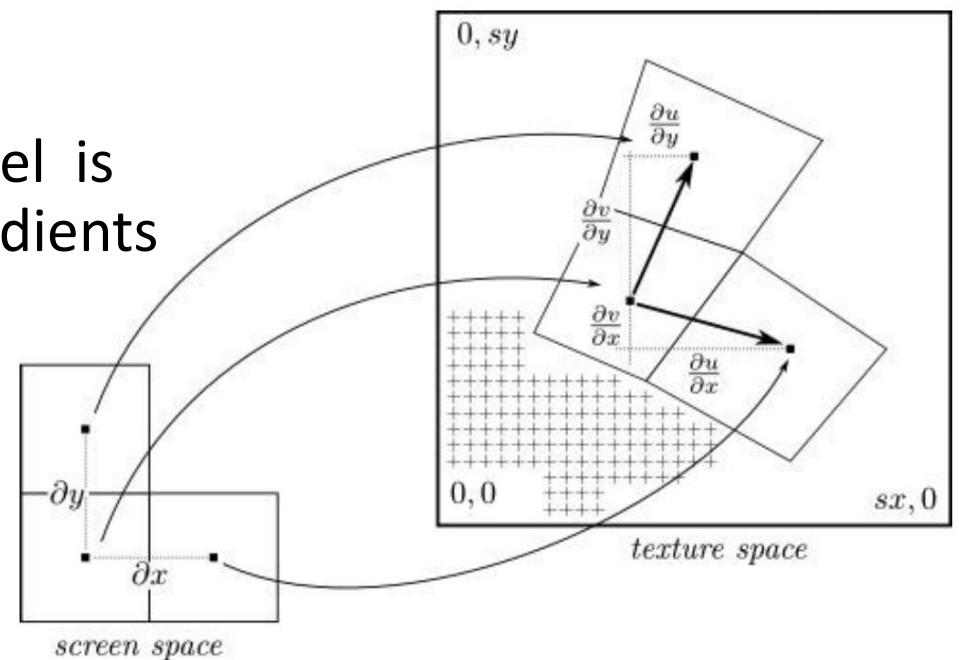
$$\begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial x} \end{pmatrix} = \begin{pmatrix} \frac{u(x+\Delta x, y) - u(x, y)}{\Delta x} \\ \frac{v(x+\Delta x, y) - v(x, y)}{\Delta x} \end{pmatrix}$$

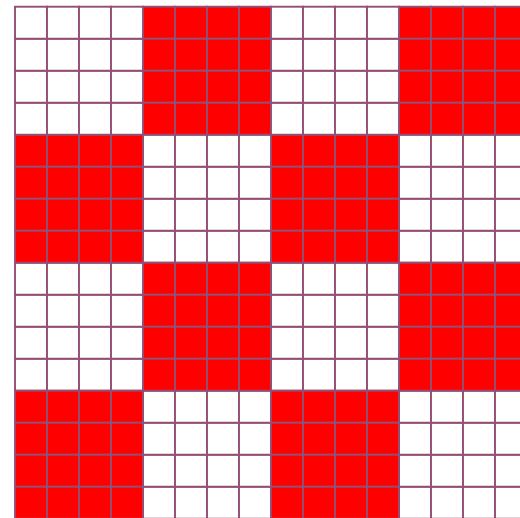
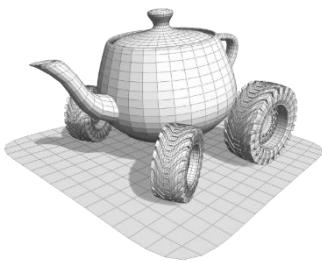


- The size of the area of texels covered by the pixel is taken as the maximal magnitude of the two gradients

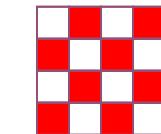
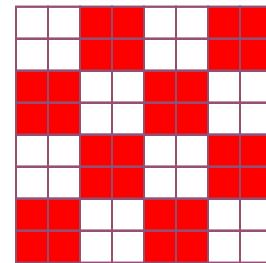
$$\rho = \max \left(\left\| \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial x} \end{bmatrix} \right\|, \left\| \begin{bmatrix} \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial y} \end{bmatrix} \right\| \right)$$

- And the mip-map level as $l = \log_2 \rho$

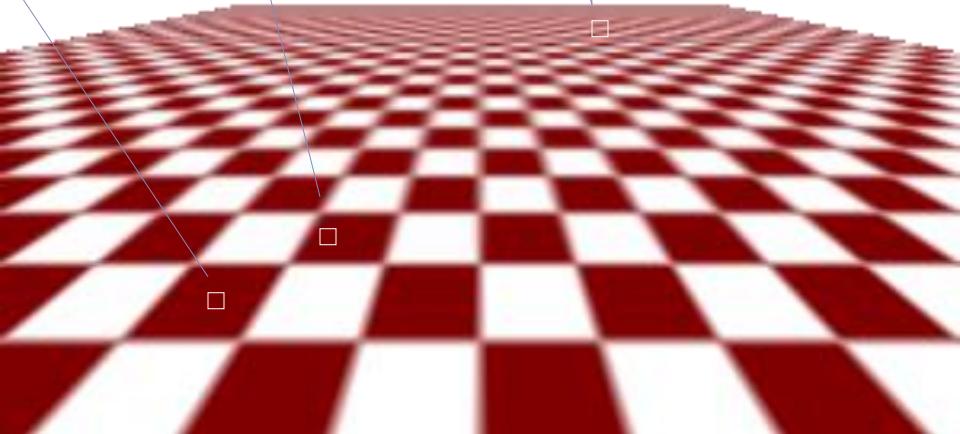
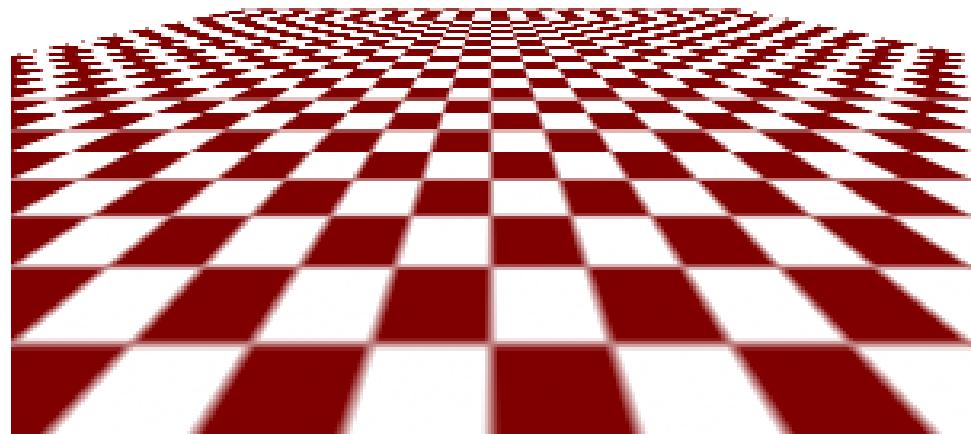


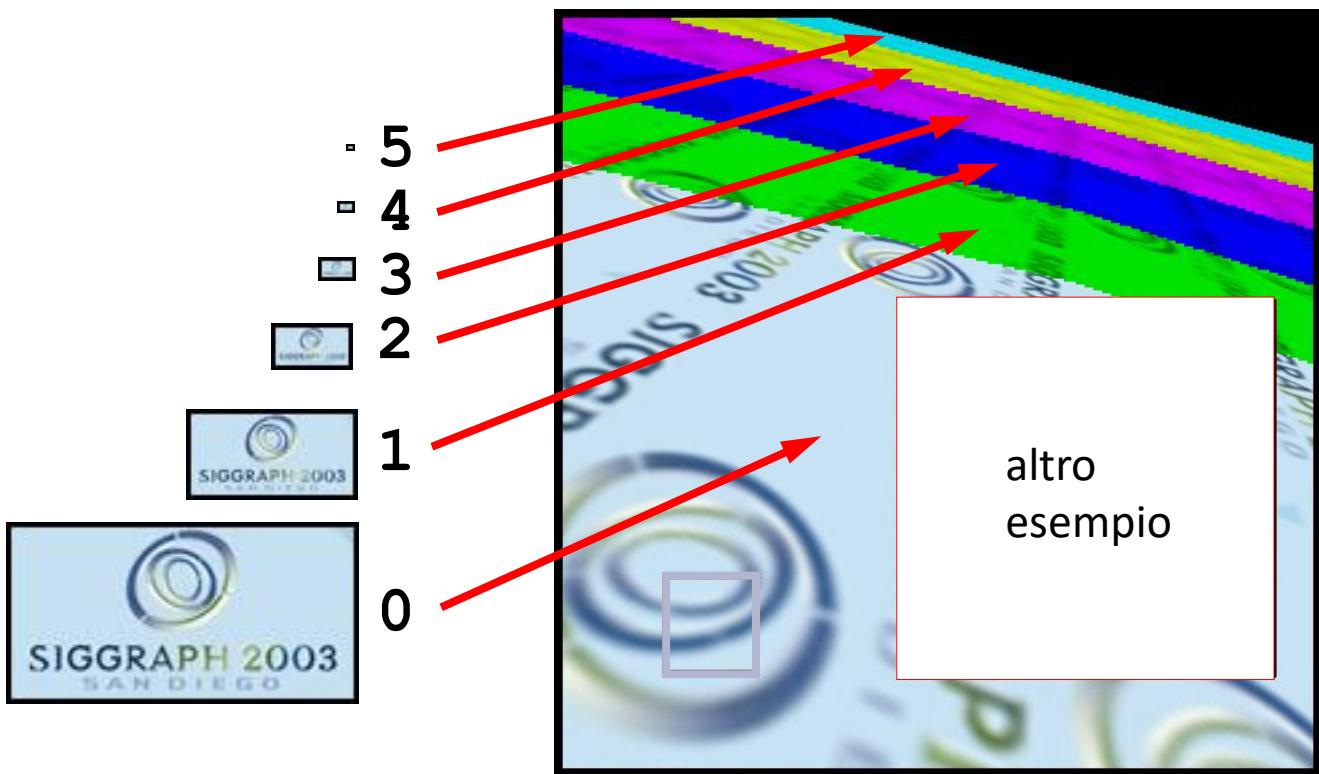
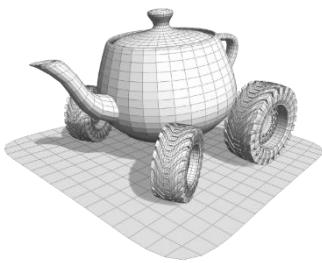


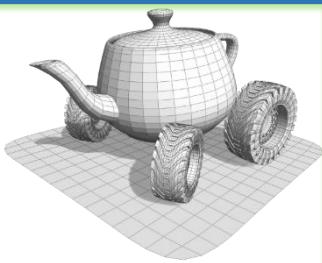
Bilinear interpolation



MIP-mapping

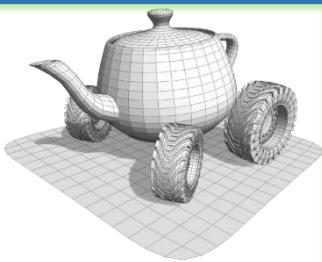






Using textures with OpenGL

1. Loading image
2. Creating a texture object
 - Uploading the image to the texture
 - Setting UV wrapping mode
 - Setting filtering mode
 - Building mipmap
3. Adding UV coordinates to vertices
4. Reading texture (FS)

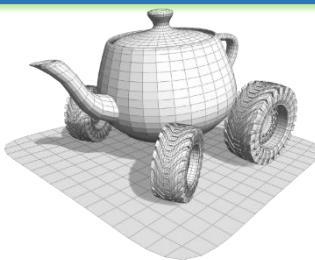


Using textures with OpenGL

1. Loading image

- Many C++ libraries provide image loading
- `stb_image` is a simple header-only library the us will use:
<https://github.com/nothings/stb>

```
unsigned char * data;
int x_size, y_size;
int n_components;
data = stbi_load(filename, &x_size,
&y_size, &n_components, 0);
```

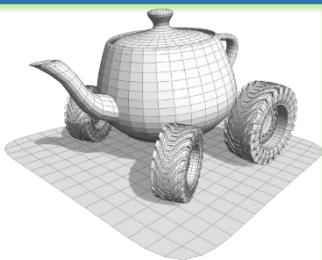


Using textures with OpenGL

1. Loading image
2. Creating a texture
 - Uploading the image to the texture
 - Setting UV wrapping mode
 - Setting filtering mode
 - Building mipmap

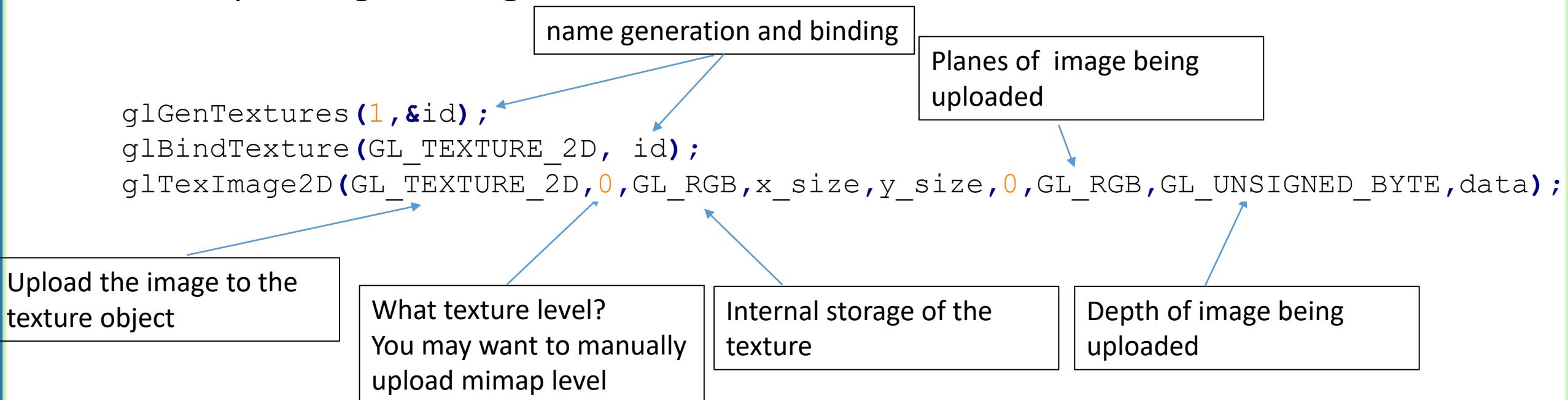
In any order

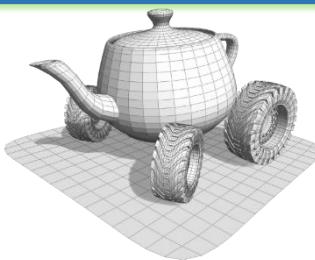
```
glGenTextures(1, &id);
 glBindTexture(GL_TEXTURE_2D, id);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, x_size, y_size, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
 GL_LINEAR);
 glGenerateMipmap(GL_TEXTURE_2D);
```



Using textures with OpenGL

1. Loading image
2. Creating a texture
 - Uploading the image to the texture





Using textures with OpenGL

1. Loading image
2. Creating a texture
 - Uploading the image to the texture
 - Setting UV wrapping mode

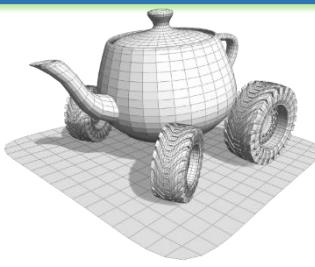
wrapping is specified for each coordinate

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Other choices

GL_MIRRORED_REPEAT
GL_CLAMP_TO_EDGE
GL_CLAMP_TO_BORDER

Target: the texture currently bound to GL_TEXTURE_2D

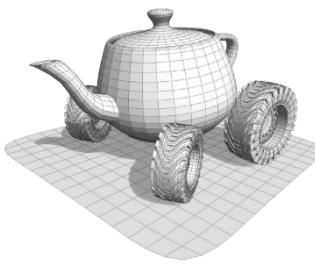


Using textures with OpenGL

1. Loading image
2. Creating a texture
 - Uploading the image to the texture
 - Setting UV wrapping mode
 - Setting filtering mode
 - Both for minification and magnification

how to sample the mipmap level(s)

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glGenerateMipmap(GL_TEXTURE_2D);
```



Using textures with OpenGL

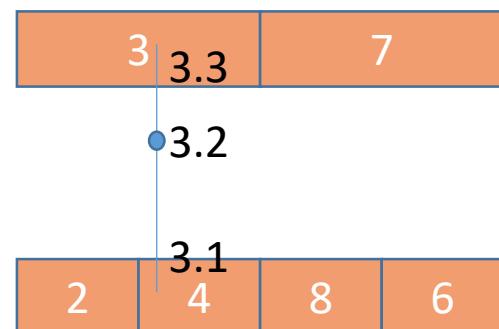


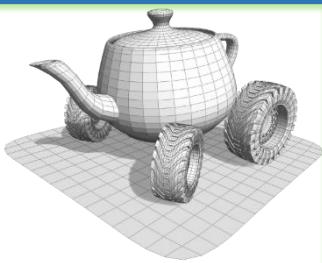
`GL_NEAREST_MIPMAP_NEAREST`: nearest level ($i+1$) and nearest texel ($v = 3$)

`GL_LINEAR_MIPMAP_NEAREST` :nearest level ($i+1$) and linear interpolation texel ($v = \text{int}(3,7)$)

`GL_NEAREST_MIPMAP_LINEAR`: nearest texel ($v=3, v=4$) in each mipmap, interpolation between results

`GL_LINEAR_MIPMAP_LINEAR`: interpolation on each mipmap and interpolation between results



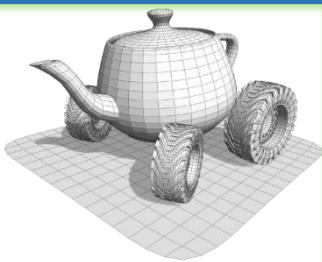


Using textures with OpenGL

1. Loading image
2. Creating a texture object
 - Uploading the image to the texture
 - Setting UV wrapping mode
 - Setting filtering mode
 - Building mipmap

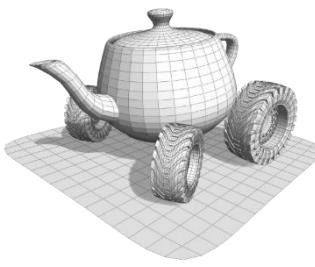
Single call, OpenGL builds the all mipmaps for us.
Recall: image sizes must be power of 2

`glGenerateMipmap(GL_TEXTURE_2D);`

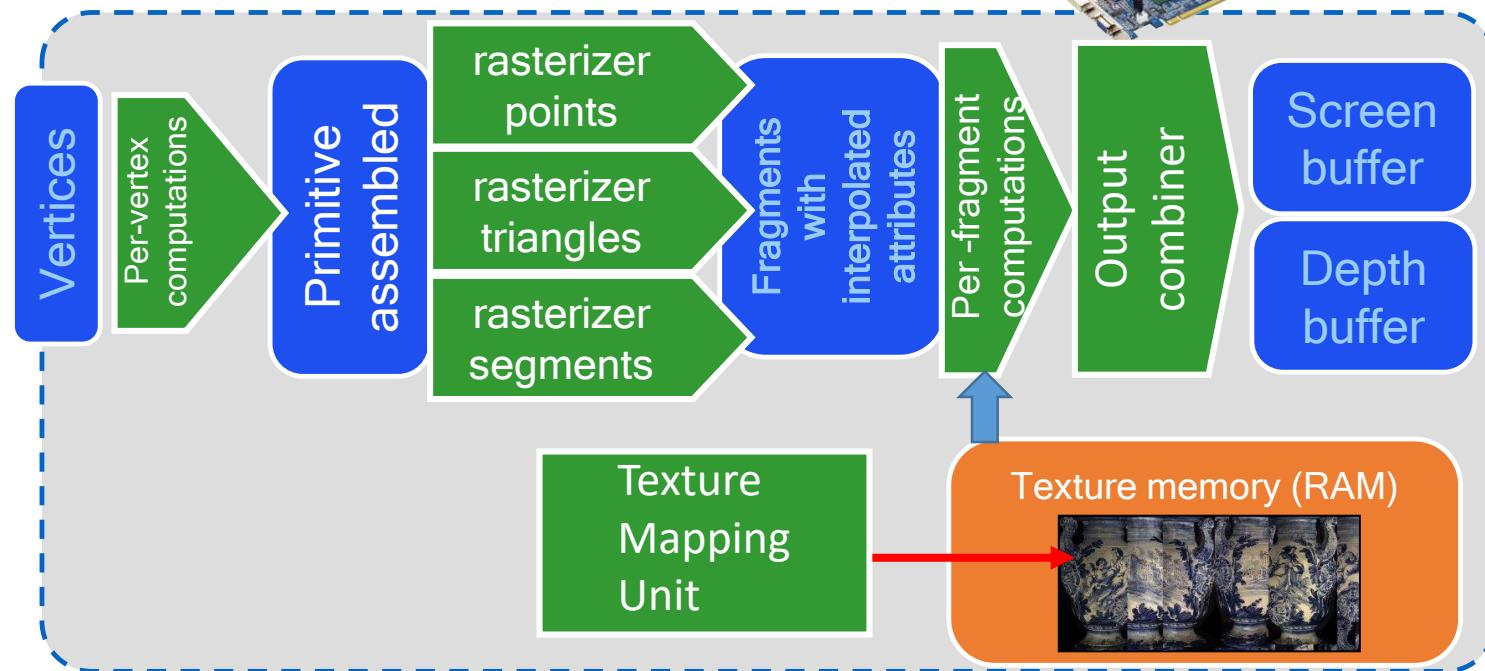


Using textures with WebGL

1. Loading image
2. Creating a texture object
 - Uploading the image to the texture
 - Setting UV wrapping mode
 - Setting filtering mode
 - Building mipmap
3. Adding UV coordinates to vertices

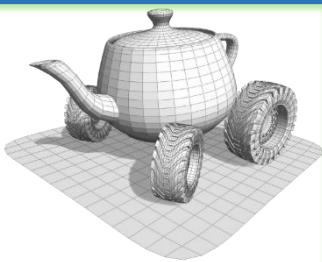


Texture (Mapping) Unit



Reading from texture memory is referred to as **Texture Look-Up**

Texture Units are a physical piece of hardware on the GPU devoted to accessing and filtering texture data
Their number is currently between 4 and 32, we can access with
`nTU = glGetParameter(GL_MAX_TEXTURE_IMAGE_UNITS);`



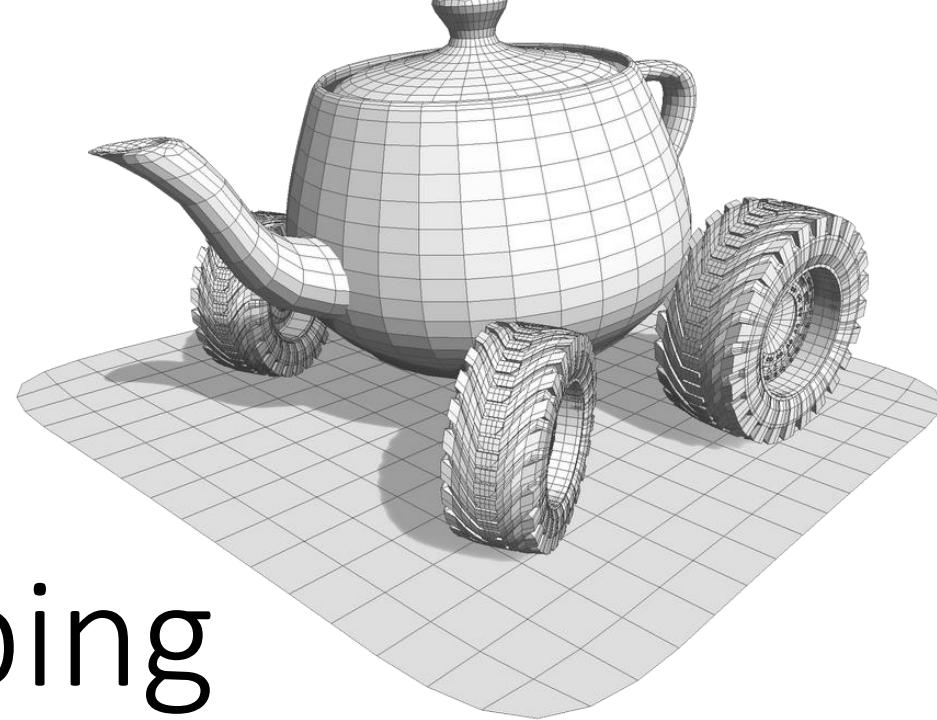
Using textures with OpenGL

1. Loading image
2. Creating a texture object
 - Uploading the image to the texture
 - Setting UV wrapping mode
 - Setting filtering mode
 - Building mipmap
3. Adding UV coordinates to vertices
4. Reading texture (Fragment Shader)

sampler2D represents a texture 2D in the shader program

```
uniform sampler2D uSampler;  
//...  
int main()  
//...  
color =  
texture2D(uSampler, vTexCoords);  
}
```

texture2D is a GLSL function to access a specified texture at a specified texture coordinates



Texture Mapping (CNT)



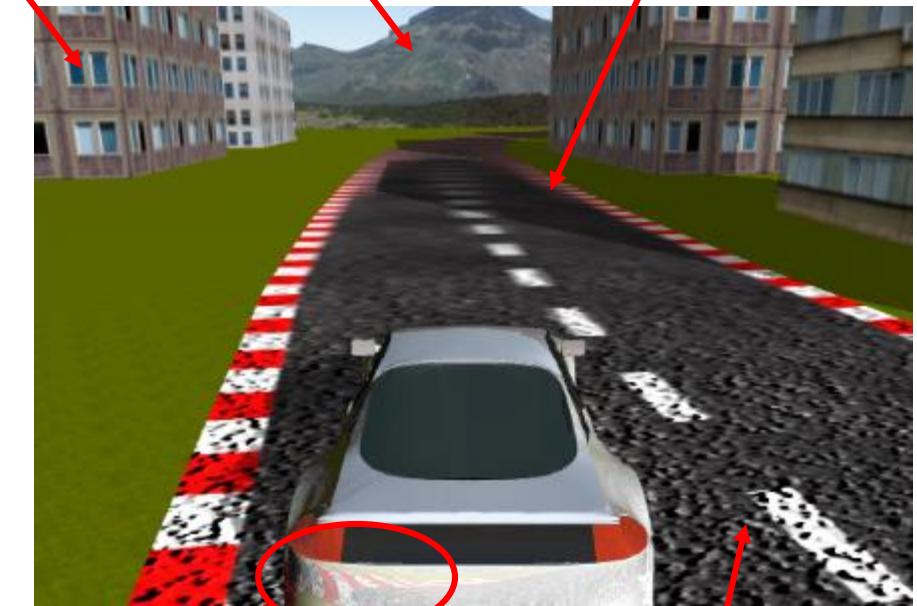
Beyond Color

- The most basic use of Texture Mapping is to provide a color to the surface
 - In lighting equation terms, it's *diffuse* material
- However, we can map all sort of data so we can use them for shading the scene
- Texture Mapping is behind most of the *tricks* to make a scene look more realistic without using raytracing

Color Mapping for color

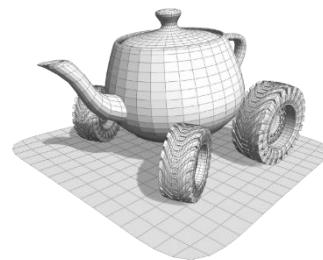
Environment map for the horizon

Shadow Mapping for shadows



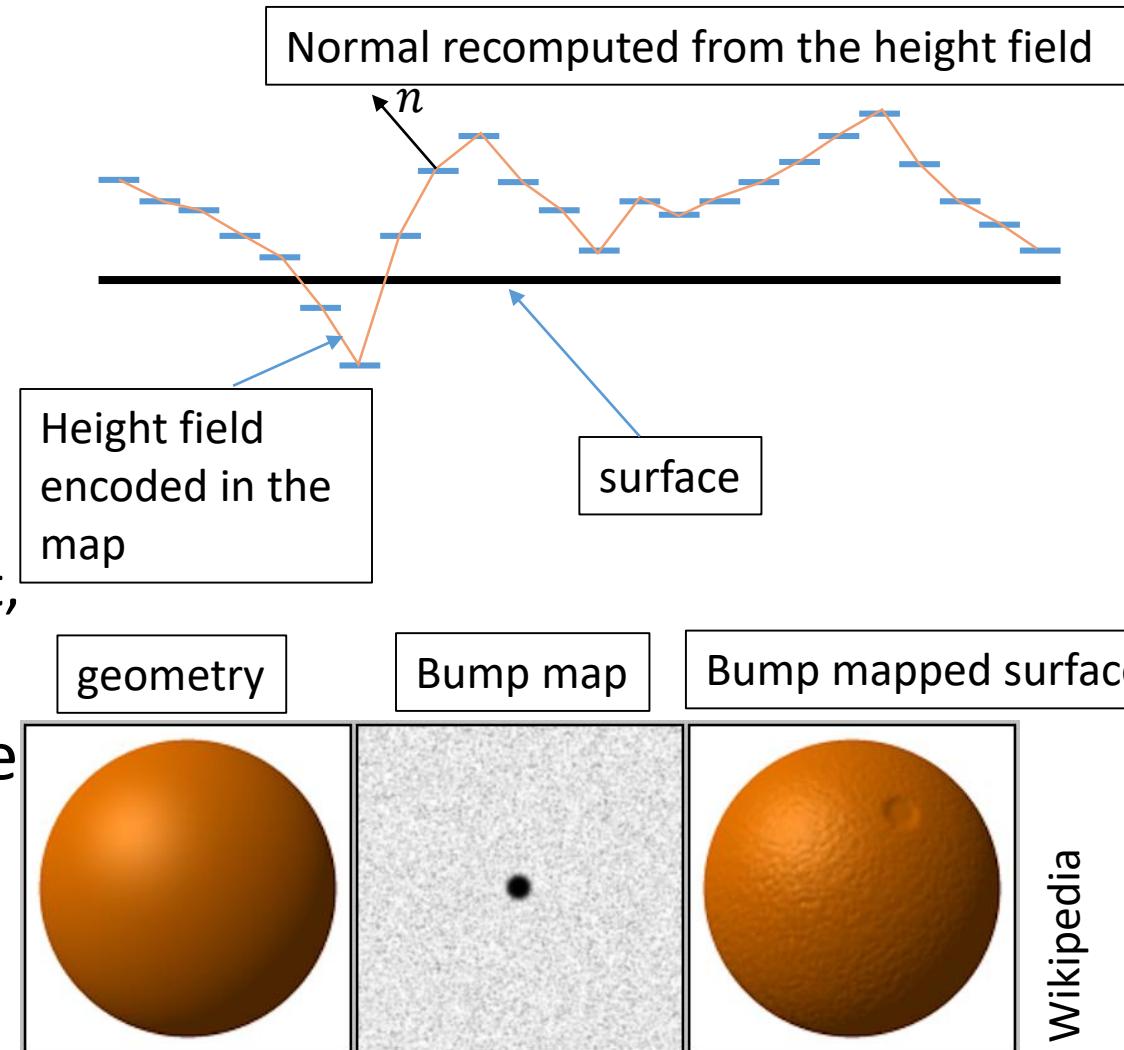
Cube Map for reflections

Normal Mapping for "rugosity"

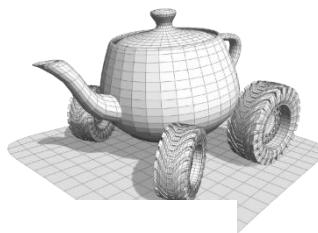


Faking Geometric Details: Bump Mapping

- **Bump Mapping** consists of encoding a *height field value* in the map
- **Bump Map:** a single channel texture where each texel encodes the height from the surface
 - Values from 0.0 to 1.0 maps to min_height, max_height user defined range
- The height field is only used to compute the normal to use for lighting, the surface itself is unchanged

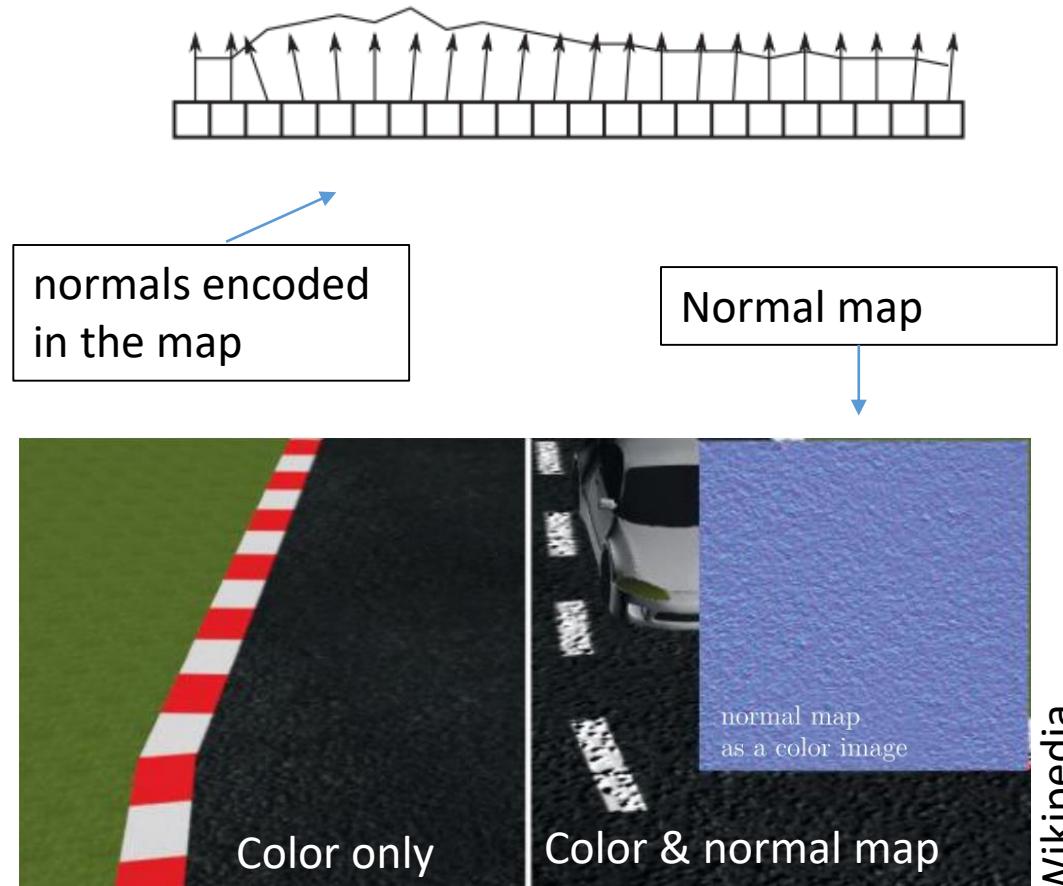


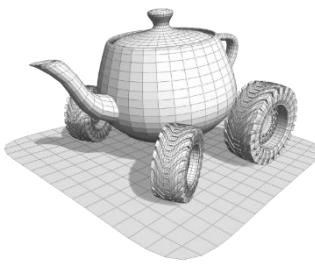
Wikipedia



Faking Geometric Details: Normal Mapping

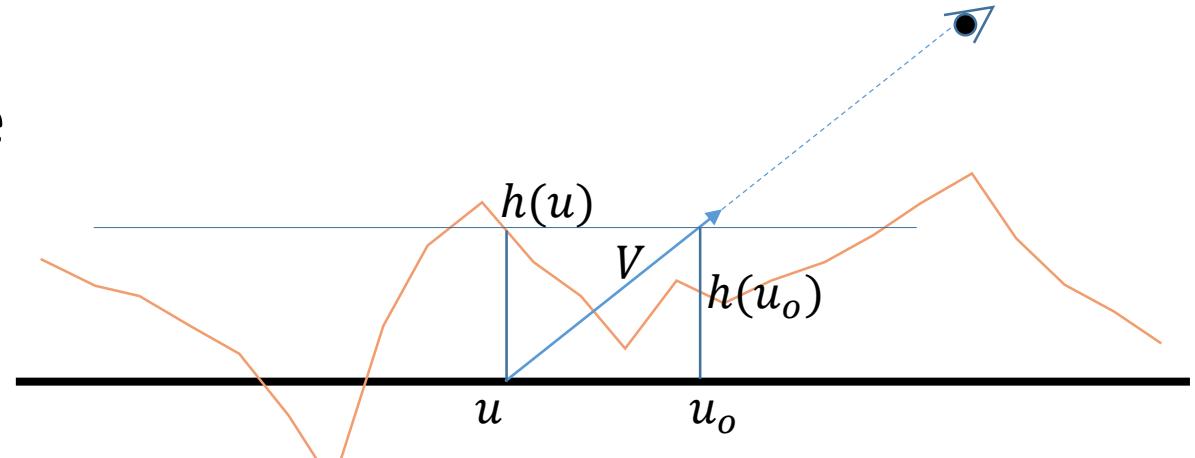
- **Normal Mapping** consists of encoding a *normal variation value* in the map
- **Normal Map:** a 3 channels texture where each texel encodes a vector to add to the “fragment normal”
 - More commonly it replaces it
- The difference with the original bump mapping is that we give directly the normal variation
 - More control on the final result
 - The original interpolated normal is still used





Faking Geometric Details: Parallax Mapping

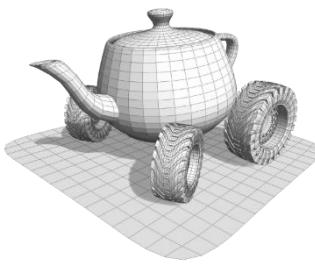
- **Parallax Mapping** improves on Normal Mapping by emulating the *parallax effect* when the point of view changes
- The map encodes the height field as for Bump Mapping
- The texture coordinates u is offsetted on the base of the height value $h(u)$ written in the map



u : point hit in texture space

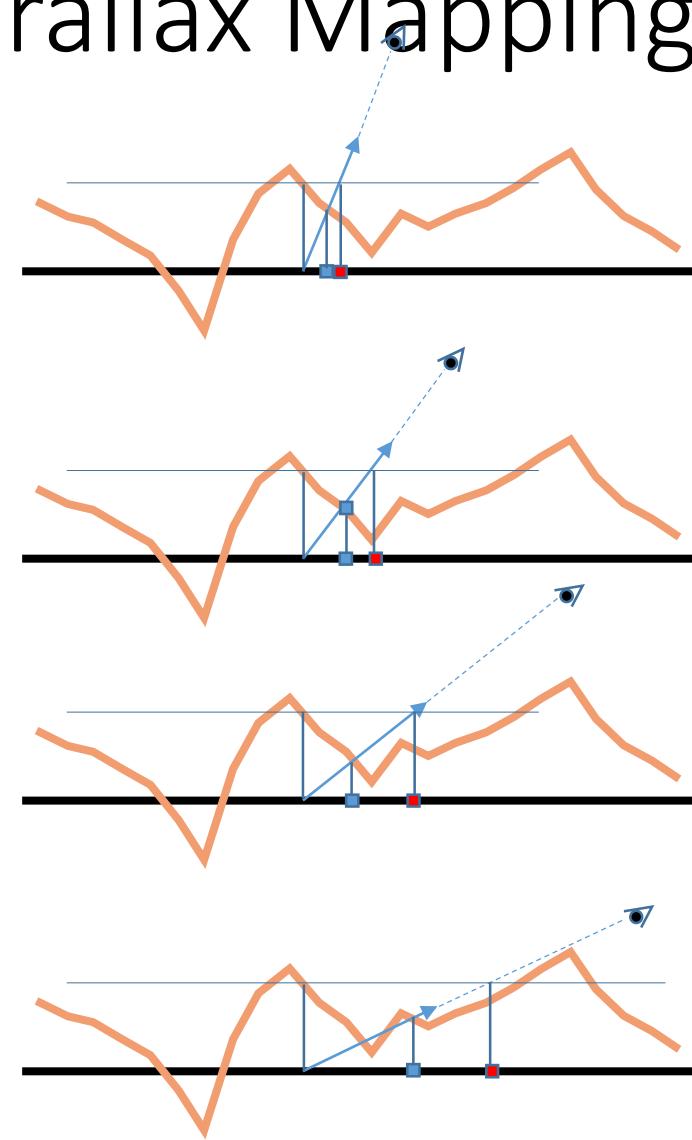
$h(u)$: height value

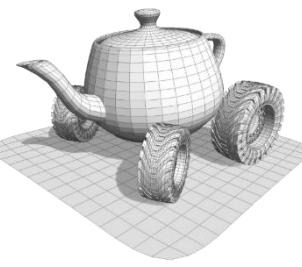
$u_0 = u + \frac{h(u)}{V_z} V_x$ projection on texture space of the point intersected by the view ray on the plane with height $h(u)$



Faking Geometric Details: Parallax Mapping

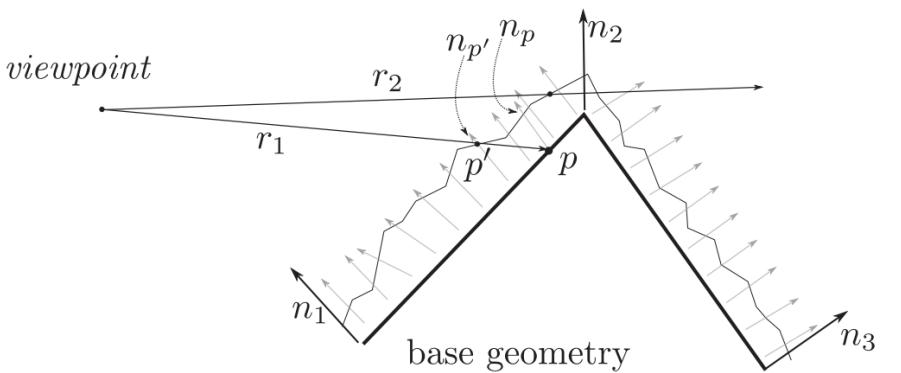
- Parallax Mapping works:
 - in the assumption that height field is not too *steep*, that is, that it does not change too quickly
 - Not for grazing angles, that is, view directions too far away from the normal
- Improvements on parallax mapping
 - Using the slope at point u to a better estimation of the offset
 - Finding the actual intersection between the ray and the height field (a.k.a. **relief mapping**)

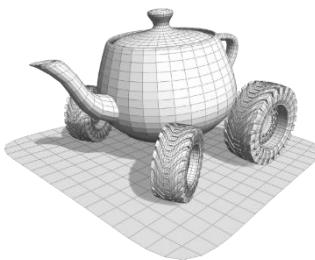




Bump Mapping

- Widely used in animated movies and videogames for “showing” detailed geometry where there isn’t
- Need to pay attention to the viewing angle and the range of height values to use
- Grazing angles are a giveaway

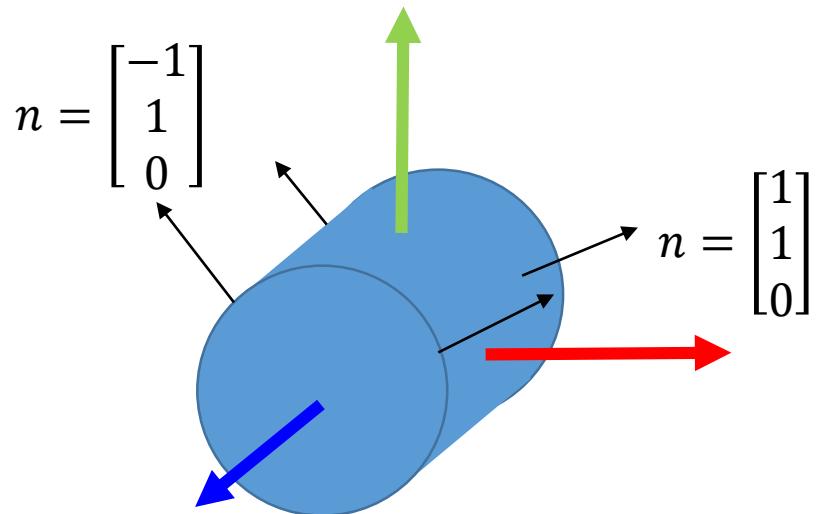




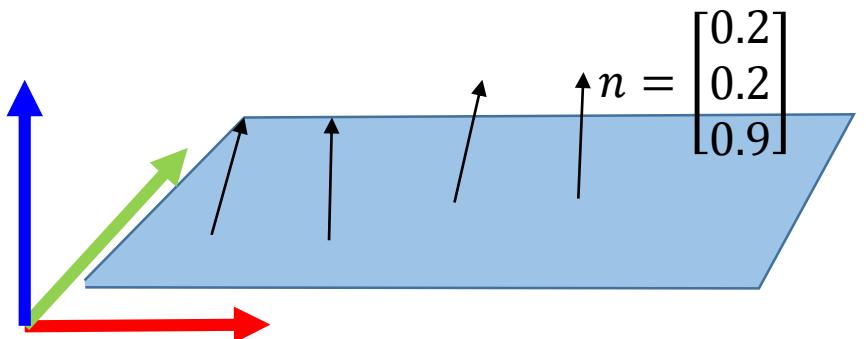
Tangent Space

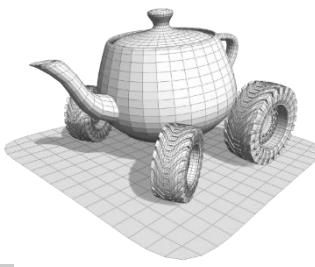
- 3D entities such as normals can be defined in **object space** (just like vertex normals)
- In that case the texture will only work for that specific mapping to geometry
 - Because we write the normal on the basis of *where* it will be used

Normals
defined in
object space



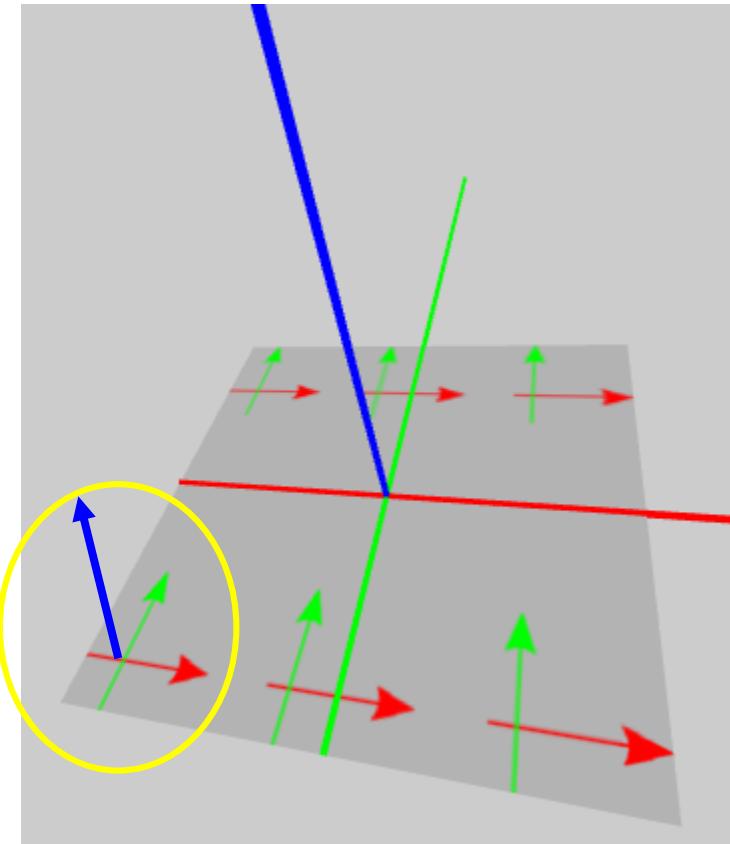
Normals
defined in
texture space

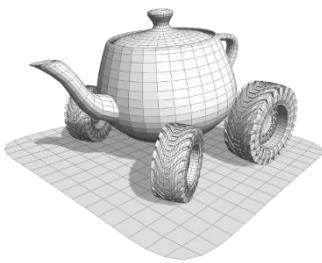




Tangent Space

- The **Tangent Space** is yet another 3D space where the XY plane is on the plane tangent to the object surface and the Z axis is oriented like the normal to the surface
 - The X axis (called **tangent**) is oriented so that its mapping in texture space is [1,0]
 - The Y axis (called **bitangent**) is oriented so that its mapping in texture space is [0,1]
- The tangent space is locally expressed with a **tangent frame**



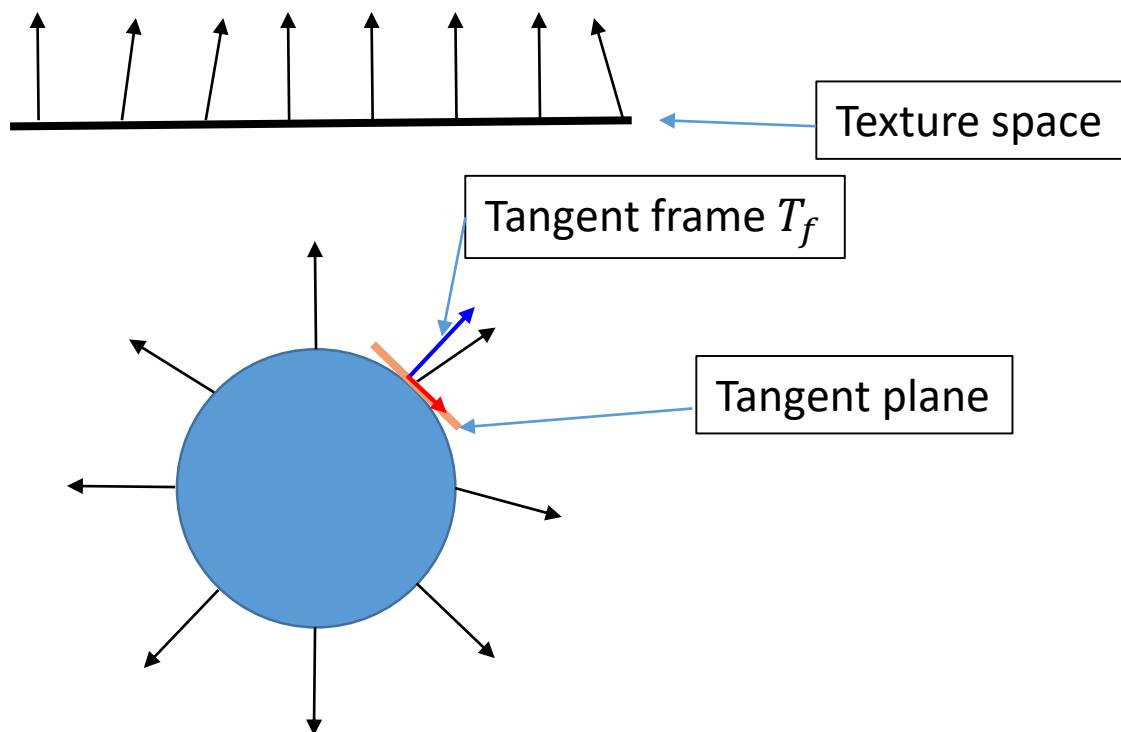


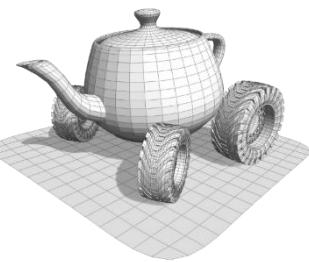
Tangent Frames

- The **Tangent Frame** maps the coordinates from *texture space* to *object space* in a *neighborhood of its origin*

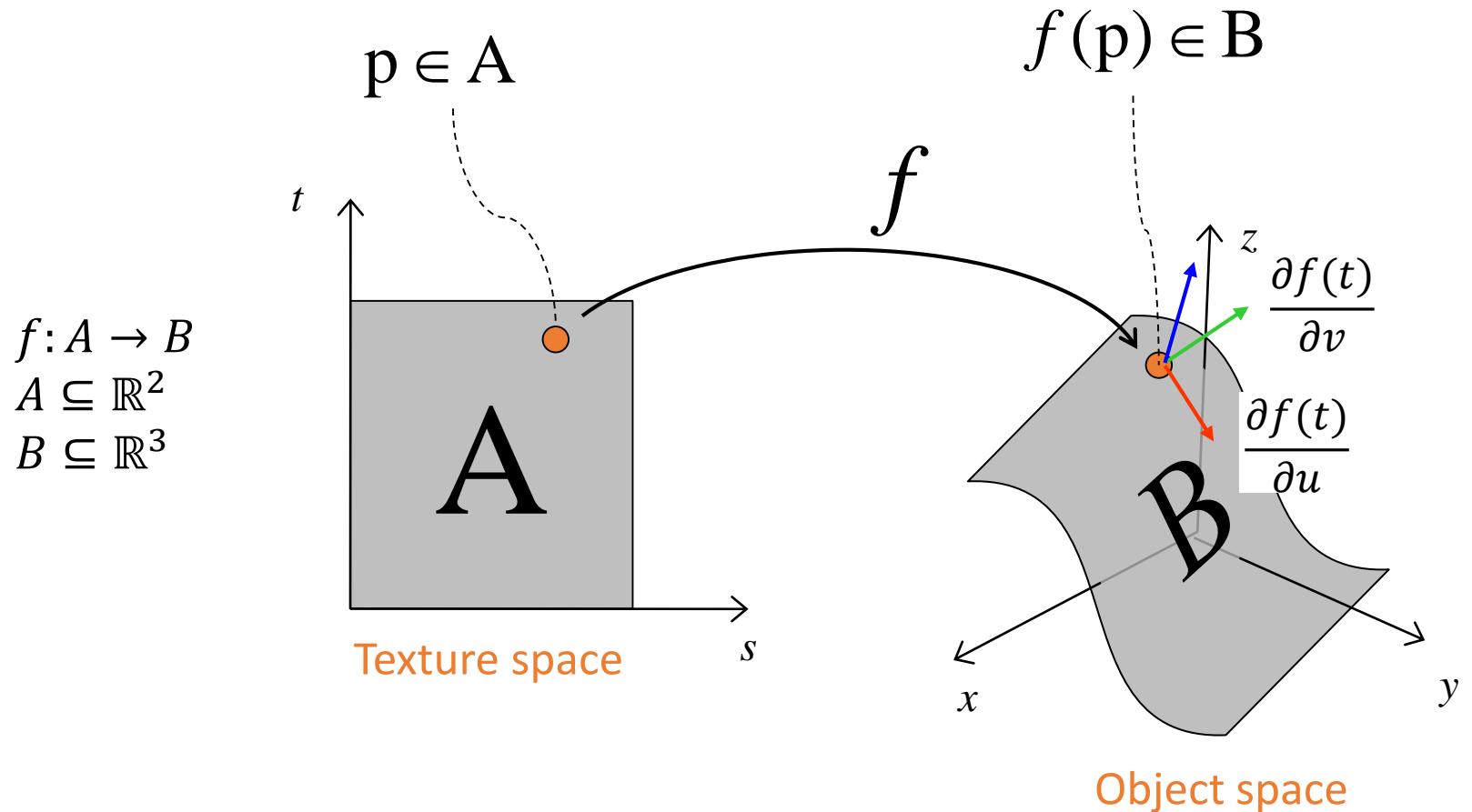
$$\mathbf{p}_{os} = T_f \mathbf{p}_{ts}$$

- How do we compute the tangent frames? (next..)





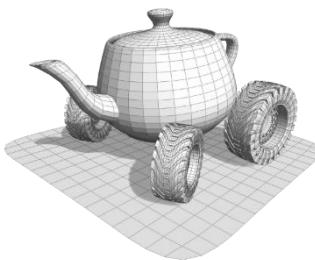
Tangent frames for Parametric Surfaces



$$y(t) = \frac{\partial f(t)}{\partial v}$$

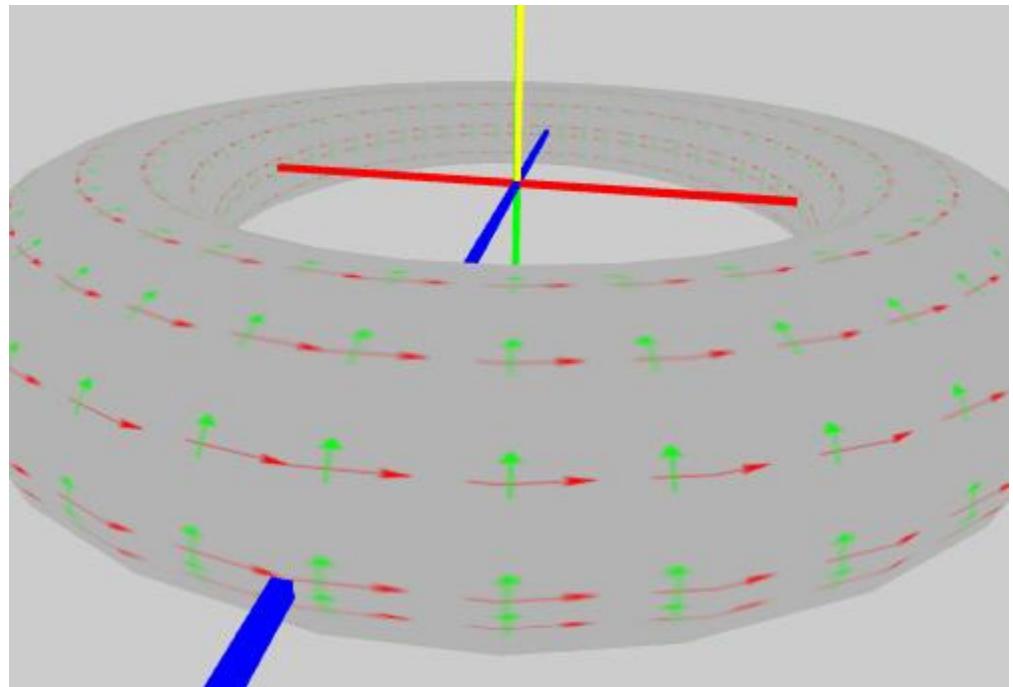
$$x(t) = \frac{\partial f(t)}{\partial u}$$

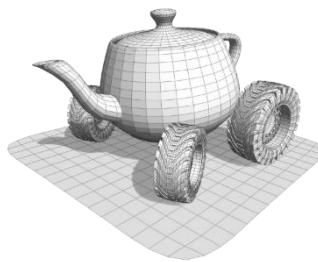
$$\mathbf{n}(t) = \mathbf{x}(t) \times \mathbf{y}(t)$$



Tangent Frames for Polygon Meshes

- UV-coordinates of the mesh are a sampling of f^{-1} (on the vertices)
- We need to find the mapping of the axes u and v from texture to object space. Together with the normal n , they are the tangent frame





Tangent Frames for Polygon Meshes

Let us consider frames:

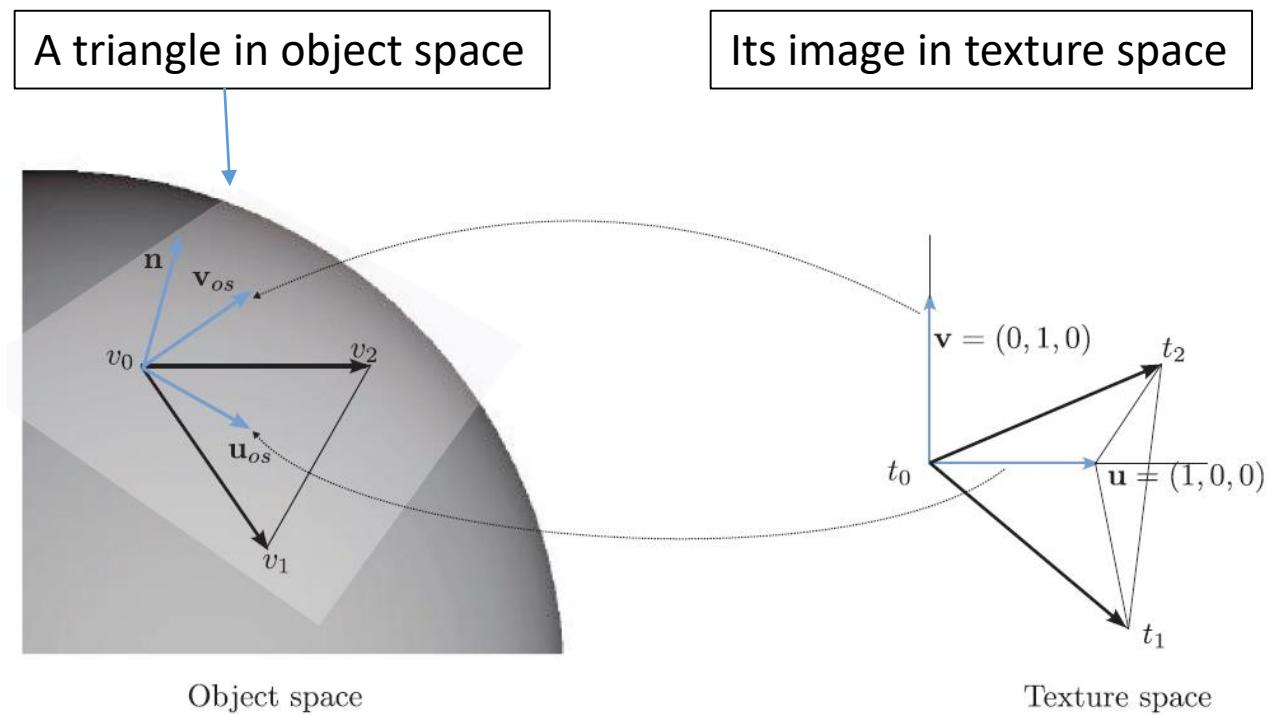
$$F_o = \{\mathbf{v}_1 - \mathbf{v}_0, \mathbf{v}_2 - \mathbf{v}_0, \mathbf{v}_0\} \text{ in object space}$$

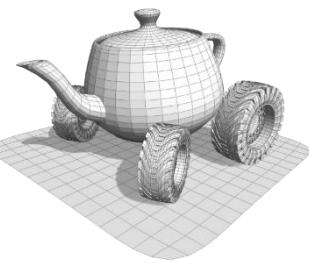
$$F_t = \{\mathbf{t}_1 - \mathbf{t}_0, \mathbf{t}_2 - \mathbf{t}_0, \mathbf{t}_0\} \text{ in texture space}$$

Let \mathbf{u}_{os} be the mapping of $\mathbf{u} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ in object space. Then \mathbf{u}_{os} and \mathbf{u} have the same coordinates in their respective frames.

Algorithm:

1. Find the coordinates of \mathbf{u} in F_t , $\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$
2. Find $\mathbf{u}_{os} = u_1(\mathbf{v}_1 - \mathbf{v}_0) + u_2(\mathbf{v}_2 - \mathbf{v}_0)$



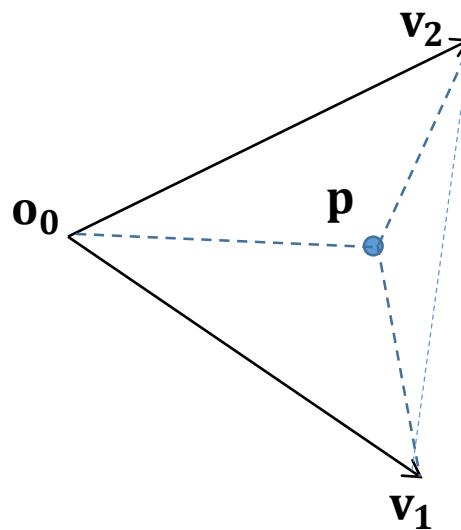


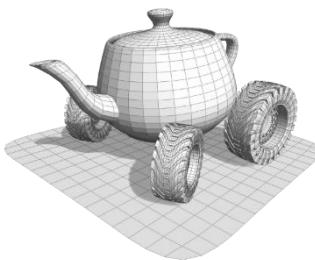
Insight

- How to find the coordinates of a point in a non orthogonal frame?

1. Find the barycentric coordinates for \mathbf{p} in the triangle $(\mathbf{o}, \mathbf{o} + \mathbf{v}_1, \mathbf{o} + \mathbf{v}_2)$
2. $\mathbf{p} = (1 - \omega_1 - \omega_2) \mathbf{o} + \omega_1(\mathbf{o} + \mathbf{v}_1) + \omega_2(\mathbf{o} + \mathbf{v}_2)$

$$\mathbf{p} = \mathbf{o} + \omega_1 \mathbf{v}_1 + \omega_2 \mathbf{v}_2$$





Tangent Frames for Polygon Meshes

$$u_1 = \frac{((t_0 + u) - t_0) \times t_{20}}{t_{10} \times t_{20}} = \frac{u \times t_{20}}{t_{10} \times t_{20}} = \frac{[1, 0]^T \times t_{20}}{t_{10} \times t_{20}} = \frac{t_{20}v}{t_{10} \times t_{20}}$$

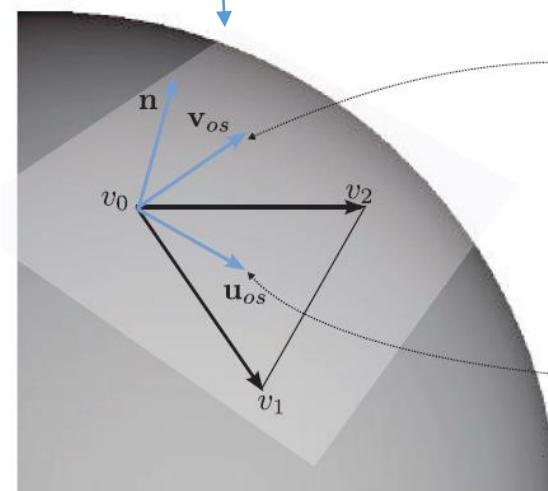
$$u_2 = \frac{t_{10} \times ((t_0 + u) - t_0)}{t_{10} \times t_{20}} = \frac{t_{10} \times u}{t_{10} \times t_{20}} = \frac{t_{10} \times [1, 0]^T}{t_{10} \times t_{20}} = \frac{-t_{10}u}{t_{10} \times t_{20}}$$

$$\mathbf{u}_{os} = \text{Normalize}(u_1(\mathbf{v}_1 - \mathbf{v}_0) + u_2(\mathbf{v}_2 - \mathbf{v}_0))$$

We can do the same for \mathbf{v}_{os} or just use the normal:

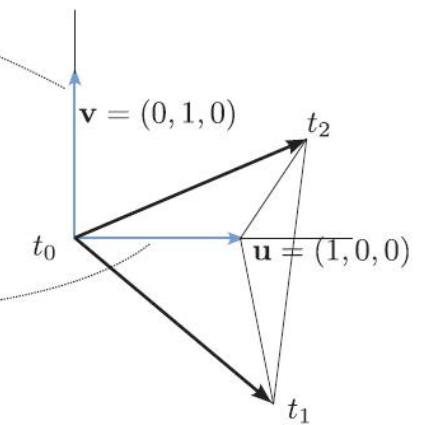
$$\mathbf{v}_{os} = \mathbf{n} \times \mathbf{u}_{os}$$

A triangle in object space

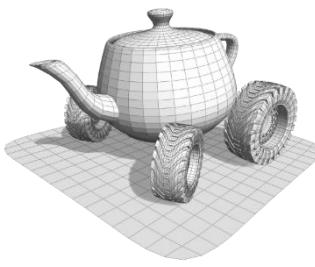


Object space

Its image in texture space

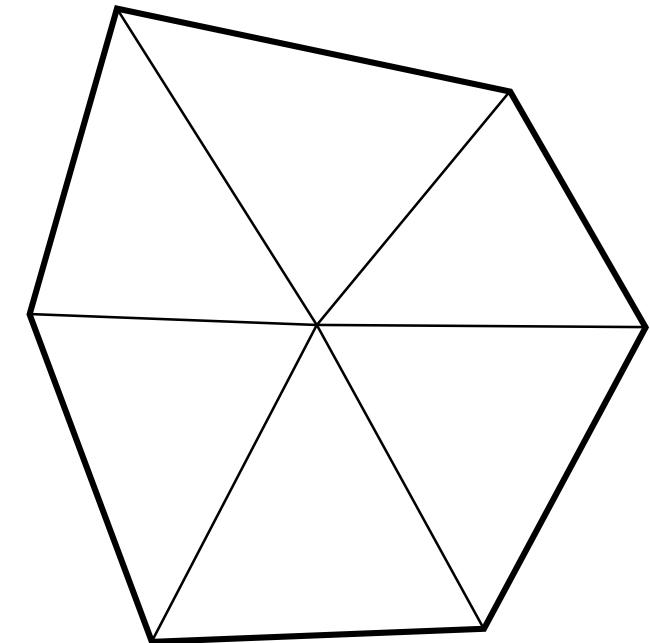


Texture space



Tangent Frames for Polygon Meshes

- Just like for per-vertex normal computation, we can compute a tangent vector for each triangle
- It will be the same only under certain conditions (*developable* meshes, see more later)
- Just like for per-vertex normal, we compute a value for each face and average (or weight the face contribution as for the normals..)



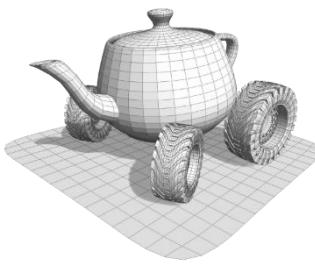


Intermezzo: finding the UV-unwrapping

- For a parametric surface, such as a torus or a Bezier patch, the UV-unwrapping is trivial

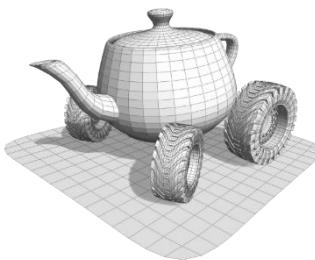
$$\begin{aligned} u(f(s, t)) &= s \\ v(f(s, t)) &= t \end{aligned}$$

- What for a generic polygon mesh? How can be it unwrapped onto texture space?
The problem is known as **mesh parametrization**



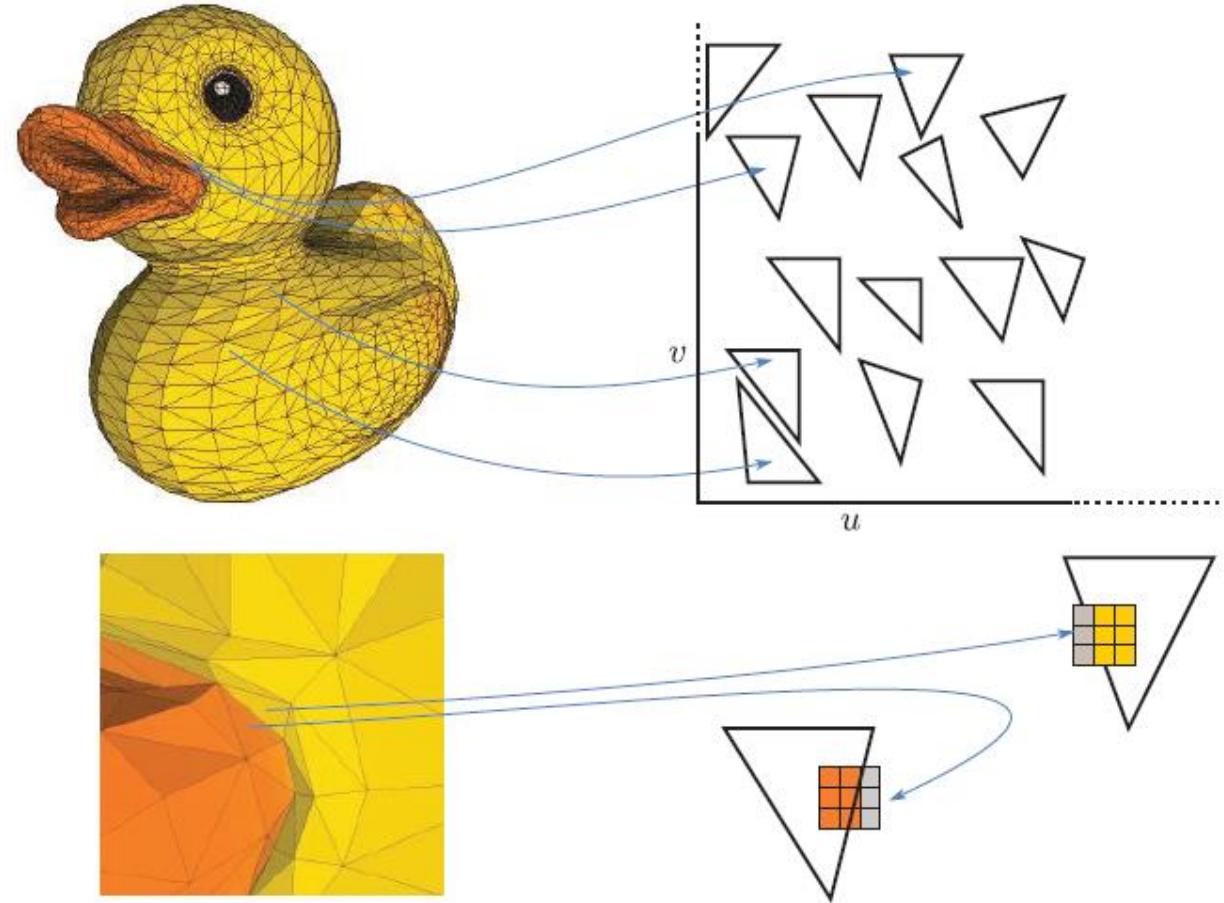
Notes on Mesh Parametrization

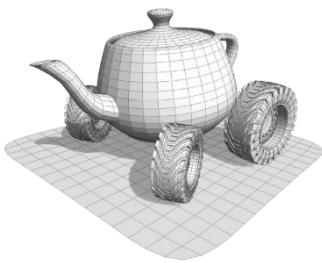
- Mesh Parametrization Problem: given a polygon mesh, find a function that maps (a portion of) $[0,1] \times [0,1]$ onto the mesh
 - Does it always exist? No
 - If it does exist, is it unique? No
 - Is there a best one? No, it depends on the metric used
 - And if the metric is given? Typically there are local minima



Notes on Mesh Parametrization

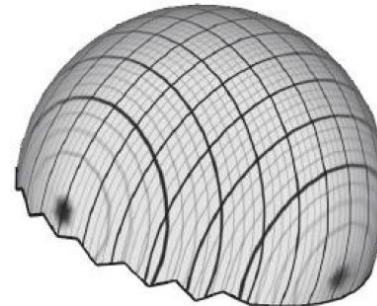
- Can we map each individual polygon on its own piece of texture space?
- Yes, but the parametrization is highly discontinuous
 - Texture filtering won't work (linear interpolation and mipmapping)
 - Impossible to paint manually





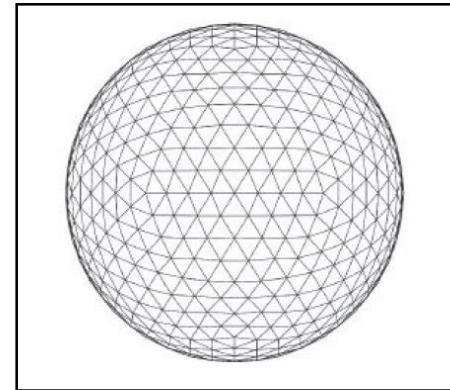
Notes on Mesh Parametrization

- Can we «flatten» the mesh in 2D altogether to obtain a continuous mapping?
- Yes (if it has borders) but the parameterization may be highly **distorted**:
 - same-size polygons may map on very different number of texels
 - Angles may not be preserved



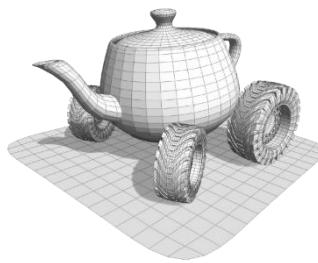
Surface in 3D

Orthogonal projection

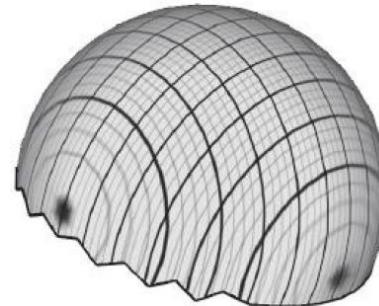


Texture Space

Notes on Mesh Parametrization

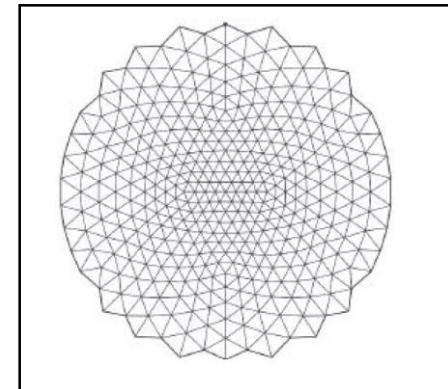


- Can we «flatten» the mesh in 2D altogether to obtain a continuous mapping?
- Yes (if it has borders) but the parameterization may be highly **distorted**:
 - same-size polygons may map on very different number of texels
 - Angles may not be preserved

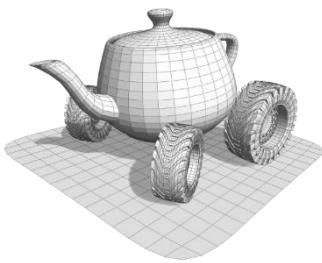


Surface in 3D

Angle preserving flattening

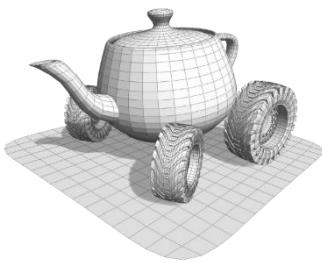


Texture Space



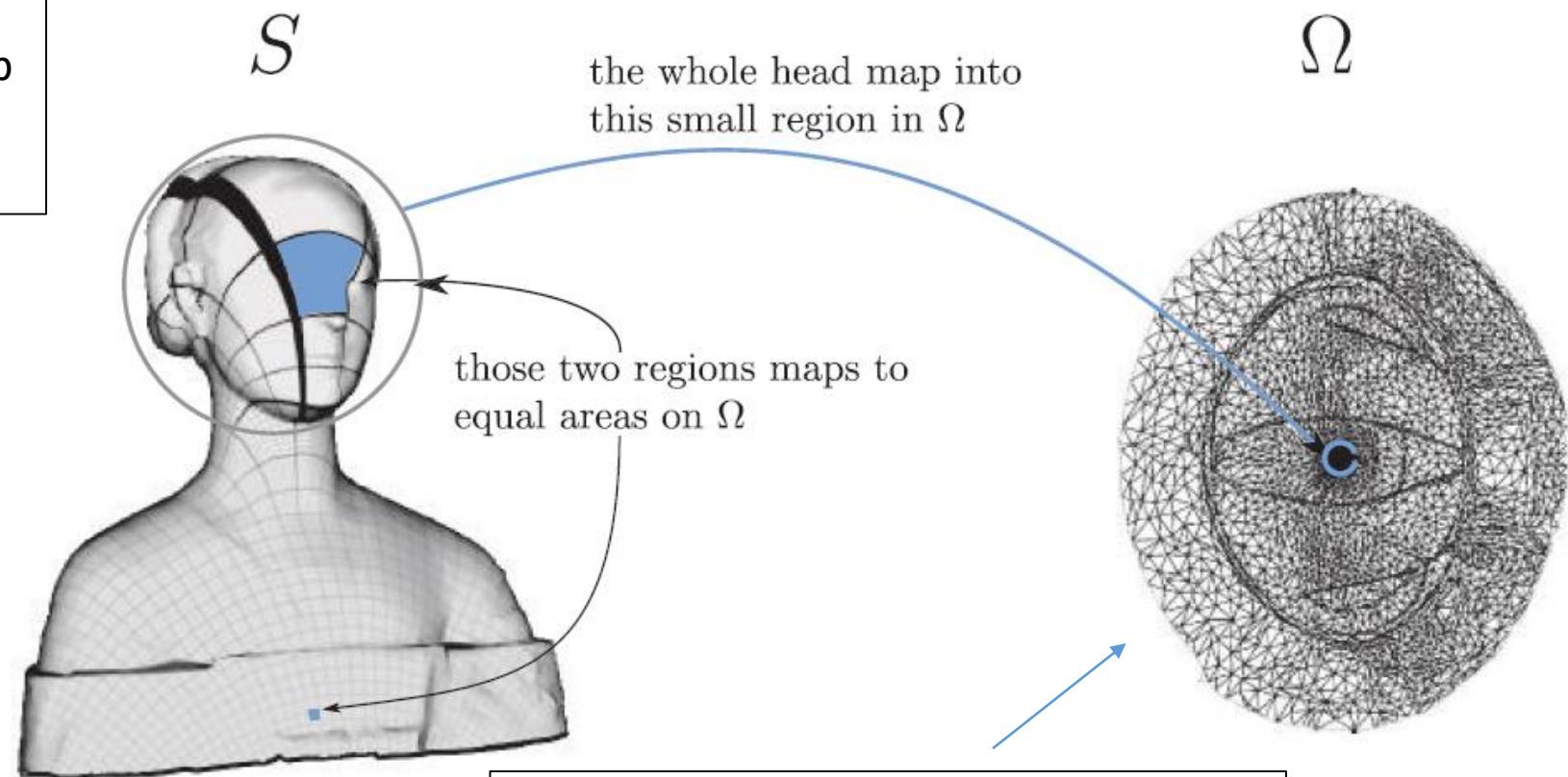
Notes on Mesh Parametrization

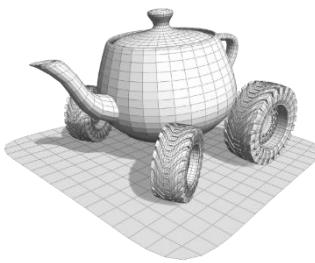
- The **quality** of a parametrization is measured with two factors:
 - How much the *area* is preserved (up to a uniform scaling factor). If a parametrization preserves the area it is called **equiareal**
 - How much the angles are preserved. If a parametrization preserves the angles it is called **conformal** or **angle preserving**
 - If a parametrization preserves both angles and area it is called **isometric**
- Does an isometric parametrization exist? Yes, all the surfaces that can be modeled by bending a sheet of paper. These surfaces are called «developable»
 - Note: the paper cannot be «stretched»



Examples of Mesh Parametrization

a texture with a regular pattern is always used to help visually assessing the distortion





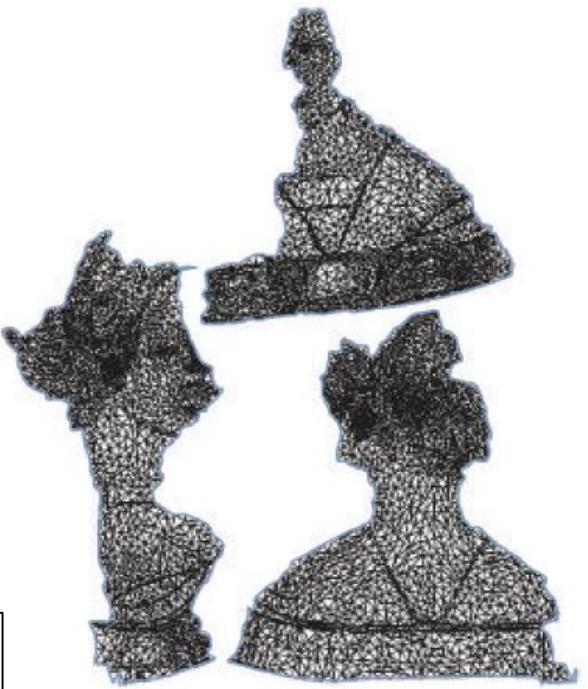
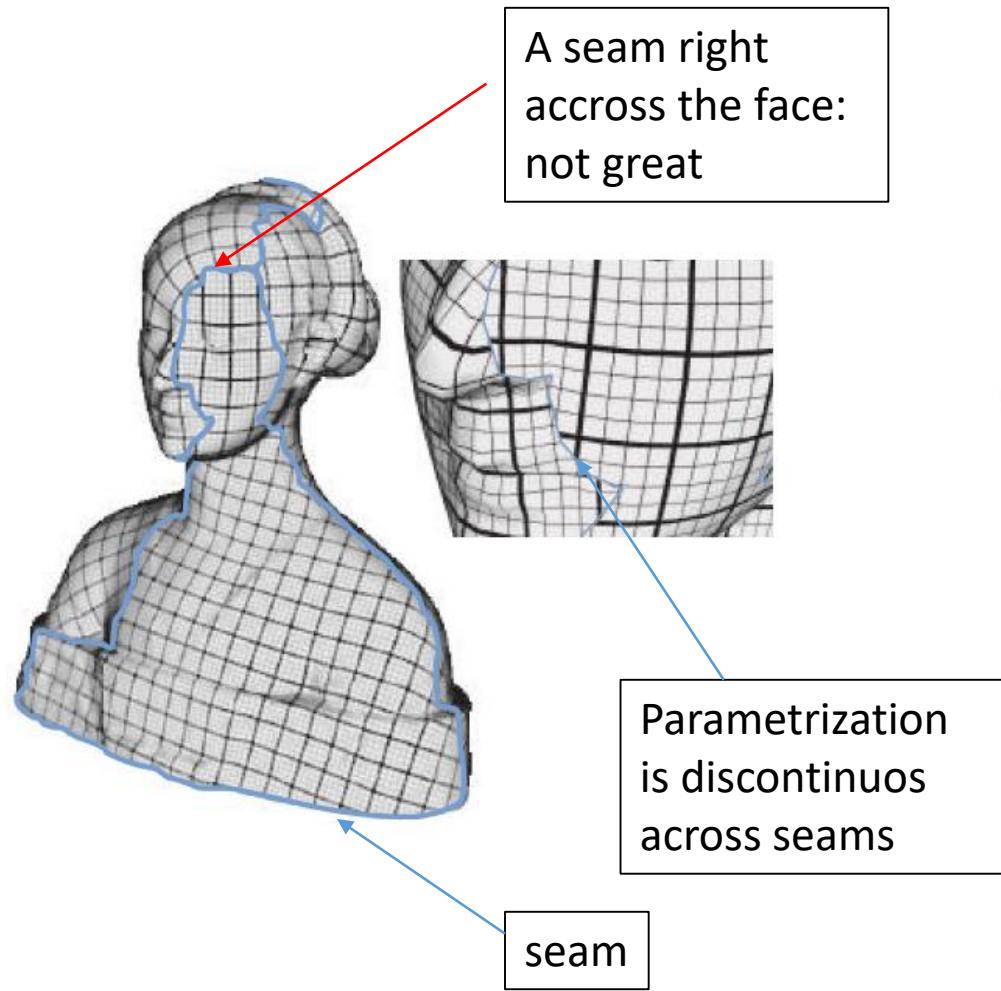
Examples of Mesh Parametrization

Typically the surface is «cut» into regions easier to parametrize.

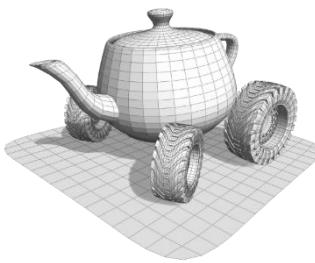
These cuts are called **seams**

Finding good seams:

- So that they partition the mesh in *almost* developable regions
- So that they are not in too-visible region of the mesh

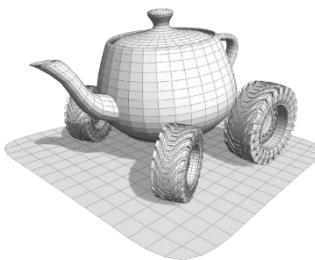


Generating texture coordinates



- Textures are not only used to model the appearance of an object
- They can be used to implement several effects eg:
 - Banners with sliding text
 - Batman signals
 - Mirror-like reflection of the environment
 - Distant horizon
 -
- All of this can be done by **generating texture and/or texture coordinates on-the-fly**





Projective Texturing

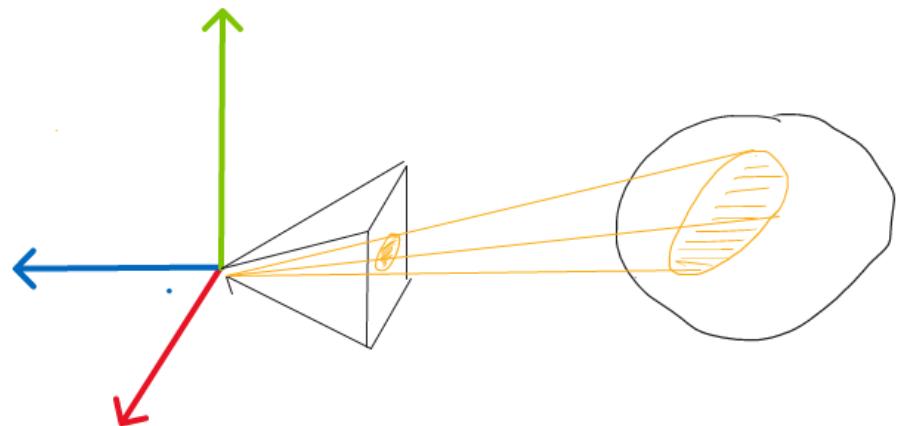
- Projecting a texture on a surface
 - Set up a view matrix pVM and a projection matrix pPM like we normally do for rendering
 - Project the points with the new matrices but, instead of assigning the outcome as output, assign the result to the texture coordinates

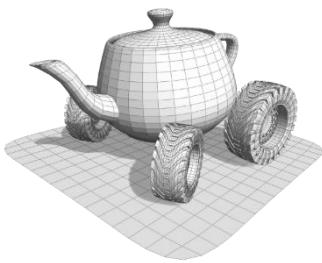
vertex shader

