# 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

# 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
- …

# Corsi collegati

- Questo corso sarà molto utile a chi proseguirà il persorso 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

Capo del VCL

colleghi al 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

Non serve comprarlo!

- Introduction to Computer Graphics: a Practical Learning Approach
  F.Ganovelli, M.Corsini, S.Pattanaik, and M.Di Benedetto
  CRC Press, 2014

È Creative Commons

- Ray Tacing 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

# 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 obscurance. 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,R
  - 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 trasforma
  - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettiche. Trasf
- **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. F
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscurance. Su
- **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,Ra
  - 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 trasforma
  - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospettiche. Trasf
- **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. P
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscurance. Su
- **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 co
  - Pipeline di rendering rasterization based ( o proiettiva) ,Raytrac
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensi
  - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rende
  - Trasformazioni di base:I frames, organizzazione gerarchica delle
  - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospett
- **Chapter 6** Lighting and Shading
  - Interazione luce/materia: Riflessione, rifrazione, assorbimento, scatte
  - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Minificatio
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscurance
- **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)

- 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)

# Argomenti

- **Chapter 1** Fondamenti
  - Il sistema di visione umano, Color spaces, illuminant, gamma co
  - Pipeline di rendering rasterization based ( o proiettiva) ,Raytrac
- **Chapter 3+** Rappresentazione di superfici e volumi tridimensi
  - Rappresentazioni esplicite e implicite di superfici e volumi
- **Chapter 4** Trasformazioni geometriche nella pipeline di rende
  - Trasformazioni di base:I frames, organizzazione gerarchica delle
  - Dallo spazio 3D allo schermo: proiezioni ortografiche e prospett
- **Chapter 6** Lighting and Shading
  - Interazione luce/materia: Riflessione, rifrazione, assorbimento,
  - Phong lighting, Cook-torrance, Oren-Nayar, Minnaert
- **Chapter 7** Textures. Concetti di base. Magnification and Mini
- **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)

- **Chapter 7** Textures. Concetti di base. Magnification and Minification. Perspective correct
- **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
- **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 Minifi
- **Chapter 8** Ombre: Shadow mapping e shadow volumes; Ambient obscu
- **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 trac

Where is the car racing??

# Introduction to Computer Graphics

# What 3D **C**omputer **G**raphics 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 pratictioners 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, anyting 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 Insitute for Gravitational Physics)

Rhinovirus 3 protein

# Scientific Visualization

(surface of a glass – Nanorobotics Lab (Carnegie Mellon University )

# 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

Illustrative rendering

- data abstract (N dimensional)
- Goal: readability
- Means: graphs, gliphs...wathever it works

# 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
  - **C**omputer **A**ided **D**esign
  - 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)

Jurassic Park - Universa Studios 1993

Star Wars: The Phantom Manace - Lucasart 1999

# 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

# Computer Graphics: applicazioni

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

Final Fantasy – Squaresoft 2001

Final Flight Of the Osiris – Squaresoft 2003

Avatar – ILM 2009

# Uncanny Valley

# Computer Graphics: applications

- Entertainment: games, the leading business $$$

Battlezone – Atari 1980

Tailgunner - Cinematronics 1979

# Computer Graphics: applicazioni

- Entertainment: games, the leading business $$$

Virtua Fighter - Sega 1993

Doom – IDsoft 1993

# Computer Graphics: applicazioni

- Entertainment: games, the leading business $$$

GTA -

World of Warcraft,
Blizzard Entertainment 2004

# 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:

- Frution / 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

acquisition

use

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 in 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-redendered + 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

information
modeling
3D model
rendering
geometry processing
Digital images
Final application

Model of the earthquake
Physical simulation
Height fields
rendering
Color coding, other manipulations
Final application

concept
e.g. for a character
3D modeller (digital artist)
e.g. using 3D Studio Max, Maya, Blender…
low-poly
polygonal mesh
rendering
Digital
Images
rigging + skinning,
LOD-ing,
uv-mapping
game engine

# 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;!x&&i<d.forms.length;i++)
x=d.forms[i][n];

   ...
```

HTML page + images, ccs, 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 )
  (  24,  25,  26,  27,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,  40 )
Lid:
  (  96,  96,  96,  96,  97,  98,  99, 100, 101, 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 ), (  1.3375,  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, 2.02500 ),
  ( -1.6000, -0.3000, 2.02500 ), ( -1.5000, -0.3000, 2.25000 ), ( -1.5000,  0.0000, 2.25000 ),
  ( -2.3000,  0.0000, 2.02500 ), ( -2.3000, -0.3000, 2.02500 ), ( -2.5000, -0.3000, 2.25000 ),
  ( -2.5000,  0.0000, 2.25000 ), ( -2.7000,  0.0000, 2.02500 ), ( -2.7000, -0.3000, 2.02500 ),
  ( -3.0000, -0.3000, 2.25000 ), ( -3.0000,  0.0000, 2.25000 ), ( -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 ),
  ( -2.5000,  0.0000, 1.35000 ), ( -2.5000,  0.0000, 1.12500 ), ( -2.5000, -0.3000, 1.12500 ),
...
```

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

1993

- Less and less different…

Virtua Fighter - Sega 1993

real time

Jurassic Park - Universal Studios 1993

offline

# Real Time VS Offline rendering

2001

- Less and less different…

real time

offline



Jurassic Park - Universal Studios 1993

# Real Time VS Offline rendering

2007

- Less and less different…

real time

offline



Jurassic Park - Universal Studios 1993

# Real Time VS Offline rendering

2021

- Less and less different…

real time

offline



Jurassic Park - Universal Studios 1993

# 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/ganov

CGCourse2023
-- 3dparty
*readme.txt*
-- src
-- code_0_setup
*main.cpp*
*readme.txt*

# Let’s setup: fecth GLFW

- Download GLFW from [https://www.glfw.o](https://www.glfw.o)
  - You can download the precopiled binaries and
  - Or you can download the source code and con
- Place .h and .lib in the 3dparty folder:

CGCourse2023
 -- 3dparty
  -- glfw
   -- include
   -- lib

# 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

# Let’s setup: set libraries

# Let’s setup: set library path

# 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 ouput 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

A constant number of indepent runs. It looks very parallelizable

Variable number of primitives. Looks bad..

```
For each pixel p:
    make a ray r (viewpoint to p)
  for each primitive o in scene:
      find intersect(r,o)
  keep the closest intersection o_j
  find color of o_j at p
```

# 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

# 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 or **bounces**
  - Higher N, more realistic reflections, longer run time

# Ray-Tracing

Refraction ray

Reflection ray

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

Primary ray

- 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…

# Ray-Tracing: the Cornell Box

Real photograph with
controlled lighting conditions

Ray traced rendered model

# Ray-Tracing: examples

# Ray-Tracing: examples

# 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

Cost for a ray

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

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”

# Rasterizing lines

- Line equation:
  - Slope→ $m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$
  - $y = y_0 + m(x - x_0)$

**Discrete Differences Analyzer (DDA)**

```
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:
x0 <= x1
-1 <= m <= 1

$$m = \frac{(y_1 - y_0)}{(x_1 - x_0)} \quad -1 < m < 1$$

*yes* → $y = y_0 + m(x - x_0)$

*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 triangles 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 triangles?
- 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

Cost for «transforming» a vertex

Depends on the primitive size on screen

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

Number of vertices

Cost of rasterizing primitive *p*

# RASTERIZATION VS RAY-TRACING

# Rendering Algorithms Paradigms

RAY-TRACING

```
for each pixel
    for each primitive
```

RASTERIZATION BASED:

```
for each primitive
    for each pixel
```

# 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,kl,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
->kl*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->kl,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, Saarbrueken - 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

Rasterization

# 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 reffered 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 in 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

- **B**ounding **V**olumes **H**ierarchies (**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 intersetion to $O(\log n)$ (on average)

# Rasterization acceleration: Z-Buffer

- The rasterization algorith 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 reffered 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 in 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 XBOX )
    - 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

```
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);

 }
```

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

It could be: GL_LINES, GL_QUADS etc..

# Clip Space

- All our coordinates will ultimately be projected in **clip space**, that is [-1,1]x[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 bufferin 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 no going to be changed. OpenGL implementation can optimize the da organization as a consequence

# Describing the render data

Normalize the data in a OpenGL predefined interval

**Stride**: how many bytes from the begin of a vertex to the next

Number of values per vertex

```
glVertexAttribPointer(positionAttribIndex,2,GL_FLOAT,false,0,0);
```

Data type (the same type as the typed array)

**Offset**: how many bytes before the first vertex

# 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

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Start from vertex 0

Send 3 vertices

Send down the pipeline a set of vertices, each group of 3 consecutive forming a triangle

# 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 `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);
```

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`

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

20 bytes for vertex 0 | 20 bytes for vertex 1 | 20 bytes for vertex n

[pos0X, pos0Y, col0R, col0G, col0B, pos1X, pos1Y, col1R, col1G, col1B,..., posnX, posnY, colnR, colnR, colnB

Offset for color

Offset for position

```
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

$$\overbrace{[\mathbf{p_1}, \mathbf{p_2}, \boxed{\mathbf{p_0}}}^{T_0}, \overbrace{\boxed{\mathbf{p_0}}, \mathbf{p_2}, \mathbf{p_3}}^{T_1}, \overbrace{\boxed{\mathbf{p_0}}, \mathbf{p_3}, \mathbf{p_5}}^{T_2}, \overbrace{\boxed{\mathbf{p_0}}, \mathbf{p_5}, \mathbf{p_6}}^{T_4}, \overbrace{\mathbf{p_3}, \mathbf{p_4}, \mathbf{p_5}}^{T_3}]$$

- 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

$$[\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}, \mathbf{p_4}, \mathbf{p_5}, \mathbf{p_6}]$$

| 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,  // 3nd 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);
```

The table of indices

Indexed data

```
GLuint positionAttribIndex = 0;
float positions[] = {   0.0, 0.0,  // 1st vertex
                        0.5, 0.0,  // 2nd vertex
                        0.5, 0.5,  // 3nd 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);
```

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 parallell
- Has its own RAM memory level
  - "RAM CPU" vs "RAM GPU"

# A over-simplified scheme

# CPU vs GPU

# CPU vs GPU: how many proc. elements

# CPU vs GPU

# CPU vs GPU

- Transistors:
  >80%
  control + cache

→ flexibility

CPU

VS

- Transistors:
  ~80% ALU

→ power (FLOPS)

GPU

1.4 G transistors → 1 TeraFLOP

# CPU vs GPU

img by https://www.karlrupp.net/

# CPU vs GPU

CPU
Core 1
Core 2
Core 3
Core n
L1 data cache
L1 instruction cache
L2 cache
L3 or LLC cache
PCIe bus
System Agent
Memory Controller
DDR-4 Memory

GDDR-5 RAM
Memory Controller
PCIe x16 3.0 host interface
Processor Cluster 1
Processor Cluster 2
Processor Cluster 3
Processor Cluster N
Streaming Multiprocessor 1
Streaming Multiprocessor N
L1 cache
L2 cache
High speed hub
GDDR-5 Memory

# 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

**Accelerated Graphics Port**
Dedicated Bus dedicato between motherboard and Graphics Accelerator
AGP 1x 1.0: 66 x 2 = 133 MHz, bandwidth 533 MB/s
AGP 1x 2.0: 66 x 4 = 266 MHz, bandwidth 1066 MB/s
AGP 1x 3.0: 66 x 8 = 533 MHz, bandwidth 2133 MB/s

# Some history: 1998

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

**Accelerated Graphics Port**
Dedicated Bus dedicato between motherboard and Graphics Accelerator
AGP 1x 1.0: 66 x 2 = 133 MHz, bandwidth 533 MB/s
AGP 1x 2.0: 66 x 4 = 266 MHz, bandwidth 1066 MB/s
AGP 1x 3.0: 66 x 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

Vertex Shader. Make the chip of the graphics board programmable.

**Short** programs and **no-branching**

# 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 flexibllity, 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 exectutions
  - 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)

Chapter 1: Rendering Paradigms Introduction to Computer Graphics: a Practical Learning Approach 18

# 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

GLSL code

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

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

GLSL code

```
/* create a fragment shader */
std::string  fragment_shader_src = "\
            out vec4 color;
            void main(void)\
            {\
             color = vec4(0.0, 0.0, 1.0, 1.0);\
            }";
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);
```

# 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

One vertex-shader runs for each vertex

Qualifier for input values

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

        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

# 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: $\mathbf{p}, \mathbf{q}$ …

**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: $\boldsymbol{v}, \boldsymbol{u}$ …

# Operations on points and vectors

- point + vector returns a point $\mathbf{p_0} + \boldsymbol{v_0} = \mathbf{p_1}$
- point – point returns a vector $\mathbf{p_1} - \mathbf{p_0} = \boldsymbol{v_0}$
- vector + vector returns a vector $\boldsymbol{v_0} + \boldsymbol{v_1} = \boldsymbol{v_1} + \boldsymbol{v_0} = \boldsymbol{v_2}$
- scalar vector returns a vector $\mathbf{p_0} + \mathbf{2}\,\boldsymbol{v_2} = \mathbf{p_4}$

# Dot product (1/3)

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

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=1}^{n} a_1 b_1 + \cdots + a_n b_n$$

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

$$\theta = arcos\left(\frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\| \|\boldsymbol{b}\|}\right)$$

# 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

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

In 2 dimensions

# Dot product (3/3)

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

$$\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\|\|\boldsymbol{b}\| cos\, \boldsymbol{\theta}$$

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

parallel

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

antiparallel

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

# Cross product (1/3)

- The **cross** product is a binary operator betwen two vector 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 3x3 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

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

# Cross product (2/3)

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

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

- The cross product is anticommutative:

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

- It distributes over summation:

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

# Cross product (3/3)

- Norm of cross product:

$$\|\boldsymbol{a} \times \boldsymbol{b}\| = \|\boldsymbol{a}\|\|\boldsymbol{b}\|\boldsymbol{sin}(\boldsymbol{\theta})$$

- The norm of a cross product equals the area of the parallelogram $\mathbf{p}, \mathbf{p} + a, \mathbf{p} + a + b, \mathbf{p} + b$
- ..which means that for a triangle $\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}$:

$$\|(\mathbf{p_1} - \mathbf{p_0}) \times (\mathbf{p_2} - \mathbf{p_0})\| = \boldsymbol{2Area}(\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2})$$

# 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

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

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

# Translation

- Translation is typically used to *move* objects around

$$T_v(\mathbf{p}) = \mathbf{p} + \boldsymbol{v} = \begin{bmatrix} \mathrm{p}_{\boldsymbol{x}} \\ \mathrm{p}_{\boldsymbol{y}} \end{bmatrix} + \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \mathrm{p}_x + v_x \\ \mathrm{p}_y + v_y \end{bmatrix}$$

Translation vector

# Scaling

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

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

- 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..

