

01 Computer Graphics Crash Course	2
02 Human Vision, Computer Colors, & Physics of Light	183
03 Rendering Equation	252
04 BRDF	285
05 Radiosity	330
06 Global Illumination	348
07 OpenGL	368
08 Textures, Mapping, & Sampling	415
09 Real-Time Shadows	527
10 Spatial Data Structures	551
11 Ray Tracing APIs Mesh Shaders	571

Crash Course into Computer Graphics

Csaba **Bálint**
first name family name

csabix@inf.elte.hu

Eötvös Loránd University,
Faculty of Informatics

Computer Graphics Lecture
Budapest 2022

Table of Contents

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
- Surfaces
- ▼ Transformations
 - ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
 - ▼ Projective transformations
- ▼ Graphics Pipeline
 - Transformations
 - Clipping
- Shading
- Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
- ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB



Course details

- ▶ Csaba Bálint csabix@inf.elte.hu (me). Room: 2-706.
- ▶ **Lecture**: Wednesday, 16:00-17:30
 - ▶ Exam: Explain 2 topics. You chose one, and I choose one.
 - ▶ Points from practice can help your course (but do not count on it).
- ▶ **Practice**: Wednesday, 17:45-19:15 and 19:30-21:00
 - ▶ Small assignment (≈ 30 points) and large assignment (100+ points)
 - ▶ Work during practice (≈ 15 points)
 - ▶ Scores above 100 points will count towards the lecture exam
- ▶ In both
 - ▶ **Grade boundaries: 40, 55, 70, 85.**

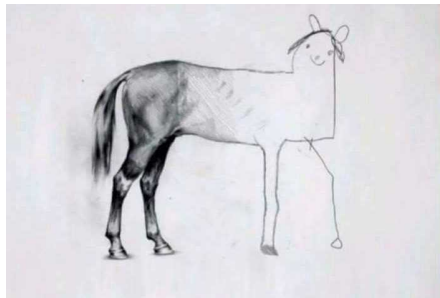


About this course

- ▶ This course was originally based on the Hungarian master course Advanced Computer Graphics
- ▶ The difficulty was lowered, now it consists of

$$\text{ARI CG} = \frac{1}{2} \cdot \text{BSc CG} + \frac{3}{5} \cdot \text{MSc CG}$$

- ▶ Prerequisites:
 - ▶ **Linear algebra** (the more the merrier)
 - ▶ **Calculus** (differentiation and integration)
 - ▶ **Geometry** (good understanding is beneficial)
- ▶ **This course is still hard.** To pass:
 - ▶ Do not miss Practices! It is **really** hard to catch up.
 - ▶ Maximize points. Ask questions.
 - ▶ Follow the Lectures.



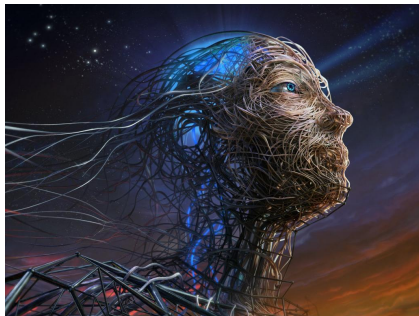
Further reading

1. *Edward Angel, Dave Shreiner*: Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL (6th Edition)
2. *Andrew Glassner*: [Principles of digital image synthesis](#)
3. *Pharr, Humphreys, Hanrahan*: [Physically Based Rendering \(From Theory to Implementation\)](#)
4. *Akenine-Möller, Haines, Hoffman*: [Real-Time Rendering \(4th edition\)](#)
5. *Tekla Tóth, Iván Eichhardt, Gábor Valasek*: [BSc Computer Graphics Lecture slides](#)



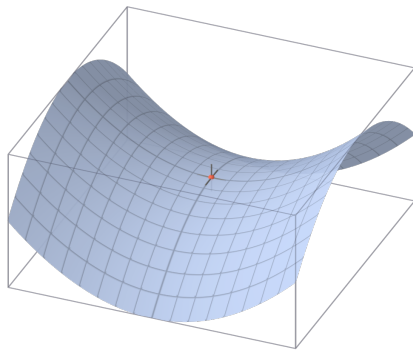
Introduction

- ▶ Computer graphics deals with the synthesis, analysis, and manipulation of visual content
- ▶ Our focus is on the basics of
 - ▶ **modeling**, i.e. how can we describe (and more importantly: store in a computer-decipherable way) our virtual worlds
 - ▶ **algorithms** which allow us to make pictures of our virtual world
 - ▶ **devices** on which we can display the results of our visualisations



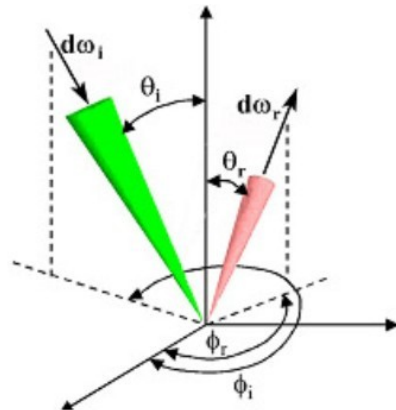
Modeling

- ▶ Geometric modeling
- ▶ Optical models
- ▶ Textures



Modeling

- ▶ Geometric modeling
- ▶ Optical models
- ▶ Textures



Modeling

- ▶ Geometric modeling
- ▶ Optical models
- ▶ Textures



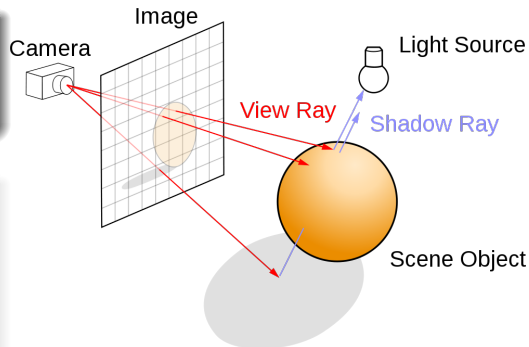
Algorithms

Approaches

- ▶ Ray tracing
- ▶ Incremental image synthesis

Light

- ▶ Reflections and refractions
- ▶ Shadows
- ▶ Global illumination
- ▶ Volumetric lighting



Henrik, Wikipedia

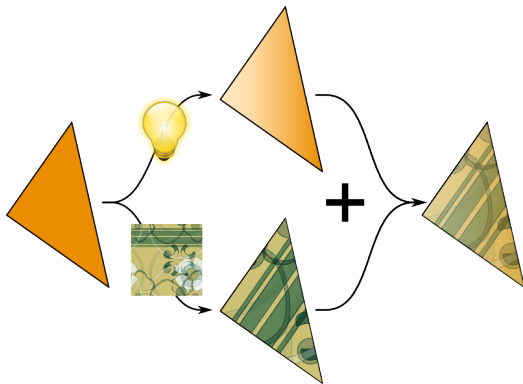
Algorithms

Approaches

- ▶ Ray tracing
- ▶ Incremental image synthesis

Light

- ▶ Reflections and refractions
- ▶ Shadows
- ▶ Global illumination
- ▶ Volumetric lighting



Algorithms

Approaches

- ▶ Ray tracing
- ▶ Incremental image synthesis

Light

- ▶ Reflections and refractions
- ▶ Shadows
- ▶ Global illumination
- ▶ Volumetric lighting



Cilles Tran, Oyonale.com



Algorithms

Approaches

- ▶ Ray tracing
- ▶ Incremental image synthesis

Light

- ▶ Reflections and refractions
- ▶ Shadows
- ▶ Global illumination
- ▶ Volumetric lighting



CryEngine2

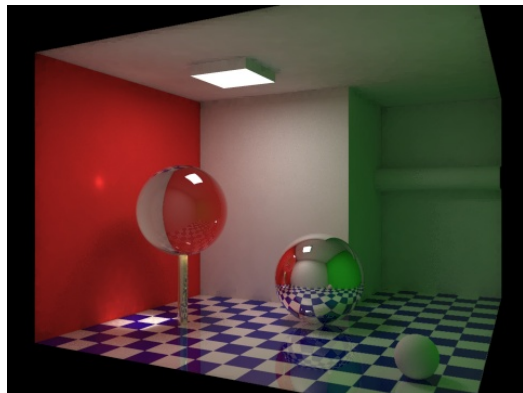
Algorithms

Approaches

- ▶ Ray tracing
- ▶ Incremental image synthesis

Light

- ▶ Reflections and refractions
- ▶ Shadows
- ▶ Global illumination
- ▶ Volumetric lighting