```
out vec4 posProjVS;  
posProjVS = uP*pT*...*vec4(pos,1.0);
```

fragment shader

```
vTexCoords = (posProjVS/posProjVS.w).xy;
```

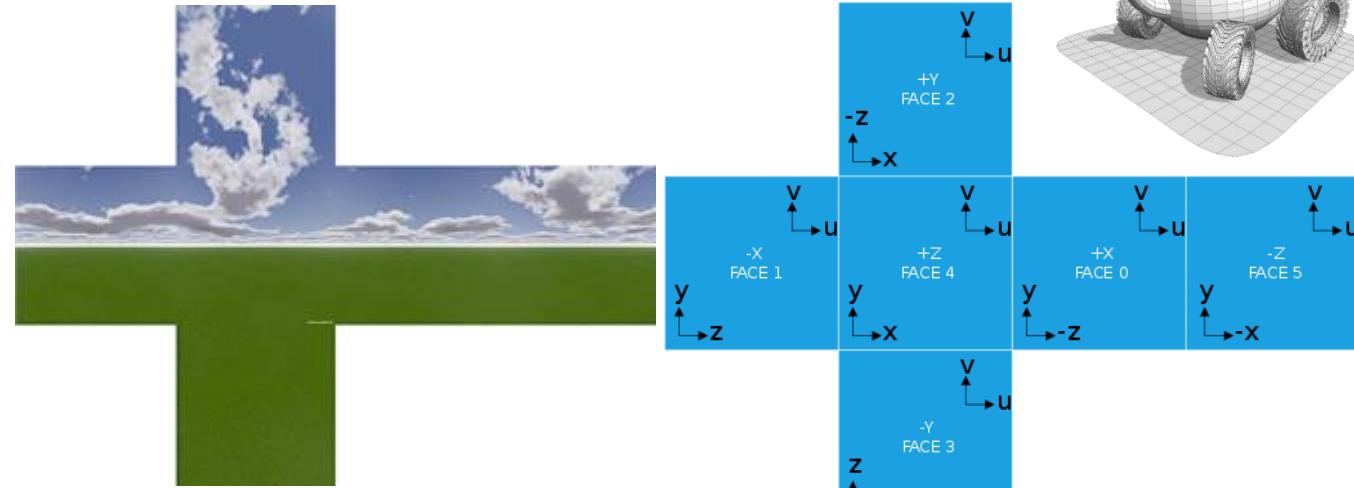
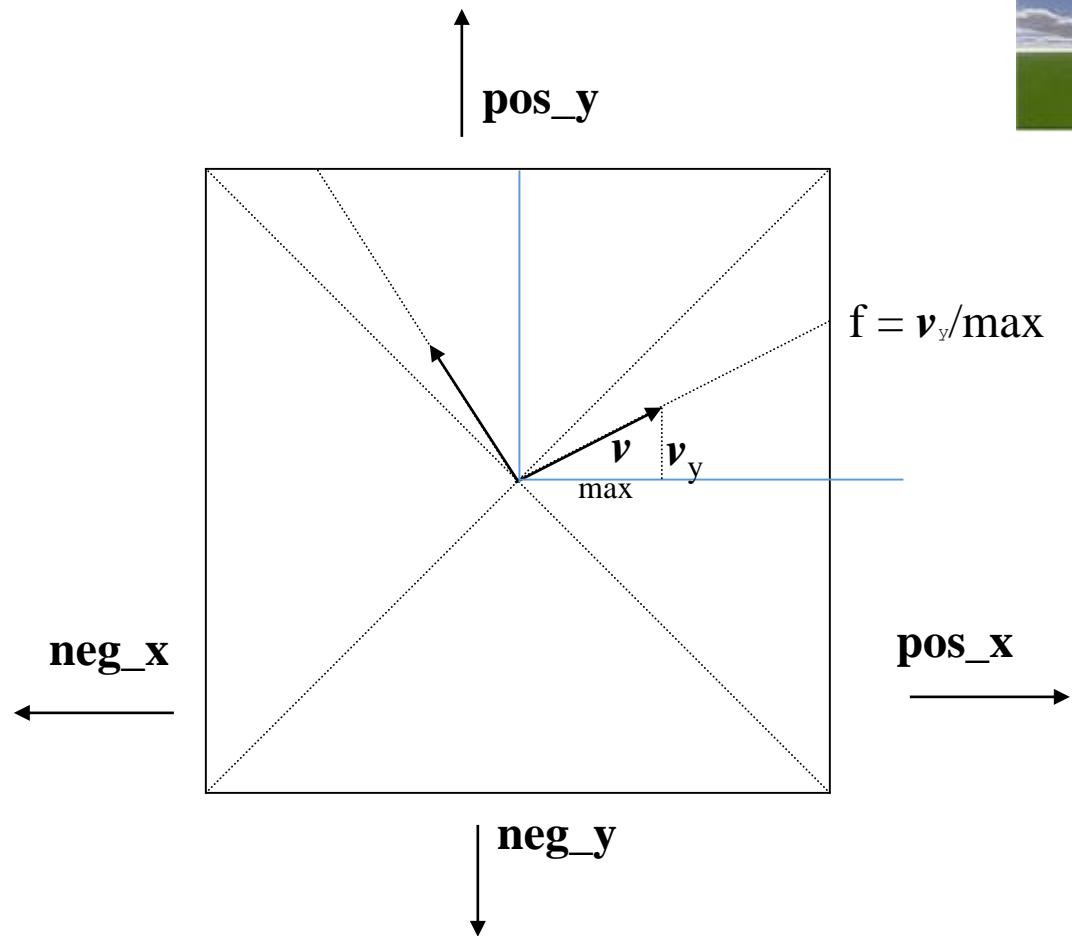




Environment Maps

- An Environment Map is a map that represents the faraway environment
- In its most basic use, it can show the horizon of the scene (it's also referred to as **skybox**)
 - Draw a large sphere (or cube) that encompasses the whole scene
 - Use texture mapping to stitch the environment map on it
 - OR
 - simply access the texture using the view direction for each pixel
- How do we map from the view directions to the texture coordinates?
(next)

Cube Mapping



What face of the texture is sampled?

$$f = \arg\max (\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z)$$

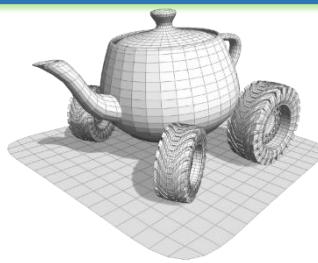
What are the texture coordinates in face s ?

$$s = ((\mathbf{v}_i / \mathbf{v}_f) * 0.5) + 0.5$$

$$t = ((\mathbf{v}_j / \mathbf{v}_f) * 0.5) + 0.5$$

Where i, j are the other coordinates

Cube Mapping



Cube Maps are supported in *GL
Instead of GL_TEXTURE_2D there are 6 texture targets

GL_TEXTURE_CUBE_MAP_POSITIVE_X
GL_TEXTURE_CUBE_MAP_NEGATIVE_X
GL_TEXTURE_CUBE_MAP_POSITIVE_Y
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
GL_TEXTURE_CUBE_MAP_POSITIVE_Z
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z

There is a dedicated sampler

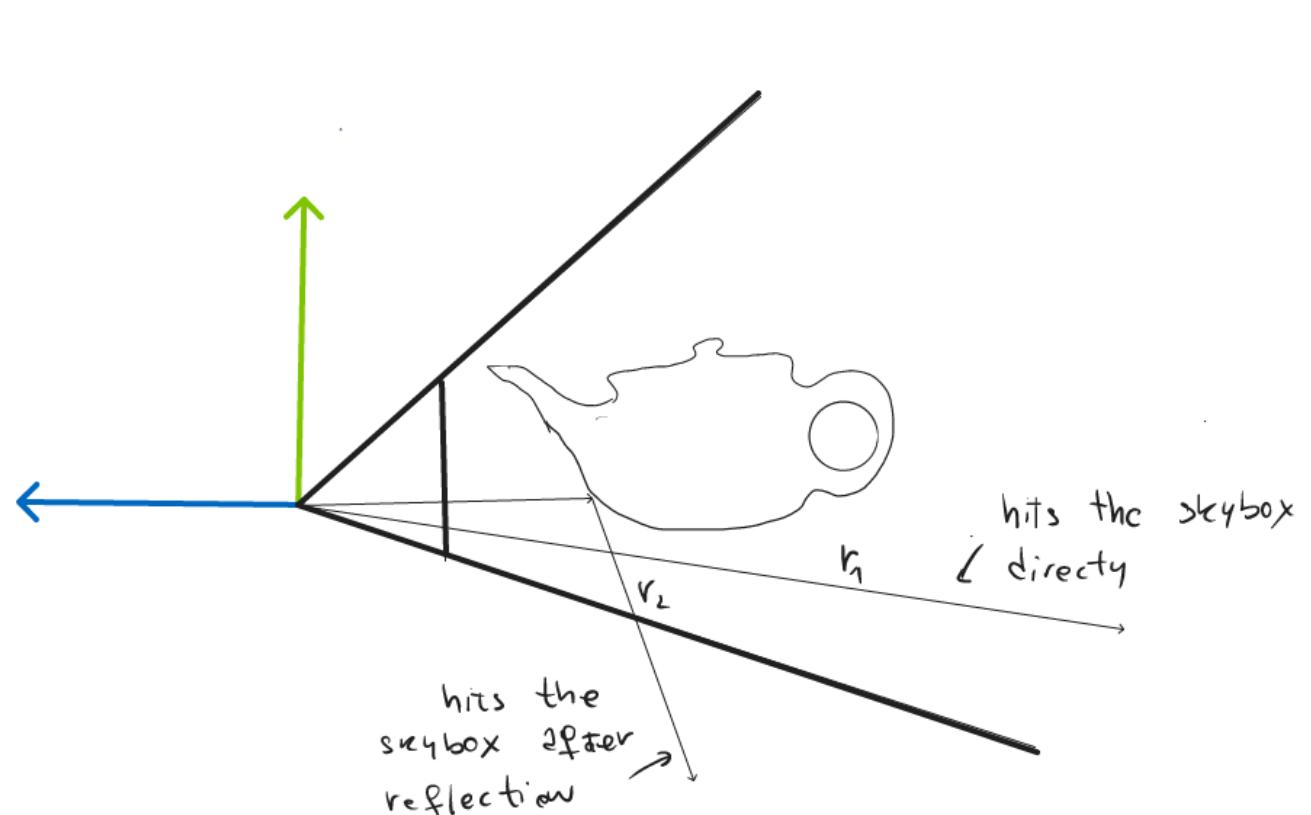
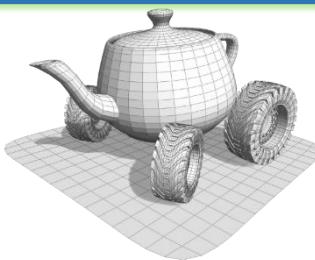
uniform **samplerCube** uSamplerCube;

There is a dedicated function to access the cubemap

textureCube(uSamplerCube, normalize(v));

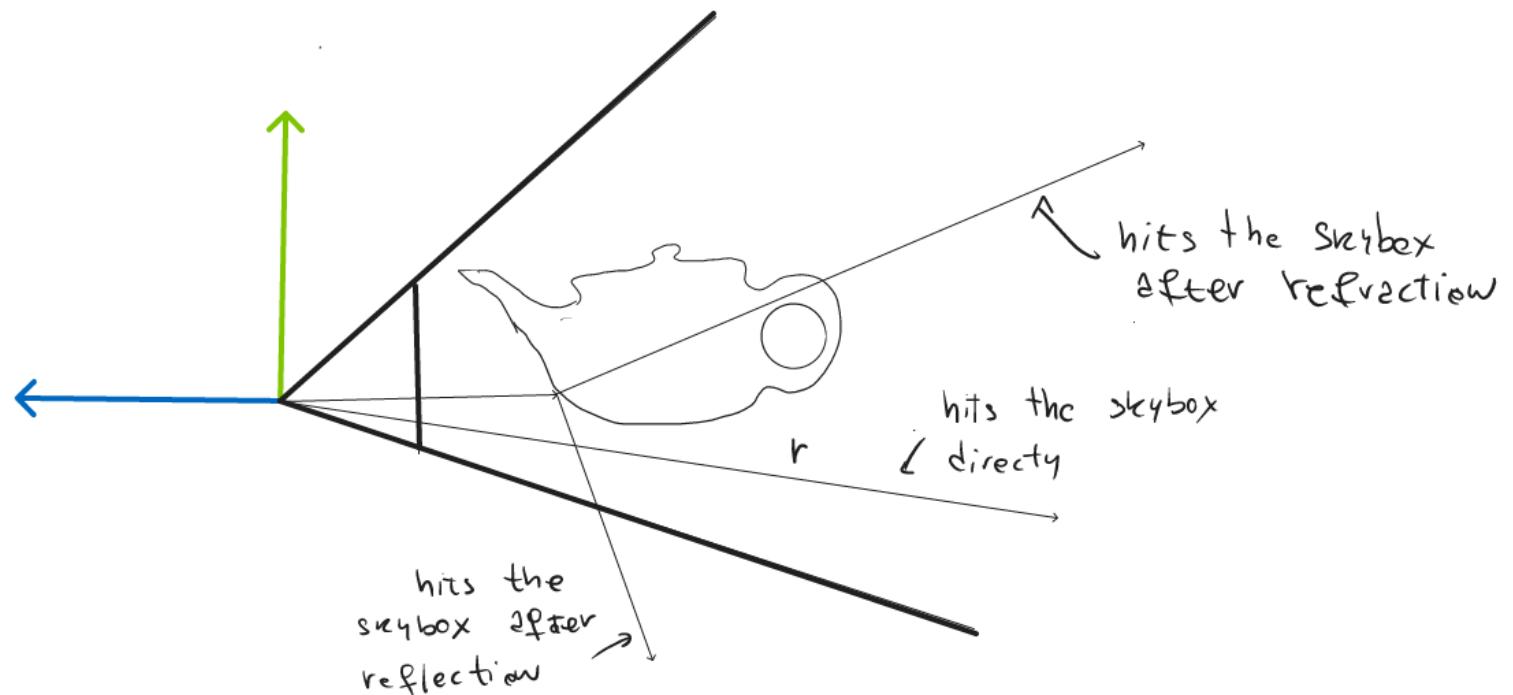
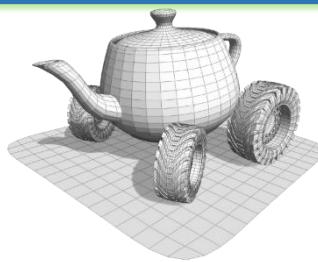
v has three components

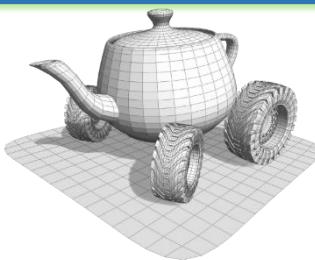
Reflection Mapping



20 X

and Refraction Mapping

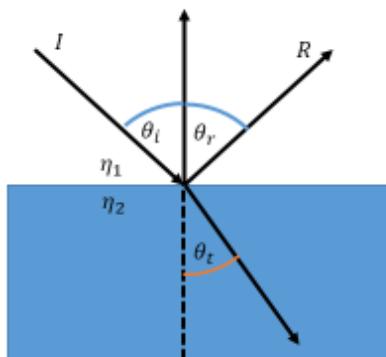




Recall from lighting..

Cook-Torrance model

- For reflection: $\theta_i = \theta_r$
- The angle of refraction follows the **Snell's law**:
$$\eta_1 \sin \theta_i = \eta_2 \sin \theta_t$$
- Where η_1, η_2 are the **indices of refraction**
- What about the *amount* of reflected light?



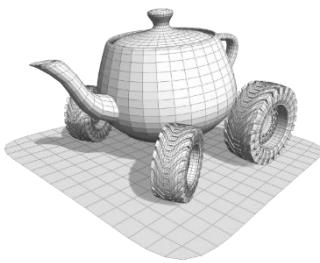
- **Schlick's approximation:** a widely used approximation of the actual Fresnel factor in specular reflection ⇒

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos(\theta))^5$$

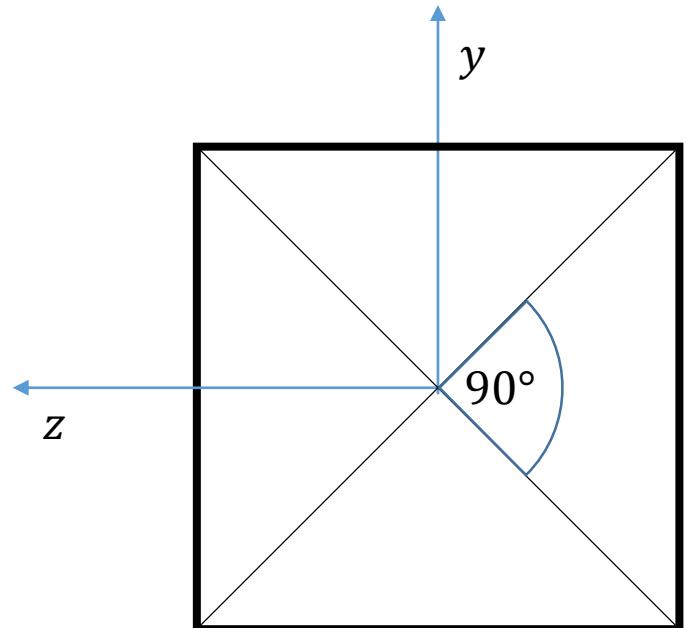
$$R_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2} \right)^2$$

Lez_11_12_advanced illumination models and shading 20_03_2022.pdf

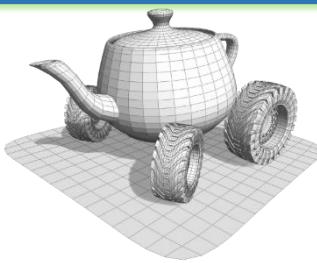
On-the-fly Environment Maps



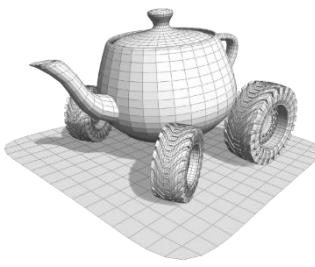
- Environment maps don't have to be static, they can be computed on- the-fly
- 6 rendering, one to create each face of the cubemap
 - looking toward $+x, -x, +y, -y, +z, -z$
 - Field of view = 90° and aspect ratio 1
- From *where* to do the renderings?



Offscreen Rendering

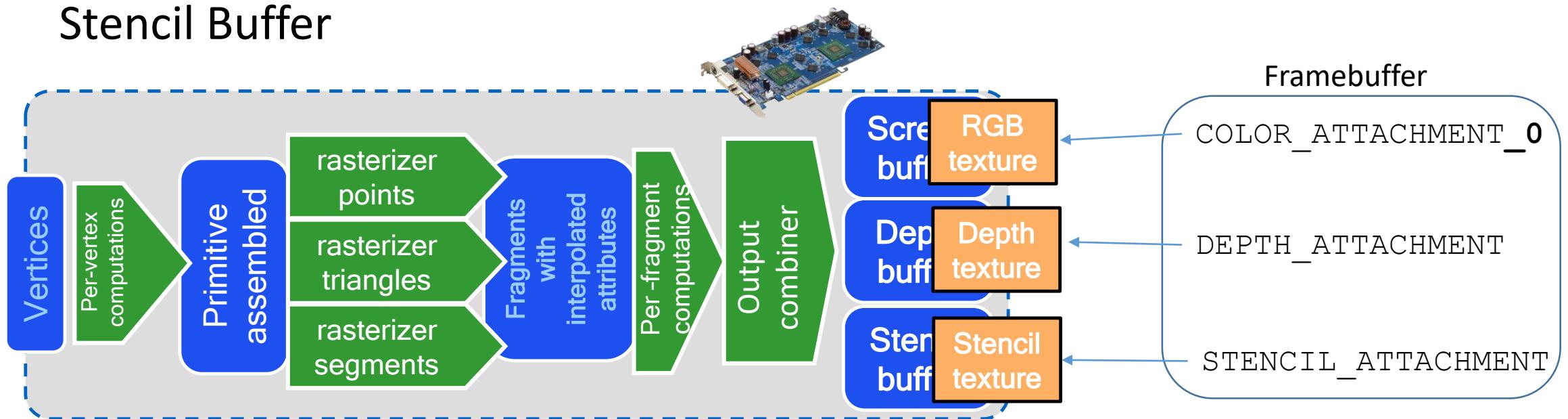


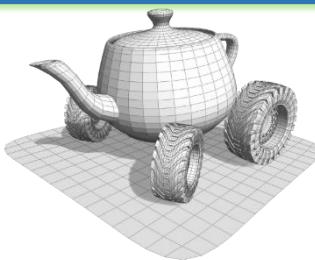
- **Offscreen rendering** stands for rendering without showing the result on screen
- A very useful features in a number of applications, among which Shadow mapping
- It relies on the *GL object Framebuffer objects



Framebuffers

- A Framebuffer is just a collection of *attachments*
- An attachment is a reference to a type of texture that will serve as a replacement for the Screen Buffer (color), the Depth Buffer and the Stencil Buffer





Creating a framebuffer object

```
glGenFramebuffers(1, &framebuffer);
 glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

You can have multiple color attachments to the same framebuffer object because the fragment shader may output more than one value (it has not happened yet)

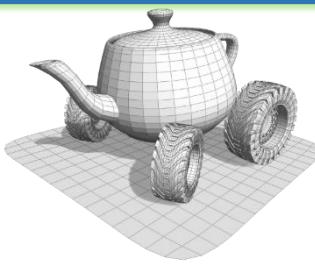
Specify a texture attachment

Same pattern as for everything else (vertex buffers, textures..)

```
glFramebufferTexture2D(
    GL_FRAMEBUFFER,           // target
    GL_COLOR_ATTACHMENT0,    // attachment point
    GL_TEXTURE_2D,           // texture target
    texture,                 // texture (as from gl.createTexture())
    0);                      // mip level
```

The texture must be created first

...what about the depth buffer?



Creating a framebuffer object

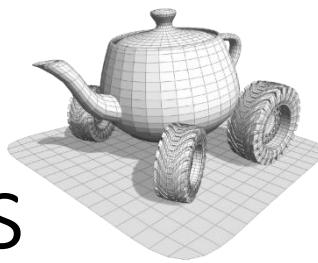
Specify a renderbuffer attachment

Define the buffer to use to play as depth buffer.
This buffer will **not** be bound as a texture and read from a shader in the future, so use a **renderbuffer** (?)

```
glFramebufferRenderbuffer(  
    GL_FRAMEBUFFER,  
    GL_DEPTH_ATTACHMENT,  
    GL_RENDERBUFFER,  
    renderbuffer  
)
```

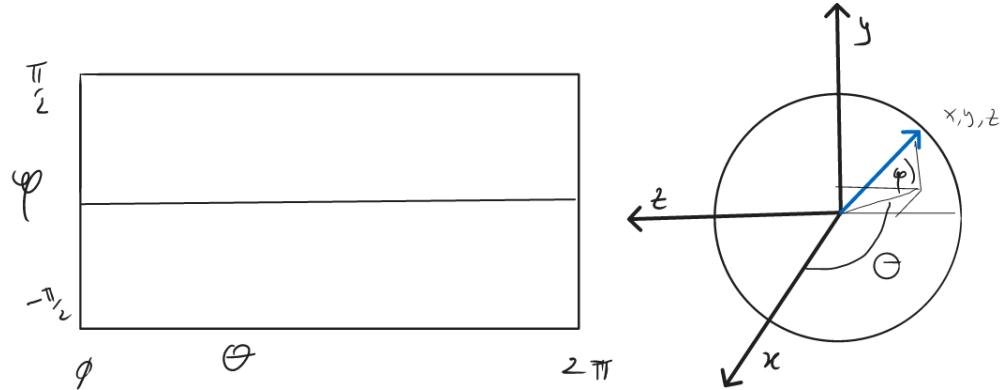
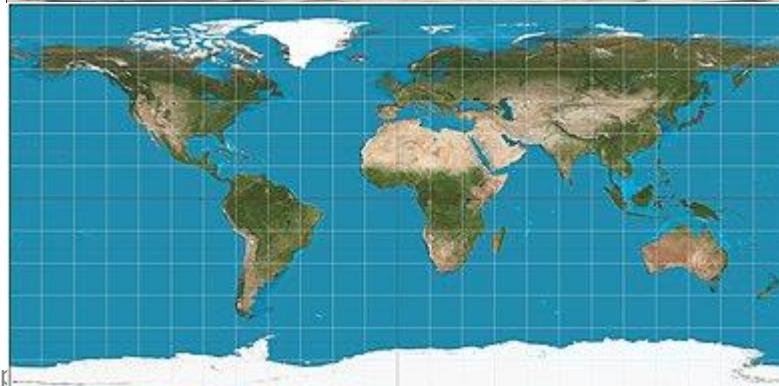
A renderbuffer is created and bound with the usual *GL pattern. Like texture objects, it contains images but it cannot be read directly from the shaders

```
glGenRenderbuffers(1, renderbuffer);  
 glBindRenderbuffer(GL_RENDERBUFFER, renderbuffer);  
  
glRenderbufferStorage(  
    GL_RENDERBUFFER,  
    GL_DEPTH_COMPONENT16,  
    width,  
    height);
```



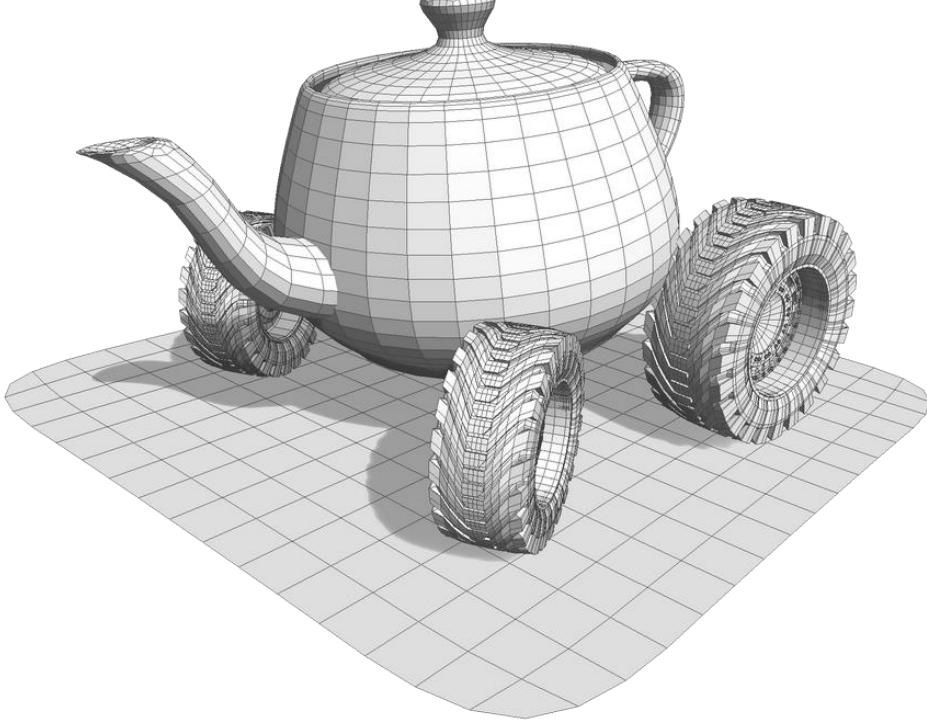
Not just Cubemaps: equirectangular images

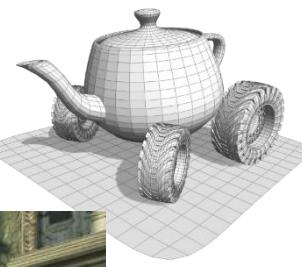
- Equirectangular images have been made popular by 360° cameras and applications



$$\theta = \text{atan} 2(x, z) + \pi$$
$$\phi = \text{atan} 2(\sqrt{x^2 + z^2}, y)$$

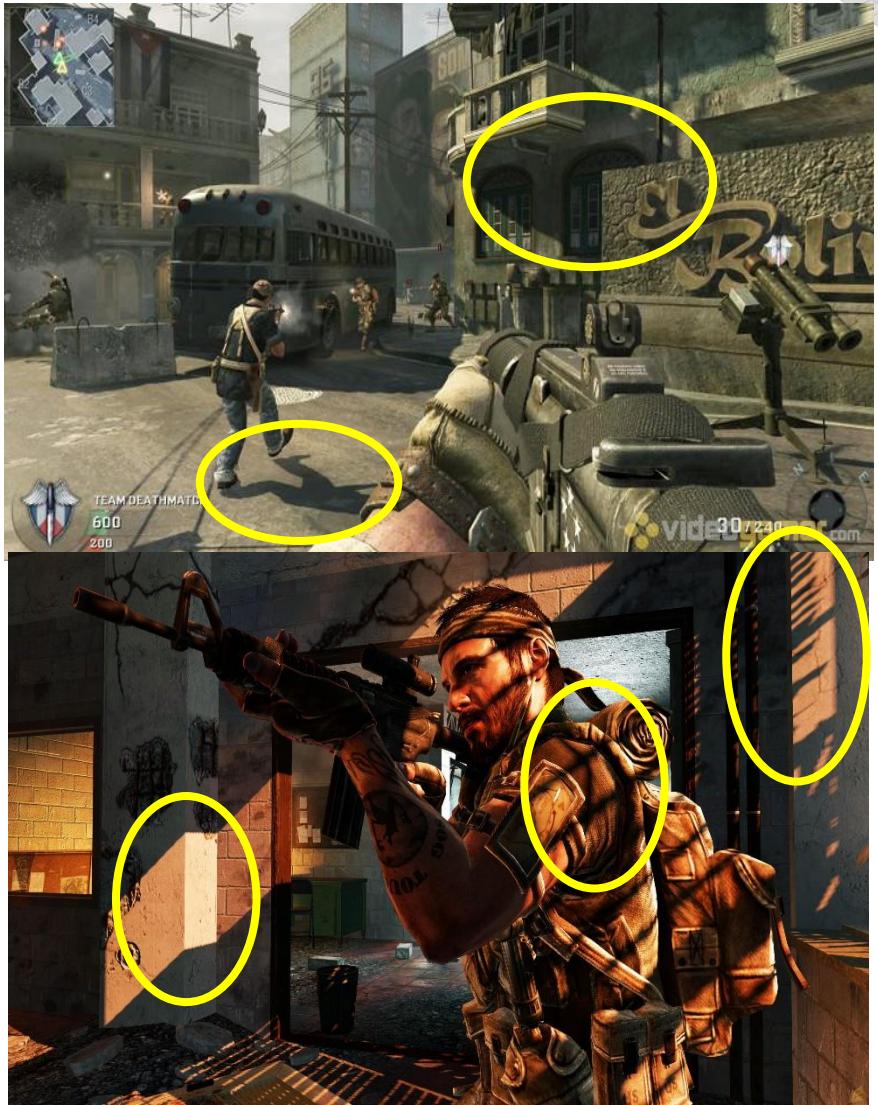
Shadows

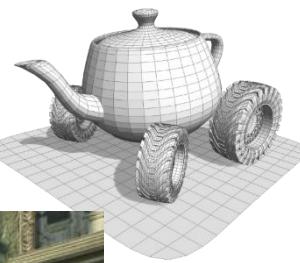




Shadows

- Shadows add realism
 - So do lens flares, depth of field, good textures
- Shadows also contribute to interpret the 3D scene
 - For good and for bad



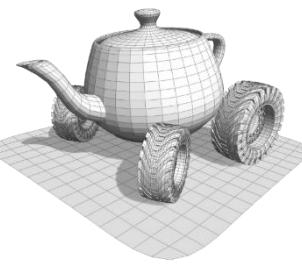


Shadows

- Shadows add realism
 - So do lens flares, depth of field, good textures
- Shadows also contribute to interpret the 3D scene
 - For good and for bad

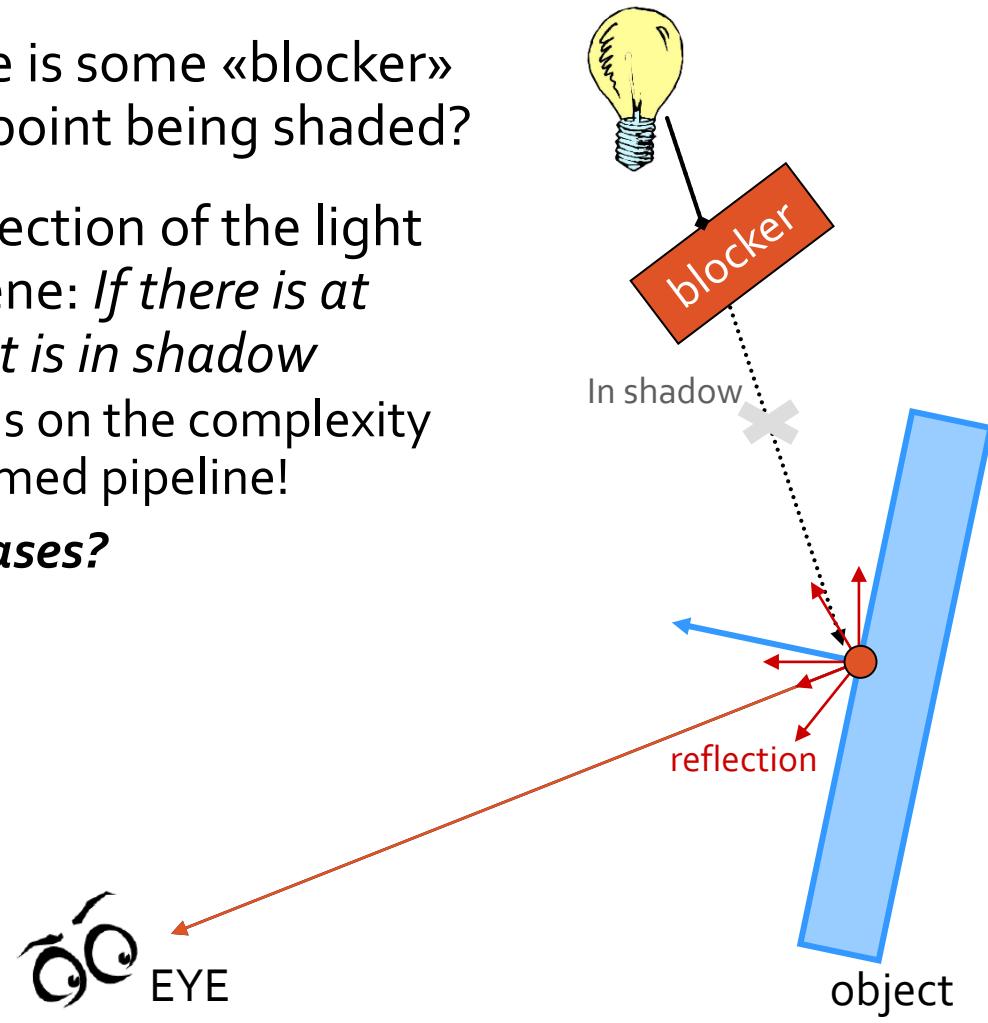


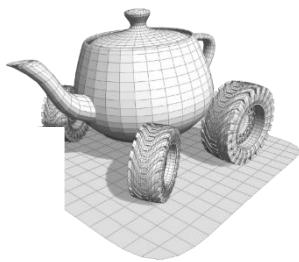
Lahaina noon - Hawaii



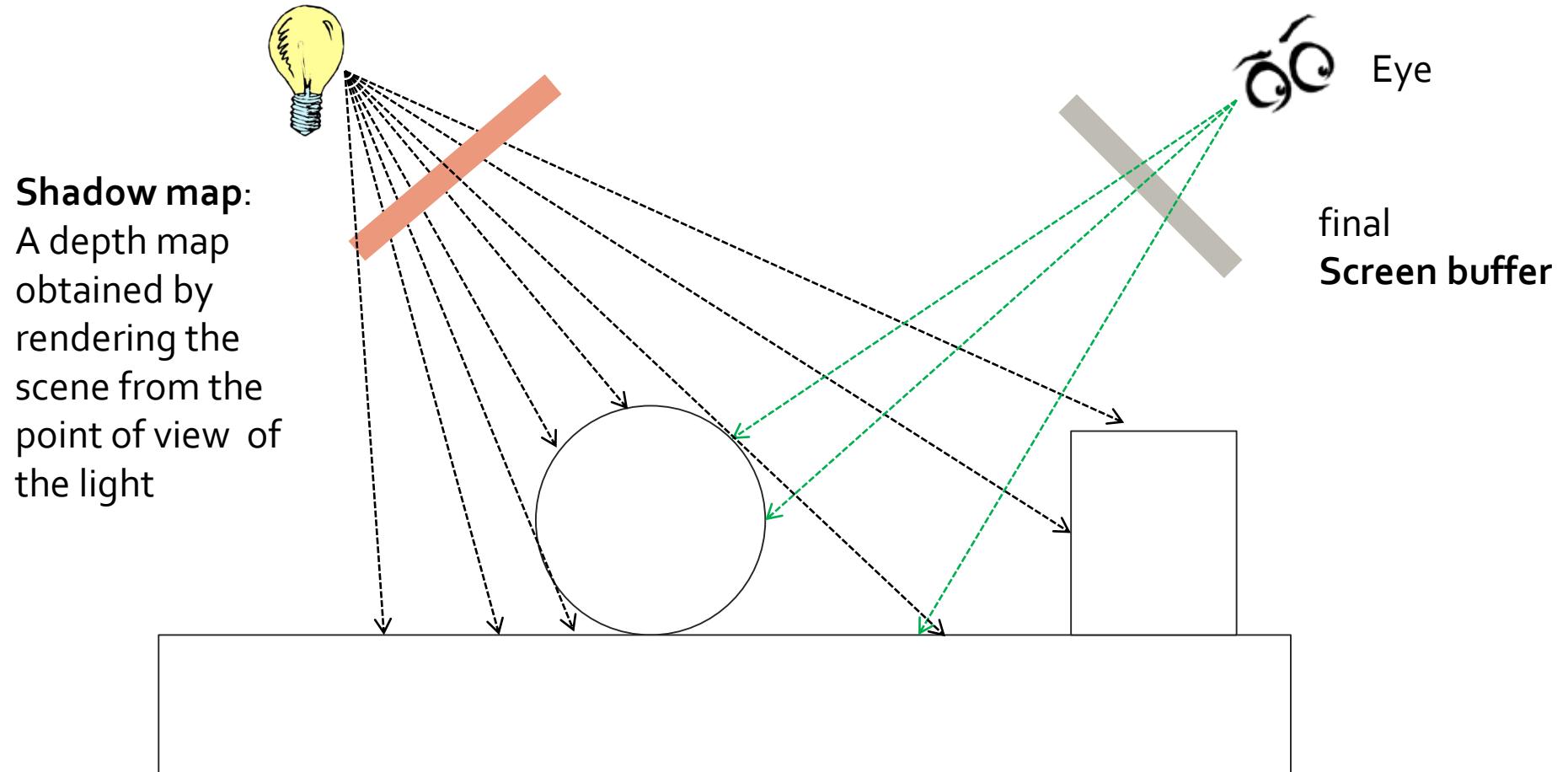
Shadows

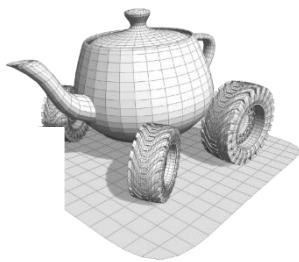
- The problem: how to know if there is some «blocker» between the light source and the point being shaded?
- Naive solution. Just test **ray** intersection of the light ray against every **object** of the scene: *If there is at least one intersection then the point is in shadow*
 - **Bad:** the computation time depends on the complexity of the scene. So much for the streamed pipeline!
 - **Q:** *can still be a solution in some cases?*



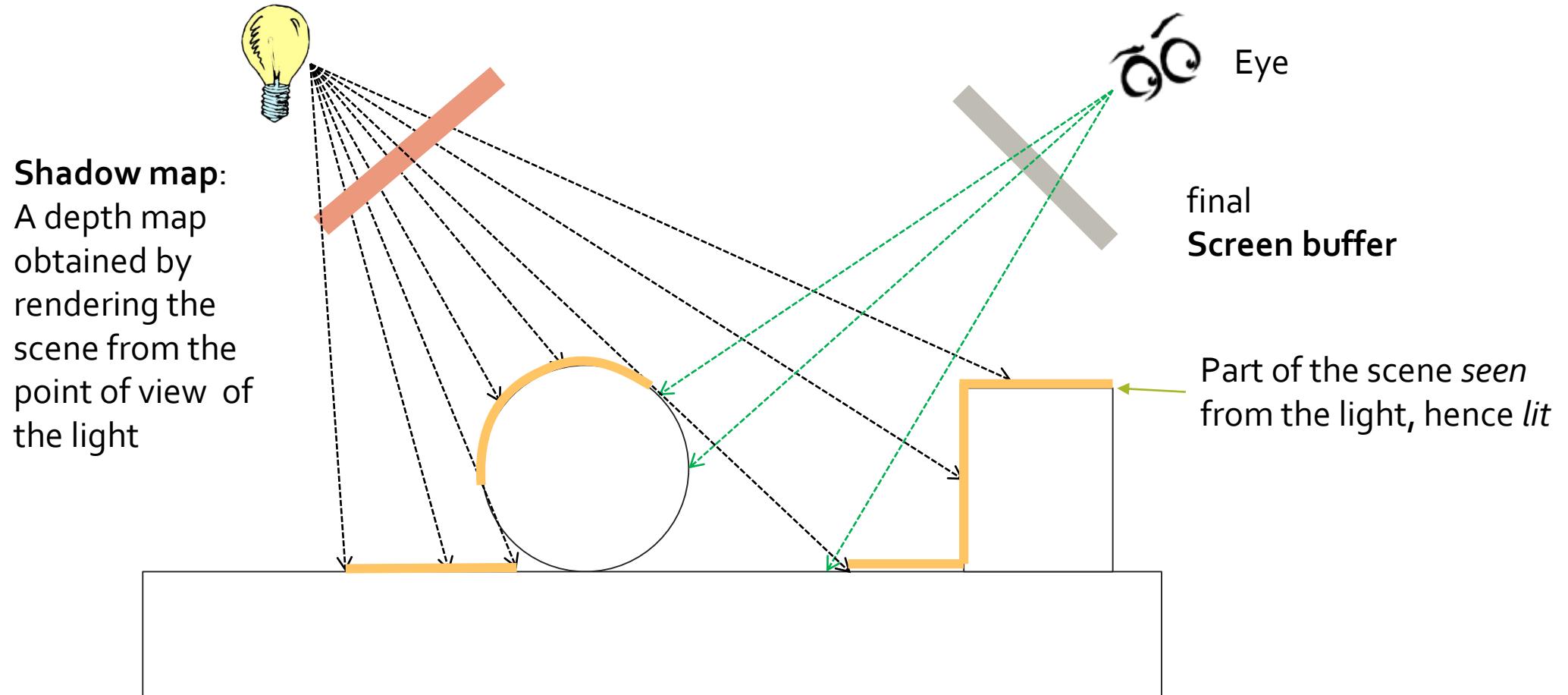


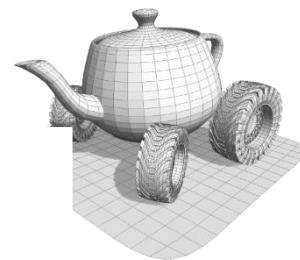
Shadow Mapping (in an ideal world)



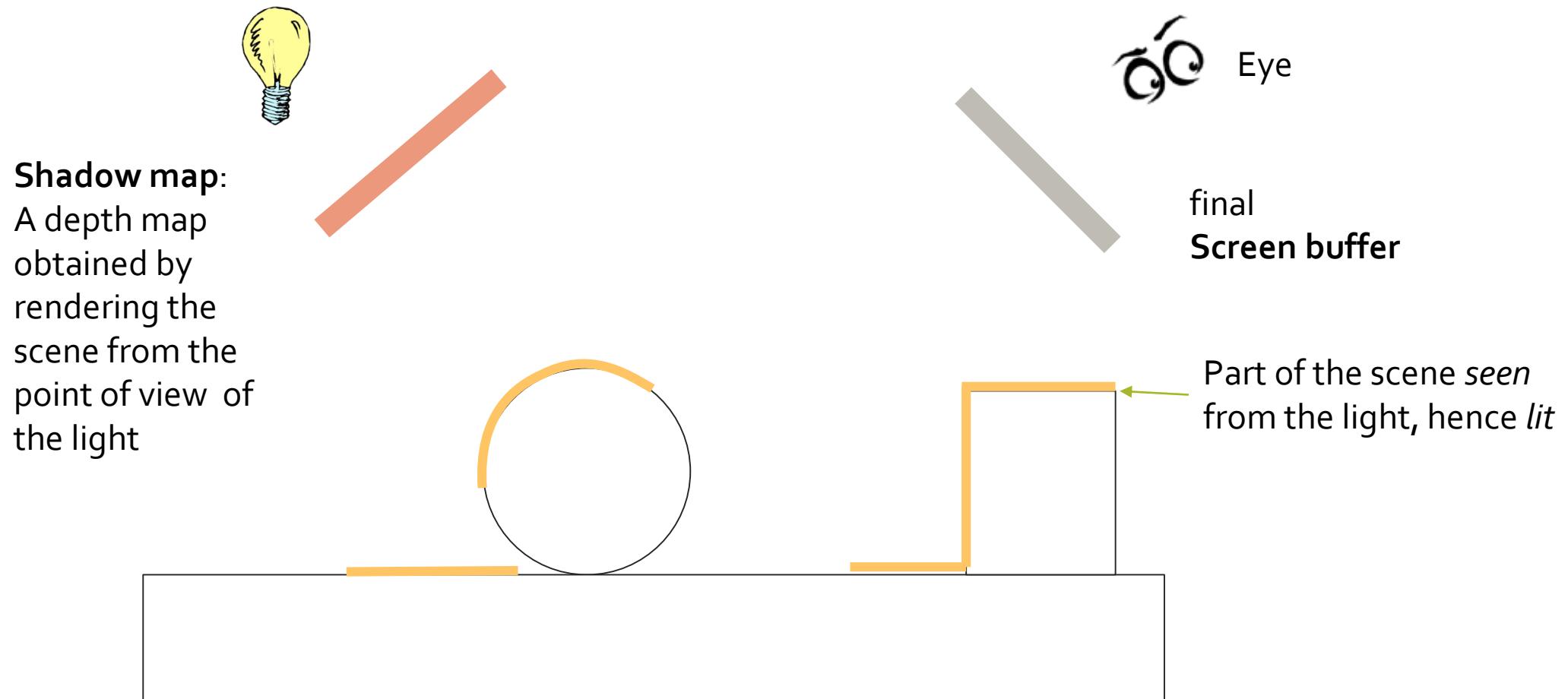


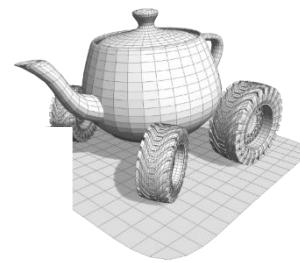
Shadow Mapping (in an ideal world)



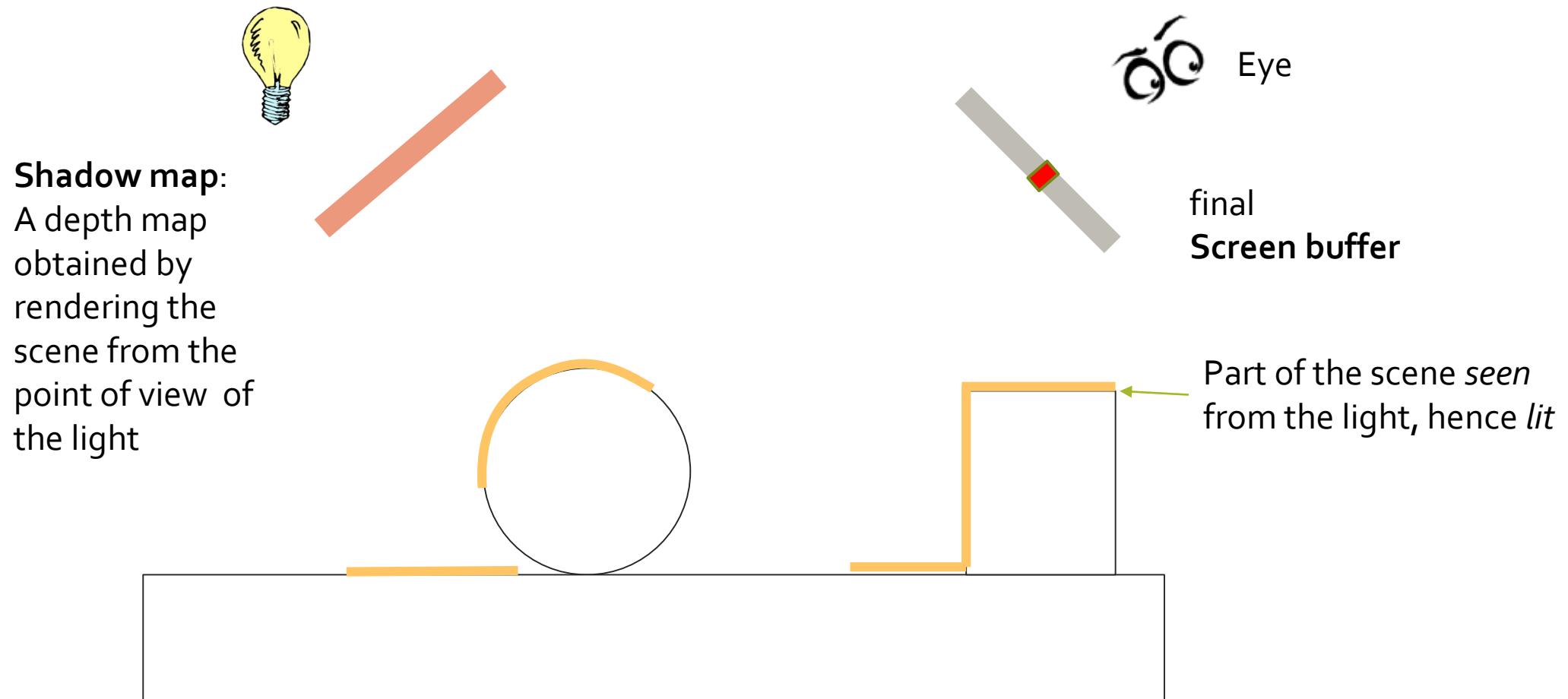


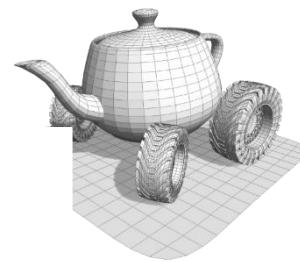
Shadow Mapping (in an ideal world)



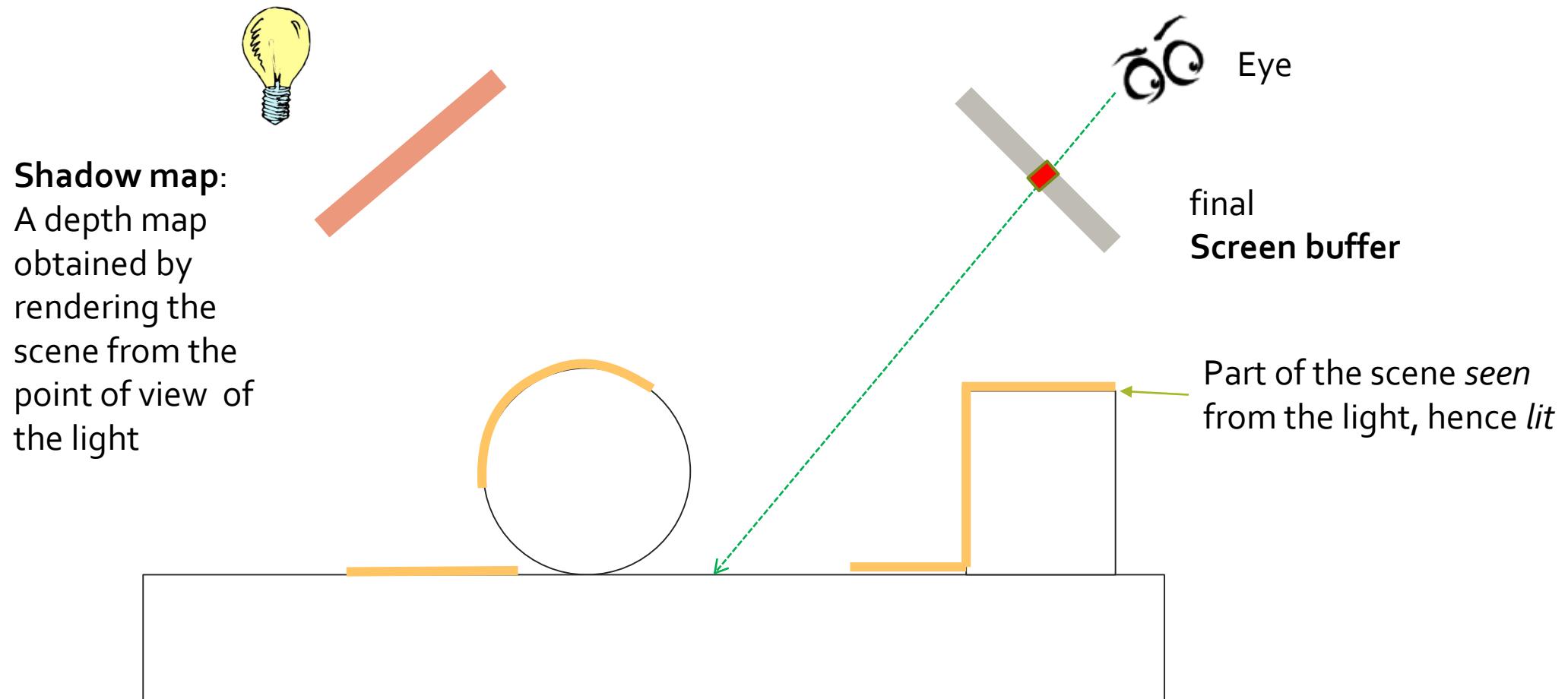


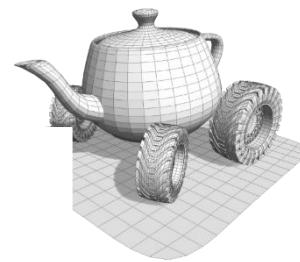
Shadow Mapping (in an ideal world)



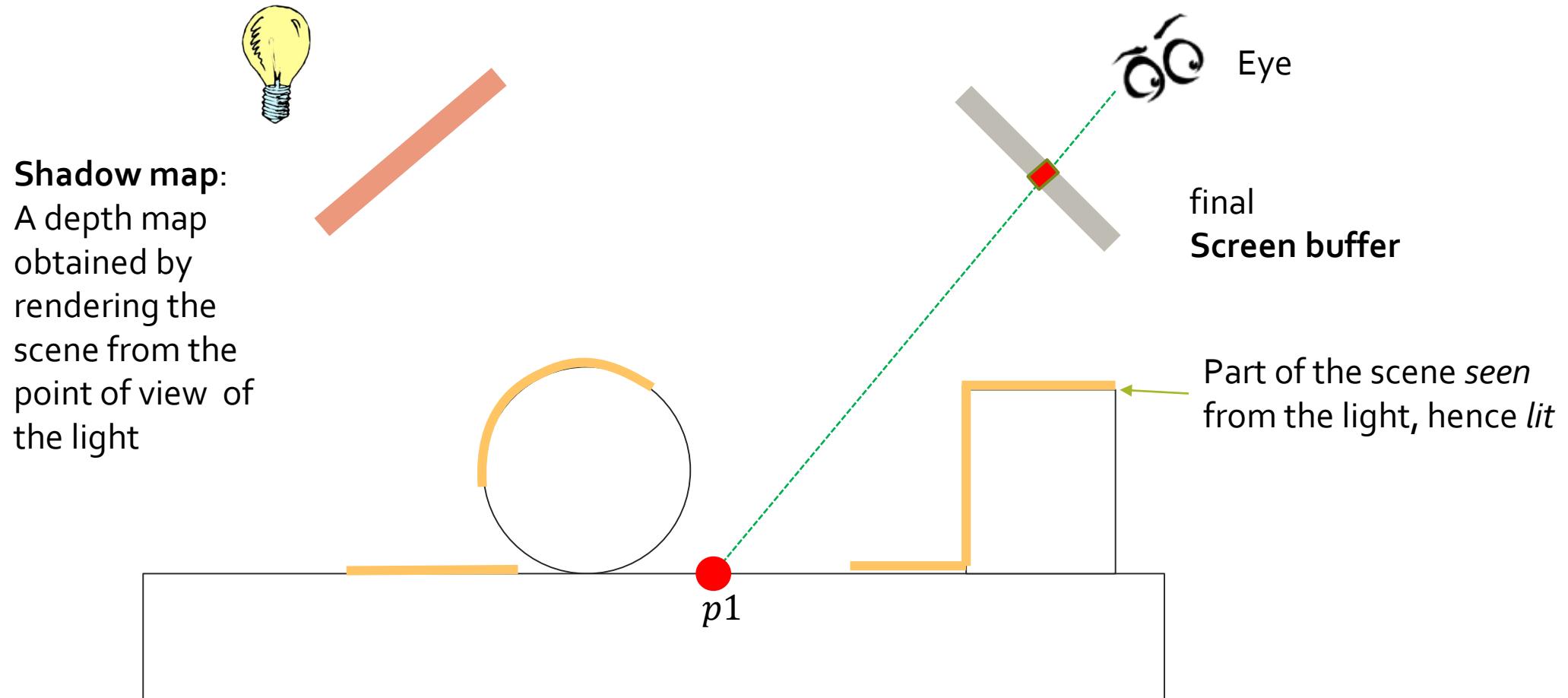


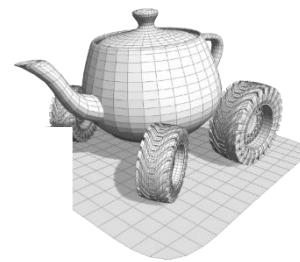
Shadow Mapping (in an ideal world)



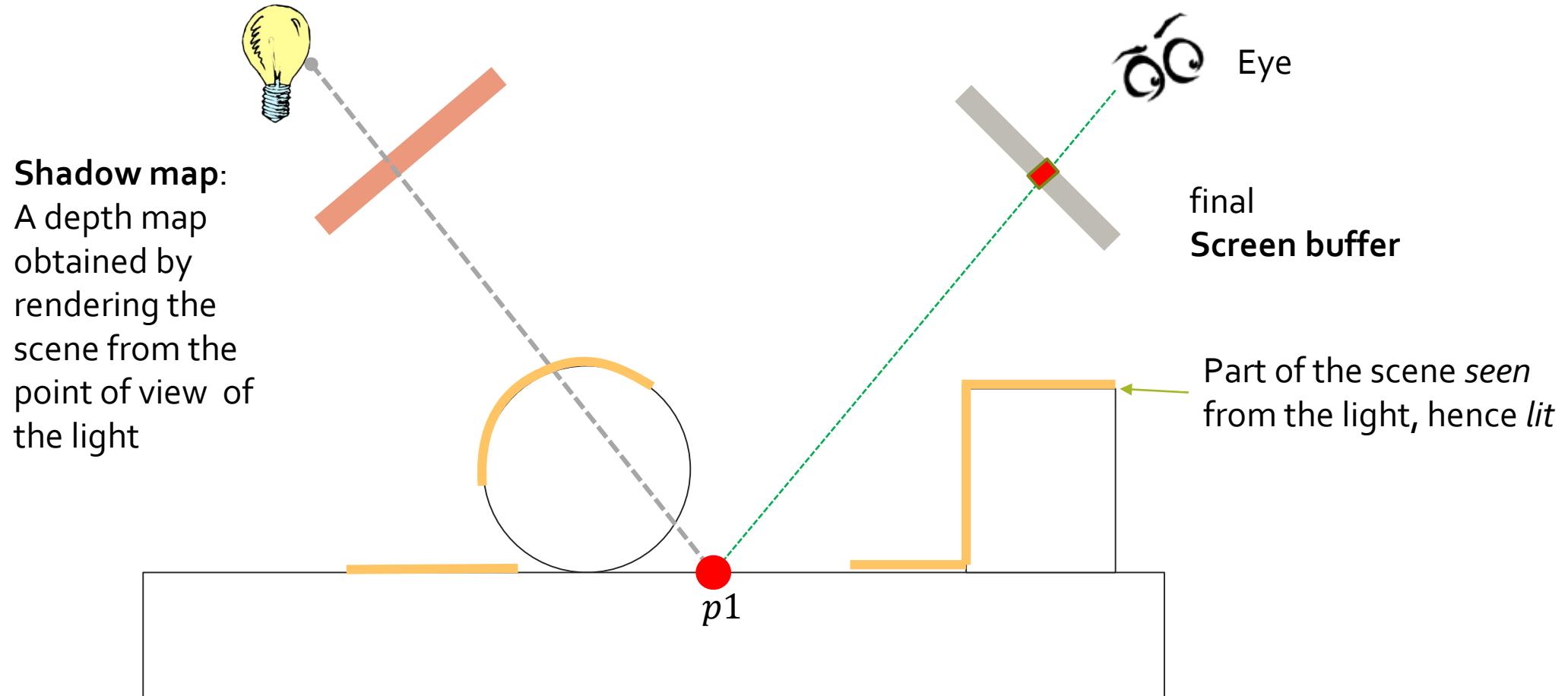


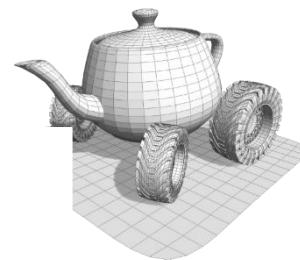
Shadow Mapping (in an ideal world)



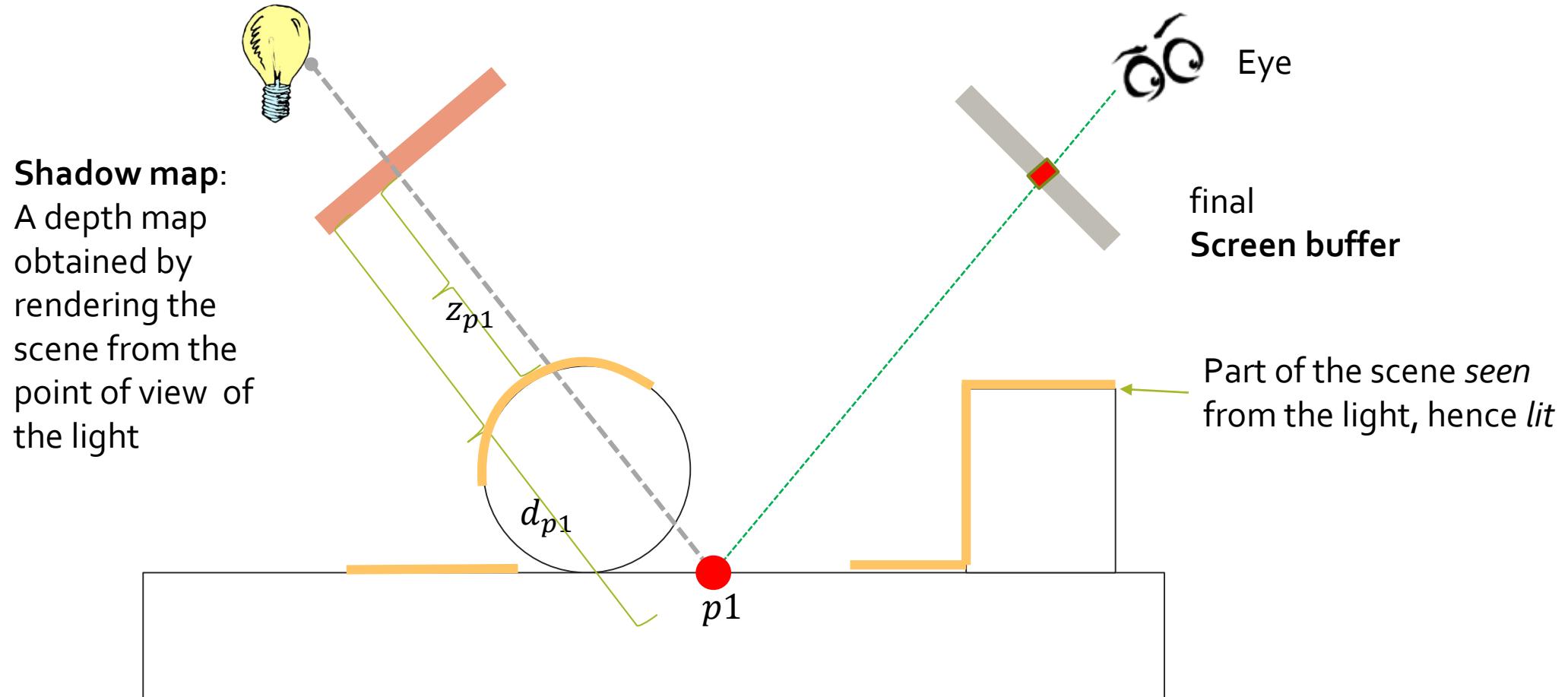


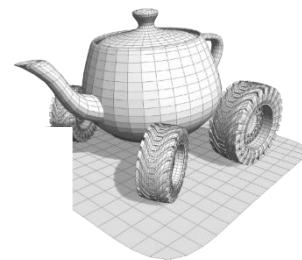
Shadow Mapping (in an ideal world)



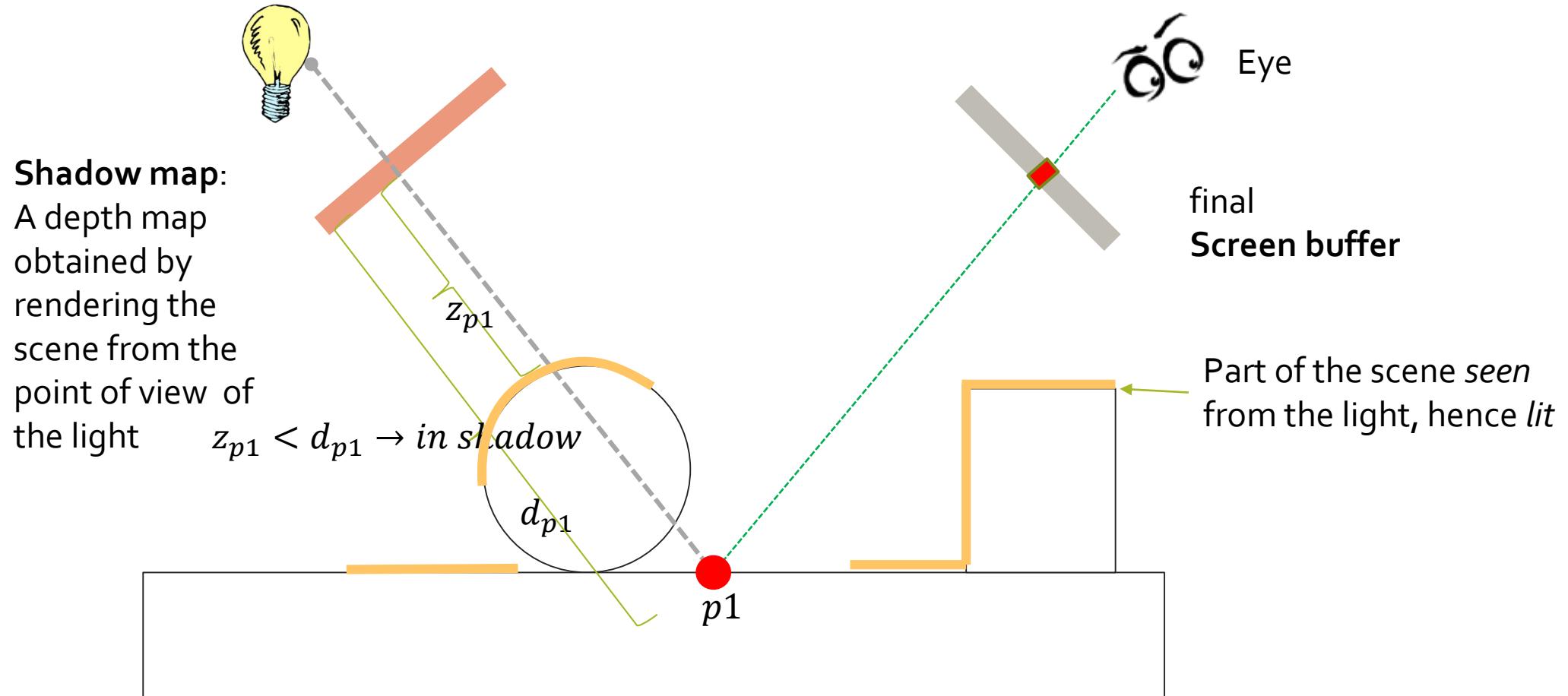


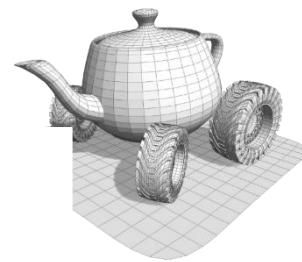
Shadow Mapping (in an ideal world)



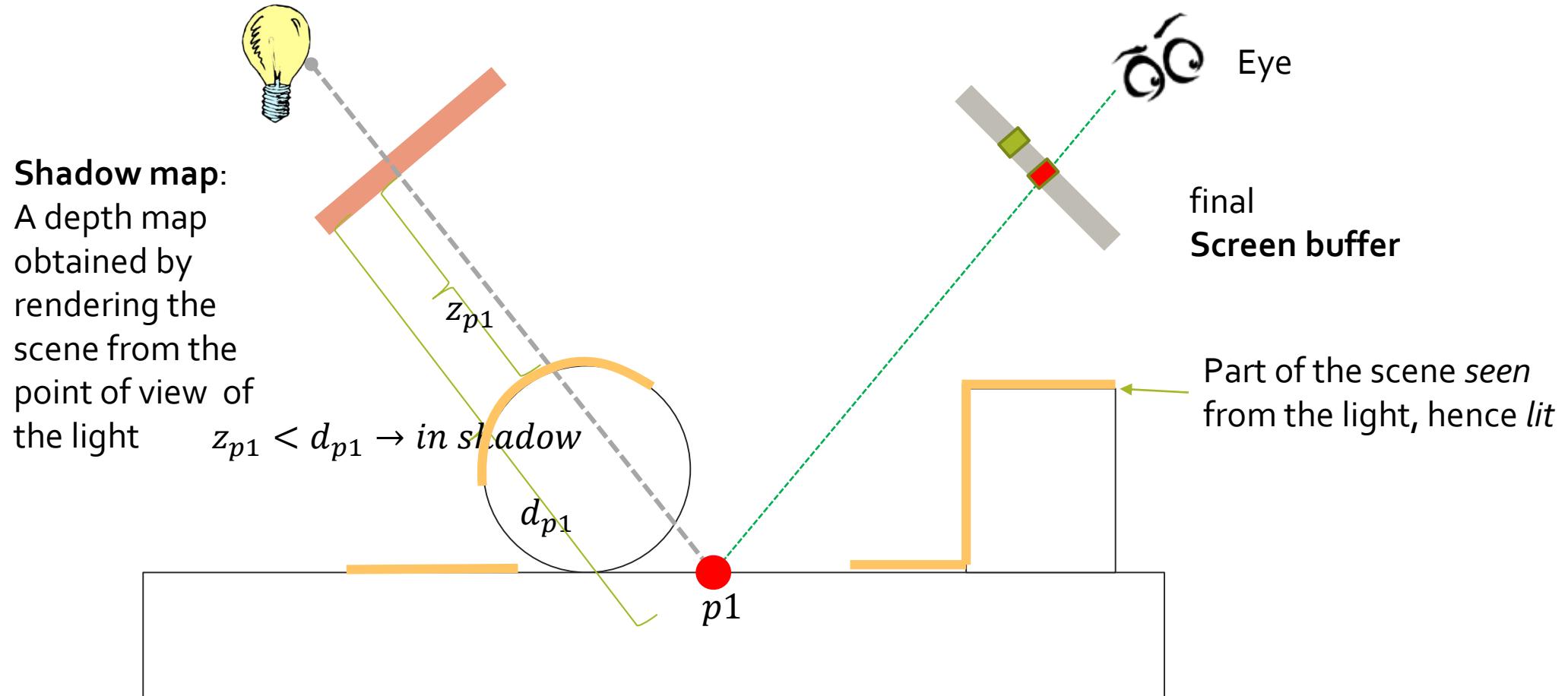


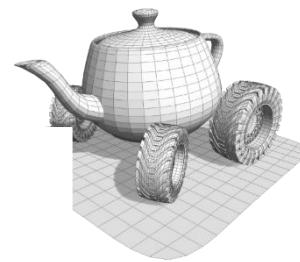
Shadow Mapping (in an ideal world)



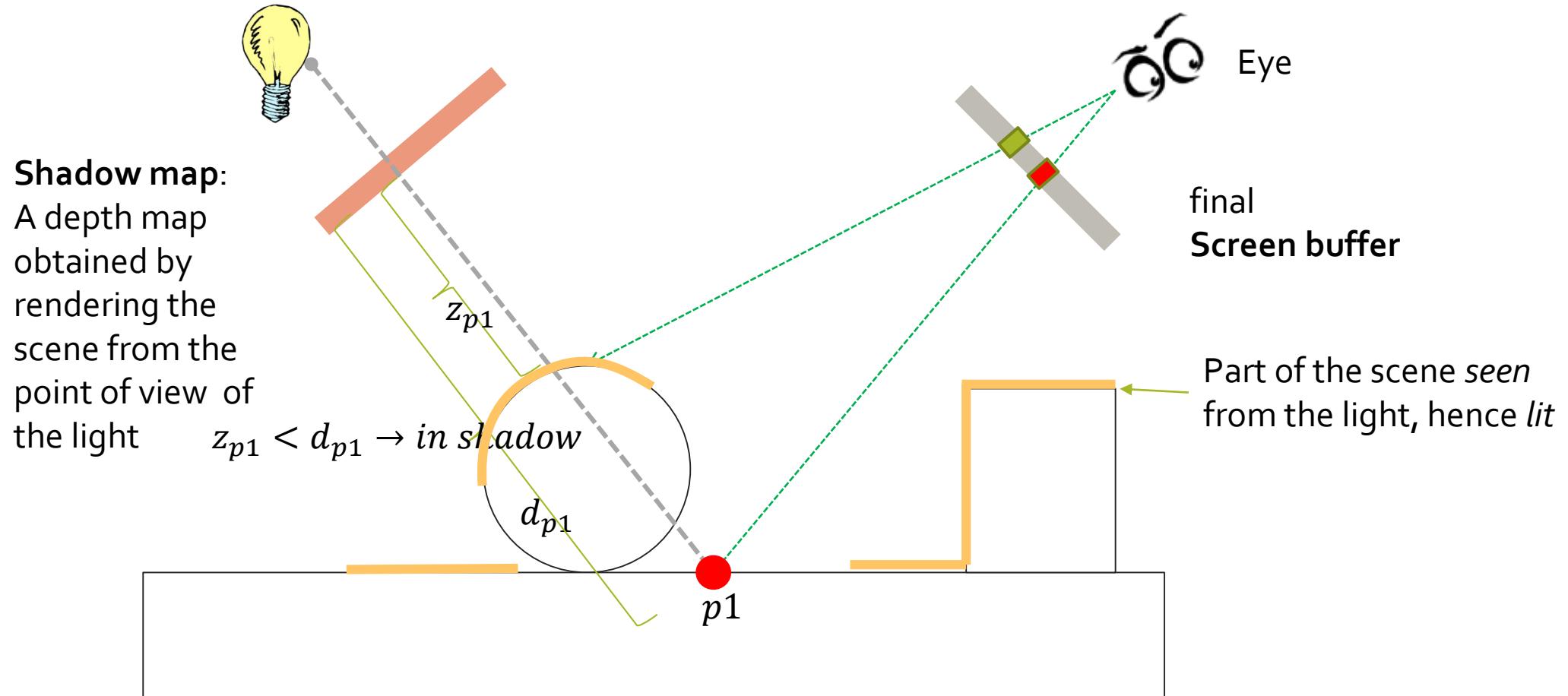


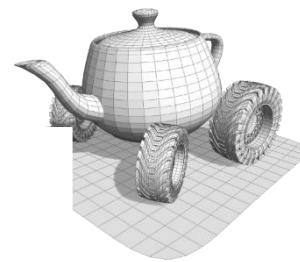
Shadow Mapping (in an ideal world)



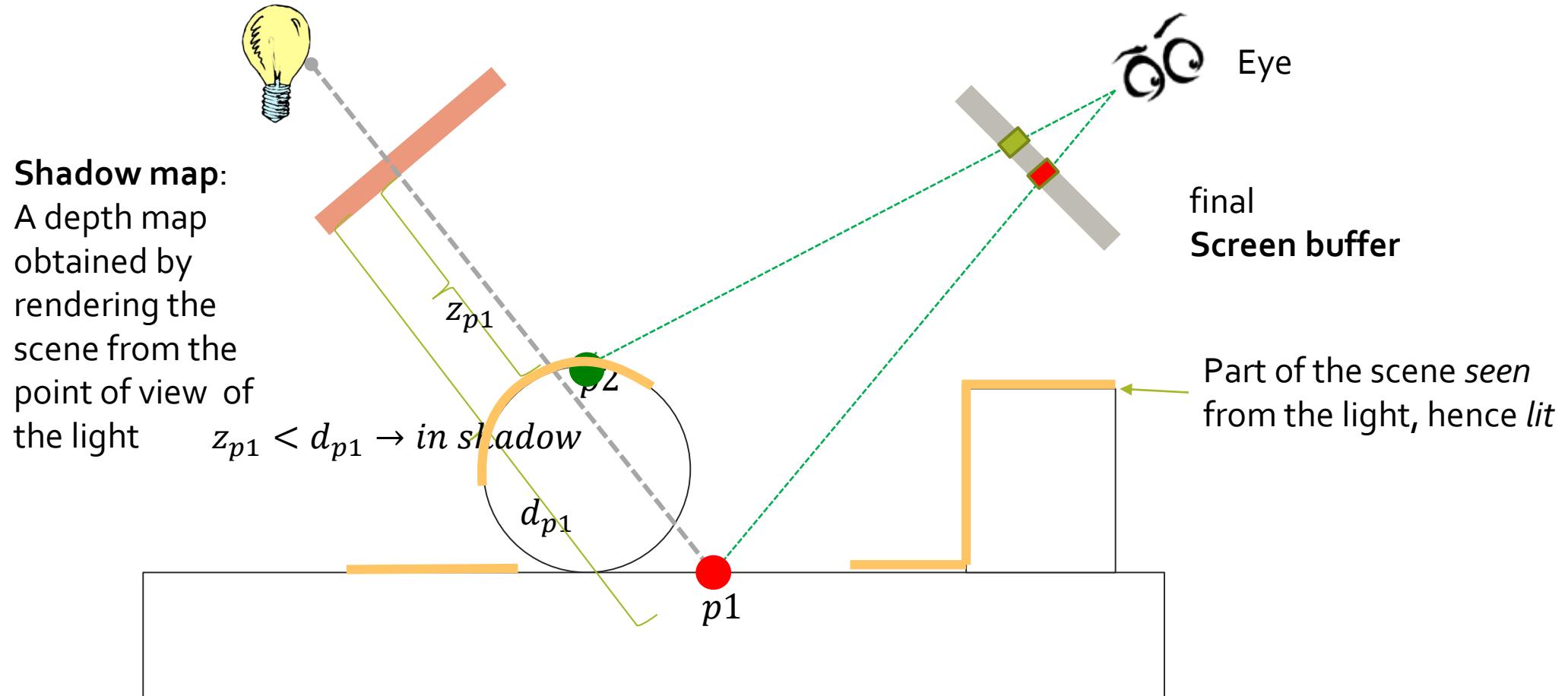


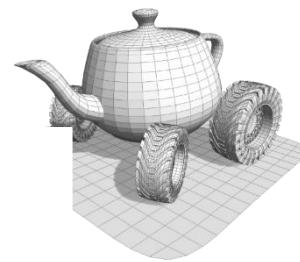
Shadow Mapping (in an ideal world)