# Rotation: Polar Coordinates

- Polar Coordinates: express points and vectors by the tuple $(\alpha, \rho)$
  - $\alpha$ is the angle with the $x$-axis
  - $\rho$ is the distance from the origin

# Rotation

A rotation by $\alpha$ of a point

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

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

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

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

# Rotation

A rotation by $\alpha$ of a point

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

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

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

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

# Rotation

A rotation by $\alpha$ of a point

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

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

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

# Shear

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

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

$$cot\theta = \frac{k \cdot p_y}{p_y}$$

# 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

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

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

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**

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ \mathbf{1} \end{bmatrix} \qquad \boldsymbol{v} = \begin{bmatrix} v_x \\ v_y \\ \mathbf{0} \end{bmatrix}$$

It's a point

It's a vector

It works for the point vector operands

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

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

$$\begin{bmatrix} \\ \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \\ \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \\ \\ \mathbf{0} \end{bmatrix}$$

# 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..

collinearity

Ratio of the distances

# 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

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

$$= \begin{bmatrix} \begin{bmatrix} a_{xx} & a_{xy} \\ a_{yx} & a_{yy} \end{bmatrix} \mathbf{p} + \boldsymbol{v} \\ 1 \end{bmatrix}$$

These will be 0

# Composition of Transformations (1/2 )

- Composing transformations means to apply a *sequence* of them
- Example: translate point $\mathbf{p} = \begin{bmatrix}1\\2\\1\end{bmatrix}$ by vector $v = \begin{bmatrix}2\\3\\0\end{bmatrix}$ and **then** scale the translated point by $2$ along both axes:

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

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

# Composition of Transformations (2/2 )

- Example: move the car from A to B
  - B is rotated by 45° clockwise
  - B is centered into $\begin{bmatrix}3\\2\\1\end{bmatrix}$

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

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

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

# 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

$T_{-c}$

$R_\alpha$

$T_c$

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

# Form of a rotation matrix around a point

- Let's apply the same transformations in Cartesian coordinates
  - That is, without extending the colum 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

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

$$\underbrace{\boldsymbol{R}\mathbf{p}}_{\text{rotation}} + \underbrace{(\mathbf{I} - \mathbf{R})\mathbf{c}}_{\text{translation}}$$

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

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

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

$$\begin{bmatrix} M^{-1} & -M^{-1}\mathbf{t} \\ 0 & 1 \end{bmatrix}$$

# 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:

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

- **Scaling:**

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

- **Shear:**

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

# Frames

- A **frame** consists of a point, called **origin** and a 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 $u = [0,1,0]^T$

# From a frame to *its* canonical frame

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

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

- In matrix form:

Axes of the frame as columns

Origin as translation

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

# Recall: matrix-vector product (1/2)

- “row x column”

# Recall: Matrix-vector product

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

# Frames as matrices

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

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

- Remember:
  - If $\mathbf{p}_\mathrm{F}$ are coordinates of point $\mathbf{p}$ expressed in the frame $F$, $F\mathbf{p}_\mathrm{F}$ are the coordinates of $\mathbf{p}$ expressed in the canonical frame
  - If $\mathbf{p}_\mathrm{C}$ are coordinates of point $\mathbf{p}$ expressed in the canonical frame $C$, $F^{-1}\mathbf{p}_\mathrm{C}$ 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 F0 to C

It transforms C into F0 !

# 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 $F_0$

$$(\mathbf{Mp})_0 = \mathbf{F}_0^{-1}\mathbf{Mp} = \mathbf{F}_1^{-1}\mathbf{p} \;\Rightarrow\; \mathbf{F}_1 = \mathbf{M}^{-1}\mathbf{F}_0$$

Coordinates of $M\mathbf{p}$ in $F_0$

Coordinates of $\mathbf{p}$ 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:

$$\mathbf{p} = F_0\mathbf{p_0} = F_1\mathbf{p}_1$$

$$\Downarrow$$

$$\mathbf{p}_1 = F_1^{-1}F_0\mathbf{p_0}$$

coordinates in the canonical frame

# 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 4x4 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 in 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 $\boldsymbol{y}$
  - pitch: rotation around axis $\boldsymbol{x}$
  - roll: rotation around axis $\boldsymbol{z}$

# Euler Angles Rotations

- demo

$\alpha = 0$, $\beta = 0$, $\gamma = 0$

$\alpha = \pi/4$, $\beta = 0$, $\gamma = 0$

$\alpha = 0$, $\beta = \pi/4$, $\gamma = 0$

$\alpha = 0$, $\beta = 0$, $\gamma = \pi/6$

# Frames of the gimbal

-1s are inverted in the book

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

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

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

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

# Gimbal lock

- $\alpha$, $\beta$, and $\gamma$ define a *surjective* function to the space of rotation
  - It means that *every* rotation can be obtained
- However, the function is not *iniective*
  - it means that more than a combination of $\alpha$, $\beta$, and $\gamma$ define the same rotation
- For example: if $\beta = 90°$ then all values of $\alpha$ and $\gamma$ 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 $\boldsymbol{a} \neq \boldsymbol{z}_r$

Normalization is not

Note: if a is too near to be collinear with z_r numberical 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 $\boldsymbol{a} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$

# Rotation around a generic axis (2/2)

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

$$\mathbf{p}' = \begin{bmatrix} \boldsymbol{x}_F \ \boldsymbol{y}_F \ \boldsymbol{r} & \begin{matrix} O_{Fx} \\ O_{Fy} \\ O_{Fz} \end{matrix} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha \\ \sin\alpha \\ 0 \\ 1 \end{bmatrix} = \cos\alpha \ \boldsymbol{x}_F + \sin\alpha \ \boldsymbol{y}_F + 0 \ \boldsymbol{r} + \mathbf{O}_F$$

$$\boxed{\begin{aligned} \mathbf{O}_F &= (\boldsymbol{p} \ \boldsymbol{r}) \ \boldsymbol{r} \\ \boldsymbol{x}_F &= \mathbf{p} - \mathbf{O}_F = \mathbf{p} - (\boldsymbol{p} \ \cdot \boldsymbol{r}) \ \boldsymbol{r} \\ \boldsymbol{y}_F &= \boldsymbol{r} \times \boldsymbol{x}_F = \boldsymbol{r} \times (\mathbf{p} - \mathbf{O}_F) = \end{aligned}}$$

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

**Rodriguez** formula:

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

# Exercise

- Find a **translation** matrix that transforms p0 into p1
  - How many exist?
- Find a **scaling** matrix that transforms p0 into p1
  - How many exits?
- Find a **rototranslation matrix** that transforms p0 into p1
  - How many exists?

# Exercise

A sinistra è raffigurato l'oggetto PesceRosso nel suo sistema di riferiento (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 $\mathbf{M}^{-1}$

M =
$$\begin{pmatrix} \square & \square & 0 & \square \\ \square & \square & 0 & \square \\ 0 & 0 & 1 & 0 \\ \square & \square & \square & \square \end{pmatrix}$$

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)**

$$\square \cdot \begin{pmatrix} \square \\ \square \\ \square \\ \square \end{pmatrix}$$

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!

$$\mathbf{p} = p_x + \boldsymbol{i}\, p_y$$
$$\mathbf{r} = cos\alpha + \boldsymbol{i}\, sin(\alpha)$$

$$(cos\alpha + \boldsymbol{i}\, sin\alpha)(p_x + \boldsymbol{i}\, p_y) =$$
$$cos\alpha\, p_x + \boldsymbol{i}^2 sin\alpha\, p_y + i(sin\alpha\, p_x + cos\alpha\, p_y) =$$
$$\boxed{\boldsymbol{i}^2 = -1}$$
$$cos\alpha\, p_x - sin\alpha\, p_y + \boldsymbol{i}\,(sin\alpha\, p_x + cos\alpha\, p_y)$$

# Quaternions (definition) (1/2)

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

$$\boldsymbol{q} = w + \boldsymbol{i}\,x + \boldsymbol{j}\,y + \boldsymbol{k}\,z$$

$$\boldsymbol{ii} = \boldsymbol{jj} = \boldsymbol{kk} = \boldsymbol{ijk} = -1$$

$$\begin{aligned} \boldsymbol{ij} &= \boldsymbol{k}, & \boldsymbol{ji} &= -\boldsymbol{k} \\ \boldsymbol{jk} &= \boldsymbol{i}, & \boldsymbol{kj} &= -\boldsymbol{i} \\ \boldsymbol{ki} &= \boldsymbol{j}, & \boldsymbol{ik} &= -\boldsymbol{j} \end{aligned}$$

mnemonic rule:
clockwise-> positive
counterCW-> negative

- It is usually written as:

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

# Quaternions (operations) (2/2)

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

- sum of two quaternions: $\mathbf{q_1} + \mathbf{q_2} = (s_1 + s_2, \boldsymbol{v_1} + \boldsymbol{v_2})$
- product of two quaternions: $\mathbf{q_1 q_2} = (s_1 s_2 - \boldsymbol{v_1 v_2}, s_1 \boldsymbol{v_2} + s_2 \boldsymbol{v_1} + \boldsymbol{v_1} \times \boldsymbol{v_2})$
- Conjugate of a quaternion: $\overline{\mathbf{q}} = (s, -\boldsymbol{v})$
- norm of a quaternion: $|\mathbf{q}| = \sqrt{\overline{\mathbf{q}}\mathbf{q}} = \sqrt{s^2 + \boldsymbol{v}^2}$
- Inverse of a quaternion: $\mathbf{q}^{-1} = \frac{\overline{\mathbf{q}}}{|\mathbf{q}|^2}$

# Quaternions

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

Are we sure it is a unit quaternion?

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

Parameters of axis-angle rotations

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

represents the rotation around the vector $\boldsymbol{v}$ by $\theta$

- 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

$$\mathbf{p} = (0, \boldsymbol{v_p})$$

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

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

where $\mathbf{p}'$ equals to $\mathbf{p}$ rotated around $\boldsymbol{v}$ by $\theta$

# Conversions

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

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

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:

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

Is the same as:

$$\mathbf{q}\ \mathbf{p}\overline{\mathbf{q}}$$

# Reasons for quaternions

- Fast to multiply
  - less floating point then 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

Same length intervals subtend different angles

# Interpolating rotations

- For constant speed: Spherical Linear Interpolation between vectors $\boldsymbol{e_o}$ *and* $\boldsymbol{e_1}$
  1. Build an orthonormal basis from $\boldsymbol{e_0}$ and $\boldsymbol{e_1}$
  2. Interpolate the angle between the vectors

$$\boldsymbol{e}_1^p = \frac{\boldsymbol{e}_1 - (\boldsymbol{e}_0 \cdot \boldsymbol{e}_1)\boldsymbol{e}_0}{|\boldsymbol{e}_1 - (\boldsymbol{e}_0 \cdot \boldsymbol{e}_1)\boldsymbol{e}_0|}$$

$$\boldsymbol{e}(t) = \cos(t\psi)\,\boldsymbol{e}_0 + \sin(t\psi)\,\boldsymbol{e}_1^p$$

# Slerp: Spherical Linear Interpolation (1/3)

- $R = \{r_0, r_1, 0\}$ is a frame.
- The vector $\boldsymbol{r}(t)$ 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)

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

$$\psi = \operatorname{acos}(r_0 \cdot r_1) = \alpha(t) + \beta(t)$$

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

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

$$\alpha(t) = t\,\psi \qquad \beta(t) = (1-t)\,\psi$$

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

# Slerp: Spherical Linear Interpolation (3/3)

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

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

$$\psi = \operatorname{acos}(q_0 \cdot q_1)$$
$$\mathbf{q}(t) = \frac{\sin t\psi}{\sin\psi}\mathbf{q}_0 + \frac{\sin(1-t)\psi}{\sin\psi}\mathbf{q}_1$$

- 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

Screen space

0

3

World space

1

View space

2

Clip/NDC 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

How to compute $\boldsymbol{x}$?

$$VRF = \begin{bmatrix} x_x & y_x & z_x & o_x \\ x_y & y_y & z_y & o_y \\ x_z & y_z & z_z & o_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Building the View Reference Frame

- The missing axis $x$ is obtained by means of vector product. Be careful:
  1. normalize *view direction* and *up*
  2. 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:
     1. Compute $\boldsymbol{x} = \boldsymbol{y}' \times \boldsymbol{z}$ , which is orthogonal to $\boldsymbol{z}$
     2. Compute $\boldsymbol{y} = \boldsymbol{x} \times \boldsymbol{x}$
- 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 ration of their side is the same for all sides:

$$p'_y : d = p_y : -p_z \Longrightarrow p'_y = -\frac{p_y}{p_z/d} \qquad p'_x = -\frac{p_x}{p_z/d}$$

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:

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

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

# Homogeneous coordinates:example

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

Canonical form

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

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

**point**

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

The unique homogeneous coordinates of **vector** with Cartesian coordinates

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

**vector**

# Equivalent homogenous coordinates

$$\begin{bmatrix} p'_x \\ p'_y \\ -d \\ 1 \end{bmatrix} = \boldsymbol{\lambda} \begin{bmatrix} p'_x \\ p'_y \\ -d \\ 1 \end{bmatrix} = \boldsymbol{\lambda_1} \begin{bmatrix} p'_x \\ p'_y \\ -d \\ 1 \end{bmatrix} = \boldsymbol{\lambda_2} \begin{bmatrix} p'_x \\ p'_y \\ -d \\ 1 \end{bmatrix} = \boldsymbol{\lambda_3} \begin{bmatrix} p'_x \\ p'_y \\ -d \\ 1 \end{bmatrix}$$

# Perspective projection

- In homogenous coordinates:

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

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

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

# Perspective projection

- In matrix form

Canonical form

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

Is it affine?