Algorithms

Approaches

- ▶ Ray tracing
- ▶ Incremental image synthesis

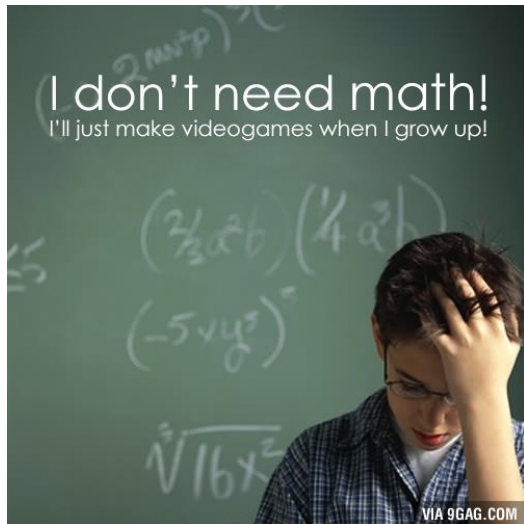
Light

- ▶ Reflections and refractions
- ▶ Shadows
- ▶ Global illumination
- ▶ Volumetric lighting



BlendELF.com





Coordinate systems

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
- Surfaces
- ▼ Transformations
 - ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
 - ▼ Projective transformations
- ▼ Graphics Pipeline
 - Transformations
 - Clipping
- Shading
- Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
- ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB

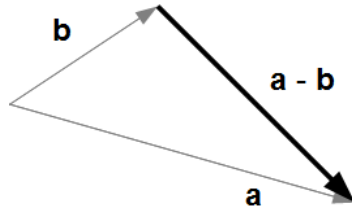
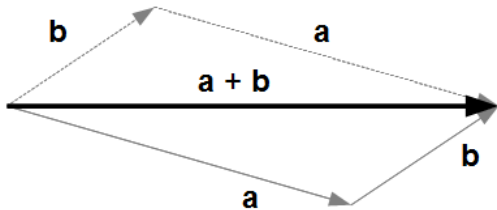


Points and vectors

- ▶ A **point** is an entity that has a location in space (or plane), but it has no extent
- ▶ A **vector** is an element of a **vector space**. Geometrically, a vector has a direction and a magnitude. All the usual operations are defined on them: vector addition, subtraction, multiplication by scalar, dot product, and cross product
- ▶ The following operations can be carried out that bridge the realm of points and vectors:
 - ▶ Difference of points yields a vector $A - B = \mathbf{v}$ that translates B to A .
 - ▶ A point plus a vector translates the point $B + \mathbf{v} = A$ to another one
 - ▶ Barycentric combination of points $\sum_i \lambda_i P_i$ where $\sum_i \lambda_i = 1$ (more details later)
 - ▶ Barycentric combination of vectors $\sum_i \lambda_i \mathbf{v}_i$ where $\sum_i \lambda_i = 0$
- ▶ In the following we presume that n points/lines/planes will be n **distinct** points/lines/planes.



Vector addition and subtraction



Vector dot product

The dot product of vectors $\mathbf{a} = [a_x, a_y, a_z]$ and $\mathbf{b} = [b_x, b_y, b_z]$ is denoted by $\langle \mathbf{a}, \mathbf{b} \rangle$ and can be computed using their coordinates as

$$\langle \mathbf{a}, \mathbf{b} \rangle = a_x b_x + a_y b_y + a_z b_z.$$

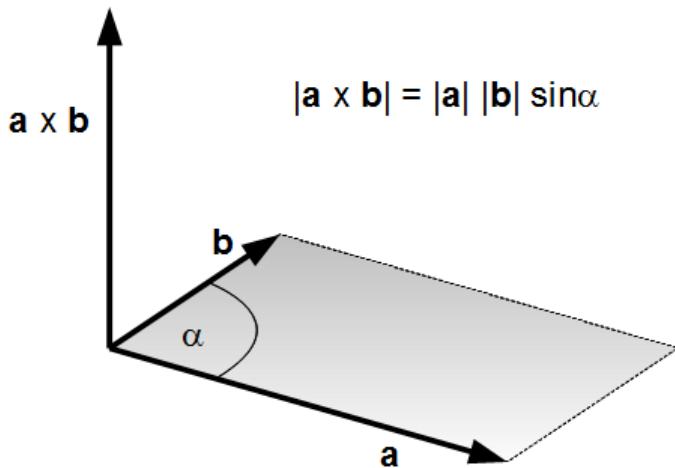
This is equivalent to

$$\langle \mathbf{a}, \mathbf{b} \rangle = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \cos(\alpha),$$

where α denotes the angle between vectors \mathbf{a} and \mathbf{b} .



Vector cross product



Vector cross product

You can use determinants to compute the cross product:

$$\begin{aligned}
 \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \\
 &= \mathbf{i} \cdot \begin{vmatrix} a_y & a_z \\ b_y & b_z \end{vmatrix} - \mathbf{j} \cdot \begin{vmatrix} a_x & a_z \\ b_x & b_z \end{vmatrix} + \mathbf{k} \cdot \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix} \\
 &= \begin{bmatrix} a_y b_z - a_z b_y \\ -a_x b_z + a_z b_x \\ a_x b_y - a_y b_x \end{bmatrix}
 \end{aligned}$$



Notation

- ▶ Points: $\mathbf{a} \in \mathbb{E}^2, \mathbf{b} \in \mathbb{E}^3$
- ▶ Vectors: $\mathbf{v} \in \mathbb{R}^n, n = 2, 3, \dots$
 - ▶ Special notation: $\mathbf{v} \in \mathbb{R}^n$ is the direction of vector \mathbf{v} , i.e. $|\mathbf{v}| = \|\mathbf{v}\|_2 = 1$.
- ▶ Lines: e, f, g, \dots
- ▶ Planes: S, \dots
- ▶ Matrices: $M, \mathbf{M} \in \mathbb{R}^{n \times m}$



Coordinate systems

- ▶ Coordinate systems allow us to uniquely represent points of the space with n-tuples (of numbers)

- ▶ E.g. $\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \in \mathbb{E}^3$

- ▶ It allows us to use algebraic and analytic tools in geometric problems

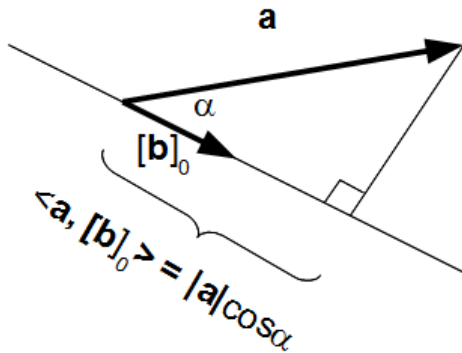


Cartesian coordinate system

- ▶ The Cartesian coordinate system uniquely assigns a pair [triple] of numbers to each finite point of the Euclidean plane [space]
- ▶ A Cartesian coordinate system is defined by a point in the space, in other words the **origin** \mathbf{o} , and an orthonormal system of three vectors, \mathbf{i} , \mathbf{j} , and \mathbf{k}
- ▶ Then the x, y, z coordinates of a point \mathbf{p} are the (signed) orthogonal projections of the vector $\mathbf{p} - \mathbf{o}$ to the orthonormal basis vectors $\mathbf{i}, \mathbf{j}, \mathbf{k}$
- ▶ **Reminder:** the (signed) orthogonal projection of vector \mathbf{a} to unit vector $[\mathbf{b}]_0$ is $\langle \mathbf{a}, [\mathbf{b}]_0 \rangle = |\mathbf{a}| \cos \angle(\mathbf{a}, [\mathbf{b}]_0)$

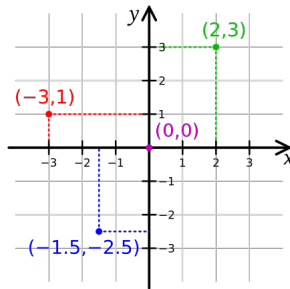


Signed orthogonal projection



Geometric interpretation

- ▶ **In other words:** $p(x, y, z)$ denotes the point of the space which we get by starting from the origin o and going x units in the direction of i , y units in the direction of j , and z units in the direction of k



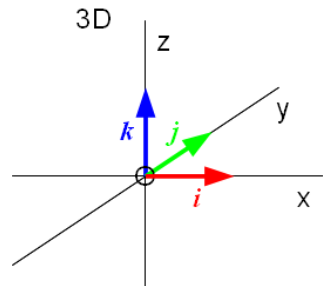
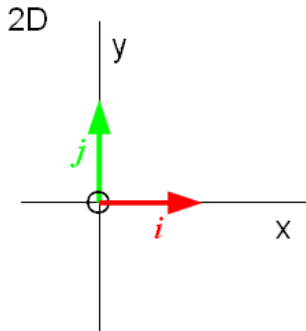
Geometric interpretation