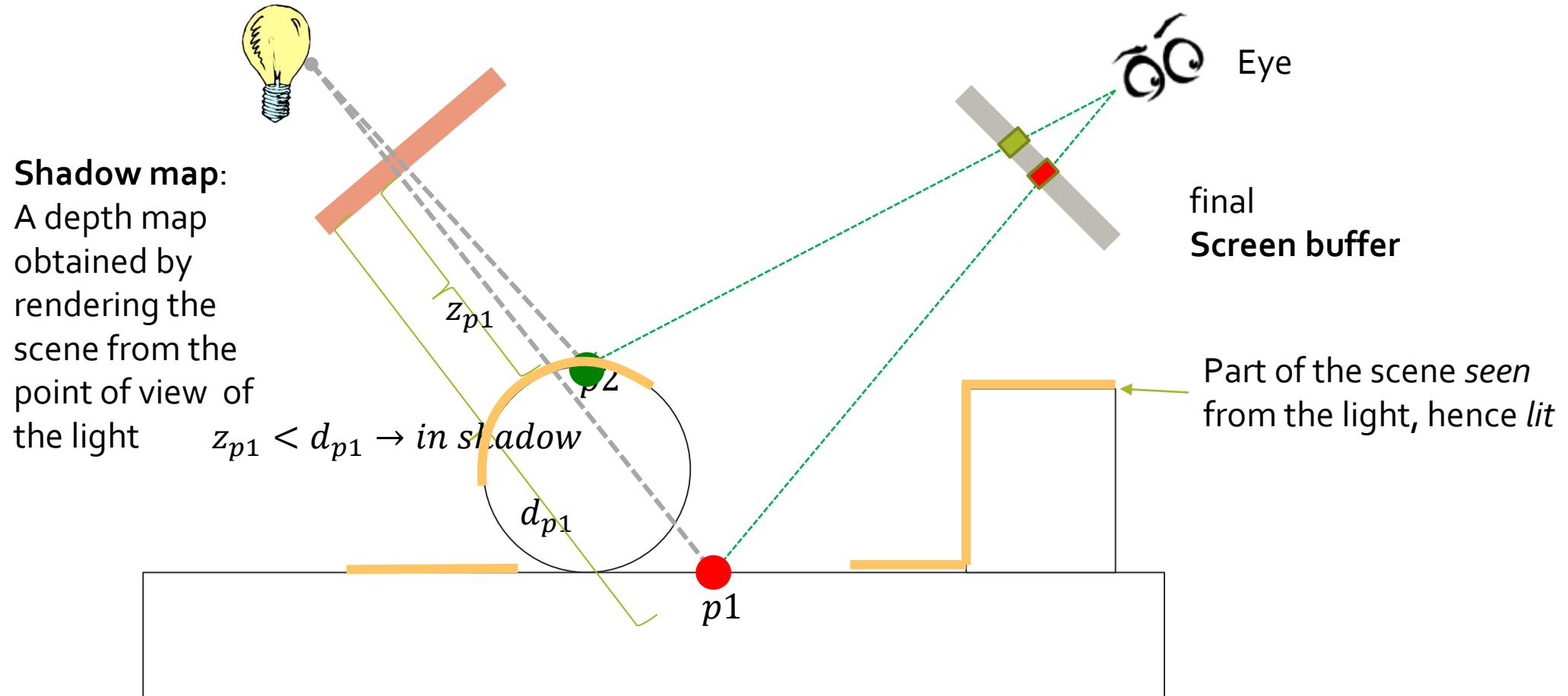


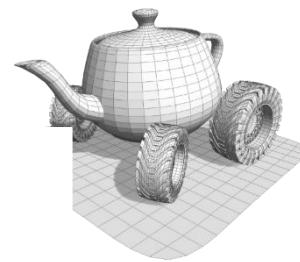
Shadow Mapping (in an ideal world)



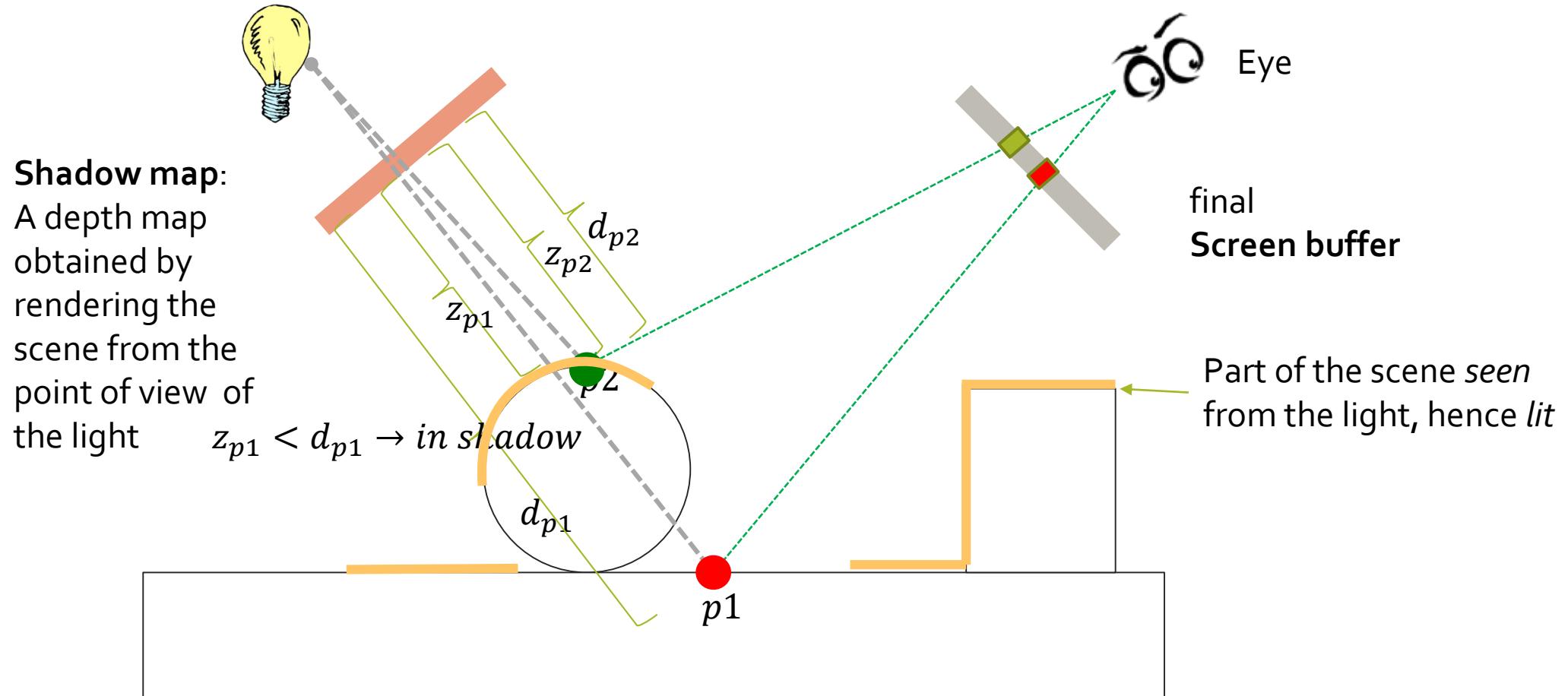


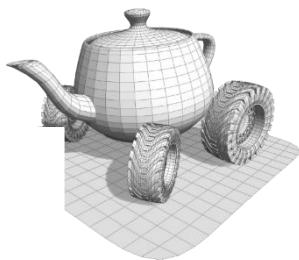
Shadow Mapping (in an ideal world)



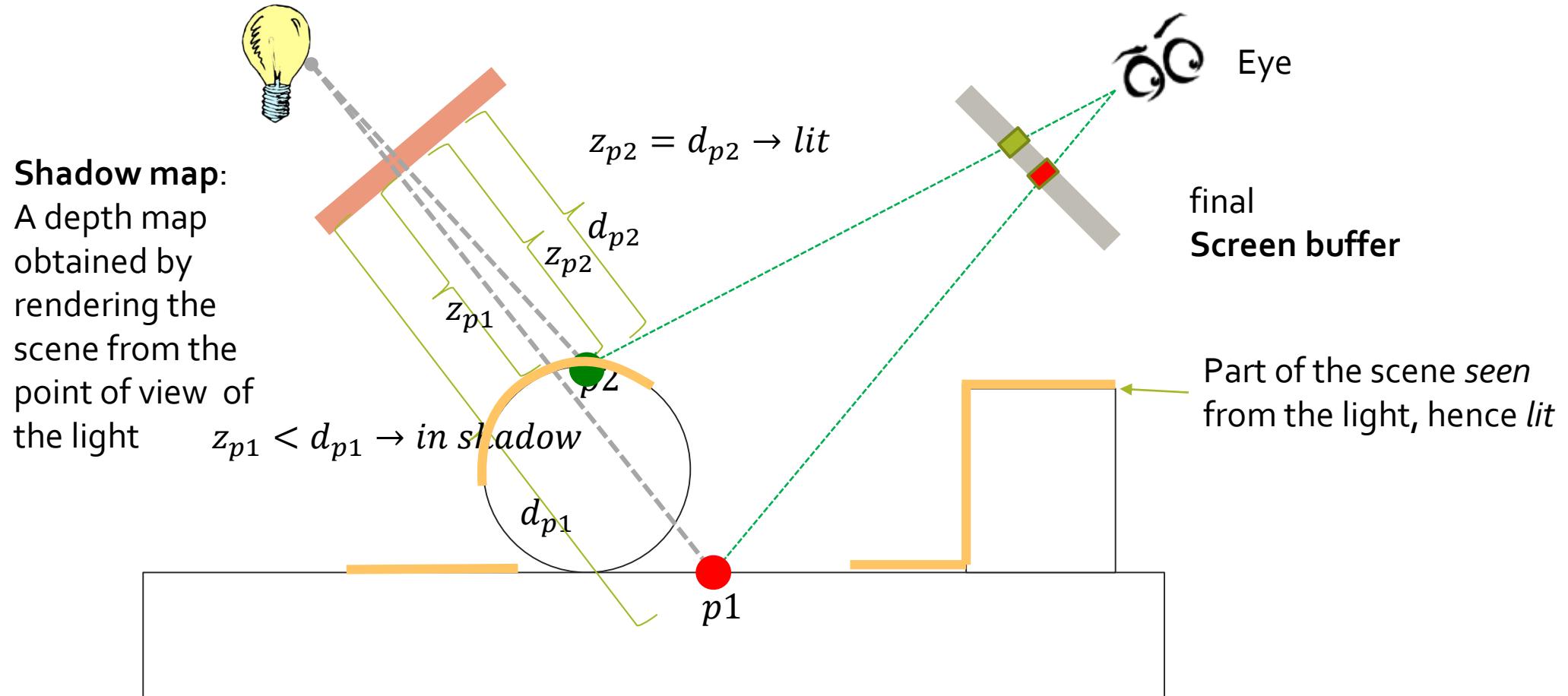


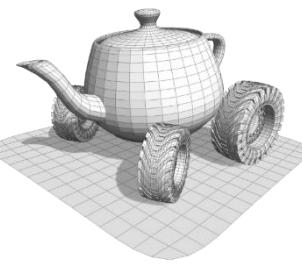
Shadow Mapping (in an ideal world)





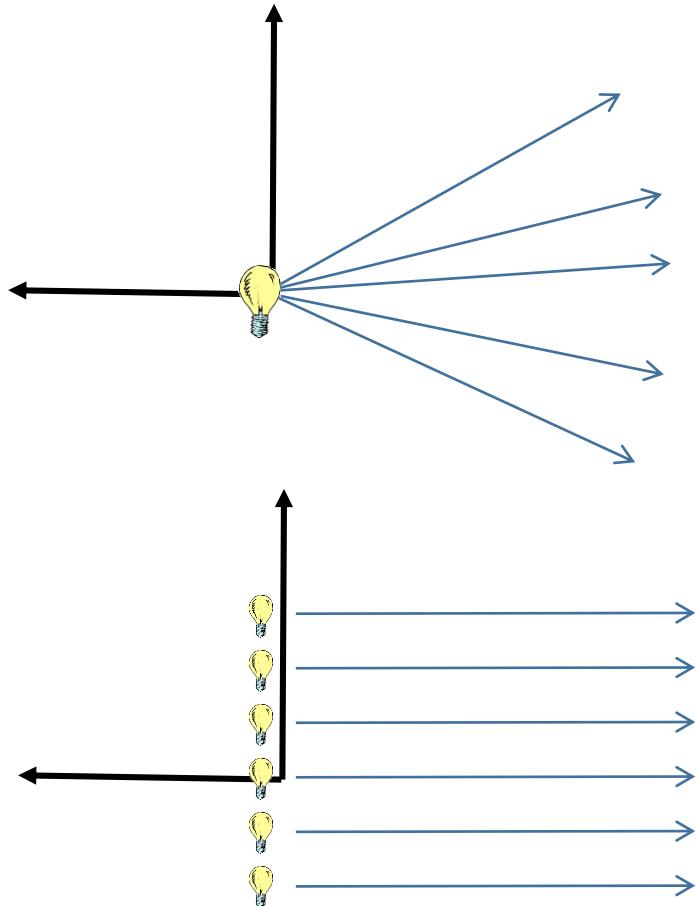
Shadow Mapping (in an ideal world)

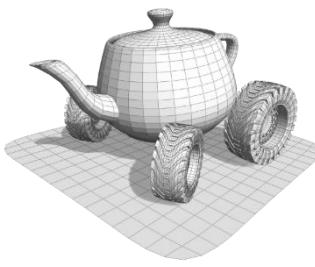




Modelling the light source

- Create a projection matrix so that the projectors match the type of light:
 - Positional (and spotlight) require a **perspective** projection, because all rays leave from a point
 - Directional (that is, very faraway, like the sun) require an **orthogonal** projection, because all rays are parallel





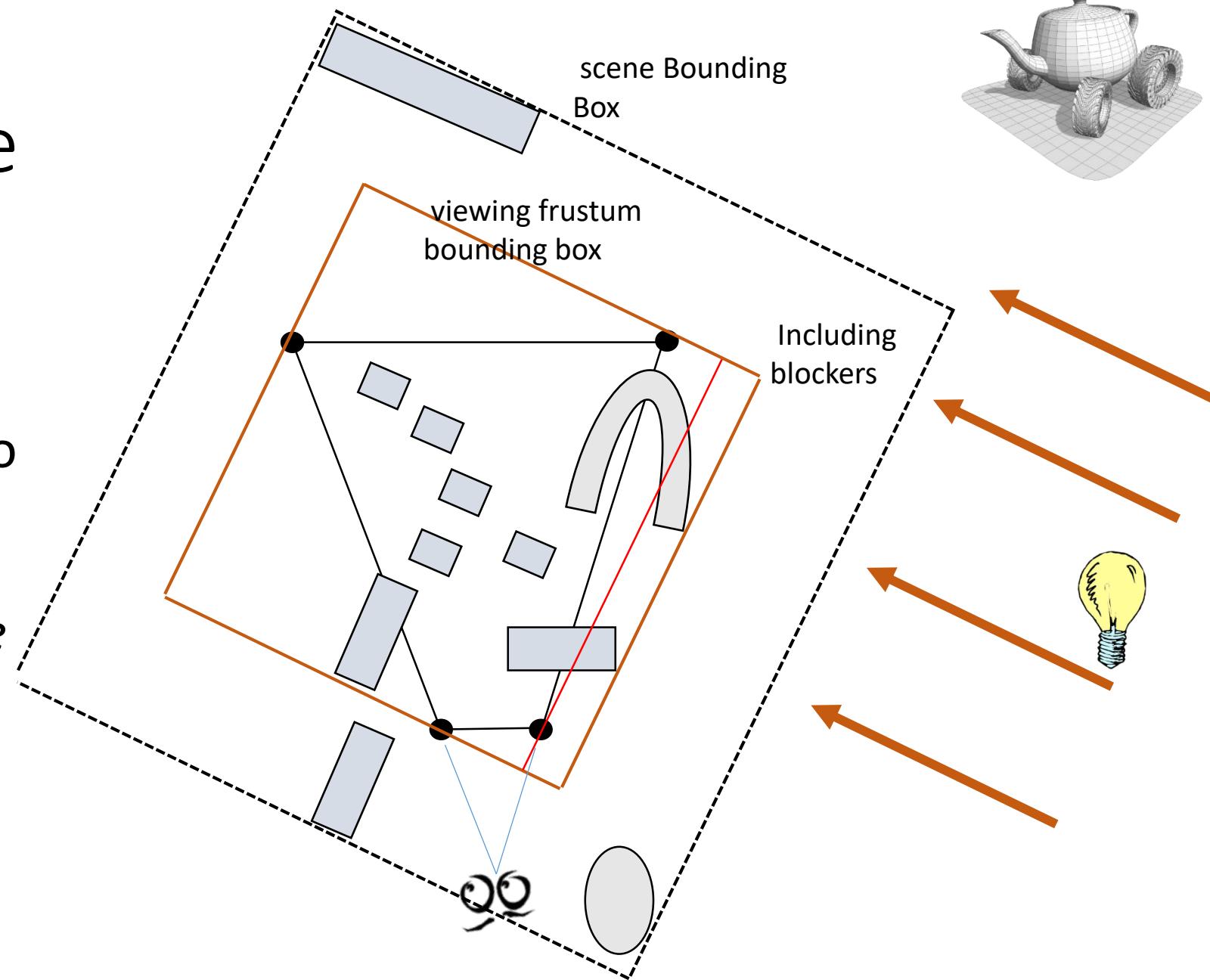
Modelling the light source

- Make sure that the view volume matches the light
 - Positional light: a single perspective matrix may not be enough
 - Spotlight: ensure the view volume encloses the cone of light
 - Directional light: ensure the view volume encloses all the visible scene and the **shadow casters** (that is, the objects that may cast a shadow in the visible scene)



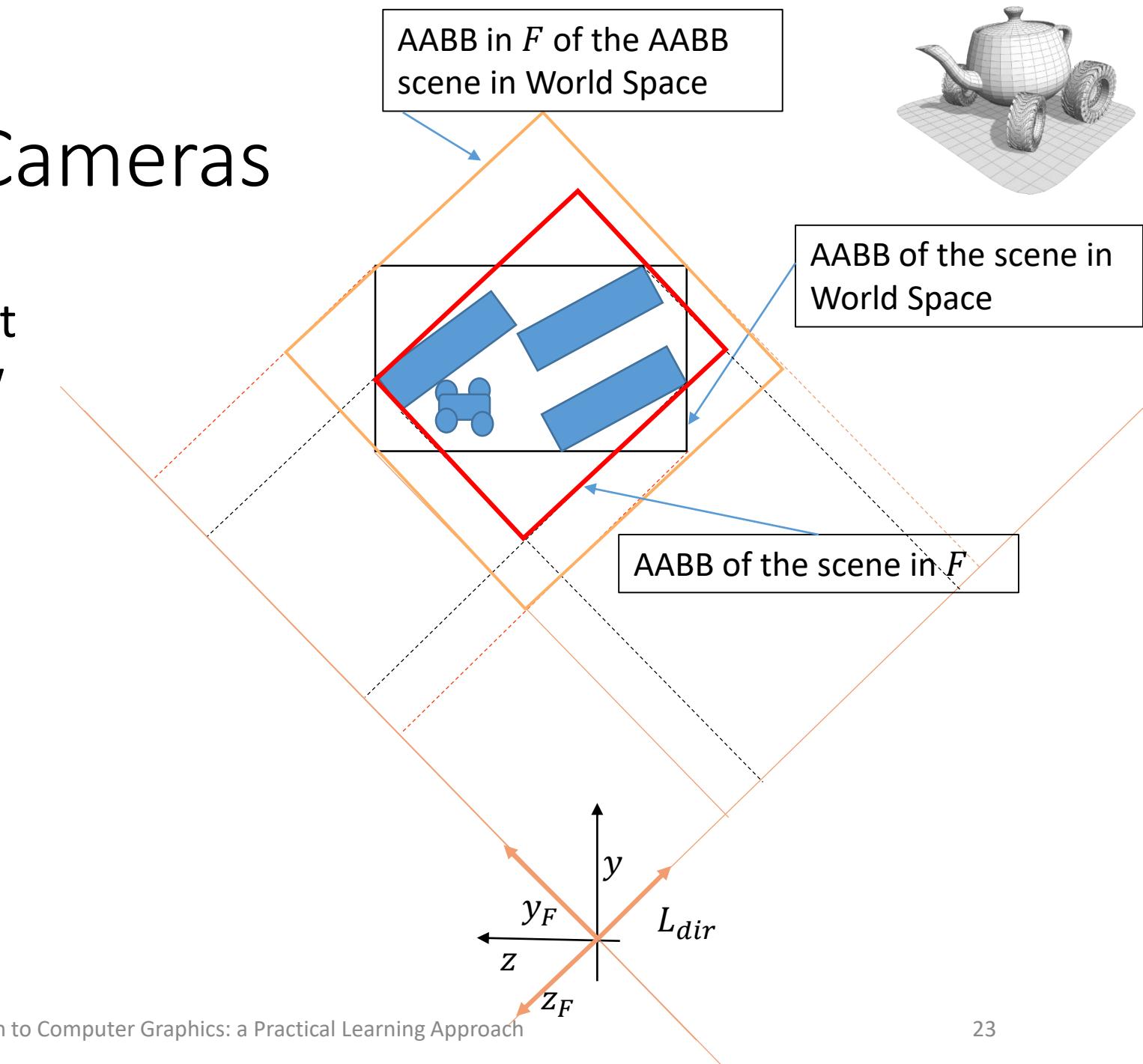
Light View Space

- We do not need a large *viewing volume* for the camera light to include all the scene
- Just include the observer *view volume* extended to include all potential blockers
- It may be not so easy to do in general



Directional Light Cameras

1. Create a frame F such that
 - $z_F = L_{dir}$. F is the view frame
2. Compute the Axis Aligned Bounding Box (AABB) of the scene on frame F
 - Exact, by projecting all the vertices on F
 - Approximated, by projecting the vertices of the AABB of the scene in world space



Directional Light Cameras

3. Set the parameters of the projection matrix as:

$$l = x_m$$

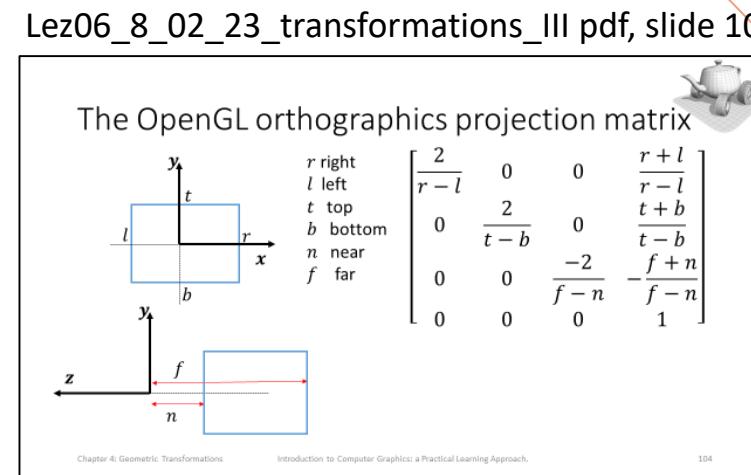
$$r = x_M$$

$$b = y_m$$

$$t = y_M$$

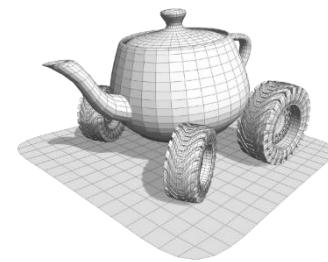
$$n = -z_m$$

$$f = -z_M$$



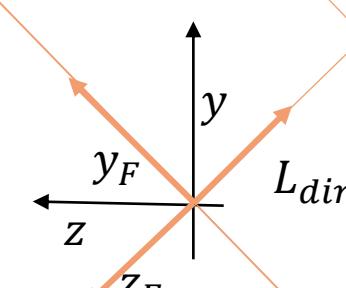
Introduction to Computer Graphics: a Practical Learning Approach

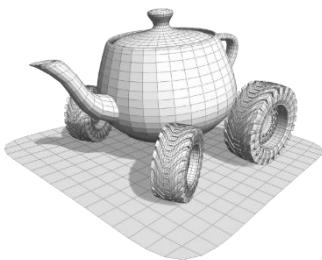
AABB in F of the AABB scene in World Space



AABB of the scene in World Space

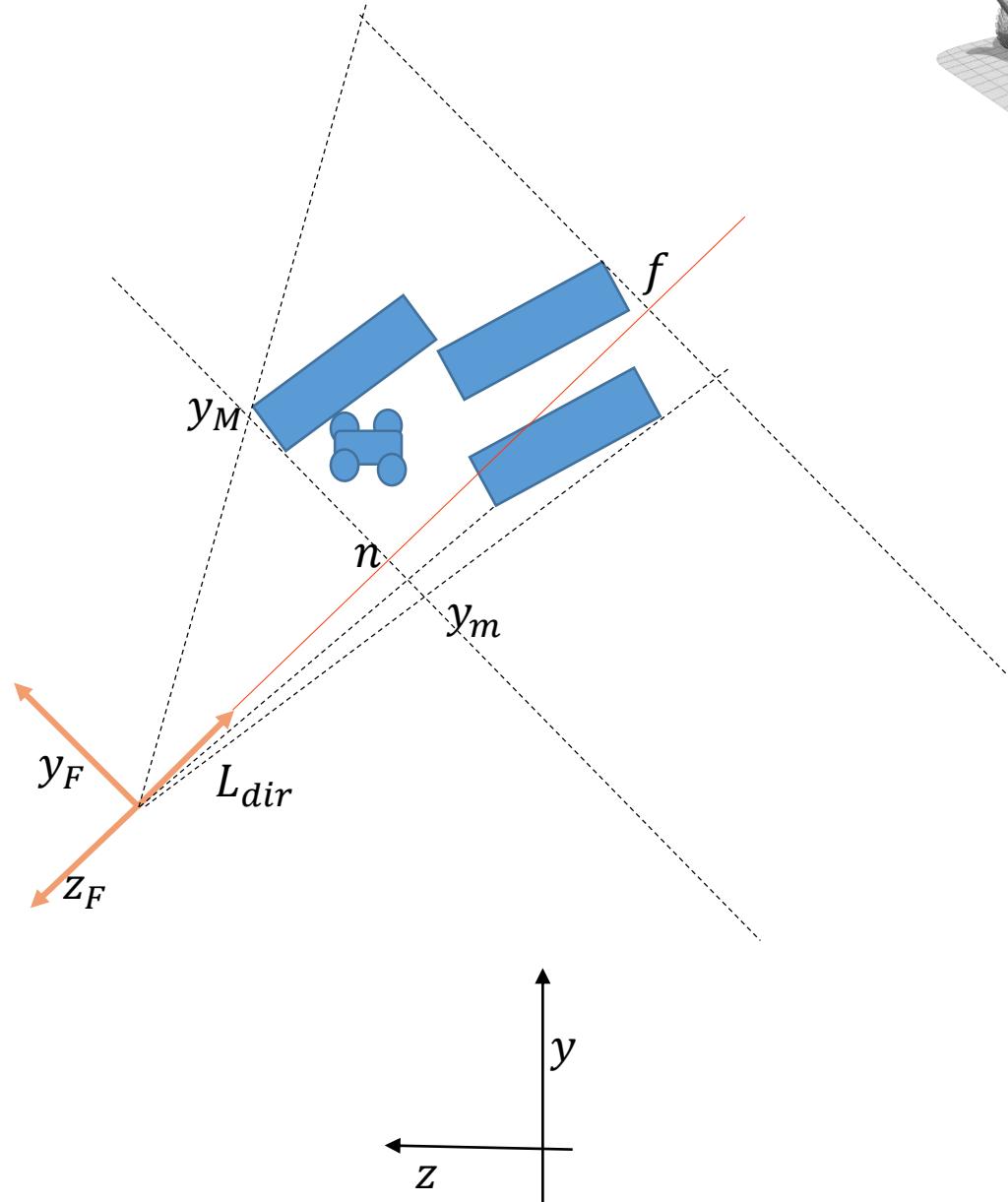
AABB of the scene in F

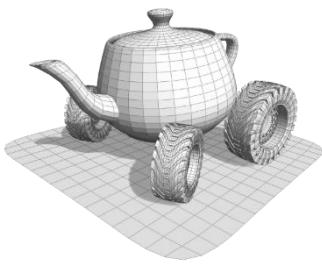




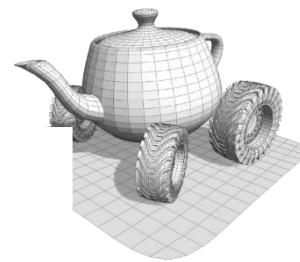
Point Light Cameras

- *Same as the directional lights* but the project the vertices on the camera plane
 1. Project on the z axis and set *near* and *far* planes
 2. Project on the near plane (parallel to the XY plane of F)





- Code: basic shadow mapping



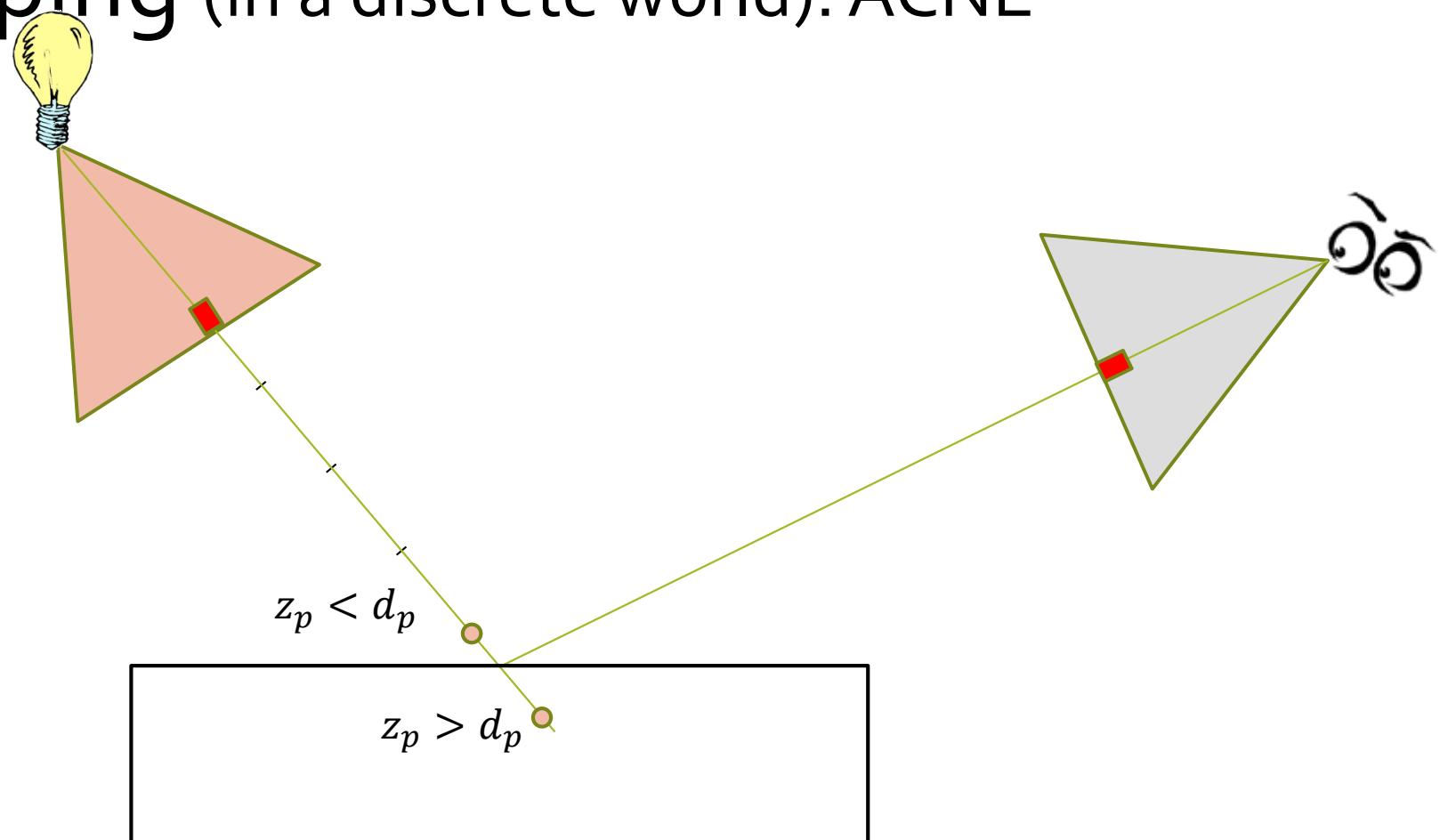
Shadow Mapping (in a discrete world): ACNE

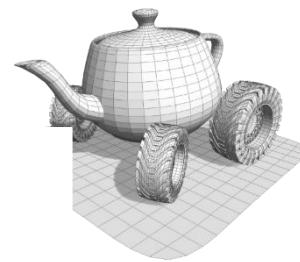
- Finite precision of depth values in the SM
- Rasterization
- For a *lit* point p it may holds either:

$$z_p < d_p$$

or

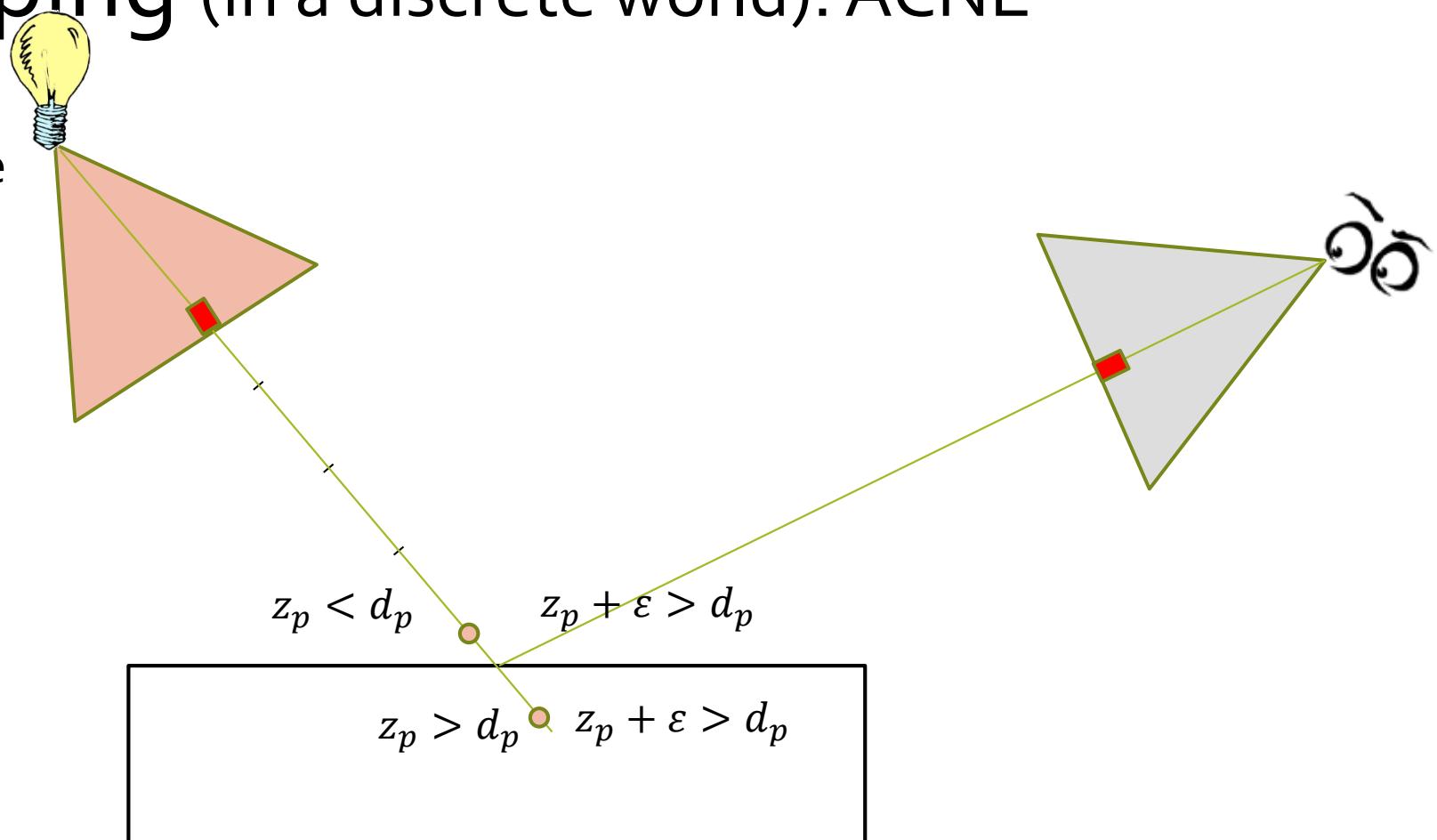
$$z_p > d_p$$

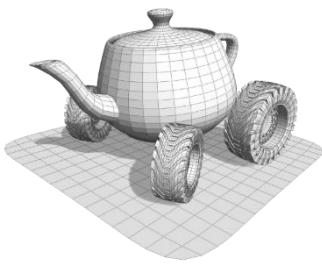




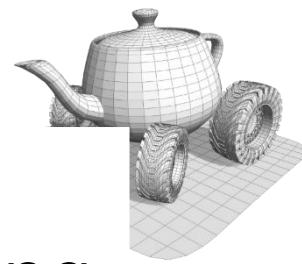
Shadow Mapping (in a discrete world): ACNE

- Solution: offset the value in the shadow map by a constant ε
- If $z_p + \varepsilon > d_p$ then p is lit
- That is, *bias* the test in favour of *being lit*
- How big must ε be ?



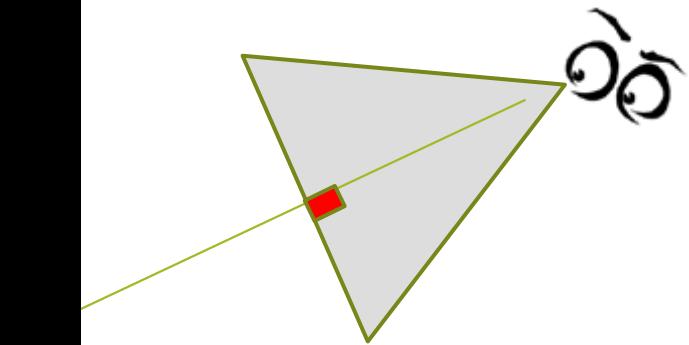
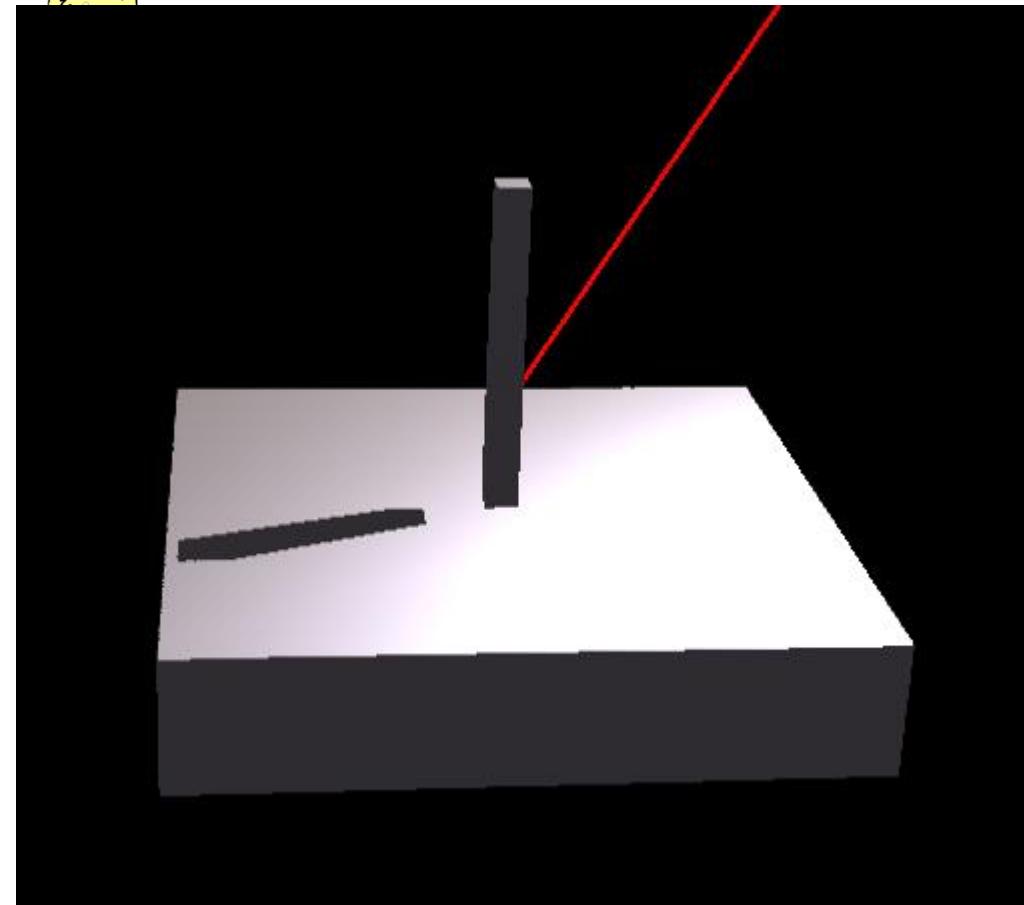


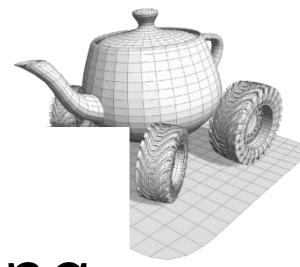
- Code: basic shadow mapping
- Code: bias



Shadow Mapping (in a discrete world): Peterpanning

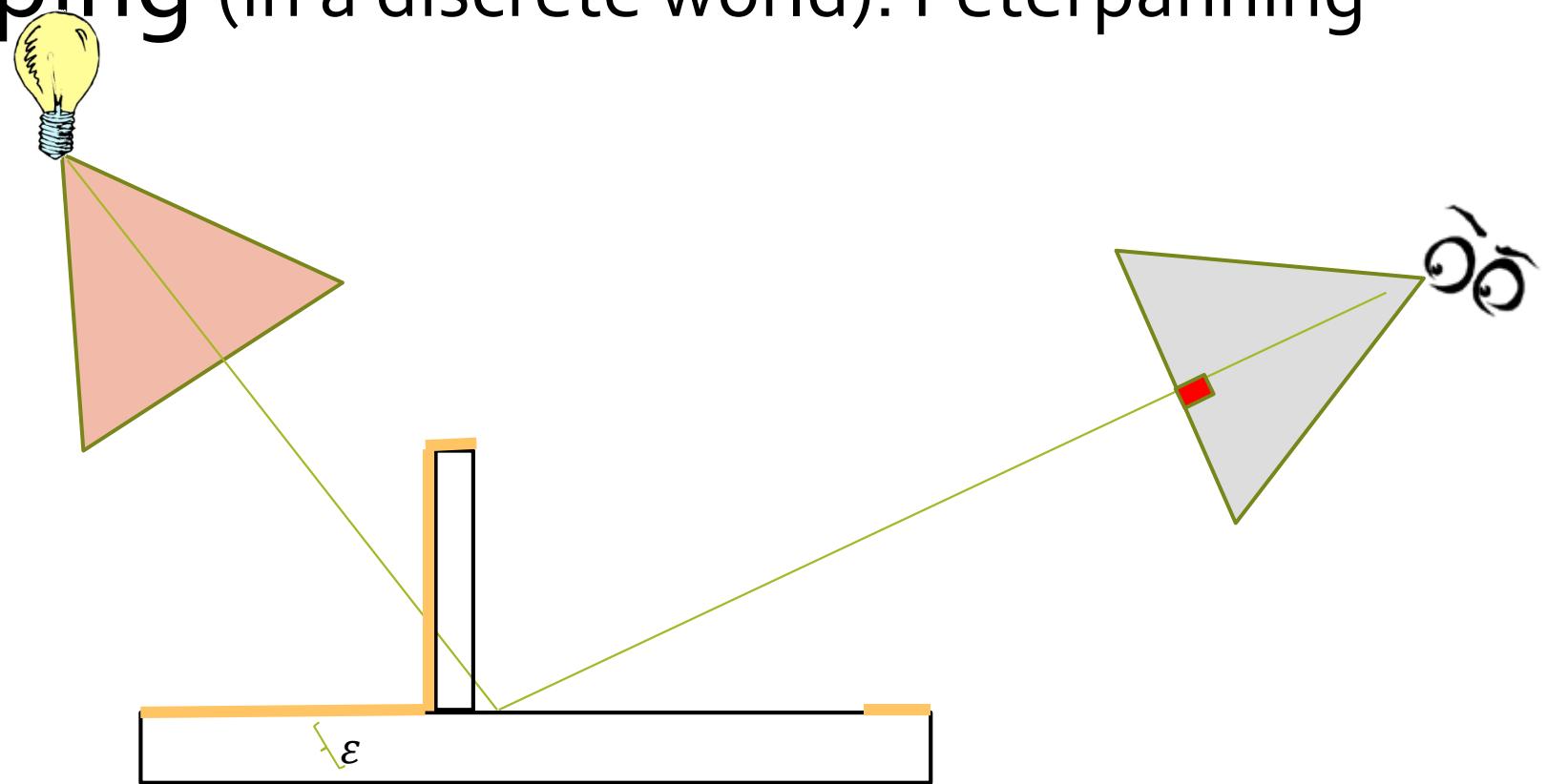
- Big enough to remove self shadowing
- Not so big to create other artifacts
- Q: is it always possible?

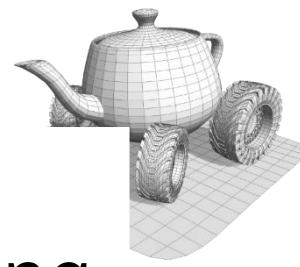




Shadow Mapping (in a discrete world): Peterpanning

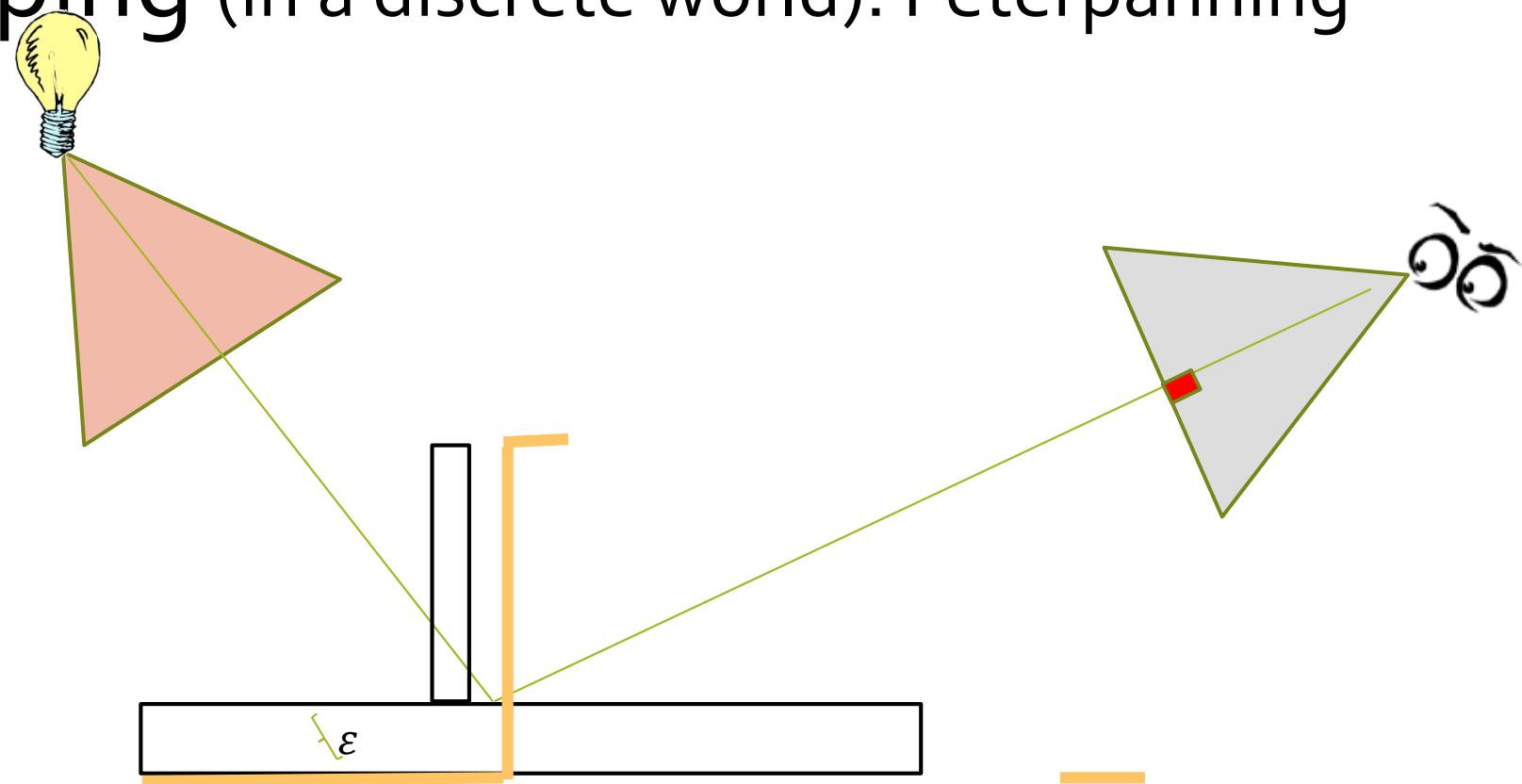
- Big enough to remove self shadowing
- Not so big to create other artifacts
- Q: is it always possible?

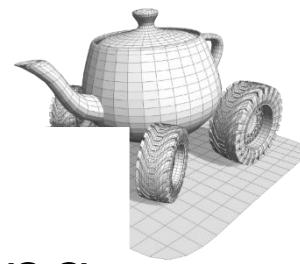




Shadow Mapping (in a discrete world): Peterpanning

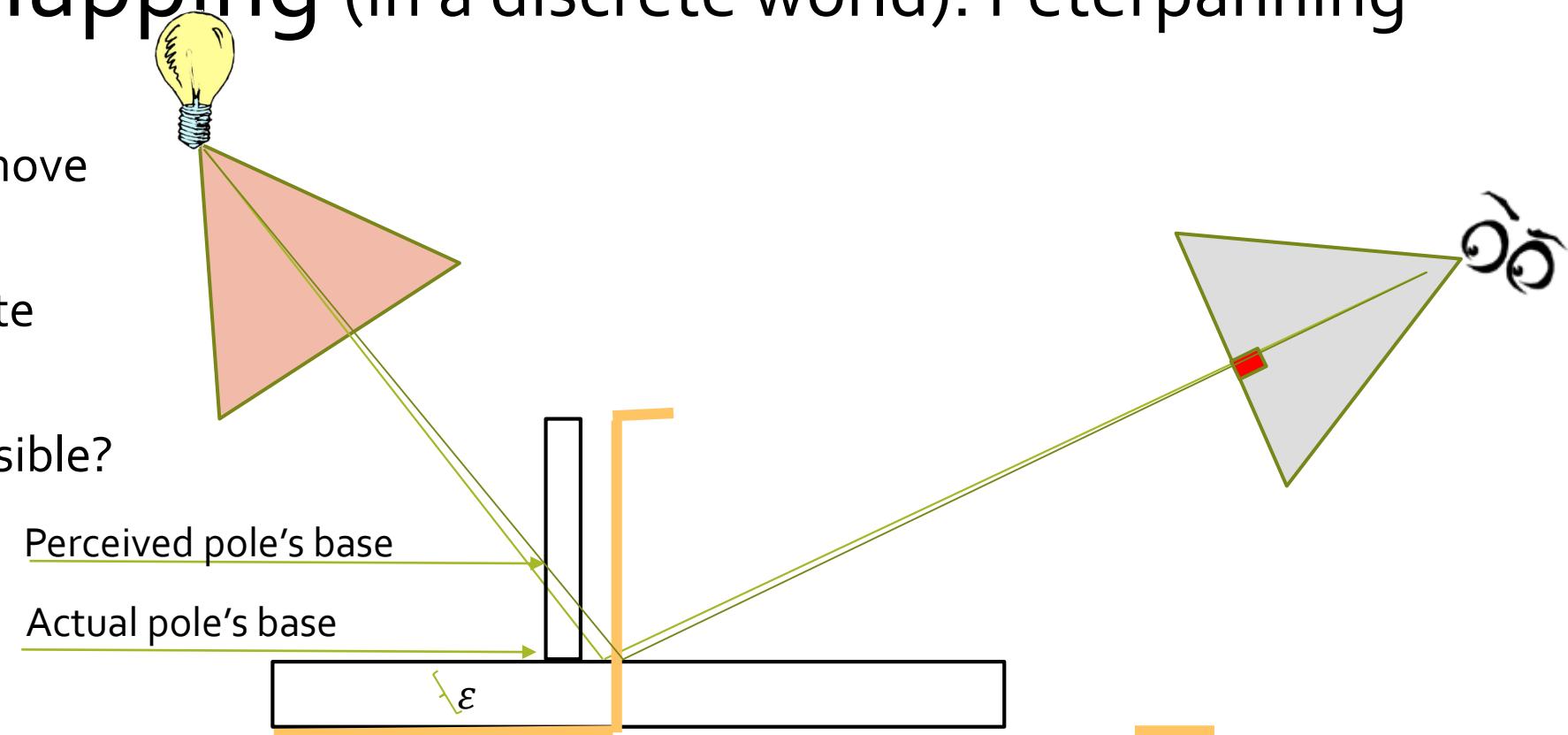
- Big enough to remove self shadowing
- Not so big to create other artifacts
- Q: is it always possible?

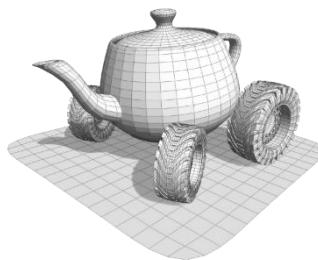




Shadow Mapping (in a discrete world): Peterpanning

- Big enough to remove self shadowing
- Not so big to create other artifacts
- Q: is it always possible?





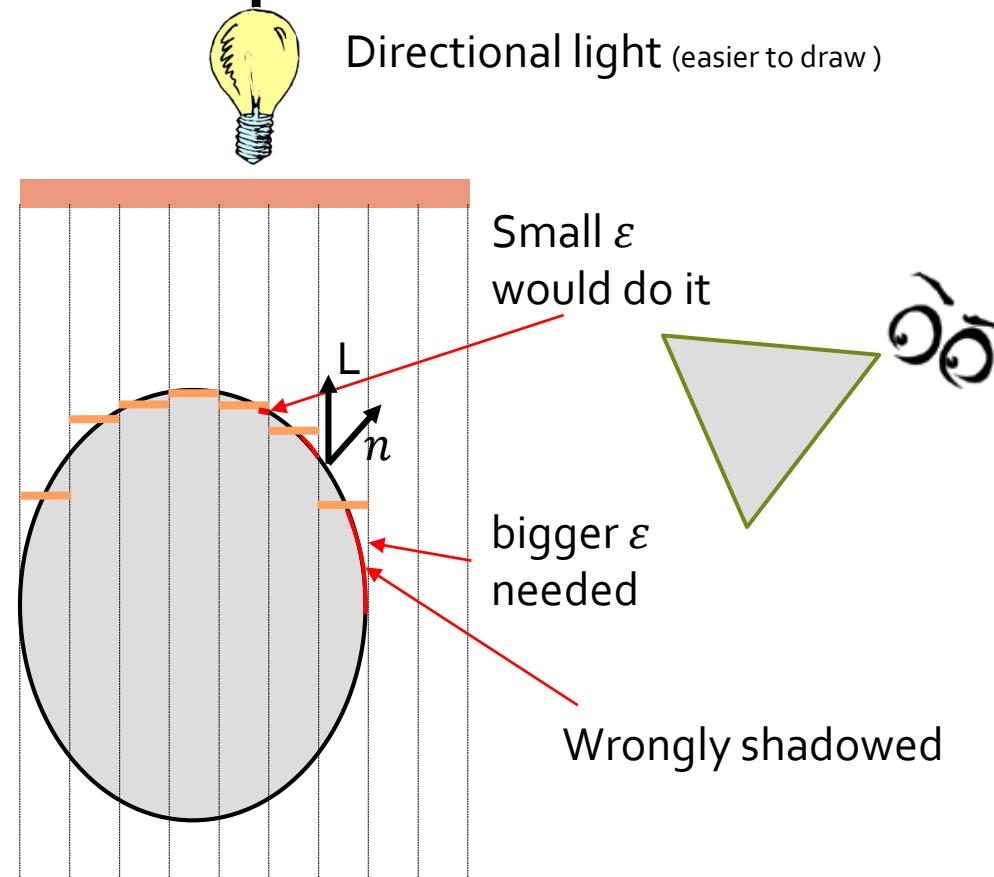
Shadow Mapping:slope dependent bias

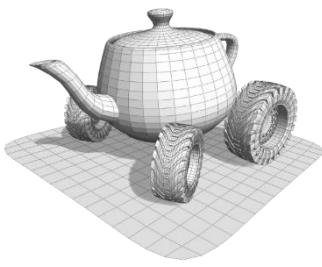
- The higher the polygon slope in *camera space*, the more incorrect self shadowing occurs
- Solution: use a *slope dependent* bias

$$\varepsilon = \text{clamp}(a \tan \arccos(n \cdot L), \min_{\varepsilon}, \max_{\varepsilon})$$

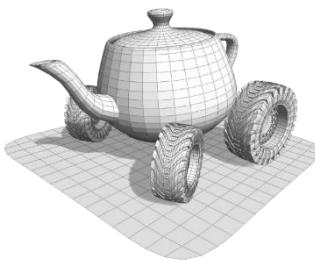
0.005 0.00001 0.01

Fine tuning «good luck» parameters



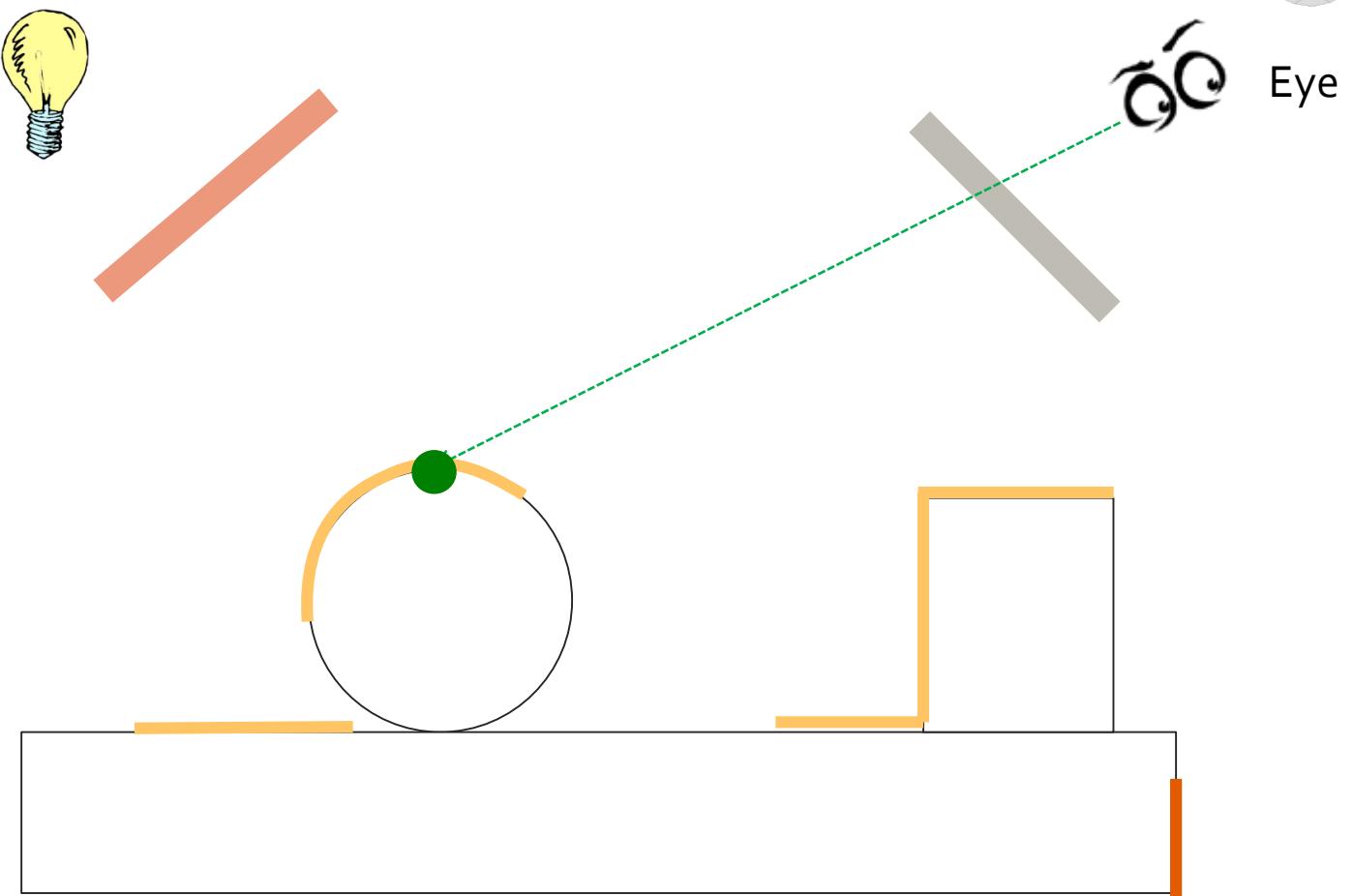


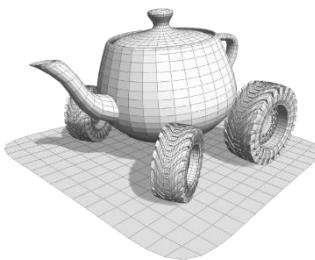
- Code: basic shadow mapping
- Code: bias
- Code: sloped bias



Special case

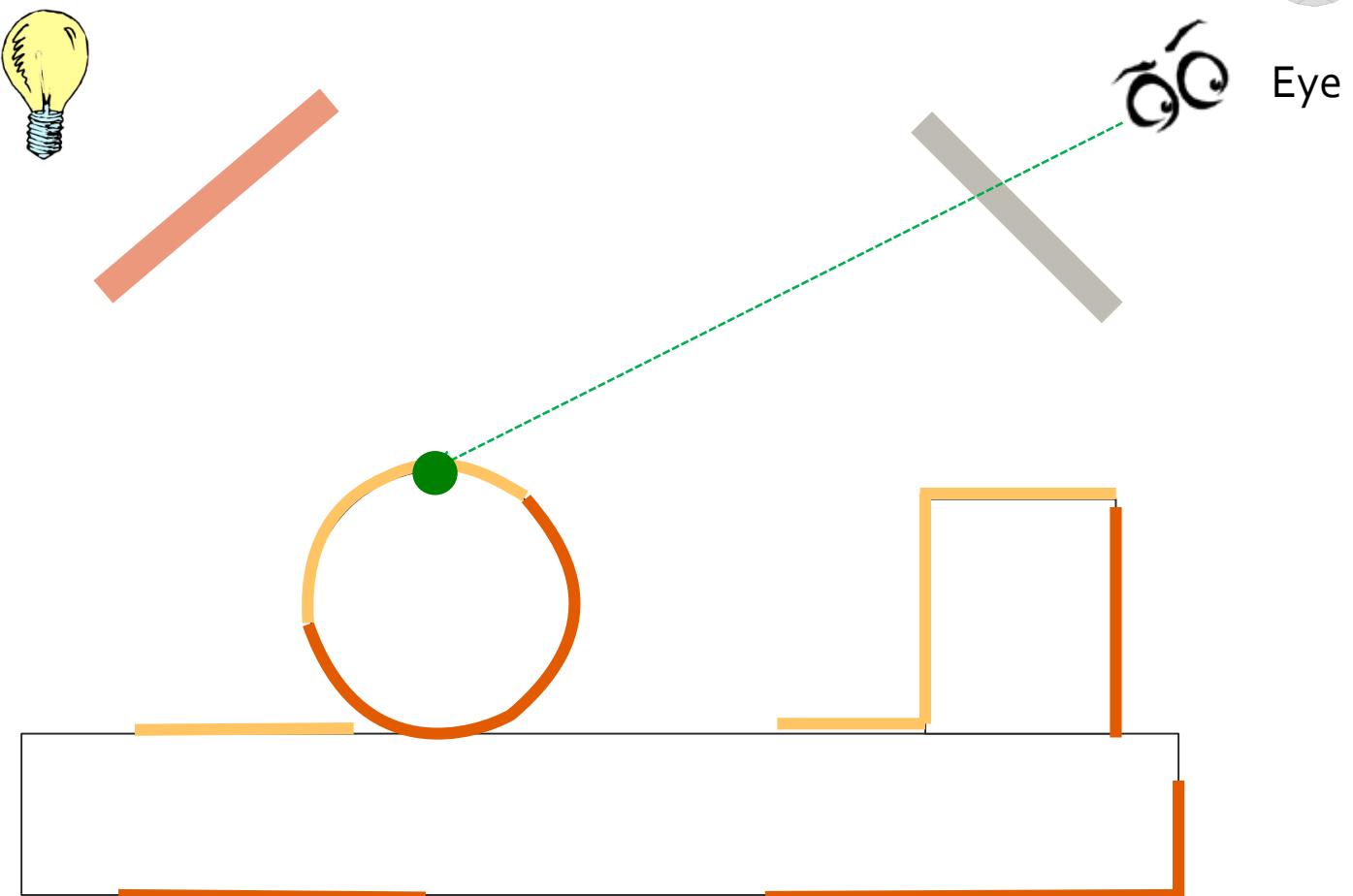
- If blockers are *watertight* (that is, closed surfaces) you can solve the acne problem by rendering *only back faces* in the shadow pass
- And all the acne goes away!

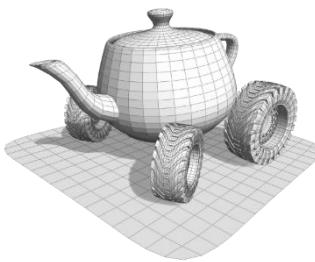




Special case

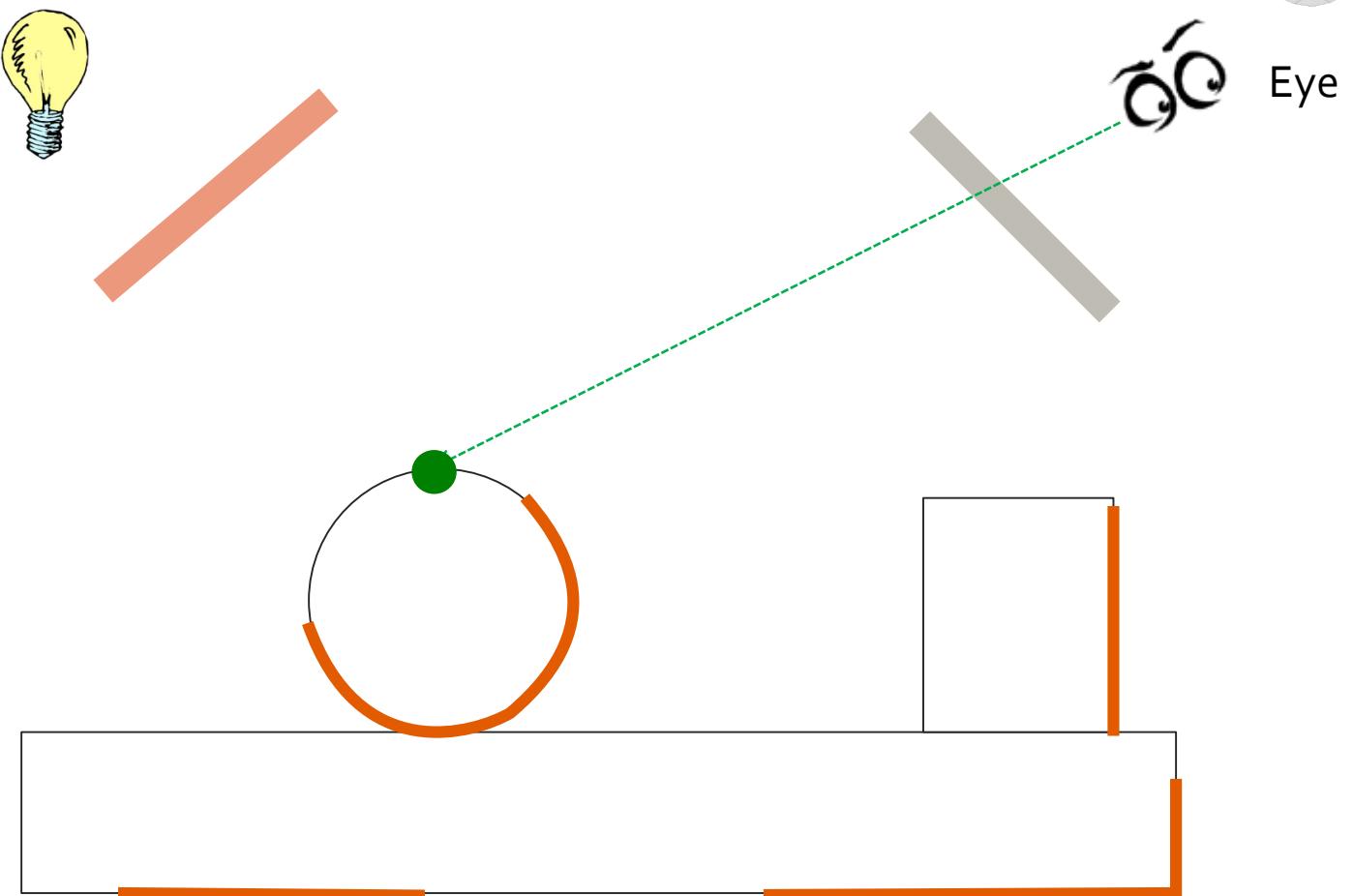
- If blockers are *watertight* (that is, closed surfaces) you can solve the acne problem by rendering *only back faces* in the shadow pass
- And all the acne goes away!

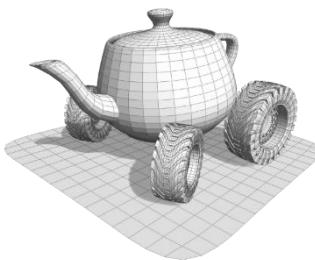




Special case

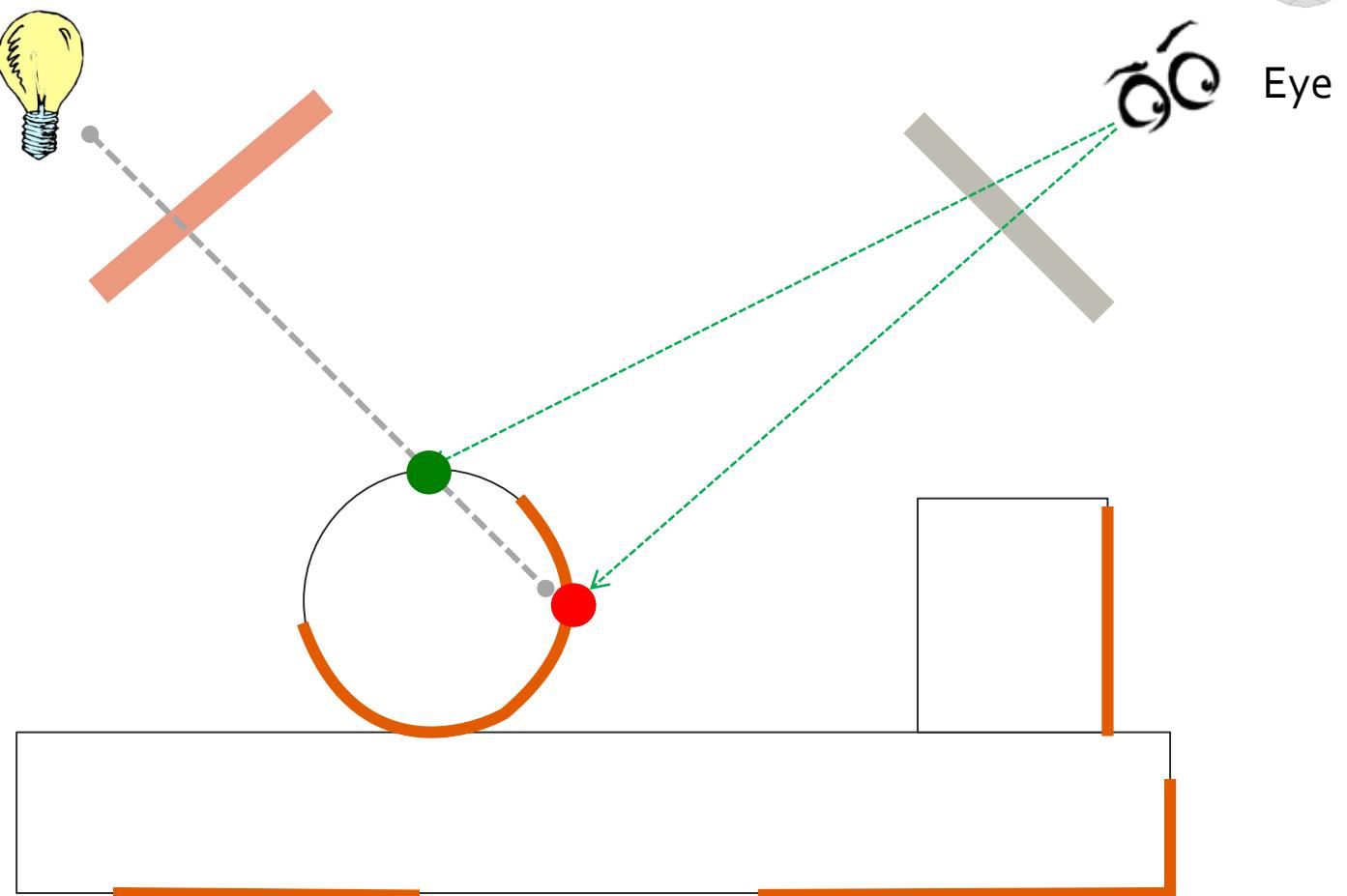
- If blockers are *watertight* (that is, closed surfaces) you can solve the acne problem by rendering *only back faces* in the shadow pass
- And all the acne goes away!

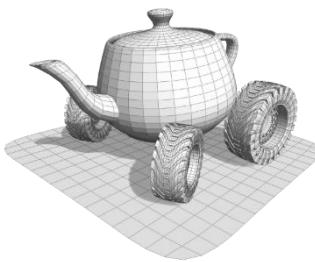




Special case

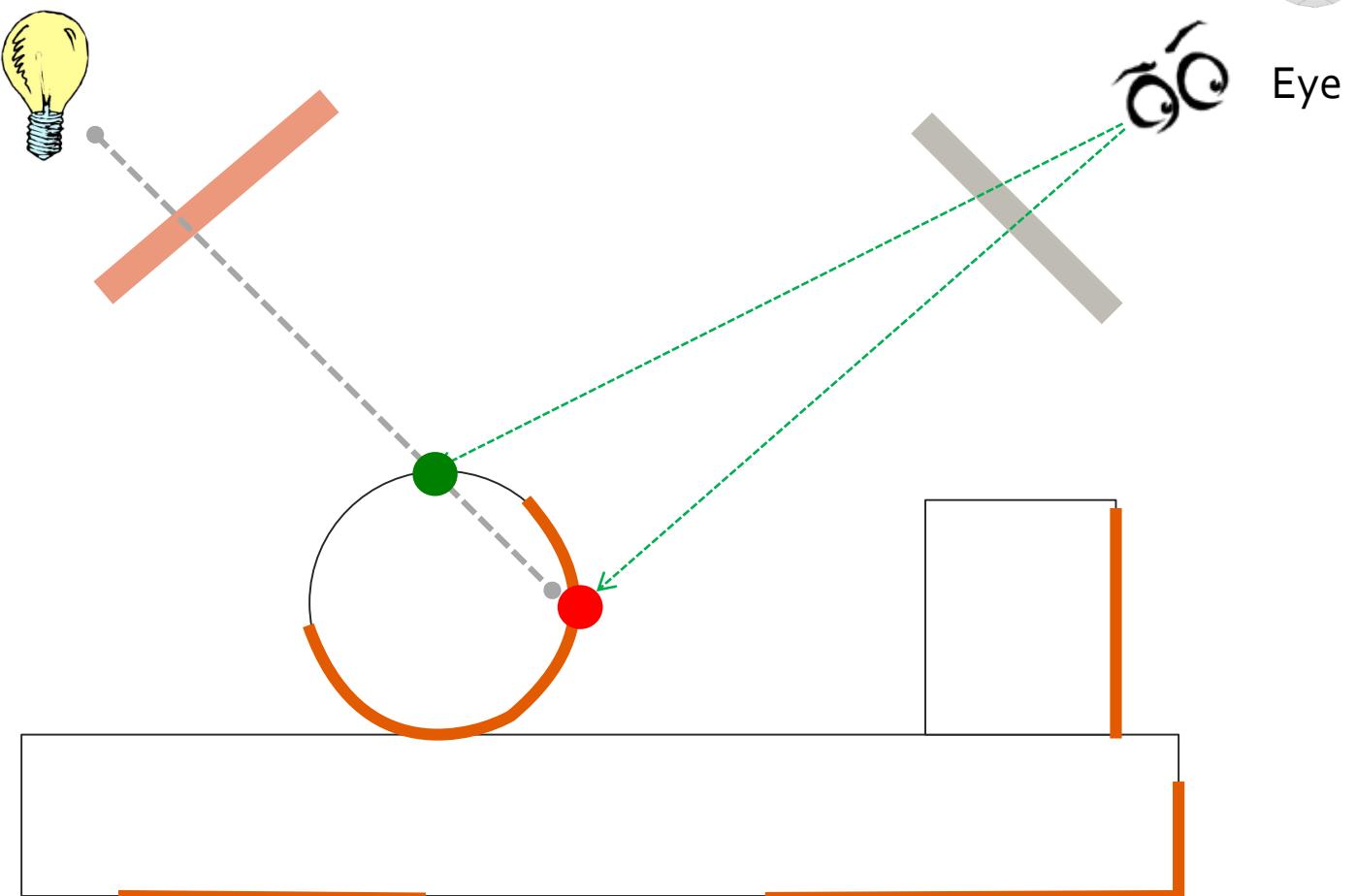
- If blockers are *watertight* (that is, closed surfaces) you can solve the acne problem by rendering *only back faces* in the shadow pass
- And all the acne goes away!
 - Or does it?!