Is it invertible?

Perspective division

# Perspective projection

- Equivalent matrix form

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

- 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 to 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

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

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

View from above

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

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

# Window coordinates in view space

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

# The **N**ormalized **D**evice **C**oordinates space

- The NDC is the space  where primitives (partially) outside the view colume 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

- Perpective 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 metaphore, we will have no lenses and our retina should be as big as the window (not a great metaphore)
- In terms of perspetive projection, is the same as if $d \to \infty$

# 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

# NDC to viewport (2/2)

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

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

# Properties preserved by transformations

A cheat sheet to remeber what transformations preserve what poperties

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 find the **lens** which will distribute the light all over the **retina**
- The retina contain receptors which are activated by light on the basis of its frequency: **rods** and **cones**

Hardware for light reception

Incoming light

# How do we see light?

Interpretation

Hardware for light reception

Transmission

*Source: Wikipedia*

- 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:
    - **R**ed (64%)  **L**ong, peak in 560 nm – 580 nm
    - **G**reen (32%) **M**iddle:  peak in 530 nm – 540 nm
    - **B**lue (2%) **S**hort:  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 ob

- Obje
- T

  - Background/surrou 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 mathing 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**.

Subject manipulates the mixture of RGB until the color of the two halves matches

Monochromatic light (spectral color)

online emulator

# 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
- **N**ational **P**hysical **L**aboratory (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

$$White = r\, m_r\, R \;\oplus\; g\, m_g\, G \;\oplus\; b\, m_b\, B$$

$$so\ that:\quad r\, m_r = g\, m_g = b\, m_b$$

Handles manoeuvred by the observers

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

- They were: $m_r : m_g : m_b = 1 : 4.39 : 0.0048$

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 **thrichromatic coeffients** $\alpha, \beta, and\ \gamma$ are defined as

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

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

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

# Luminance for a wavelength

- Let $C_\lambda$ be the spectral color for the wavelength $\lambda$

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

- The luminance of $C_\lambda$ 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

*spectral color*

*non spectral color*

# 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

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

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

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

From the original papers

# 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

**Alychne**: 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

Youtube - @jselan

# CIE xy chromaticity diagram

- Normalizing XYZ we obtain

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

- *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** visibile 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 tristimulous 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)

[RGB Mixer - Computer Science Field Guide (csfieldguide.org.nz)]

# 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 obvoius one or except if you spent ages using it
  - Try finding RGB for brown or pink

# Color Spaces: HSV

- **H**ue, **S**aturation and **V**alue may be though 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](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

This is wrong in the book!
Δ insted of V

$$M = \max\{R, G, B\}$$
$$m = \min\{R, G, B\}$$
$$\Delta = M - m$$

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

# Color Spaces: HSL

- **H**ue **S**aturation **L**ightness 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)]

# Color Spaces: HSL & HSL comparison

**H**ue **S**aturation **L**ightness

**H**ue **S**aturation **V**alue

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

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

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

$$\Delta = M - m$$

from RGB to

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

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

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

$$V = M$$

# 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* ligthness, 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
- $X_n, Y_n, and\ Z_n$ are illuminant dependent normalization factors

$$
\begin{aligned}
L^* &= 116\, f(X/X_n) - 16 \\
a^* &= 500\, f(X/X_n) - f(Y/Y_n) \\
b^* &= 200\, f(Y/Y_n) - f(Z/Z_n)
\end{aligned}
$$

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

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

http://davidjohnstone.net/pages/lch-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:

$$\Delta E^*_{ab} = \sqrt{(L^*_2 - L^*_1)^2 + (a^*_2 - a^*_1)^2 + (b^*_2 - b^*_1)^2}$$

- ΔE < 1 : not perceptible, 1 < ΔE < 2 close observation needed to perceive the difference, 2 < ΔE < 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?

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

- 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?

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

- $\frac{I_{max}}{I_{min}}$ = **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
  - **H**igh **D**ynamic **R**ange 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, .., n

Value I want to display

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

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

# Gamma correction (the historical reason)

- **C**athode-**R**ay **T**ube 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 **C**atode **R**ay **T**ube 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 ➡ $P \cdot V^{-1} \cdot \underbrace{M_9 \cdot M_4 \cdot M_3}_{\text{Model matrix}}$

# 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)

| | Matrix stack | |
|---|---|---|
| *Op* | *current* | |
| | | $\varnothing$ |
| set $P\ V^{-1}$ | $P\ V^{-1}$ | $\varnothing$ |
| **push** | $P\ V^{-1}$ | $P\ V^{-1}$ |
| ***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 w1 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

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

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

# 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 $\mathbf{p}_0'$ and $\mathbf{p}_1'$ 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

Before transforming the whole scene in view space, rotate it

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

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

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

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

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

$$\underbrace{(\mathbf{o} - \mathbf{c})(\mathbf{o} - \mathbf{c}) - \boldsymbol{r}^2}_{C} + t^2\mathbf{d}^2 + t\,\underbrace{2\boldsymbol{d}(\mathbf{o} - \mathbf{c})}_{B} = \mathbf{0}$$

A

$$\Rightarrow \quad At^2 + Bt + C = 0$$

$$t = \frac{-\mathrm{B} \pm \sqrt{(\mathrm{B}^2 - 4\mathrm{AC})}}{2\mathrm{A}}$$

# 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 envents

# 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

Position in pixels

$$\mathbf{p}_x = l + \frac{clientX}{sizeX} * (r - l)$$

$$\mathbf{p}_y = b + \frac{(sizeY - clientY)}{sizeY} * (t - b)$$

size in pixels

$$\begin{bmatrix} \mathbf{p}_x \\ \mathbf{p}_y \\ -n \\ 0 \end{bmatrix}$$

Vector in 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

- **H**idden **S**urface **R**emoval (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 to closest
  - Every primitive will write into the frame buffer
  - The occluded one will be overdrawn by closest primtives
- 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 drawn
  - 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? It not, no need to sort..
  3. Do their projections on the y axis overlap? It not, no need to sort..
  4. Hang on….

# Depth Sort: the separating plane

- A **separating plane** between to 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? It not, no need to sort..
  3. Do their projections on the y axis overlap? It 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 algorith 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 |

+

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| .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 | 1.0 | 1.0 |
| .5 | .5 | .5 | .5 | .5 | 1.0 | 1.0 | 1.0 |
| .5 | .5 | .5 | .5 | 1.0 | 1.0 | 1.0 | 1.0 |
| .5 | .5 | .5 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| .5 | .5 | 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 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

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

# Where in the pipeline?

# Where in the pipeline?

Color (RGB) for each fragment

Depth (z value) for each fragment

Depth per vertex is computed in the vertex shader

Depth value is interpolated along with the other attributes during rasterization

Pseudo code of depth test

```
if  ( d <= DepthBuffer[x,y] ) {
        ScreenBuffer[x,y] = (r,g,b) ;
        DepthBuffer[x,y]  = d ;
}
```

# Z Coordinates of a point in the View Frustum

This range can be changed by glDepthRange(min,max)

| Object space | View space | NDC space | Depth buffer |
|---|---|---|---|
| | $near < z < far$ | $-1 < z < +1$ | $0.0 < z < 1.0$ |