- ▶ That is, given the orthonormal basis vectors \mathbf{i} , \mathbf{j} , \mathbf{k} , the Cartesian coordinates $[x, y, z]^T$ denote the following point of the Euclidean space:

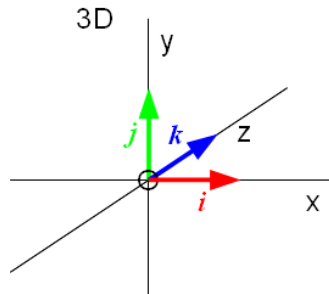
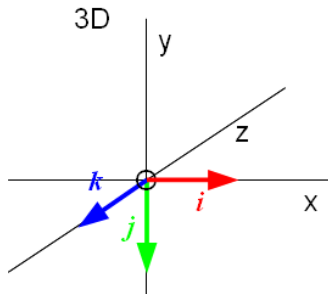
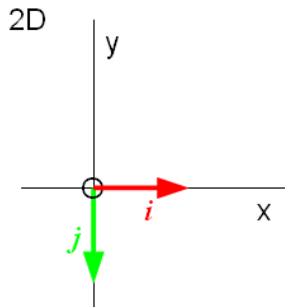
$$\begin{aligned}\mathbf{p} &= \mathbf{o} + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \\ &= \mathbf{o} + x \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + z \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\end{aligned}$$



Right-handed coordinate systems

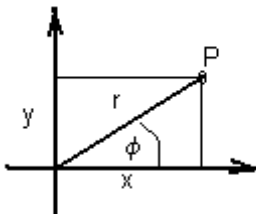


Left-handed coordinate systems



Polar coordinates

- ▶ Each point of the plane is determined by a distance from a fixed point o (**reference point**) and an angle from a fixed (reference) direction (**polar axis**)
- ▶ The polar coordinates of p are (r, ϕ) :
 - ▶ $r \geq 0$: $r = |p - o|$
 - ▶ $\phi \in [0, 2\pi)$: the angle between the line through o and p and the reference direction



Polar coordinates: Conversions

- ▶ Polar \rightarrow Cartesian: $(r, \varphi) \rightarrow (x, y)$
 - ▶ $(x, y) = r \cdot (\cos \phi, \sin \phi)$
- ▶ Cartesian \rightarrow polar: $(x, y) \rightarrow (r, \varphi)$
 - ▶ $r = \sqrt{x^2 + y^2}$
 - ▶

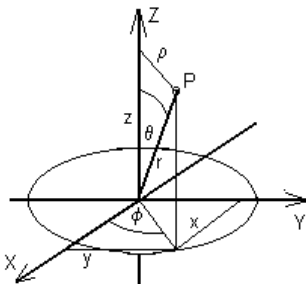
$$\phi = \text{atan2}(y, x) = \begin{cases} \text{atan}\left(\frac{y}{x}\right), & x > 0 \wedge y \geq 0 \\ \text{atan}\left(\frac{y}{x}\right) + 2\pi, & x > 0 \wedge y < 0 \\ \text{atan}\left(\frac{y}{x}\right) + \pi, & x < 0 \\ \frac{\pi}{2}, & x = 0 \wedge y > 0 \\ \frac{3\pi}{2}, & x = 0 \wedge y < 0 \end{cases}$$

- ▶ Origin of the Cartesian system = polar reference point
- ▶ x Cartesian axis = polar axis
- ▶ What happens at $x = 0, y = 0$? When $r = 0$, any angle will result in $[0, 0]$



Spherical coordinates

- ▶ A planar polar coordinate system plus an additional axis (Z-axis, zenith), perpendicular to the polar plane
- ▶ The spherical coordinates (r, φ, θ) of \mathbf{p} denote:
 - ▶ φ : the polar angle of \mathbf{p} 's projection onto the polar plane
 - ▶ $\theta \in [0, \pi]$: the angle between the line through \mathbf{o} and \mathbf{p} and the Z-axis
 - ▶ r : $r = |\mathbf{p} - \mathbf{o}|$



Spherical coordinates: Conversions

- ▶ Spherical \rightarrow Cartesian: $(r, \varphi, \theta) \rightarrow (x, y, z)$

$$x = r \cos \varphi \sin \theta, \quad y = r \sin \varphi \sin \theta, \quad z = r \cos \theta$$

- ▶ Cartesian \rightarrow spherical: $(x, y, z) \rightarrow (r, \varphi, \theta)$

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\varphi = \operatorname{atan2}(y, x)$$

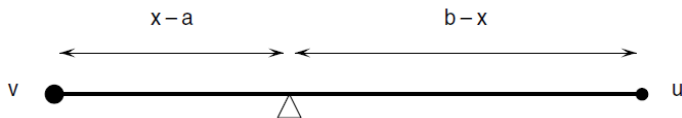
$$\theta = \arccos \frac{z}{r},$$

$$r \neq 0$$



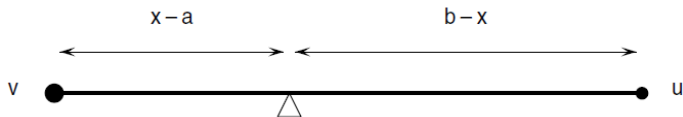
Barycentric coordinates

- ▶ August Ferdinand Möbius [1827]
- ▶ Motivation: a more balanced representation of the region of interest
- ▶ The term is derived from *barycenter*, meaning *center of gravity*.



What $u, v > 0$ weights should we put at the ends of the rod if we want the rod to stay in balance when elevated at the point denoted by the triangle?

Motivation



What $u, v > 0$ weights should we put at the ends of the rod if we want the rod to stay in balance when elevated at the point denoted by the triangle?

- ▶ Let x be the position of the triangle
- ▶ The rod will be balanced if $(x - a)v = (b - x)u$
- ▶ The above only determines the **ratio** of u and v !
- ▶ Using a **normalisation** condition of $u + v = 1$

$$u = \frac{x - a}{b - a}, \quad v = \frac{b - x}{b - a}$$



Barycentric coordinate system

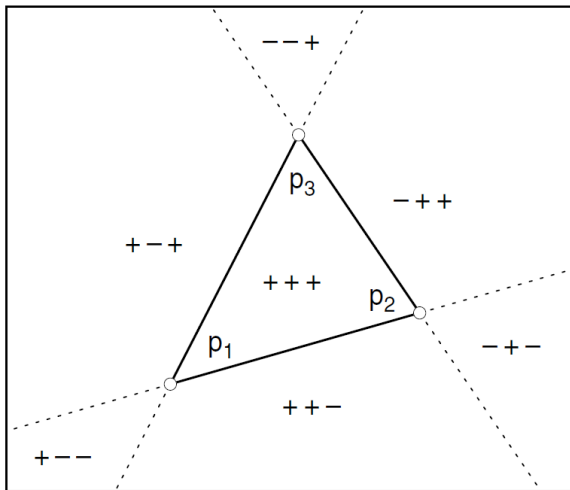
- ▶ Let $\mathbf{a}_0, \dots, \mathbf{a}_n$ be $n + 1$ affinely independent points in \mathbb{E}^n
- ▶ Then every \mathbf{x} point of \mathbb{E}^n can be expressed as a **barycentric combination** of basis points $\mathbf{a}_0, \dots, \mathbf{a}_n$, i.e. there exists $\lambda_i \in \mathbb{R}$, $i = 0, \dots, n$, such that

$$\mathbf{x} = \sum_{i=0}^n \lambda_i \mathbf{a}_i \quad \text{where} \quad \sum_{i=0}^n \lambda_i = 1.$$

- ▶ We need 3 points in the plane, and 4 in space to cover all dimensions
- ▶ The case of $\forall i : \lambda_i \geq 0$ is called **convex combination**



Planar barycentric coordinate system



Barycentric \rightarrow Cartesian conversion

- ▶ Let (u, v, w) be the barycentric coordinates wrt.
 $\mathbf{p}_1 = (x_1, y_1), \mathbf{p}_2 = (x_2, y_2), \mathbf{p}_3 = (x_3, y_3) \in \mathbb{E}^2$ ((x_i, y_i) given in Cartesian coordinates)
- ▶ Then the Cartesian coordinates of the point represented by the barycentric coordinates (u, v, w) can be computed as $\mathbf{x} = u\mathbf{p}_1 + v\mathbf{p}_2 + w\mathbf{p}_3$, i.e.

$$x = ux_1 + vx_2 + wx_3$$

$$y = uy_1 + vy_2 + wy_3$$



Cartesian \rightarrow barycentric conversion

- ▶ Let $\Delta(\mathbf{a}, \mathbf{b}, \mathbf{c}) := \begin{vmatrix} 1 & 1 & 1 \\ a_x & b_x & c_x \\ a_y & b_y & c_y \end{vmatrix}$, $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{E}^2$
- ▶ Remark: $\Delta(\mathbf{a}, \mathbf{b}, \mathbf{c})$ equals to twice the signed area of the triangle spanned by $\mathbf{a}, \mathbf{b}, \mathbf{c}$ (the signed area is positive if the 3 vertices come in a counter-clockwise direction, otherwise it is negative)
- ▶ Remark: in space $\Delta(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \langle (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}), \mathbf{n} \rangle$, where \mathbf{n} is the normal of the 3 points' plane



Cartesian \rightarrow barycentric conversion

Let $\mathbf{x} \in \mathbb{E}^2$ be a point. Then its barycentric coordinates wrt.