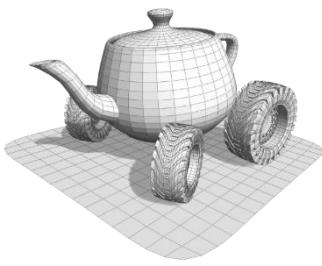




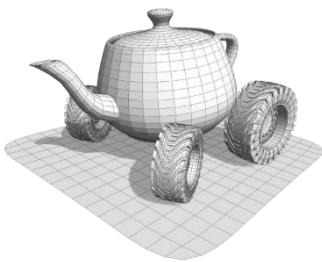
Special case

- If blockers are *watertight* (that is, closed surfaces) you can solve the acne problem by rendering *only back faces* in the shadow pass
- And all the acne goes away!
 - Or does it?!
- Check if the normal is away from the light, if it does it's in shadow



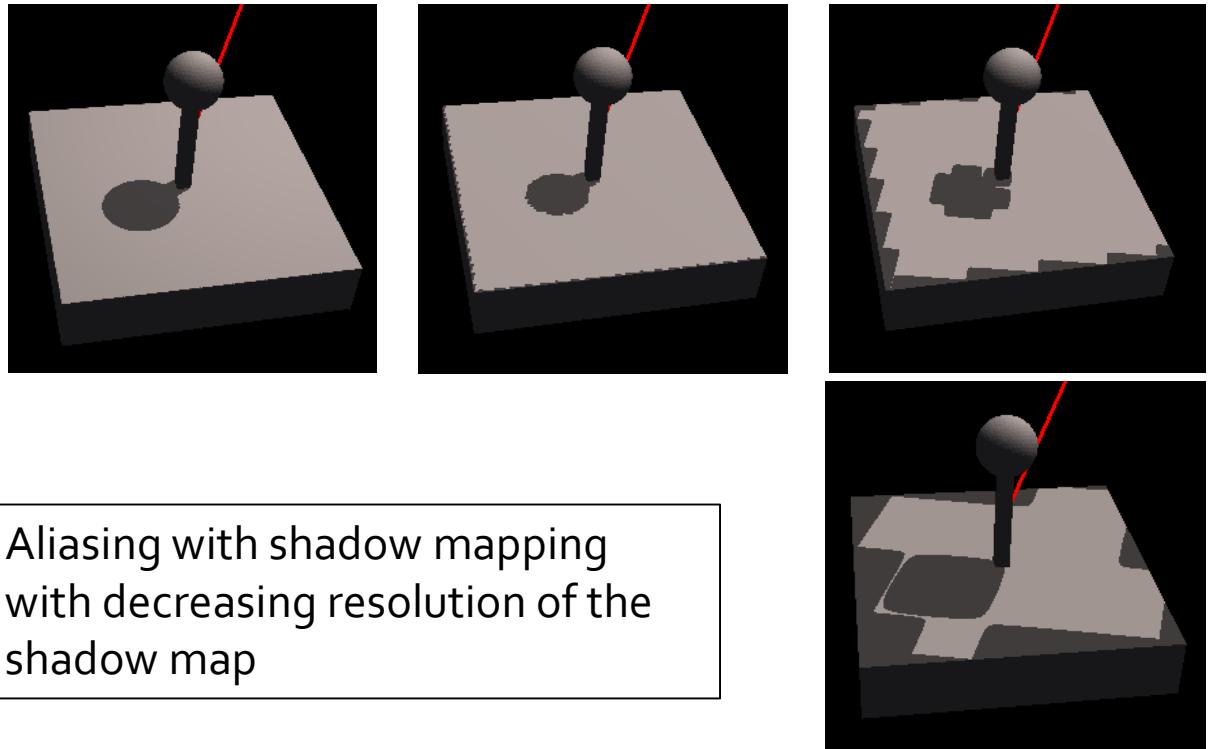


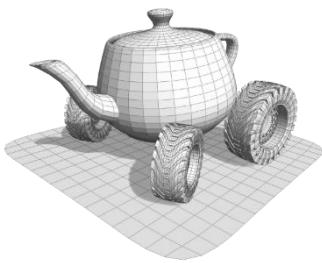
- Code: basic shadow mapping
- Code: bias
- Code: sloped bias
- Code: back faces



Shadow map resolution and aliasing

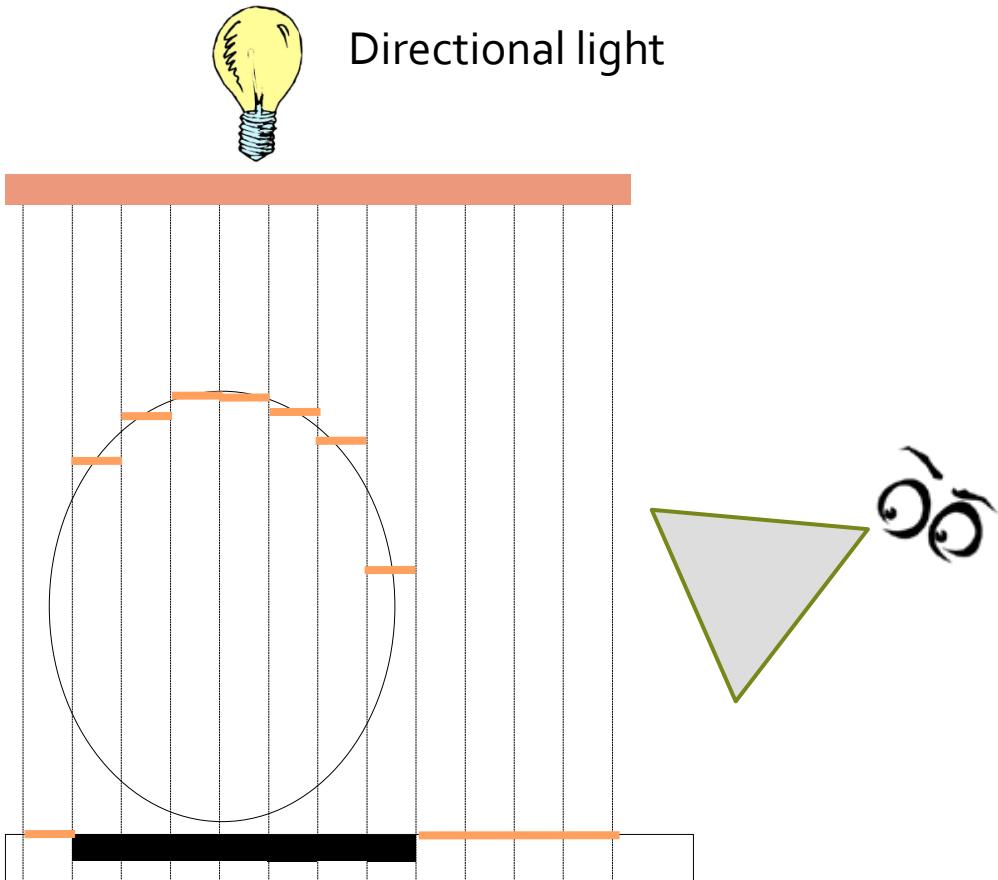
- It's the *texture magnification* problem, that is, one texel projecting over more pixels
 - *Q: can't you just pre filter (blur) the shadow map then? (A: later..)*
- Aliasing makes borders of the shadows look *jagged*

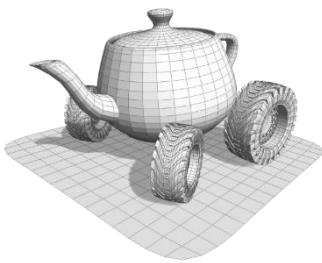




Percentage Closer Filtering

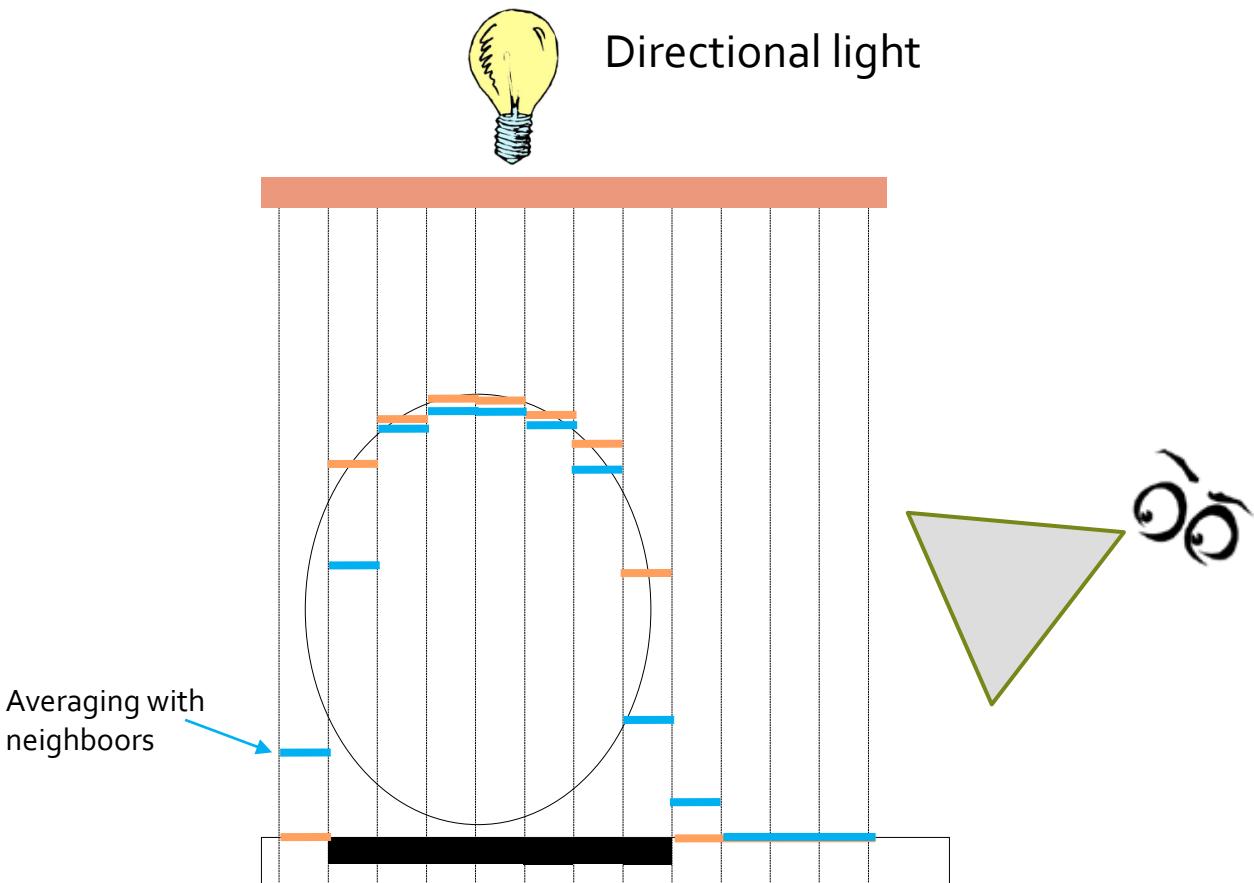
- Prefiltering the shadow map would just create a smooth version of it
- Percentage Closer Filtering: instead of sample one texel, sample the neighborhood and average the results
 - Softer borders, no more on-off result, but a range of values

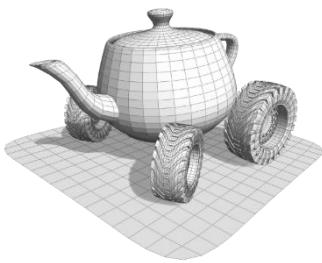




Percentage Closer Filtering

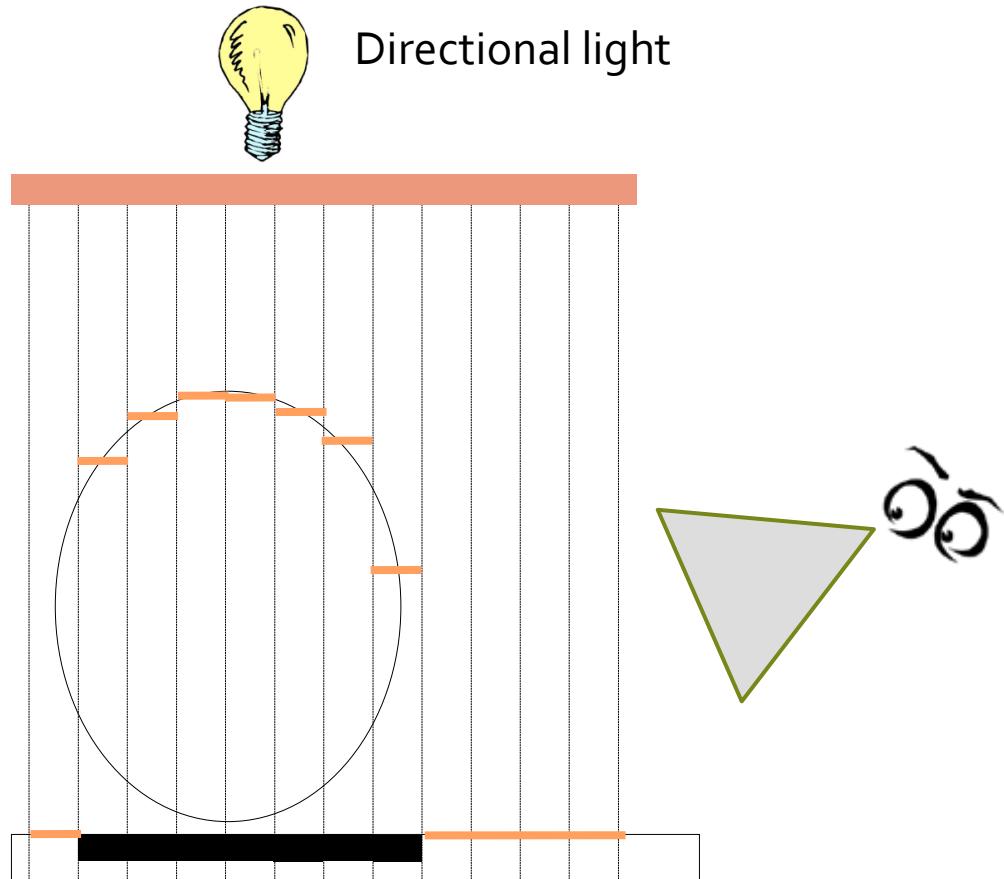
- Prefiltering the shadow map would just create a smooth version of it
- Percentage Closer Filtering: instead of sample one texel, sample the neighborhood and average the results
 - Softer borders, no more on-off result, but a range of values

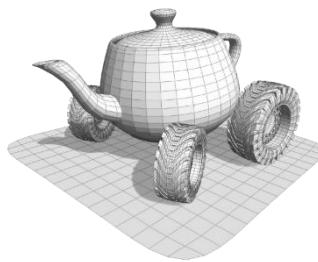




Percentage Closer Filtering

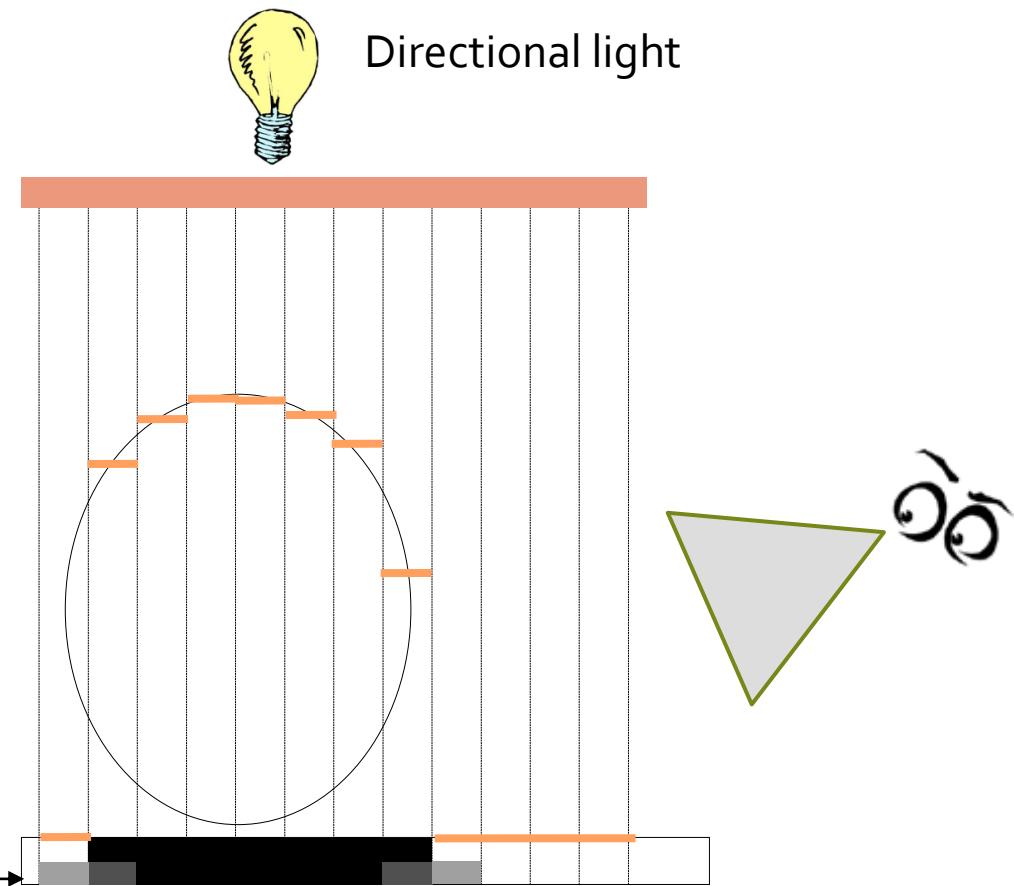
- Prefiltering the shadow map would just create a smooth version of it
- Percentage Closer Filtering: instead of sample one texel, sample the *neighborhood* and average the results
 - Softer borders, no more on-off result, but a range of values

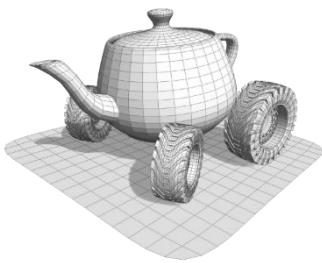




Percentage Closer Filtering

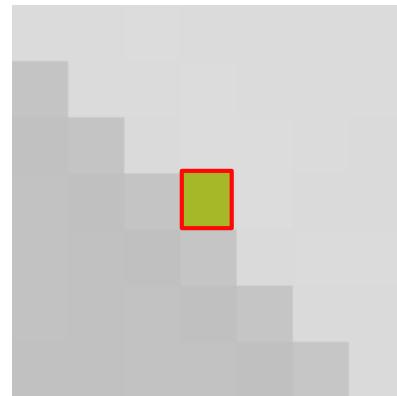
- Prefiltering the shadow map would just create a smooth version of it
- Percentage Closer Filtering: instead of sample one texel, sample the *neighborhood* and average the results
 - Softer borders, no more on-off result, but a range of values



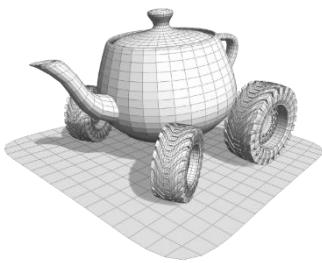


Percentage Closer Filtering

- PCF computes the share of depth values in the kernel that are smaller than the reference value z_r

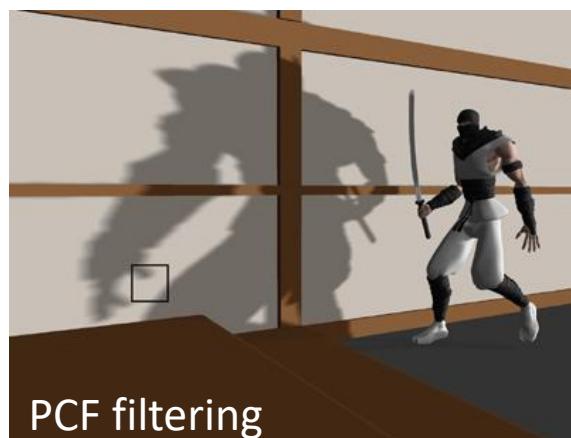
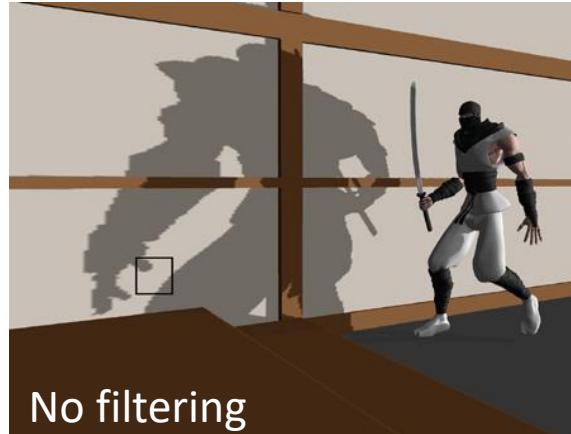


$$lit = 1 - \frac{1}{\#K} \sum_{ij \in K} z_{ij} < z_r) = \frac{1}{\#K} \sum_{ij \in K} z_{ij} > z_r)$$



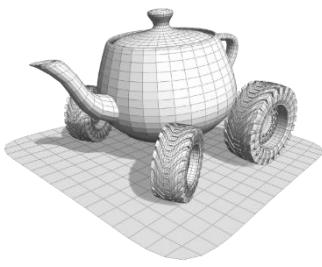
PCF: results

Credit: Nvidia

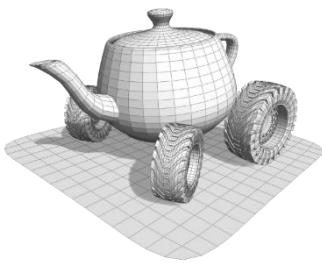


viewPass Fragment shader

```
uniform sampler uShadowMap;
varying vec4 vPos_LS;
void main(void) {
    vec4 color;
    vec3 d_pos = vPos_LS.xyz / vPos_LS.w * .5 + .5;
    float z = texture2D(uShadowMap, d_pos.xy).x;
    float in_shadow = 0.0
    for( dx = -1.5; dx <= 1.5; dx = dx + 1.0)
        for( dy = -1.5; dy <= 1.5; dy = dy + 1.0){
            z = texture2D(uShadowMap, d_pos.xy+vec2(dx,dy)).x;
            if( z + bias < d_pos.z)
                in_shadow = in_shadow + 1.0;
        }
    in_shadow = in_shadow / 16.0;
    gl_FragData[0] = computeColorWithDirectLight(in_shadow);
}
```

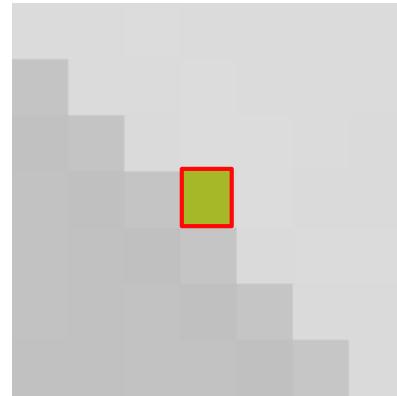


- Code: basic shadow mapping
- Code: bias
- Code: sloped bias
- Code: back faces
- Code: PCF



Percentage Closer Filtering

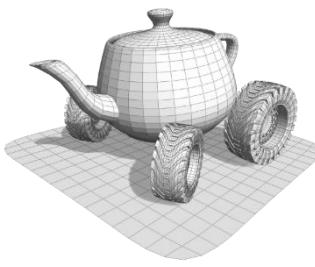
- PCF computes the share of depth values in the kernel that are smaller than the computed value z_r



$$lit = 1 - \frac{1}{\#K} \sum_{ij \in K} z_{ij} < z_r) = \frac{1}{\#K} \sum_{ij \in K} z_{ij} > z_r)$$

- **Note:** We can see it as the probability of any texel z in the kernel to store a depth greater than the reference value

$$\mathbf{P}(z_{hk} > z_r) = lit = \frac{1}{\#K} \sum_{ij \in K} (z_{ij} \geq z_r)$$



Filtering in PCF: Variance Shadow Maps

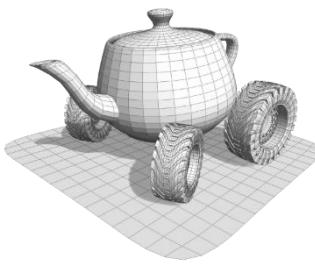
- Key idea of Variance Shadow Map: treat the depth value in the shadow map like *distribution*:

$$\mathbf{P}(z_{hk} > z_r) = lit = \frac{1}{\#K} \sum_{ij \in K} (z_{ij} \geq z_r)$$

Probability of a texel in the kernel to have a value greater than z

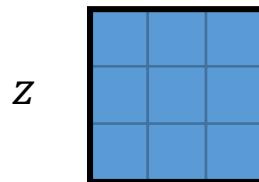
Percentage Closer Filtering

- The distribution is not known but its first and second *moments* can be approximated

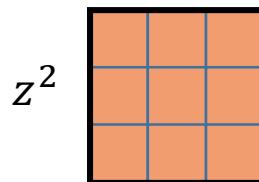


Variance Shadow Maps

- When rendering from the light camera, write 2 maps
 - One for the depth z
 - One for the square of the depth z^2



Averaging over the kernel gives $E(z)$

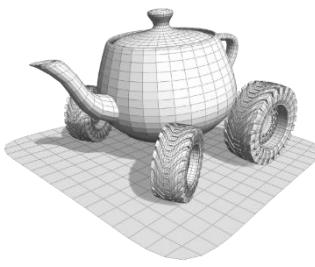


Averaging over the kernel gives $E(z^2)$

- So we can compute *mean* and *variance* of the distribution

$$\mu = E(z)$$

$$\sigma^2 = E(z^2) - E(z)^2$$



Variance Shadow Mapping

- A nice result from calculus [**Chebyshev's inequality**] says that

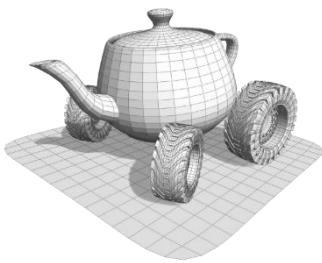
If $z_r > \mu$:

$$\mathbf{P}(z_{hk} > z_r) \leq \frac{\sigma^2}{\sigma^2 + (z_r - \mu)^2}$$

Variance

Mean

- That is, the PCF value for a given pixel is bound by a value that we can precompute
- With VSM that upper bound is used **as** the PCF value
 - which means **NO** multiple accesses to the shadow map at rendering time
 - Using the hardware enabled **filtering** of the texture (for example mipmap)



Filtering

- Applying a **filter** to an image consists in replacing each pixel with a weighted sum of the pixels in the neighborhood
- A filter is represented with a (generally) square matrix of weights, named **kernel**. The size of the matrix is named **kernel size**

F:Filter

1	0	-1
2	0	-2
1	0	-1

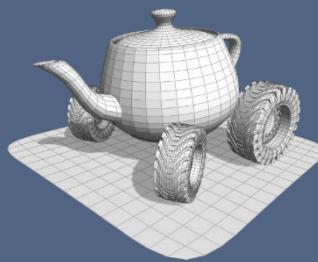
A:Original image

* Kernel size = 1

B: Filtered image

convolution

$$B(i, j) = \sum_{h,k=-N}^N F(h + N, k + N)A(i + h, j + k)$$



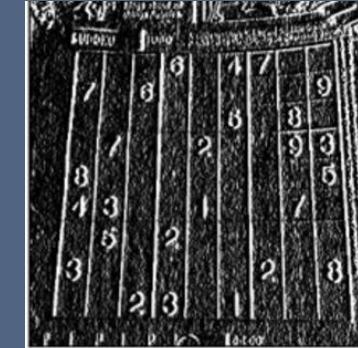
Edge detection: Sobel operator

- The Sobel operator computes the image derivatives along x and y axes, which make the horizontal and vertical edges standout

$$\begin{array}{ccc} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{array}$$



*



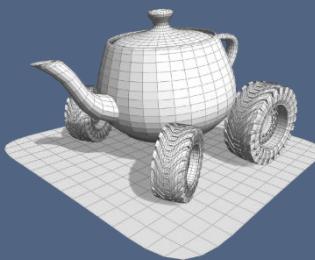
=

$$\begin{array}{ccc} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{array}$$



*

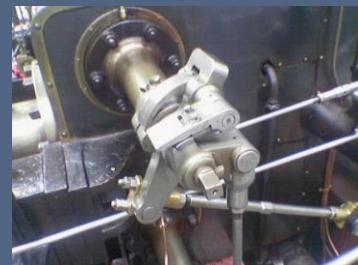




Edge detection: Sobel operator

- The Sobel operator efficiently computes image derivatives along x and y axes, which make the horizontal and vertical edges standout

1	0	-1
2	0	-2
1	0	-1



*

$$= \frac{\partial I}{\partial x}$$

-1	-2	-1
0	0	0
1	2	1

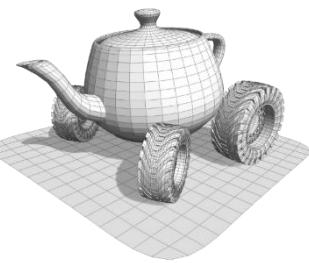


*

$$= \frac{\partial I}{\partial y}$$

$$\sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2} =$$





Filtering

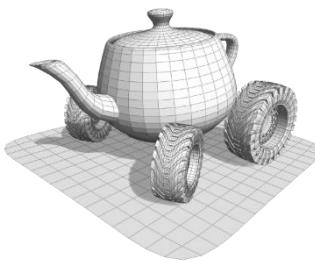
- Example: our Percentage Closer Filtering is a filter with constant weights $\frac{1}{2(N+1)}$ on the depth test result

$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$

*

Shadow Map test results

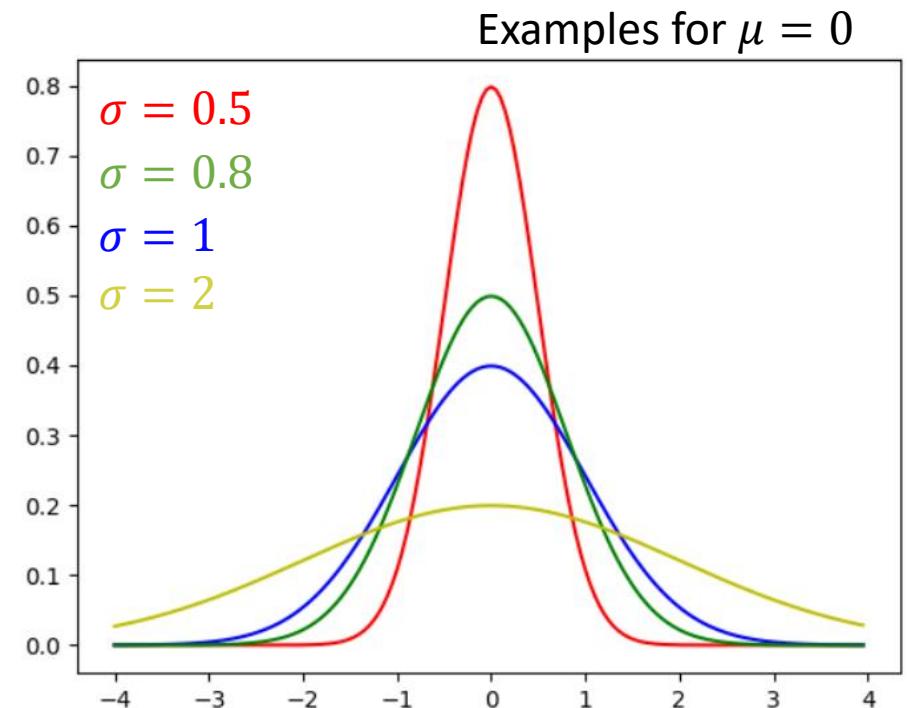
0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0
$z_{ij} > z_r$									

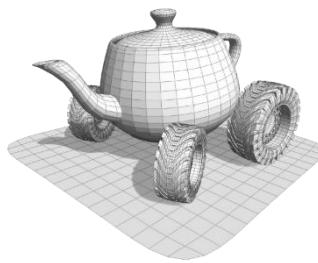


Gaussian Blur

- **Blurring** is used for “smoothing” the image (as PCF does with the shadow)
- **Gaussian Blur:** apply a filter where the weights are computed with the Gaussian Probability Distribution Function
- **Gaussian Distribution**
 - Concentrated around the mean value
 - Smooth decay to 0 at 3 standard deviation distance (3σ)

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)}{\sigma^2}}$$





Gaussian Blur

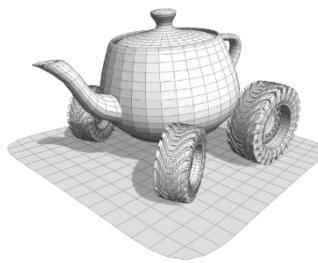
- The 2D kernel for Gaussian Blur is obtained as the product of two Gaussian functions

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{\sigma^2}}$$

$$k(x, y) = f(x)f(y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{(x^2+y^2)}{\sigma^2}}$$

Example for $\sigma = 0.8$ and kernel size = 3

0.00000067	0.00002292	0.00019117	0.00038771	0.00019117	0.00002292	0.00000067
0.00002292	0.00078633	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	0.00019117
0.00038771	0.01330373	0.11098164	0.22508352	0.11098164	0.01330373	0.00038771
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	0.00019117
0.00002292	0.00078633	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00000067	0.00002292	0.00019117	0.00038771	0.00019117	0.00002292	0.00000067



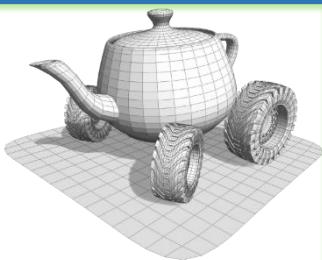
Decomposing Gaussian Blur

- The weights of the 2D Gaussian kernel are obtained as a product of two functions, so we can apply two one dimension kernels
- This reduce the cost of applying a kernel of size N from $(2N+1)^2$ to $2(2N+1)$

$$\text{Blur}(i, j) = \sum_{k=-N}^N \sum_{h=-N}^N f(h + N)f(k + N)I(i + h, j + k) =$$
$$\sum_{k=-N}^N f(k + N) \sum_{h=-N}^N f(h + N)I(i + h, j + k)$$

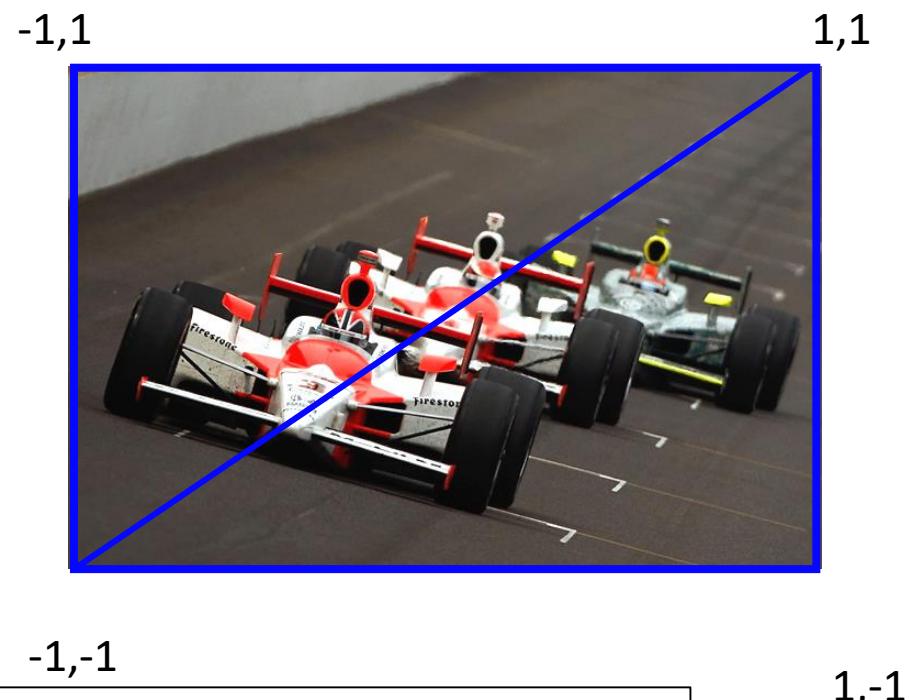



$$1^\circ \text{ pass, sum the pixels in the same row}$$
$$2^\circ \text{ pass, sum the pixels in the same column (of the result of the } 1^\circ \text{ pass)}$$



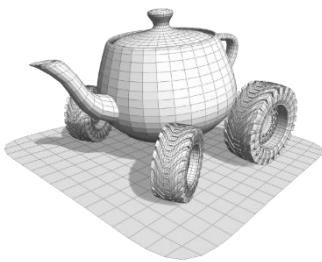
Implementing image filters in *GL

1. Load the image to be filtered onto a 2D texture
2. Set the render to a framebuffer
3. Render a polygon so that it entirely covers the viewport the viewport and with the texture stitched on it
4. Compute the filter result in the fragment shader

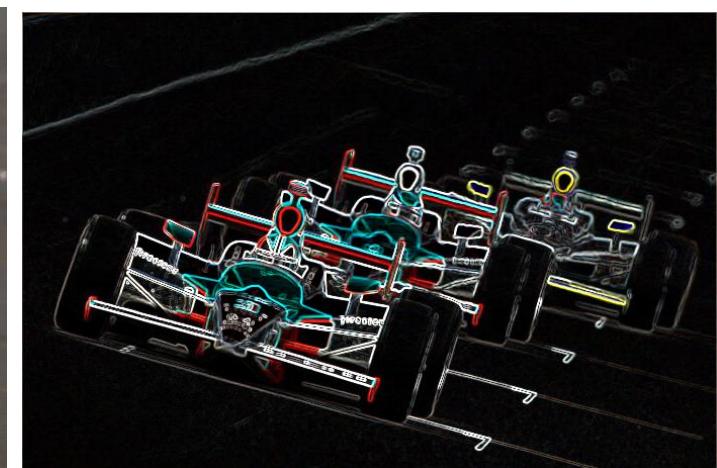


This technique is named “Fullscreen quad” and it will be used again

- For deferred shading
- For ray tracing



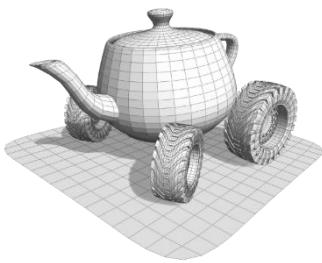
Examples



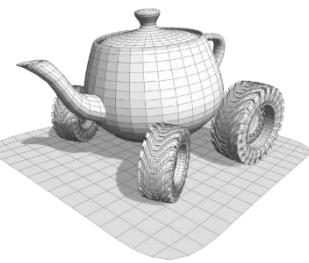
Constant weights (box filter)

Gaussian Blur(box filter)

Sobel Edge Detector

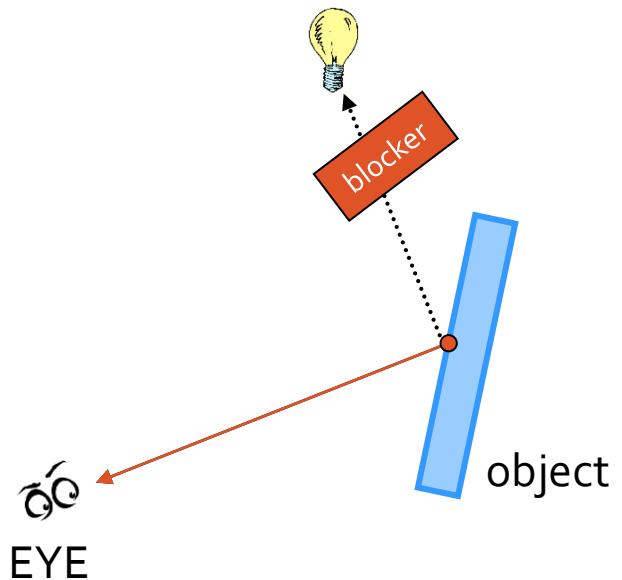


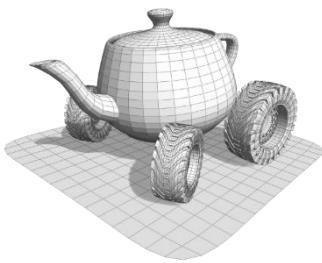
- Code: basic shadow mapping
- Code: bias
- Code: sloped bias
- Code: back faces
- Code: PCF
- Code: VSM



Beyond Shadow Mapping

- Few simple-geometry blockers
 - Pass the object to the shader program and do ray-object intersection in the fragment shader
- Flat receiver
 - Just re-draw the shadow casters projecting their vertices on the flat receiver





Shadows created by «ambient» light

- Let us go back to the equation (Phong + direct shadows)

$$L_o(x, \omega_r) = L_e(x, \omega_r) + k_a + \sum_{\forall i} \text{ReflFunc}(L(x, \omega_i))$$

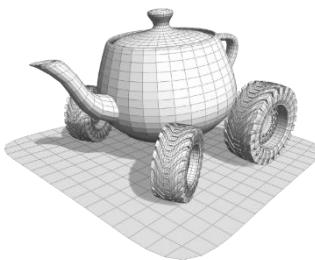
- let us see k_a as the integral of light coming from every direction in the hemispere

$$L_o(x, \omega_r) = L_e(x, \omega_r) + k_a + \sum_{\forall i} \text{ReflFunc}(L(x, \omega_i))$$

$$k_a = \int_{\Omega} L_a(\omega_i) d\omega_i$$

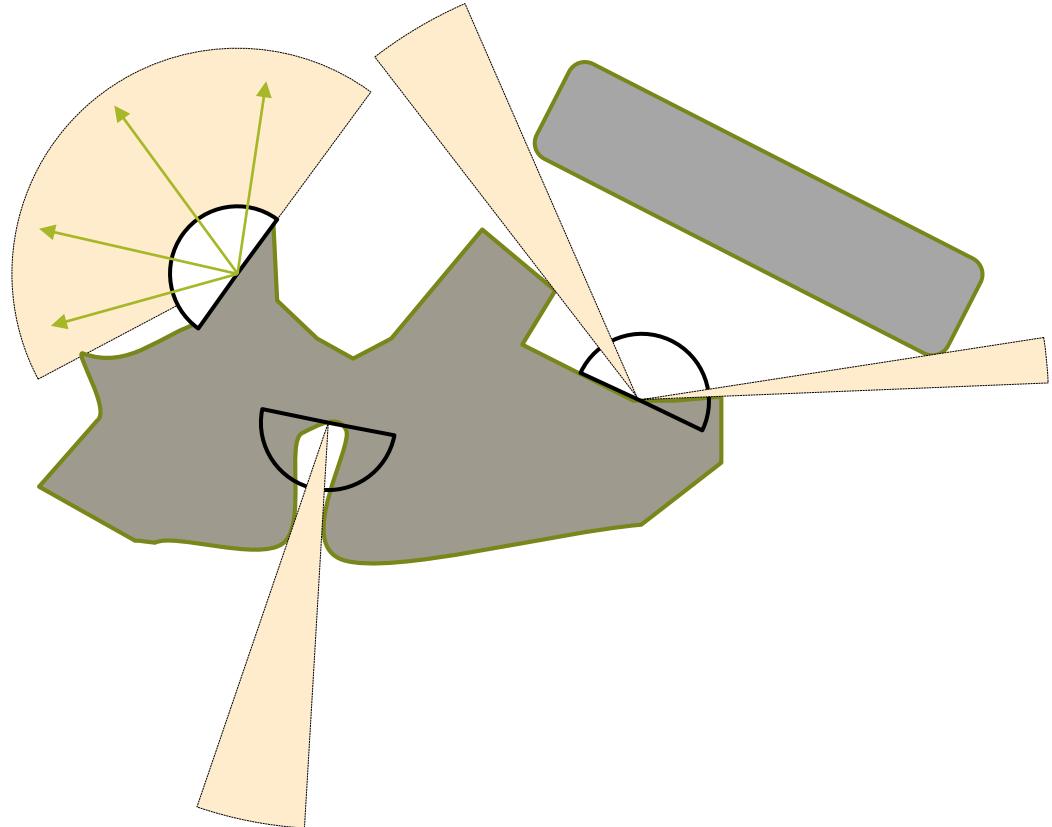
- Ambient Obscurrence** (or Occlusion): and add a shadow scaling term based on the blockers that can prevent light from reaching point x

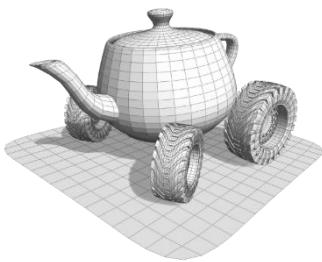
$$k_a = \int_{\Omega} L_a(\omega_i) \text{isBlocked}(x, \omega_i) d\omega_i$$



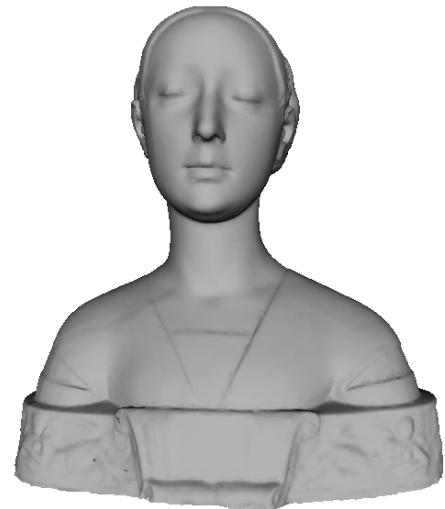
Ambient Occlusion/Obscurrence

1. For each surface point x , shoot a ray along every direction on the hemisphere
2. Count *how many* rays intersect some part of the scene
3. Store the AO value **per vertex** (for dense tessellation) or in a **texture** (for parametrized models)
4. Scale the ambient term by the ratio of intersecting rays
 - **Not good for rasterization pipeline**
 - **Only static scenes/rigid objects**

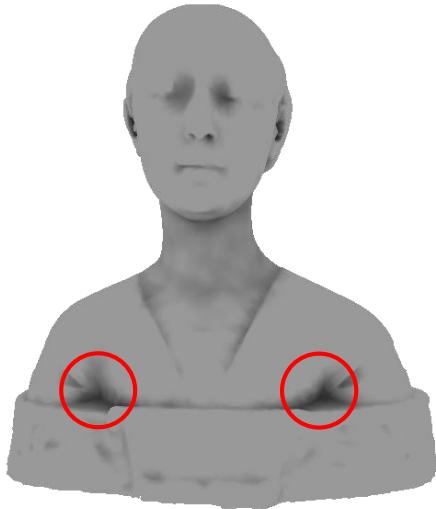




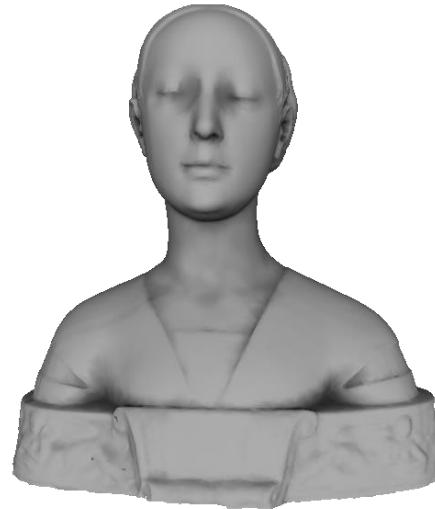
Ambient Occlusion: example



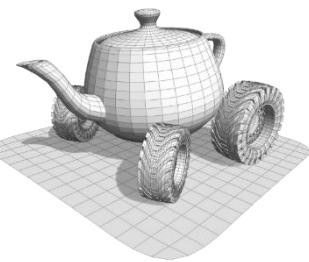
Just Phong Lighting



Only Ambient Occlusion term



Phong + AO



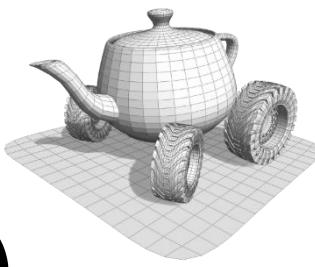
Ambient Occlusion: another example



Just Phong Lighting

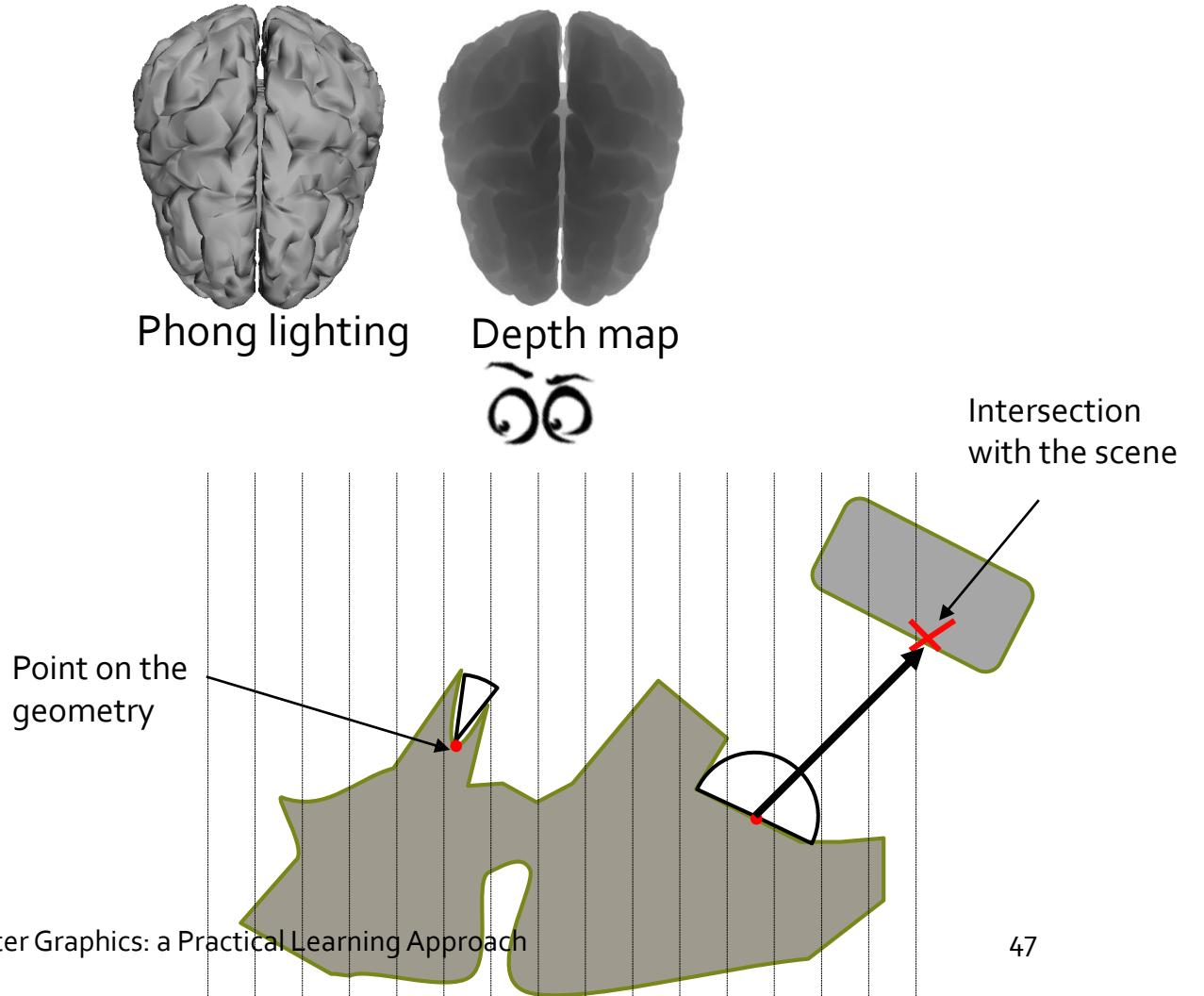


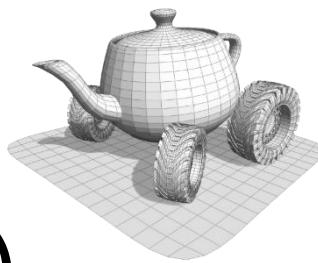
Phong + AO



Screen Space Ambient Occlusion (SSAO)

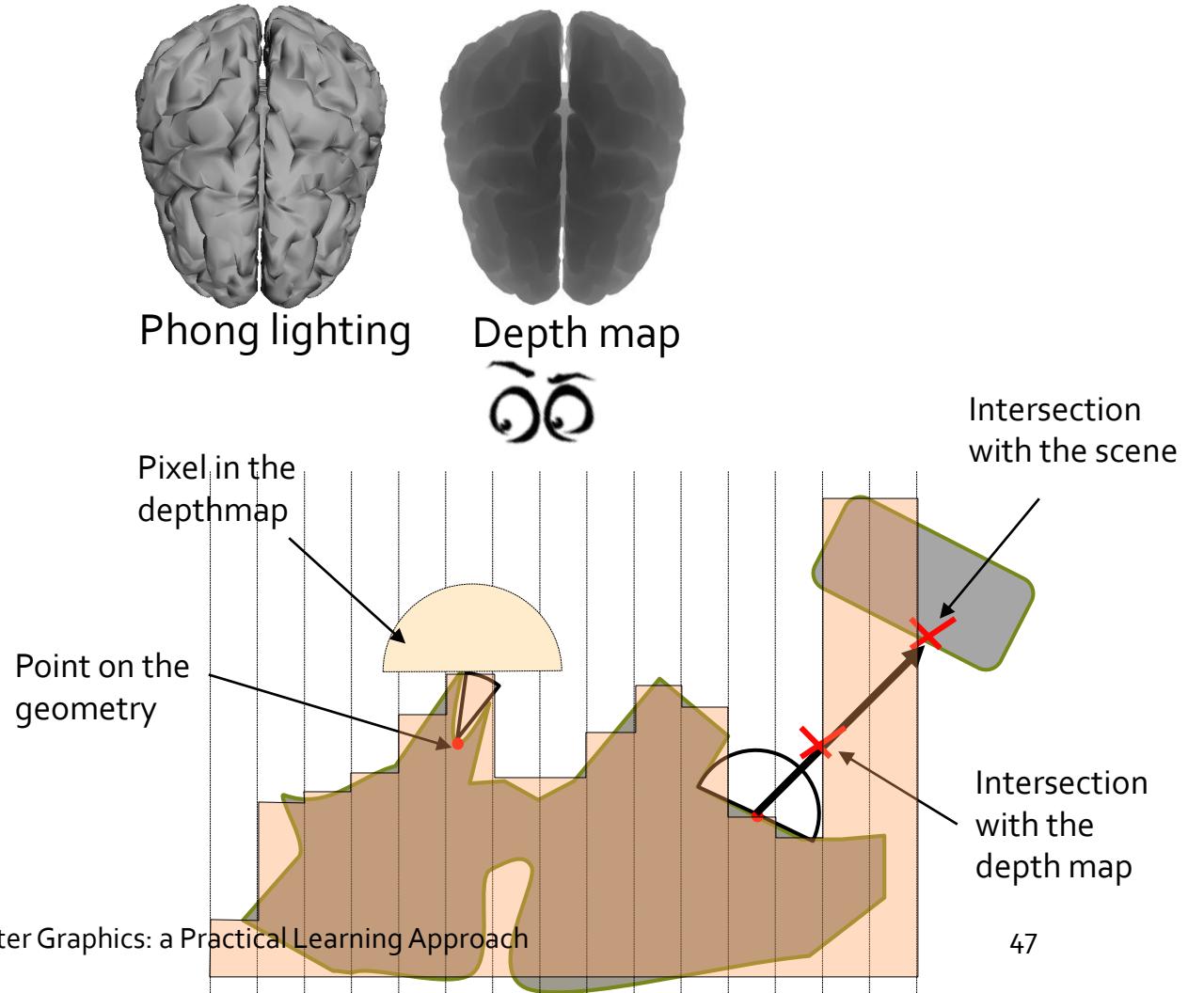
- Screen Space means the rasterized scene from the viewer is all the geometry we can use..
- ..which is a **discrete and partial** representation of the scene geometry, so:
 1. The computation is done **per pixel** and each pixel represents a small patch of visible geometry
 2. There is no scene through which to shoot rays to check for blockers

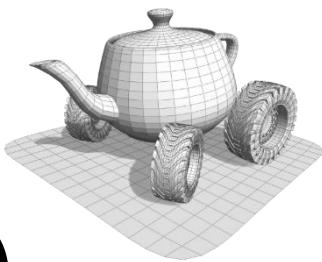




Screen Space Ambient Occlusion (SSAO)

- Screen Space means the rasterized scene from the viewer is all the geometry we can use..
- ..which is a **discrete and partial** representation of the scene geometry, so:
 1. The computation is done **per pixel** and each pixel represents a small patch of visible geometry
 2. There is no scene through which to shoot rays to check for blockers





Screen Space Ambient Occlusion (SSAO)

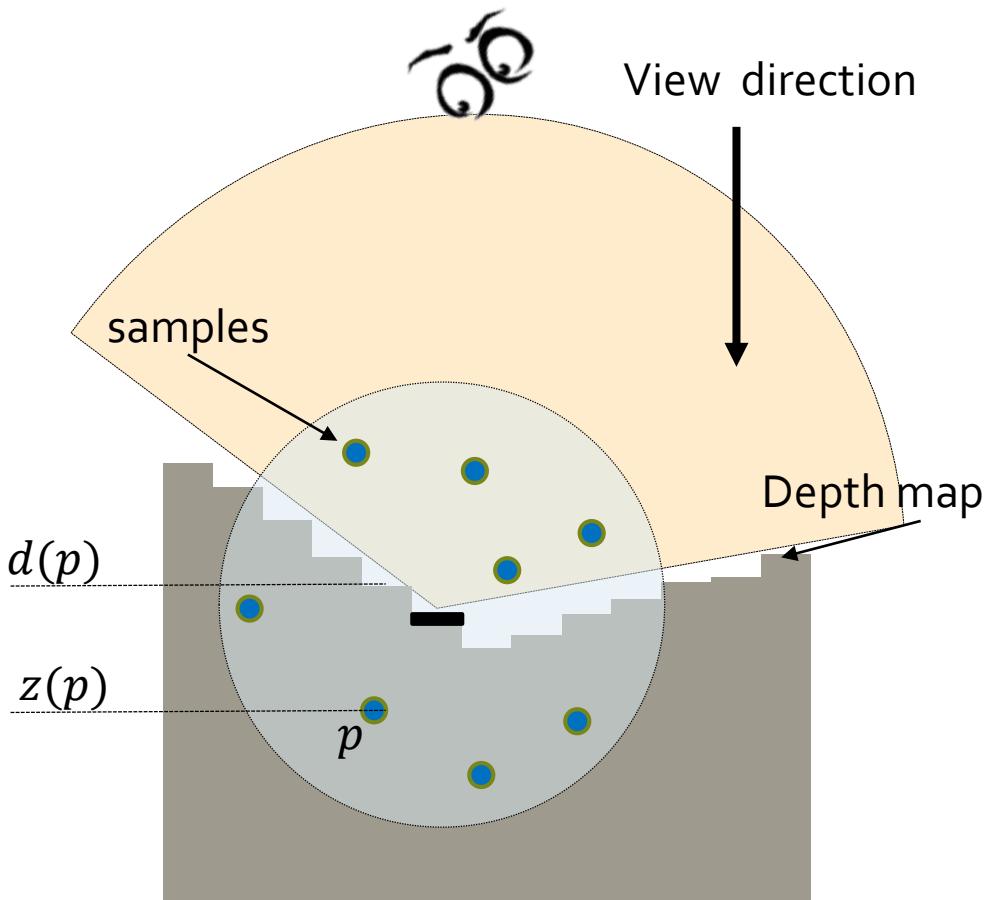
- How to: for each pixel, test for a set of points within a distance r from the pixel how many are visible, that is:

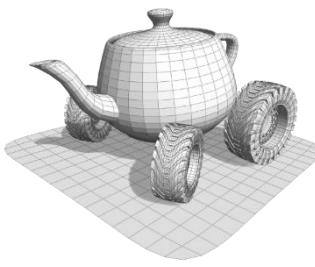
$$AO = 1 - \frac{1}{\#S} \sum_{p \in S} V(p)$$

S = set of samples

$$V(p) = \begin{cases} 1 & z(p) < d(p) \\ 0 & \text{otherwise} \end{cases}$$

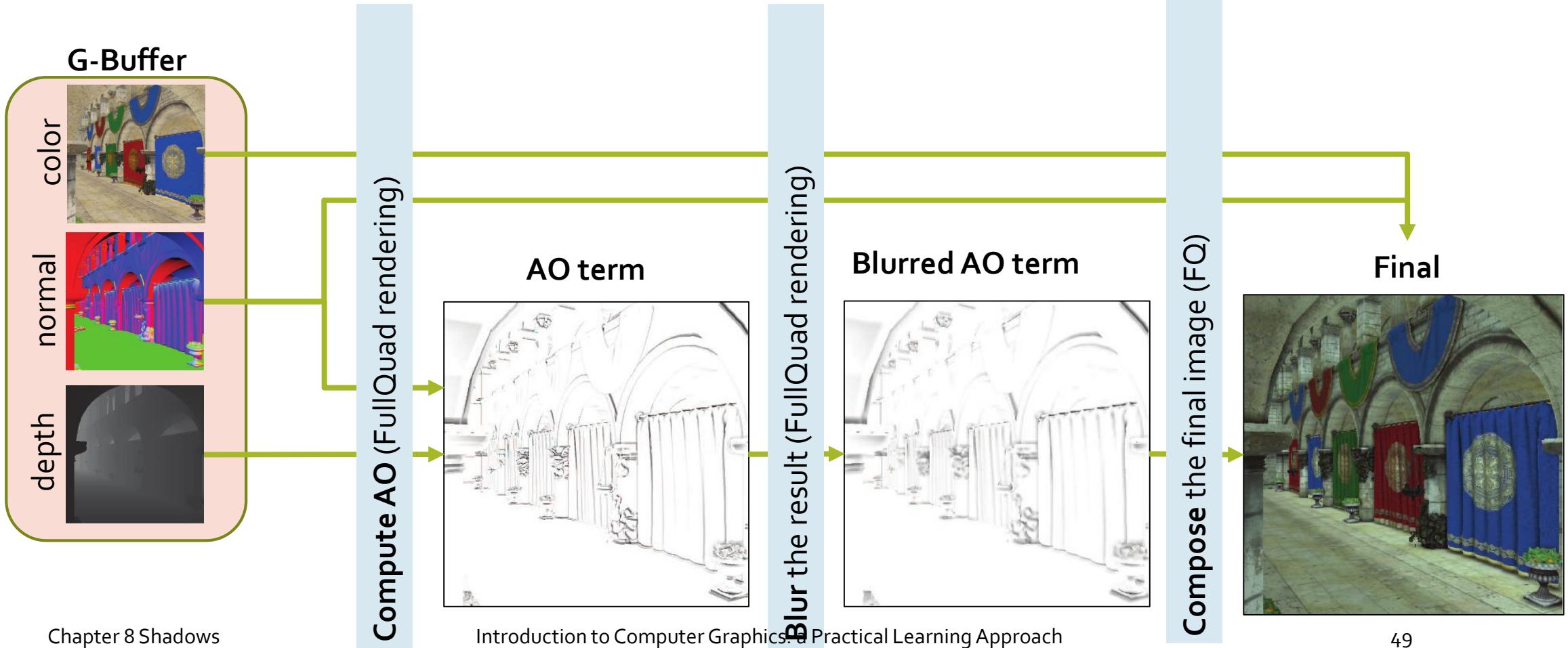
- Q: how many samples? Where do you put them, exactly? How big is r ?

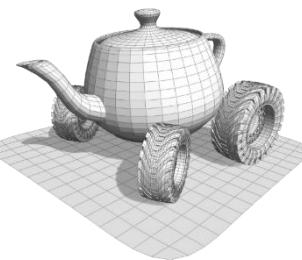




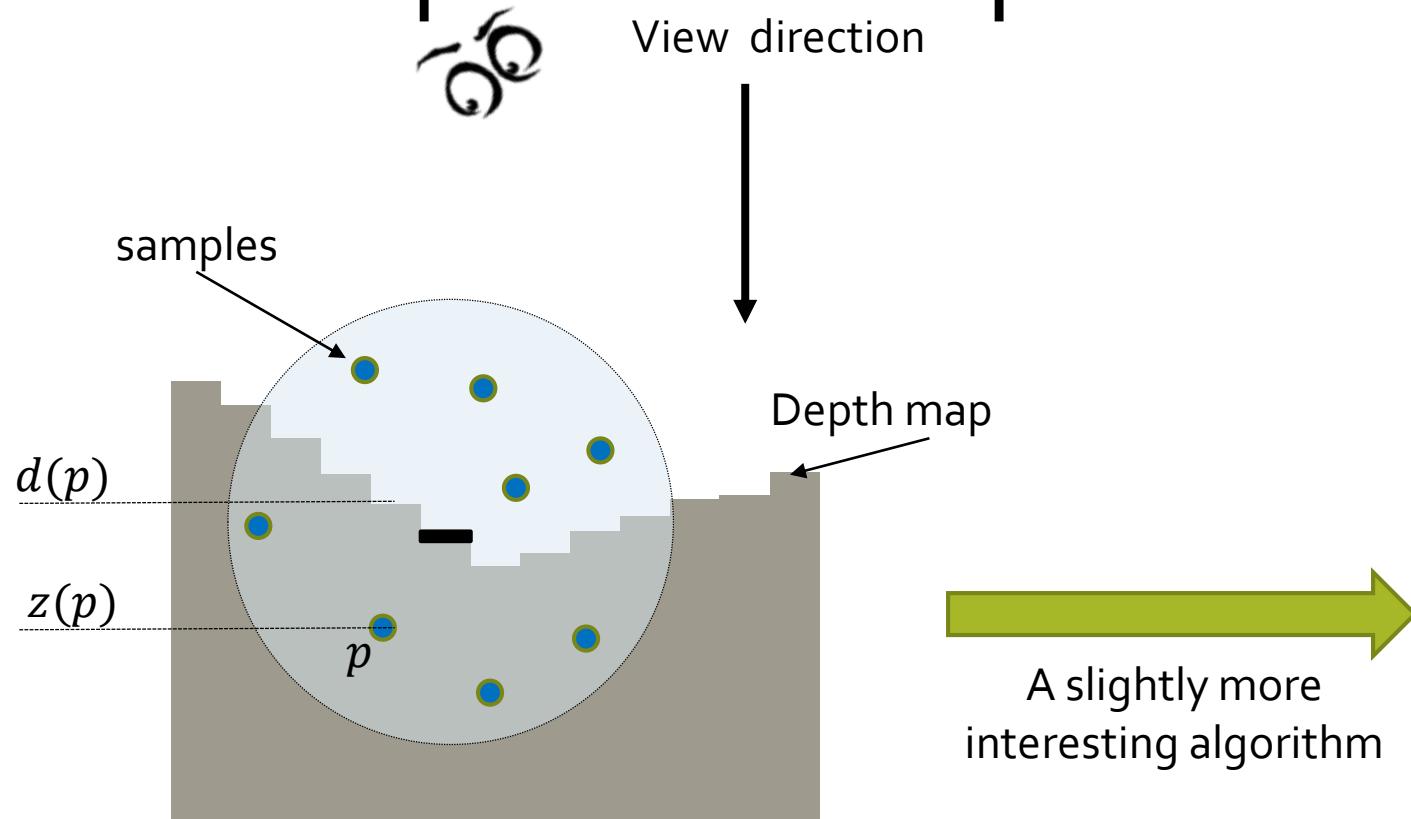
AO pipeline

Rasterize Scene and write out buffers

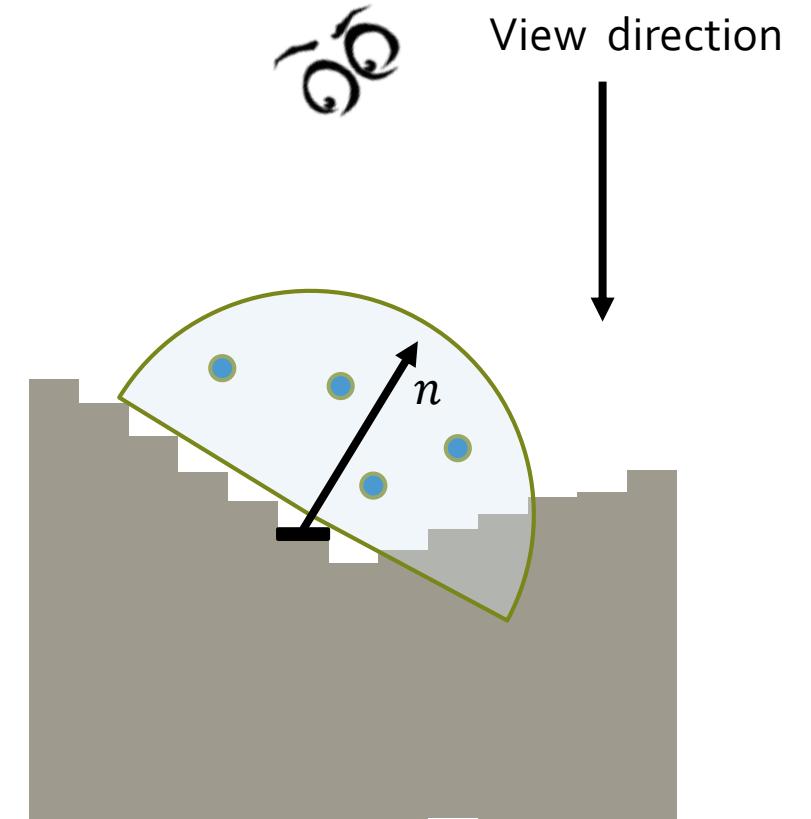




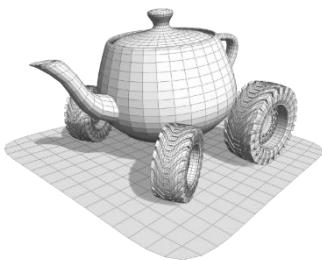
AO Pipeline: Compute AO



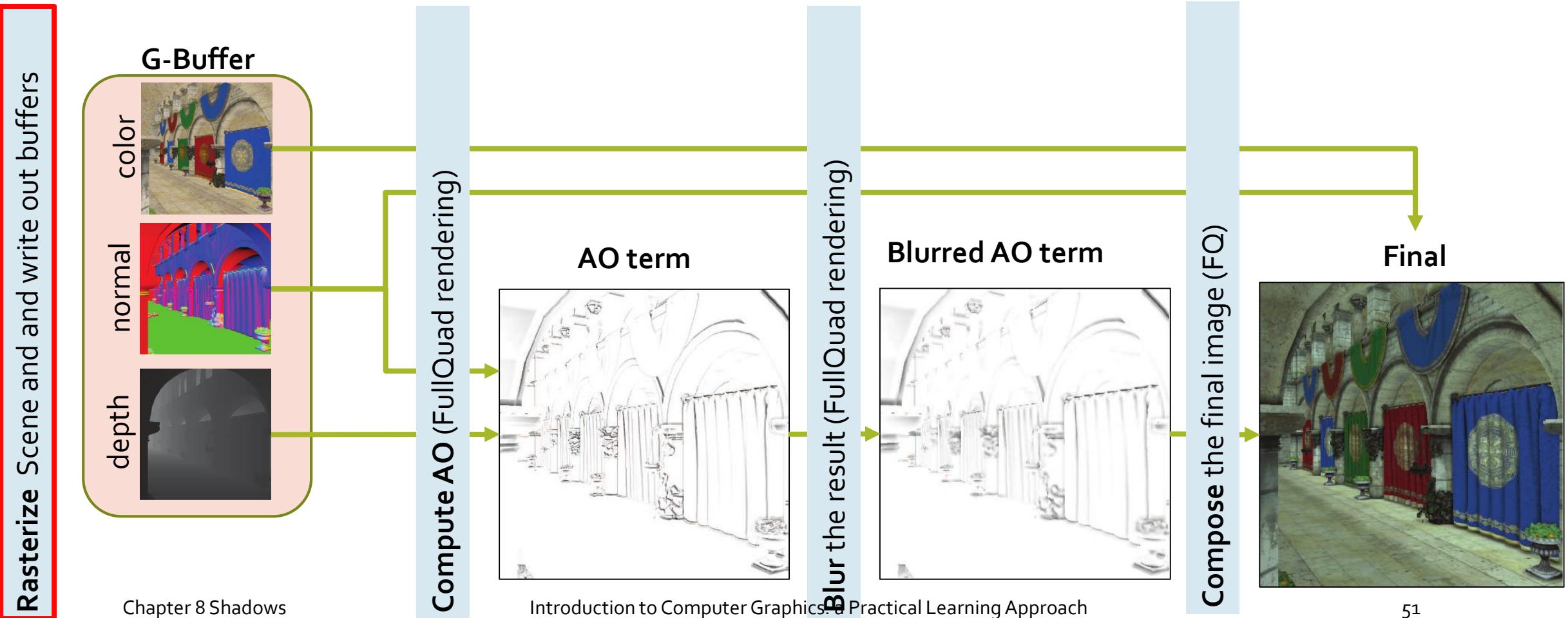
Samples in screen space
Normal not used
Samples further away than p kind of wasted

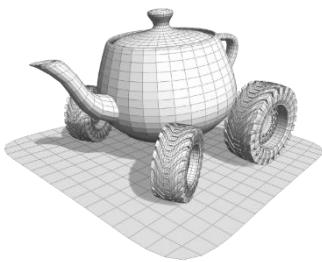


Samples in world space
Use the normal to place the hemisphere
Only potentially visible samples



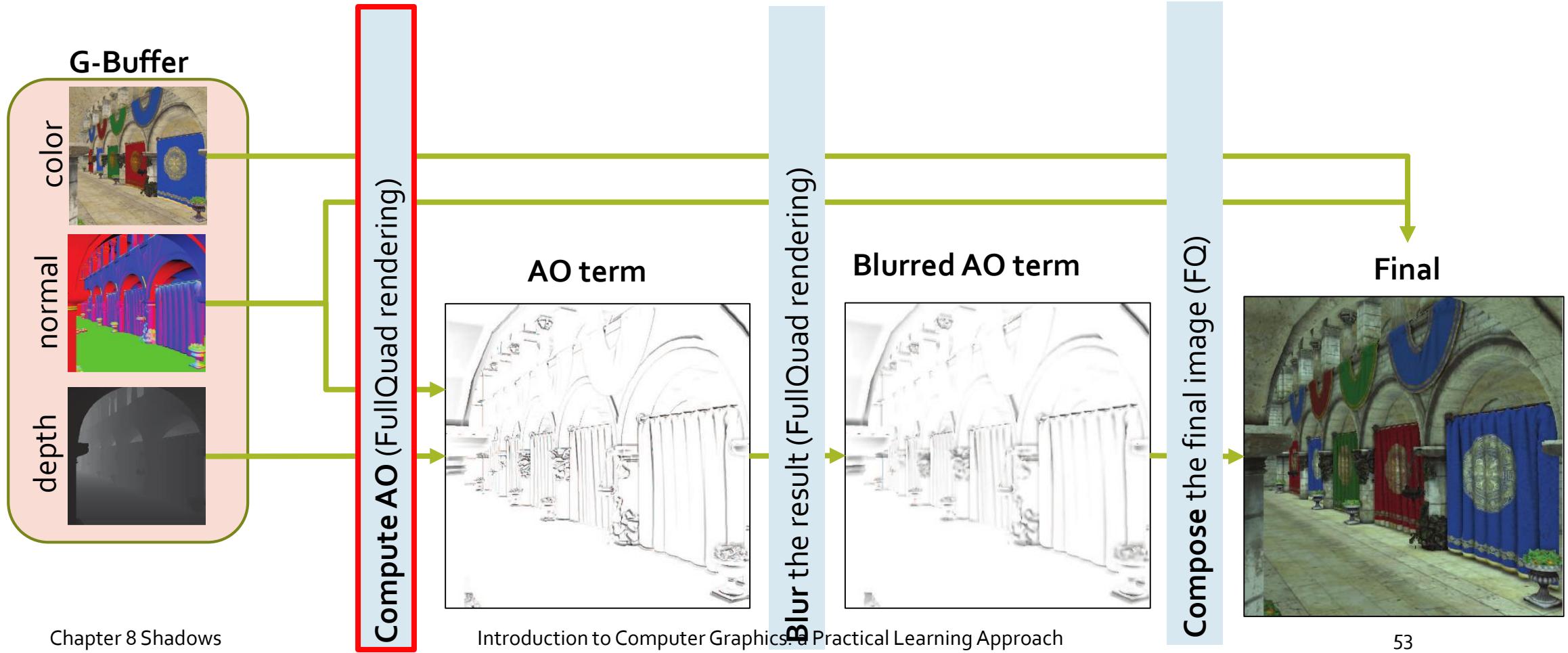
AO pipeline: Create the buffers

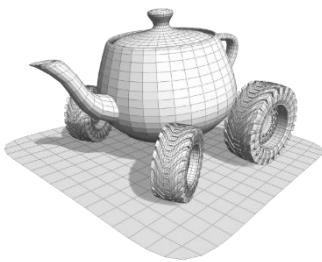




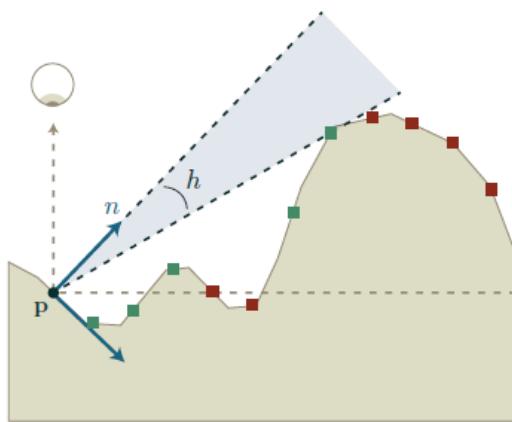
AO pipeline: Compute AO

Rasterize Scene and write out buffers

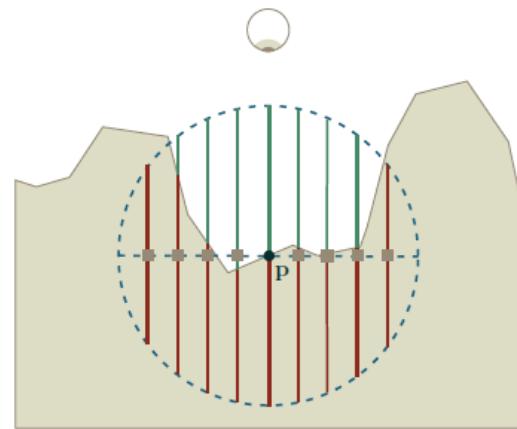




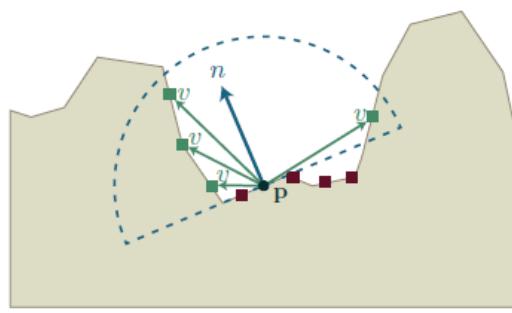
SSAO: some other methods



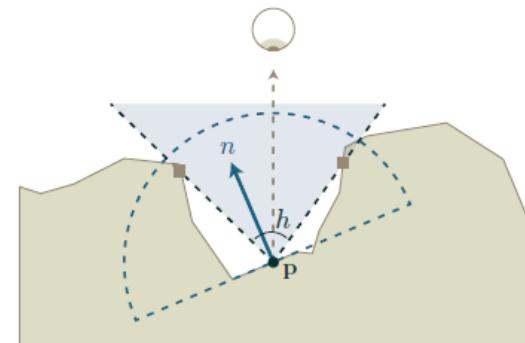
Horizon based with ray
marching
[Bavoil et al. 2008]



Volumetric AO with line
samples
[Loos and Sloan 2010]

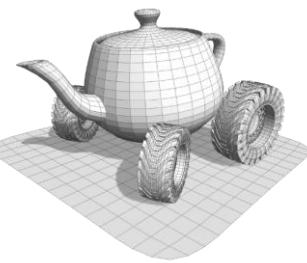
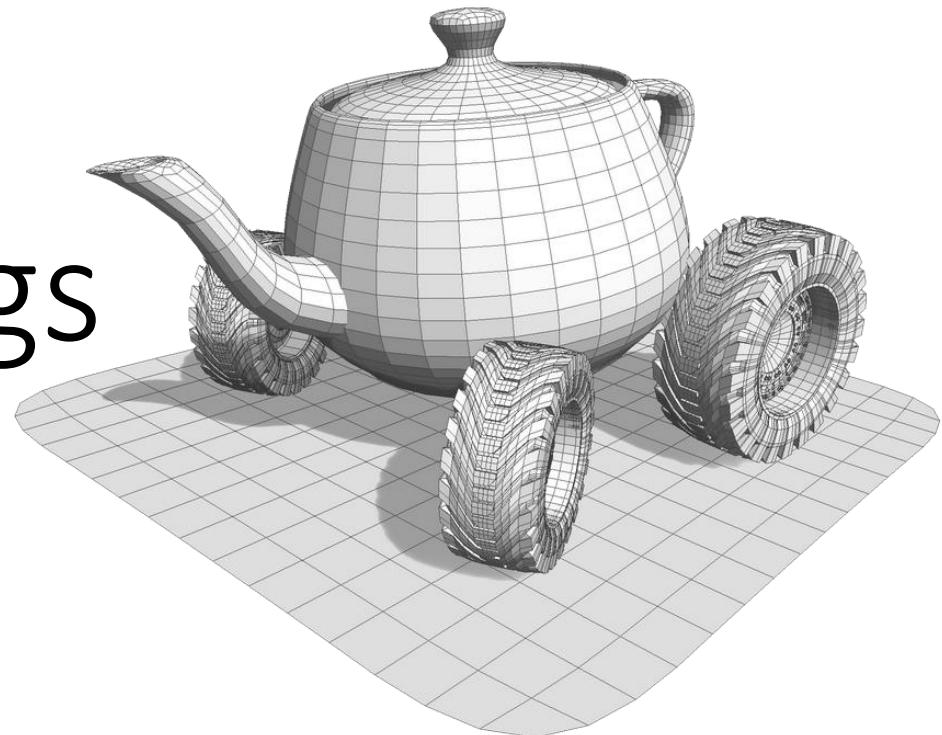


Alchemy AO with projected
samples and special ρ
[McGuire et al. 2011]

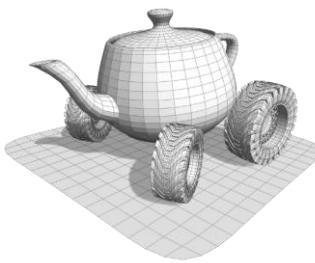


Horizon Based AO with
paired samples
[Mittring 2012]

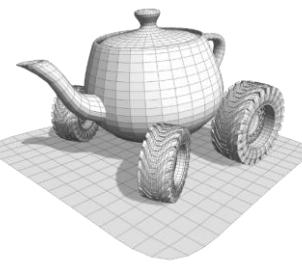
More things



Tools we have / Techniques we know



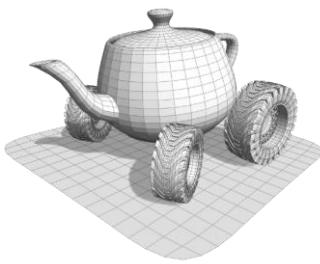
- **Texture mapping:** adding detailed information to the geometric description
- With texture mapping hardware we can do:
 - Better surface description
 - Texture mapping color ☺
 - Normal mapping / bump mapping / parallax mapping ...
 - Not just color by geometry
 - Global lighting effects
 - Environment maps
 - Not just color or geometry but bounces of light
 - Shadow maps
 - Ambient Obscurance



Tools we have / Techniques we know

- **Frame Buffer Objects**
 - Offline rendering -> multipass rendering
- **Rendering on Multiple targets**
 - Creating G-Buffers (geometry buffers)
- **Full-screen quad:** rendering a large quad just to activate the fragment shader on every pixel
- **Deferred shading**
 - Rendering the scene offline storing depth, normal, material etc..
 - Do the final rendering with a full-screen quad using the previously stored maps

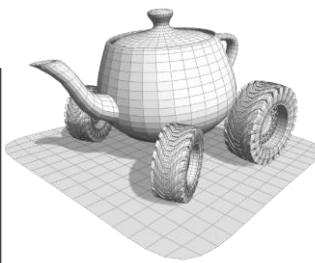
A world of possibilities



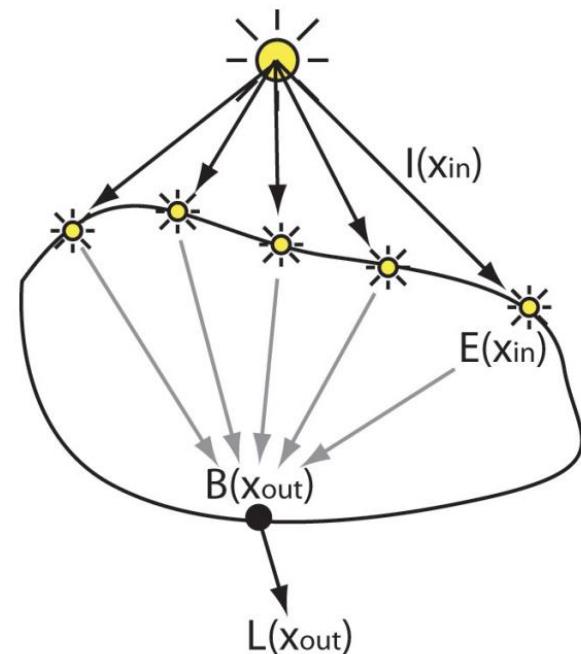
- With these tools at hand we take many steps towards a more photorealistic rendering
- Today: a (largely incomplete) set of examples on various aspects of real-time rendering
- Ratio: with the knowledge we have and the paper we should be able to implement these examples and their follow-ups

GI: Translucent Shadow Maps

- **Translucent** materials let the light through them
- Light partially enters the solid, it's refracted and exits (refracted again)
 - Recall: refraction angles and amount of light are governed by *refraction index* of the media, Snell's law and Fresnel effect



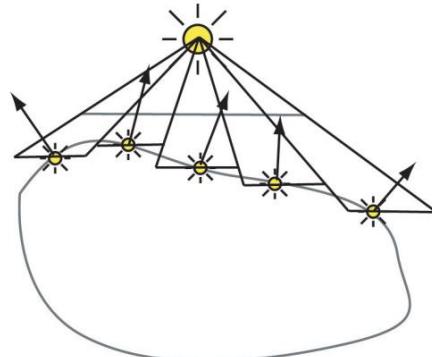
A back lit translucent object



GI: Translucent Shadow Maps

- A TSM is a shadow map that also includes normal and irradiance per texel
- Algorithm:
 - Pass 1: render from the light and build the TSM
 - Pass 2: render the scene and **access the TSM** to gather radiance

How?