# 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-fighthing

- **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

```
                GL_NEVER
                GL_EQUAL
                GL_GREATER
                GL_LEQUAL
glDepthFunc(GL_LESS     );
                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 that 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 recangular 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..

glStencilOp( GLenum sfail, GLenum dpfail, GLenum dppass);

What to do if:

- sfail: The stencil test fails
- dpfail: The depth test fails
- dppass: Both pass

# 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 $\alpha$ value

- The $\alpha$ value is the fourth component of the RGBa color and its says how opaque the color is:
  - $\alpha = 0$ -> fully transparent
  - $\alpha = 1$ -> 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

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

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

Example

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

learnopengl.com

# 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 thealpha 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*

# 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

- **S**uper **S**ampling **A**nti **A**liasing (**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

- **M**ulti **S**ampling **A**nti **A**liasing (**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" form 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 $\mathbf{p}_{0_{xy}}, \mathbf{p}_{1_{xy}}, \mathbf{p}_{2_{xy}}$:
  - If positive -> front facing
  - If negative -> back facing, discard

# Where it’s done

when the coordinates have already been transformed in NDC space, before rasterizing them

# 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

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

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

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”

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

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

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

$\mathbf{p}_0$ 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

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

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

$$(\mathbf{p_0} + \boldsymbol{v}\, t)_x = x_{min} \Rightarrow t = \frac{x_{min} - \mathbf{p0}_x}{\boldsymbol{v}_x}$$

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

$intersects \Leftrightarrow max(t_0, t_1) < min(t_2, t_3)$

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

# Clipping segments (3/ 4)

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

$$
\begin{aligned}
t_{min} &= \max(0, \text{largest entry value}) \\
t_{max} &= \min(1, \text{smallest exit value})
\end{aligned}
$$

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

$intersects \Leftrightarrow max(t_0, t_1) < min(t_2, t_3)$

$t_0$ is an entry if $\boldsymbol{v}_x > 0$, exit otherwise
$t_1$ is an entry if $\boldsymbol{v}_x < 0$, exit otherwise
$t_2$ is an entry if $\boldsymbol{v}_y > 0$, exit otherwise
$t_3$ is an entry if $\boldsymbol{v}_y < 0$, 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 (**pi**cture **x el**ement)

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | 136 | 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 | 202 | 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 | 191 | 122 | 81 | 54 | 50 | 49 | 44 | 43 | 49 | 44 | 44 | 55 | 56 | 91 | 148 | 128 | 69 | 151 | 164 |
| 193 | 143 | 92 | 67 | 50 | 50 | 53 | 60 | 52 | 48 | 43 | 45 | 61 | 57 | 77 | 143 | 137 | 75 | 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 Dept**
  - e.g: RGB 8 **s**
  - 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: resX · resY · imageDepth
  - a 240×480 “true color” image
    memory occupation = 240 · 480 · 3 bytes = 345.600 bytes
  - 4000×2000 (8 MegaPixel) “true color”: 4000 · 2000 · 3 = 24 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)

## Volumetric Models

# Types of 3D digital models

## Point based

## segments

Hey! Those are not 3 dimensional !1!!

# 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 $S: \mathbb{R}^2 \rightarrow \mathbb{R}^3$

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

# surfaces

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

$S(x, y, z)$=.... ??

# Volumetric models

- Voxel based: the 3D version of the raster images
  - **Pi**x-el = picture element
  - **Vo**x-el = volume element

- **C**onstructive **S**olid **G**eometry
  - 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

1994

70.000 tri

# Manageable resolution along the years

1997

1.200.000 tri

# Manageable resolution along the years

**2002**

2.000.000.000 **tri**

# Performance vs quality

p e r f o r m a n c e

q u a l i t y

# 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?

A n-side polygon

*(n-2)* triangles

Triangulation of a polygon:

Not so obvious in 3D

# 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 vertices
- 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

Are they borders or are they holes?

# 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
- Ofter we want some “value” (or signal) to be associated within 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?

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

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

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

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 <u>always</u> what we want?

# Normal per-vertex

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

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

Faes sharing vertex $i$

# Normal per-vertex

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

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

widely used, it assumes that the tessellation is fairly regular.
Many thin faces contribute more that one large face..

# Normal per-vertex

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

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

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

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

Faes 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»

# 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...

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

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

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

# 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

Vertex duplication

Normals stored in 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*:
  - $f(\mathbf{p}) < 0$ <==> $\mathbf{p}$ inside
  - $f(\mathbf{p}) > 0$ <==> $\mathbf{p}$ outside
  - $f(\mathbf{p}) = 0$ <==> $\mathbf{p}$ on the surface

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

# 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 **p** be a point on the surface: $f$ (**p**) = 0
- What is the normal of the surface at $\mathbf{p}$?
  - That is: toward which direction **d** do we have to move away from **p** to maximize the value of $f$ (**p** + $k$ **d**)
- Answer: take the gradient of $f$ (**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 folling operators
  - complement: - $f_A$
  - intersection: max( $f_A$ , $f_B$ )
  - union: min( $f_A$ , $f_B$ )
  - difference: max( $f_A$ , - $f_B$ )

min(a,b)

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 $x_i$

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

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

# 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 $\alpha$

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

support

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

# Functions to use

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

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

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

- For example:

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

Discontinuous derivative
Is bad

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

Not a good function.

# Implicit Surfaces: metaballs

Support radius R

$\alpha = 0.2$

$\alpha = 0.4$

$\alpha = 0.6$

$\alpha = 0.8$

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 buldge

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

Region with sparse points

Implicit surface

# 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?

# 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

# Parametric curve

$$f : A \to B$$
$$A \subseteq \mathbb{R}$$
$$B \subseteq \mathbb{R}^3$$

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

# Parametric segment

$f: A \rightarrow B$

$A \subseteq \mathbb{R}$

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

$$f(t) = \mathbf{p}_0 \cdot (1 - \mathrm{t}) + \mathbf{p}_1 \cdot \mathrm{t} = (\mathbf{p}_1 - \mathbf{p}_0) \cdot t + \mathbf{p}_0 \qquad t \in [0,1]$$

1D 3D

# 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 slideS...

- Spline: piecewise **polynomial** curve with degree > 1
- It is defined on a sequence of **control points**
- It is said to be **interpolant** is it passes through the control points

interpolant

# Spline

- Spline: piecewise **polynomial curve** with degree > 1
- It is defined on a sequence of **control points**
- It is said to be **interpolant** is it passes through the control points
- It is said to be **approximant** if it passes nearby

# Spline formulation

Blending functions

$$f(x) = \sum_{i=0}^{n} \mathbf{p}_i B_i(t) \qquad 0 \le t \le 1$$

Polynomes on t

Remember the segment?

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

$$B_0(t) = 1 - t$$
$$B_1(t) = t$$

# Bèzier curves

- A widely used type of curve which uses **Bernstein polynomials** as blending functions

$$f(t) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(t) \qquad 0 \le t \le 1$$

Degree n, control points n+1

Binomial coefficient

$$\binom{n}{i} = \frac{n!}{i!\,(n-i)!}$$

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Bernstein polynome

$$B_{i,n}(t) = 0, i < 0 \; or \; i > n$$

Remember the segment?

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

$B_{0,1}(t) = 1 - t$

$B_{1,1}(t) = t$

# 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 <u>always</u> 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 <u>always</u> 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

$$B_{i,n}(t) = B_{i-1,n-1}(t)\,t + B_{i,n-1}(t)(1-t)$$
$$B_{i,n}(t) = 0, i < 0 \; or \; i > n$$
$$B_{0,1}(t) = 1 - t, \qquad B_{1,1}(t) = t$$

$$f(\mathbf{t}) = \mathbf{p_0} B_{0,2}(t) + \mathbf{p_1} B_{1,2}(t) + \mathbf{p_2} B_{2,2}(t)$$

$$\cancel{B_{-1,1}(t)t} + B_{0,1}(t)\,(1-t)$$

$$B_{0,1}(t)t + B_{1,1}(t)\,(1-t)$$

$$B_{1,1}(t)t + \cancel{B_{2,1}(t)}\,(1-t)$$

$$f(\mathbf{t}) = \mathbf{p_0} B_{0,1}(t)(1-t) + \mathbf{p_1} B_{1,1}(t)(1-t) + \mathbf{p_1} B_{0,1}(t)\,t + \mathbf{p_2} B_{1,1}(t)t$$

$$\boxed{f(\mathbf{t})} = \boxed{[\mathbf{p_0} B_{0,1}(t) + \mathbf{p_1} B_{1,1}(t)]}(1-t) + \boxed{[\mathbf{p_1} B_{0,1}(t) + \mathbf{p_2} B_{1,1}(t)]}\;t$$

# Bèzier curves of degree 2

- n=2 → 3 control points p0,p1,p2
- *De Casteljau* algorithm to find the point f(t)
  - Find the point $\mathbf{p}_{01}(t)$ of the linear Bèzier curve between p0 and p1
  - Find the point $\mathbf{p}_{12}(t)$ of the linear Bèzier curve between p1 and p2
  - Find the point $p(t)$ 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 Bèzier curves: weigths

$$f(\mathrm{t}) = (1-\mathrm{t})^2\ \mathbf{p}_0 + 2\,\mathrm{t}\,(1-\mathrm{t})\ \mathbf{p}_1 + \mathrm{t}^2\ \mathbf{p}_2$$

# Cubic Bézier curves: weigths

$$
\begin{aligned}
f(t) \quad = \quad & (1-t)^3\, \mathbf{p}_0 \\
& + 3\,t\,(1-t)^2\, \mathbf{p}_1 \\
& + 3\,t^2\,(1-t)\, \mathbf{p}_2 \\
& + t^3\, \mathbf{p}_3
\end{aligned}
$$

# Cubic Bézier curves: examples

# Tangent to a parametric curve

- The tangent vector to a curve is found as:

$$\lim_{dt \to 0} \frac{f(\mathrm{t} + \mathrm{dt}) - f(\mathrm{t})}{dt}$$

- 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

$$\frac{d}{dt}f(t) = \sum_{i=0}^{n} \mathbf{p_i} \frac{d}{dt} B_{i,n}(t)$$

- The derivative of a Bernstein polynome is found as:

$$\frac{d}{dt}B_{i,n}(t) = n\left(B_{i-1,n-1}(t) - B_{i,n-1}(t)\right)$$

- Which ultimately gives:

$$\frac{d}{dt}f(t) = n\sum_{i=0}^{n-1} \mathbf{p_{i+1}}\, B_{i-1,n-1}(t) - \mathbf{p_i}\, B_{i,n-1}(t)$$

# 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 $C^{\infty}$
- 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

$$f : \mathrm{A} \rightarrow \mathrm{B}$$

$$\mathrm{A} = [0, 2\pi] \times [0,1]$$

$$f\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} r \ \cos s \\ h \cdot t \\ r \ \sin s \end{pmatrix}$$

# Normal of a parametric surface

$$v_2 = \frac{\partial f}{\partial t}(p) \qquad v_1 = \frac{\partial f}{\partial s}(p)$$

$$\boldsymbol{n} = \frac{\boldsymbol{v_1} \times \boldsymbol{v_2}}{[\![\boldsymbol{v_1} \times \boldsymbol{v_2}]\!]}$$

# Bezier patch

Example: cubic Bezier patch
n=3
(3+1)*(3+1)=16 control points

- A **Bezier patch** is defined as

$$f(s,t) = \sum_{i=0}^{n} \sum_{j=0}^{n} \mathbf{p}_{i,j} B_{i,n}(t) B_{j,n}(t) \quad 0 \le t \le 1$$

where $\mathbf{p}_{i,j}$ $i,j = 0 \ldots n$ is a grid of n+1 control points

# Bezier patch

- Note that:

$$f(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{n} \mathbf{p}_{i,j} B_{i,n}(s) B_{j,n}(t) \quad 0 \le t \le 1$$

$$=$$

$$f(s,t) = \sum_{i=0}^{n} B_{i,n}(s) \sum_{j=0}^{n} \mathbf{p}_{i,j} B_{j,n}(t) \quad 0 \le t \le 1$$

$$=$$

$$f(s,t) = \sum_{j=0}^{n} B_{j,n}(t) \sum_{i=0}^{n} \mathbf{p}_{i,j} B_{i,n}(s) \quad 0 \le t \le 1$$

$p_{0,2}$
$p_{1,3}$
$p_{1,2}$
$p_{2,3}$
$p_{0,0}$
$p_{0,1}$
$p_{2,2}$
$p_{1,1}$
$p_{1,0}$
$p_{3,3}$
$p_{2,1}$
$p_{3,2}$
$p_{2,0}$
$p_{3,1}$
$p_{3,0}$

$f(s,t)$
s
t
t
t
t

$f(s,t)$

# 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:

$$\frac{d}{ds}f(s,t) = \sum_{j=0}^{n} B_{j,n}(t) \quad \underbrace{\frac{d}{ds}\sum_{i=0}^{n} \mathbf{p}_{i,j} B_{i,n}(s)} \quad 0 \le s,t \le 1$$

Derivative of the Bezier curve for points:
$\mathbf{p}_{i,0}, \mathbf{p}_{2,0}, \dots, \mathbf{p}_{n,0}$

- Same for $\frac{d}{dt}f(s,t)$

# Normal

- Knowing the derivatives along s and t, the normal is found as their cross product

$$\mathrm{N}(s,t) = normalize\left(\frac{d}{ds}f(s,t) \times \frac{d}{dt}f(s,t)\right)$$

# 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), … (0.0,0.1), (0.1,0.1)… , (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: $sizeX \cdot sizeX \cdot sizeZ$
  - 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

Input taken by some additive 3D printers

3D printer

Printed objects

# Creating Voxelized Volumes

Voxelization

Triangle

2-manifold

and cl

zed volume

it × 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)

C and D are empty

Cube size halves at each level

Example in 2D, each quad is partitioned in four quads (**quad-tree**)

A
B
C
D
E

A

B
C
D
E

0
0 1
1 1
1
1 0
1 1
0 1
0 1
1
1
1
0 1
0 0
1
1
1
1 1 1 1 1 0
0 0 0 0 0 0

A
B
C
D
E

A

B
C
D
E

# 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

- **C**omputerized **T**omography 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!

Plane for cresting the raster image

Values for the pixels are found by interpolating those from the neighbors cells/grid corners (more later..)

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](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

# Example: Computational Fluid Dynamics

Source: pointwise.com

# Example: Computational Fluid Dynamics

Source: [Ruiz-Girones Roca Sarrate 2012]

# 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 **p** (*x,y,z*)
  - “normal” **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
- **S**tructure **f**rom **M**otion (SfM): a point cloud can be produced just from a set of normal photographs (**C**omputer **V**ision)
- Points also have color (from the images)

# Point clouds from real world

Material properties may not be ideal for active scanning

Very typical of SfM. Only points with distinctive color features are created

Very typical of SfM.

3D active scanning:
A single «take» usually does not captre the whole surface. They mush be re-aligned

Berger et al. / A Survey of Surface Reconstruction from Point Clouds

# Rendering point cloud: surfel

- **Surfel** (**surf**ace **el**ement): 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

img by Mario Botsch, 2004

This specific model is not obtained as a point cloud fromthe 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

(old) Images by (old) QSplat software

# 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 bythe 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

$$\exists t: \; f(\mathrm{O})(1-t) + f(\mathrm{O})t = \alpha$$

$$\Rightarrow t = \frac{\alpha - f(\mathrm{O})}{f(\mathrm{O}) - f(\mathrm{O})}$$

$$\mathbf{p}(\mathrm{O})\,(1-t) + \mathbf{p}(\mathrm{O})t = \mathbf{p}$$

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

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**LookUpTable**

**0 : nil**
**1 : {e0,e3,e8}**
**2 : ….**
**3 : ….**
**….**
**…**
**255: nil**

aribtary slice

direct volume rendering

Marching cube meshes

direct volume rendering

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 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!!
things we will not see in this course

# 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 it 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 |
|---|---|---|
| $\Phi$ | 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 |
|---|---|---|
| $\Phi$ | 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 |
|---|---|---|
| $\Phi$ | Radiant flux | Watt (J/s) |
| $E = \frac{d\Phi}{dA}$ | Irradiance/exitance | $Watt/m^2$ |
| $I = \frac{d\Phi}{d\omega}$ | Directional intensity | $Watt/s_r$ |

Directional Intensity: amount of flux per unit **solid angle** leaving from a point towards a direction $\omega$
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 if $S/r^2 = 4\pi$ steradians

Wikimedia commons

# Some quantities: intensity (2/2)

| symbol | name | unit |
|---|---|---|
| $\Phi$ | Radiant flux | Watt (J/s) |
| $E = \frac{d\Phi}{dA}$ | Irradiance/exitance | $Watt/m^2$ |
| $I = \frac{d\Phi}{d\omega}$ | Directional intensity | $Watt/s_r$ |

Directional Intensity: amount of flux per unit **solid angle** leaving from a point towards a direction $\omega$

The solid angle is expressed in **steradians**

E.g. the intensity $\Phi$ of a light emitted uniformly along all directions by a point light source is $\Phi/4\pi$

# Some quantities: radiance

| symbol | Name | unit |
|---|---|---|
| $\Phi$ | 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→ obj1 → obj2 → ··· → 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
- …

Way 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 indicent 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

$\Omega$ All possible directions (that is, unit vectors in the hemisphere)

# The radiance equation for local lighting

Emissive component

$$L_o(x,\vec{\omega}_r) = \boxed{L_e(x,\vec{\omega}_r)} + L_r(x,\vec{\omega}_r)$$

$$L_r(x,\vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x,\vec{\omega}_i,\vec{\omega}_r) L_i(x,\vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

# 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

environment

$$L_o(x,\vec{\omega}_r) = L_e(x,\vec{\omega}_r) + L_r(x,\vec{\omega}_r)$$

$$L_r(x,\vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x,\vec{\omega}_i,\vec{\omega}_r) L_i(x,\vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

# Incident light

- For each position $x$
  $L_i(x, \ldots)$ represents
  the distribution of incident light

$L_i(x, \vec{\omega}_i)$ = how much light arrives to $x$ 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

BRDF

$$L_o(x,\vec{\omega}_r) = L_e(x,\vec{\omega}_r) + L_r(x,\vec{\omega}_r)$$

$$L_r(x,\vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x,\vec{\omega}_i,\vec{\omega}_r) L_i(x,\vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

# The radiance equation for local lighting: BRDF

$f_r(x, \vec{\omega}_i, \vec{\omega}_r)$ Function representing the fraction of incident light from $\omega_i$ 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)

$f_r$ =

photon =

$f_r(\vec{\omega}_i, \vec{\omega}_r)$ = On 100 arriving at
from direction $\vec{\omega}_i$,
how many will bounce along $\omega_r$?

$f_r$: $\mathbb{R}^4 \rightarrow \mathbb{R}$

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 $\omega_i$ arrive L lumens (N "tennis balls"), how many hit the neighborhood of $x$?

# Cosine law

- From direction $\omega_i$ arrive L lumens (N "tennis balls"), how many hit the neighborhood of $x$?

# Cosine law

- From direction $\omega_i$ arrive L lumens (N "tennis balls"), how many hit the neighborhood of $x$?

# Cosine law

- From direction $\omega_i$ 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 $\omega_i$ 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 $\omega_i$ 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

$\max(\cos \boldsymbol{\alpha}, \mathbf{0})$

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

$$L_r(x, \vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) \boxed{(\vec{\omega}_i \cdot \vec{n})} d\vec{\omega}_i$$

# 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 $x$"

$$L_r(x,\vec{\omega}_r) = \int_{\vec{\omega}_i \in \Omega} \overbrace{f_r(x,\vec{\omega}_i,\vec{\omega}_r)}^{(C)} \overbrace{L_i(x,\vec{\omega}_i)}^{(A)} \overbrace{(\vec{\omega}_i \cdot \vec{n})}^{(B)} \overbrace{d\vec{\omega}_i}^{(D)}$$

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 $\vec{\omega}_0, \vec{\omega}_1, \ldots, \vec{\omega}_{N-1}$
    - 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)

$$\int_{\vec{\omega}_i \in \Omega} \ldots \cdot L_i(x, \vec{\omega}_i) \cdot \ldots \, d\vec{\omega}_i$$

Here $i$ stands for "incident"

Here $i$ stands for $i^{th}$

- t: sum (of few lights ! )

$$\sum_{i=0}^{N-1} \ldots \cdot L_i \cdot \ldots$$

Constant light from light i

- For colored light:

$$\sum_{i=0}^{N-1} \ldots \cdot \begin{pmatrix} L_i^R \\ L_i^G \\ L_i^B \end{pmatrix} \cdot \ldots$$

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 outoing 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 microscpoic level…
    The surface show micro-facet randomly oriented

# Approximation 2 :
# purely diffusive material

- From a function varying over incident and outgoing direction

$$\sum_{i=0}^{N-1} \ldots \cdot f_r(x, \vec{\omega}_i, \vec{\omega}_r) \cdot \ldots$$

- To a constant, called ≪albedo≫

$$\sum_{i=0}^{N-1} \ldots \cdot B_x \cdot \ldots$$

**albedo** (of point x):
ration between incident and outgoing light (to and from x).

- For colored material, 3 costants:

$$\sum_{i=0}^{N-1} \ldots \cdot \begin{pmatrix} D_x^R \\ D_x^G \\ D_x^B \end{pmatrix} \cdot \ldots$$

**diffusive color (or "ba color")**
(of point x)

# Approximation 2 : recap

- A purely diffusive material (or Labertian 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:
  - costant 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 that the original one
  - Diffusive materials
  - Material described by constant BDRF (1 or 3 parameters)
  - But it is still physically based, and to ≪energy≫ is conserved
    - That is, reflection does not increase of decrese 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

=

ambient

+

reflection

+

emission

One RGB componend for each term

# *Emissive term*

- LEDs, ligh bulbs, a stub…
- Does not depend on lights, does not lit objects, it only lit our eye
- Just an additive component
  - Costant  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

just
*Ambient*
*component*

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 if 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
$\theta=70^0$

Big diffuse term
$\theta=35^0$

Maximal diffuse term
$\theta=0^0$

# Diffuse reflection term

ZERO
diffuse term
$\theta=90^0$

ZERO
diffuse term
$\theta>90^0$

(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 direction
- 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

$$I_{spec} = I_{luce\ spec} \cdot k_{materiale\ \ spec} \cdot \cos^n \alpha$$

$$= I_{lucespec} \cdot k_{materialespec} \cdot (\hat{R} \cdot \hat{V})^n$$

# Specular reflection term: Blinn's variant

- Blinn–Phong light model:
  - Semplification 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

    *"half-way" vector*

  - (so we don't have to compute R )

# Specular reflection term: Blinn's variant

phong: $$I_{spec} = I_{lucespec} \cdot k_{materialspec} \cdot (\hat{R} \cdot \hat{V})^n$$

blinn-phong: $$I_{spec} = I_{lucespec} \cdot k_{materialspec} \cdot (\hat{H} \cdot \hat{N})^n$$

"half-way" vector

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

# I 4 fattori che consideriamo

Final light

=

ambient

+

Diffuse reflection

+

Specular reflection

+

emission

# Classic *GL lighting equation

$$I_{tot} = I_{ambient} \cdot k_{ambientMat} + I_{diffuse} \cdot k_{diffuseMat} \cdot (\hat{L} \cdot \hat{N}) + I_{specular} \cdot k_{specularMat} \cdot (\hat{H} \cdot \hat{N})^{n} + k_{emissiveMat}$$

$$\hat{H} = \frac{(\hat{L} + \hat{V})}{\left|\hat{L} + \hat{V}\right|}$$

Properties of material

Properties of light

# Materials...

| Material | GL_AMBIENT | GL_DIFFUSE | GL_SPECULAR | GL_SHININESS |
|---|---|---|---|---|
| Emerald | 0.0215<br>0.1745<br>0.0215<br>0.55 | 0.07568<br>0.61424<br>0.07568<br>0.55 | 0.633<br>0.727811<br>0.633<br>0.55 | 76.8 |
| Jade | 0.135<br>0.2225<br>0.1575<br>0.95 | 0.54<br>0.89<br>0.63<br>0.95 | 0.316228<br>0.316228<br>0.316228<br>0.95 | 12.8 |
| Obsidian | 0.05375<br>0.05<br>0.06625<br>0.82 | 0.18275<br>0.17<br>0.22525<br>0.82 | 0.332741<br>0.328634<br>0.346435<br>0.82 | 38.4 |
| Pearl | 0.25<br>0.20725<br>0.20725<br>0.922 | 1.0<br>0.829<br>0.829<br>0.922 | 0.296648<br>0.296648<br>0.296648<br>0.922 | 11.264 |
| Ruby | 0.1745<br>0.01175<br>0.01175<br>0.55 | 0.61424<br>0.04136<br>0.04136<br>0.55 | 0.727811<br>0.626959<br>0.626959<br>0.55 | 76.8 |
| Turquoise | 0.1<br>0.18725<br>0.1745<br>0.8 | 0.396<br>0.74151<br>0.69102<br>0.8 | 0.297254<br>0.30829<br>0.306678<br>0.8 | 12.8 |
| Black Plastic | 0.0<br>0.0<br>0.0<br>1.0 | 0.01<br>0.01<br>0.01<br>1.0 | 0.50<br>0.50<br>0.50<br>1.0 | 32 |
| Black Rubber | 0.02<br>0.02<br>0.02<br>1.0 | 0.01<br>0.01<br>0.01<br>1.0 | 0.4<br>0.4<br>0.4<br>1.0 | 10 |
| Brass | 0.329412<br>0.223529<br>0.027451<br>1.0 | 0.780392<br>0.568627<br>0.113725<br>1.0 | 0.992157<br>0.941176<br>0.807843<br>1.0 | 27.8974 |
| Bronze | 0.2125<br>0.1275<br>0.054<br>1.0 | 0.714<br>0.4284<br>0.18144<br>1.0 | 0.393548<br>0.271906<br>0.166721<br>1.0 | 25.6 |
| Polished Bronze | 0.25<br>0.148<br>0.06475<br>1.0 | 0.4<br>0.2368<br>0.1036<br>1.0 | 0.774597<br>0.458561<br>0.200621<br>1.0 | 76.8 |
| Chrome | 0.25<br>0.25<br>0.25<br>1.0 | 0.4<br>0.4<br>0.4<br>1.0 | 0.774597<br>0.774597<br>0.774597<br>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. lighbulbs

# 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:

$$f_{attenuation} = \left(\frac{1}{c \cdot d_{\mathrm{L}}^2}\right)$$

Arbitrary factor

distance $m$

# Positional light

- In practical terms, that is too much
- Often replaced with a polynomial:

$$f_{attenuation} = \min\left(\frac{1}{c_1 + c_2 d_{\mathrm{L}} + c_3 d_{\mathrm{L}}^2},\ 1\right)$$

Arbitrary factors

# Modelling lights

$$I_{tot} = \left( \begin{array}{l} I_{ambient} \cdot k_{ambientMat} + \\ I_{diffuse} \cdot k_{diffuseMat} \cdot (\hat{L} \cdot \hat{N}) + \\ I_{specular} \cdot k_{specularMat} \cdot (\hat{H} \cdot \hat{N})^n + \end{array} \right) \cdot f_{attenuation}$$

$$k_{emissiveMat}$$

$$f_{attenuation} = \min\left( \frac{1}{c_1 + c_2 d_{\mathrm{L}} + c_3 d_{\mathrm{L}}^2}, \quad 1 \right)$$

# 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

$$f_s = \begin{cases} 1 & \alpha < \theta_{inner} \\ \cos(\alpha - \theta_{inner})\text{^}n & \theta_{inner} < \alpha < \theta_{outer} \\ 0 & \alpha > \theta_{outer} \end{cases}$$

$\alpha$ : angle between $\omega$ and $S_{dir}$

$$f_s = f(S_{dir}, \theta_{inner}, \theta_{outer}, n)$$

# Spotlights

$$I_{tot} = \left( \boxed{I_{ambient}} \cdot k_{ambientMat} + \boxed{I_{diffuse}} \cdot k_{diffuseMat} \cdot \left(\hat{L} \cdot \hat{N}\right) \cdot f_s + \boxed{I_{specular}} \cdot k_{specularMat} \cdot (\hat{H} \cdot \hat{N})^n \cdot f_s + k_{emissiveMat} \cdot f_s \right) \cdot f_{attenuation}$$

$$f_{attenuation} = \min\left(\frac{1}{c_1 + c_2 d_{\mathrm{L}} + c_3 d_{\mathrm{L}}^2},\ 1\right)$$

$$f_s = f\left(L, spot_{direction}, spot_{cutoffAngle}, spot_{beamwidth}\right)$$

# 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 $\boldsymbol{u}$ tangent to the surface and the normal $\boldsymbol{n}$. Since they are orthogonal $\boldsymbol{n}^T\boldsymbol{u} = 0$

- If M is the the transformation from object space to view (or world) space, can we say that the transformated $n$ and $u$ still orthogonal?

$$(\mathrm{M}\boldsymbol{n})^T(\mathrm{M}\boldsymbol{u}) = 0\ ?$$

- **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)

- $\boldsymbol{n}^{\boldsymbol{T}}\boldsymbol{u} = 0 \Rightarrow \boldsymbol{n}^{\boldsymbol{T}}\mathrm{M}^{-1}\mathrm{M}\boldsymbol{u} = 0 \Rightarrow (\boldsymbol{n}^{\boldsymbol{T}}\mathrm{M}^{-1})(\mathrm{M}\boldsymbol{u}) = \boldsymbol{0}$
- Therefore the transformed normal $\boldsymbol{n}'$ is:

$$\boldsymbol{n}'^{\boldsymbol{T}} = (\boldsymbol{n}^{\boldsymbol{T}}\mathrm{M}^{-1}) \Rightarrow \boldsymbol{n}' = (\boldsymbol{n}^{\boldsymbol{T}}\mathrm{M}^{-1})^{\boldsymbol{T}} \Rightarrow$$
$$\boldsymbol{n}' = \boldsymbol{M}^{-\boldsymbol{1}^{\boldsymbol{T}}}\boldsymbol{n}$$

- The normal vector must be transformed by the inverse transposed of the matrix transforming the points
- Is it *always* necessary?
  - No, if M preserves lengths and angles is not necessary
  - if 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)

$$n' = \left( \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix}^{-1} \right)^{T} n = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} n$$

# 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

$$L_{\mathrm{r}} = \frac{\rho}{\pi} L_{\mathrm{p}} \int_{p \in A} \frac{\cos\theta \cos\theta_{\mathrm{p}}}{r_{\mathrm{p}}^2} dA_{\mathrm{p}}$$

- 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..)

$$L_r = L_i \frac{D\ G\ F}{(N \cdot L)(N \cdot V)}$$

# 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

$$G_2 = \frac{2(N \cdot H)(N \cdot L)}{V \cdot H}$$ ← Shadowing effect

$$G_1 = \frac{2(N \cdot H)(N \cdot V)}{V \cdot H}$$ ← Masking effect

$$G = \min(1, G_1, G_2)$$

# Cook-Torrance model

- **Fresnel effect**: light at a grazing 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 if 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 $\eta_1, \eta_2$ 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

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left[1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right]$$

$$c = \sqrt{V \cdot H}$$

$$g = \sqrt{c^2 + \eta^2 - 1}$$

- **Schlick's approximation**: a widely used approximation of the actual Fresnel factor in specular reflection ⇒

$$F = R_0 + (1 - R_0)(1 - \cos(\theta))^5$$

$$R_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2$$

# Fresnel effect on materials

Gold F(0) = 0.82
Silver F(0)=0.95

Glass F(0) = 0.04
Diamond F(0) = 0.15

*Stephen H. Westin s, Cornell*

# 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 labertian lighting

Image of the moon

Shades varies greatly

Edges going black

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 diretion
- 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 $\sigma$, the larger the spectrum of slopes, the higher the roughness

# The Oren-Nayar model

$$L_r = \rho\, I_i (N \cdot L)(A + BC \sin\alpha \tan\beta)$$

$$A = 1.0 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.009}$$

$A$ and $B$ depend on roughness

$$C = \max(0.0, L'V')$$
$$L' = L - (L \cdot N)N$$
$$V' = V - (L \cdot N)N$$

$C$ depends on the **azimuth angle** beween 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:

$$L(x, lights, objects\ in\ the\ scene) = final_color$$

- **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
  - Gauraud shading: lighting is computed per-vertex and color is intepolated over the fragments
  - Phong shading: lighiting 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, there fore 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
- Gaurad shading solves the hard discontinuity problem along the edges between triangles
  - Or does it?

# Gourad Shading: where in the pipeline?

# Gourad Shading: results

# Gourad Shading: problem 1

- The common barycentric coordinates on the edges do not completely solve the problem of visible tessellation

# Gourad Shading: problem 2

- Results of Gauraud shading are highly dependent on (fineness of) tessellation
- E.g. Specular highlights are likely to be missed or noticably 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 |
|---|---|---|---|
| Gauraud Shading | Computes transformations and Lighing | color | Pass throug |
| 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 lenght
- We need to normalize the vector before using it
- Is it entirely correct? Not really...

Where have we seen this before?

Interpolate this!

# Comparing shading modes

- Gauraud Shading: cheaper to compute, because vertices are normally much less the fragments
- Phong Shading: more costly but more precise results
  - Especially with small highlights

Flat shading | Gauraud shading | Phong Shading

Comparison using Phong Lighting

# 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

glTF

PLY

VRML

X3D

OFF

STL

COLLADA

.3ds

# .OFF

- Very simple ASCII format
- Only supports vertex positions, faces and color per face
- Createb for geometry processing, not meant for rendering

comment

#vert, #faces,#edges

Vertex positions

#verts of a face, list of corners, color