$\mathbf{p}_1 = (x_1, y_1), \mathbf{p}_2 = (x_2, y_2), \mathbf{p}_3 = (x_3, y_3) \in \mathbb{E}^2$ affinely independent points are:

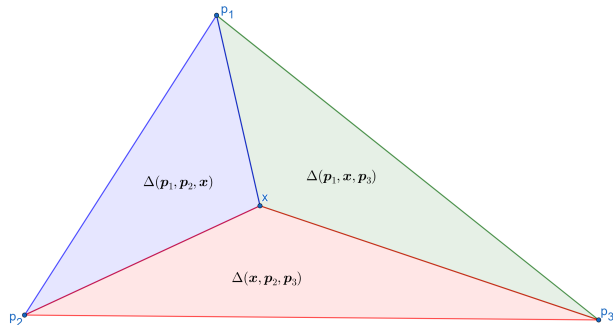
$$u = \frac{\Delta(\mathbf{x}, \mathbf{p}_2, \mathbf{p}_3)}{\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)}$$

$$v = \frac{\Delta(\mathbf{p}_1, \mathbf{x}, \mathbf{p}_3)}{\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)}$$

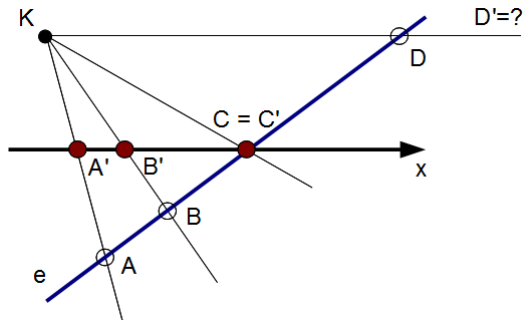
$$w = \frac{\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{x})}{\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)}$$

Such that

$$\mathbf{x} = u\mathbf{p}_1 + v\mathbf{p}_2 + w\mathbf{p}_3$$



Motivation



- ▶ D' is not on the Euclidean plane, since the projection line is parallel with the x axis
- ▶ Let us consider the same orientation of the lines (their direction) as a **point**!
- ▶ This way all parallel lines will have a common point at infinity



Definition

- ▶ Each line has an additional point, an **ideal point** on it, such that
 - ▶ The ideal points of parallel lines are the same
 - ▶ The ideal points of all the lines of the plane form an **ideal line**
 - ▶ The ideal lines of parallel planes coincide
 - ▶ All the ideal points of the space form the **ideal plane**

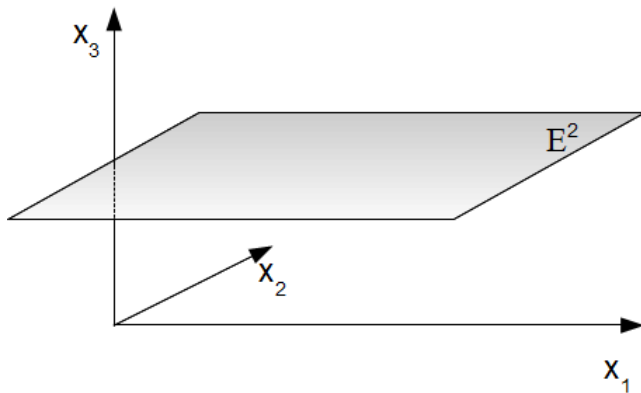


Properties

- ▶ Projective plane: the projective closure of \mathbb{E}^2 , that is all the points of \mathbb{E}^2 and its ideal line
 - ▶ Two points always determine a line in the projective plane (remark: remember, by two points we mean two *different* points!)
 - ▶ Two lines always determine a point in the projective plane!
 - ▶ ...
- ▶ Projective space: the projective closure of \mathbb{E}^3 , that is \mathbb{E}^3 plus its ideal plane
 - ▶ Three points always determine a plane (unless they are colinear).
 - ▶ Three distinct planes (not all three having the same orientation) always determine a point
 - ▶ ...

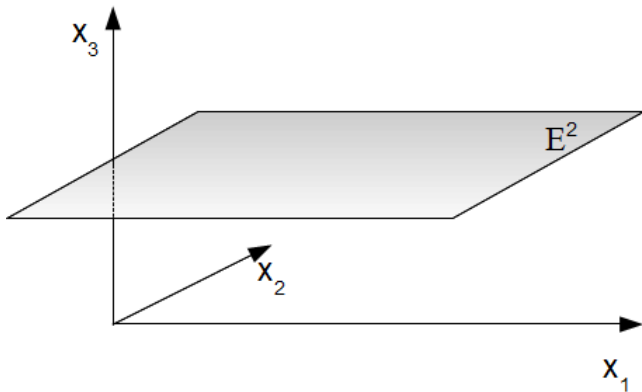


Embedding the Euclidean plane

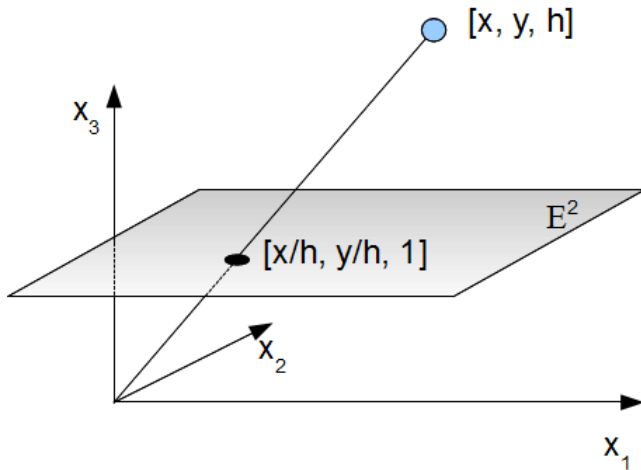


Embedding the Euclidean plane

● $[x, y, h]$



Embedding the Euclidean plane



Cartesian \rightarrow homogeneous conversion

- ▶ For each $\mathbf{p}(x, y, z) \in \mathbb{E}^3$ point, let its homogeneous coordinates be:

$$\begin{aligned}\mathbf{p}(x, y, z) &\rightarrow [x, y, z, 1] \\ &\approx h[x, y, z, 1] \\ &= [hx, hy, hz, h], \quad h \neq 0\end{aligned}$$

- ▶ For each $\mathbf{v} = [x, y, z]^T \in \mathbb{R}^3$ vector :

$$\begin{aligned}[x, y, z] &\rightarrow [x, y, z, 0] \\ &\approx h[x, y, z, 0] \\ &= [hx, hy, hz, 0], \quad h \neq 0\end{aligned}$$



Homogeneous \rightarrow Cartesian conversion

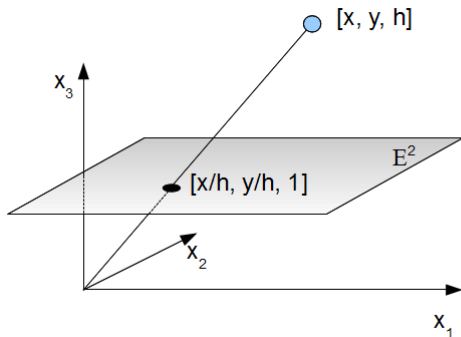
► What does $[x_1, x_2, x_3, x_4]$ denote in the Euclidean space?