A back lit translucent object



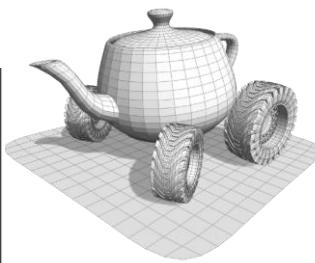
irradiance



depth



normals



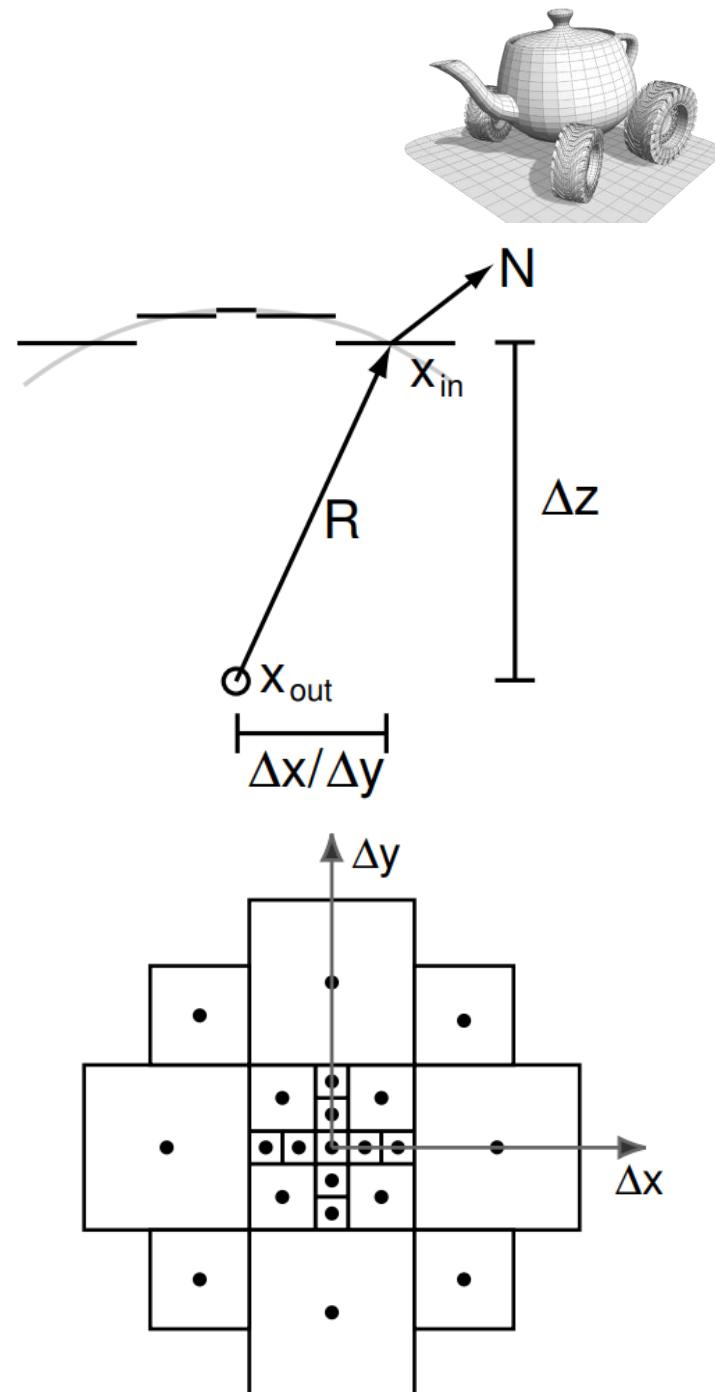
GI: Translucent Shadow Maps

- Pass 2:
 - Find the TSM coordinates as for Shadow Mapping
 - Find the amount of irradiance going towards $x_{out} - x_{in}$
 - It depends on N, not covered in our course

Diffuse subsurface reflectance function

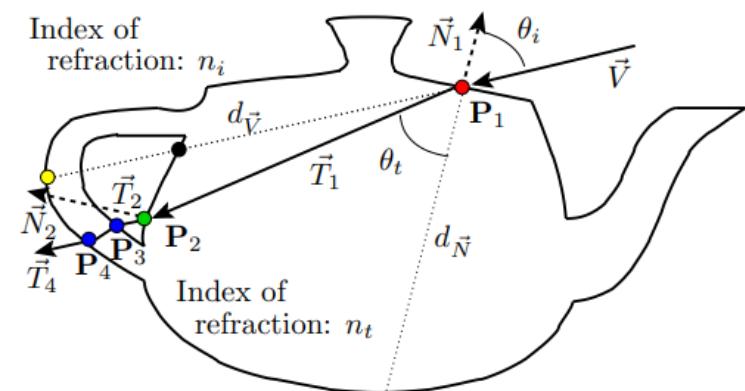
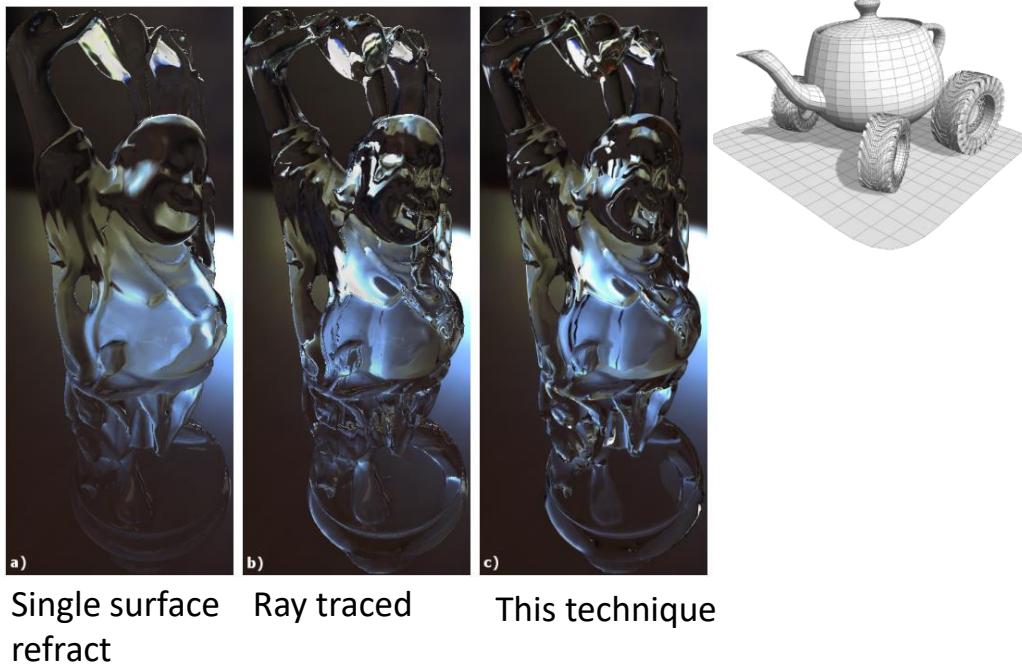
$$B(x_{out}) = \int_S E(x_{in}) R_d(x_{in}, x_{out}) dx_{in}$$

Part of light entering the solid



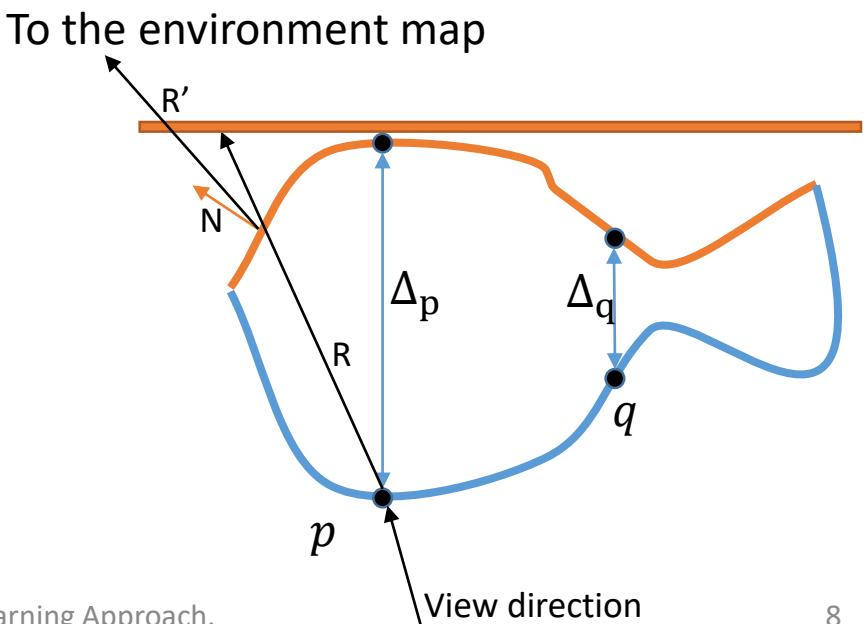
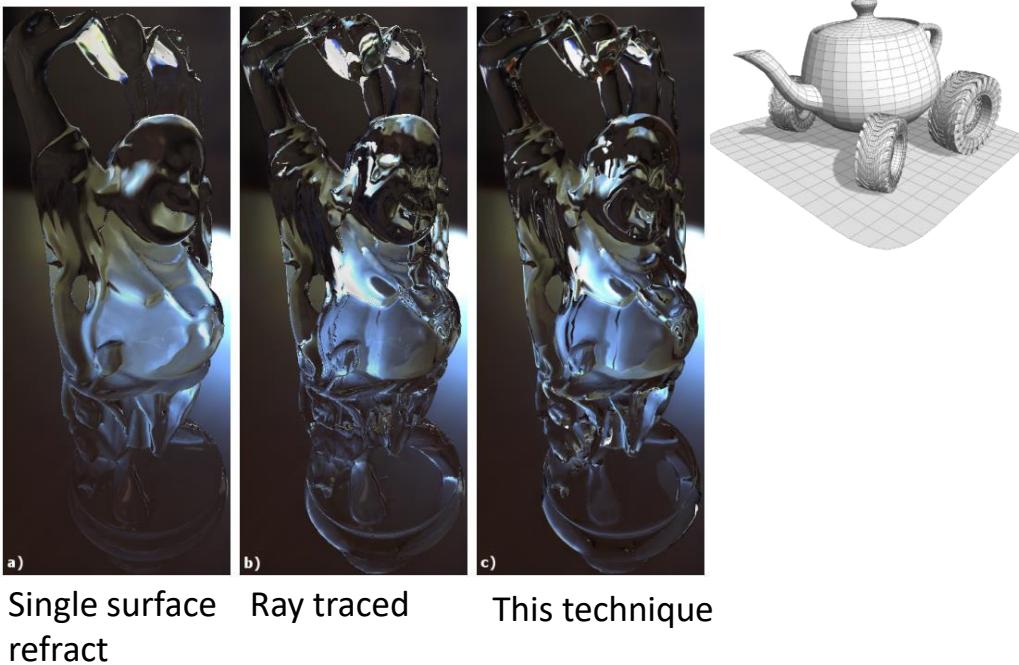
GI: approximated refraction

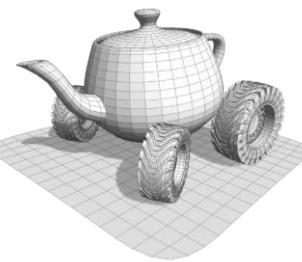
- Two passes, both from the view
 - Pass 1: render the back faces and so build the depth map for the back of the object
 - Pass 2:
 - render normally,
 - compute the refracted ray for the pixel
 - use the depth difference to compute the depth of the object
 - estimate the exiting point
 - compute the exiting refracted ray to access the environment map



GI: approximated refraction

- Two passes, both from the view
 - Pass 1: render the back faces and so build the depth map for the back of the object
 - Pass 2:
 - render normally,
 - compute the refracted ray for the pixel
 - use the depth difference to compute the depth of the object
 - estimate the exiting point
 - compute the exiting refracted ray to access the environment map

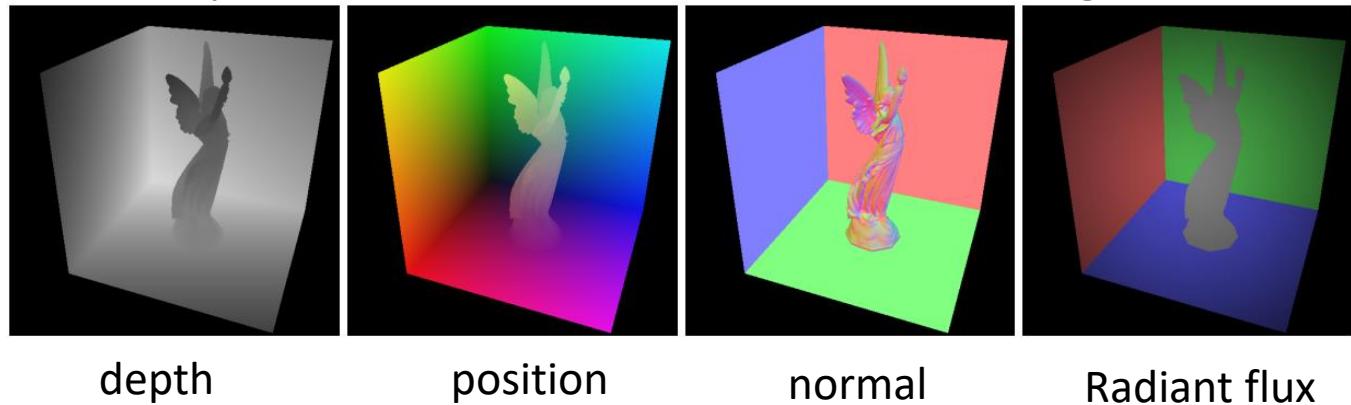




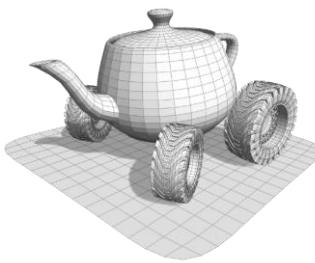
GI: Reflective Shadow Maps

- Approximate light's first bounce
- Two passes:
 - Pass 1: create the RSM
 - Pass 2: render from the view
 - Idea: in the fragment shader, read **the entire RSM**. For each texel, using the 3D position, the normal and the radian flux, compute the contribution for the current fragment

This greenish color...

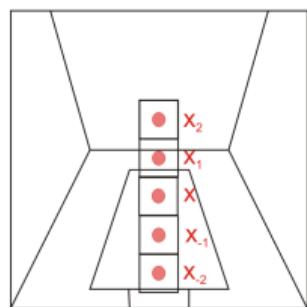
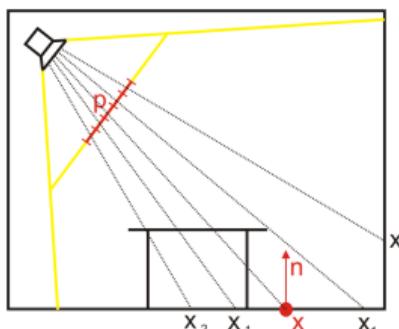


GI: Reflective Shadow Maps

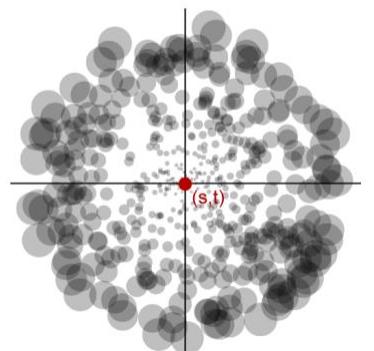


- Approximate light's first bounce
- Two passes:
 - Pass 1: create the RSM
 - Pass 2: render from the view
 - Idea: in the fragment shader, read **the entire RSM**. For each texel, using the 3D position, the normal and the radian flux, compute the contribution for the current fragment
 - Implementation: read only a small neighbourhood (hundreds) in RSM space

This greenish color...



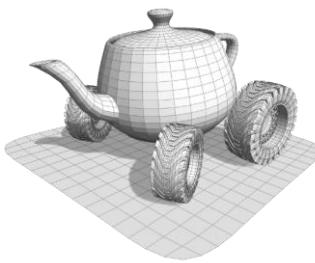
If two points are close in WS,
they are close in TS



Lightmaps

- **Lightmap:** a texture that contains a radiance value:
 - The **view independent** component of lighting (that is, diffuse lighting)
 - The texels of the lightmap are referred to as **lumels**
- Idea: the view independent lighting can be computed once in an accurate but time-consuming way and stored in the lightmap
 - As known as **backed** rendering
- Q: how to fill the lightmaps?





Light probes

- Lightmaps store irradiance from surface, light probes store irradiance in 3D points
- Computation: for each probe
 - Compute radiance from each possible direction, then:
 - Compute Irradiance for each possible direction

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

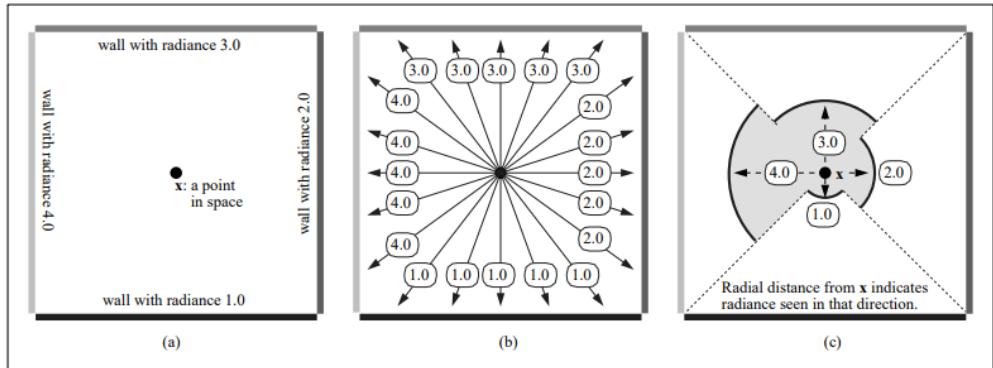
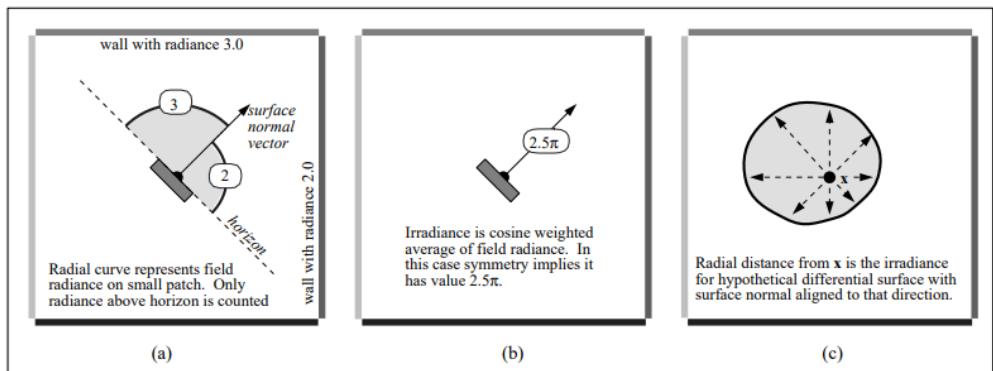


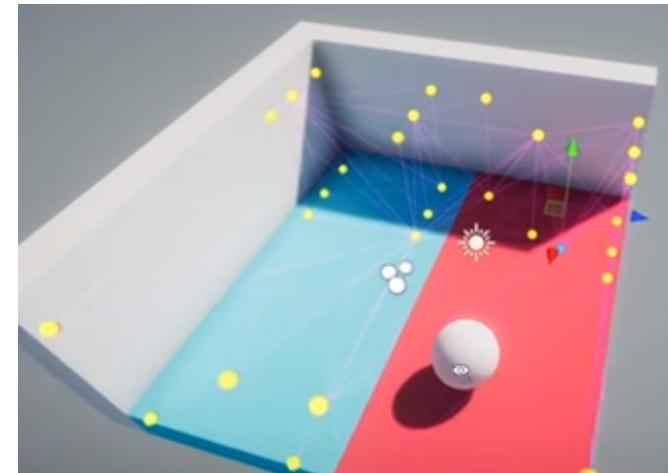
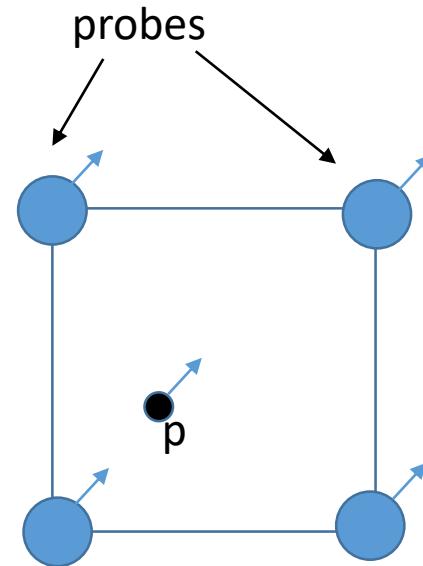
Figure 1: Building a radial plot of radiance as seen from a point in space.



The Irradiance Volume, Greger et al. 1998

Light probes

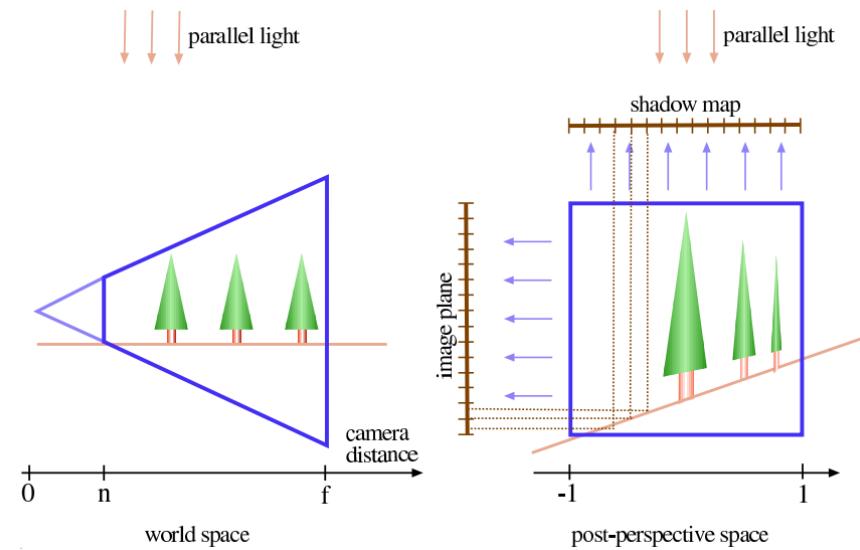
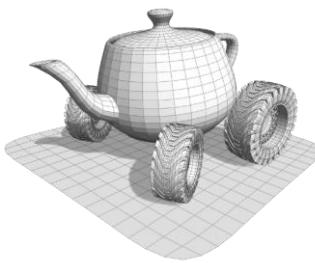
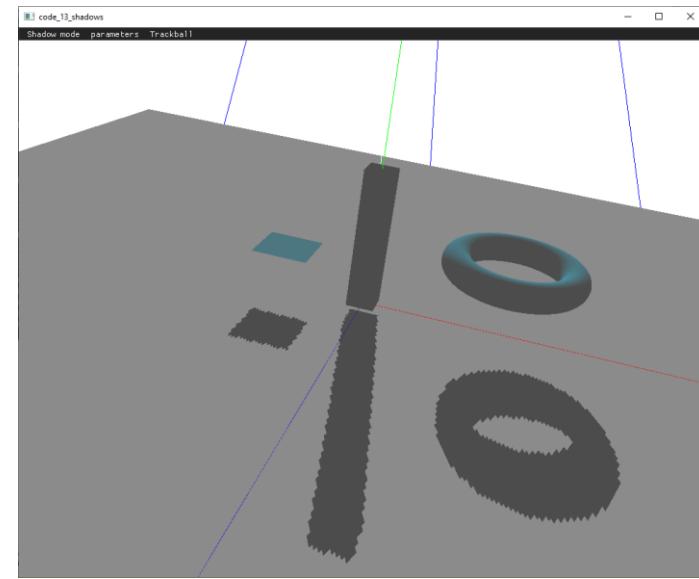
- Using light probes: for a point p and normal n
 - Locate the probes from which p can be interpolated
 - Interpolate the irradiance for p from the probes
- Implementation: how to store a probe?
 - Cube maps
 - Spherical Harmonics
 - ..



Unity Tutorial on light probes

Perspective Shadow Maps

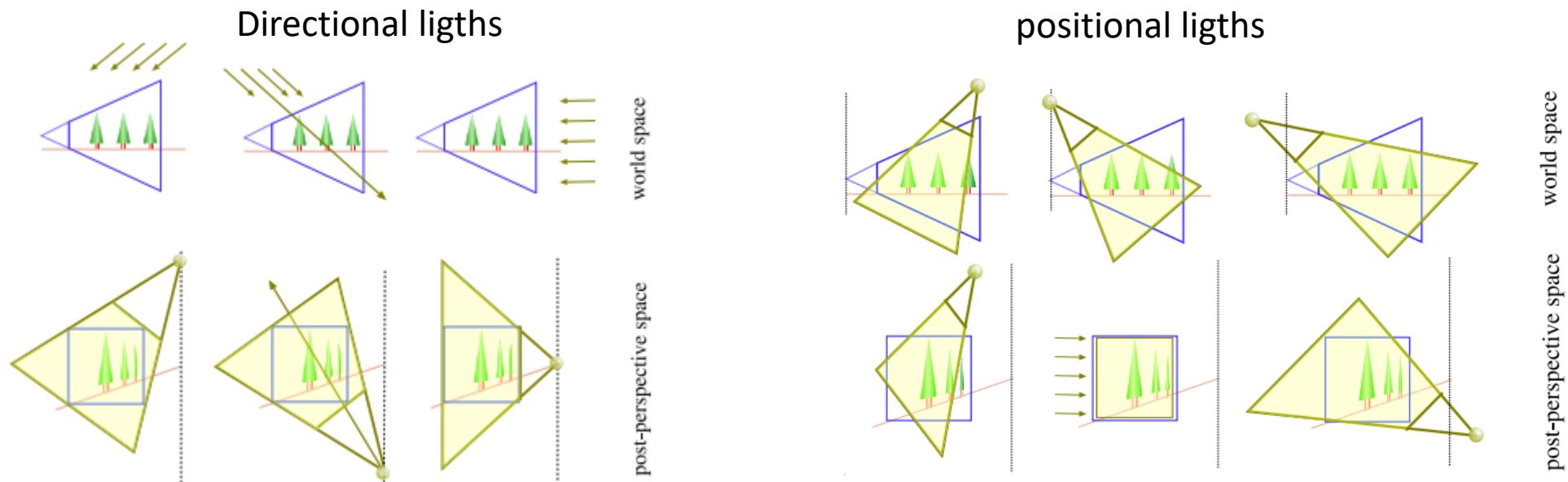
- **aliasing:** texels in the shadow map bigger than the pixels on screen
- **Perspective aliasing:** aliasing due to perspective projection
- Idea of PSM: create the shadow maps *after* the perspective transformation
 - That is: with the scene in NDC space

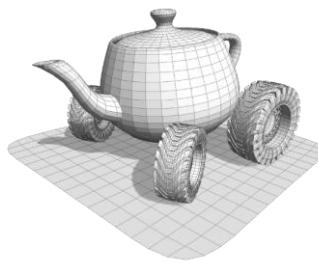




Perspective Shadow Maps

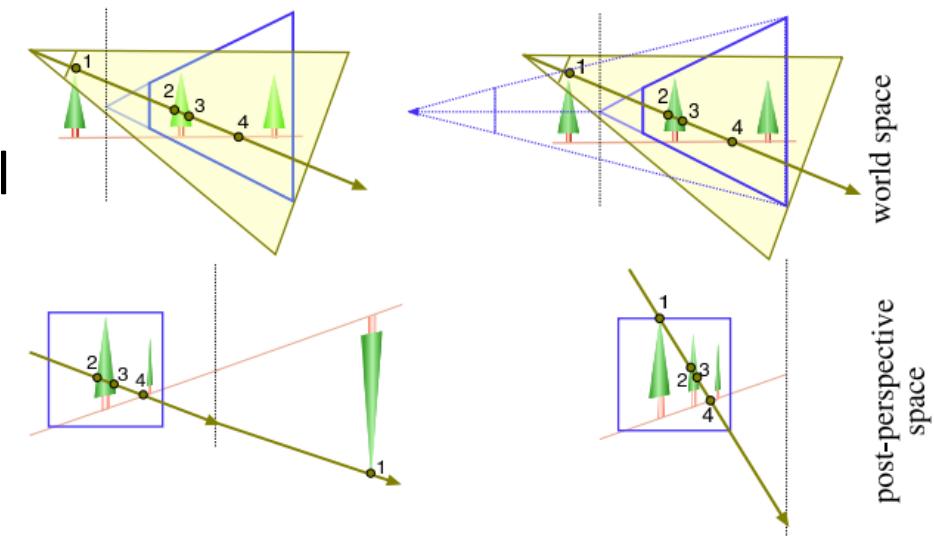
- Algorithm: just like regular Shadow Mapping but the vertex are transformed in NDC when creating the shadow map
- Note: the light source must be transformed as well from WS to NDC

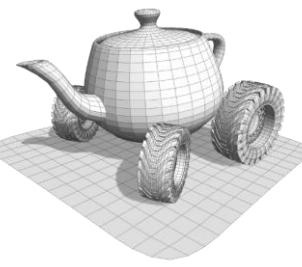




Perspective Shadow Maps

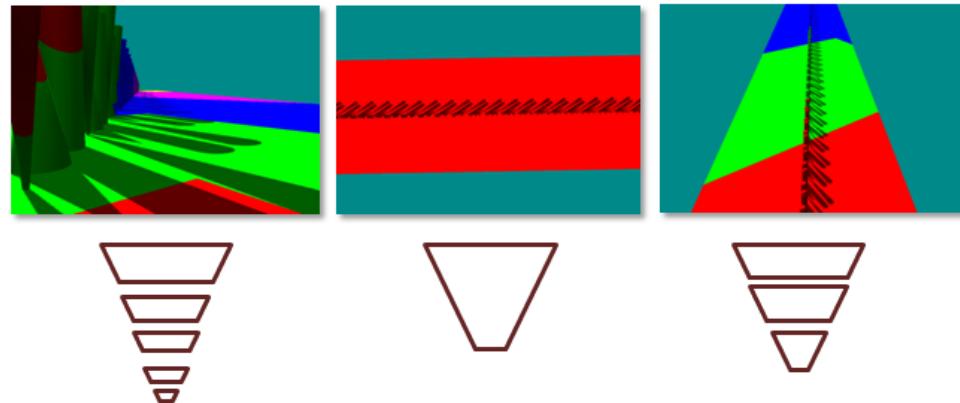
- Problems:
 - The quality depends heavily on the light position/direction
 - Positioning the lights becomes less intuitive
 - Difficult handling of light behind the viewer
 - Bias to compensate acne varies greatly per-texel

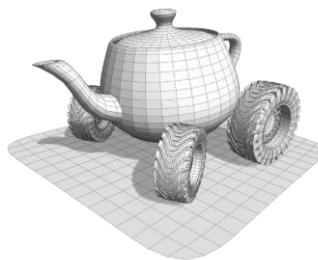




Cascade Shadow Mapping

- Q: couldn't we have a shadow map with a **varying** resolution?
 - Higher res. near the point of view, decreasing with the distance..
- A simple idea :
 1. split the view frustum in multiple frusta
 2. create the proper shadow map for each frustum
 3. In the final pass, choose the proper shadow in the fragment shader

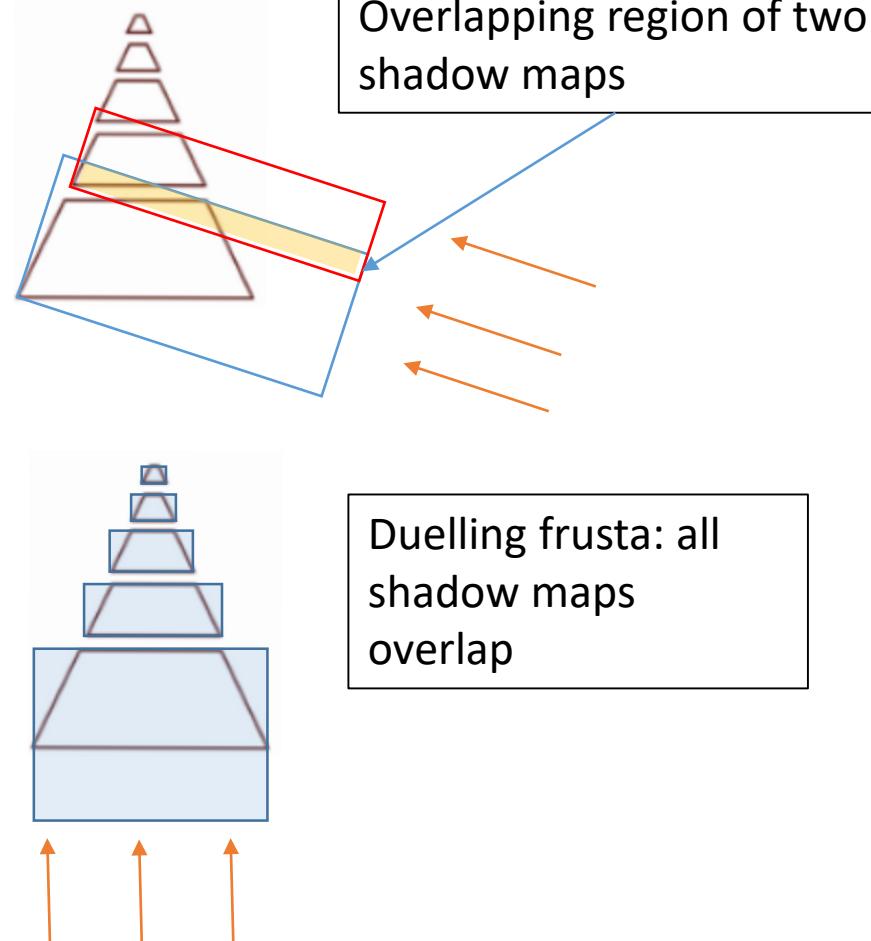


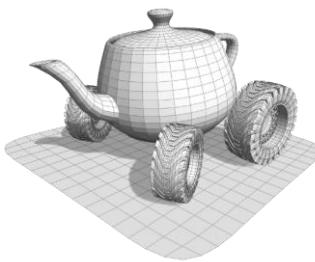


Cascade Shadow Mapping

2. create the proper shadow map for each frustum

- Compute the frustum bounding box in light space
- Set the projection to fit the bounding box
- Bounding boxes will overlap

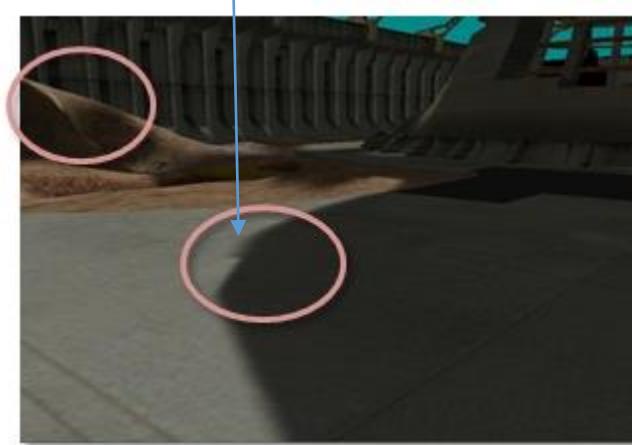




Cascade Shadow Mapping

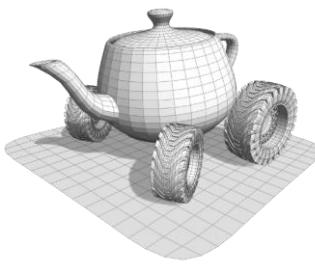
3. In the final pass, choose the proper shadow in the fragment shader
 - Switching shadow maps may cause visible discontinuities (seams) in the shadow
 - when in the border, interpolate between neighbor shadow maps

Seam between two different shadow maps



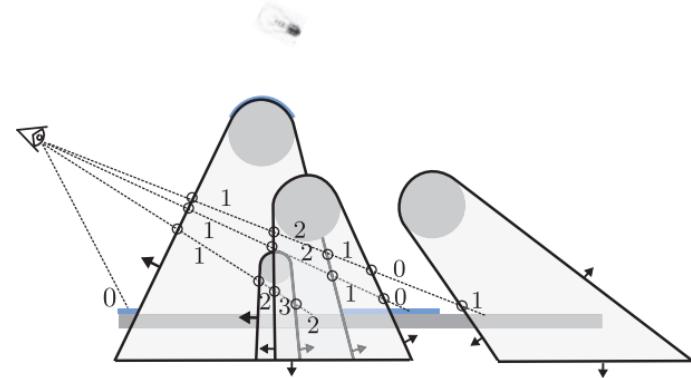
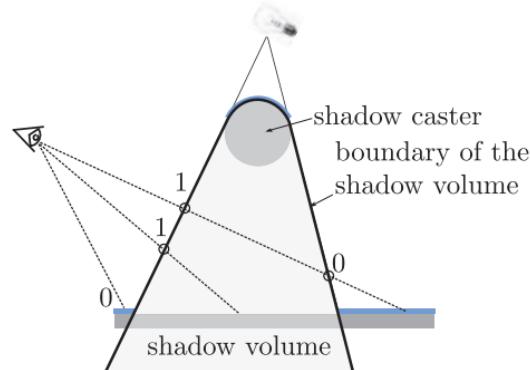
Interpolating between adjacent shadow maps

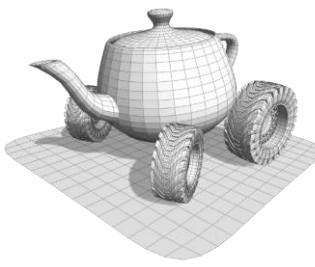




Shadow Volumes

- **Shadow volume:** a 3D region where everything in it is in shadow
 - for each light and object, create a polygon mesh representing the boundary of the corresponding shadow volume
 - the points inside at least one shadow volume are in shadow
 - Note: the union of two shadow volumes is a shadow volume

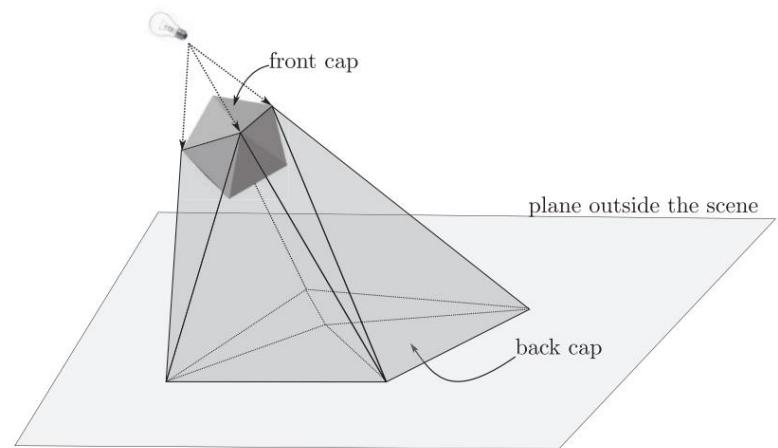
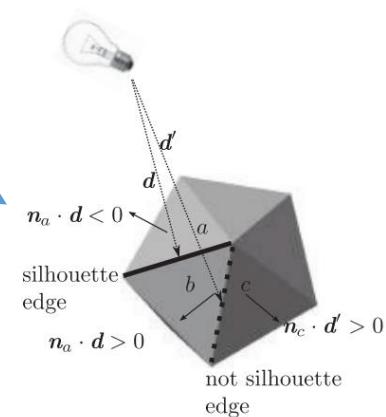




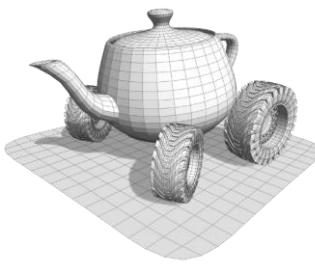
Shadow Volumes

- Creating a shadow volume (for one light and one object):
 - Run over all silhouette edges of the polygon
 - Extend the edge to «infinity» and create a quad

A silhouette edge is an edge shared by two faces with opposite orientation with respect to the light



Shadow Volumes



- Rendering with shadow volumes: use the stencil buffer to count how many times a view ray enters and exits the scene
 1. Render the scene as if all the fragments were in shadow
 2. Disable writing on depth and color buffer
 3. **Stencil Test:** «always pass, increment on depth fail»; **Culling on front faces**
 - Render the shadow volumes: it counts how many time the view ray exits a shadow volume
 4. **Stencil Test:** «always pass, decrement on depth fail»; **Culling on back faces**
 - Render the shadow volumes: it counts how many time the view ray enter a shadow volume
 5. **Stencil Test:** «pass if 0»
 - Render the scene as if all fragments were lit. Those in shadow will not be written because of the stencil test

Shadow Volumes

- Pros:
 - Pixels precise, no aliasing
 - Handles difficult cases
 - No texture accesses involved, just one extra render pass for the depth map
- Cons:
 - Only for static light- shadow caster relations
 - Non closed objects causes light bleed (because the way silhouette is found)
 - Preprocessing object's geometry would solve it

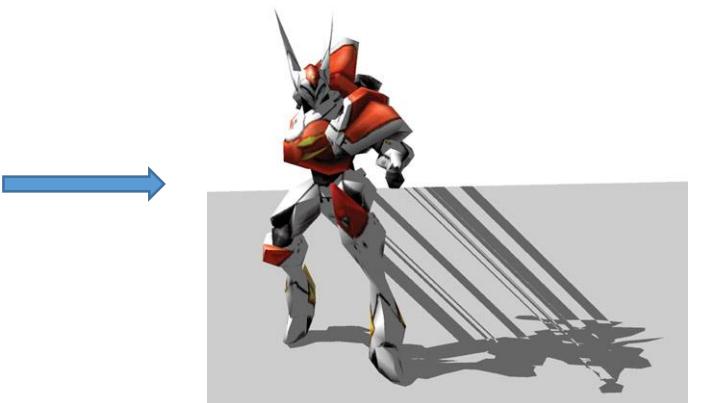


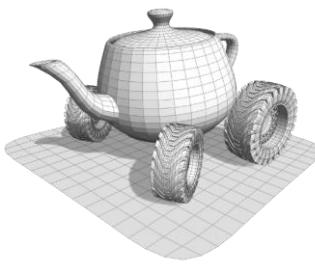
GPU Gems II: chapter 9



Horrible for shadow mapping:

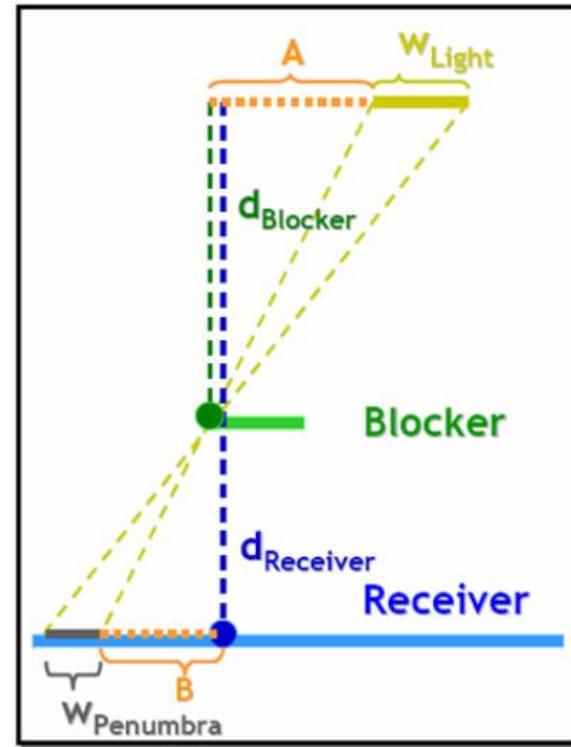
- Most of scene is in shadow-> waste of texels
- Bad perspective aliasing: a small hole project a large light



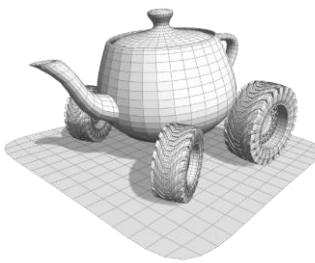


Percentage Closer Soft Shadows (PCSSF)

- Key idea: use PCF but adapt the kernel size to soften the shadows with the size of and distance from the light and the *distance from the blocker*
- How to find the distance from the blocker?
 - By sampling the depth map **around** the projection of the point and average the depth values found
- Pros/cons:
 - Pros: works well, a minor mod to SM, nice results
 - Cons: not very cheap (finding the blocker)

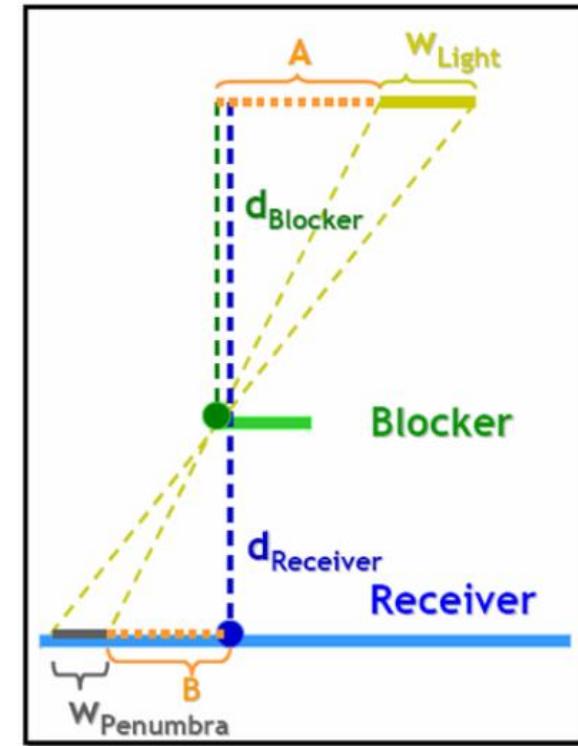
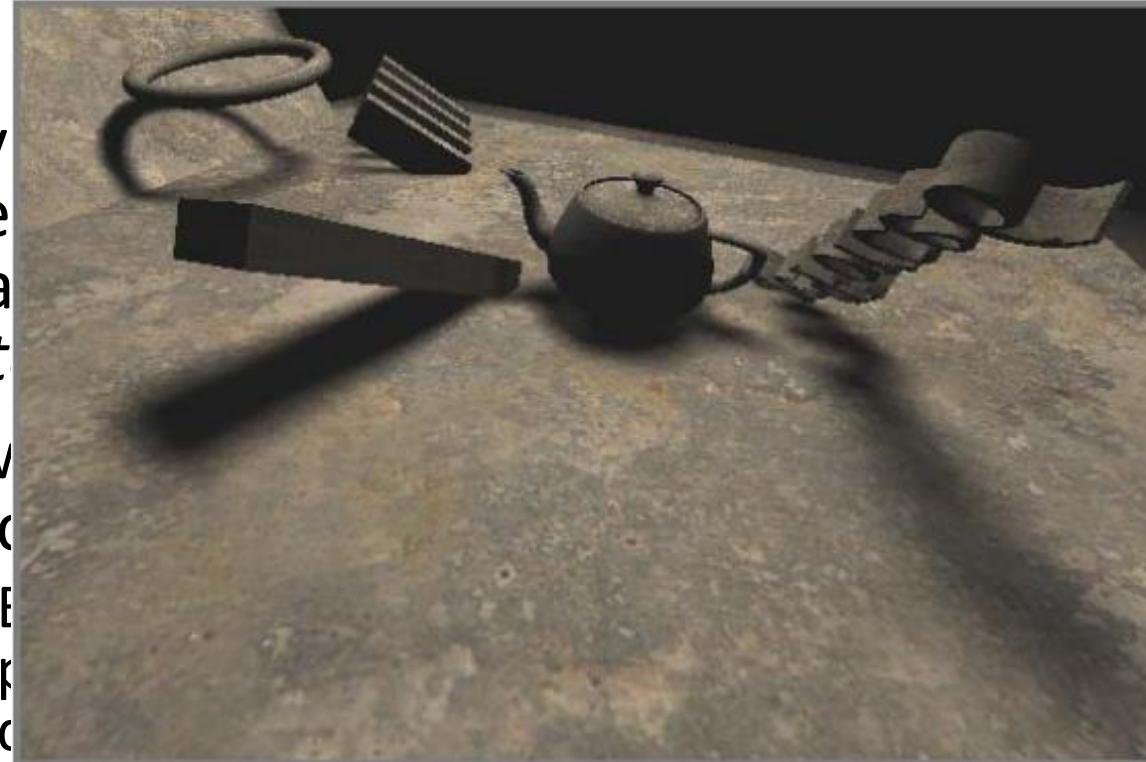


$$w_{\text{Penumbra}} = \frac{(d_{\text{Receiver}} - d_{\text{Blocker}}) \cdot w_{\text{Light}}}{d_{\text{Blocker}}}$$

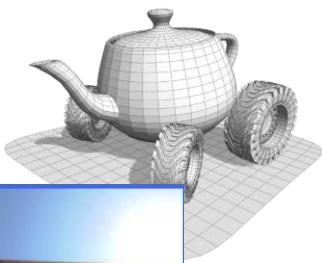


Percentage Closer Soft Shadows (PCSSF)

- Key size of a *dist*
- How blocker
• Easier
• Faster
• Closer
- Pros/cons:

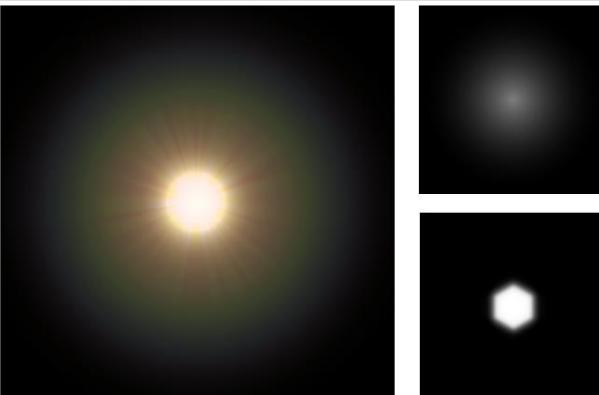
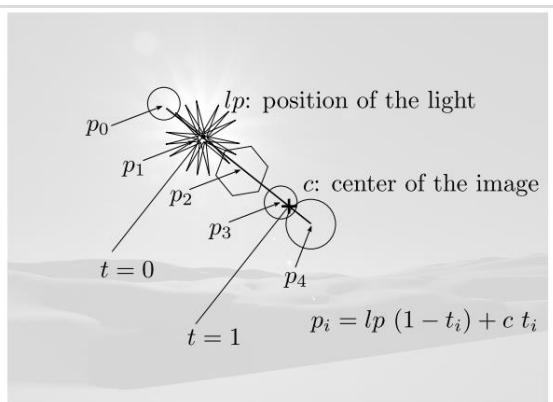


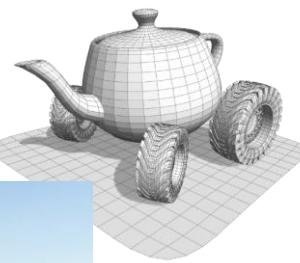
$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$



Lens flares

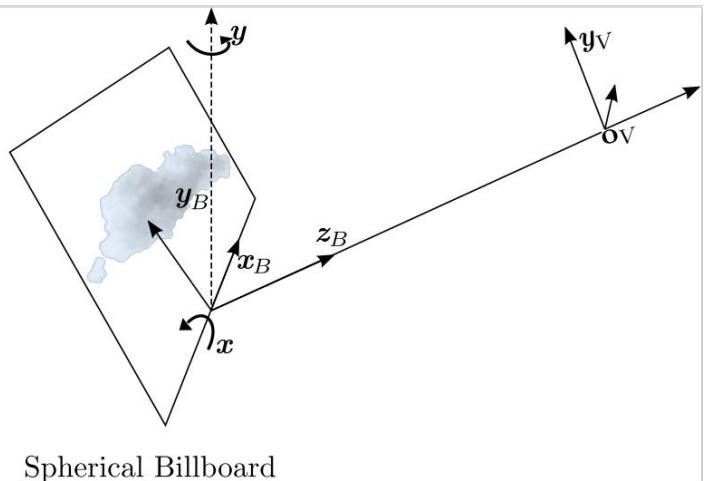
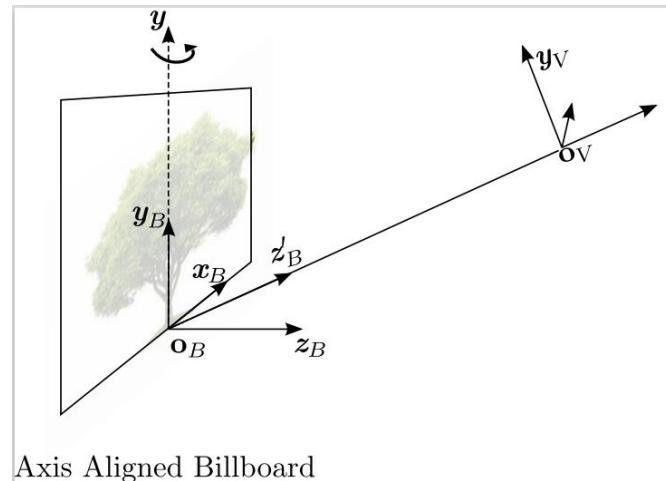
- Lens Flare: annoying artifacts created of light scattering in the lens system
 - Also an annoying overused effect in video games
- How: entirely in images space, draw a series of textures polygons with alpha value along a line from the light projection towards the opposite bottom
- Q: how to know when a point light is seen?

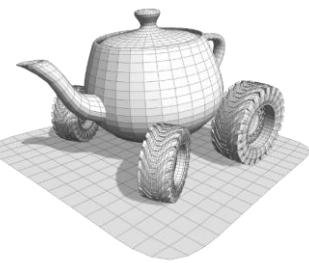




Billboards

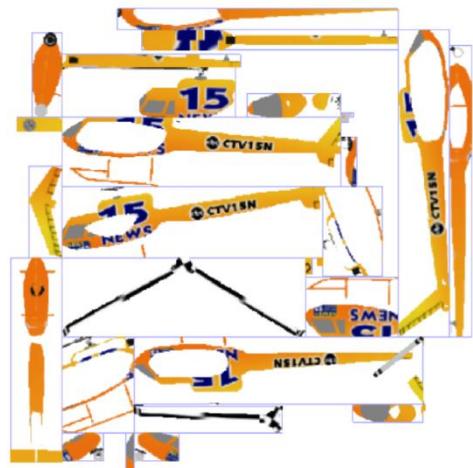
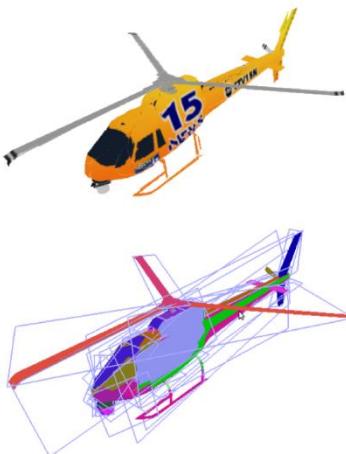
- Widely used for rotation invariant objects (like trees)
- Draw a quad (the billboard) with a texture with alpha component
- Keep it oriented towards the viewer

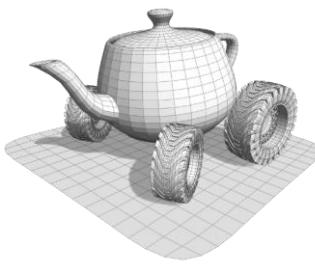




Billboard clouds

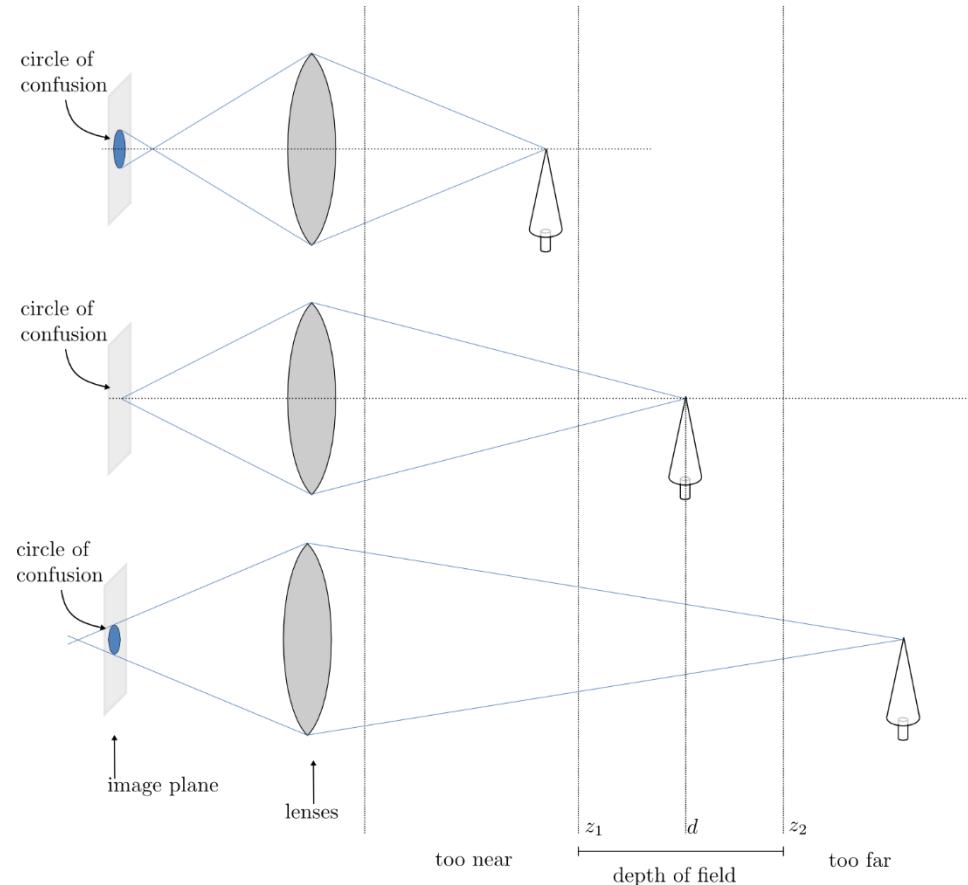
- Extension of a single billboard: multiple billboards for each «flat» portion of the 3d object
- Need to be computed in preprocessing

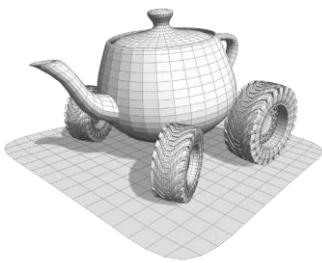




Depth-of-field

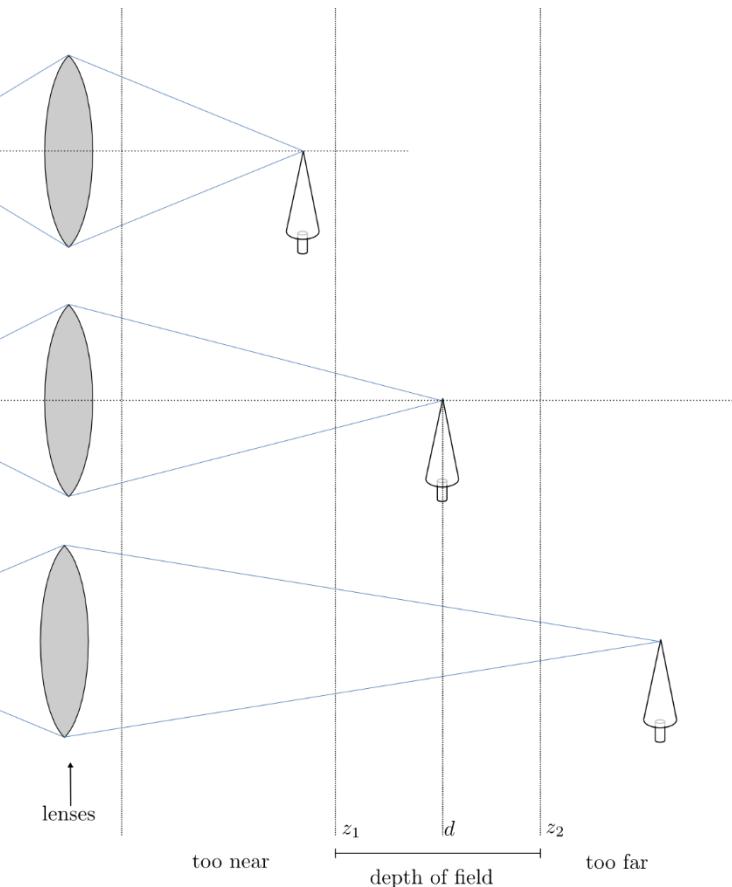
- Because of the lenses, points outside a given interval (in depth) will not project exactly on a single point but in a region
- That region is called *circle of confusion*
- The size of the interval is called *depth of field*
- Algorithm:
 1. Render the scene normally and output the depth buffer
 2. Blur the fragments proportionally to how much the depth value is outside the depth of field

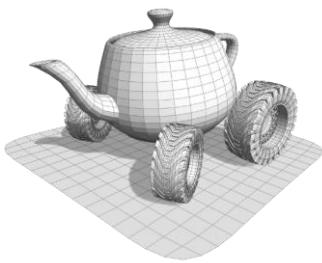




Depth-of-field

- Because the camera only has a finite number of pixels, it cannot record every point in the scene exactly.
- That means that some points will be blurred.
- The size of the blurred area depends on the *depth of field*.
- Algorithms for rendering depth of field:
 1. Ray casting: calculate the depth value for each pixel.
 2. Back buffering: store depth values for each pixel in a buffer, and then use them to determine how much the depth value is outside the depth of field.





Bibliography

Translucent Shadow Maps, Dachsbacher, Carsten and Stamminger, Marc, Eurographics Workshop on Rendering, 2003

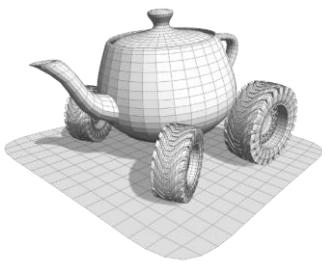
An Approximate Image-Space Approach for Interactive Refraction, Chris Wyman, Siggraph 2005

Reflective Shadow Maps, Dachsbacher, Carsten and Stamminger, Marc, I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games

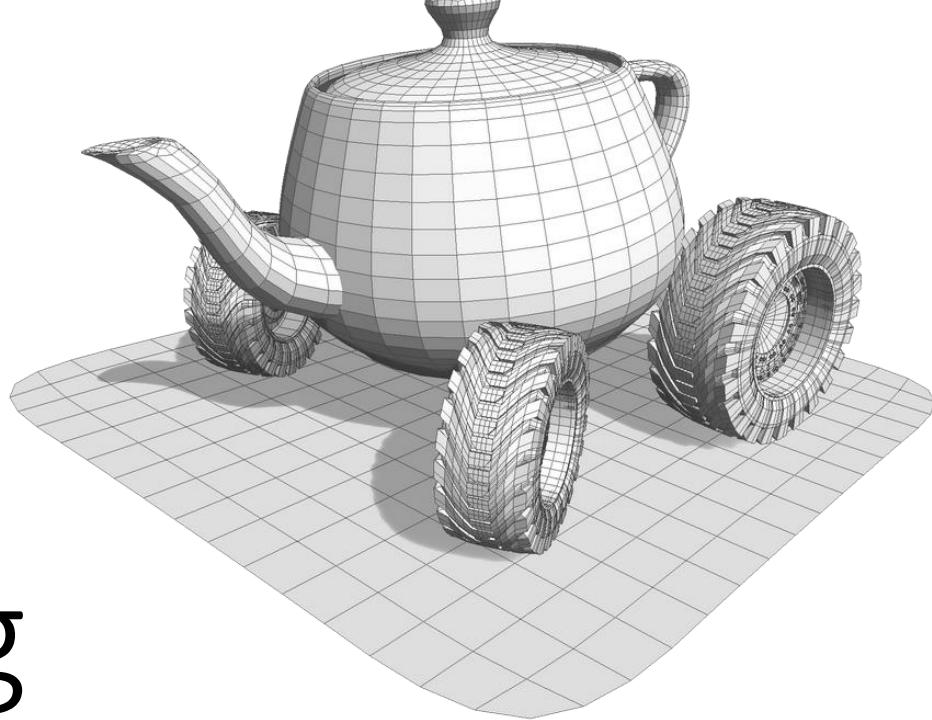
Perspective Shadow Maps, Stamminger and Drettakis, Siggraph 2002

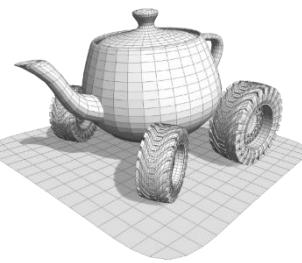
Cascaded Shadow Maps, Nvidia report, Rouslan Dimitrov, 2007

Percentage Closer Soft Shadow Filtering, Randima Fernando, Siggraph 2005



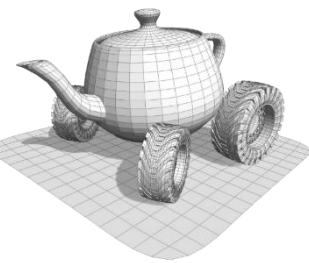
Ray Tracing





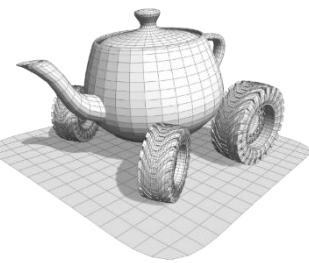
Rasterize this...





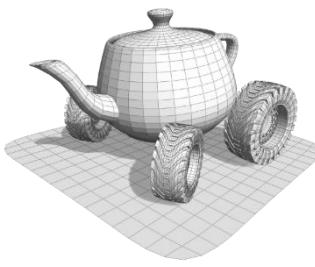
Rasterization + Ray Tracing





Rasterization + Ray Tracing



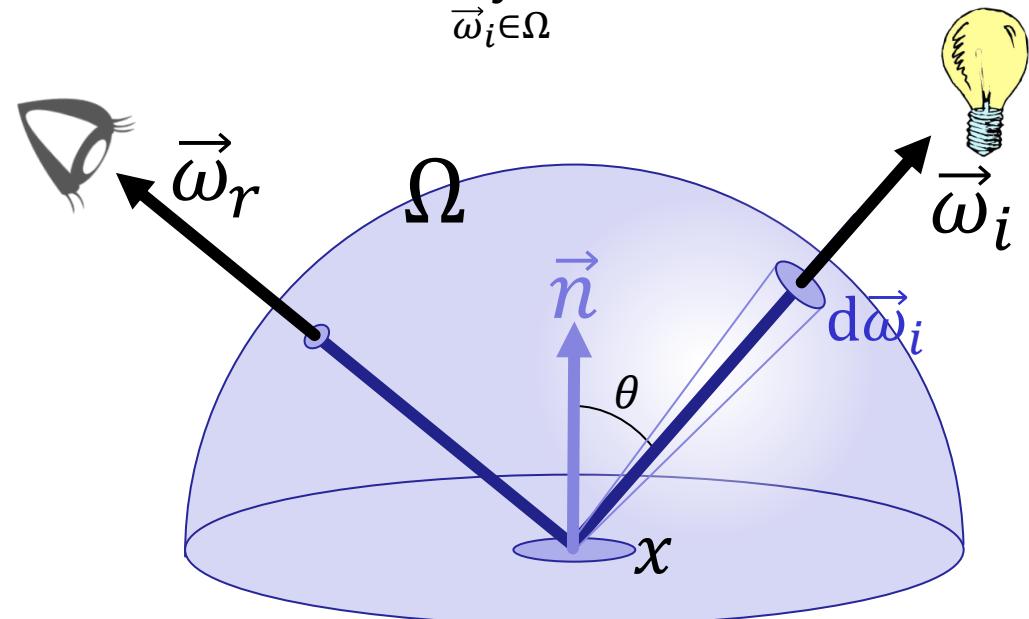


Introduction: Lighting recap

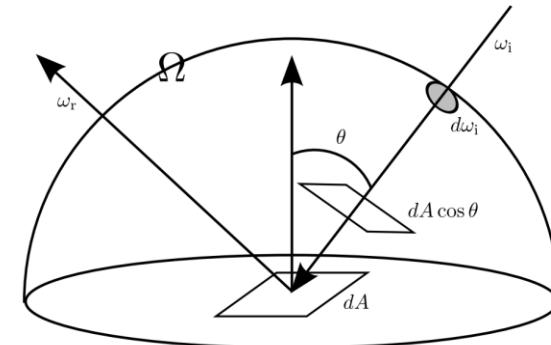
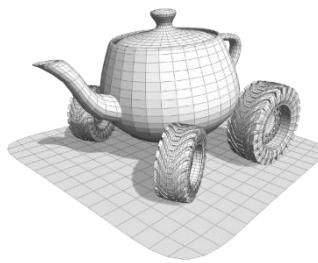
- Recall the Rendering Equation

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) \cos(\theta) d\vec{\omega}_i$$

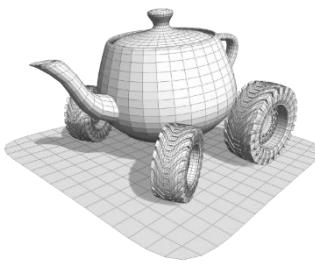
Emissive component **BRDF environment** **Cosine law**



Introduction: Lighting recap

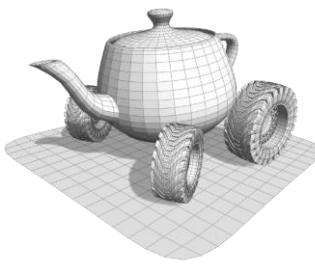


symbol	Name	unit	
Φ	Radiant flux	Watt (J/s)	total amount of light passing through an area/volume.
$E = \frac{d\Phi}{dA}$	Irradiance/exitance	$\text{Watt } m^{-2}$	Irradiance/exitance: amount of flux (arriving or leaving) per unit of area.
$I = \frac{d\Phi}{d\omega}$	Directional intensity	$\text{Watt } s_r^{-1}$	Directional Intensity: amount of flux per unit solid angle leaving from a point towards a direction ω
$L = \frac{d^2\Phi}{dA \cos \theta \, d\omega}$	Radiance	$\text{Watt } s_r^{-1} m^{-2}$	Radiance: flow of radiation per unity of area and per unit of solid angle



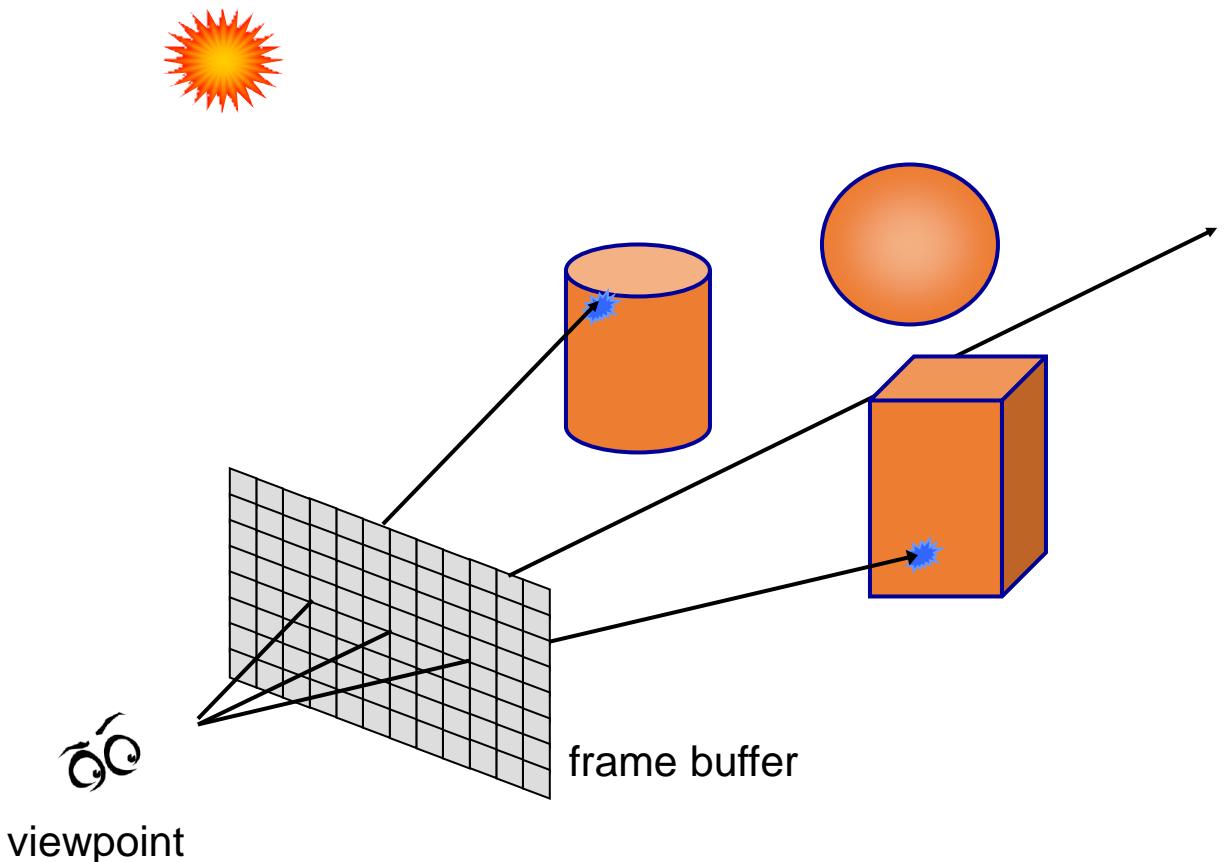
Particle Transport Interpretation

- Let's consider these quantities in terms of travelling *photons*
 - Flux: number of emitted photons;
 - Irradiance: the number of photons arriving on a unit area
 - Radiance: the number of photons travelling through a unit area in a unit direction
- In these terms, the BRDF $f_r(x, \vec{\omega}_i, \vec{\omega}_r)$ can be seen as the *probability* that a photon arriving at x from $\vec{\omega}_i$ is scattered towards $\vec{\omega}_r$
- Path Tracing: evaluating the rendering equation by a **random walk** through the scene starting from the eye



Ray Tracing

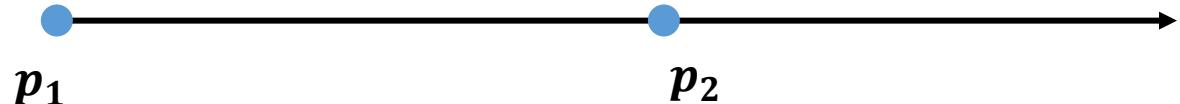
- Key idea: follow the light rays back to where they come from
- For each pixel «shoot» a ray from the eye through that pixel and check if it hits something in the scene
- Looks easy, let's implement it!





Ray Tracing: the “Ray” part

- A **ray** is defined by
 - a start point and a direction.
 - or
 - a start point and another point defining the direction



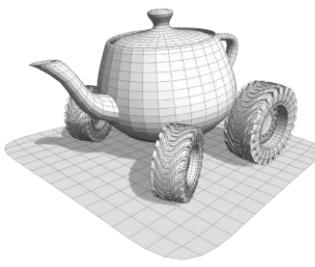
- Parametric equation (as a function of t):

$$\mathbf{p} = (1 - t) \mathbf{p}_1 + t \mathbf{p}_2 \quad \text{with } t > 0$$

$$\mathbf{p} = \mathbf{p}_1 + t (\mathbf{p}_2 - \mathbf{p}_1) \quad \text{with } t > 0$$

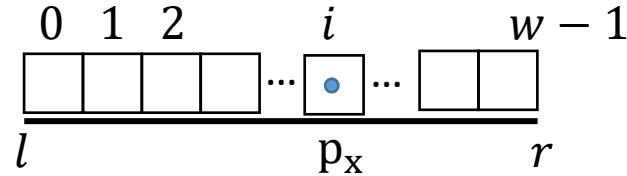
What if $t \in [0,1]$?

What if $t \in [-\infty, \infty]$?



Generate view rays

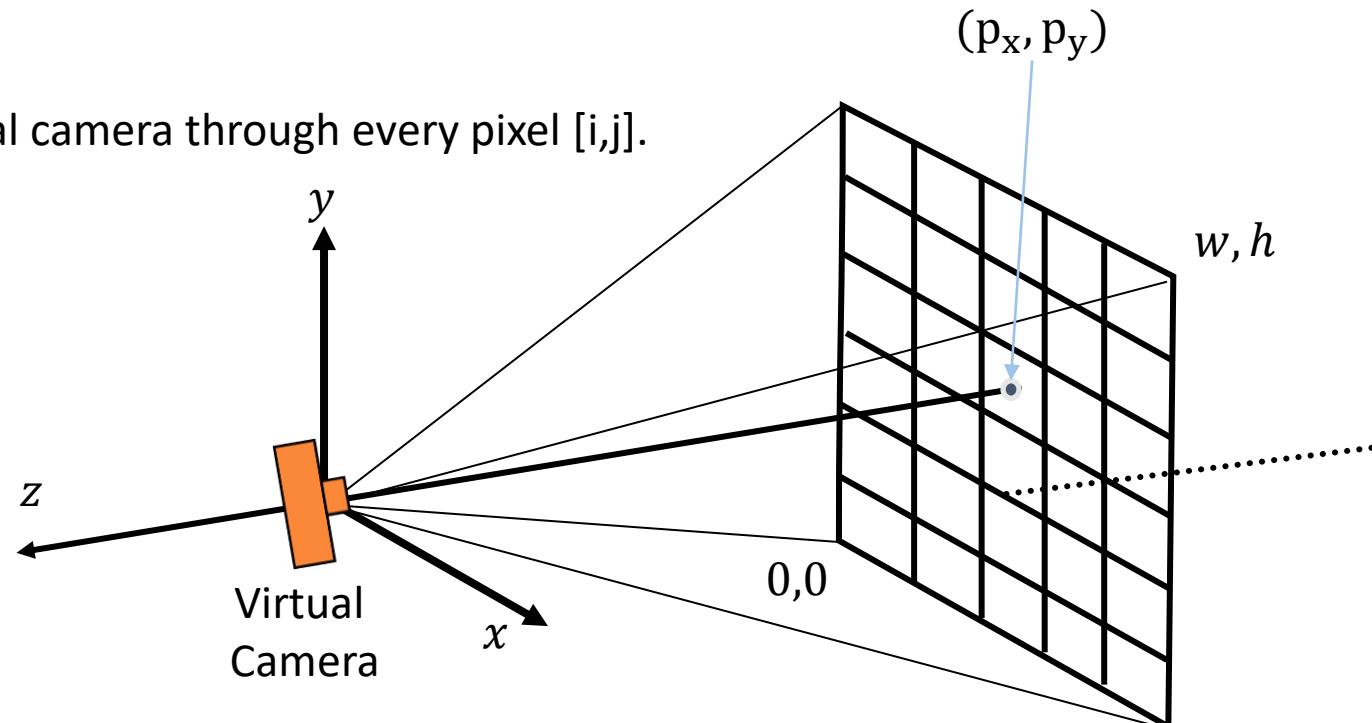
Generate ray starting from eye point of the virtual camera through every pixel $[i,j]$.

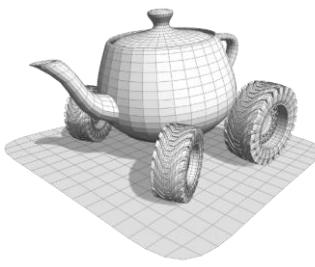


l and r are in view space units (e.g. mm)
Pixels coordinates are in $[0,0] \times [w,h]$

$$\mathbf{p}_x = l + \frac{\left(\frac{1}{2} + i\right)}{w}(r - l) = -\frac{s_x}{2} + \frac{\left(\frac{1}{2} + i\right)}{w}s_x \quad \text{If } r = -l \text{ and } s_x = r - l$$

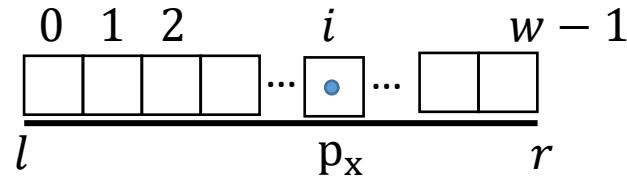
$$\mathbf{p}_y = b + \frac{\left(\frac{1}{2} + j\right)}{h}(t - b) = -\frac{s_y}{2} + \frac{\left(\frac{1}{2} + j\right)}{w}s_y \quad \text{If } t = -b \text{ and } s_y = t - b$$





Generate view rays (in view space)

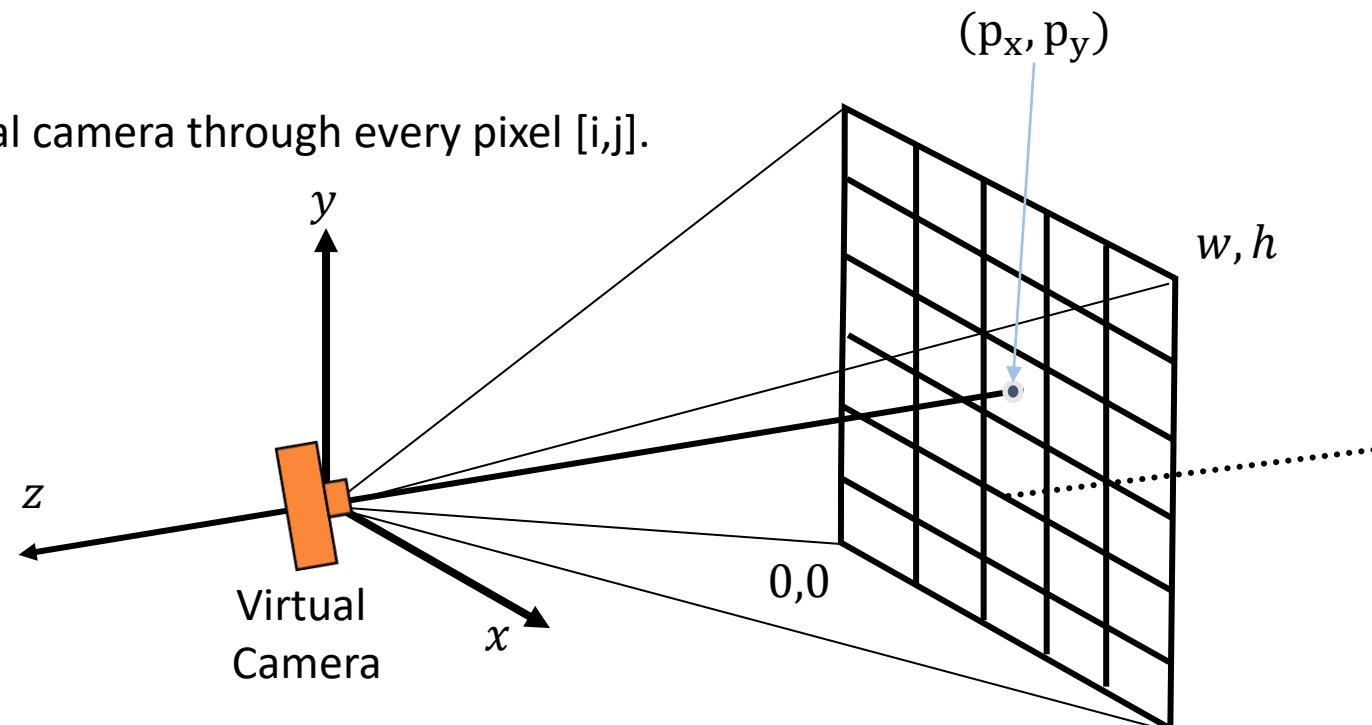
Generate ray starting from eye point of the virtual camera through every pixel $[i,j]$.