```
OFF
# cube.off
# A cube
8 6 12
 1.0  0.0 1.4142
 0.0  1.0 1.4142
-1.0  0.0 1.4142
 0.0 -1.0 1.4142
 1.0  0.0 0.0
 0.0  1.0 0.0
-1.0  0.0 0.0
 0.0 -1.0 0.0
4  0 1 2 3  255 0 0 #red
4  7 4 0 3  0 255 0 #green
4  4 5 1 0  0 0 255 #blue
4  5 6 2 1  0 255 0
4  3 2 6 7  0 0 255
4  6 5 4 7  255 0 0
```

# .STL (3DSystems)

- **ST**ereo**L**itography
- Only supports vertex positions and triangle faces
- No metadata
- Delevoped 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

- **PoLY**gon file format (also Stanfor File Format)
- Supports per vertex attributes («properties»): color, normal, texture coordinates
- Supports user-defined properties
  - You could define the material for Phong lighiting..
- 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 ....
```

# .VRL and .X3D

- **V**irtual **R**eality **M**odeling **L**anguage
- 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** **T**ransmission **F**ormat
- Developed for efficient streaming/loading/rendering of 3D scenes
  - The «jpeg of 3D objects»
- Describe a scene
  - Hierarchies
  - Animation
  - **P**hysically **B**ased **R**endering

# .obj

- Developed for modeling and rendering
- Define geometric objects alone (no scene hierarcy, 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)

comment

Vertex position

Texture coordinates (next week)

Vertex normals

face

```
# 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
```

A face specifies, for each vertex, an index to its position, one to its tex coords, and one to its normal

# .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** (**tex**ture **el**ement) 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?