► $x_4 \neq 0$: it's the following point of the Euclidean space:

$$[x_1, x_2, x_3, x_4] \approx \left[\frac{x_1}{x_4}, \frac{x_2}{x_4}, \frac{x_3}{x_4}, 1 \right] = \mathbf{p} \left(\frac{x_1}{x_4}, \frac{x_2}{x_4}, \frac{x_3}{x_4} \right)$$

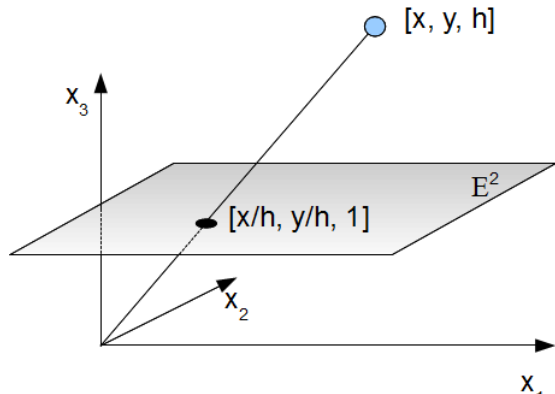
► $x_4 = 0$ and $x_1^2 + x_2^2 + x_3^2 \neq 0$: it is an ideal point, i.e. a vector!

► $x_i = 0, i = 1, 2, 3, 4$: cannot happen.



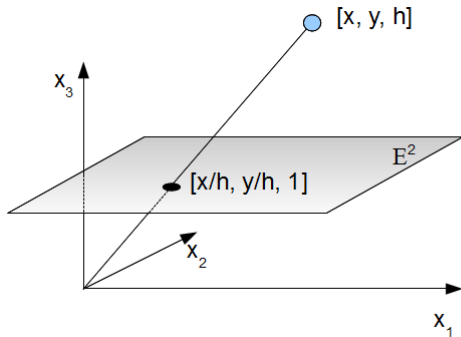
Homogeneous coordinates

- ▶ Let $c \neq 0$ then:
 - ▶ $[0, 0, 0, c]$ origin
 - ▶ $[c, 0, 0, 0]$ ideal point of the x axis
 - ▶ $[0, c, 0, 0]$ ideal point of the y axis
 - ▶ $[0, 0, c, 0]$ ideal point of the z axis



Homogeneous coordinate properties

- ▶ In the projective plane the point and the line, in the projective space the point and plane are dual entities
- ▶ Be careful, not everything is valid in the projective space that was valid in the Euclidean!
 - ▶ One point does not split a line into two! But two points do!
 - ▶ One line does not split the plane into two! But two different lines do!
 - ▶ The line segment between two points is not unique!



Curves and surfaces

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
 - Surfaces
- ▼ Transformations
 - ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
 - ▼ Projective transformations
 - ▼ Graphics Pipeline
 - Transformations
 - Clipping
- ▼ Shading
 - Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
- ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB



Curves and surfaces

- ▶ Motivation
 - ▶ We can now represent the points of the plane or space by numbers (their coordinates)
 - ▶ How can we represent *nice* sets of points, e.g. a line in the plane or a plane in space?
 - ▶ We seek the answer to this in the Cartesian coordinate system
- ▶ Informally, the curves and surfaces are special subsets of space - i.e. they are sets of points
- ▶ How can we define these - usually infinite – sets?
 - ▶ **explicit:** $y = f(x) \rightarrow$ what happens when the curve folds onto itself?
 - ▶ **parametric:** $\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}, t \in \mathbb{R}$
 - ▶ **implicit:** $f(x, y) = 0$, for example: $x^2 + y^2 - 9 = 0$

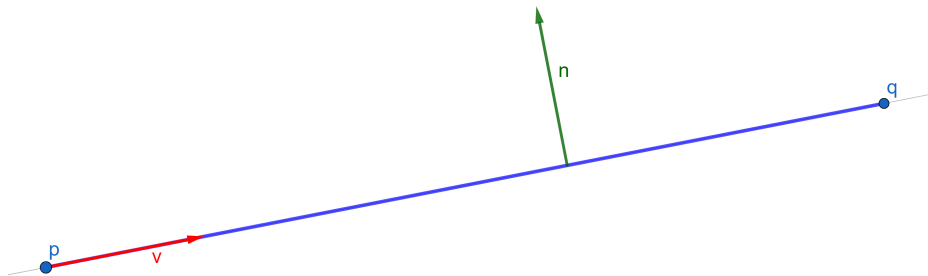


Implicit equation of line with a point and normal

- ▶ High school: $y = mx + b$. Problem: vertical lines!
- ▶ Let $\mathbf{p}(p_x, p_y)$ be a point on the line and $\mathbf{n} = [n_x, n_y]^T \neq \mathbf{0}$ a vector, a **normal** perpendicular to the line.
- ▶ All $\mathbf{x}(x, y)$ points of the plane that satisfy the following are exactly the points of the line:

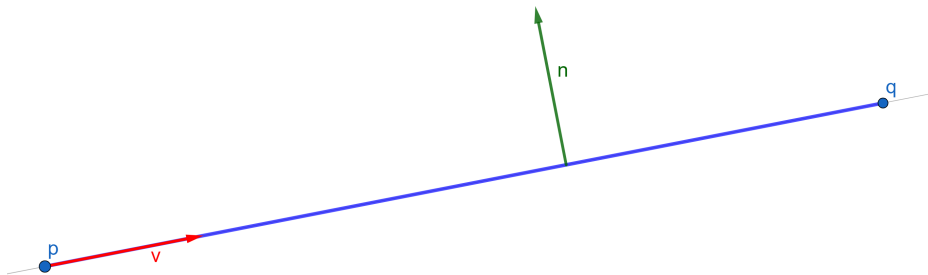
$$\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle = 0 \implies (x - p_x)n_x + (y - p_y)n_y = 0$$

- ▶ Two half-planes: $\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle < 0$ and $\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle > 0$



The homogeneous implicit equation of the line on the plane

- ▶ The equation $ax + by + c = 0$ is the implicit equation of the line on the plane
- ▶ In the previous representation, choosing $a = n_x$, $b = n_y$ and $c = -(p_x n_x + p_y n_y)$, $a^2 + b^2 \neq 0$ we get the implicit equation of the line going through \mathbf{p} , with normal \mathbf{n}
- ▶ If $a^2 + b^2 = 1$ then this is the **Hesse normal form**

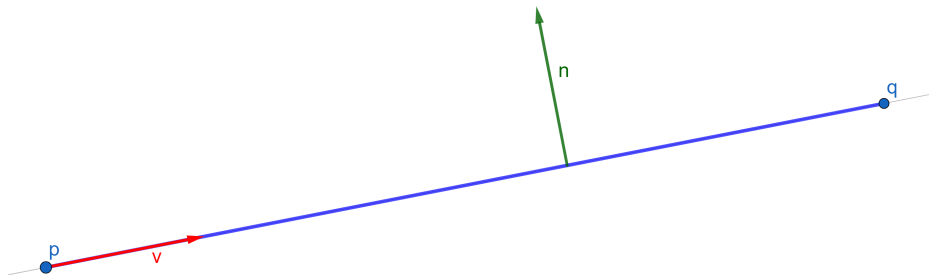


Homogeneous implicit equation with determinant

- ▶ Let $p(p_x, p_y)$ and $q(q_x, q_y)$ be two distinct points on the line. A point $x(x, y)$ belongs to the line if:

$$\begin{vmatrix} x & y & 1 \\ p_x & p_y & 1 \\ q_x & q_y & 1 \end{vmatrix} = 0$$

- ▶ This determinant is twice the signed area of the triangle spanned by x , p , q which vanishes whenever they are collinear.



Parametric equation of lines

- ▶ Let $\mathbf{p}(p_x, p_y, p_z)$ be a point on the line and $\mathbf{v} = [v_x, v_y, v_z]^T \neq \mathbf{0}$ a **direction vector** of the line (a vector parallel to the line):

$$\mathbf{x}(t) = \mathbf{p} + t\mathbf{v} = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} p_x + tv_x \\ p_y + tv_y \\ p_z + tv_z \end{bmatrix} \quad (t \in \mathbb{R})$$

- ▶ Let \mathbf{p} and \mathbf{q} be two points of the line. The previous form can be attained by setting $\mathbf{v} = \mathbf{q} - \mathbf{p}$:

$$\mathbf{x}(t) = \mathbf{p} + t(\mathbf{q} - \mathbf{p}) = \begin{bmatrix} (1-t)p_x + tq_x \\ (1-t)p_y + tq_y \\ (1-t)p_z + tq_z \end{bmatrix} \quad (t \in \mathbb{R})$$

- ▶ Note that the above is linear interpolation or the barycentric interpolation of two points.