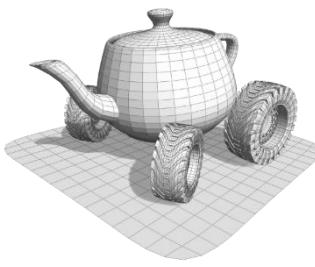


l and r are in view space units (e.g. mm)
Pixels coordinates are in $[0,0] \times [w,h]$

$$\mathbf{p}_x = l + \frac{\left(\frac{1}{2} + i\right)}{w}(r - l) = -\frac{s_x}{2} + \frac{\left(\frac{1}{2} + i\right)}{w}s_x \quad \text{If } r = -l \text{ and } s_x = r - l$$

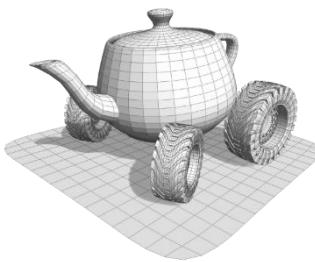
$$\mathbf{p}_y = b + \frac{\left(\frac{1}{2} + j\right)}{h}(t - b) = -\frac{s_y}{2} + \frac{\left(\frac{1}{2} + j\right)}{w}s_y \quad \text{If } t = -b \text{ and } s_y = t - b$$





Ray Tracing: the «tracing» part

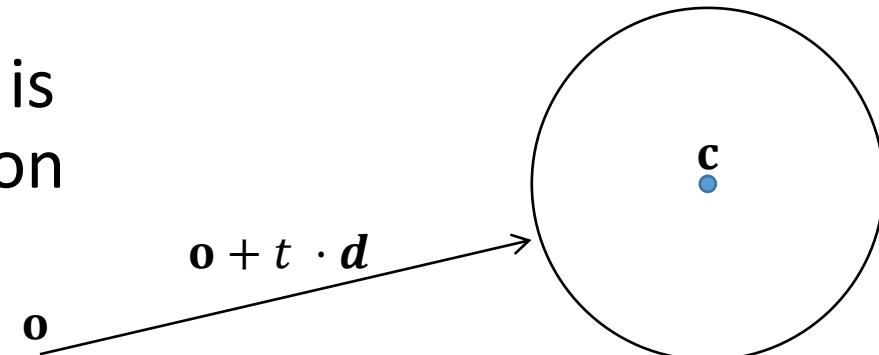
- The basic building block of **tracing** is finding the intersection of the ray with the (objects of the) scene.
- The objects can be described in several different ways (remember?)
 - Parametric
 - Implicit
 - Polygon meshes
 - Voxels (for volumes)
 - Constructive Solid Geometry (CSG)



Ray-Sphere intersection

- A sphere centered at \mathbf{c} with radius r is described with the implicit formulation

$$S = \{\mathbf{p} \in \mathbb{R}^3 \mid (\mathbf{p} - \mathbf{c})(\mathbf{p} - \mathbf{c}) = r^2\}$$



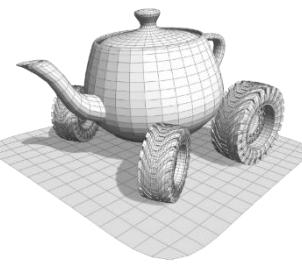
- Find if for some t , the ray $\mathbf{o} + t \cdot \mathbf{d}$ belongs to S

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c})(\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2$$

$$At^2 + Bt + C = 0$$

$$((\mathbf{o} - \mathbf{c}) + t\mathbf{d})((\mathbf{o} - \mathbf{c}) + t\mathbf{d}) = r^2$$

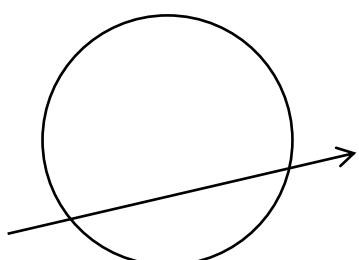
$$\underbrace{(\mathbf{o} - \mathbf{c})(\mathbf{o} - \mathbf{c}) - r^2}_{C} + t^2\mathbf{d}^2 + t \underbrace{2\mathbf{d}(\mathbf{o} - \mathbf{c})}_{B} = 0 \quad \Rightarrow \quad t = \frac{-B \pm \sqrt{(B^2 - 4AC)}}{2A}$$



Ray-Sphere intersection

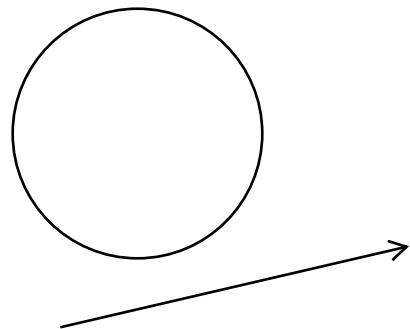
- The solutions of the 2° degree equation tell in which case we are

$$t_1 \neq t_2, \\ t_1, t_2 > 0.0$$



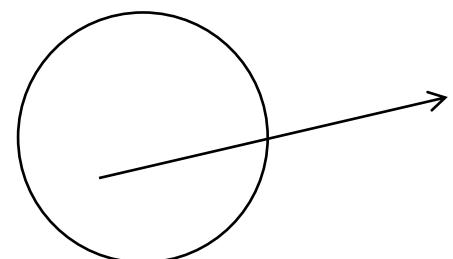
The ray hits the
sphere at $\min(t_1, t_2)$

$$t_1, t_2 \in \mathbb{C}$$



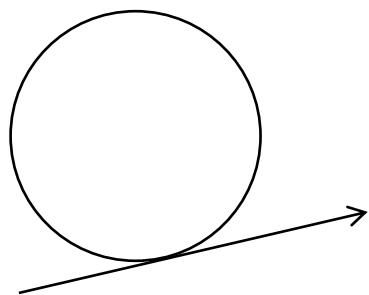
The ray misses the
sphere

$$t_1 \neq t_2, \\ t_{\min} < 0, \\ t_{\max} > 0.0$$

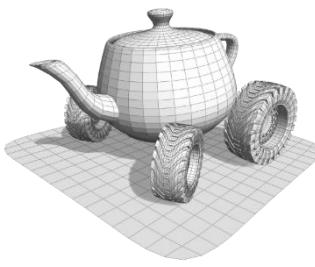


The origin of the ray
is inside the sphere

$$t_1 = t_2$$



The ray is tangent to
the sphere



Ray-Plane intersection

- A plane can be described as a point on the plane and a normal

$$Pl = \{ \mathbf{p} \in \mathbb{R}^3 \mid (\mathbf{p} - \mathbf{c}) \mathbf{n} = \mathbf{0} \}$$

- Substituting the ray equation for p

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c})\mathbf{n} = \mathbf{0}$$

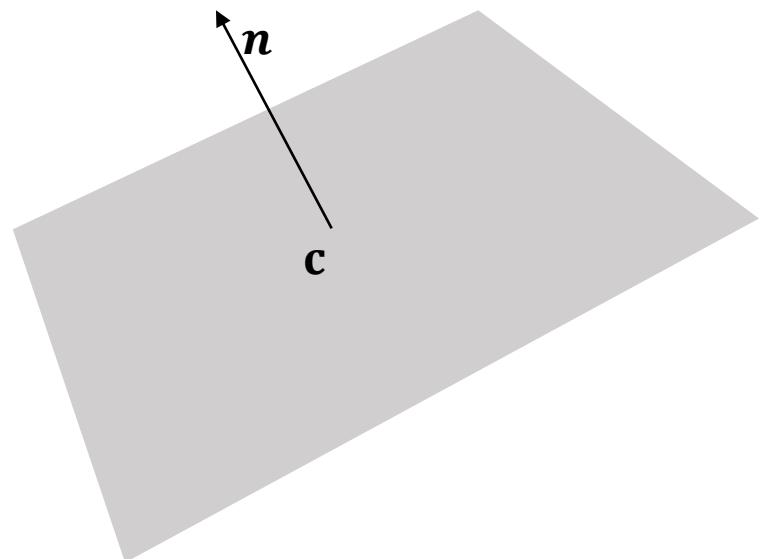
$$t = -\frac{(\mathbf{o} - \mathbf{c})\mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

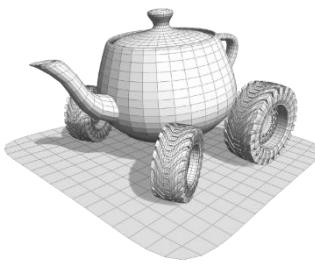
$\mathbf{d} \cdot \mathbf{n} \neq \mathbf{0} \Rightarrow$ one solution

$\mathbf{d} \cdot \mathbf{n} = \mathbf{0} \Rightarrow$ ray parallel to plane

$\mathbf{d} \cdot \mathbf{n} = \mathbf{0}$ and $(\mathbf{o} - \mathbf{c}) \cdot \mathbf{n} = \mathbf{0} \Rightarrow$ ray included in plane

$t > 0 \Rightarrow$ the view ray intersects the plane



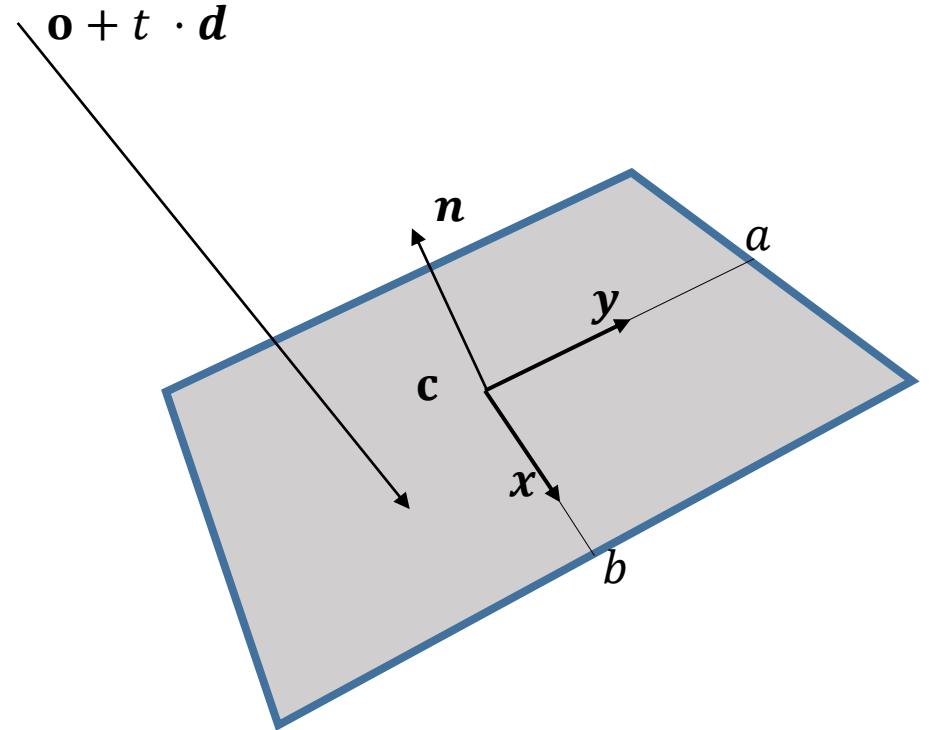


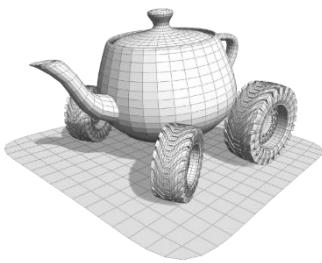
Ray-Quad intersection

- We need a reference frame $F = \{\mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{n}\}$ to define the quad as:

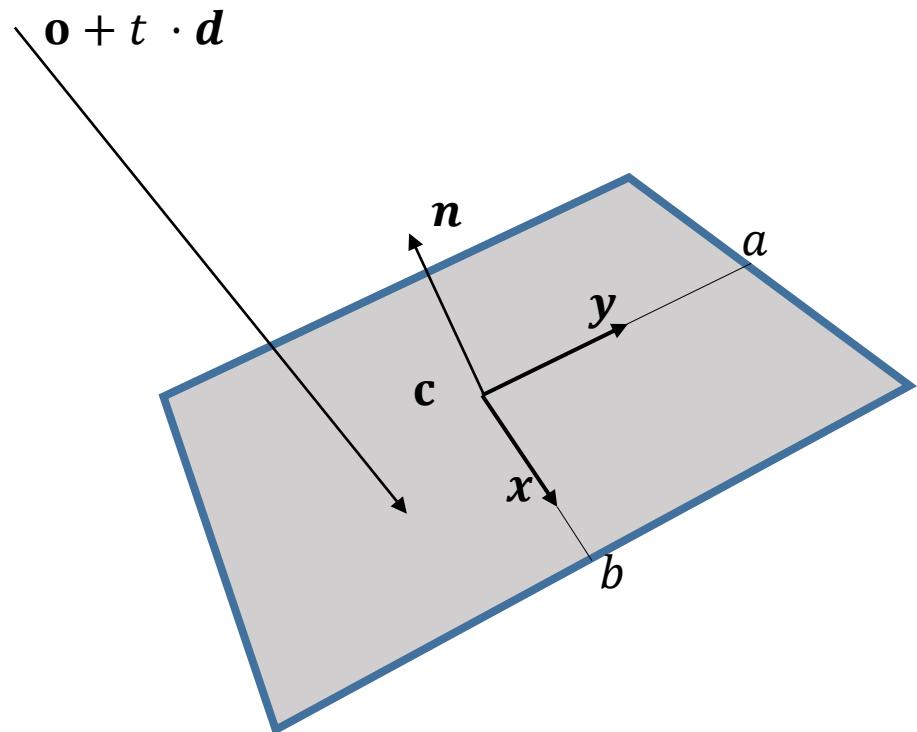
$$Q = \{(x, y, 0) \mid -a < x < a, -b < y < b\}$$

- Intersection test: we *could*
 1. Do ray-plane intersection test and find t_i
 2. Express the intersection point $\mathbf{o} + t_i \cdot \mathbf{d}$ in F and test x and y
- Intersection test: we do
 1. Express the ray in F
 2. Do **ray-XY** plane intersection test and find t_i
 3. Test x and y





World or View Space VS Object Space

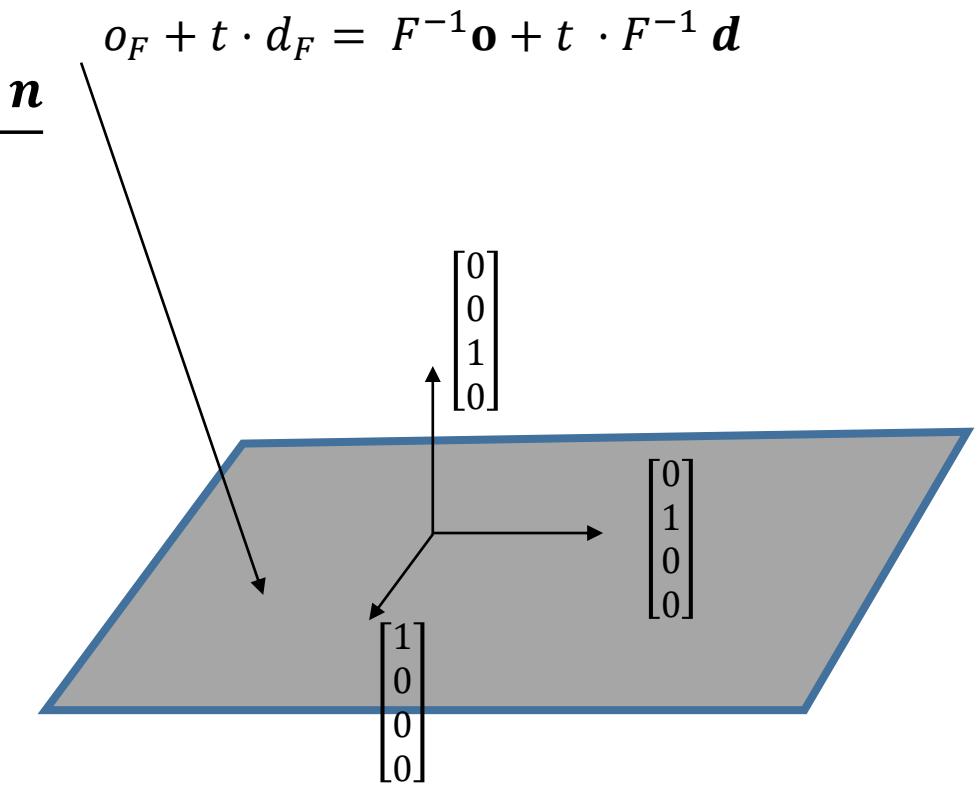


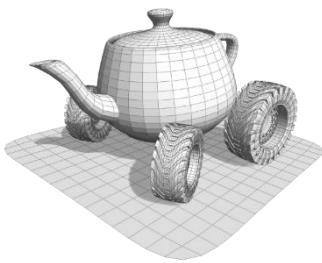
$$t = -\frac{(\mathbf{o}_F - \mathbf{c}_F) \cdot \mathbf{n}}{d_f \cdot \mathbf{n}}$$

$$\mathbf{n} = [0, 0, 1, 0]^T$$

$$t = -\frac{\mathbf{o}_{F_z}}{d_{f_z}}$$

The intersection test is now one dimensional

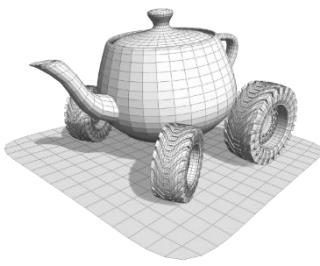




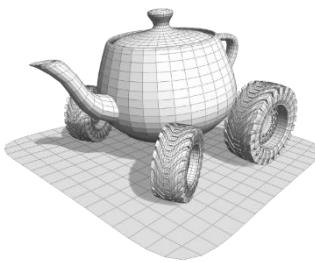
Intersection space

- Primitives are normally defined in their own frame
- Transforming objects more complex than spheres (which are rotationally invariant) and planes may lead to significantly more complex and cumbersome intersection tests
- In real scenes expressing the intersection points in the frame of the object will be required anyway (texturing, anisotropic lighting ...)
- Testing in object space is:
 - More elegant
 - Works for every objects
 - Simplifies the intersection test

IMPLEMENTATION

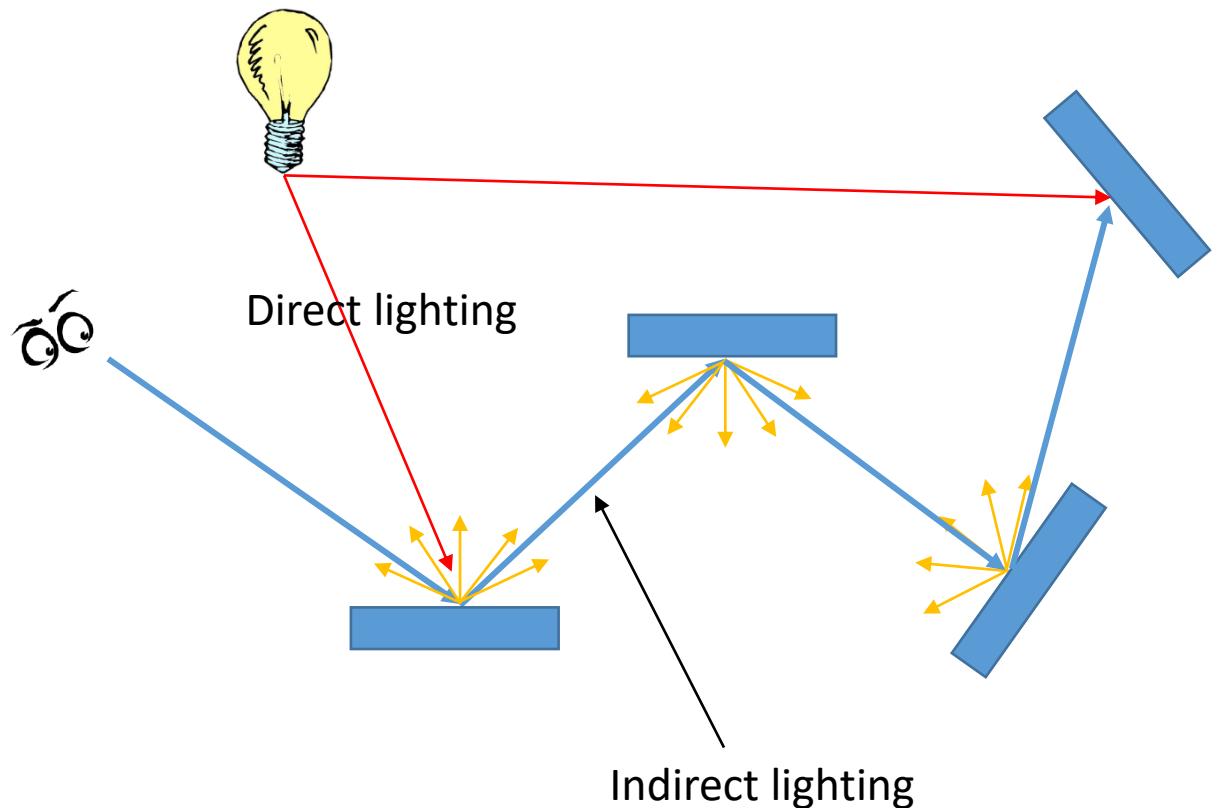


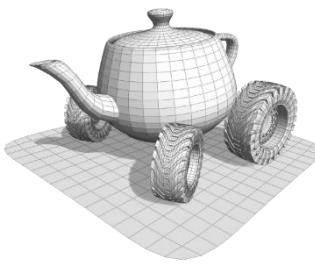
- [CGCourse Unipi Raytracer I \(shadertoy.com\)](#)



Next Event Estimation

- *Hardcore* reverse path tracing:
 - The ray bounces until it hits an emissive surface (or an environment lighting)
- For a point light we would get a black screen
- **NEE:** Decompose $L_r(x, \vec{\omega}_r)$ in
 - Direct lighting: light arriving at x from a light source
 - Indirect lighting: light arriving at x after scattering





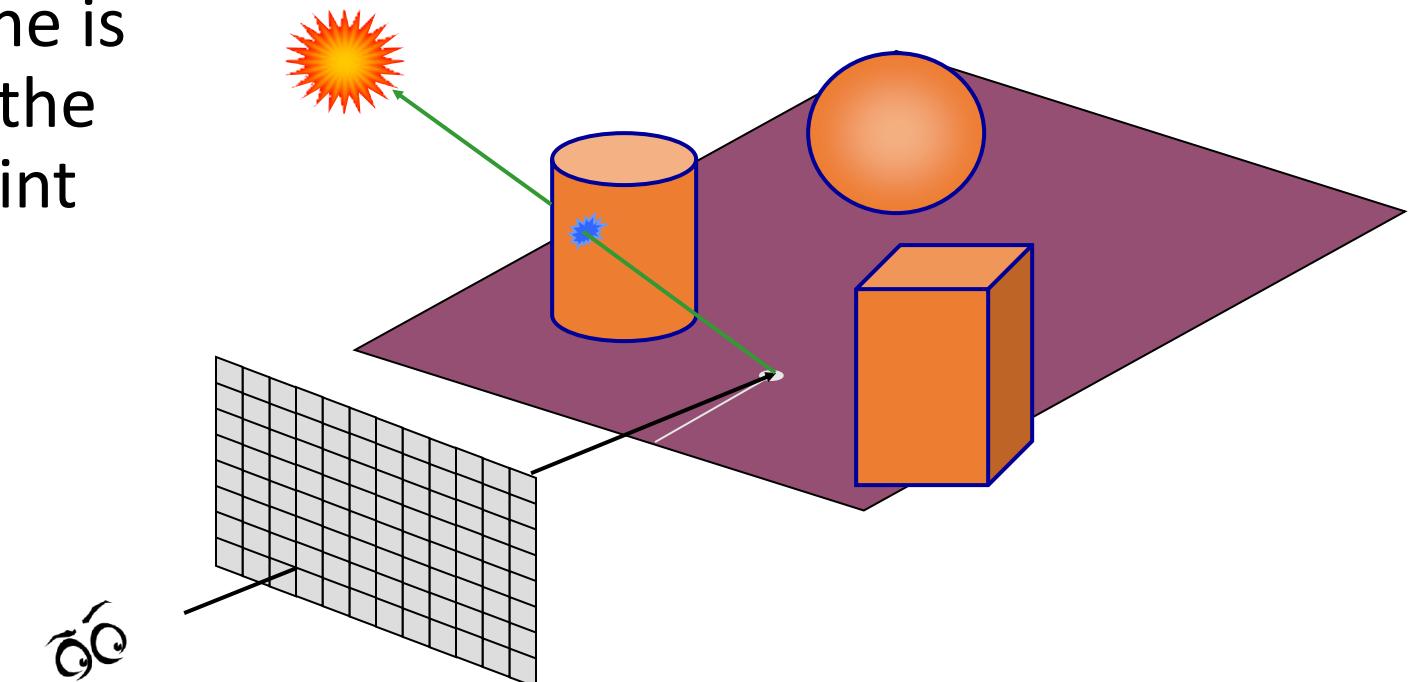
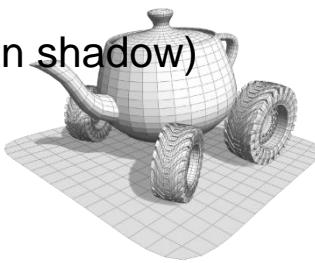
Lighting and Shading

- Ray Tracing ≠ Physically Based Rendering
 - Ray Tracing may produce more realistic images thanks to the more exhaustive sampling of the scene but it's complementary to the lighting model
 - We can still use our approximated BRDF and the result will be impressive
- Raytracer II :
 - direct lighting only (that is, no bounces)
 - Shadow due to a pointlight

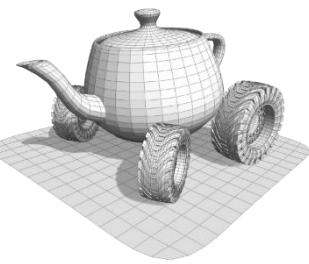
Shadow ray

- Easy to handle: shadows (sharp ones)
- If the intersection ray-scene is found, trace another ray (the **shadow ray**) from that point to the position of the lightsources.
*If it intersects the scene
than is in shadow*

Shadow ray (if it intersect something it is in shadow)



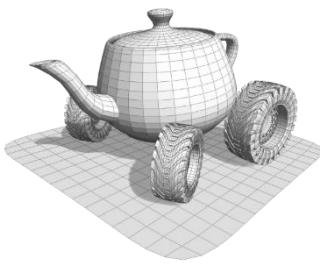
From lecture...



Shadow Ray

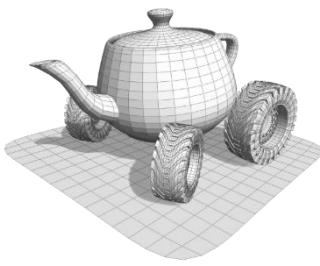
- When a ray hits a surface generate a new ray towards each point light
- Shadow acne (even here??). The problem is the same as for the Shadow Maps *but* we are not constrained with shadow maps and depth buffer rasterization:
 - Count the intersection only from $t_{min} \geq \epsilon$, that is, apply a small bias and you'll be fine

Implementation

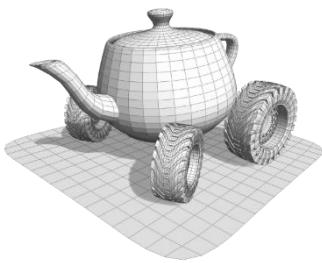


- <https://www.shadertoy.com/view/7t2BzK>

Sampling problems

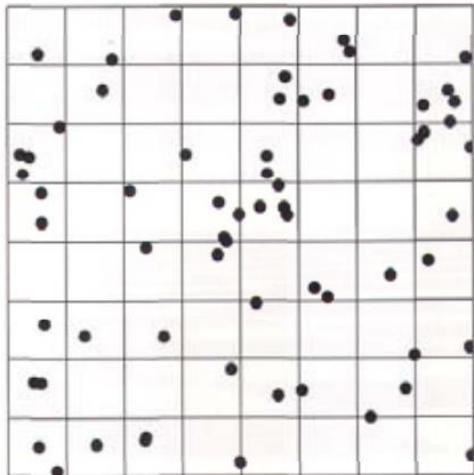


- Sampling is a fundamental block of ray tracing (of rendering, really)
- Aliasing
 - On the silhouette of the objects
 - On the silhouette of the shadows
 - On the highlights
- With the Rasterization Pipeline & OpenGL, we have MSAA and SSAA
- How to do antialiasing ?
 - Generate more samples per pixels and average



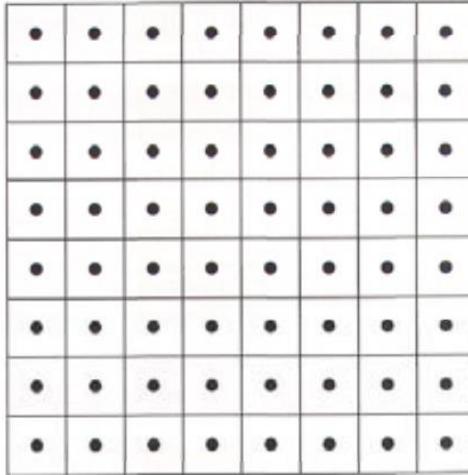
Jittering

Random



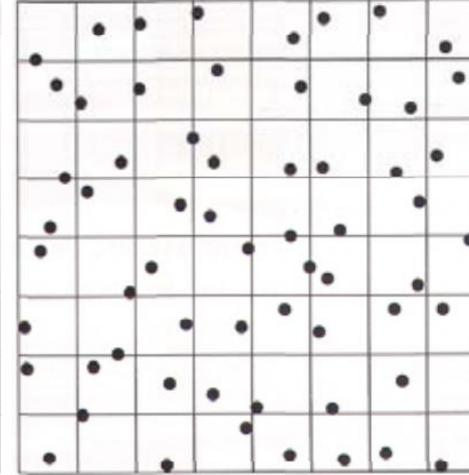
Domain not uniformly sampled

Uniform



Domain uniformly but not in a random way

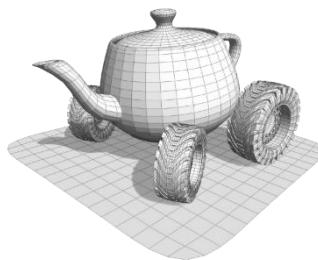
Jittered



Uniform & random.
Trading aliasing for noise

```
S JitteredSampling(){  
    for each Cell in GRID  
        S = S + RandomPointInTheCell()  
    return S  
}
```

Jittering: from aliasing to noise



256 samples per pixel as reference



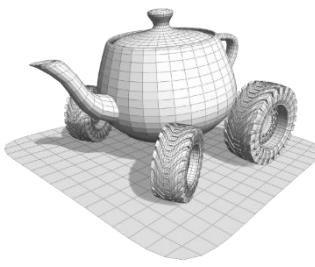
1 sample per pixel (no jitter)



1 sample per pixel (jittered)



4 samples per pixel (jittered)



Indirect Lighting

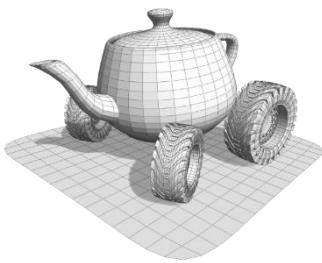
- Recall the Rendering Equation

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) \text{ BRDF } L_i(x, \vec{\omega}_i) \text{ environment } \cos(\theta) d\vec{\omega}_i$$

- We made the drastic simplification of considering only the **direct lighting**, that is, the light arriving from light emitting surfaces.

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \sum_{i=0}^{NLights} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i \cos(\theta)$$

- With path tracing we can **approximate that integral** instead



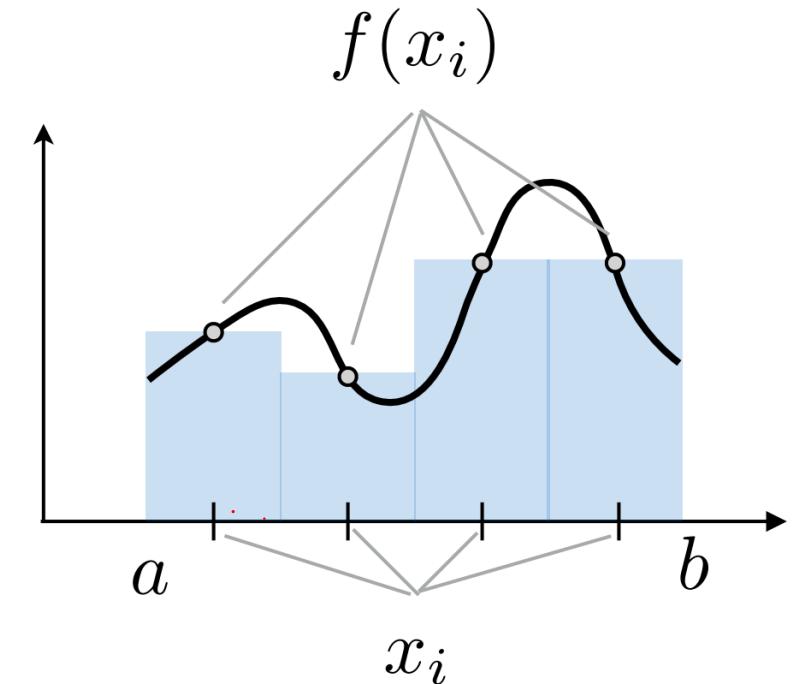
Montecarlo Integration

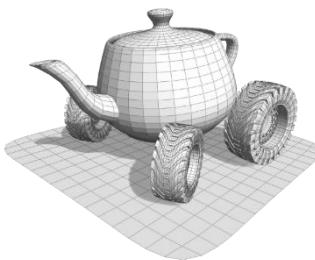
- The Rendering Equation involves integrating over an integral. How can it be done?
- **Numerical Quadrature:** sample the function at n equal spaces positions and approximate the integral with the corresponding boxes

$$I = \int_a^b f(x)dx$$

$$I \approx \sum_{i=1}^N f(x_i) \frac{b-a}{N}$$

$$x_i = a + \left(i - \frac{1}{2}\right) \frac{b-a}{N}$$





Montecarlo Integration

$$I = \int_a^b f(x) dx$$

Definition of first momentum (mean value)

It can be estimated as:

Then consider:

$$E[g(x)] = \int_a^b g(x)p(x) dx$$

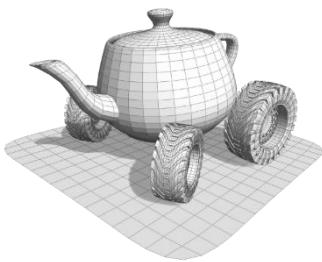
$$E[g(x)] \approx \frac{1}{N} \sum_{i=0}^{N-1} g(x_i)$$

Which we did for
PCF shadow
mapping

$$g(x) = \frac{f(x)}{p(x)} \text{ with uniform } p(x) = \frac{1}{b-a}$$

$$I = \int_a^b f(x) dx = \int_a^b \frac{f(x)}{p(x)} p(x) dx \Rightarrow I \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(x_i)}{p(x_i)}$$





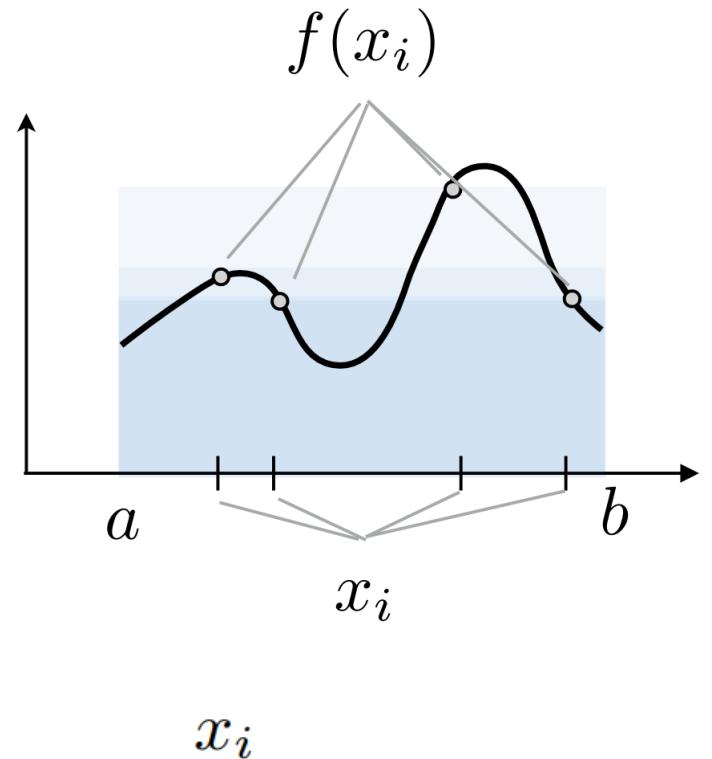
Montecarlo Integration

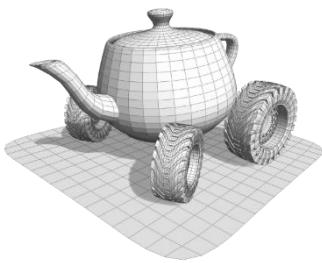
- Pick the value of the function at uniform random positions and average their value

$$\text{since } p(x_i) = \frac{1}{b-a}$$

$$I \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(x_i)}{p(x_i)} = \frac{1}{N} \sum_{i=0}^{N-1} f(x_i)(b-a)$$

$$I \approx \frac{1}{N} \sum_{i=0}^{N-1}$$

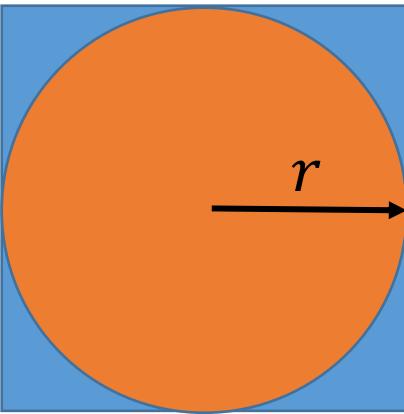




Example: Compute π

- π is the area of the unit circle ($r = 1$) and it can be written as

$$\pi = \int_{-1}^1 \int_{-1}^1 f(x, y) dx dy, \quad f(x, y) = \begin{cases} 1, & x^2 + y^2 \leq 1 \\ 0, & \text{otherwise} \end{cases}$$



- Therefore its Montecarlo approximation is:

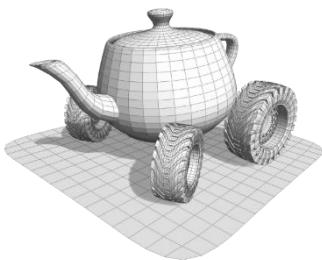
$$\pi \approx \frac{1}{N} \sum_i \frac{f(x_i, x_j)}{p(x_i, x_j)} = \frac{4}{N} \sum_i f(x_i, x_j)$$

$p(x_i, x_j) = \frac{1}{4}$

$$Area(\text{square}) = 4r^2$$

$$Area(\text{circle}) = \pi r^2$$

$$\pi = 4 \frac{Area(\text{circle})}{Area(\text{square})}$$



Example: Monte Carlo Integration

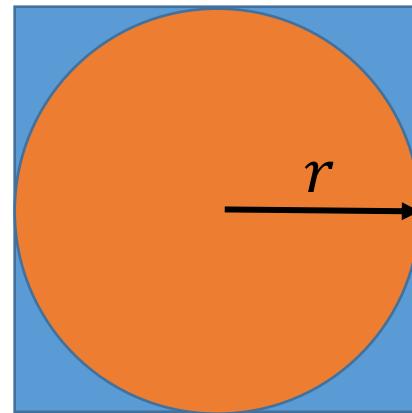
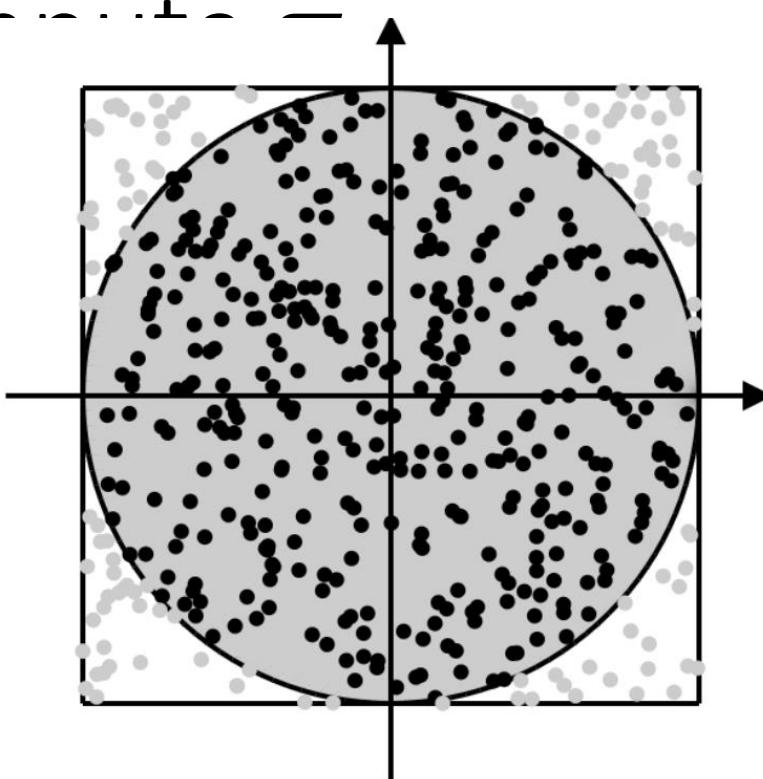
- π is the area of the unit circle, which can be written as

$$\pi = \int_{-1}^1 \int_{-1}^1 f(x, y) dx dy$$

- Therefore its Monte Carlo estimate is

$$\pi \approx \frac{1}{N} \sum_i \frac{f(x_i, x_j)}{p(x_i, x_j)} = \frac{4}{N} \sum_i f(x_i, x_j)$$

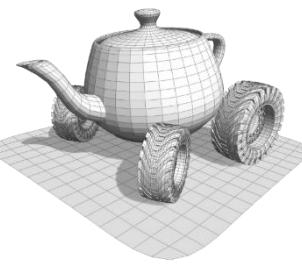
$$p(x_i, x_j) = \frac{1}{4}$$



$$Area(square) = 4r^2$$

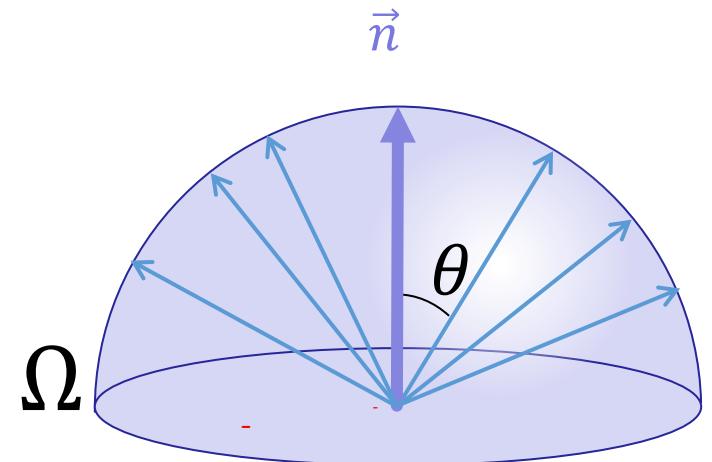
$$Area(circle) = \pi r^2$$

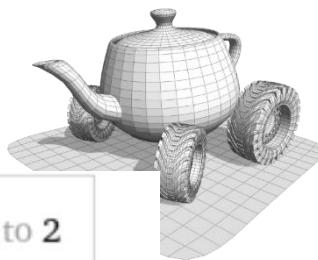
$$\pi = 4 \frac{Area(circle)}{Area(square)}$$



Sampling the Hemisphere

- We could sample the hemisphere uniformly
 1. Generate $p \in [-1,1]^3$
 2. If $|p| > 1$ goto 1
 3. $r = \frac{p}{|p|}$
 4. If $r.y < 0 r = -r$
- However, Montecarlo integration will converge faster (that is, with less samples) if the sampling distribution is similar to the function to integrate
- **Importance sampling** adapts the sampling to the BRDF





Importance sampling

- Example: we want to integrate

$$f(x) = \cos\left(\frac{\pi}{16}x\right)\left(1 - \left(\frac{x}{2}\right)^2\right)$$

between -2 and 2

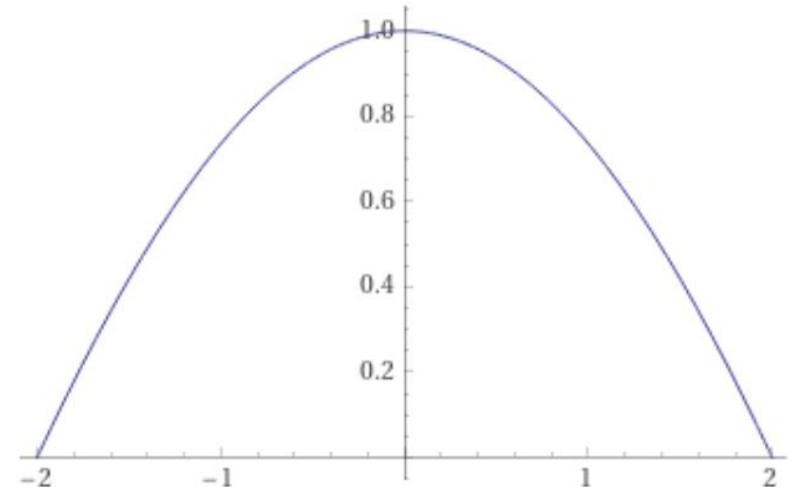
$$\left(\int_{-2}^2 f(x) \approx 2.626\right)$$

- Let's apply MS with as before, with

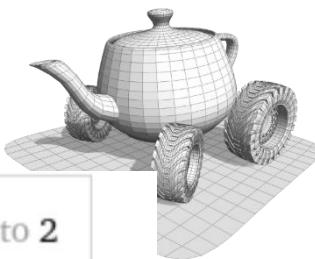
$$p(x_i) = \frac{1}{2 - (-2)} = \frac{1}{4}$$
$$\int_{-2}^2 f(x) \approx \frac{1}{N} \sum_{i=1}^N f(x_i)$$

plot $\cos\left(\frac{\pi}{16}x\right)\left(1 - \left(\frac{x}{2}\right)^2\right)$ $x = -2 \text{ to } 2$

Plot



	draw 1	draw 2	draw 3	draw 4	draw 5
10	2.94341	2.40858	2.76952	2.876	2.64717
100	2.62364	2.56741	2.6706	2.79356	2.66041
1000	2.62495	2.59014	2.60286	2.63216	2.71231
10000	2.6082	2.61996	2.64177	2.62103	2.63204
50000	2.6186	2.63309	2.62388	2.6239	2.63615
100000	2.62671	2.62731	2.62884	2.62103	2.62815



Importance sampling

- The function looks similar to cos between $-\pi/2$ and $\pi/2$, let's try with a cosine distribution

$$q(x) = \cos\left(\frac{\pi}{4}x\right)$$

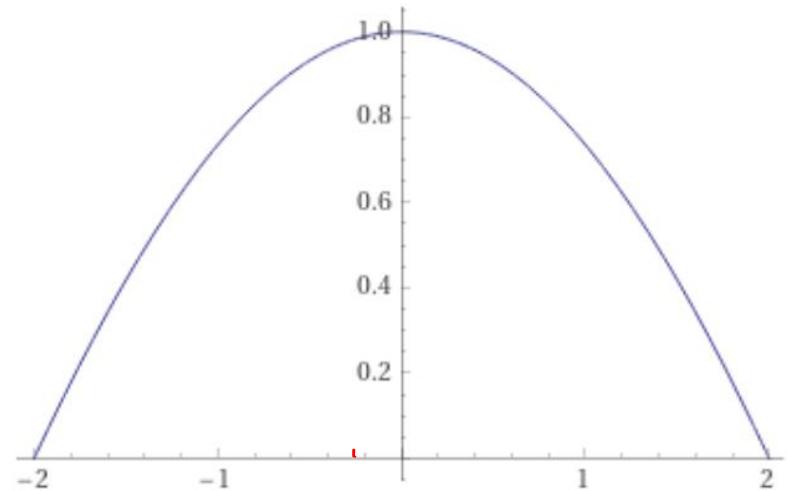
1. Normalize p over the interval so that $\int_a^b p(x) dx = 1$

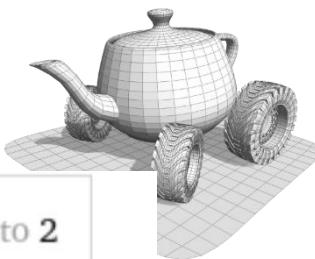
$$\int_a^b \cos\left(\frac{\pi}{4}x\right) dx = \frac{4}{\pi} \sin\left(\frac{\pi}{4}x\right) \Big|_{-2}^b = \frac{8}{\pi} \Rightarrow p(x) = \frac{\cos\left(\frac{\pi}{4}x\right)}{\frac{8}{\pi}}$$

$$\int_{-2}^2 f(x) \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x)}$$

plot $\cos\left(\frac{\pi}{16}x\right)\left(1 - \left(\frac{x}{2}\right)^2\right)$ $x = -2$ to 2

Plot





Importance sampling

- The function looks similar to cos between $-\pi/2$ and $\pi/2$, let's try with a cosine distribution

$$q(x) = \cos\left(\frac{\pi}{4}x\right)$$

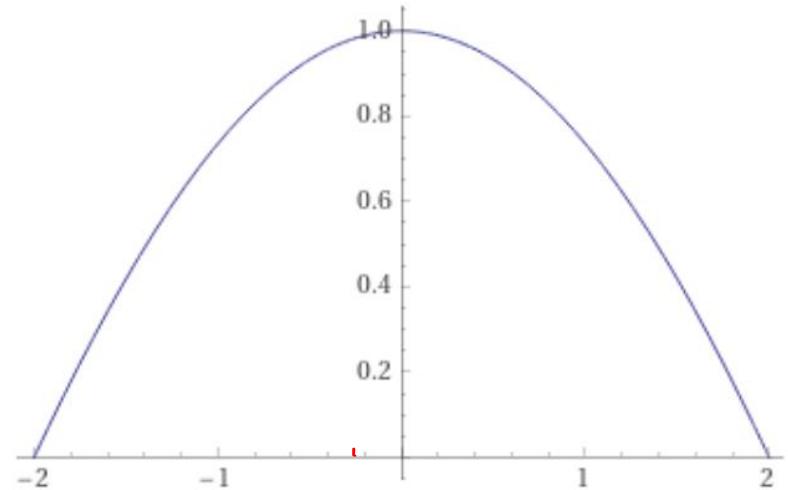
- Normalize p over the interval so that $\int_a^b p(x) dx = 1$

$$\int_a^b \cos\left(\frac{\pi}{4}x\right) dx = \frac{4}{\pi} \sin\left(\frac{\pi}{4}x\right) \Big|_{-2}^b = \frac{8}{\pi} \Rightarrow$$

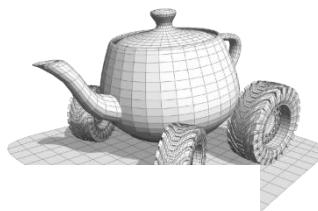
$$\int_{-2}^2 f(x) dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x)}$$

plot $\cos\left(\frac{\pi}{16}x\right)\left(1 - \left(\frac{x}{2}\right)^2\right)$ $x = -2 \text{ to } 2$

Plot



N samples	draw 1	draw 2	draw 3	draw 4	draw
5					
10	2.64991	2.59437	2.61827	2.62912	2.59323
100	2.63657	2.63133	2.62164	2.61984	2.63111
1000	2.62034	2.62826	2.62618	2.62185	2.62924
10000	2.62573	2.6246	2.6254	2.62539	2.62491
50000	2.62652	2.62498	2.62579	2.62558	2.62648
100000	2.62575	2.62591	2.62588	2.62592	2.62565



Importance sampling

- The function looks similar to cos between $-\pi/2$ and $\pi/2$, let's try with a cosine distribution

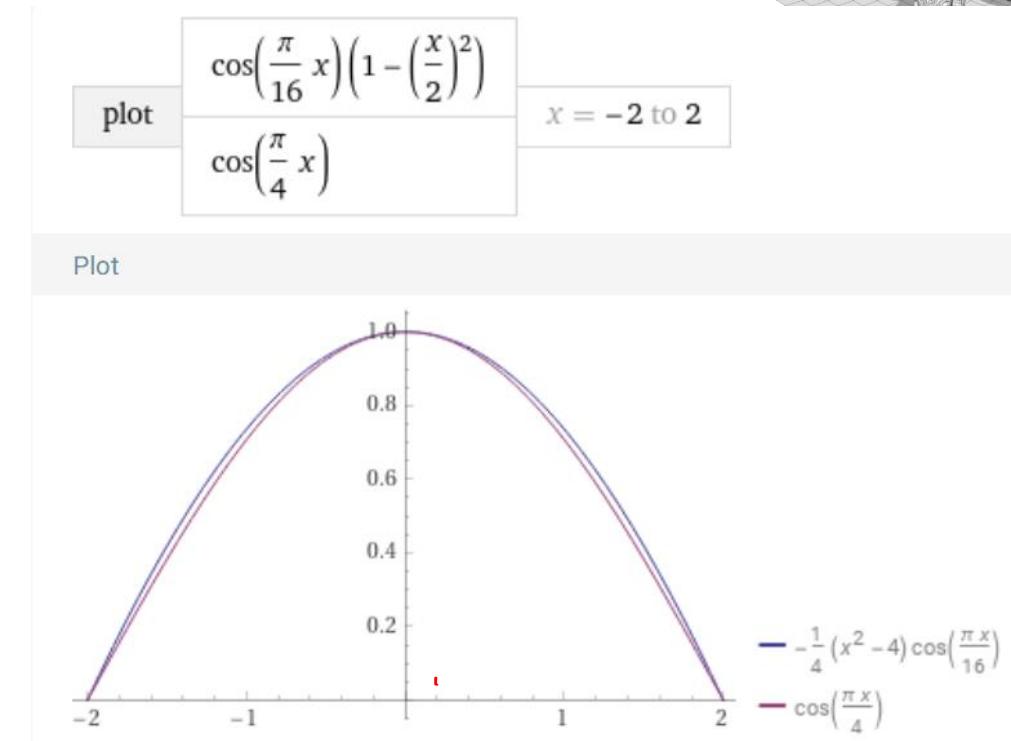
$$q(x) = \cos\left(\frac{\pi}{4}x\right)$$

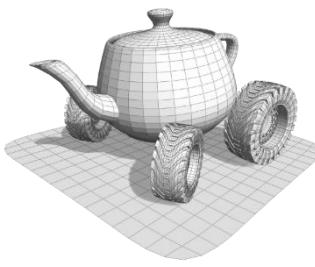
- Normalize p over the interval so that $\int_a^b p(x) dx = 1$

$$\int_a^b \cos\left(\frac{\pi}{4}x\right) dx = \frac{4}{\pi} \sin\left(\frac{\pi}{4}x\right) \Big|_{-2}^b = \frac{8}{\pi} \Rightarrow$$

$$\int_{-2}^2 f(x) dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x)}$$

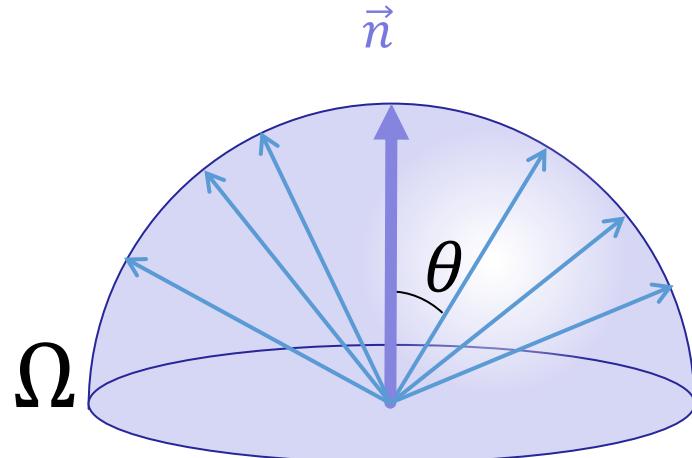
N samples	draw 1	draw 2	draw 3	draw 4	draw
5					
10	2.64991	2.59437	2.61827	2.62912	2.59323
100	2.63657	2.63133	2.62164	2.61984	2.63111
1000	2.62034	2.62826	2.62618	2.62185	2.62924
10000	2.62573	2.6246	2.6254	2.62539	2.62491
50000	2.62652	2.62498	2.62579	2.62558	2.62648
100000	2.62575	2.62591	2.62588	2.62592	2.62565

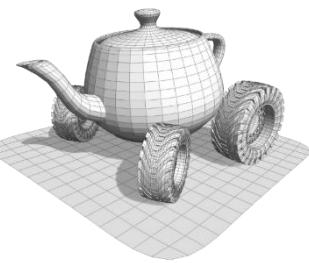




Sampling the Hemisphere

- We could sample the hemisphere uniformly
 1. Generate $p \in [-1,1]^3$
 2. If $|p| > 1$ goto 1
 3. $r = \frac{p}{|p|}$
 4. If $r \cdot y < 0 r = -r$
- However, Montecarlo integration will converge faster if the sampling distribution is similar to the function to integrate
- **Importance sampling** adapts the sampling to the BRDF



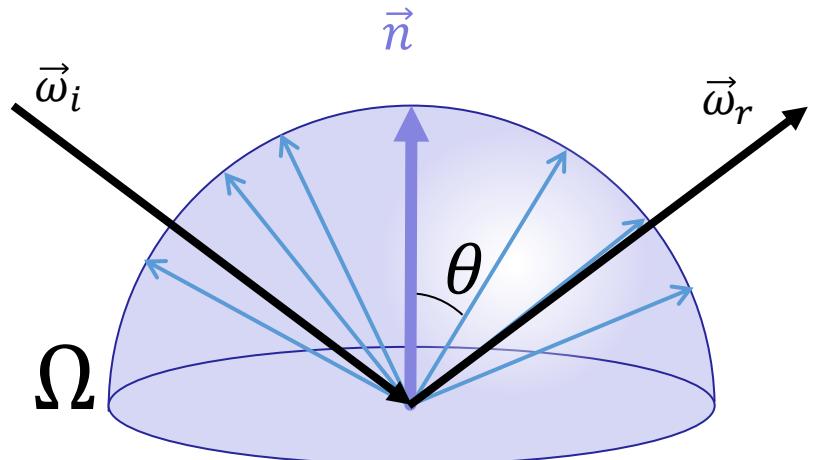


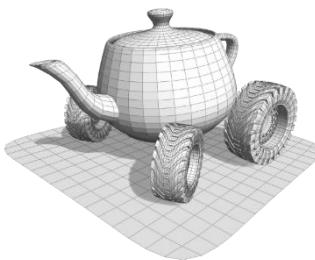
Importance Sampling

- Consider a perfectly specular material:

$$BRDF_{spec} = f(x) = \begin{cases} 1, & \vec{\omega}_i = \text{spec. } \vec{\omega}_r \\ 0, & \text{otherwise} \end{cases}$$

- Why should we sample directions other than $\vec{\omega}_r$?





Importance Sampling Lambertian Surfaces

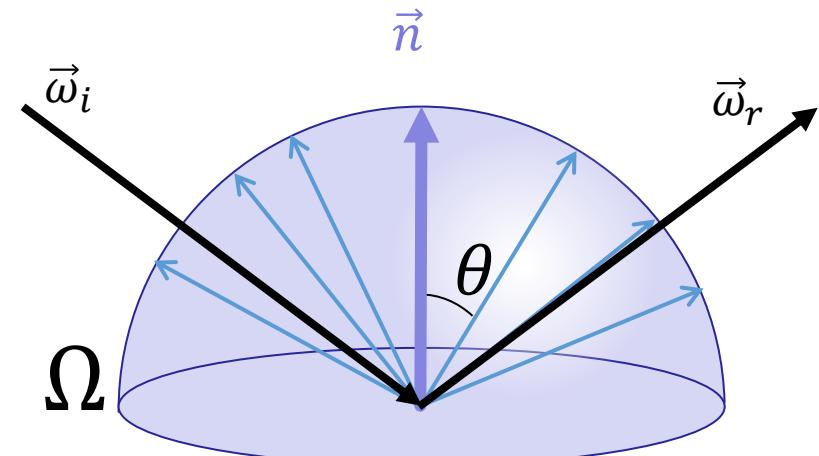
- Lambertian (perfectly diffuse) surface reflect light equally in all directions: $f_r = k$

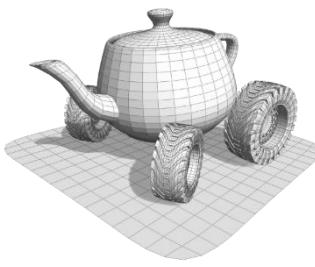
$$\int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) \cos(\theta) d\vec{\omega}_i =$$

$$\int_{\vec{\omega}_i \in \Omega} k \cos(\theta) L_i(x, \vec{\omega}_i) d\vec{\omega}_i$$

this tells that the contributes $L_i(x, \vec{\omega}_i)$ (unknown) will be weighted by $\cos(\theta)$

This is the contribution we don't know about

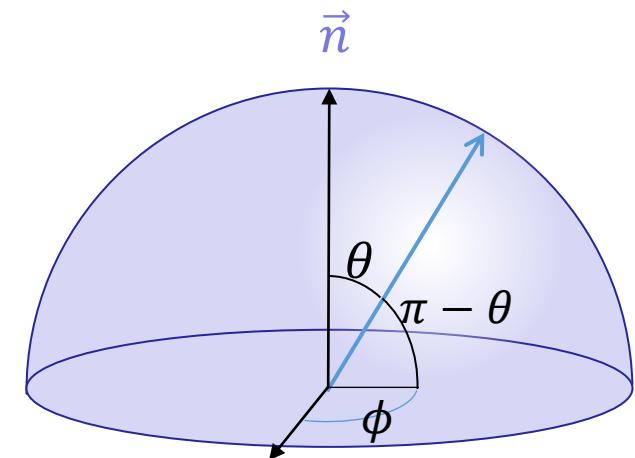


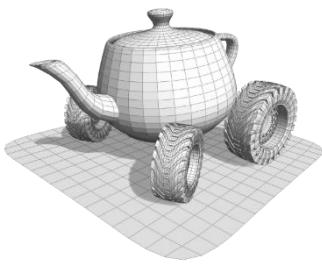


Importance Sampling Lambertian Surfaces

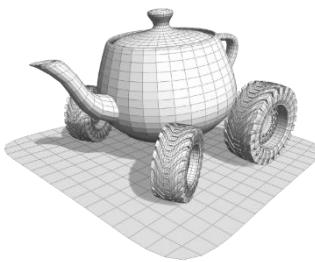
- Recall. How to draw a value from distribution d
 - Draw a uniform value $a \in [0,1]$
 - Take $x = CDF(d)^{-1}(a)$
- direction with cosine distribution in polar coordinates:
 - Draw a value from cosine distribution for the elevation
 - Draw a uniform value in $[0,1]$ for azimuth (b)

$$r = [a^2 \cos(2\pi b), a^2 \sin(2\pi b), \sqrt{1 - a^2}]$$



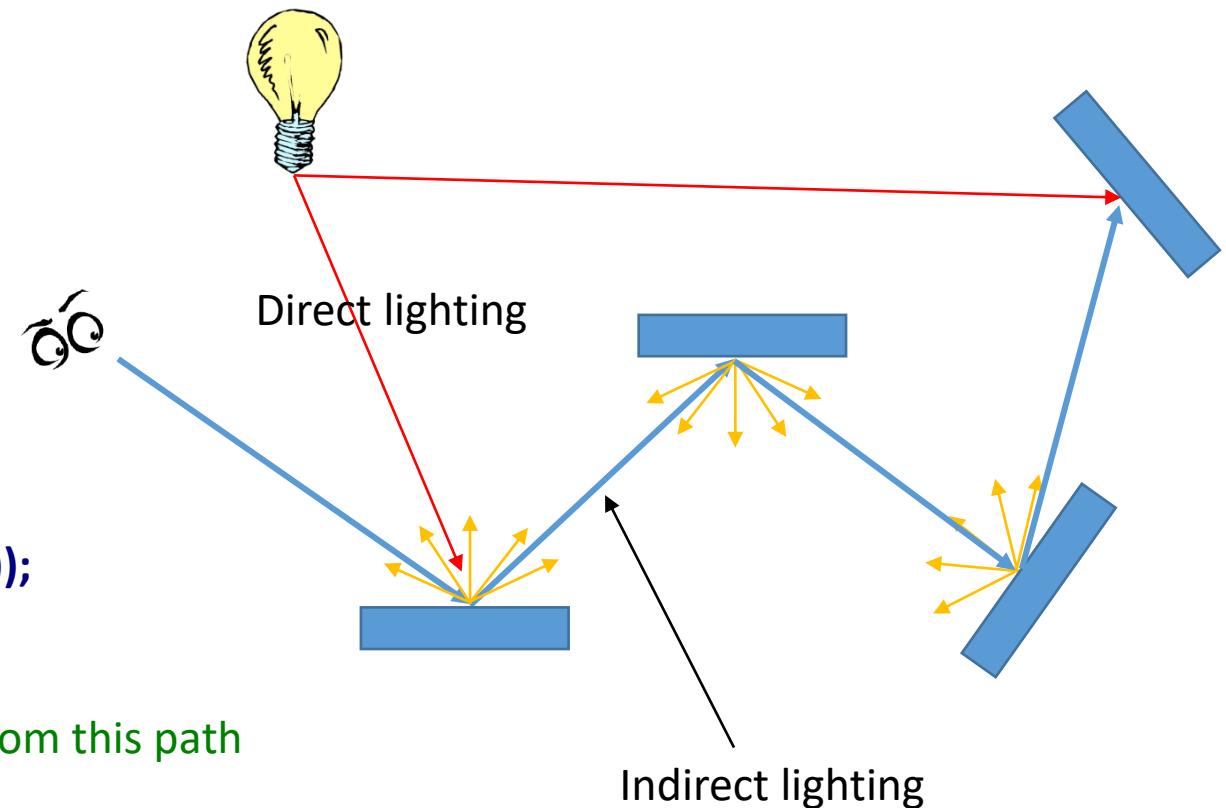


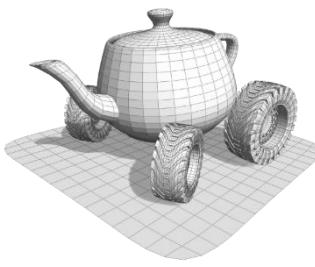
<https://www.shadertoy.com/view/flijfz3>



Path tracing with Next Event Estimation

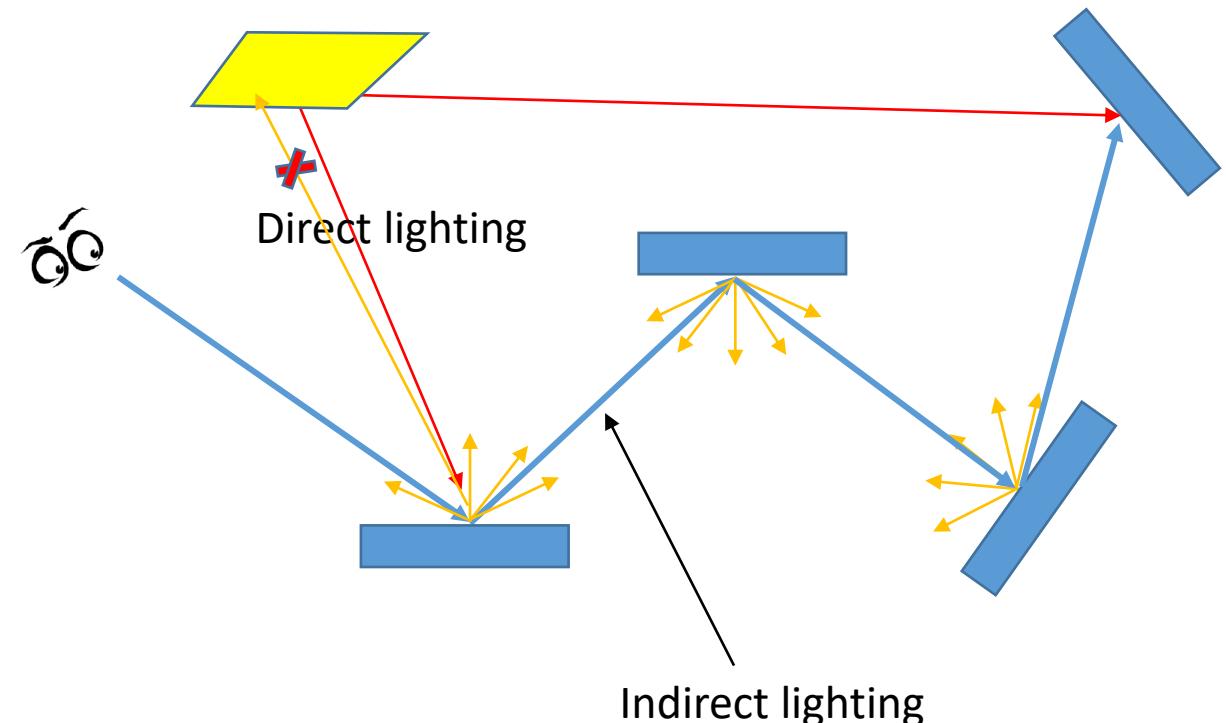
```
ind_L = vec3(1); // maximum indirect light possible  
dir_L = vec3(0); // indirect light  
ray = primary_ray(x,y);  
  
for (int i=0; i< MAX_BOUNCES; ++i){  
    if(hit_scene(ray,reflRay,attenuation)){  
        ind_L *= attenuation;  
        for(each light L_i)  
            dir_L += ind_L*(direct_lighting(L_i));  
    }else{  
        //ind_L*=env_light;  
        ind_L = vec(0); // no indirect light from this path  
    }  
    ray = reflRay;  
}
```

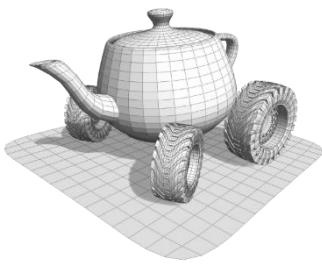




Path tracing with Next Event Estimation

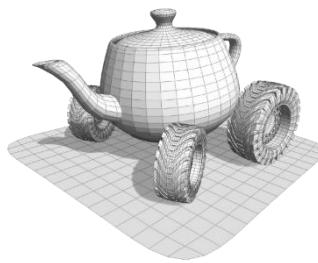
- If direct and indirect lighting are decoupled, rays hitting light sources must be discarded
 - Do not count the same contribution multiple times
- Direct light is weighted by $\cos(\theta)$, indirect light is not (because with importance sampling it is accounted for in the sampling distribution)



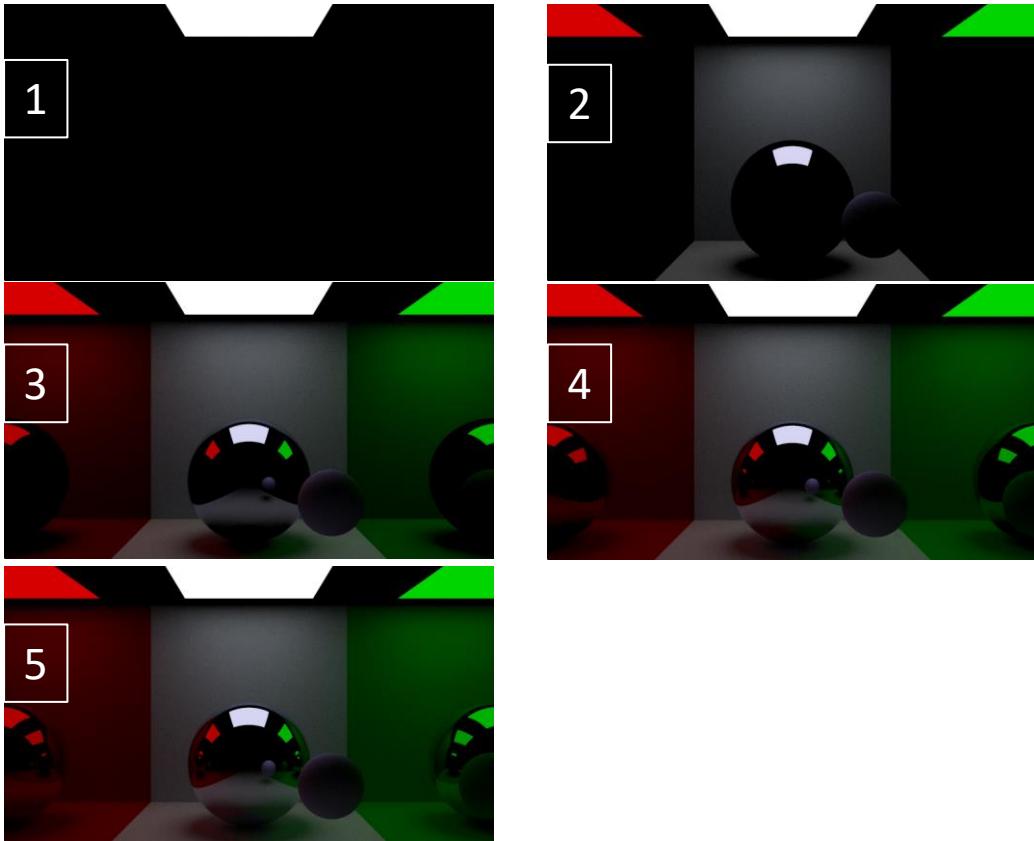


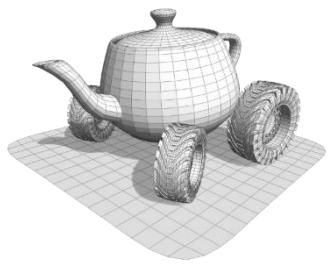
- <https://www.shadertoy.com/view/Nl2fz3>

How many bounces?



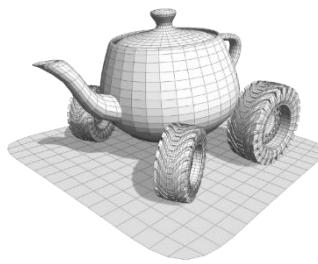
- A very large number: better images but long time for the result
- Just a few: fast results but possibly incorrect images
- What does a “good” number of bounces depend on?
 - The property of the materials
 - The composition of the scene
 - The illumination
 - The *specific path!*





<https://www.shadertoy.com/view/7I2fR3>

Limiting the number of bounces: Russian Roulette



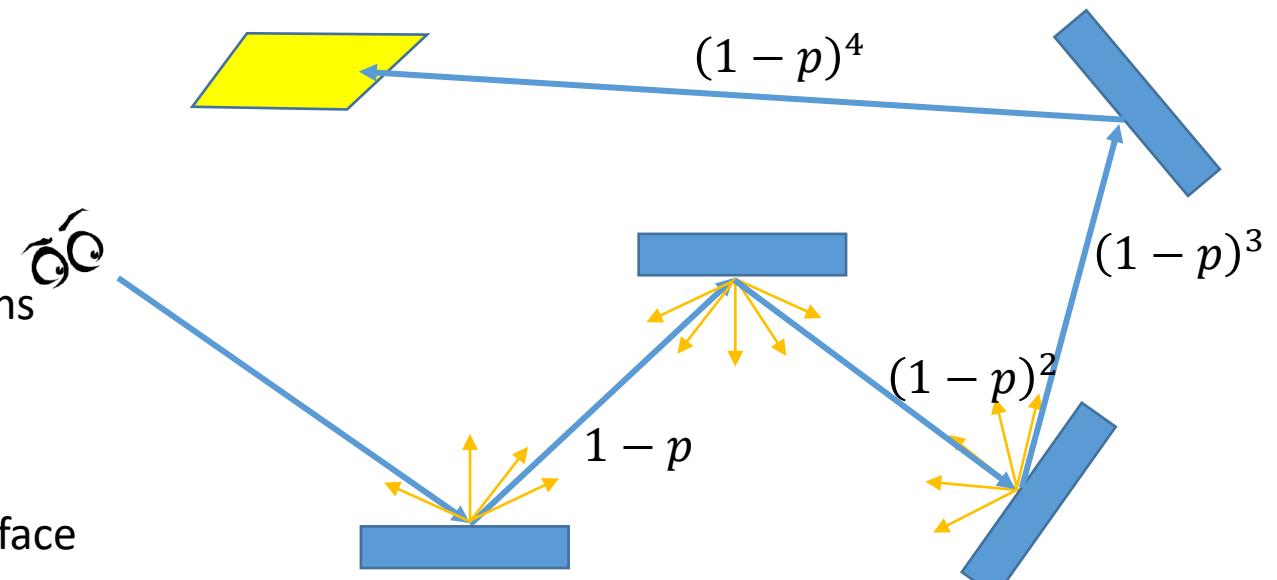
- At each bounce, stop iterating with a probability p
- If a new ray is generated (with probability $(1 - p)$) its contribution is scaled by $\frac{1}{1-p}$

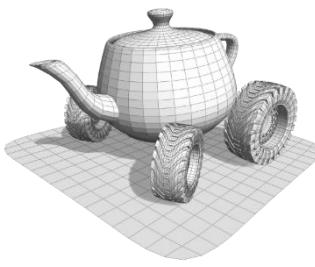
Say we send N primary rays for in the same directions

In this scene made of specular material the light is reached after 4 bounces

Without stopping the iteration until an emissive surface is found the contribution of the N rays will be $N \cdot L$

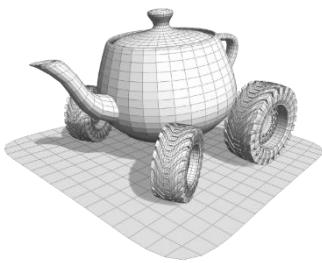
With Russian roulette only $(1 - p)^4 N$ will reach the light, but their contribution is scaled by $\frac{1}{(1-p)^4}$ so the final contribution is still $N \cdot L$





Choosing p

- $p = 0 \rightarrow$ no stopping of the iterations, $p = 1.0 \rightarrow$ ray casting only
- A fixed value for p is a trade-off between time and quality
- A good criterion: set p proportional to the residual light contribution
 - Example: say $\frac{1}{2}$ of the light is absorbed at any bounce. After 8 bounces the remaining part of the light $\frac{1}{2^8} \approx 0.0039$.
 - If we generate another ray and it hits an emissive surface, its contribution will be scaled by 0.0039 anyway -> we can afford approximating

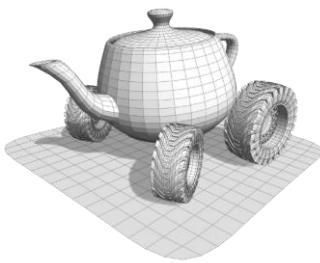
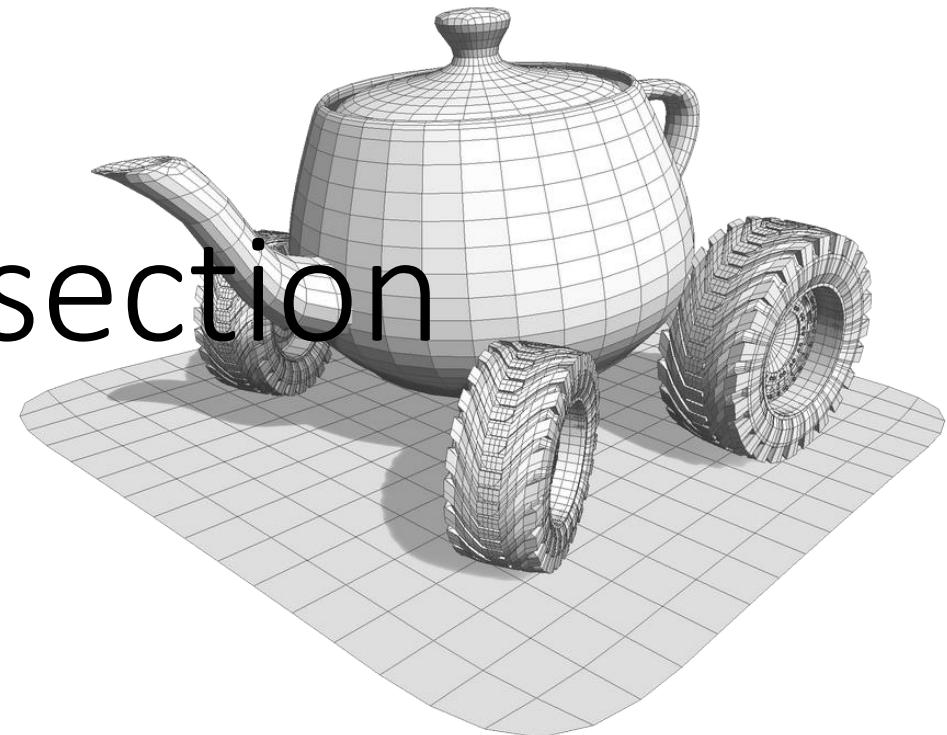


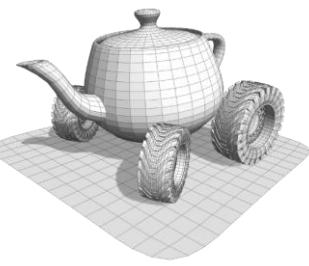
Pseudocode

```
ind_L = vec3(1);    // maximum indirect light possible
dir_L = vec3(0);    // indirect light
ray = primary_ray(x,y);

for (int i=0; i< MAX_BOUNCES; ++i){
    if(hit_scene(ray,reflRay,attenuation)){
        p = (1- ind_L);
        float v = random(0,1);
        if(v<p) i= MAX_BOUNCES ;
        ind_L *= attenuation;
        ind_L /= (1-p);
        for(each light L_i)
            dir_L += ind_L*(direct_lighting(L_i));
    }else{
        ind_L = vec(0); // no indirect light from this path
    }
    ray = reflRay;
}
```

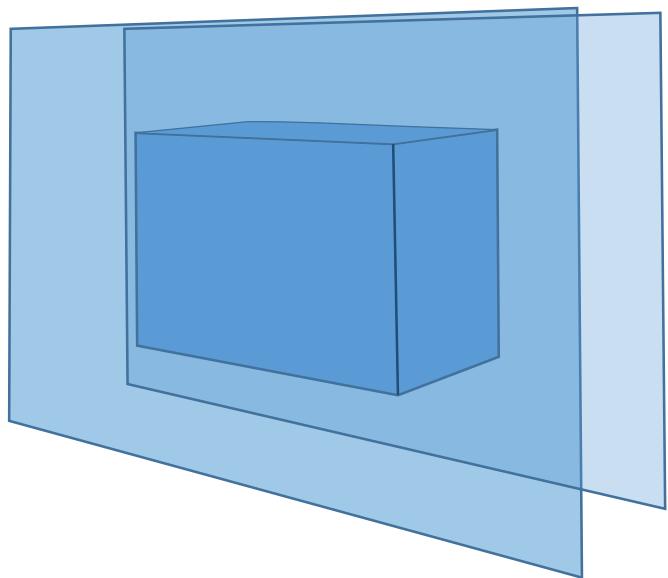
Ray-scene intersection

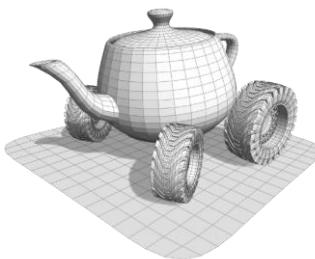




Ray-Box intersection (1/3)

- It can be done by testing for each of its faces as quads (see previous lecture)
- More efficient: use **Slabs**. A slab is a region of space bounded by two parallel planes
- A box is the intersection of 3 slabs





Ray-Box intersection (2/3)

- The algorithm is just just the Liang-Barky one for clipping segments against the view volume in clip space
- AABB = Clip space (or NDC space)
- The segment needs to be clip = the ray intersects the AABB

Clipping segments (3/4)

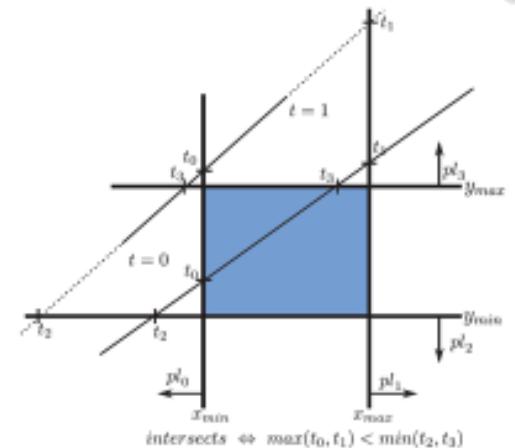
- **Liang-Barsky** use the parametric equation of the line

$$s(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0) = \mathbf{p}_0 + t\mathbf{v}$$

and find the crossing point between the line and the planes, e.g.