Reading from texture memory is referred to as **Texture Look-Up**

Texture coordinates as vertex attribute

Text coords are interpolated

Interpolated cords are read by the FS, which **reads** the corresponding data from **Texture Memory**

The most basic pattern, other choices are possible

# 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?

x has barycentric coords $c_0, c_1, c_2$ in triangle $v_1\ v_2\ v_3$

P(x) has barycentric coords $c_0, c_1, c_2$ in triangle $P(v_1)\ P(v_2)\ P(v_3)$

This is true for linear projections

# Attributes Interpolation with Perspective Projection

- Perpective 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 is 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 attibutes 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 $\mathbf{p}'$

$$\boxed{c'_0, c'_1 \neq c_0, c_1}$$

# Attributes Interpolation with Perspective Projection

$$\mathbf{p}' = \begin{pmatrix} c'_0\, x'_0 + c'_1\, x'_1 \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} x'_0 \\ z'_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ z_0 \\ -\frac{z_0}{d} \end{bmatrix} = \begin{bmatrix} \frac{x_0}{-z_0/d} \\ \frac{z_0}{-z_0/d} \\ 1 \end{bmatrix}$$

$$x'_0 = \frac{x_0}{-z_0/d} \qquad x'_1 = \frac{x_1}{-z_1/d}$$

$$\mathbf{p}' = \begin{pmatrix} c'_0 \dfrac{x_0}{(-z_0/d)} + c'_1 \dfrac{x_1}{(-z_1/d)} \\ 1 \end{pmatrix}$$

# Attributes Interpolation with Perspective Projection

$$w_o = -\frac{z_0}{d} \qquad w_1 = -\frac{z_1}{d}$$

$$\mathbf{p}' = \begin{pmatrix} c'_0 \frac{x_0}{w_0} + c'_1 \frac{x_1}{w_1} \\ 1 \end{pmatrix}$$

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

$$w_o = -\frac{z_0}{d} \qquad w_1 = -\frac{z_1}{d}$$

$$\mathbf{p}' = \begin{pmatrix} c'_0 \dfrac{x_0}{w_0} + c'_1 \dfrac{x_1}{w_1} \\ 1 \end{pmatrix}$$

$$\mathbf{p}' = \begin{pmatrix} \dfrac{c'_0}{w_0} x_0 + \dfrac{c'_1}{w_1} x_1 \\ \dots \\ 1 \end{pmatrix} \cdot \frac{1}{\dfrac{c'_0}{w_0} + \dfrac{c'_1}{w_1}} = \begin{pmatrix} \dfrac{\dfrac{c'_0}{w_0}}{\dfrac{c'_0}{w_0} + \dfrac{c'_1}{w_1}} x_0 + \dfrac{\dfrac{c'_1}{w_1}}{\dfrac{c'_0}{w_0} + \dfrac{c'_1}{w_1}} x_1 \\ \dots \\ \dfrac{1}{\dfrac{c'_0}{w_0} + \dfrac{c'_1}{w_1}} \end{pmatrix}$$

# 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 $\mathbf{p}$

$$\mathbf{p}' = \begin{pmatrix} \overbrace{\frac{\frac{c'_0}{w_0}}{\frac{c'_0}{w_0}+\frac{c'_1}{w_1}}}^{c_0} x_0 + \overbrace{\frac{\frac{c'_1}{w_1}}{\frac{c'_0}{w_0}+\frac{c'_1}{w_1}}}^{c_1} x_1 \\ \dots \\ \frac{1}{\frac{c'_0}{w_0}+\frac{c'_1}{w_1}} \end{pmatrix}$$

# 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

if (u<0) u←0; if (u>1) u←1;
if (v<0) v←0; if (v>1) v←1;

# UV-coordinates wrapping: repeating

$$u \leftarrow u - \lfloor u \rfloor$$
$$v \leftarrow v - \lfloor v \rfloor$$

# 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 place side-by-side with a copy of itself

# 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

Screen space

Texture space

# 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

Screen Space

**magnification**

the pixel is smaller than the texel

**minification**

the pixel is bigger than the texel

Pixel shape in Texture Space

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

$c_{ij} == value\ of\ texel\ at\ position\ \mathbf{p}_{ij}$

Just like degree 1 Bezier patches

Bilinear interpolation

$$c = [\, c_{00}(1-u) + c_{10}\, u \,]\,(1-v) + [\, c_{01}(1-u) + c_{11}\, u \,]\, v$$

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

# Texture Filtering: Minification

- With Neares 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

Nearest Filtering

Bilinear interpolation

texture

# MipMapping

- **M**ultum **I**n **P**arvo (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

5
4
3
2
1
0
SIGGRAPH 2003
SIGGRAPH 2003
SAN DIEGO
altro
esempio

# 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

name generation and binding

Planes of image being uploaded

```
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);
```

Upload the image to the texture object

What texture level?
You may want to manually upload mimap level

Internal storage of the texture

Depth of image being uploaded

# Using textures with OpenGL

1. Loading image
2. Creating a texture
   - Uploading the image to the texture
   - Setting UV wrapping mode

Other choices

GL_MIRRORED_REPEAT
GL_CLAMP_TO_EDGE
GL_CLAMP_TO_BORDER

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);
```

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 = 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

# 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

normals encoded in the map

Normal map

Wikipedia

# 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*

$$\mathbf{p}_{os} = T_f \mathbf{p}_{\text{ts}}$$

- How do we compute the tangent frames? (next..)

# Tangent frames for Parametric Surfaces

$f: A \rightarrow B$
$A \subseteq \mathbb{R}^2$
$B \subseteq \mathbb{R}^3$

$$\boldsymbol{y}(t) = \frac{\partial f(t)}{\partial v}$$
$$\boldsymbol{x}(t) = \frac{\partial f(t)}{\partial u}$$
$$\boldsymbol{n}(t) = \boldsymbol{x}(t) \times \boldsymbol{y}(t)$$

# 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:

$F_o = \{\mathbf{v}_1 - \mathbf{v}_0, \mathbf{v}_2 - \mathbf{v}_0, \mathbf{v}_0\}$ in object space

$F_t = \{\mathbf{t}_1 - \mathbf{t}_0, \mathbf{t}_2 - \mathbf{t}_0, \mathbf{t}_0\}$ in texture space

Let $\boldsymbol{u_{os}}$ be the mapping of $\boldsymbol{u} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ in object space. Then $\boldsymbol{u_{os}}$ and $\boldsymbol{u}$ have the same coordinates in their respective frames.

Algorithm:

1. Find the coordinates of $\boldsymbol{u}$ in $F_t$, $\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$
2. Find $\boldsymbol{u_{os}} = u_1\,(\mathbf{v}_1 - \mathbf{v}_0) + u_2(\mathbf{v}_2 - \mathbf{v}_0)$

# 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 $(\mathbf{o}, \mathbf{o} + v_1, \mathbf{o} + v_2)$
2. $\mathbf{p} = (1 - \omega_1 - \omega_2)\ \mathbf{o}\ + \omega_1(\mathbf{o} + \boldsymbol{v_1}) + \omega_2(\mathbf{o} + \boldsymbol{v_2})$

$$\mathbf{p} = \mathbf{o} + \omega_1 v_1 + \omega_2 v_2$$

# Tangent Frames for Polygon Meshes

$$u_1 = \frac{((t_0 + u) - t_0) \times t_{20}}{t_{10} \times t_{20}} = \frac{u \times t_{20}}{t_{10} \times t_{20}} = \frac{[1,0]^T \times t_{20}}{t_{10} \times t_{20}} = \frac{t_{20v}}{t_{10} \times t_{20}}$$

$$u_2 = \frac{t_{10} \times ((t_0 + u) - t_0)}{t_{10} \times t_{20}} = \frac{t_{10} \times u}{t_{10} \times t_{20}} = \frac{t_{10} \times [1,0]^T}{t_{10} \times t_{20}} = \frac{-t_{10u}}{t_{10} \times t_{20}}$$

$$\boldsymbol{u_{os}} = Normalize(u_1\,(\mathbf{v}_1 - \mathbf{v}_0) + u_2(\mathbf{v}_2 - \mathbf{v}_0))$$

We can do the same for $\mathbf{v}_{os}$ or just use the normal:

$$\boldsymbol{v_{os}} = \boldsymbol{n} \times \boldsymbol{u}_{\mathrm{os}}$$

# 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

$$\mathrm{u}\big(f(s,t)\big) = s$$
$$\mathrm{v}\big(f(s,t)\big) = t$$

- 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]x[0,1] onto the mesh
  - Does it always exists? 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 parametrizazion 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 altogheter 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 altogheter 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 measure with two factors:
  - How much the *area* is preserved (up to a uniform scalig factor). If a parametrization preserves the area it is called **equiareal**
  - How much the angles are preserved. If a parametrization preserves the angels it is called **conformal** or **angle preserving**
  - If a parametrization preserves both angles and area it is called **isometric**
- Do isometric parametrization exists? 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

This is a very distorted parameterization

# Examples of Mesh Parametrization

Typically the surface is «cut» into regions easier to parametrize.
These cut 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 to for rendering
  - Project the points with the new matrices but, instead of assigning the outome 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?

$$f = argmax\ (\boldsymbol{v}_x, \boldsymbol{v}_y, \boldsymbol{v}_z)$$

What are the texture coordinates in face s?

$$s = ((\boldsymbol{v}_i / v_f) * 0.5) + 0.5$$
$$t = ((\boldsymbol{v}_j / v_f) * 0.5) + 0.5$$

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

# 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 $\eta_1, \eta_2$ 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,+x,-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

Framebuffer

COLOR_ATTACHMENT_0

DEPTH_ATTACHMENT

STENCIL_ATTACHMENT

# Creating a framebuffer object

```
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

Same pattern as for everything else (vertex buffers, textures..)

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

```
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,                 // target
GL_DEPTH_ATTACHMENT,            // attachment point
GL_RENDERBUFFER,                // render buffer target (only choice)
renderbuffer                    // render buffer to use
);
```