Lines

- ▶ A line can be represented in the projective plane by the **line-coordinate** triplet $\mathbf{e} = [e_1, e_2, e_3]$. A point $\mathbf{x} = [x_1, x_2, x_3]^T$ belongs to the line iff

$$\langle \mathbf{e}, \mathbf{x} \rangle = e_1x_1 + e_2x_2 + e_3x_3 = 0$$

- ▶ This goes for the ideal line of the projective plane: $[0, 0, 1]$ are the line-coordinates of the ideal line, since all points of the form $[x_1, x_2, 0]$ will satisfy $\langle \mathbf{e}, \mathbf{x} \rangle = 0$

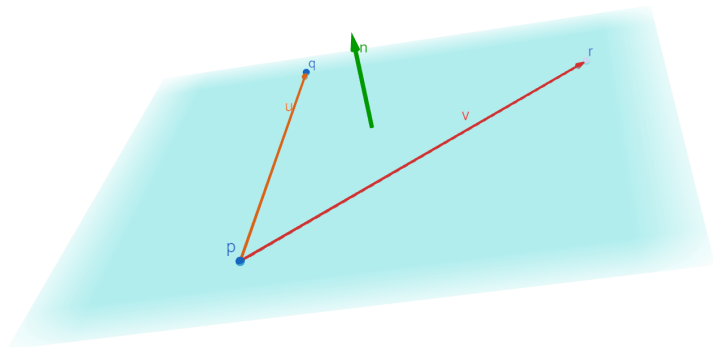


Implicit equation of the plane using a point and normal

- ▶ Let $\mathbf{p}(p_x, p_y, p_z)$ be a point on the plane and $\mathbf{n} = [n_x, n_y, n_z]^T$ be a vector perpendicular to the plane. Then \mathbf{x} is a point on the plane iff:

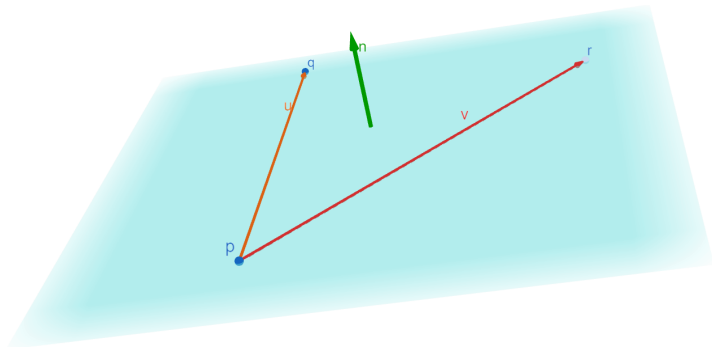
$$\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle = 0$$

- ▶ Half-spaces: $\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle < 0$, $\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle > 0$



Homogeneous implicit equation of the plane

- ▶ The implicit equation of the plane is in the form $ax + by + cz + d = 0$
- ▶ From the previous slide choosing $a = n_x$, $b = n_y$, $c = n_z$ and $d = -n_x p_x - n_y p_y - n_z p_z$ determines the plane going through \mathbf{p} , with \mathbf{n} perpendicular to it
- ▶ Hesse normal-form: $a^2 + b^2 + c^2 = 1$

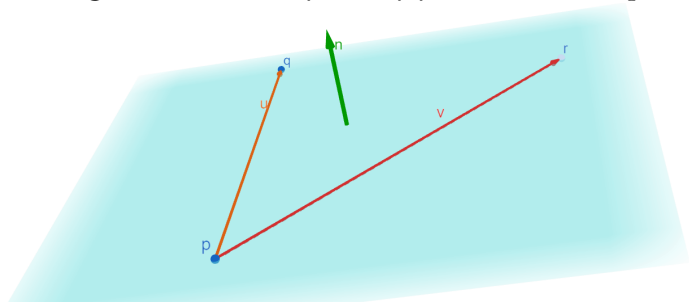


Homogeneous implicit equation with determinant

- Let $\mathbf{p}(p_x, p_y, p_z)$, $\mathbf{q}(q_x, q_y, q_z)$, $\mathbf{r}(r_x, r_y, r_z)$ be three, affinely independent points. Then x lies on the plane spanned by $\mathbf{p}, \mathbf{q}, \mathbf{r}$ iff

$$\begin{vmatrix} x & y & z & 1 \\ p_x & p_y & p_z & 1 \\ q_x & q_y & q_z & 1 \\ r_x & r_y & r_z & 1 \end{vmatrix} = 0$$

This determinant is the signed volume of a paralelepipedon with sides $\mathbf{p} - \mathbf{x}, \mathbf{q} - \mathbf{x}, \mathbf{r} - \mathbf{x}$.



Parametric equation of the plane – using three points

- ▶ Let $\mathbf{p}, \mathbf{q}, \mathbf{r}$ be three affinely independent points. Then all points \mathbf{x} belonging to the plane spanned by $\mathbf{p}, \mathbf{q}, \mathbf{r}$ can be expressed in the form

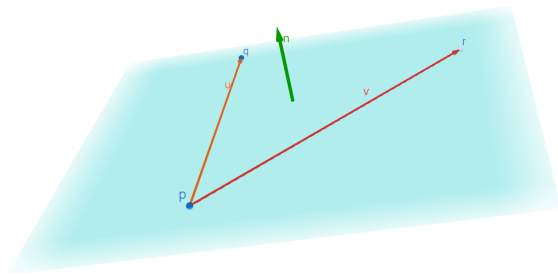
$$\mathbf{x}(s, t) = \mathbf{p} + s(\mathbf{q} - \mathbf{p}) + t(\mathbf{r} - \mathbf{p})$$

where $s, t \in \mathbb{R}$.

- ▶ This is a barycentric form, just like in the case of the parametric line through two points

$$\mathbf{x}(s, t) = (1 - s - t)\mathbf{p} + s\mathbf{q} + t\mathbf{r}$$

since $(1 - s - t) + s + t = 1$

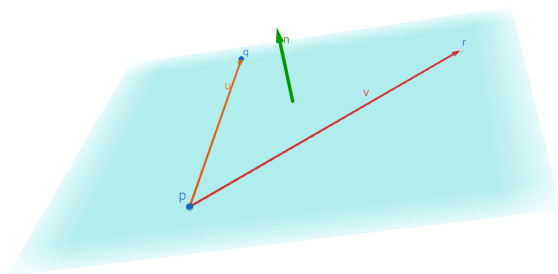


Parametric equation of the plane – using a point and spanning vectors

- ▶ Let p be a point of the plane and u, v two, linearly independent vectors in the plane

$$x(s, t) = p + su + tv$$

- ▶ We can get this from the previous one by setting $u = q - p$, $v = r - p$



Homogeneous form of a plane

- ▶ The **plane coordinates** are analogous to the definition of the line-coordinates. In the projective space the 4-tuple $\mathbf{s} = [s_1, s_2, s_3, s_4]$ defines a plane. All points $\mathbf{x} = [x_1, x_2, x_3, x_4]^T$ belong to the plane iff

$$\langle \mathbf{s}, \mathbf{x} \rangle = s_1x_1 + s_2x_2 + s_3x_3 + s_4x_4 = 0$$

- ▶ Some example planes in homogeneous form:
 - ▶ $[0, 0, 0, c]$ ideal plane
 - ▶ $[c, 0, 0, 0]$ the YZ plane
 - ▶ $[0, c, 0, 0]$ the XZ plane
 - ▶ $[0, 0, c, 0]$ the XY plane



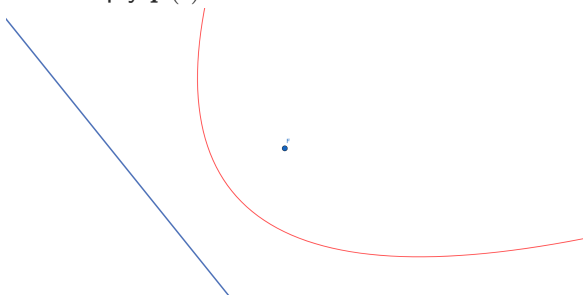
Curves and surfaces

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
 - Surfaces
- ▼ Transformations
 - ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
 - ▼ Projective transformations
 - ▼ Graphics Pipeline
 - Transformations
 - Clipping
- ▼ Shading
 - Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
- ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB



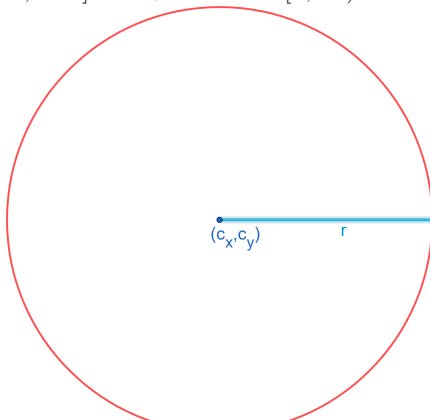
Parabola

- ▶ The parabola of focus point $(0, p)$ about the Y axis can be written
 - ▶ implicitly as $x^2 = 4py$,
 - ▶ explicitly as $y = \frac{x^2}{4p}$, $x \in \mathbb{R}$, and
 - ▶ parametrically as $\mathbf{p}(t) = [t, \frac{t^2}{4p}]^T$, $t \in \mathbb{R}$.
- ▶ How can we translate this, such that its focus point becomes $(0, p) + \mathbf{c}$?
- ▶ In implicit and explicit formulation one has to work the coordinates of the translation (c_x, c_y) into the formulation (e.g. $(x - c_x)^2 = 4p(y - c_y)$ in implicit)
- ▶ In parametric form it is simply $\mathbf{p}(t) + \mathbf{c}$