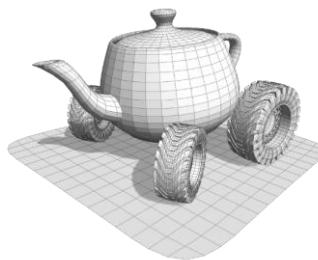
$$(\mathbf{p}_0 + t\mathbf{v})_x = x_{min} \Rightarrow t = \frac{x_{min} - \mathbf{p}_0.x}{\mathbf{v}.x}$$

- the 4 intersections define «entry» and exit «points»
 - entry: from positive to negative halfspace
 - exit : from negative to positive halfspace



t_0 is an entry if $\mathbf{v}_x > 0$, exit otherwise
 t_1 is an entry if $\mathbf{v}_x < 0$, exit otherwise
 t_2 is an entry if $\mathbf{v}_y > 0$, exit otherwise
 t_3 is an entry if $\mathbf{v}_y < 0$, exit otherwise





Ray-Box intersection (3/3)

- The algorithm is just just the Liang-Barky one for clipping segments against the view volume in clip space
- AABB = Clip space (or NDC space)
- The segment needs to be clip = the ray intersects the AABB

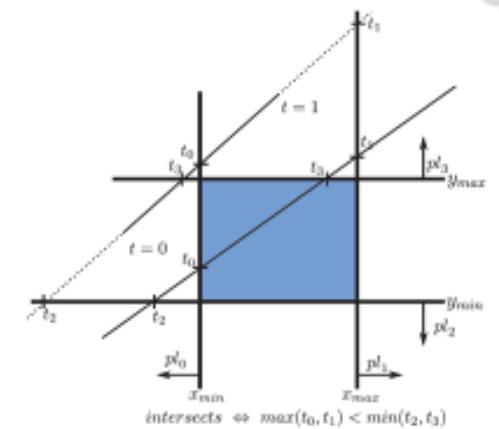
Clipping segments (3/ 4)

- If the first exit point is past the last entry point there is no intersection

$$t_{min} = \max(0, \text{largest entry value})$$

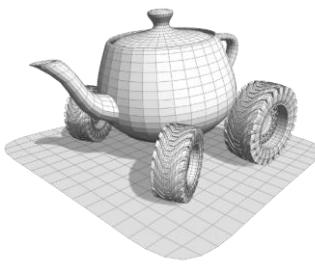
$$t_{max} = \min(1, \text{smallest exit value})$$

- Otherwise $p'_0 = s(t_{min})p'_1 = s(t_{max})$ is the clipped segment



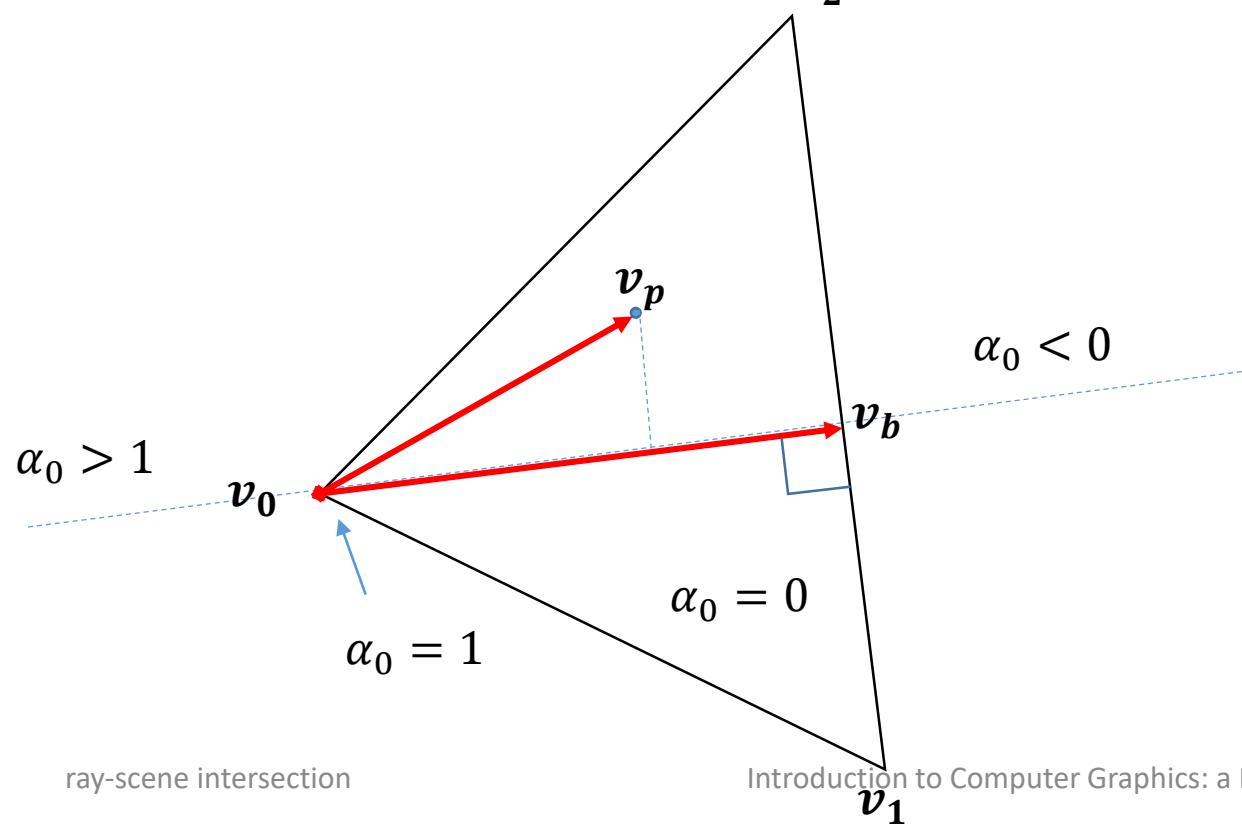
t_0 is an entry if $v_x > 0$, exit otherwise
 t_1 is an entry if $v_x < 0$, exit otherwise
 t_2 is an entry if $v_y > 0$, exit otherwise
 t_3 is an entry if $v_y < 0$, exit otherwise





Ray-triangle Intersection

- First step: find the ray-plane intersection
- Second step: find one barycentric coordinate

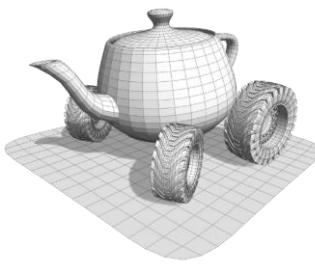


$$\mathbf{v}_{b0} = \mathbf{v}_{20} - \frac{\mathbf{v}_{20}\mathbf{v}_{21}}{|\mathbf{v}_{21}|} \mathbf{v}_{21}$$

$$\alpha_0 = 1 - \frac{\frac{\mathbf{v}_{p0} \cdot \mathbf{v}_{b0}}{|\mathbf{v}_{b0}|}}{\frac{\mathbf{v}_{20} \cdot \mathbf{v}_{b0}}{|\mathbf{v}_{b0}|}} = \frac{\mathbf{v}_{p0} \cdot \mathbf{v}_{b0}}{\mathbf{v}_{20} \cdot \mathbf{v}_{b0}}$$

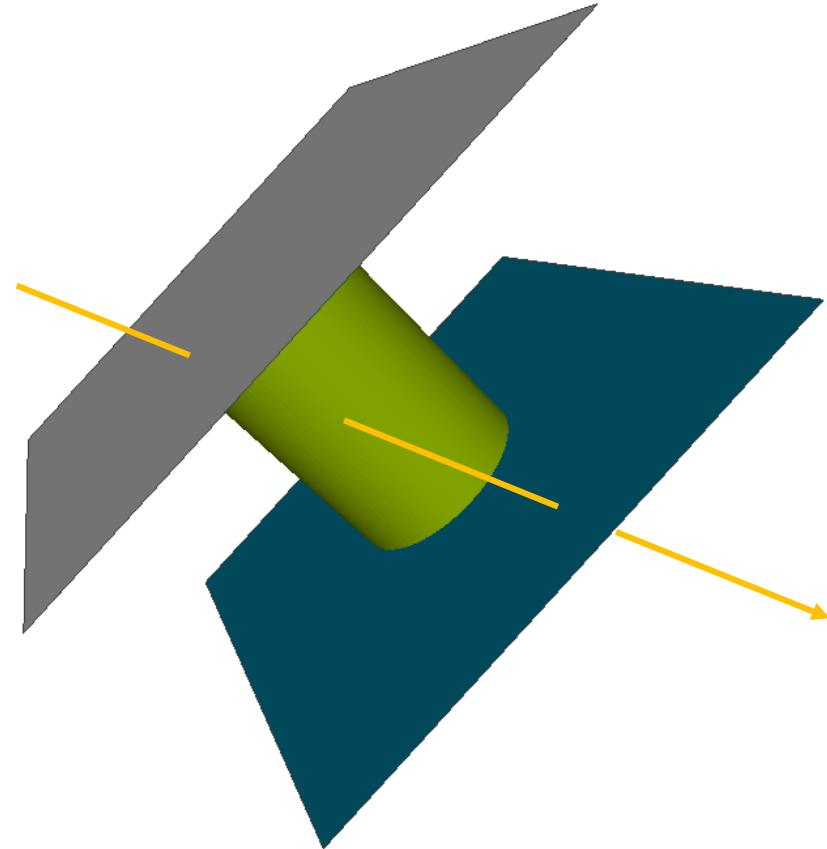
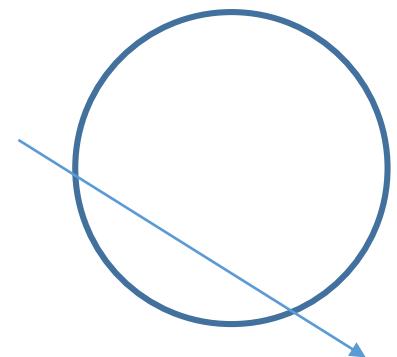
$0 < \alpha_0 < 1 \rightarrow v_p \text{ Inside the triangle}$

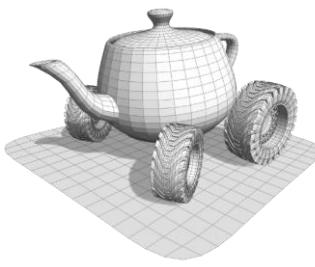
This is false!!!
One assumption is missing



Ray-Cylinder intersection

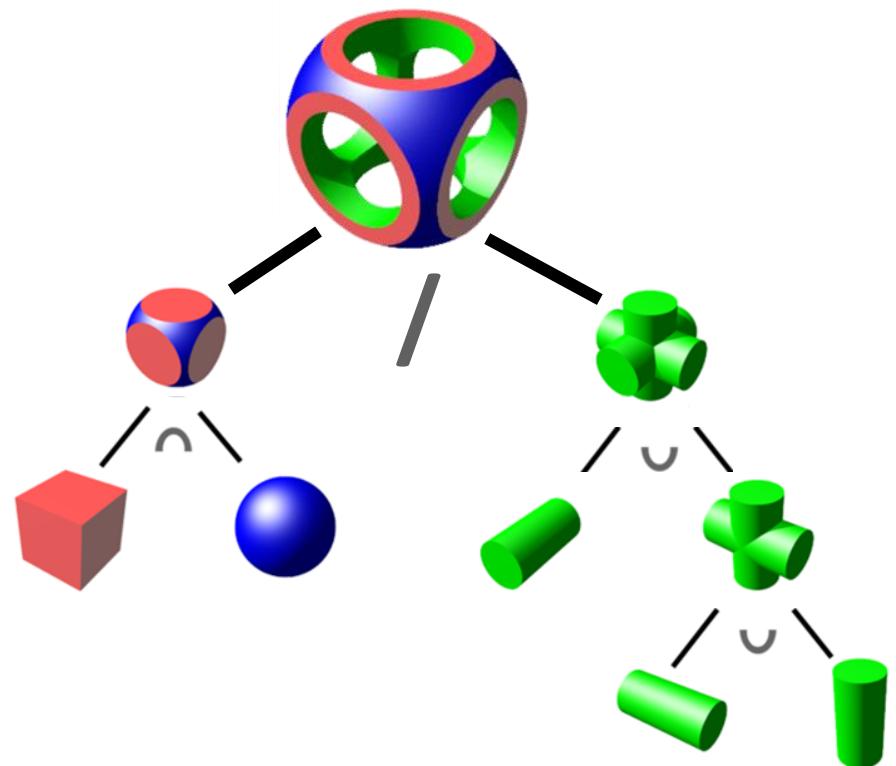
- Put two slabs on the bottom and top side of the cylinder
- Check if the projection of the portion of ray between the slabs intersects the base circle

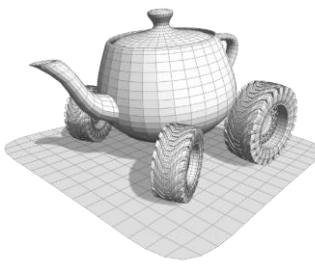




Ray-CSG intersection

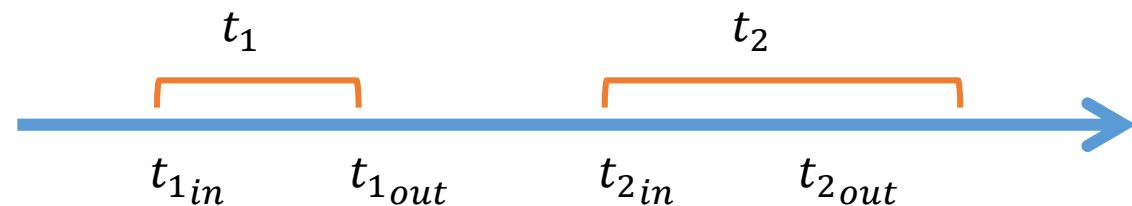
- Recall: a CSG model is formed by combining simple primitives by **union**, **intersection** and **difference**
 - It is conveniently represented by a binary tree where each node is an operation over its two children
- Initial requirement:
 - The ability to find **all** the intersections of a ray with a primitive



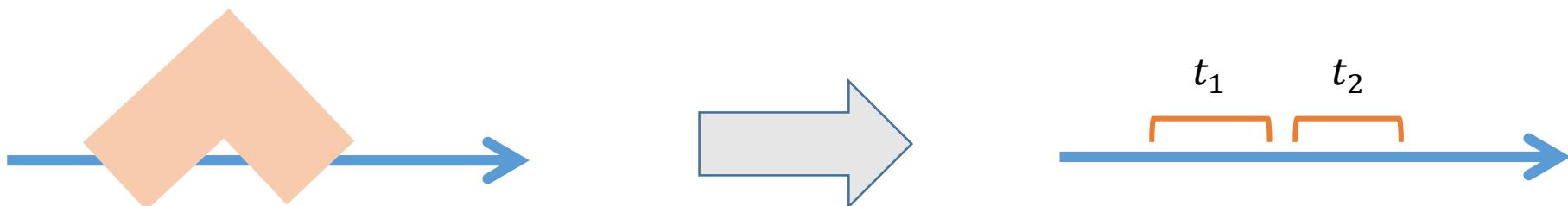


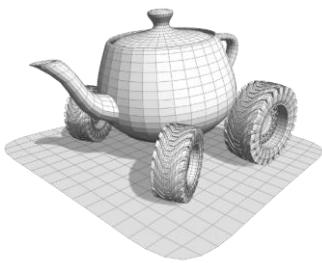
Ray-CSG intersection

- **Span:** an interval t_{in}, t_{out} along a ray

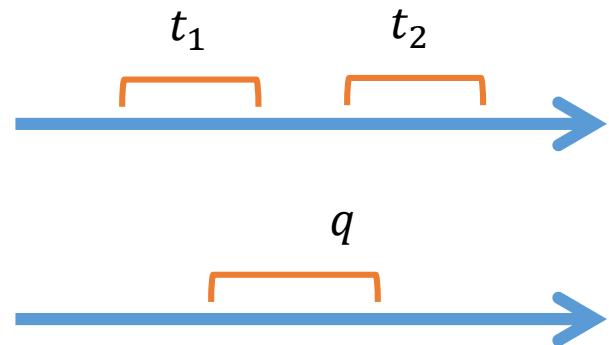


- The intersection of a ray with a watertight object produces a series of spans (from 0 to n)





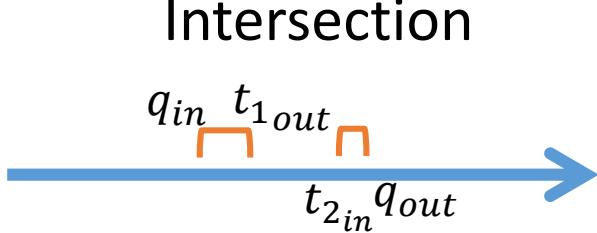
Operations between series of spans



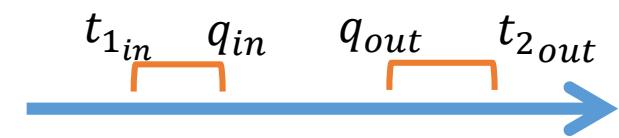
Union

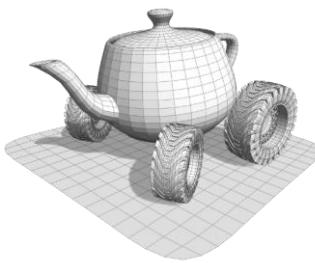


Intersection



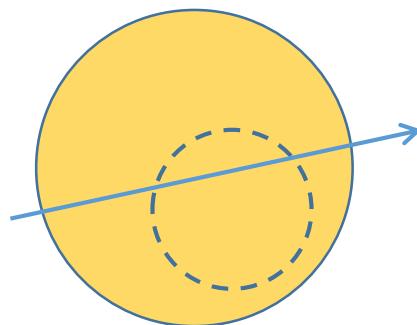
Difference



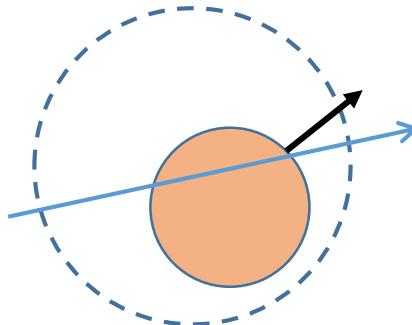


Ray-CSG intersection

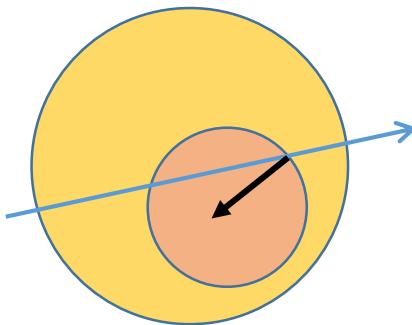
- Algorithm:
 - Find the spans on the leaves
 - Merge them bottom-up along the tree
- What about the normals?
 - Store them along with the intersection points
 - note: for the *difference* operator the normals of q_{out} intersection must be negated.



ray-scene intersection



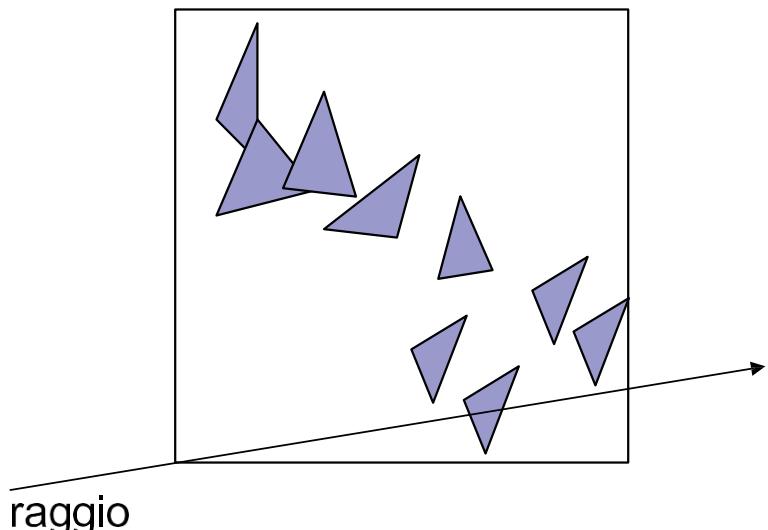
Introduction to Computer Graphics: a Practical Learning Approach.



Intersezione raggio - scena

Come trovo l'intersezione di un raggio con la scena (che supponiamo composta di n triangoli) ?

Per ogni triangolo, calcolo l'intersezione (eventuale) con il raggio. Il risultato è l'intersezione più vicina al punto di partenza del raggio.



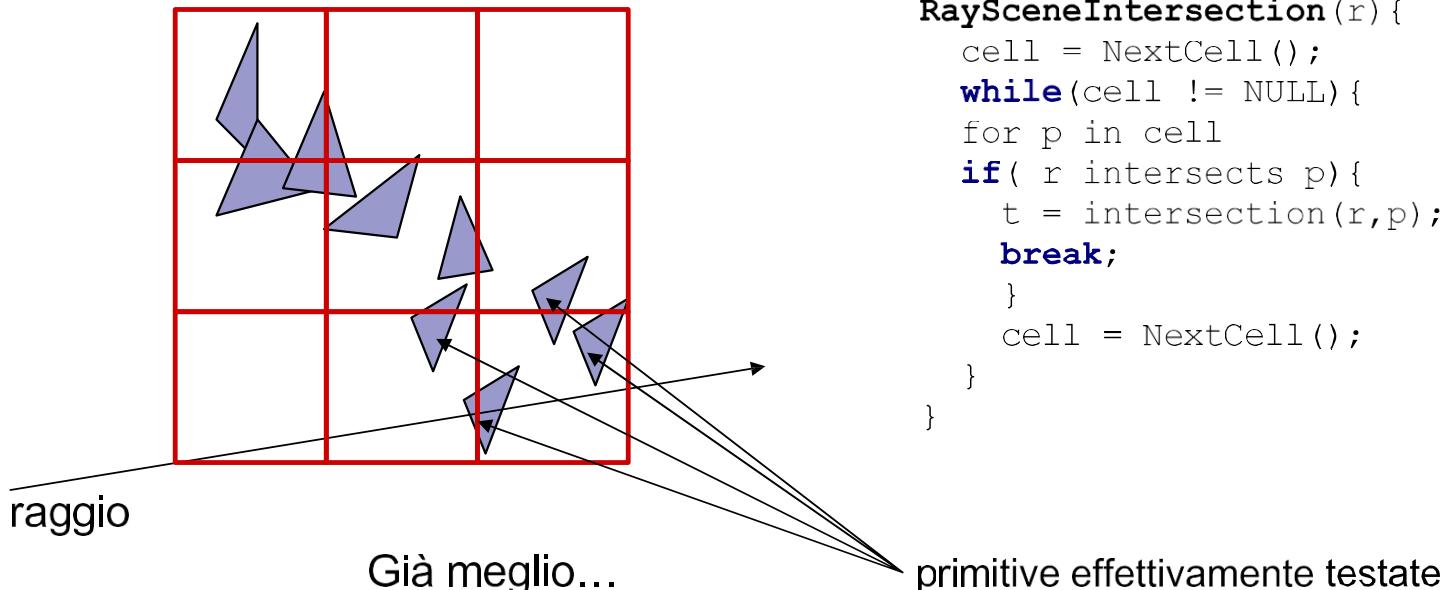
```
RaySceneIntersection(r) {  
    closest = infinite;  
    closest_p = NULL;  
    for p in triangles  
        if( r intersects p){  
            t = intersection(r,p);  
            if(t < closest){  
                closest = t;  
                closest_p = p;  
            }  
        }  
}
```

Peggio di così non si può fare!

Strutture di indicizzazione spaziale

Idea: cerchiamo di ridurre il numero di test di intersezione raggio-triangolo

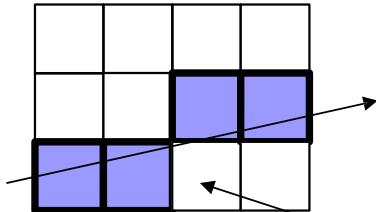
1. Immagino una griglia uniforme nello spazio
2. Ad ogni cella associo l'insieme di primitive che la intersecano
3. **Rasterizzo** il raggio sulla griglia: per ogni cella rasterizzata, eseguo **l'intersezione** tra il raggio e le primitive associate



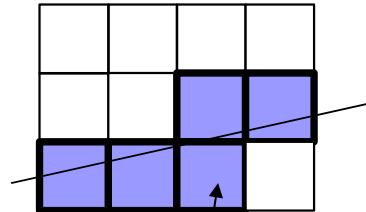
DDA-3d: rasterizzazione in 3D

- È molto simile alla rasterizzazione dei segmenti in 2d
 - Simile, non uguale
 - Anche il problema non è uguale

Per disegnare la linea



Per il ray tracing



Per il ray tracing mi interessano **tutte** le celle intersecate dal raggio

DDA-3d: rasterizzazione in 3D [Amanatides&Woo 87]

- Niente direzioni preferenziali ($m < 1$, $m > 1$ etc..)
- L'algoritmo si muove da un'intersezione con un bordo di una cella all'altra
- L'intersezione può essere con ognuno dei piani principali (XY, XZ, YZ)
- Forma parametrica del raggio

$$r = \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} \cdot t + \begin{bmatrix} cx \\ cy \\ cz \end{bmatrix}$$

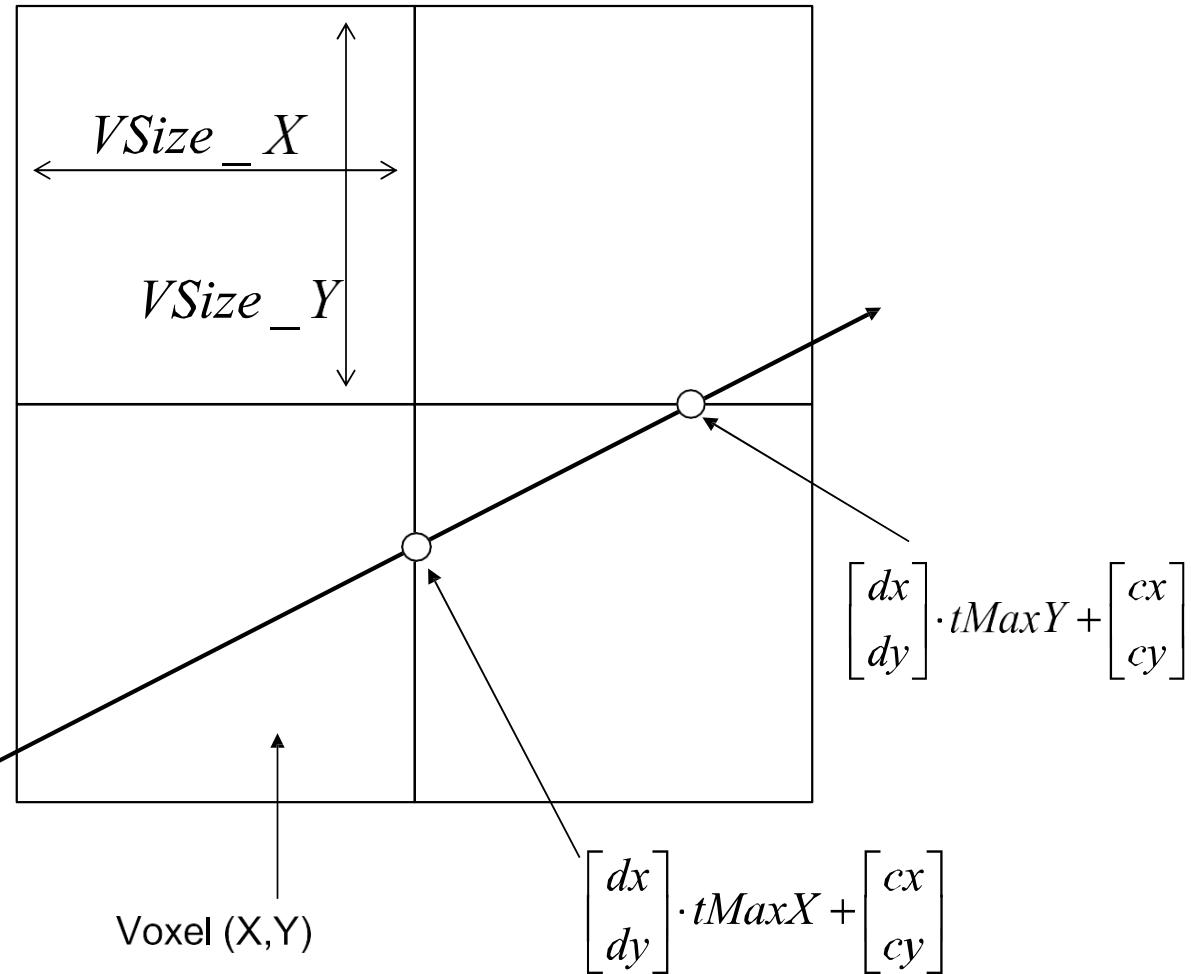
DDA-3d: rasterizzazione in 3D [Amanatides&Woo 87]

■ Caso 2D.

$$tDeltaX = \frac{VSize_X}{dx}$$

$$tDeltaY = \frac{VSize_Y}{dy}$$

$$r = \begin{bmatrix} dx \\ dy \end{bmatrix} \cdot t + \begin{bmatrix} cx \\ cy \end{bmatrix}$$



DDA-3d: rasterizzazione in 3D [Amanatides&Woo 87]

■ Variabili:

- X,Y: variabili intere che identificano il voxel corrente
- tMaxX (tMaxY): valore massimo di t per il quale la coordinata X (Y) del voxel che contiene il punto rimane immutata
- tDeltaX (tDeltaY): valore di cui incrementare t per spostare un punto lungo il raggio della dimensione del voxel in direzione X (Y)

■ Inizializzazione di X,Y:

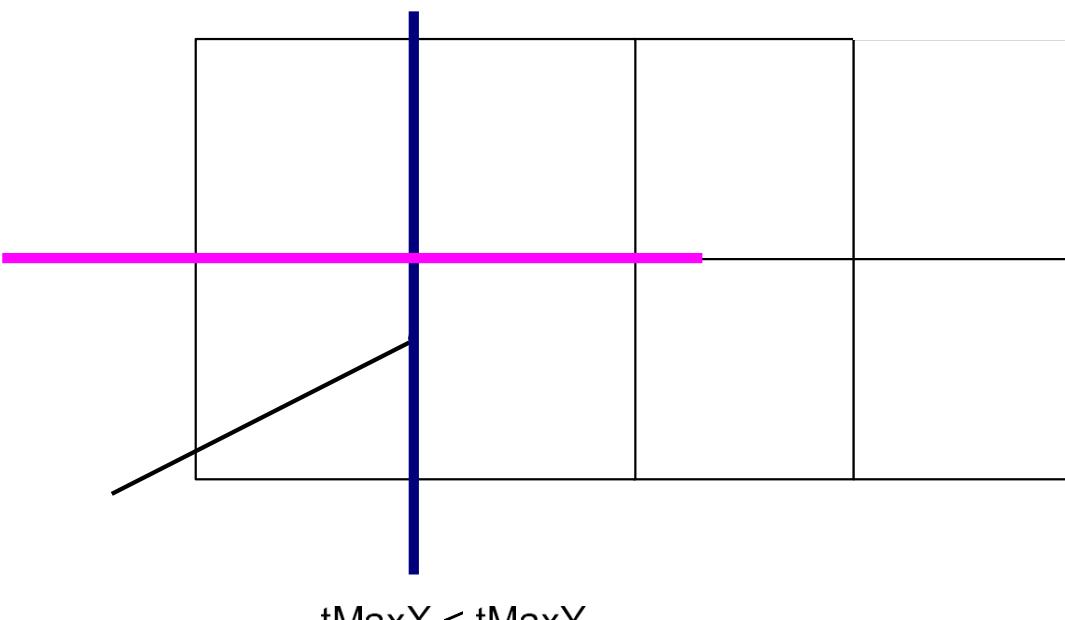
- Voxel di partenza del raggio (se il raggio parte all'interno della regione “voxelizzata”)
- Primo voxel colpito da raggio

DDA-3D: rasterizzazione in 3D [Amanatides&Woo 87]

```
DDA3D(r) {  
    //inizializza X,Y  
    do {  
        if(tMaxX < tMaxY) {  
            tMaxX = tMaxX + tDeltaX;  
            X = X + stepX;//stepX ==1 o -1. è il segno di dx  
        } else  
        {  
            tMaxY = tMaxY + tDeltaY;  
            Y = Y + stepY;  
        }  
    }while( (X,Y) fuori dalla regione ));  
}
```

DDA-3D: esempio

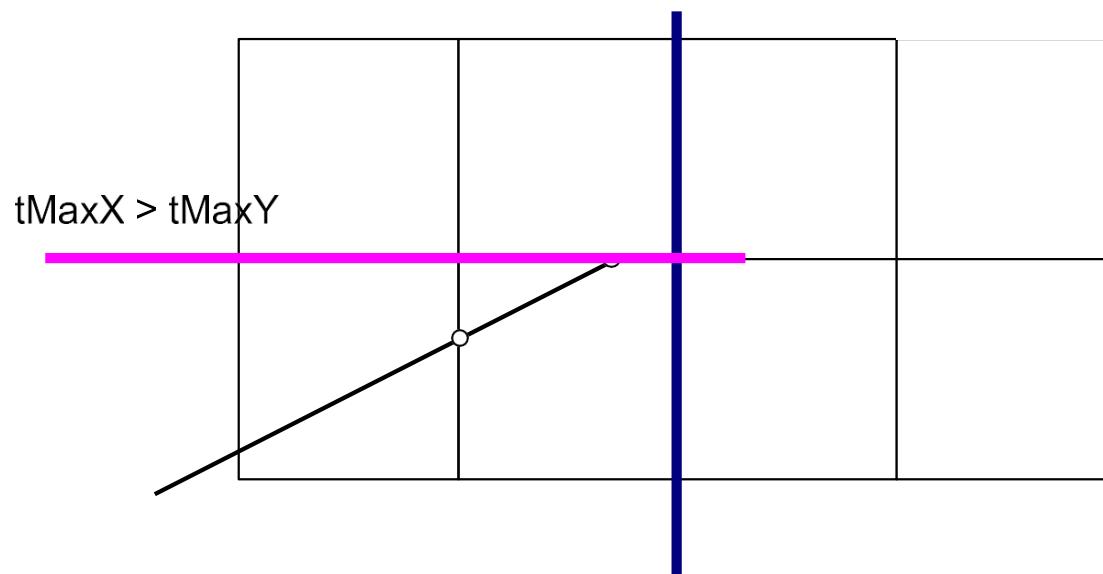
X = 0
Y = 0



DDA-3D: esempio

X = 1

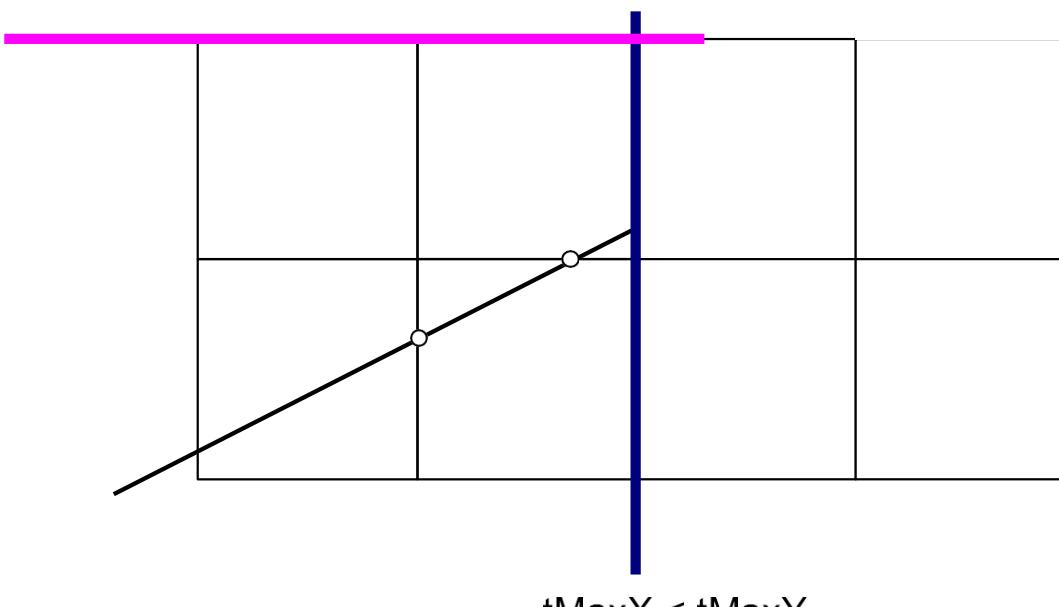
Y = 0



DDA-3D: esempio

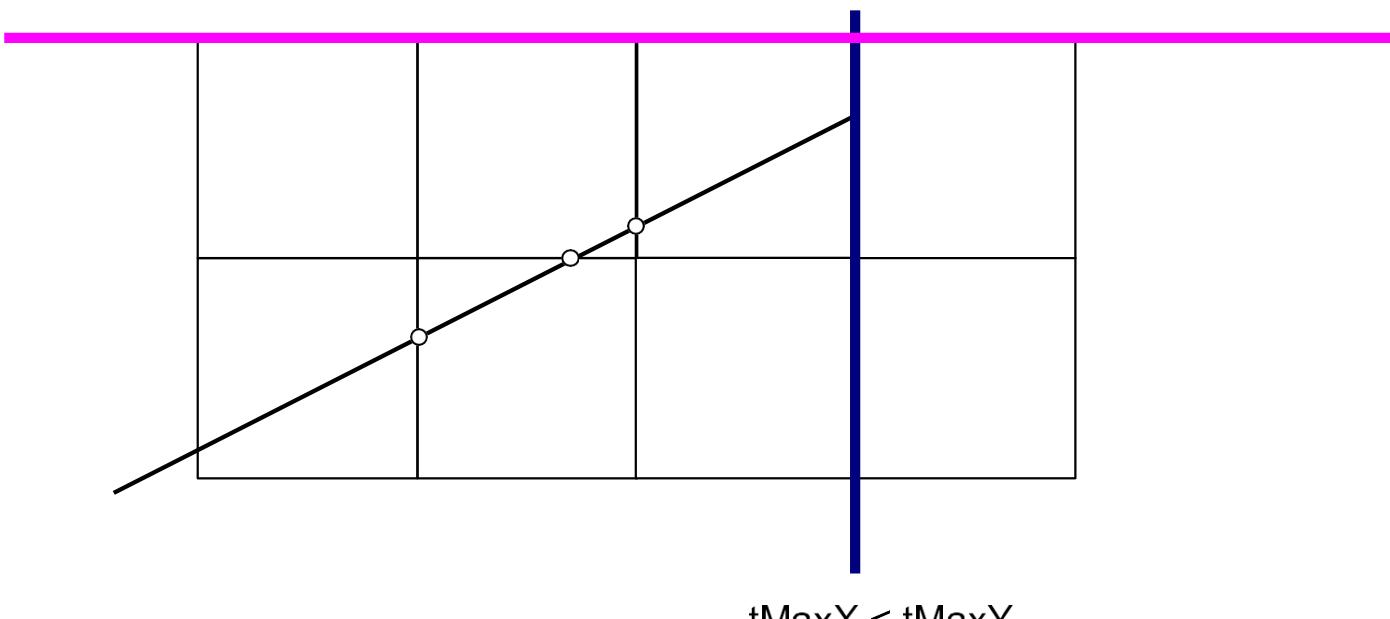
X = 1

Y = 1



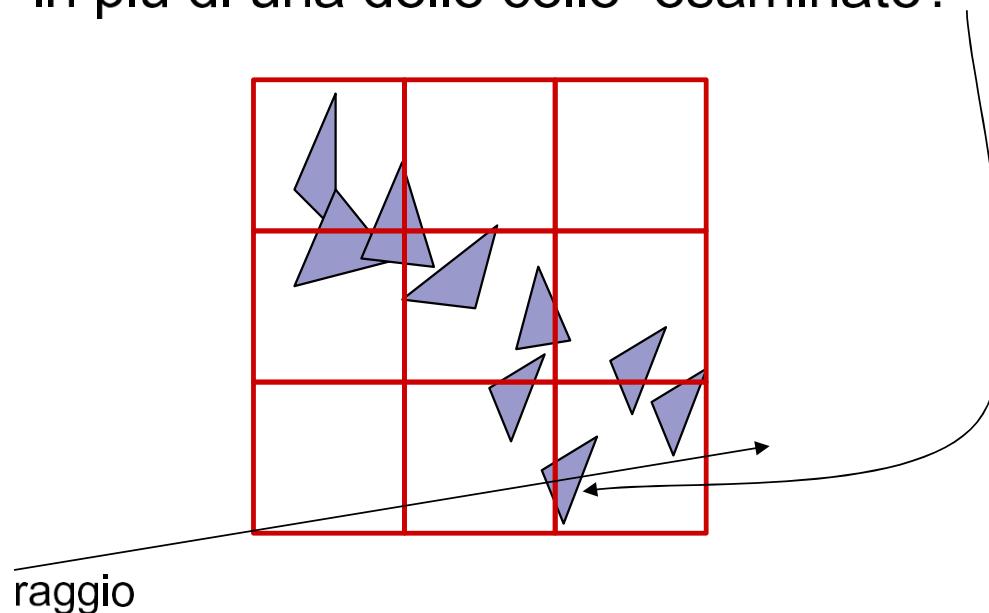
DDA-3D: esempio

X = 2
Y = 1



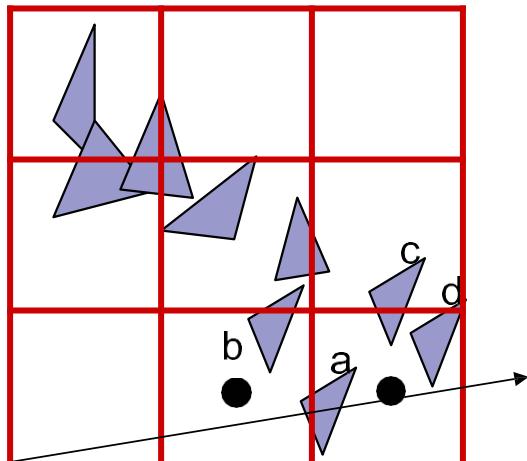
Quanti test di intersezione?

- Usando la griglia regolare abbiamo utilizzato la seguente osservazione
 - Se il raggio non interseca una cella, di sicuro non interseca niente che sia interamente contenuto nella cella
- Quante volte viene testata una primitiva che è presente in più di una delle celle esaminate?



Mailboxing

- Ogni raggio ha un suo ID (es: un intero)
- Ad ogni primitiva si associa l'ID dell'ultimo raggio per cui è stato eseguito il test di intersezione



ray-scene intersection

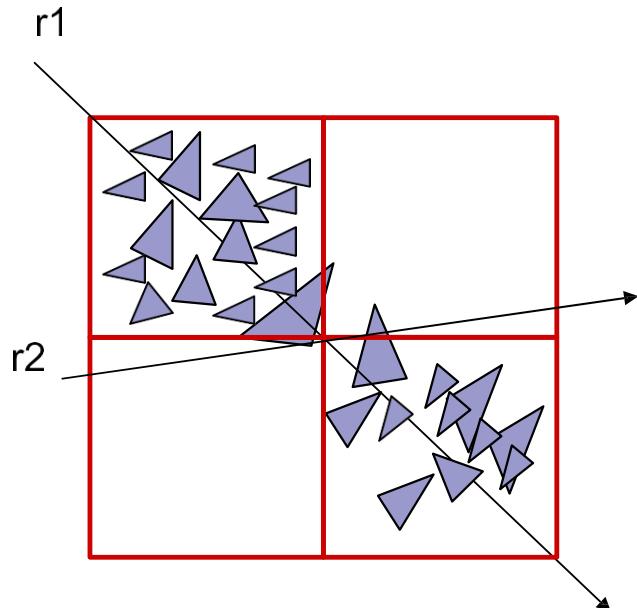
Raggio ID=56

Introduction to Computer Graphics: a
Practical Learning Approach.

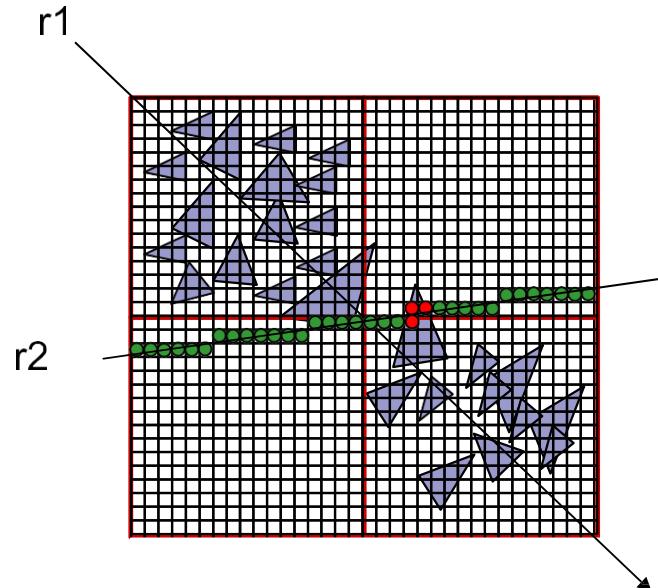
Intersection(ray,b)
b.ID = 56;
Intersection(ray,a)
a.ID = 56;

Skip Intersection(ray,a)
Intersection(ray,c)
c.ID=56;
Intersection(ray,d)
d.ID=56;

Quanto devono essere grandi i voxel?



Se sono troppo grandi
devo comunque fare molti
test di intersezione con le
primitive

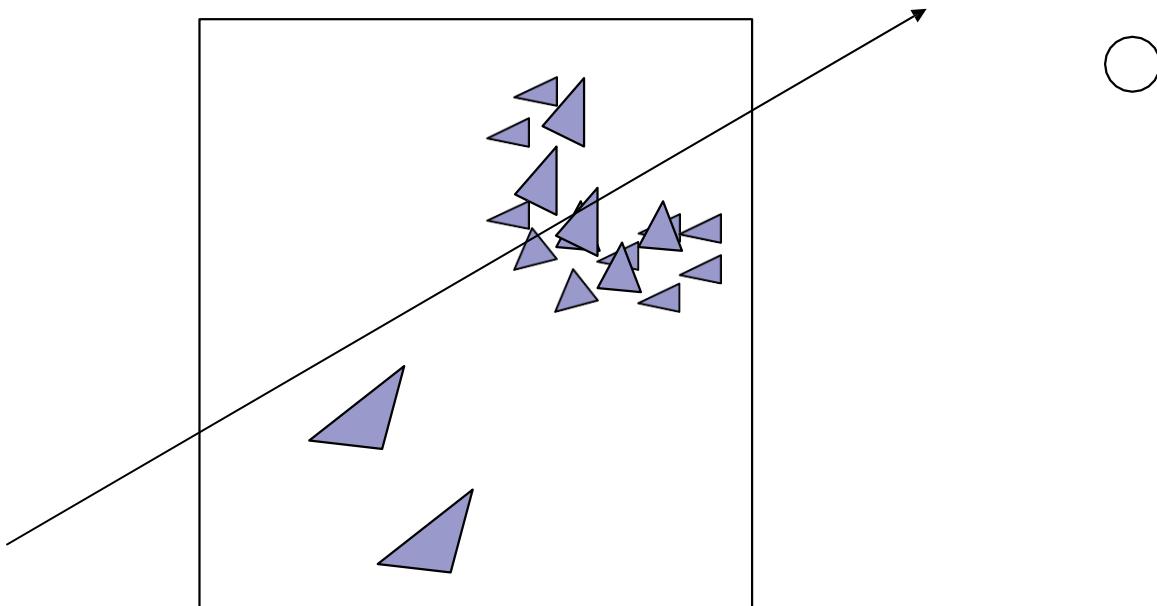


Se sono troppo piccoli la
rasterizzazione rallenta

Nelle zone in cui le primitive sono densamente distribuite, converrebbe avere voxel più piccoli
Nelle zone in cui le primitive sono poche, converrebbe avere voxel più grandi

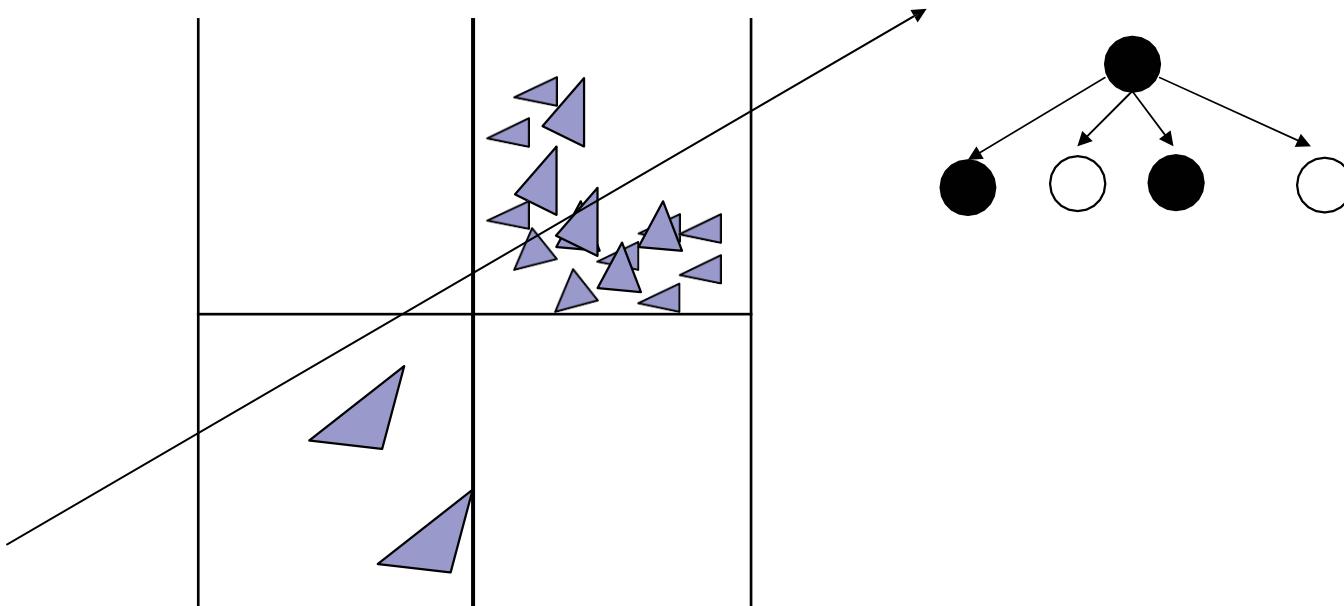
Strutture di indicizzazione spaziale gerarchiche

- Il principio usato per le griglie regolari:
 - Se un raggio non interseca un voxel, di sicuro non interseca niente ivi contenuto
- Applichiamo questo principio ricorsivamente



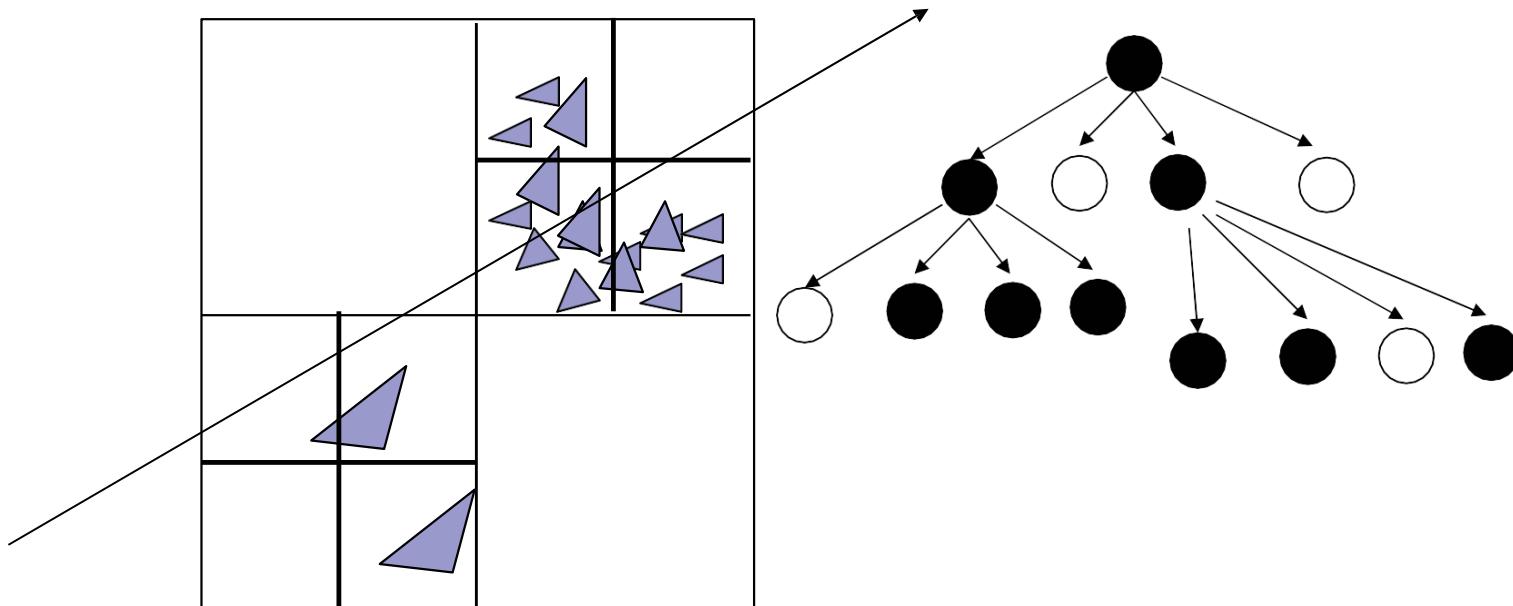
Strutture di indicizzazione spaziale gerarchiche

- Il principio usato per le griglie regolari:
 - Se un raggio non interseca un voxel, di sicuro non interseca niente ivi contenuto
- Applichiamo questo principio ricorsivamente



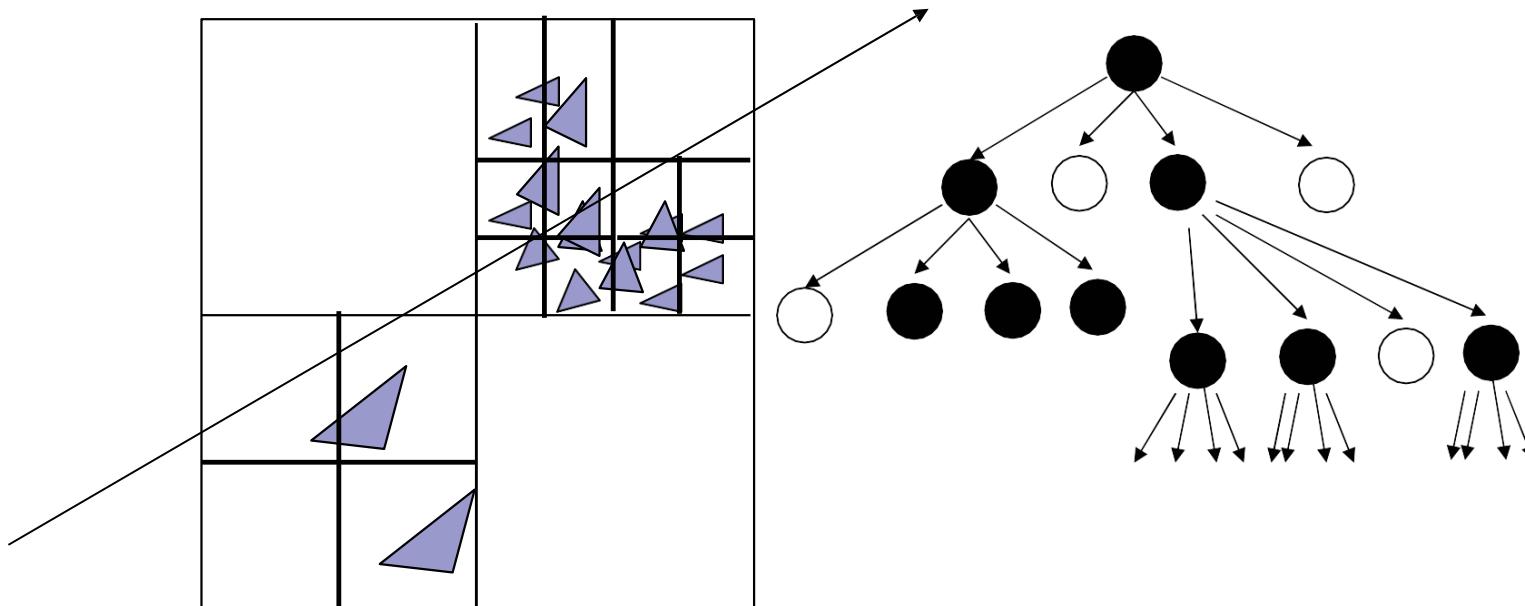
Strutture di indicizzazione spaziale gerarchiche

- Il principio usato per le griglie regolari:
 - Se un raggio non interseca un voxel, di sicuro non interseca niente ivi contenuto
- Applichiamo questo principio ricorsivamente



Strutture di indicizzazione spaziale gerarchiche

- Il principio usato per le griglie regolari:
 - Se un raggio non interseca un voxel, di sicuro non interseca niente ivi contenuto
- Applichiamo questo principio ricorsivamente



Algoritmo

```
TestTree(Ray ray, Node node, float &t, Primitive p) {
    toVisit = {node}
    t = infinite; intersected = NULL;
    while( (toVisit != {}) && (t == infinite) ){
        n = pop(toVisit);
        if(Intersect(ray,n) )
            if(IsLeaf(n)){
                for each primitive p in node
                    t' = Intersect(ray,p);
                    if(t' < t)
                        {
                            t = t';
                            intersected = p;
                        }
            } else
            {
                for n' children of n
                    toVisit = toVisit U n'
            }
        } // end while
    } // end function
```

Rasterizzazione

Intersezione raggio primitiva

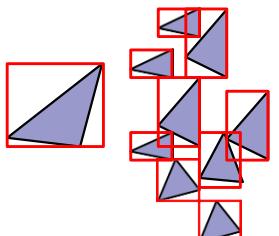
bool IsLeaf(n){
 return n.contained_primitives < min_prim
}

Scelta possibile (non unica): se il numero di primitive contenute nel nodo è minore di un valore prefissato *min_prim* allora il nodo è una foglia, cioè la cella corrispondente non viene ulteriormente suddivisa

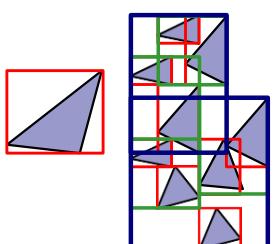
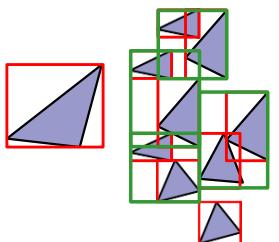
Introduction to Computer Graphics: a Practical Learning Approach.

Strutture di indicizzazione spaziale gerarchiche

- Due tipi di suddivisione:
- Suddivisione delle primitive

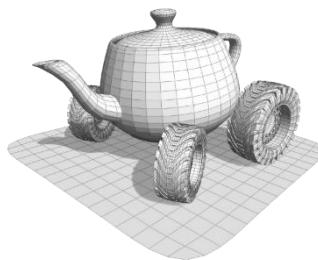


- In genere costruita Bottom-Up
- Ogni primitiva è interamente dentro il nodo
- I BV si intersecano (male, aumenta il numero di intesezioni raggio BV)
- I BV sono disposti in maniera irregolare (male, non posso rasterizzare)



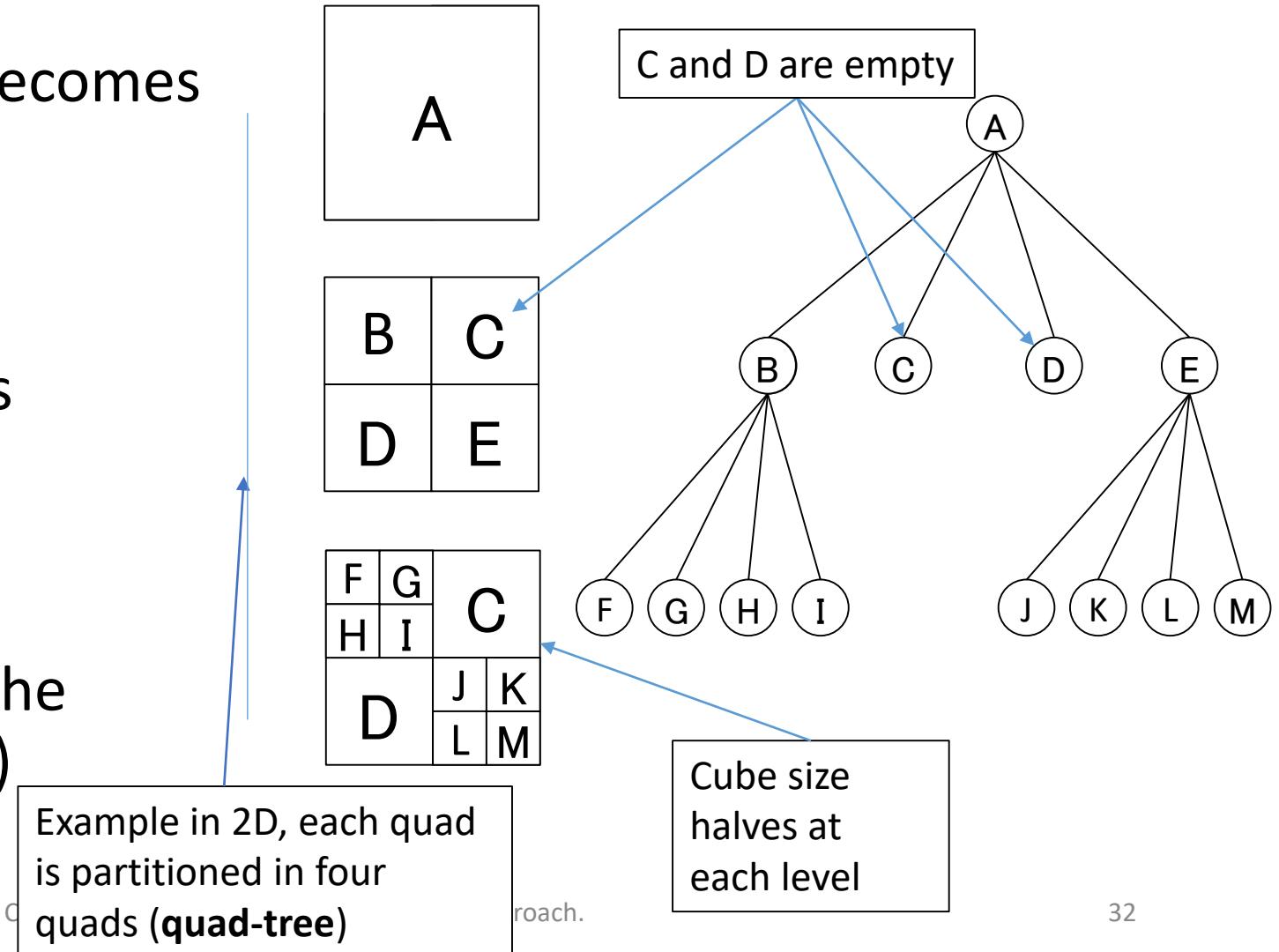
Strutture di indicizzazione spaziale gerarchiche

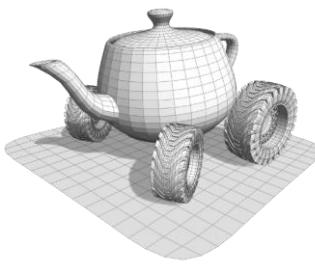
- Due tipi di suddivisione:
- Suddivisione dello spazio (come nell'esempio):
 - In genere costruita Top-Down
 - Ogni primitiva può essere interamente dentro il nodo o no (nell'esempio no)
 - Le celle non si intersecano
 - Le celle sono disposte in maniera regolare o irregolare (nell'esempio in maniera regolare)
- Limitiamoci a quella più adatta per il Ray-Tracing



Oct-Tree, Quad-tree

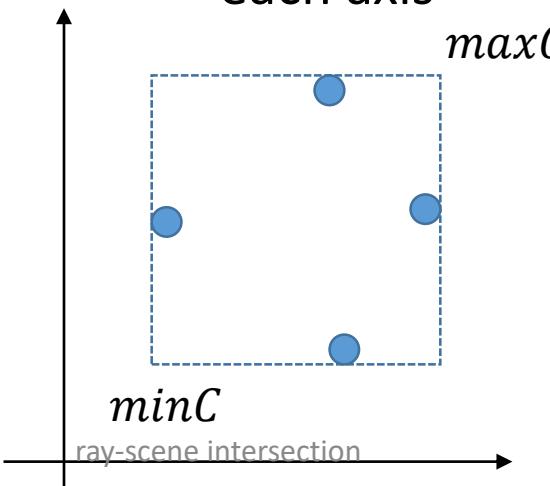
- Flat-brute storage of data becomes easily unmanageable
- Hierarchical representation the volume is recursively partitioned in 8 subvolumes
 - Recursion stops on empty subvolumes
 - Predefined maximal depth
- The cost is quadratic with the resolution (instead of cubic)





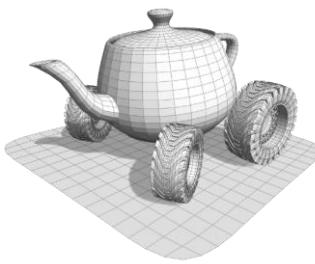
Axis-Aligned Bounding Boxes (AABB)

- **AABB:** a box that *bounds* a set of primitives/objects and which sides are parallel to the canonical planes (XY, XY, ZY)
 - Already seen in the shadow mapping slides, in order to compute the projection part of the light matrix
 - Identified by the *minC* and *maxC* corners
- Computing the AABB of a set of points:
 - Iterate over all the points, find the minimum and maximum value of the coordinates along each axis

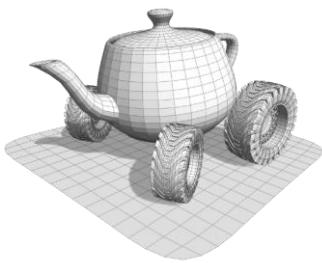


```
// find bounding box of points[]
vec3 minC=vec3(max_float,max_float,max_float);
vec3 maxC=vec3(-max_float,-max_float,-max_float);
for(int i=0; i < points.size(); ++i)
    for(int j = 0; j < 3; ++j){
        if(point[i][j] < minC[j]) minC[j] = points[i][j]; else
            if(point[i][j] > maxC[j]) maxC[j] = points[i][j];
```

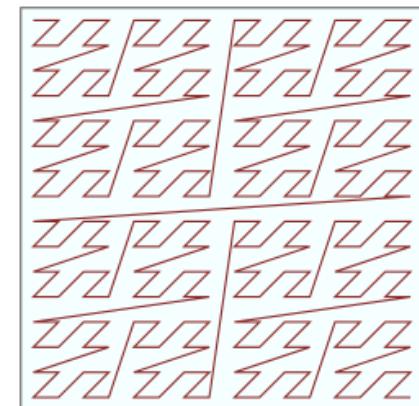
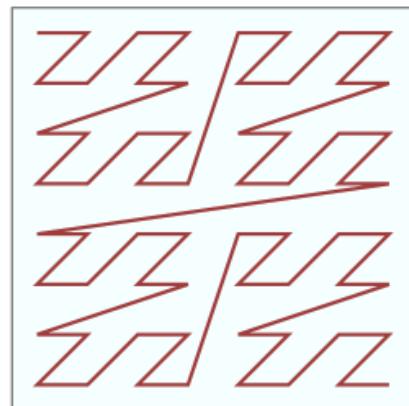
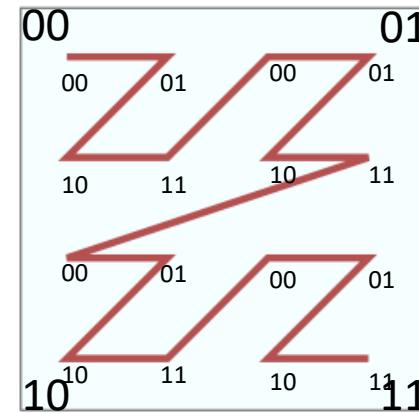
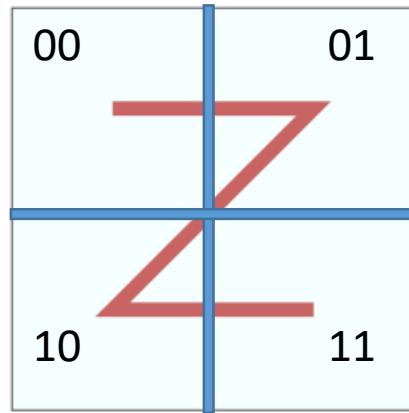
Oct-tree (Quad-tree) data-structure



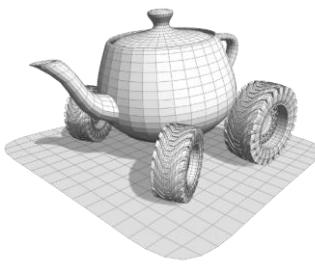
- Pointers: each node of the oct-tree (quad-tree) stores 8 (4) pointers to its children
 - Neighbor nodes may be well apart in memory (poor memory coherence)
 - Reaching a leaf from the root requires several reference pointers (= random accesses)
- **Linear *-tree:** encode non-empty nodes with a integer value so that:
 - Neighbors values refer to spatially neighbors nodes



Space Filling Curve: Z-Filling Curve

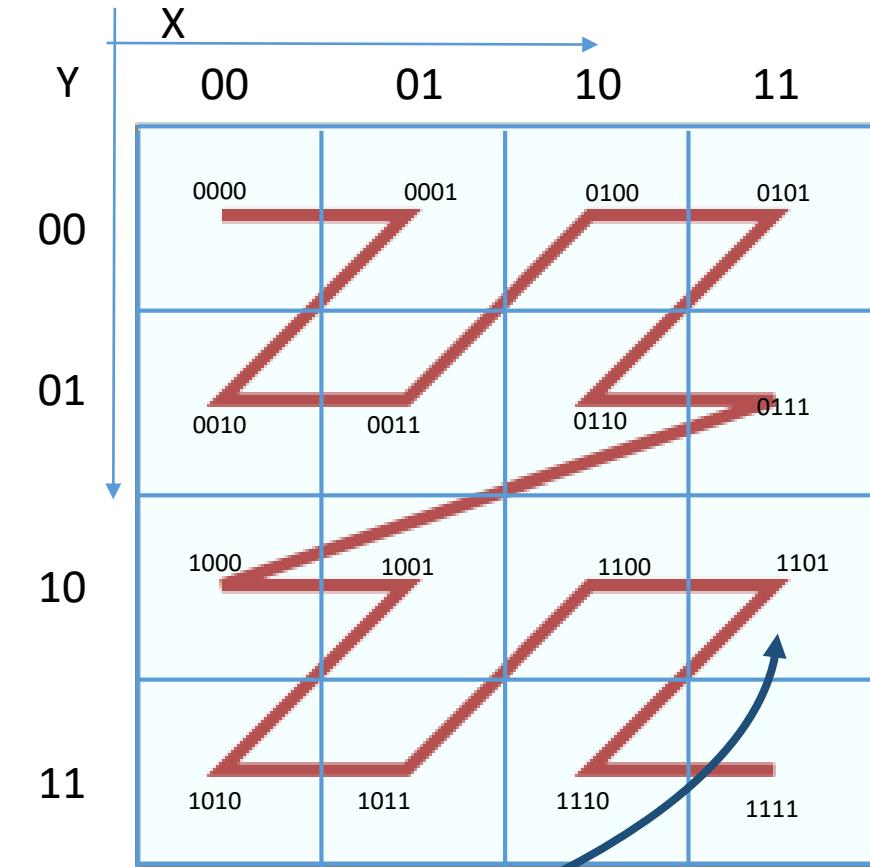
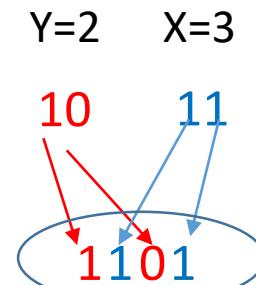


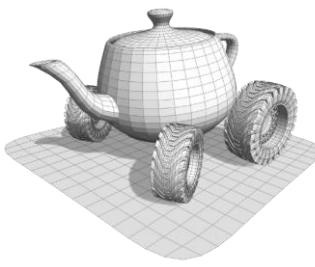
wikipedia



Z-Filling Curve (on the plane)

- For a given subdivision, the position of a cell (Z-order) from its xy coordinates and viceversa is obtained in O(1) with shift and shuffle bit operations

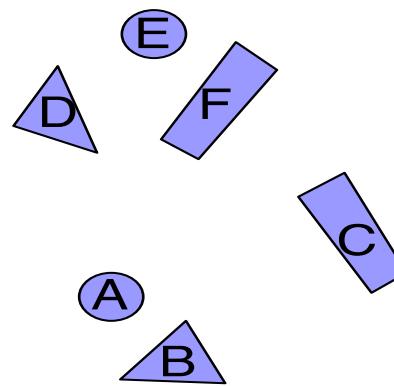


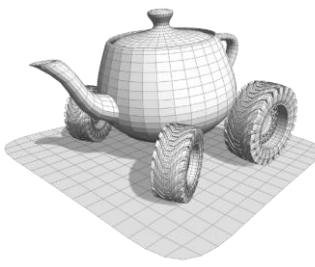


Binary Space Partition-Tree (BSP)

- **Description:**

- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**

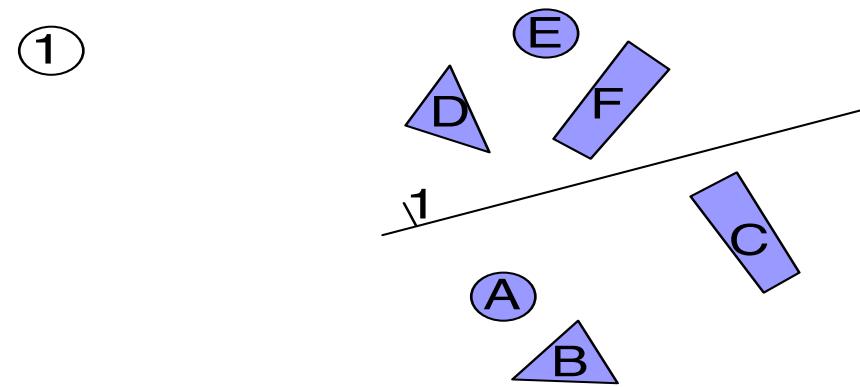


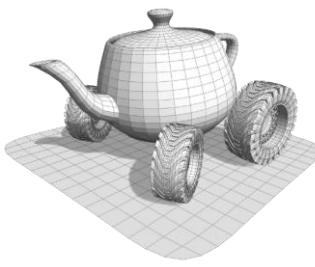


Binary Space Partition-Tree (BSP)

- **Description:**

- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**

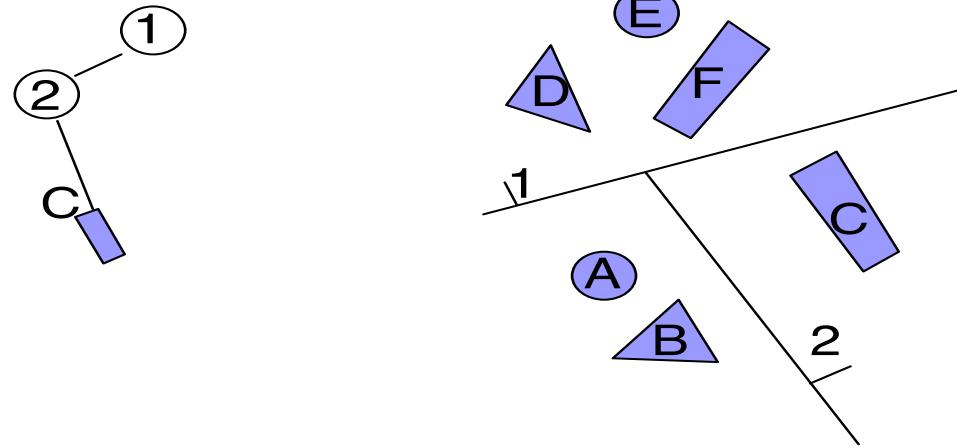


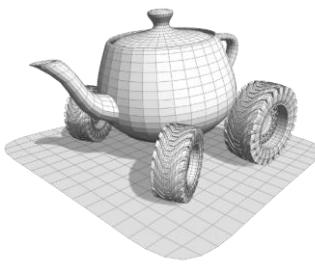


Binary Space Partition-Tree (BSP)

- **Description:**

- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**

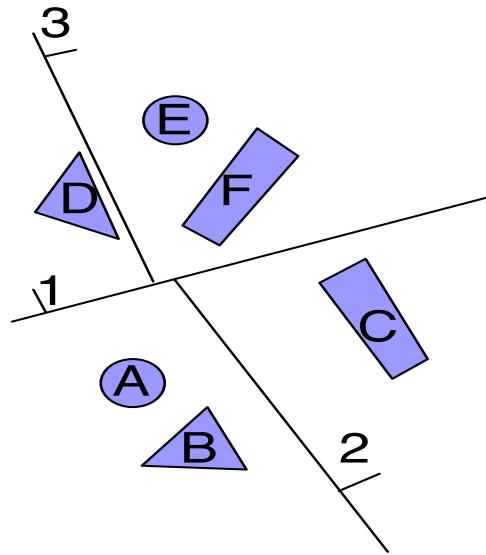
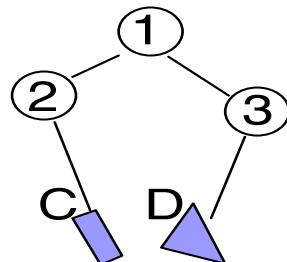


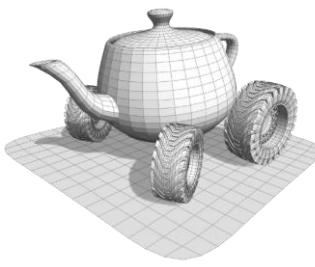


Binary Space Partition-Tree (BSP)

- **Description:**

- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**

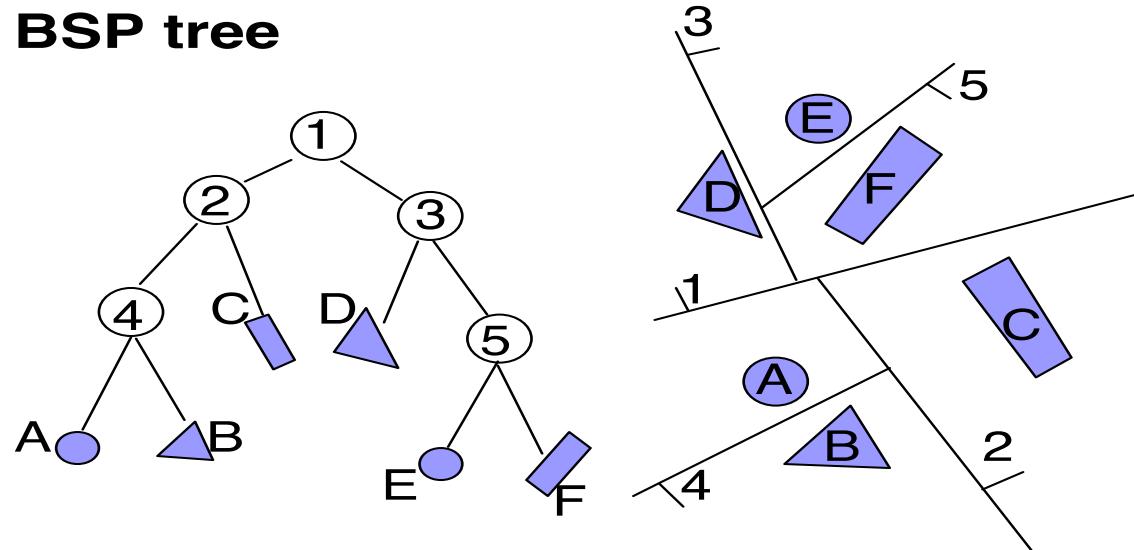


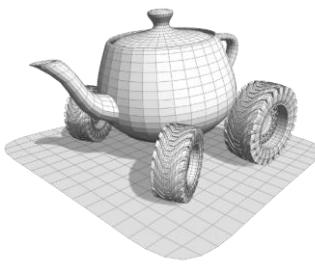


Binary Space Partition-Tree (BSP)

- **Description:**

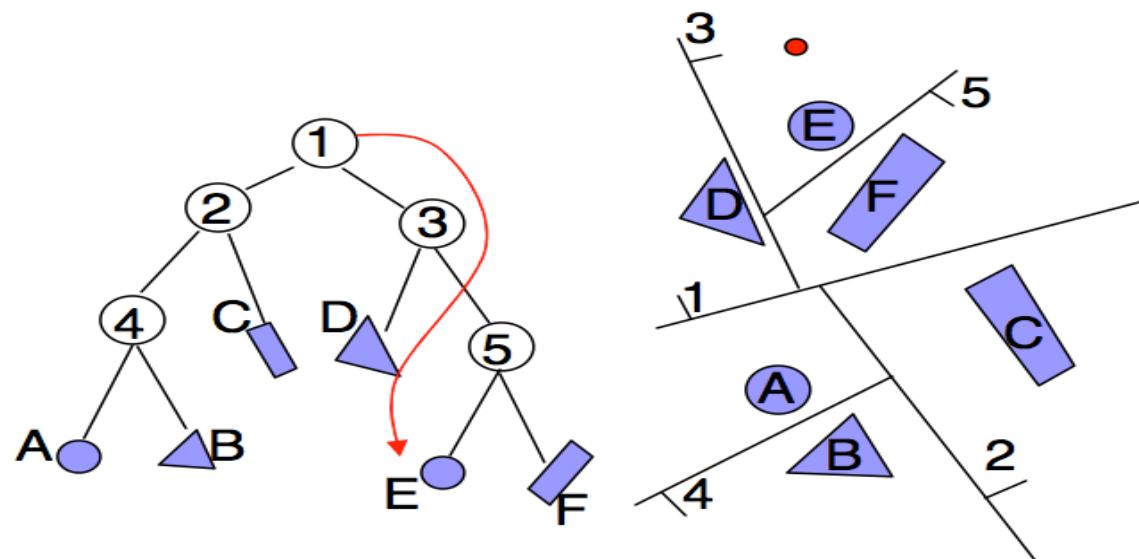
- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**

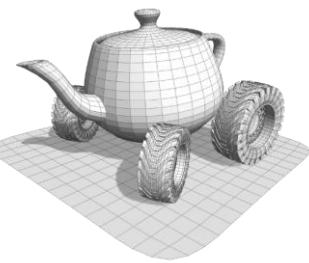




Binary Space Partition-Tree (BSP)

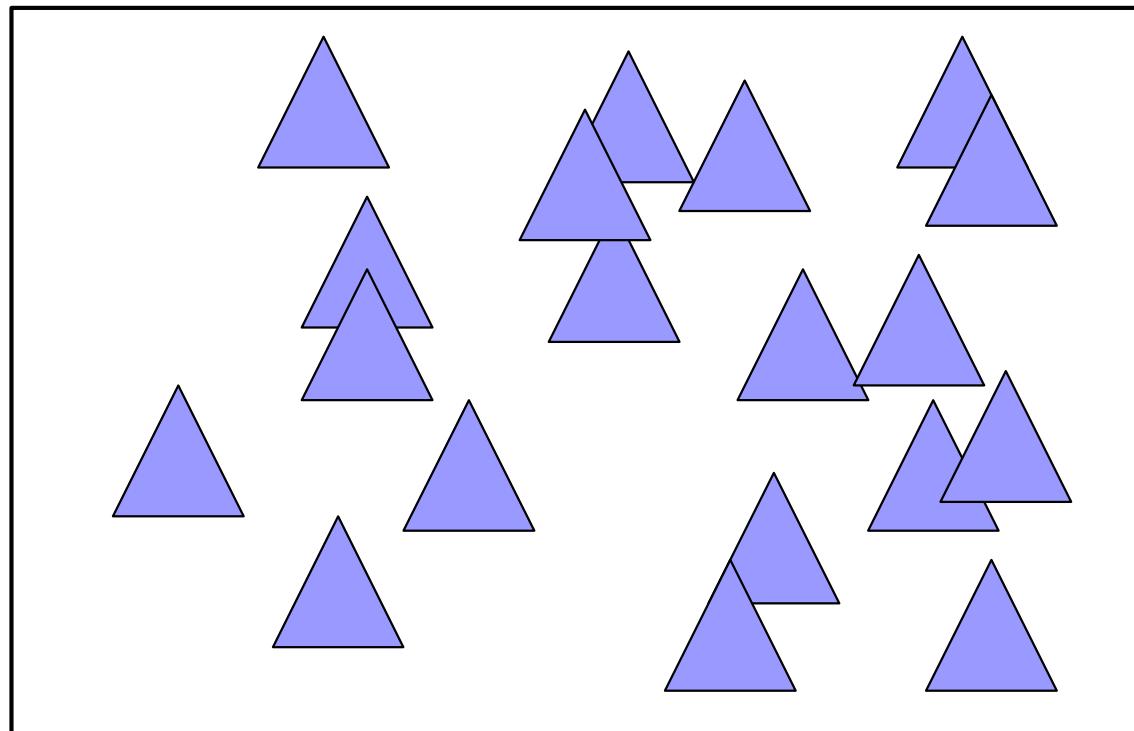
- **Query:** is the point p inside a primitive?
 - Starting from the root, move to the child associated with the half space containing the point
 - When in a leaf node, check all the primitives
- **Cost:**
 - Worst: $O(n)$
 - Aver: $O(\log n)$



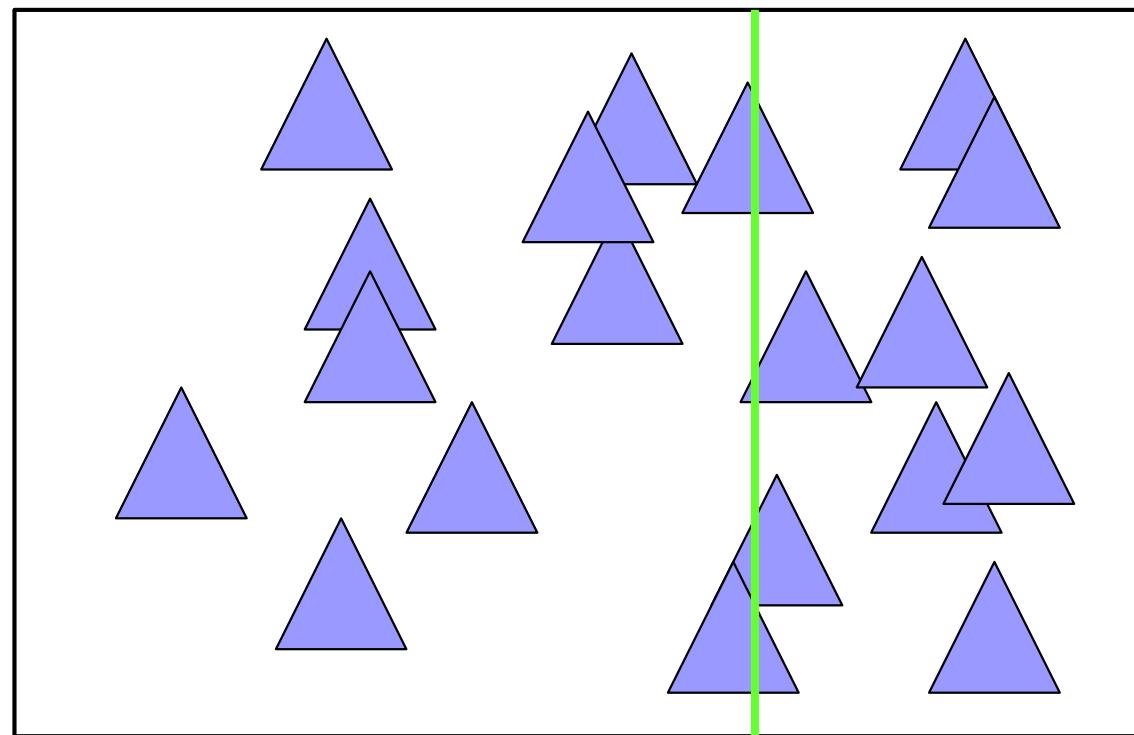
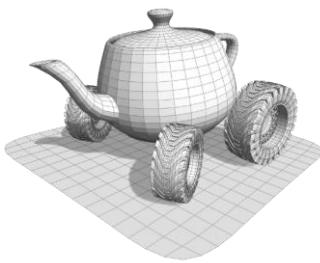


kD-Trees

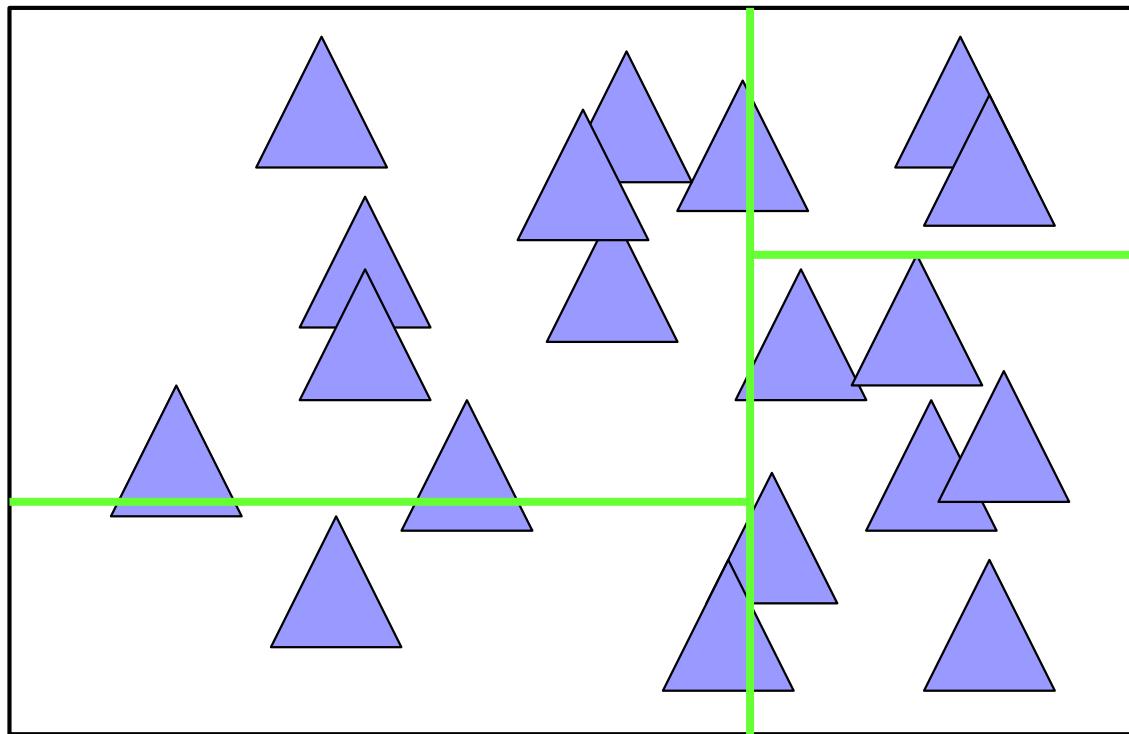
- Basically, an axis-aligned BSP



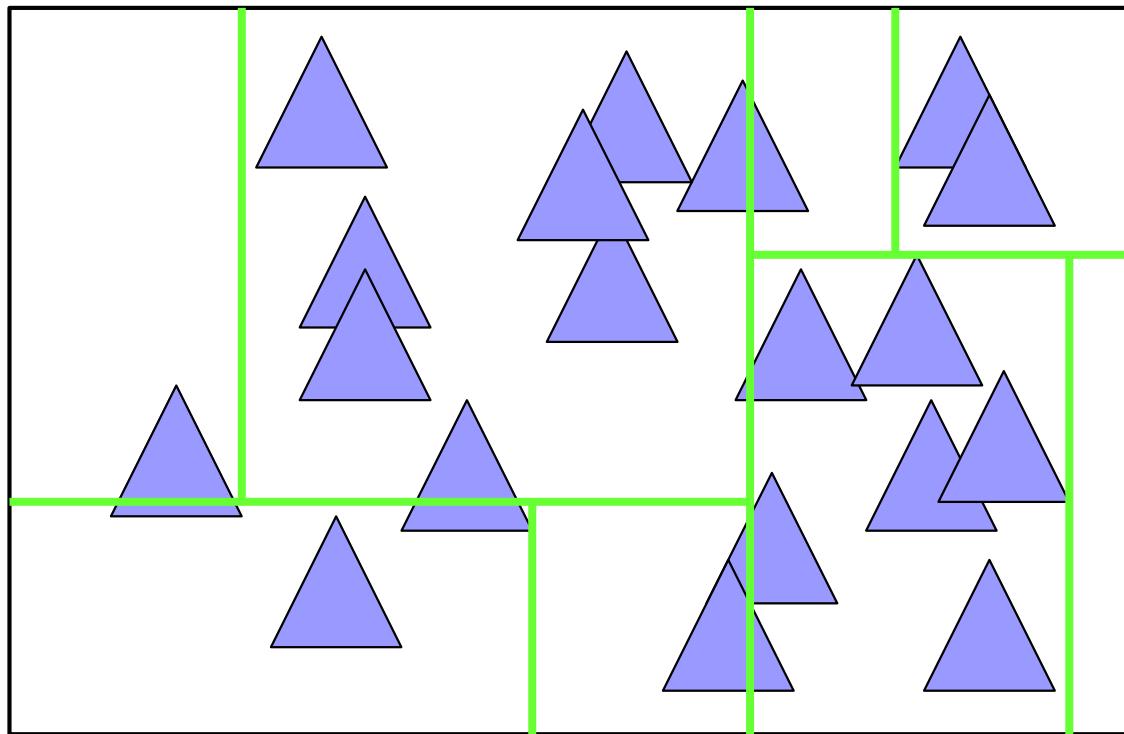
kD-Trees



kD-Trees



kD-Trees



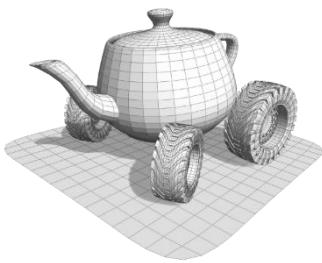
Costruire un kD-tree

- Dati:

- axis-aligned bounding box (“cell”)
 - lista di primitive geometriche (triangoli)

- Operazioni base

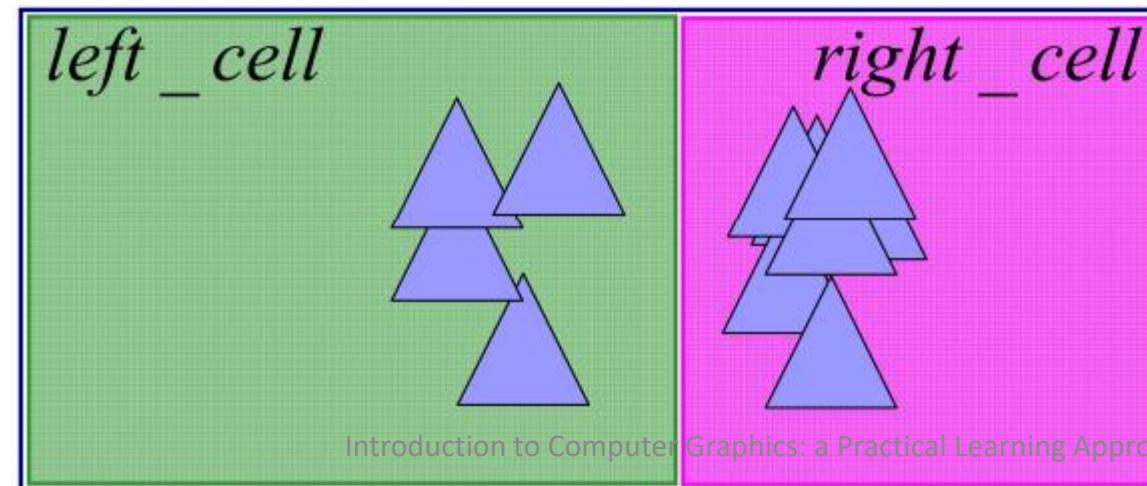
- Prendi un piano ortogonale a un asse e dividi la cella in due parti (**in che punto?**)
 - Distribuire le primitive nei due insiemi risultanti
 - Ricorsione
 - Criterio di terminazione (**che criterio?**)

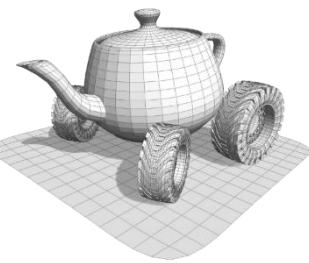


Building an efficient kD-tree

- Where to split the cell?
 - Where the cost is minimized
- What is the cost?