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,                // target
GL_DEPTH_COMPONENT16,           // internal representation
width,                          // size
height);
```

# Not just Cubemaps: equirectangular images

- Equirectangular images have been made popular by 360° cameras and applications

$\theta = \mathrm{atan2}(x, z) + \pi$

$\varphi = \mathrm{atan2}(\sqrt{x^2+z^2}, y)$

# 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

# 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 map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**: A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

Eye

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

final
**Screen buffer**

Part of the scene *seen* from the light, hence *lit*

# Shadow Mapping (in an ideal world)

Eye

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

final
**Screen buffer**

Part of the scene *seen* from the light, hence *lit*

# Shadow Mapping (in an ideal world)

**Shadow map**: A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**: A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Shadow Mapping (in an ideal world)

**Shadow map**:
A depth map obtained by rendering the scene from the point of view of the light

# Modelling the light source

- Create a projection matrix so that the projectors match the type of light:
  - Positional (and spotligh) require a **perspective** projection, because all rays leave from a point
  - Directional (that is, very faraway, like the sun) require an **orthogonal** projection, bause 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 the parameters of the projection matrix as:

$$l = x_m$$
$$r = x_M$$
$$b = y_m$$
$$t = y_M$$
$$n = -z_m$$
$$f = -z_M$$

Lez06_8_02_23_transformations_III pdf, slide 103

# 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 $\varepsilon$
- If $z_p + \varepsilon > d_p$ then $p$ is lit
- That is, *bias* the test in favour of *being lit*
- How big must $\varepsilon$ 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

$$\varepsilon = \text{clamp}(a \tan \text{acos}(n \cdot L), \min_\varepsilon, \max_\varepsilon)$$

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*

Aliasing with shadow mapping with decreasing resolution of the shadow map

# 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( dx = -1.5; dx <=1.5; dx = dx + 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: threat 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 $z^2$

$z$

Averaging over the kernel gives $E(z)$

$z^2$

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 $z_r > \mu$:

$$\mathbf{P}(z_{hk} > z_r) \cancel{\leq} = \frac{\sigma^2}{\sigma^2 + (z_r - \mu)^2}$$

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 weigths, named **kernel**. The size of the matrix is named **kernel size**

**F**:Filter

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |

Kernel size = 1

*

**A**:Original image

=

**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

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |

*

=

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

*

=

# 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 |

$*$ $= \frac{\partial I}{\partial y}$

$$\sqrt{\left(\frac{\partial I}{\partial y}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2} =$$

# Filtering

- Example: our Percentage Closer Filtering is a filter with constant weigths $\frac{1}{2(N+1)}$ 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 0
0 1 1 1 1 0 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 ($3\sigma$)

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)}{\sigma^2}}$$

Examples for $\mu = 0$

# Gaussian Blur

- The 2D kernel for Gaussian Blur is obtained as the product of two Gaussian functions

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{\sigma^2}}$$

$$\mathrm{k}(x,y) = f(x)f(y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{(x^2+y^2)}{\sigma^2}}$$

Example for $\sigma = 0.8$ and kernel size = 3

$$\begin{bmatrix} 0.00000067 & 0.00002292 & \mathbf{0.00019117} & 0.00038771 & \mathbf{0.00019117} & 0.00002292 & 0.00000067 \\ 0.00002292 & 0.00078633 & 0.00655965 & 0.01330373 & 0.00655965 & 0.00078633 & 0.00002292 \\ \mathbf{0.00019117} & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & 0.00655965 & \mathbf{0.00019117} \\ 0.00038771 & 0.01330373 & 0.11098164 & \mathbf{0.22508352} & 0.11098164 & 0.01330373 & 0.00038771 \\ \mathbf{0.00019117} & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & 0.00655965 & \mathbf{0.00019117} \\ 0.00002292 & 0.00078633 & 0.00655965 & 0.01330373 & 0.00655965 & 0.00078633 & 0.00002292 \\ 0.00000067 & 0.00002292 & \mathbf{0.00019117} & 0.00038771 & \mathbf{0.00019117} & 0.00002292 & 0.00000067 \end{bmatrix}$$

# 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)

$$Blur(i,j) = \sum_{k-N}^{N} \sum_{h=-N}^{N} f(h+N) f(k+N) I(i+h, j+k) =$$

$$\underbrace{\sum_{k=-N}^{N} f(k+N) \underbrace{\sum_{h=-N}^{N} f(h+N) I(i+h, j+k)}_{\text{1° pass, sum the pixels in the same row}}}_{\text{2° pass, sum the pixels in the same column (of the result of the 1° pass)}}$$

# 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 name “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)

$$L_o(x,\omega_r) = L_e(x,\omega_r) + \boldsymbol{k_a} + \sum_{\forall i} \boldsymbol{ReflFunc}(L(x,\omega_i))$$

- let us see $\boldsymbol{k_a}$ as the integral of light coming from every direction in the hemispere

$$L_o(x,\omega_r) = L_e(x,\omega_r) + k_a + \sum_{\forall i} \boldsymbol{ReflFunc}(L(x,\omega_i))$$

$$\boldsymbol{k_a} = \int_{\Omega} L_a(\omega_i) d\omega_i$$

- **Ambient Obscurance** (or Occlusion): and add a shadow scaling term based on the blockers that can prevent light from reaching point $x$

$$\boldsymbol{k_a} = \int_{\Omega} L_a(\omega_i)\, isBlocked(x,\omega_i) d\omega_i$$

# Ambient Occlusion/Obscurance

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

Phong lighting

Depth map

# 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

Phong lighting

Depth map

# 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

$$V(p) = \begin{cases} 1 & z(p) < d\,(p) \\ 0 & otherwise \end{cases}$$

- **Q:** how many samples? Where do you put them, exactly? How big is $r$?

# AO pipeline

Rasterize Scene and and write out buffers
G-Buffer
depth
normal
color
Compute AO (FullQuad rendering)
AO term
Blur the result (FullQuad rendering)
Blurred AO term
Compose the final image (FQ)
Final

# AO Pipeline: Compute AO

A slightly more interesting algorithm

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

# 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 $\rho$
[McGuire et al.2011]

Horizon Based AO with paired samples
[Mittring 2012]

[Images from Frederik P. Aalund Bachelor Thesis]

# 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

A back lit translucent object

- **Translucent** materials let the light trough 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

# GI: Translucent Shadow Maps

- A TSM is a shadow map that also includes normal and irradiance per textel
- 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

Single surface refract | Ray traced | This technique

- 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

Single surface refract | Ray traced | This technique

# 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…

depth position normal Radiant flux

# 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 poinst 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 ligthing)
  - The texels of the lightmap are referred to as **lumels**
- Idea: the view independ 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 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

Directional ligths

positional ligths

# 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

# 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

# Percentace 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 chep (finding the blocker)

$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$

# Percentace Closer Soft Shadows (PCSSF)

- Key
  size
  of a
  *dist*
- Hov
  bloc
  - E
    p
    c
- Pros/cons:
  - Pros: works well, a minor mod to SM, nice results
  - Cons: not very chep (finding the blocker)

$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$

# Lens flares

- Lens Flare: annoying artifacts create 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

- Extention of a single billboard: multiple billboards for each «flat» portion of the 3d objecte
- Need to be computed in preprocessing

# Depth-of-field

- Because of the lenses, points outside a given interval (in depth) will not porject 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

- Becau
  given
  exactl
- That r
- The si
  *field*
- Algori
  1. R
     c
  2. B
     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

$$L_o(x,\vec{\omega}_r) = \underbrace{L_e(x,\vec{\omega}_r)}_{\text{Emissive component}} + \int_{\vec{\omega}_i \in \Omega} \underbrace{f_r(x,\vec{\omega}_i,\vec{\omega}_r)}_{\text{BRDF}} \; \underbrace{L_i(x,\vec{\omega}_i)}_{\text{environment}} \; \underbrace{\cos(\theta) d\vec{\omega}_i}_{\text{Cosine law}}$$

# Introduction: Lighting recap

| symbol | Name | unit | |
|---|---|---|---|
| $\Phi$ | Radiant flux | Watt (J/s) | total amount of light passing through an area/volume. |
| $E = \frac{d\Phi}{dA}$ | Irradiance/ exitance | $Watt\ m^{-2}$ | Irradiance/exitance: amount of flux (arriving or leaving) per unit of area. |
| $I = \frac{d\Phi}{d\omega}$ | Directional intensity | $Watt\ s_r^{-1}$ | Directional Intensity: amount of flux per unit **solid angle** leaving from a point towards a direction $\omega$ |
| $L = \frac{d^2\Phi}{dA\cos\theta\, d\omega}$ | Radiance | $Watt\ s_r^{-1} m^{-2}$ | Radiance: flow of radiation per unity of area and per 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 $t$):

$$\boldsymbol{p} = (1 - t)\,\boldsymbol{p}_1 + t\,\boldsymbol{p}_2 \qquad with\ t > 0$$

$$\boldsymbol{p} = \boldsymbol{p}_1 + t\,(\boldsymbol{p}_2 - \boldsymbol{p}_1) \qquad with\ t > 0$$

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]x[w,h]

$$\mathbf{p_x} = \mathrm{l} + \frac{\left(\frac{1}{2} + \mathrm{i}\right)}{\mathrm{w}}(\mathrm{r} - \mathrm{l}) = -\frac{\mathrm{s_x}}{2} + \frac{\left(\frac{1}{2} + \mathrm{i}\right)}{\mathrm{w}} s_x$$

If $r = -l$ and $\mathrm{s_x} = r - l$

$$\mathbf{p_y} = \mathrm{b} + \frac{\left(\frac{1}{2} + \mathrm{j}\right)}{\mathrm{h}}(\mathrm{t} - \mathrm{b}) = -\frac{\mathrm{s_y}}{2} + \frac{\left(\frac{1}{2} + \mathrm{j}\right)}{\mathrm{w}} s_y$$

If $t = -b$ and $\mathrm{s_y} = t - b$

# 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]x[w,h]

$$\mathbf{p_x} = \mathrm{l} + \frac{\left(\frac{1}{2} + \mathrm{i}\right)}{\mathrm{w}}(\mathrm{r} - \mathrm{l}) = -\frac{s_x}{2} + \frac{\left(\frac{1}{2} + \mathrm{i}\right)}{\mathrm{w}} s_x$$

If $r = -l$ and $\mathrm{s_x} = r - l$

$$\mathbf{p_y} = \mathrm{b} + \frac{\left(\frac{1}{2} + \mathrm{j}\right)}{\mathrm{h}}(\mathrm{t} - \mathrm{b}) = -\frac{s_y}{2} + \frac{\left(\frac{1}{2} + \mathrm{j}\right)}{\mathrm{w}} s_y$$

If $t = -b$ and $\mathrm{s_y} = t - b$

# 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

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

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

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

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

$$\underbrace{(\mathbf{o}-\mathbf{c})(\mathbf{o}-\mathbf{c}) - \boldsymbol{r}^2}_{C} + \underbrace{t^2\mathbf{d}^2}_{A} + \underbrace{t\,2\boldsymbol{d}(\mathbf{o}-\mathbf{c})}_{B} = \mathbf{0}$$

$$\Rightarrow \quad At^2 + Bt + C = 0$$

$$t = \frac{-\mathrm{B} \pm \sqrt{(\mathrm{B}^2 - 4\mathrm{AC})}}{2\mathrm{A}}$$

# 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

$$\mathrm{Pl} = \{\mathbf{p} \in \mathbb{R}^3 \mid (\mathbf{p} - \mathbf{c})\boldsymbol{n} = \mathbf{0}\}$$

- Substituting the ray equation for p

$$(\mathbf{o} + t\boldsymbol{d} - \mathbf{c})\boldsymbol{n} = \mathbf{0}$$

$$t = -\frac{(\mathbf{o} - \mathbf{c}) \cdot \boldsymbol{n}}{d \cdot \boldsymbol{n}}$$

$d \cdot \boldsymbol{n} \neq \mathbf{0} \Rightarrow$ one solution

$d \cdot \boldsymbol{n} = \mathbf{0} \Rightarrow$ ray parallel to plane

$d \cdot \boldsymbol{n} = \mathbf{0}\ \boldsymbol{and}\ (\mathbf{o} - \mathbf{c}) \cdot \boldsymbol{n} = \mathbf{0}$ $\Rightarrow$ray included in plane

$t > 0$ $\Rightarrow$the view ray intersects the plane

# Ray-Quad intersection

- We need a reference frame $F = \{\mathbf{c}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{n}\}$ to define the quad as:

$$\mathrm{Q} = \{(\boldsymbol{x}, \boldsymbol{y}, \mathbf{0}) \mid -\boldsymbol{a} < \boldsymbol{x} < \boldsymbol{a}, -\boldsymbol{b} < \boldsymbol{y} < \boldsymbol{b}\}$$

- Intersection test: we *could*
  1. Do ray-plane intersection test and find $t_i$
  2. Express the intersection point $\mathbf{o} + \mathrm{t_i} \cdot d$ 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

$$\mathbf{o} + t \cdot \boldsymbol{d}$$

$$t = -\frac{(\mathbf{o_F} - \mathbf{c_F}) \cdot \boldsymbol{n}}{d_f \cdot \boldsymbol{n}}$$

$$\boldsymbol{n} = [0,0,1,0]^T$$

$$t = -\frac{\mathbf{o_F}_{\boldsymbol{z}}}{d_{f_{\boldsymbol{z}}}}$$

The intersection test is now one dimensional

$$o_F + t \cdot d_F = F^{-1}\mathbf{o} + t \cdot F^{-1}\,\boldsymbol{d}$$

# Intersection space

- Primitives are normally defined in their own frame
- Transforming objects more complex than spheres (which are rotationally invariant) and planes my 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 and 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 ligthting: 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 exaustive 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 (if it intersect something it is in shadow)

# 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

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 | Uniform | Jittered |
|---|---|---|
| | | |
| Domain not uniformly sampled | Domain uniformly but not in a random way | Uniform & random. Trading aliasing for noise |