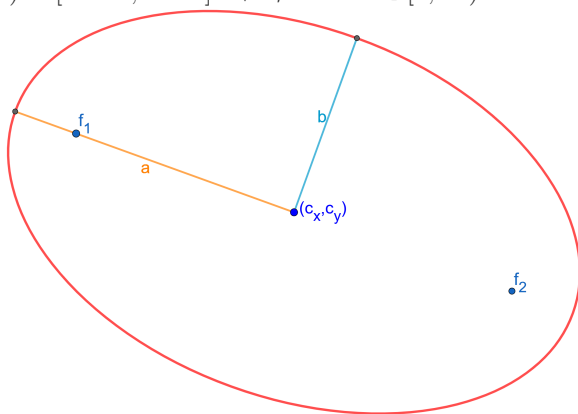
Circle

- ▶ Let us consider the circle of origin $\mathbf{c} \in \mathbb{E}^2$ and radius $r > 0$.
 - ▶ Implicit: $(x - c_x)^2 + (y - c_y)^2 = r^2$
 - ▶ Explicit: impossible to express the entire circle! However, a semi-circle is doable:
 $\mathbf{c} = \mathbf{0}, r = 1$, where $y = \pm\sqrt{1 - x^2}, x \in [-1, 1]$
 - ▶ Parametric: $\mathbf{p}(t) = r[\cos t, \sin t]^T + \mathbf{c}$, where $t \in [0, 2\pi)$



Ellipse

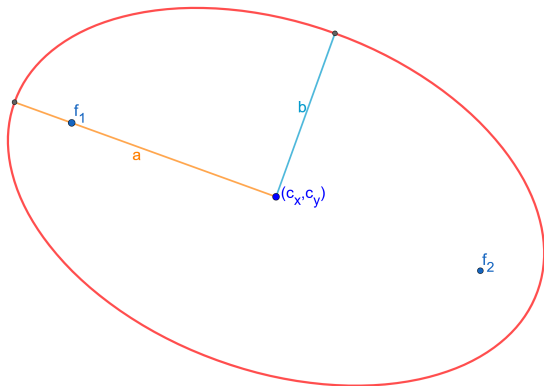
- ▶ The ellipse of center point $\mathbf{c} \in \mathbb{E}^2$ having the X and Y axes as its major and minor axes, with major and minor radii $a > 0$, $b > 0$, respectively, can be written as
 - ▶ Implicit: $\frac{(x-c_x)^2}{a^2} + \frac{(y-c_y)^2}{b^2} = 1$
 - ▶ Explicit: same deal as with the circle...
 - ▶ Parametric: $\mathbf{p}(t) = [a \cos t, b \sin t]^T + \mathbf{c}$, where $t \in [0, 2\pi)$



Ellipse

- ▶ How about having major and minor axes different from the axes of the coordinate system?
 - ▶ Implicit: seems kind of elaborate... (we will re-visit this during ray-surface intersections)
 - ▶ Parametric: a mere change of basis – let the directions of the new major and minor axes be denoted by k, l unit vectors. Then

$$p(t) = a \cos t k + b \sin t l + c, \quad t \in [0, 2\pi)$$

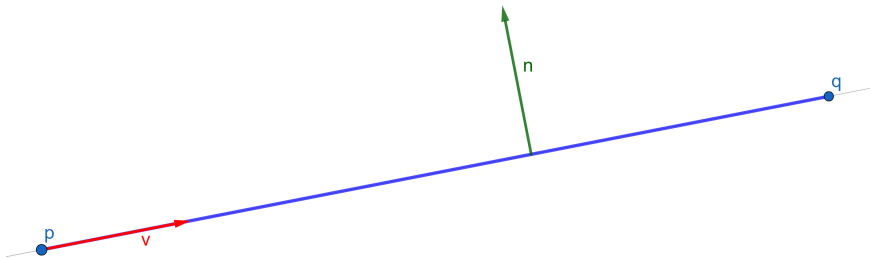


Line segment

- ▶ The line connecting the two points $\mathbf{a}, \mathbf{b} \in \mathbb{E}^3$ can be written parametrically as

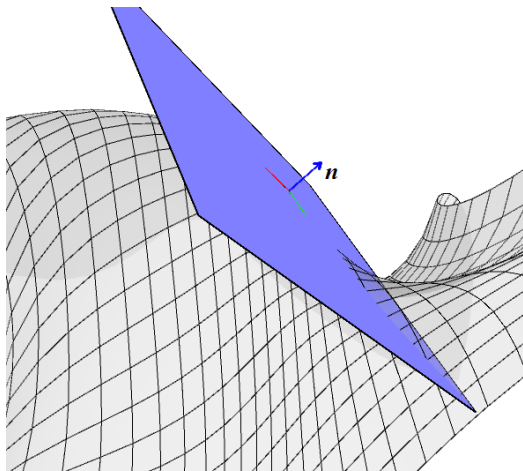
$$\mathbf{p}(t) = (1 - t)\mathbf{a} + t\mathbf{b}, \quad (t \in \mathbb{R})$$

- ▶ If $t \in [0, 1]$, then the above gives the line segment connecting \mathbf{a}, \mathbf{b} .



Representation of Surfaces

- ▶ Explicit: $z = f(x, y)$
- ▶ Implicit: $f(x, y, z) = 0$
- ▶ Parametric: $\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix},$
 $(u, v) \in [a, b] \times [c, d]$



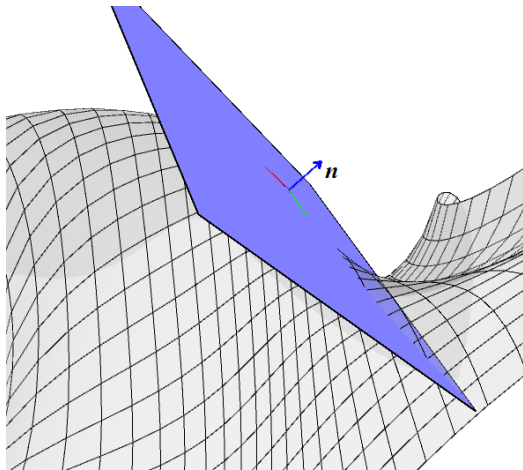
Surface normal

- ▶ Parametric surfaces:

$$\mathbf{n}(u, v) = \partial_u \mathbf{p}(u, v) \times \partial_v \mathbf{p}(u, v)$$

- ▶ Implicit surfaces: $\mathbf{n}(x, y, z) = \nabla f$, that is, the gradient

$$\nabla f = [f_x, f_y, f_z]^T .$$



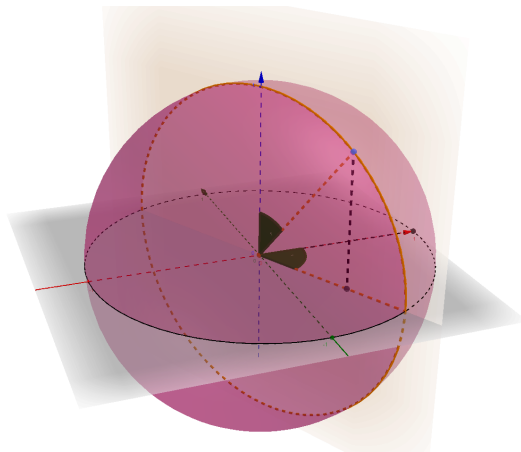
Sphere

The sphere of center $\mathbf{c}(c_x, c_y, c_z)$ and radius $r > 0$ can be written

- ▶ implicitly as $f(x, y, z) = (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 = 0$
- ▶ parametrically as

$$\mathbf{p}(u, v) = r \begin{bmatrix} \cos u \sin v \\ \sin u \sin v \\ \cos v \end{bmatrix} + \mathbf{c}$$

where $(u, v) \in [0, 2\pi) \times [0, \pi]$.