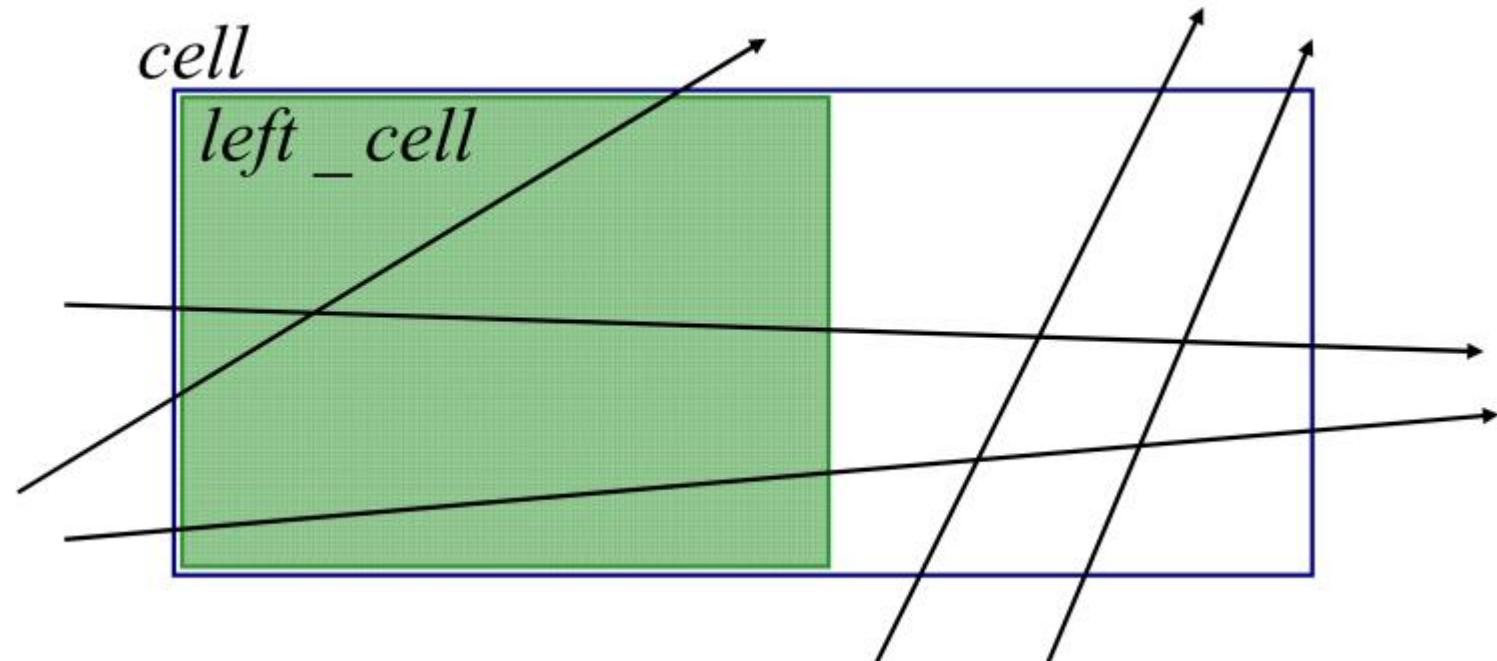
$$Cost(cell) = cost_traversal +$$
$$cell \quad \frac{Prob(left_cell|cell) Cost(left_cell)}{Prob(right_cell|cell) Cost(right_cell)}$$

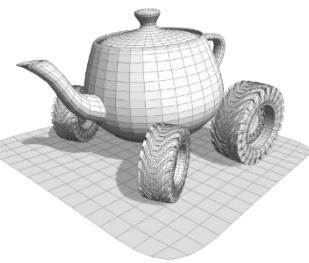




Building an efficient kD-tree

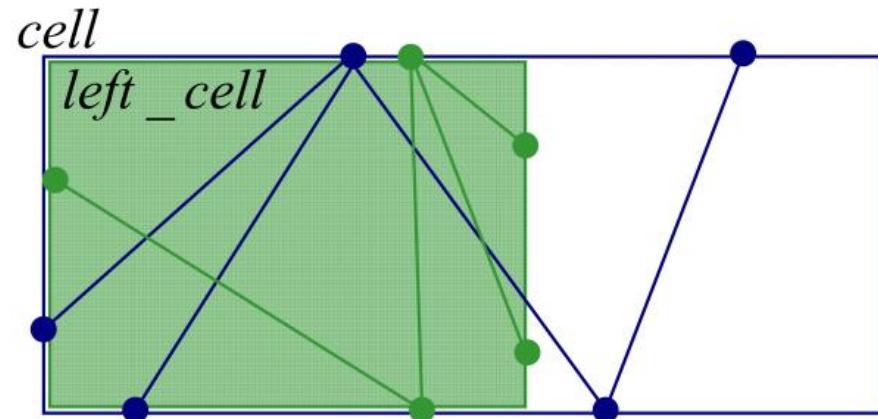
- Knowing that a ray intersect *Cell*, what is the probability that it will also intersect *left_cell* ?



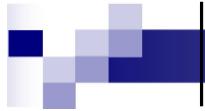


Building an efficient kD-tree

$$Prob(left_cell|cell) = \frac{\#rays\ intersecting \left.\text{left_cell}\right.}{\#rays\ intersecting \text{cell}}$$

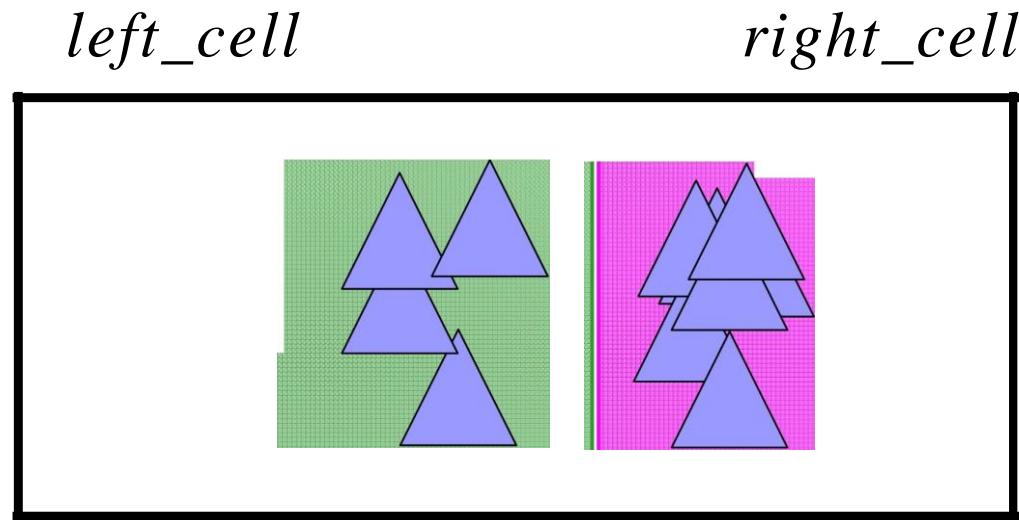


$$Prob(left_cell|cell) = \frac{\int_{\sigma(left_cell)} \int_{\sigma(left_cell)} da\ da}{\int_{\sigma(cell)} \int_{\sigma(cell)} da\ da} = \frac{Area(left_cell)^2}{Area(cell)^2} = \frac{Area(left_cell)}{Area(cell)}$$



$cost(left_cell)$

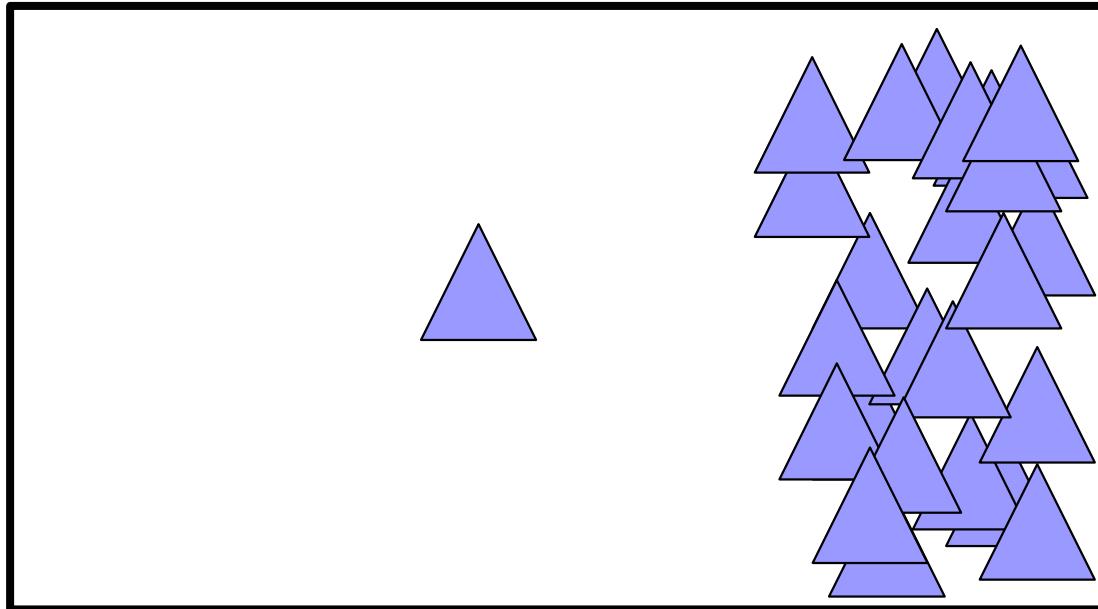
- Sapendo che ii raggio interseca la cella $left_cell$, qual'e ii costo di testare l'intersezione con i triangoli?
- Si approssima con ii numero di triangoli che toccano la cella $cell$



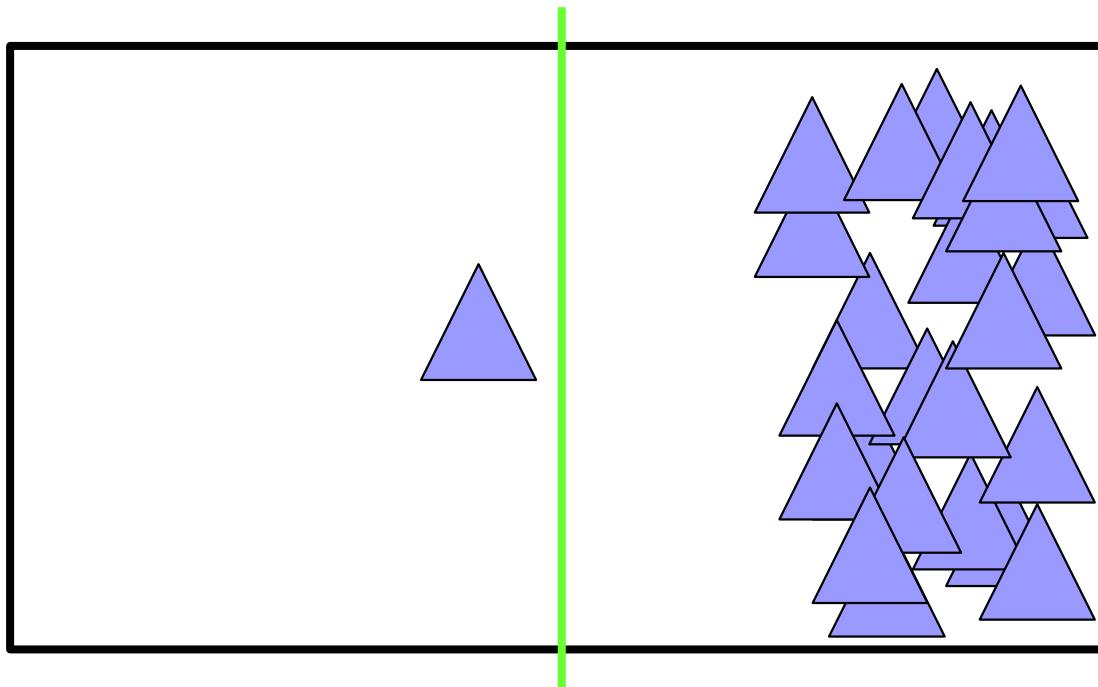
$Cost(left_cell) = 4$

Esempio

- Come si suddivide la cella qui sotto?

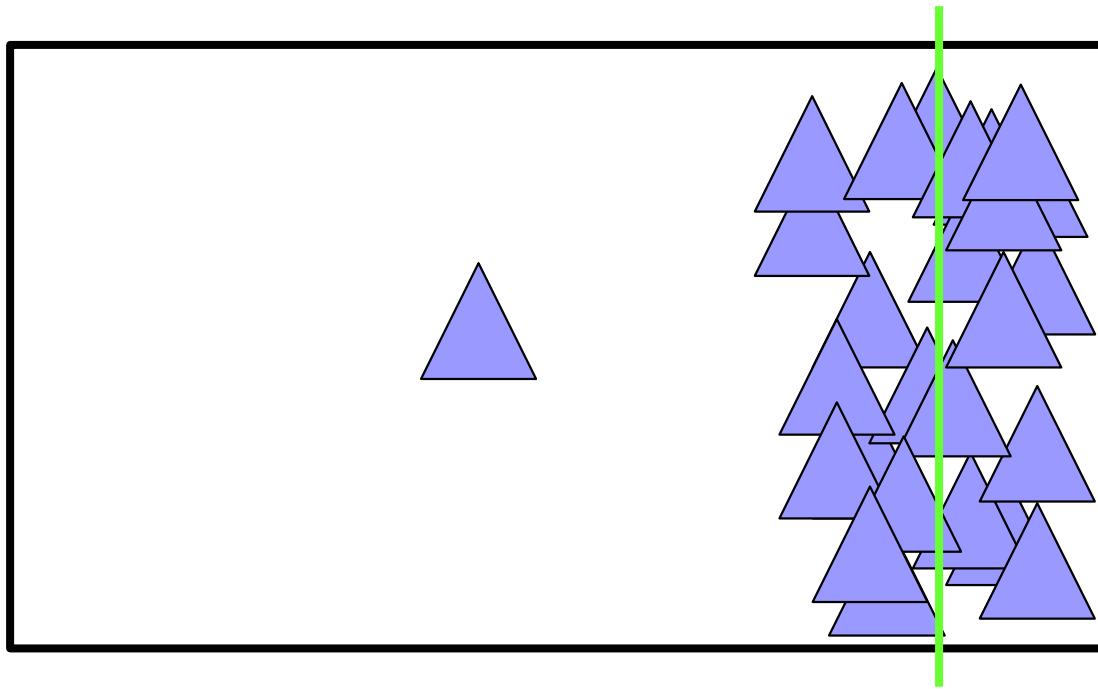


A metà



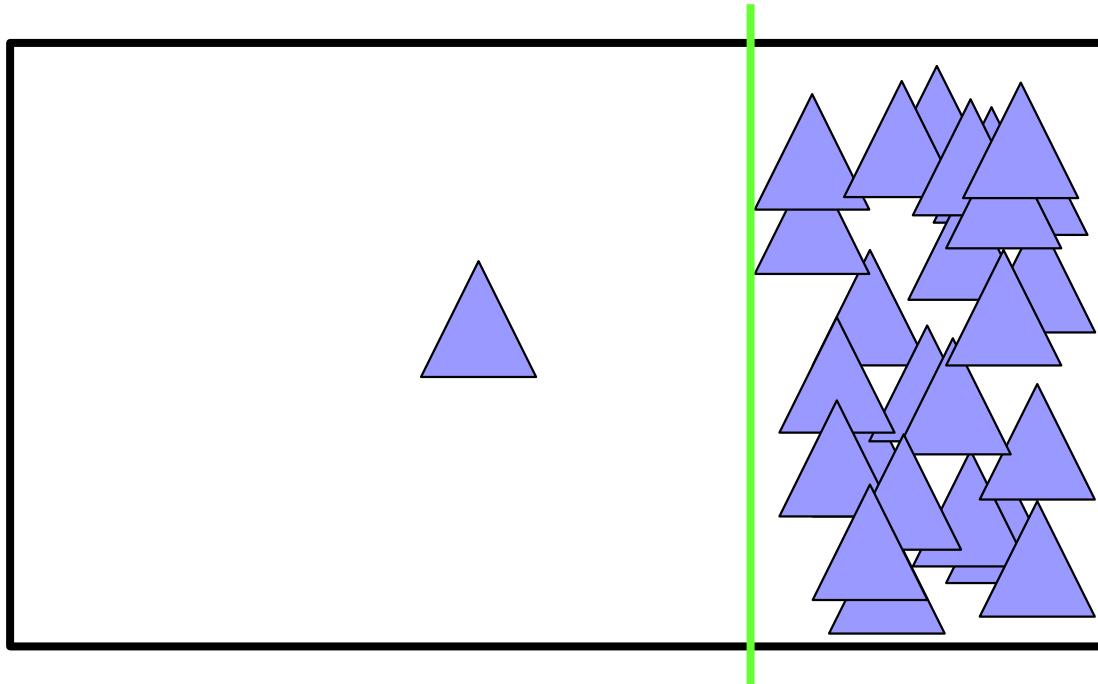
- Non tiene conto dei costi

Nel punto mediano



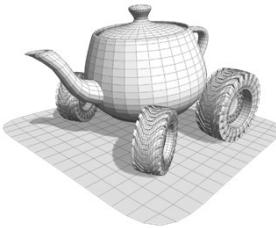
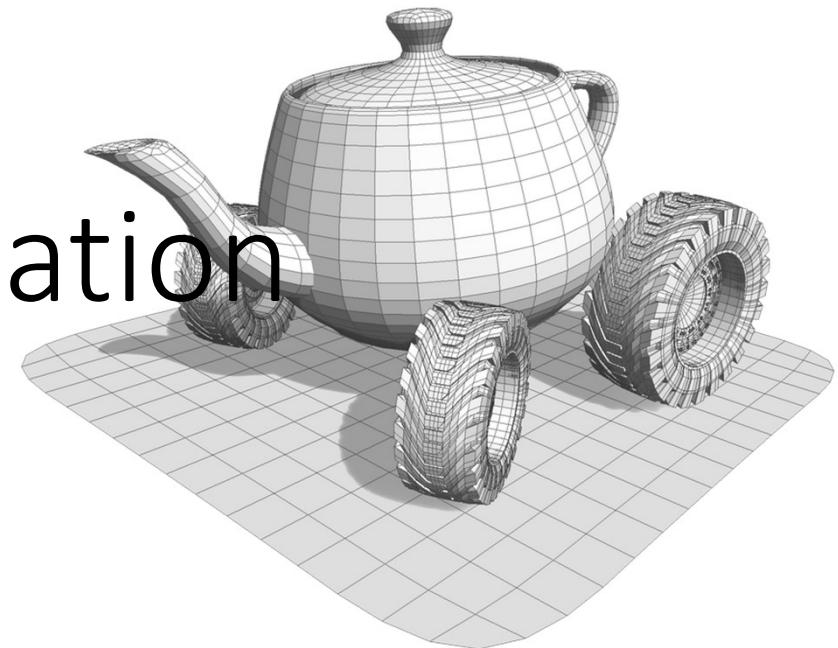
- Rende uguali i costi di *left_cell* e *right_cell*
- Non tiene conto delle probabilità

Ottimizzando il costo

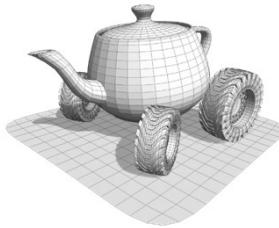


- Separa bene spazio vuoto
- Distribuisce bene la complessità

Global Illumination



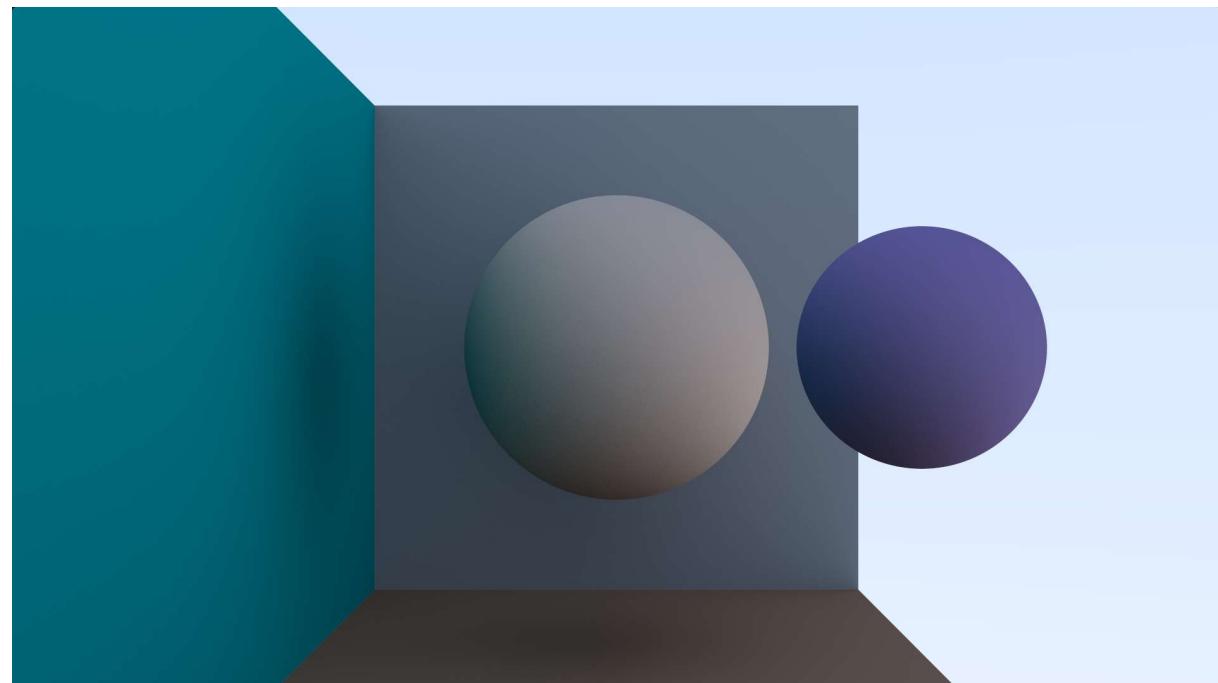
Global Illumination

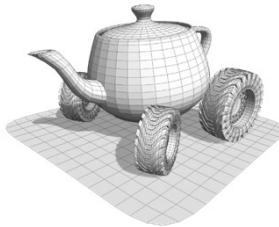


- Light effects that ray tracing handles efficiently? Those which require one ray..
 - Direct illumination
 - Specular reflection
 - Specular refraction
 - Transparency
 - Hard shadows
- We saw we can integrate over the hemisphere for including diffuse reflection, but:
 1. It is costly
 2. It's not very convenient for diffuse surfaces (why?)

Diffuse lighting

- Diffuse lighting changes *slowly* over the surfaces
- Diffuse lighting is light dependent but view independent
- Diffuse lighting contributes a lot to realism
- Could we compute a good approximation of diffuse lighting once for all?





Radiosity

- Recall (again) the rendering equation:

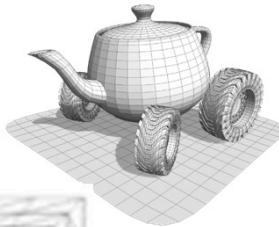
$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) \cos(\theta) d\vec{\omega}_i$$

Emissive component BRDF environment Cosine law

- We made the drastic simplification of considering only the direct lighting, that is, the light arriving from light **emitting** surfaces

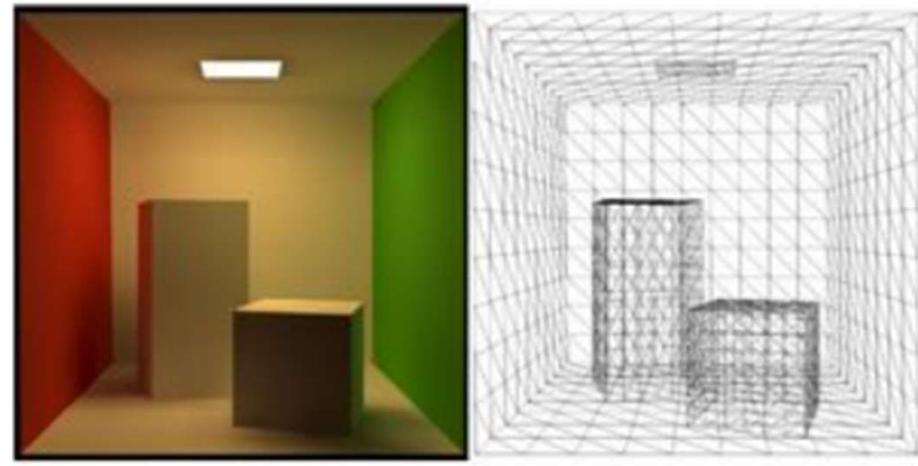
$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \sum_{i=0}^{NLights} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i \cos(\theta)$$

- With the Radiosity technique, **all** lit points are emitting surfaces



Radiosity

- Consider the scene surface partitioned in small **patches** (quadrilaterals, for example)



$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \sum_{i=0}^{NLights} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i \cos(\theta)$$

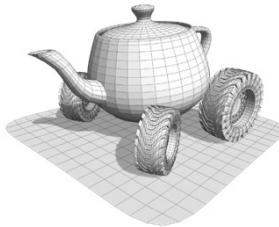
Radiance: direction dependent

Fraction of light from direction $\vec{\omega}_i$ leaving towards $\vec{\omega}_r$

Radiant exitance (or Radiosity): direction **independent**

$$B_i = B_i^e + \rho_i \sum_{j=0}^n F_{i \rightarrow j} B_j$$

Fraction of light from patch j arriving at patch i



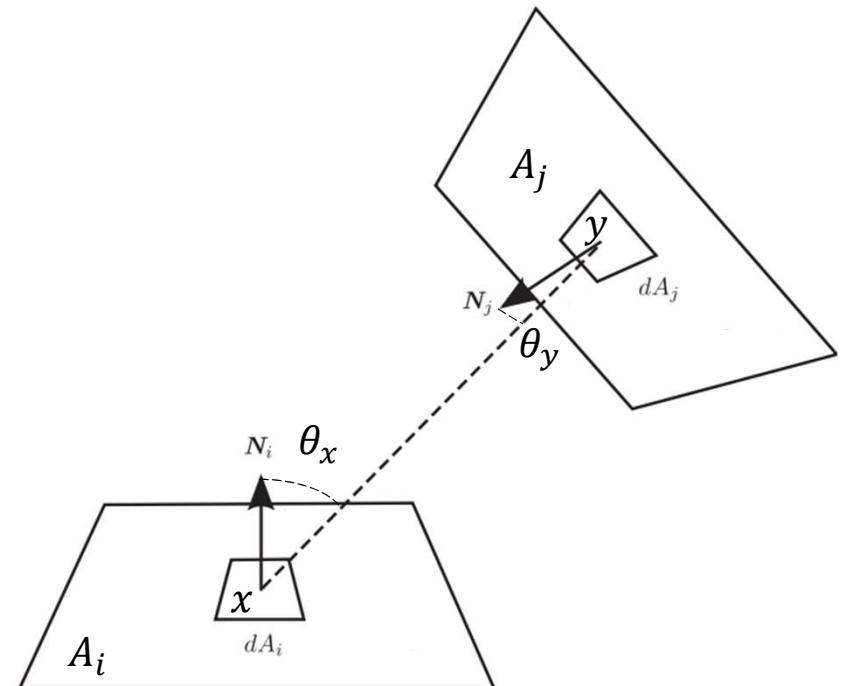
Radiosity: Form Factor

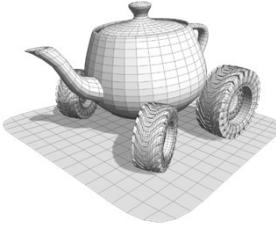
- **Form factor:** fraction of the light leaving a patch j reaches patch i , for all pairs of patches (i, j) : $i, j \in [1, N], i \neq j$

How much *flux* Φ_x for the area light A_j ?

$$\Phi_x = \frac{\Phi_j}{A_j} \int_{A_j} \frac{\cos\theta_x \cos\theta_y}{\pi r^2} V(x, y) dA_j$$

$$V(x, y) = \begin{cases} 1 & x \text{ is visible from } y \\ 0 & \text{otherwise} \end{cases}$$





Radiosity: Form Factor

How much *flux* Φ_i for the area light A_j ? Integrate Φ_x over A_j

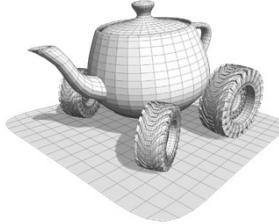
$$\Phi_i = \frac{\Phi_j}{A_j} \int_{A_i} \int_{A_j} \frac{\cos\theta_x \cos\theta_y}{\pi r^2} V(x, y) dA_j dA_i$$

$$\boxed{\frac{\Phi_i}{\Phi_j}} = \frac{1}{A_j} \int_{A_i} \int_{A_j} \frac{\cos\theta_x \cos\theta_y}{\pi r^2} V(x, y) dA_j dA_i = F_{j \rightarrow i}$$

flux on area A_i
flux from area A_j

Note: if we swap i and j this term remains the same.
It only depends on the geometry, not on the flux

$$\Rightarrow F_{j \rightarrow i} A_j = F_{i \rightarrow j} A_i$$



Radiosity: Transport Equation

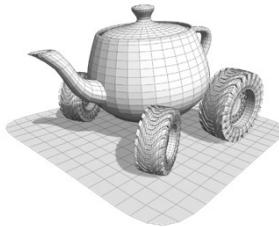
$$\Phi_i = \Phi_i^e + \rho_i \sum_{j=0}^n F_{j \rightarrow i} \Phi_j =$$

Recall: $B = \frac{\partial}{\partial A} \phi = \frac{\Phi}{A}$ (if the flux is constant over the area)

$$\frac{\Phi_i}{A_i} = \frac{\Phi_i^e}{A_i} + \rho_i \sum_{j=0}^n F_{j \rightarrow i} \frac{\Phi_j}{A_i}$$

$$B_i = B_i^e + \rho_i \sum_{j=0}^n F_{i \rightarrow j} B_j$$

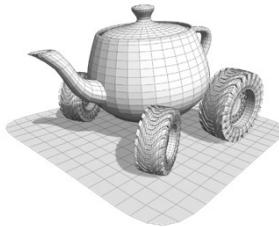
$$\frac{F_{j \rightarrow i}}{A_i} = \frac{F_{i \rightarrow j}}{A_j}$$



Radiosity: set up the system

- A linear equation for each patch: assemble all the equations in a linear system
 - $\rho_i, F_{i \rightarrow j}$ known for all i, j
 - B_i^e known (the lights)
 - B_i unknown (the radiosity for each patch)

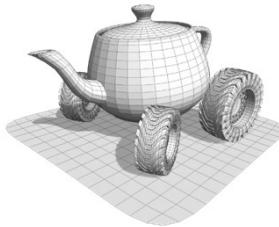
$$\begin{bmatrix} 1 - \rho_1 F_{1 \rightarrow 1} & -\rho_1 F_{1 \rightarrow 2} & \cdots & -\rho_1 F_{1 \rightarrow n} \\ \rho_2 F_{2 \rightarrow 1} & 1 - \rho_2 F_{2 \rightarrow 2} & \ddots & \vdots \\ \vdots & & \ddots & \vdots \\ -\rho_n F_{n \rightarrow 1} & \cdots & -\rho_n F_{n \rightarrow n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} B_1^e \\ B_2^e \\ \vdots \\ B_n^e \end{bmatrix}$$



Radiosity: solving the system

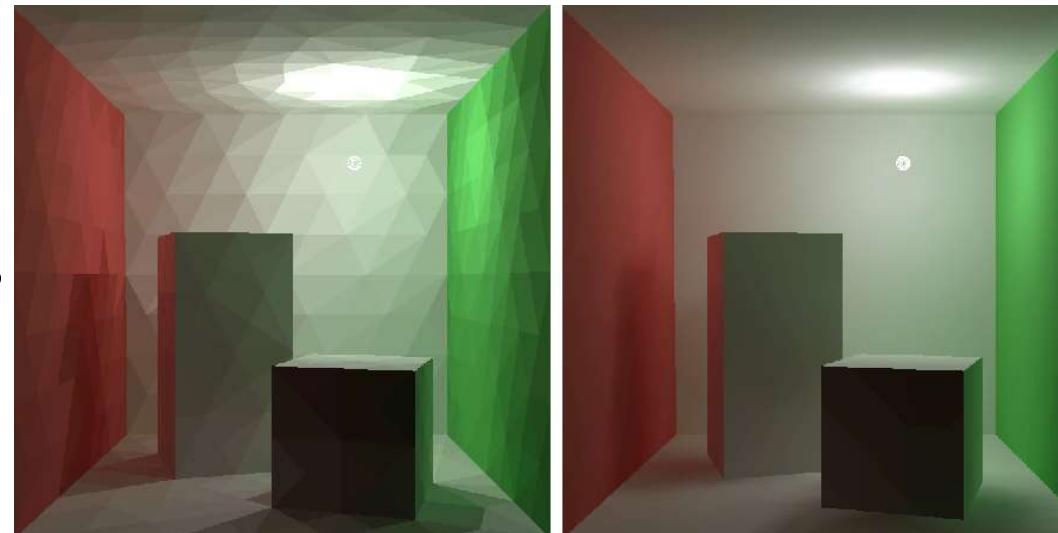
- For a meaningful result we need a fine discretization of the surface
 - Many thousand of patches are a common scenario
- Direct inversion of the matrix is quite impractical, iterative methods are preferred

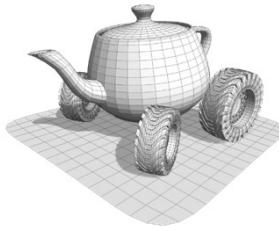
```
// Gauss-Seidel Iteration: Radiosity Gathering Method
while (not converged)
    for patch  $i=1$  to  $N$ 
         $B_i = B_i^e + \rho_i \sum_j B_j F_{i \rightarrow j}$ 
    endfor
endwhile
```



Radiosity: rendering

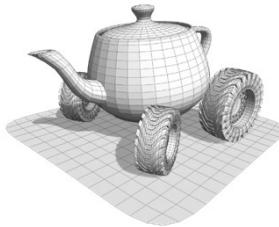
- Recall: the solution is found *per-patch*
 - That is, each patch has a constant radiance...visible differences between adjacent cells
- Radiance is computed at the vertices (as the average of the incident face) and the interpolated
 - In other words _____ shading





Radiosity: conclusion

- Pros
- Cons:
 - Requires a complete tessellation of the scene, which may be difficult for non parametric surfaces
 - The tessellation influences the final result
 - What if a patch crosses would-be radiosity discontinuities?
 - It's output insensitive
 - Does not matter the view, you still have to solve the whole problem
- How to obtain a similar result and avoid these drawbacks?

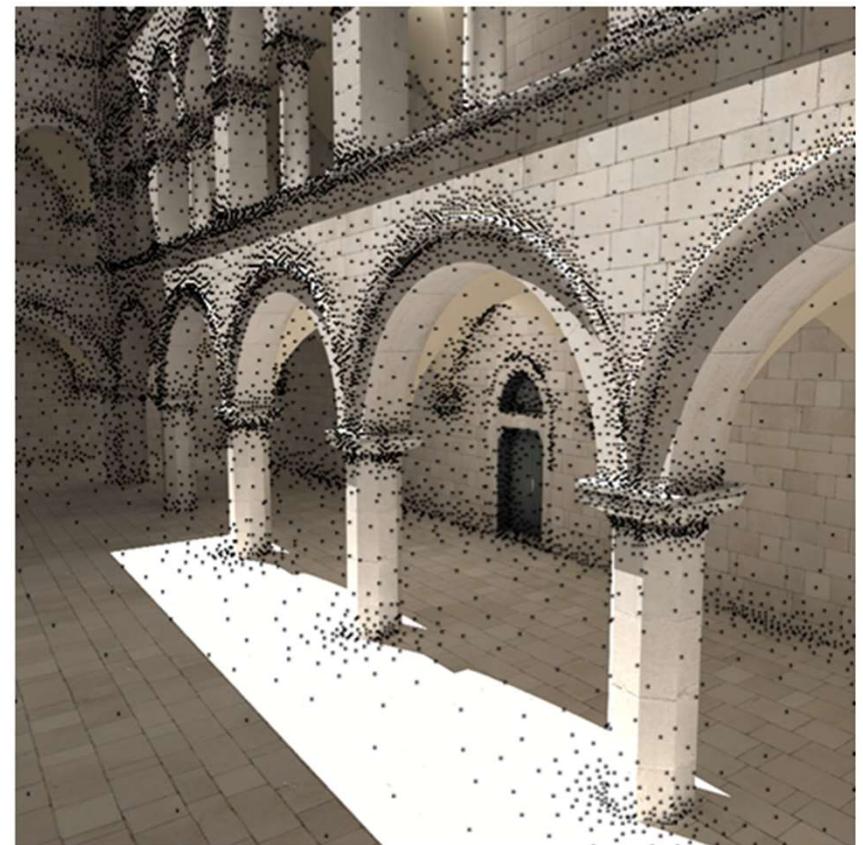


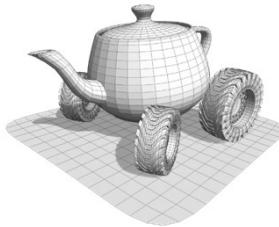
Irradiance caching

- Key idea: compute irradiance only for sparse visible points of the scene
- Interpolate/extrapolate elsewhere
- Algorithm:

```
Irradiance(x) {  
    Y = {y | near X and stored Irr(y)};  
    if( Y not empty )  
        Irr_x = InferIrradianceFrom(Y);  
    else  
        Irr_x = ComputeIrradianceAt(x);  
    return Irr_x;  
}
```

How ?



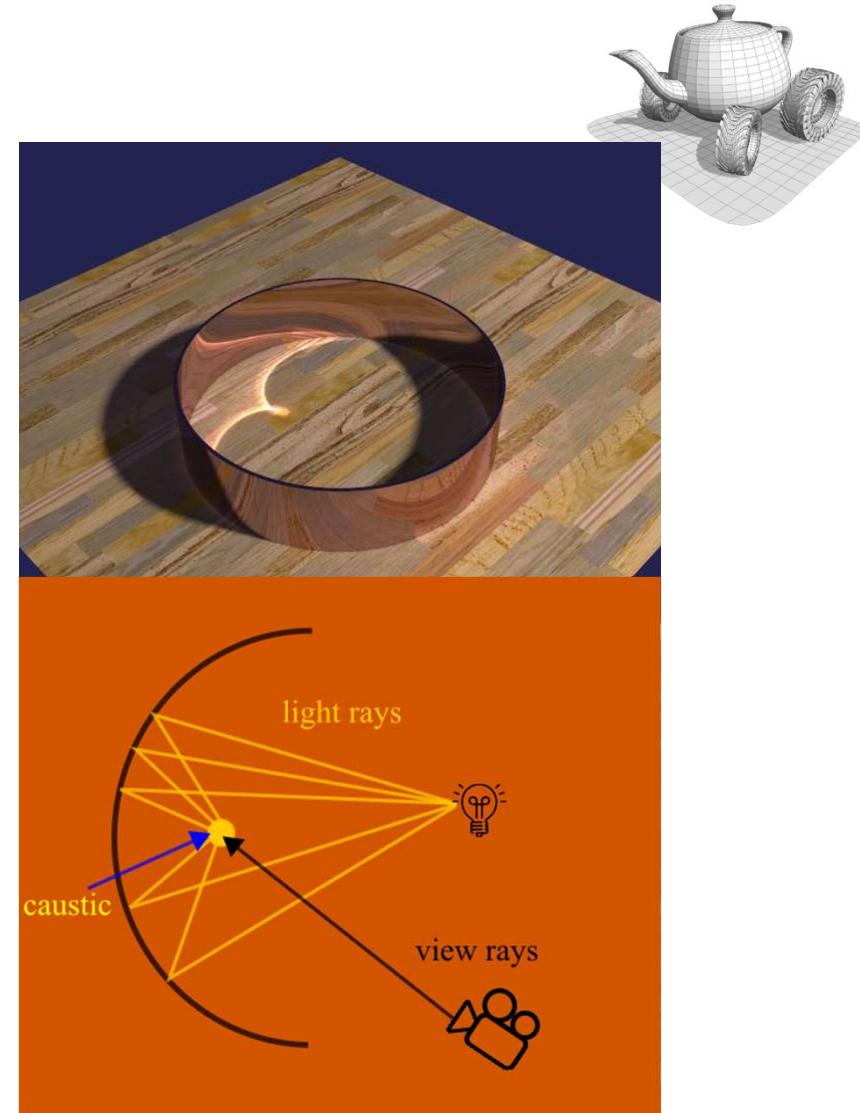


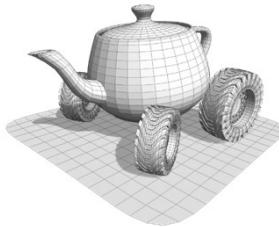
Irradiance Caching

- Computing irradiance at x : trace paths on the scene by Montecarlo Importance Sampling (cosine weighted)
- Irradiance interpolation
 - Heuristic evaluation of the Irradiance derivative $\epsilon_i(x, \vec{n})$, that say how quickly the irradiance will change around x
 - Interpolation weights of the stored samples is set to $w_i(x, \vec{n}) = \frac{1}{\epsilon_i(x, \vec{n})}$
 - Samples with w_i above a given threshold are used for interpolation
 - If there are none the irradiance needs to be evaluated

Raytrace these!

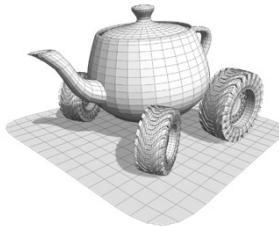
- Caustics: light rays that are focused on a region because of the shape of the reflective/refractive surfaces
- Q: Could we compute these effects with ray tracing?
 - A: it would take an extremely dense sampling of directions
 - A: it would produce a noisy image
- What if we trace rays from the light source and watch them accumulate on the scene (sort of) ?





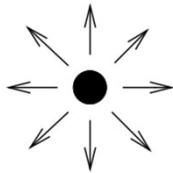
Photon mapping at a glance

- A two-step algorithm
 1. **Emit** the photons from the light sources and trace them in the scene
Every time a photon hit a non-specular surface it leaves part of its energy
Terminate when the remaining energy is below a threshold or a maximum number of bounces is reached
 2. **Gather**: do a ray-tracing step, gather photon in the neighbor of the hit point and compute the radiance.

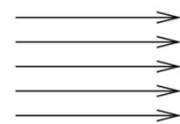


Emitting photons

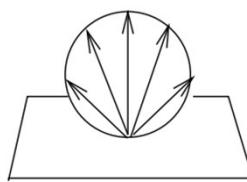
positional



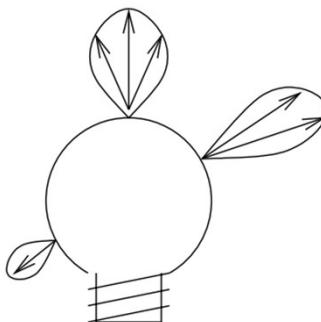
directional



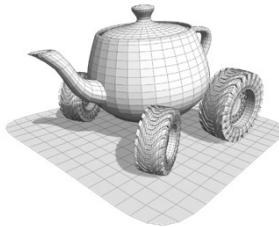
Area light



Generic surface light



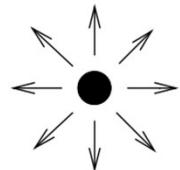
- Photon mapping models the *radiant flux* Φ (Watt) of light as *partitioned* in photons
- If n photons are emitted from a light with power P_l , each photon will *carry* $P_p = \frac{P_l}{n}$
- ..in the assumption photons will *enough* and *uniformly distributed* over the exiting direction



Emitting photons

Generic surface light

positional

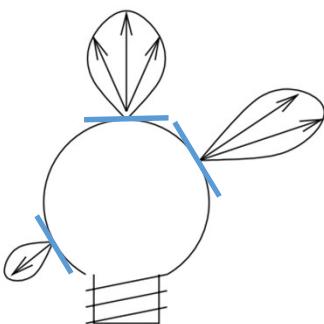
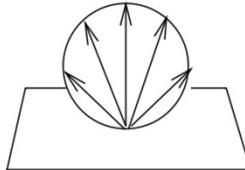


Pick a random direction on the sphere

directional



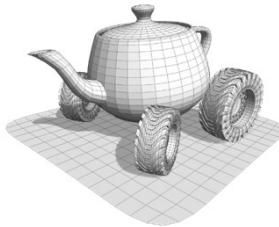
Area light



Uniformly distributed random direction on the sphere

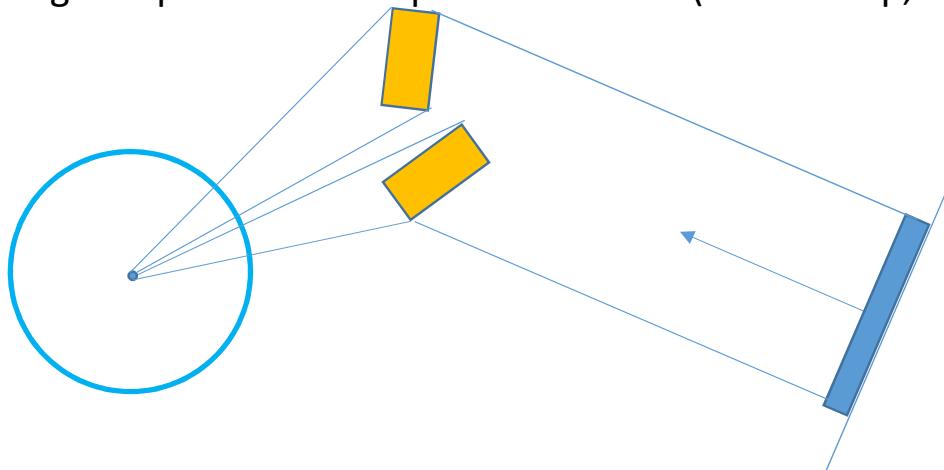
- 1. Generate $d \in [-1,1]^3$
- 2. If $|d| > 1$ goto 1
- 3. $r = \frac{d}{|d|}$
- 4. If $r.y < 0 r = -r$

On the hemisphere

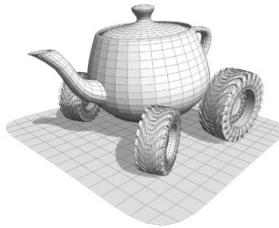


Emitting photons: improving efficiency

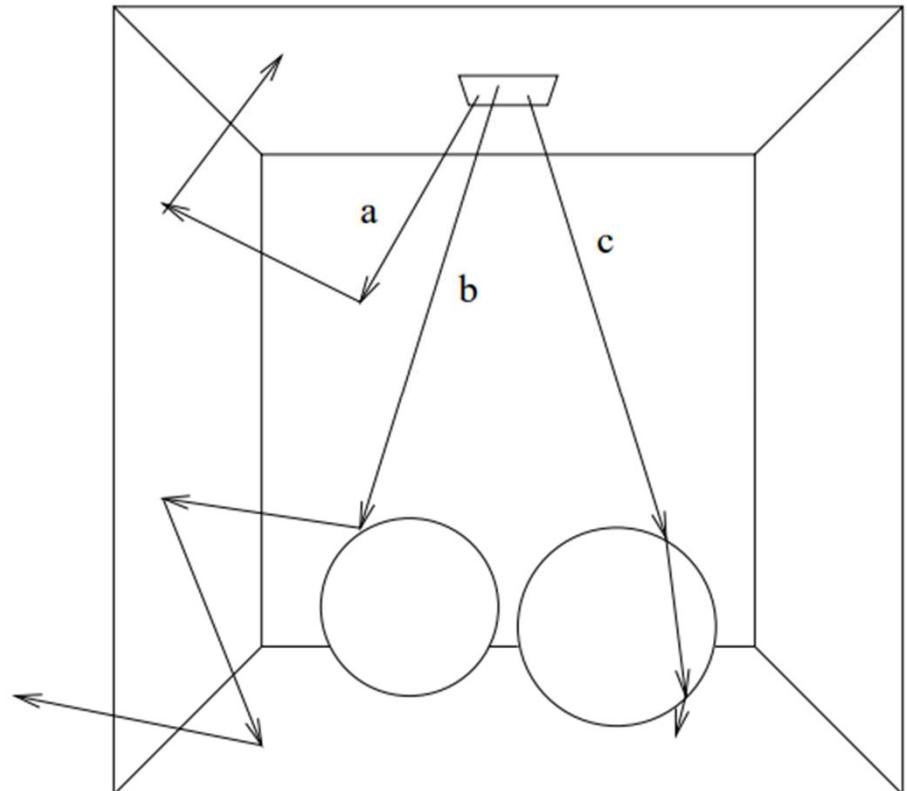
- Do not emit photons that are not going to hit any part of the scene
- **Projection Map:** a map that store, for each possible emission direction, if they the photon will or will not hit the scene
- Implementation:
 - directional light: a texture covering the projection of the scene on a plane orthogonal to the ray direction
 - Positional light: a parametrized spherical surface (a cubemap, an equilatetral map..)



Tracing Photons



- Photons hit the scene, they can do:
 - Specular Reflection/Refraction (S)
 - Diffuse Reflection (D)
 - Absorbtion (A)
- Example:
 - a) DDA
 - b) SDD
 - c) SSA



Tracing Photons

- When a photon hits a point with diffuse coefficient d , specular coefficient s and absorption $1 - s - d$ we could
 - Generate two new photons with power scaled by d and s (resp.).
 - BAD: number of photons doubles at every bounce while their energy decreases

OR

- Generate:
 - No photon with probability $1 - s - d$
 - One photon in a random direction with probability d
 - One photon in the specular direction with probability s

Russian roulette

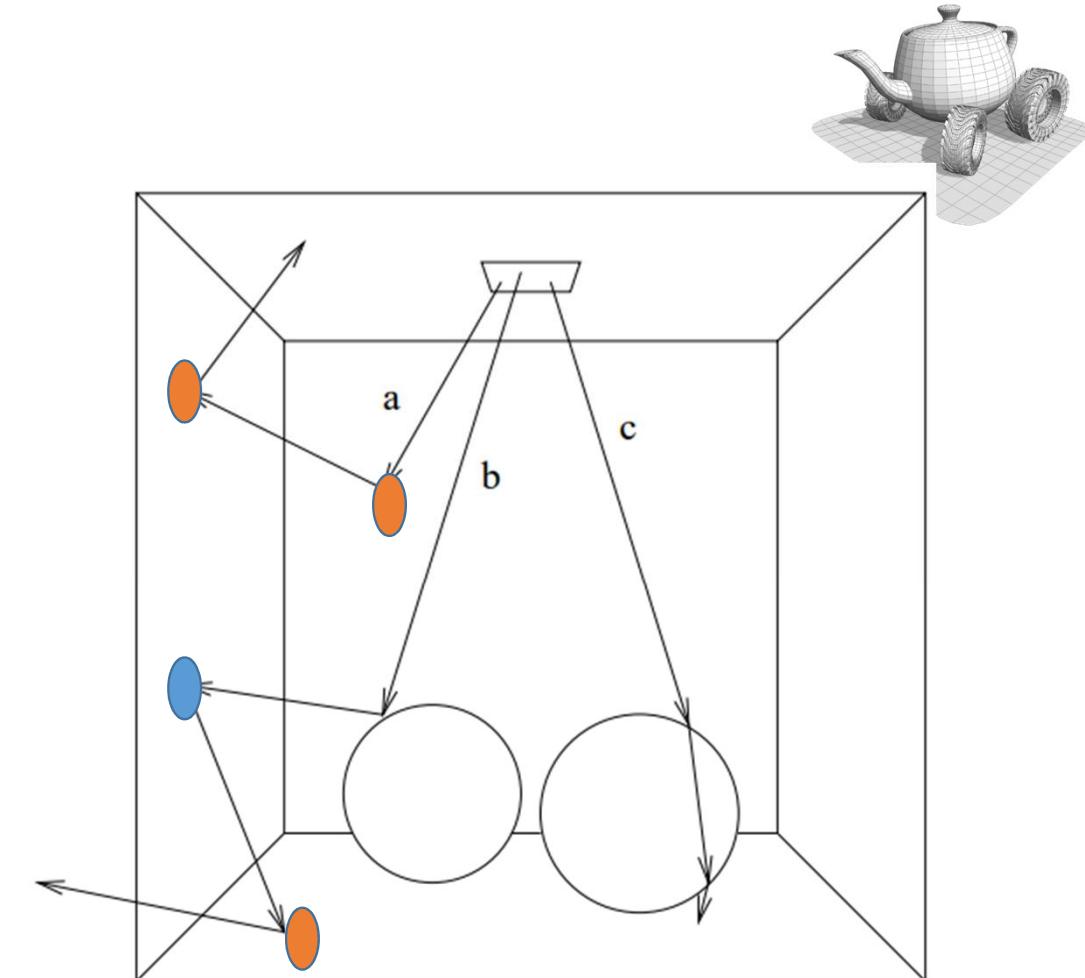


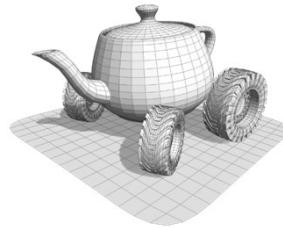
$\xi \in [0, d]$	→	diffuse reflection
$\xi \in]d, s + d]$	→	specular reflection
$\xi \in]s + d, 1]$	→	absorption

$$\xi \text{ uniform } \in [0,1]$$

Storing Photons

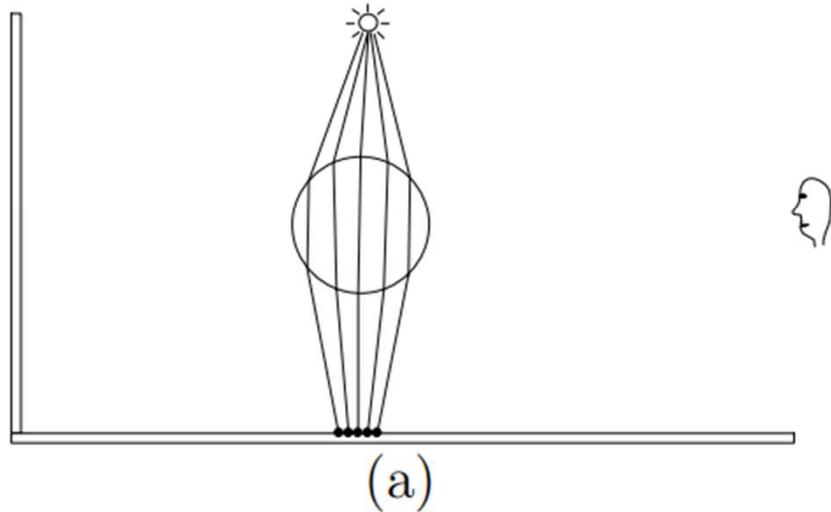
- Photons are stored in 3D data structures called **Photon Map**
- 2 types of photon maps
 - **Caustic photon map**, for photons that hits a specular surface before hitting a *diffuse* surface (like path b)
 - **Global photon map**, for all photons hitting a *diffuse* surface
- Q: why just *diffuse* surfaces?



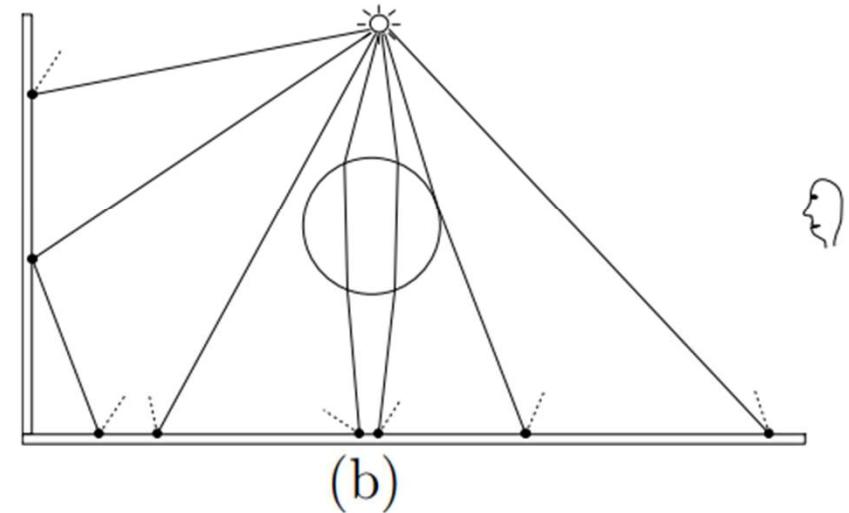


Photon Maps

Caustics photon map



Global photon map



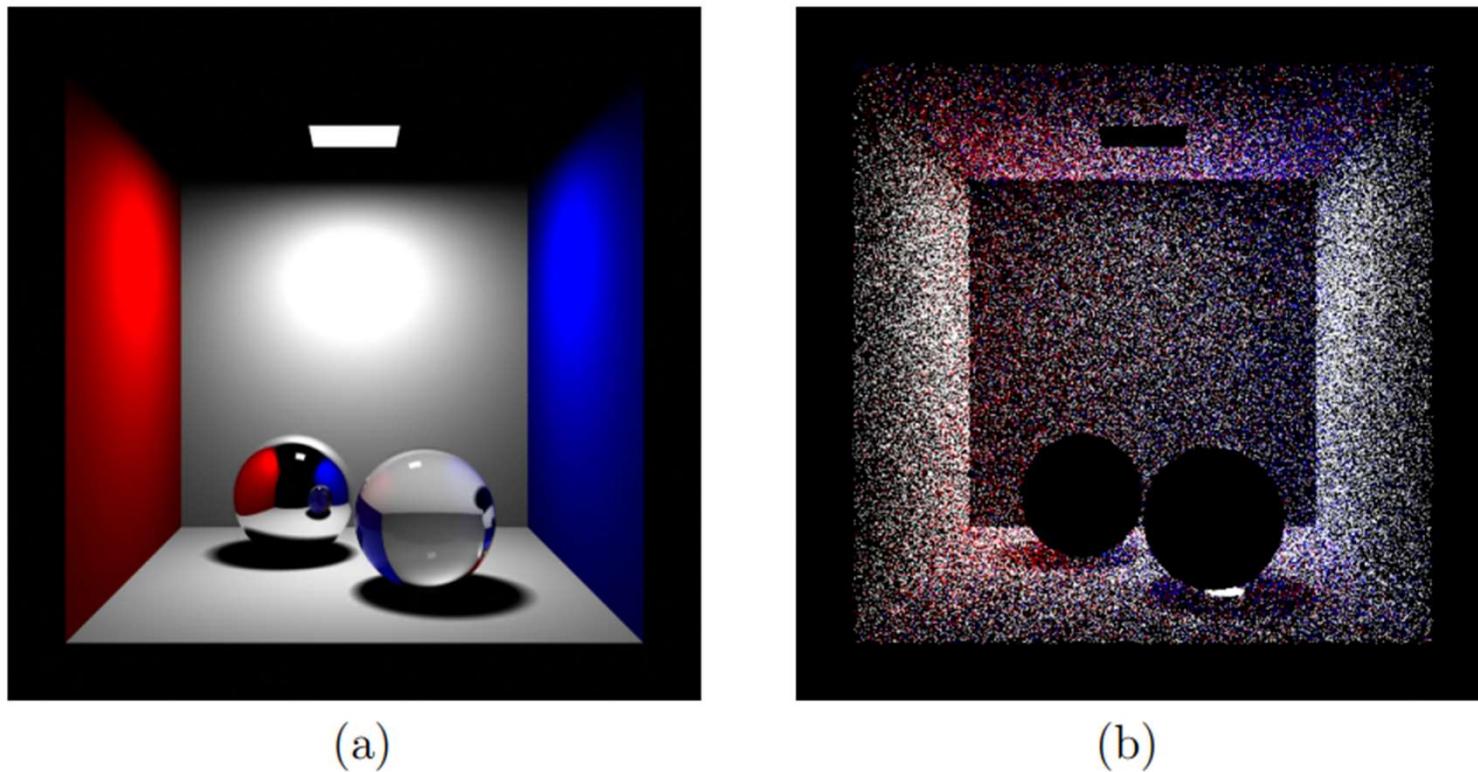
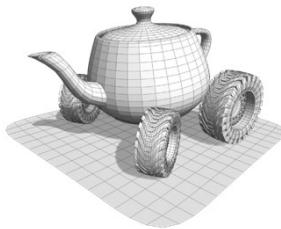
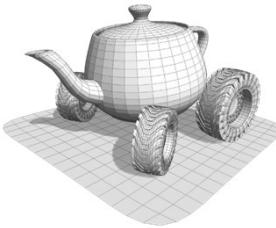
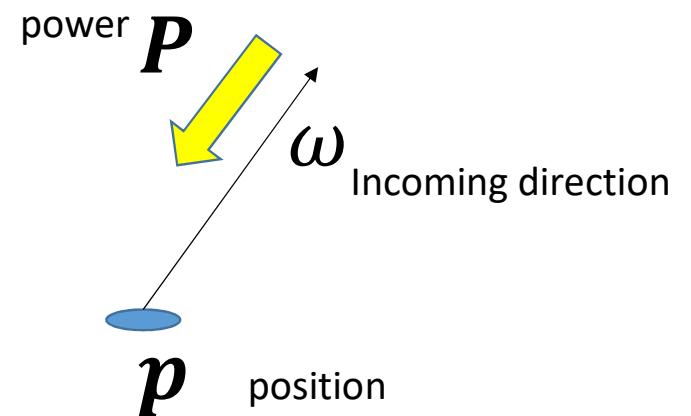


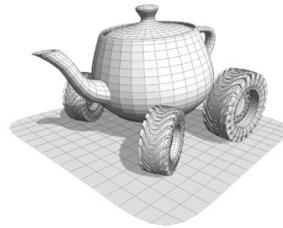
Figure 4: “Cornell box” with glass and chrome spheres: (a) ray traced image (direct illumination and specular reflection and transmission), (b) the photons in the corresponding photon map.

Storing a Photon



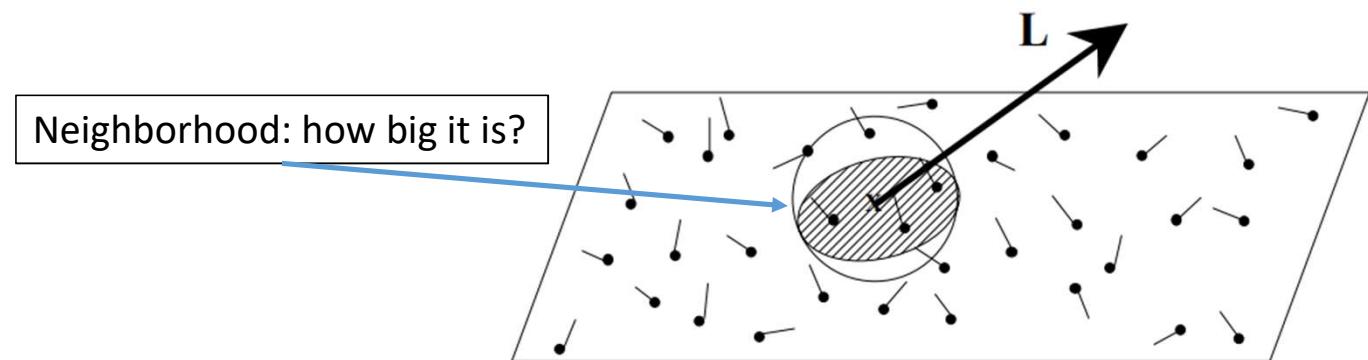
- A photon is stored as: position, incidence direction and power
- When the first pass of PM is done, the surfaces will be full of photons
- Second pass: for a given point p and a given direction (a view ray) estimate the radiance
- Q1: what's the difference with Lightmaps?
- Q2: what's the direction ω is used for?

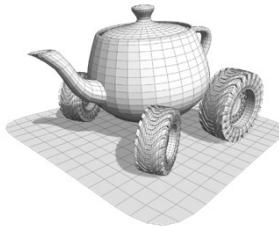




Gathering: ray-tracing pass

- Ray tracing pass
 - Radiance due to specular reflection and direct illumination as before:
 - Bounce toward the reflected direction (Montecarlo Importance Sampling)
 - Radiance due to diffuse lighting
 - find for photons *around* the hit point
 - Compute the radiance using the photons





From photons to radiance

- We want to compute radiance $L_r(x, \vec{\omega})$

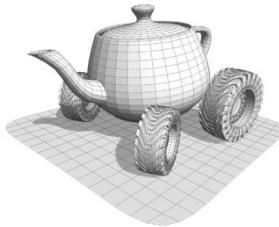
$$L_r(x, \vec{\omega}) = \int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}') |\vec{n}_x \cdot \vec{\omega}'| d\omega'_i$$

- We don't know $L_i(x, \vec{\omega}')$, photons do not store radiance but **radiant flux (Watt)**

- Recall that $L = \frac{d^2\Phi}{dA \cos \theta d\omega}$

$$\begin{aligned} L_r(x, \vec{\omega}) &= \int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) \frac{d^2\Phi_i(x, \vec{\omega}')}{\cos \theta_i d\omega'_i dA_i} |\vec{n}_x \cdot \vec{\omega}'| d\omega'_i \\ &= \int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) \frac{d^2\Phi_i(x, \vec{\omega}')}{dA_i}. \end{aligned}$$

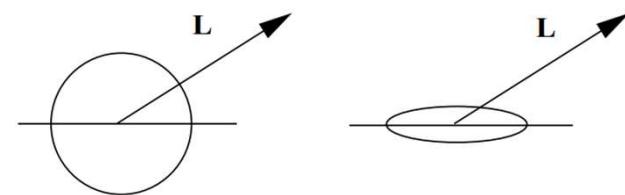
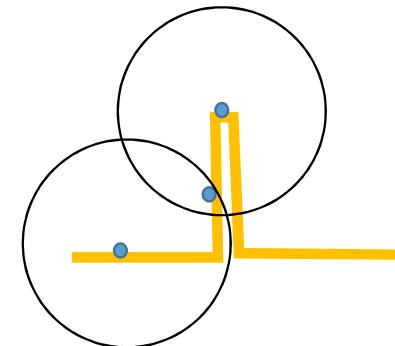
- Counting only on the available photons $\rightarrow L_r(x, \vec{\omega}) \approx \sum_{p=1}^n f_r(x, \vec{\omega}_p, \vec{\omega}) \frac{\Delta\Phi_p(x, \vec{\omega}_p)}{\Delta A}$



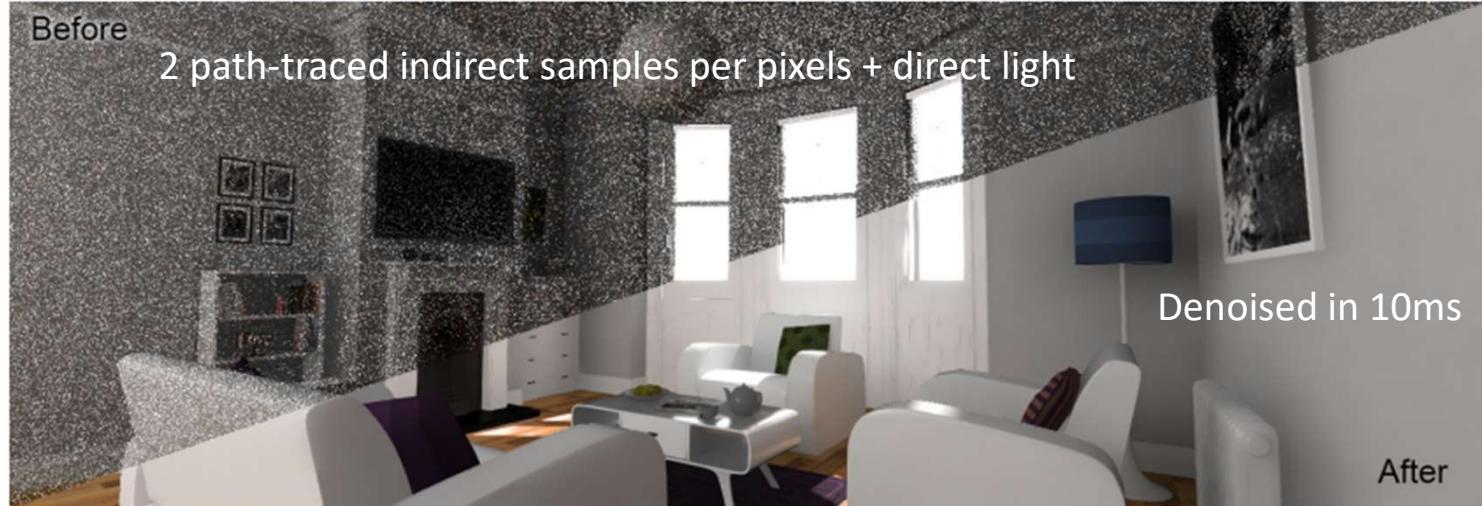
Finding the photons

- Key idea: search around x until n photons are found
 - It requires assumption on the density of the photon map
 - It may include corners and sharp edges
 - Compress the sphere along the surface normal (towards becoming a disk)
- How to find the photons quickly?
 - Use a hierarchical space indexing data structure such as kd-tree

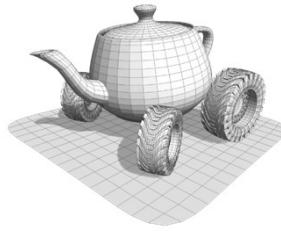
$$L_r(x, \vec{\omega}) \approx \sum_{p=1}^n f_r(x, \vec{\omega}_p, \vec{\omega}) \frac{\Delta\Phi_p(x, \vec{\omega}_p)}{\Delta A}$$



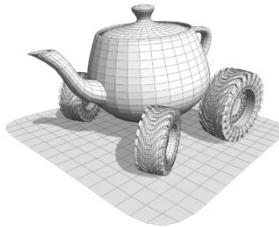
Denoising



An Efficient Denoising Algorithm
for Global Illumination, Mara et al. HPG '17,



- Denoising = remove the noise
 - More like «reconstruct a coherent signal from a mess»
- Denoising techniques had a revamp after real-time ray tracing became mature enough



Denoising by Filtering: just one example

- Recall the Gaussian kernel (see Variance Shadow maps lec.)
- better than averaging, it still «blur» the result
 - Oblivious to the «value» of the sample
- **Bilateral filtering:** a variation of the Gaussian kernel where the value is taken into account

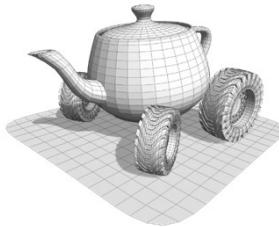
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)}{\sigma^2}}$$

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in \mathcal{S}} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|) I_{\mathbf{q}}$$

$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in \mathcal{S}} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)$$

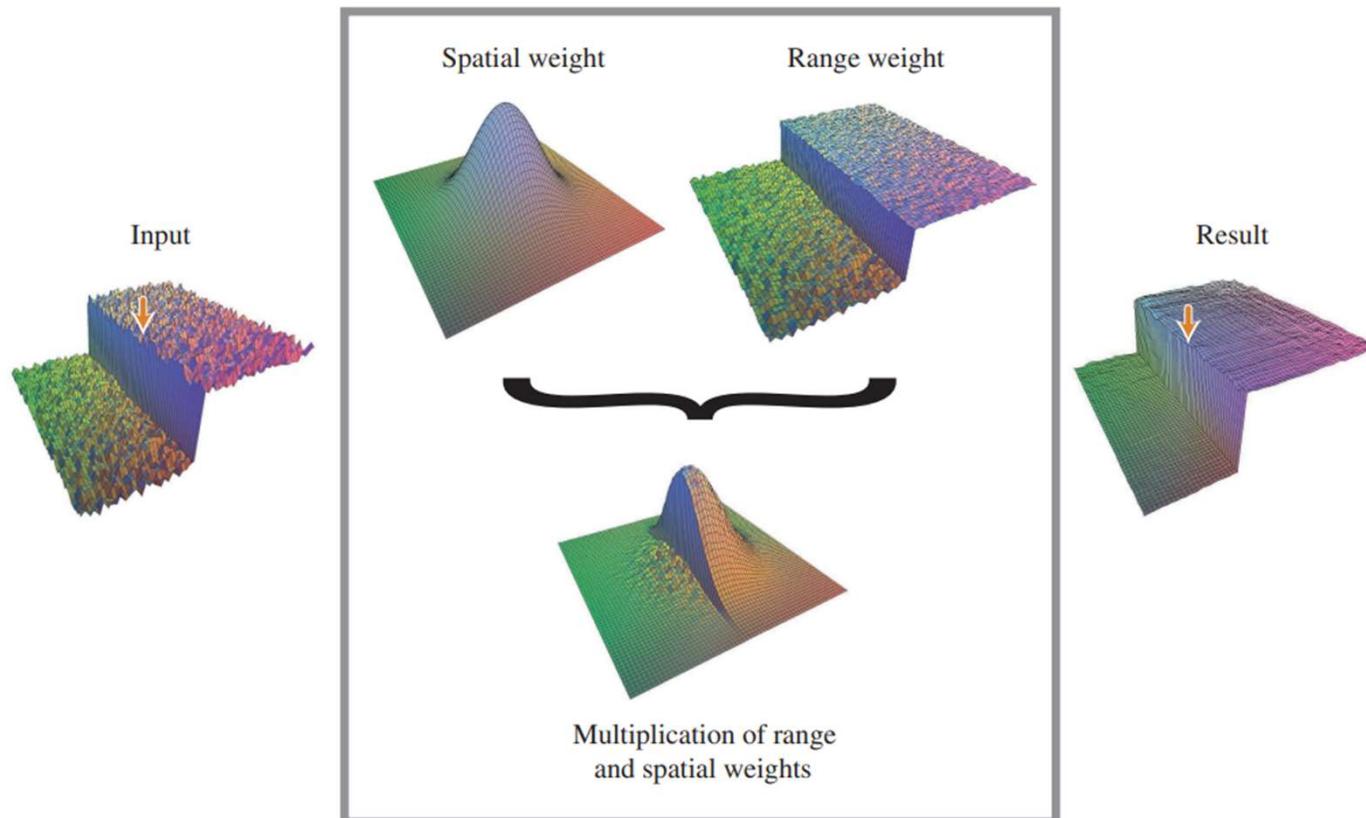
Gaussian Term

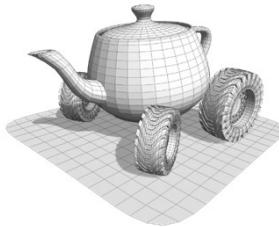
Values dissimilarity



Bilateral Filtering

Bilateral filter weights at the central pixel





references

- Radiosity (chap-11, dispense_2023.pdf)
- Irradiance caching: <https://cs.dartmouth.edu/wjarosz/publications/dissertation/chapter3.pdf>
- photon mapping <http://graphics.stanford.edu/courses/cs348b-01/course8.pdf>