```
S JitteredSampling(){
for each Cell in GRID
   S = S + RandomPointInThe Cell()
return S
}
```

# Jittering: from aliasing to noise

256 samples per pixel as reference

1 sample per pixel (jittered)

1 sample per pixel (no jitter)

4 samples per pixel (jittered)

# Indirect Lighting

- Recall the Rendering Equation

$$L_o(x,\vec{\omega}_r) = \overset{\text{Emissive component}}{L_e(x,\vec{\omega}_r)} + \int_{\vec{\omega}_i \in \Omega} \overset{\text{BRDF}}{f_r(x,\vec{\omega}_i,\vec{\omega}_r)} \; \overset{\text{environment}}{L_i(x,\vec{\omega}_i)} \; \overset{\text{Cosine law}}{\cos(\theta)} d\vec{\omega}_i$$

- 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

$$I = \int_a^b f(x)dx \qquad I \approx \sum_{i=1}^{N} f(x_i)\frac{b-a}{N}$$

$$x_i = a + \left(i - \frac{1}{2}\right)\frac{b-a}{N}$$

# Montecarlo Integration

$$I = \int_a^b f(x)dx$$

Definition of first momentum (mean value)

$$E[g(x)] = \int_a^b g(x)p(x)\,dx$$

It can be estimated as:

$$E[g(x)] \approx \frac{1}{N}\sum_{i=0}^{N-1} g(x_i)$$

Which we did for PCF shadow mapping

Then consider:

$$g(x) = \frac{f(x)}{p(x)} \; with\; uniform\; p(x) = \frac{1}{b-a}$$

$$I = \int_a^b f(x)dx = \int_a^b \frac{f(x)}{p(x)} p(x)\,dx \Rightarrow I \approx \frac{1}{N}\sum_{i=0}^{N-1} \frac{f(x_i)}{p(x_i)}$$

# Montecarlo Integration

- Pick the value of the function at uniform random positions and average their value

since $p(x_i) = \dfrac{1}{b-a}$

$$I \approx \frac{1}{N}\sum_{i=0}^{N-1}\frac{f(x_i)}{p(x_i)} = \frac{1}{N}\sum_{i=0}^{N-1} f(x_i)(b-a)$$

$$I \approx \frac{1}{N}\sum_{i=0}^{N-1}$$

# Example: Compute $\pi$

- $\pi$ is the area of the unit circle ($r = 1$) and it can be written as

$$\pi = \int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dx\,dy, \qquad f(x,y) = \begin{cases} 1, & x^2+y^2 \le 1 \\ 0, & otherwise \end{cases}$$

- Therefore its Montecarlo approximation is:

$$\pi \approx \frac{1}{N}\sum_i \frac{f(x_i,x_j)}{p(x_i,x_j)} = \frac{4}{N}\sum_i f(x_i x_j)$$

$$p(x_i,x_j) = \frac{1}{4}$$

$$Area(square) = 4r^2$$

$$Area(circle) = \pi r^2$$

$$\pi = 4\frac{Area(circle)}{Area(square)}$$

# Example: Co

- $\pi$ is the area of th
  can be written as

$$\pi = \int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dx$$

- Therefore its Mor

$$\pi \approx \frac{1}{N}\sum_i \frac{f(x_i,x_j)}{p(x_i,x_j)} = \frac{4}{N}\sum_i f(x_i x_j)$$

$$p(x_i,x_j) = \frac{1}{4}$$

$$Area(square) = 4r^2$$

$$Area(circle) = \pi r^2$$

$$\pi = 4\frac{Area(circle)}{Area(square)}$$

# 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

| plot | $\cos\left(\frac{\pi}{16}x\right)\left(1-\left(\frac{x}{2}\right)^2\right)$ | $x = -2$ to $2$ |
|---|---|---|

Plot

- Example: we want to integrate $f(x) = \cos(\frac{\pi}{16}x)(1-\left(\frac{x}{2}\right)^2)$ between -2 and 2 ($\int_{-2}^{2} f(x) \approx 2.626$)
- Let's apply MS with as before, with

$$p(x_i) = \frac{1}{2-(-2)} = 1/4$$

$$\int_{-2}^{2} f(x) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{\frac{1}{4}}$$

| N samples | 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

| plot | $\cos\left(\frac{\pi}{16}x\right)\left(1-\left(\frac{x}{2}\right)^2\right)$ | $x = -2$ to $2$ |
|---|---|---|

Plot

- The function looks similar to cos between –pi/2 and pi/2, let's try with a cosine distribution

$$q(x) = \cos(\frac{\pi}{4}x)$$

1. Normalize p over the interval so that $\int_a^b p(x) = 1$

$$\int_a^b \cos(\frac{\pi}{4}x) = \frac{4}{\pi}\sin\frac{\pi}{4}x\Big|_{-2}^{2} = \frac{8}{\pi} \Rightarrow p(x) = \frac{\cos(\frac{\pi}{4}x)}{\frac{8}{\pi}}$$

$$\int_{-2}^{2} f(x) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x)}$$

# Importance sampling

| plot | $\cos\left(\frac{\pi}{16}x\right)\left(1-\left(\frac{x}{2}\right)^2\right)$ | $x = -2$ to $2$ |
|---|---|---|

Plot

- The function looks similar to cos between –pi/2 and pi/2, let's try with a cosine distribution

$$q(x) = \cos(\frac{\pi}{4}x)$$

1. Normalize p over the interval so that $\int_a^b p(x) = 1$

$$\int_a^b \cos(\frac{\pi}{4}x) = \frac{4}{\pi}\sin\frac{\pi}{4}x\Big|_{-2}^{2} = \frac{8}{\pi} \Rightarrow$$

$$\int_{-2}^{2} f(x) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x)}$$

| 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 |

# Importance sampling

- The function looks similar to cos between –pi/2 and pi/2, let's try with a cosine distribution

$$q(x) = \cos(\frac{\pi}{4}x)$$

1. Normalize p over the interval so that $\int_a^b p(x) = 1$

$$\int_a^b \cos(\frac{\pi}{4}x) = \frac{4}{\pi}\sin\frac{\pi}{4}x\Big|_{-2}^{2} = \frac{8}{\pi} \Rightarrow$$

$$\int_{-2}^{2} f(x) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x)}$$

| 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 $r.y < 0$ $r = -r$
- However, Montecarlo integration will converge faster is 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:

$$BRDF_{spec} = f(x) = \begin{cases} 1, & \vec{\omega}_i = spec.\vec{\omega}_r \\ 0, & otherwise \end{cases}$$

- Why should we samples directions other than $\vec{\omega}_r$?

# Importance Sampling Lambertian Surfaces

- Lambertian (perfectly diffuse) surface reflect light equally I all directions: $f_r = k$

$$\int_{\vec{\omega}_i \in \Omega} f_r(x, \vec{\omega}_i, \vec{\omega}_r) \;\; L_i(x, \vec{\omega}_i) \;\; \cos(\theta) d\vec{\omega}_i =$$

$$\int_{\vec{\omega}_i \in \Omega} \boxed{k \cos(\theta)} \; \boxed{L_i(x, \vec{\omega}_i)} \, d\vec{\omega}_i$$

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)

$$\mathrm{r}= \left[a^2 \cos(2\pi\, b)\,, a^2\ \sin(2\pi\, b)\,, \sqrt{1-a^2}\,\right]$$

https://www.shadertoy.com/view/fljfz3

# 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 samping 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/7l2fR3

# 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 $(1-\mathrm{p})^4\ N$ will reach the light, but their contribution is scaled by $\frac{1}{(1-\mathrm{p})^4}$ 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

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

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

$$(\mathbf{p_0} + \boldsymbol{v}\, t)_x = x_{min} \Rightarrow t = \frac{x_{min} - \mathbf{p0}_x}{\boldsymbol{v}_x}$$

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

Chapter 5: Turning Vertices into Pixels  Introduction to Computer Graphics: a Practical Learning Approach.  51

# 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

Chapter 5: Turning Vertices Into Pixels  Introduction to Computer Graphics: a Practical Learning Approach.  52

# Ray-triangle Intersection

- First step: find the ray-plane intersection
- Second step: find one barycentric coordinate

$$\mathbf{v_{b0}} = \mathbf{v_{20}} - \frac{\mathbf{v_{20}}\mathbf{v_{21}}}{|\boldsymbol{v_{21}}|}\mathbf{v_{21}}$$

$$\alpha_0 = \mathbf{1} - \frac{\frac{\mathbf{v_{p0}}\ \mathbf{v_{b0}}}{|\mathbf{v_{b0}}|}}{\frac{\mathbf{v_{20}}\ \mathbf{v_{b0}}}{|\mathbf{v_{b0}}|}} = \frac{\mathbf{v_{p0}}\ \mathbf{v_{b0}}}{\mathbf{v_{20}}\ \mathbf{v_{b0}}}$$

$0 < \alpha_0 < 1$ ⇒ $\boldsymbol{v_p}$ 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.

# 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

Già meglio…

```
RaySceneIntersection(r){
  cell = NextCell();
  while(cell != NULL){
  for p in cell
  if( r intersects p){
    t = intersection(r,p);
    break;
    }
    cell = NextCell();
  }
}
```

# DDA-3d: rasterizzazione in 3D

- È molto simile alla rasterizzazione dei segmenti in 2d
  - Simile, non uguale
  - Anche il problema non è uguale

# 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

tMaxX < tMaxY

# DDA-3D: esempio

X = 1
Y = 0

# DDA-3D: esempio

X = 1
Y = 1

tMaxX < tMaxY

# DDA-3D: esempio

X = 2
Y = 1

tMaxX < tMaxY

# 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

Intersection(raggio,b)
b.ID = 56;
Intersection(raggio,a)
a.ID = 56;

**Skip** Intersection(raggio,a)
Intersection(raggio,c)
c.ID=56;
Intersection(raggio,d)
d.iD=56;

# Quanto devono essere grandi i voxel?

Se sono troppo grandi devo comunque fre 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

# 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 maiera 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 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)

Example in 2D, each quad is partitioned in four quads (**quad-tree**)

Cube size halves at each level

# Axis-Aligned Bounding Boxes (AABB)

- **AABB:** a box that *bounds* a set of primitives/objects and which sides are parallels 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 *max*C 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 minC=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-empy 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-aigned 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?

$$Cost(cell) = cost_traversal + Prob(left_cell|cell)\ Cost(left_cell) + Prob(right_cell|cell)\ Cost(right_cell)$$

$cell$

# 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

$$Prob(left_cell|cell) = \frac{\#rays\ intersecting\ \boldsymbol{left_cell}}{\#rays\ intersecting\ \boldsymbol{cell}}$$

$$Prob(left_cell|cell) = \frac{\int_{\sigma(left_cell)}\int_{\sigma(left_cell)} da\, da}{\int_{\sigma(cell)}\int_{\sigma(cell)} da\, da} = \frac{Area(left_cell)^2}{Area(cell)^2} = \frac{Area(left_cell)}{Area(cell)}$$

# $cost(left_cell)$

- 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

$cell$

$left_cell$ $right_cell$

$$Cost(left_cell) = 4$$

# 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 (againg) the rendering equation:

$$L_o(x,\vec{\omega}_r) = \underbrace{L_e(x,\vec{\omega}_r)}_{\text{Emissive component}} + \int_{\vec{\omega}_i \in \Omega} \underbrace{f_r(x,\vec{\omega}_i,\vec{\omega}_r)}_{\text{BRDF}} \; \underbrace{L_i(x,\vec{\omega}_i)}_{\text{environment}} \; \underbrace{\cos(\theta)d\vec{\omega}_i}_{\text{Cosine law}}$$

- 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** (quadrilaters, 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 **in**dependent

$$B_i = B_i^e + \rho_i \sum_{i=0}^{n} F_{i \to j} B_j$$

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* $\Phi_x$ for the area light $A_j$ ?

$$\Phi_x = \frac{\Phi_j}{A_j} \int_{A_j} \frac{cos\theta_x \, cos\theta_y}{\pi r^2} \, V(x, y) dA_j$$

$$V(x, y) = \begin{cases} 1 & x \; is \; visible \; from \; y \\ 0 & otherwise \end{cases}$$

# Radiosity: Form Factor

How much *flux* $\Phi_i$ for the area light $A_j$ ? Integrate $\Phi_x$ over $A_j$

$$\Phi_i = \frac{\Phi_j}{A_j} \int_{A_i} \int_{A_j} \frac{cos\theta_x \, cos\theta_y}{\pi r^2} \, V(x,y) \, dA_j \, dA_i$$

$$\boxed{\frac{\Phi_i}{\Phi_j}} = \frac{1}{A_j} \int_{A_i} \int_{A_j} \frac{cos\theta_x \, cos\theta_y}{\pi r^2} \, V(x,y) \, dA_j \, dA_i = F_{j \to i}$$

$$\frac{flux \;\; on \; area \;\; A_i}{flux \; from \; area \; A_j}$$

Note: if we swap $i$ and $j$ this term remains the same.
It only depends on the geometry, not on the flux

$$\Longrightarrow F_{j \to i} \;\; A_j = F_{i \to j} \;\; A_i$$

# Radiosity: Transport Equation

$$\Phi_i = \quad \Phi_i^e \quad + \rho_i \sum_{i=0}^{n} F_{j \to i} \Phi_j \; =$$

Recall: $B = \frac{\partial}{\partial A}\phi = \frac{\Phi}{A}$ (if the flux is constant over the area)

$$\frac{\Phi_i}{A_i} = \frac{\Phi_i^e}{A_i} \quad + \rho_i \sum_{i=0}^{n} F_{j \to i} \frac{\Phi_j}{A_i}$$

$$\frac{F_{j \to i}}{A_i} = \frac{F_{i \to j}}{A_j}$$

$$B_i = B_i^e \quad + \rho_i \sum_{i=0}^{n} F_{i \to j} B_j$$

# Radiosity: set up the system

- A linear equation for each patch: assemble all the equations in a linear system
  - $\rho_i, F_{i\to j}$ known for all $i, j$
  - $B_i^e$ known (the lights)
  - $B_i$ unknown ( the radiosity for each patch)

$$\begin{bmatrix} 1-\rho_1 F_{1\to 1} & -\rho_1 F_{1\to 2} & \cdots & -\rho_1 F_{1\to n} \\ \rho_2 F_{2\to 1} & 1-\rho_2 F_{2\to 2} & & \\ \vdots & & \ddots & \vdots \\ -\rho_n F_{n\to 1} & & \cdots & -\rho_n F_{n\to n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} B_1^e \\ B_2^e \\ \vdots \\ B_n^e \end{bmatrix}$$

# 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 + ρ_i Σ_j B_j F_{i→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 discontuities?
  - 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:

How ?

```
Irradiance(x){
Y = {y near X| stored Irr(y)};
if( Y not empty )
    Irr_x = InferIrradianceFrom(Y);
else
    Irr_x = ComputeIrradianceAt(x);
return Irr_x;
}
```

# 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) ?

[**Global Illumination using Photon Maps, Henrik Wann Jensen**]

# 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 if below a threshold or a maximum number of bounces is reached
  2. **Gather**: do a ray-tracing step, gather phton in the neighbor of the hit point and compute the radiance.

# Emitting photons

- Photon mapping models the *radiant flux* $\Phi$ (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

positional

Pick a random direction on the sphere

directional

Pick a random position

Area light

Generic surface light

Pick a random position on the surface and a random direction on the hemisphere

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, is 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 equilater 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 absorbtion $1 - s - d$ we could

1. 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**

2. 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

$\xi$ $uniform \in [0,1]$

$$\begin{aligned} \xi \in [0, d] &\longrightarrow \text{diffuse reflection} \\ \xi \in ]d, s+d] &\longrightarrow \text{specular reflection} \\ \xi \in ]s+d, 1] &\longrightarrow \text{absorption} \end{aligned}$$

# 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

(a) (b)

*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 $\omega$ 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 $L = \dfrac{d^2\Phi}{dA \cos\theta d\omega}$

$$\begin{aligned} L_r(x, \vec{\omega}) &= \int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) \frac{d^2\Phi_i(x, \vec{\omega}')}{\cos\theta_i \, d\omega_i' \, dA_i} |\vec{n}_x \cdot \vec{\omega}'| \, d\omega_i' \\ &= \int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) \frac{d^2\Phi_i(x, \vec{\omega}')}{dA_i} . \end{aligned}$$

- Counting only on the available photons

$$L_r(x, \vec{\omega}) \approx \sum_{p=1}^{n} f_r(x, \vec{\omega}_p, \vec{\omega}) \frac{\Delta\Phi_p(x, \vec{\omega}_p)}{\Delta A}$$

# 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.)

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)}{\sigma^2}}$$

- better than averaging, it still «blur» the result
  - Oblivious to the «value» of the sample
- **Bilateral filtering:** a variation of the Gaussian kernel were the value is taken into account

Gaussian Term

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q}\in\mathcal{S}} G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)\, I_{\mathbf{q}}$$

Values dissimilarity

$$W_{\mathbf{p}} = \sum_{\mathbf{q}\in\mathcal{S}} G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)$$

# Bilateral Filtering

# 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