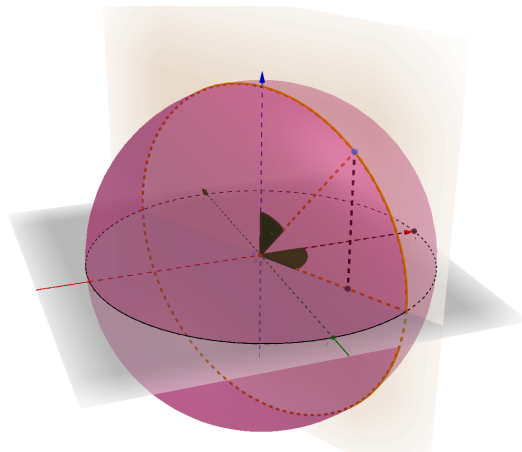


Sphere

Surface normal:

- ▶ Implicit: $\nabla f(x, y, z) = 2 \begin{bmatrix} x - c_x \\ y - c_y \\ z - c_z \end{bmatrix}$ (needs to be normalized)

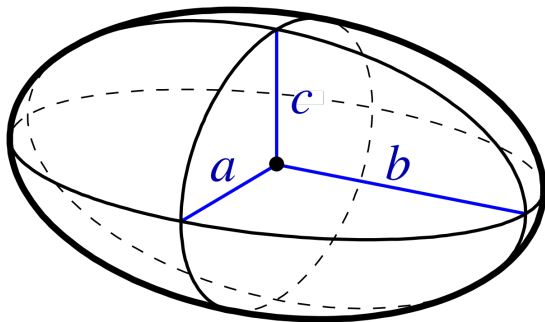
- ▶ Parametric: $\mathbf{n}(u, v) = \begin{bmatrix} \cos u \sin v \\ \sin u \sin v \\ \cos v \end{bmatrix}$



Ellipsoid

► Implicit: $\frac{(x-c_x)^2}{a^2} + \frac{(y-c_y)^2}{b^2} + \frac{(z-c_z)^2}{c^2} = 1$

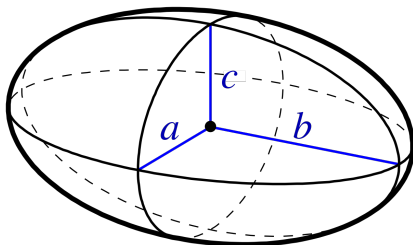
► Parametric: $\mathbf{p}(u, v) = \begin{bmatrix} a \cos u \sin v \\ b \sin u \sin v \\ c \cos v \end{bmatrix} + \mathbf{c} \quad (u, v) \in [0, 2\pi) \times [0, \pi]$



Ellipsoid

Surface normal:

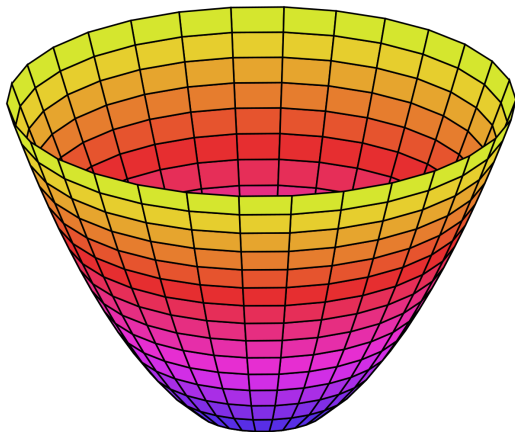
- ▶ implicit: $\nabla f = 2 \cdot \begin{bmatrix} \frac{x-c_x}{a^2} \\ \frac{y-c_y}{b^2} \\ \frac{z-c_z}{c^2} \end{bmatrix}$ (needs to be normalized)
- ▶ parametric: $\mathbf{n} = -\sin v \cdot \begin{bmatrix} bc \cos u \sin v \\ ac \sin u \sin v \\ ab \cos v \end{bmatrix}$ (needs to be normalized)



Simple paraboloid

- ▶ Explicit: $z = ax^2 + by^2$
- ▶ Implicit: $ax^2 + by^2 - z = 0$
- ▶ Parametric (derived from explicit):

$$\mathbf{p}(u, v) = \begin{bmatrix} u \\ v \\ au^2 + bv^2 \end{bmatrix} + \mathbf{c} \quad (u, v) \in \mathbb{R}^2$$

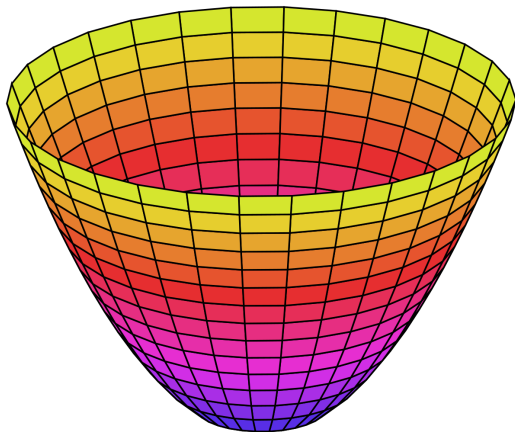


Simple paraboloid

Surface normal:

▶ Implicit: $\nabla f(x, y, z) = \begin{bmatrix} 2ax \\ 2by \\ -1 \end{bmatrix}$ (needs to be normalized)

▶ Parametric: $\mathbf{n}(u, v) = \begin{bmatrix} -2au \\ -2bv \\ 1 \end{bmatrix}$ (needs to be normalized)



A word of caution

- ▶ Most mathematical formulae treat axis z as the up direction
- ▶ This holds for the equations shown previously
- ▶ However, in computer graphics *up* is many times axis y .
- ▶ So swap the appropriate coordinates!
- ▶ Note that, this changes the right handedness of the coordinate system (to left handed).



Transformations

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
- Surfaces
- ▼ Transformations
- ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
- ▼ Projective transformations
- ▼ Graphics Pipeline
 - Transformations
 - Clipping
- Shading
- Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
- ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB



Motivation

- ▶ A transformation maps a point in one coordinate system to another point in another coordinate system
- ▶ We will need different coordinate systems more often than one would think! (E.g. object, world, eye, screen)
- ▶ The complex geometric entities of our scenes are made of smaller, simpler elements
- ▶ These elements need to be *brought together* → transformations
- ▶ The entities of our scenes can move around → transformations
- ▶ Our scene has to be rendered into a 2D image → transformations



Transformations

- ▶ Our expectations
 - ▶ defined for all points
 - ▶ map a point to a point, a line to a line, a plane to a plane
 - ▶ preserve coincidence relation
 - ▶ The image should be unique and *reversible* (provided the dimensions of the domain and the range are the same)

Remark

- ▶ Remember: our points are stored using their coordinates in an appropriate coordinate system
- ▶ Transformations work on these points, i.e. on the coordinates
- ▶ From now on, let us associate the points of the Euclidean space \mathbb{E}^3 (or plane \mathbb{E}^2) with the vectors of \mathbb{R}^3 (or \mathbb{R}^2) by setting an origin $\mathbf{o} \in \mathbb{E}^3$, and to $\mathbf{p} \in \mathbb{E}^3$ let $\mathbf{p} = \mathbf{p} - \mathbf{o}$ be its vector



Linear mappings

- ▶ The mapping $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ is linear, iff for $\forall \mathbf{a}, \mathbf{b} \in \mathbb{R}^3$ and $\lambda \in \mathbb{R}$
 - ▶ $\phi(\mathbf{a} + \mathbf{b}) = \phi(\mathbf{a}) + \phi(\mathbf{b})$ (additivity)
 - ▶ $\phi(\lambda\mathbf{a}) = \lambda\phi(\mathbf{a})$ (homogeneity)
- ▶ Reminder: a linear mapping $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ can be represented with an $A \in \mathbb{R}^{m \times n}$ matrix:
 $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$, $\mathbf{x} \in \mathbb{R}^n$.



Projective and affine transformations

- ▶ Transformations of the projective space/plane that map lines to lines are the **projective transformations**
- ▶ A transformation that preserves lines and parallelism (maps parallel lines to parallel lines) is an **affine transformation**
- ▶ Remark: affine transformations cannot map points at infinity to finite points and vice versa. Projective transformation, on the other hand, can.



Properties

- ▶ The projective and affine transformations form an algebraic group with the operation of concatenation, i.e.
 - ▶ the group is closed under concatenation
 - ▶ concatenation is associative
 - ▶ there exists an identity element (identity transformation)
 - ▶ each transform has an inverse (provided they preserve the dimension)
- ▶ Attention: this group is **not commutative!**



Affine transformations

- ▶ Every affine transformation can be written as a linear transformation followed by a translation
- ▶ That is, they can be represented by a $\mathbf{A} \in \mathbb{R}^{3 \times 3}$ matrix and $\mathbf{b} \in \mathbb{R}^3$ vector:

$$\varphi(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$$

- ▶ Using homogeneous coordinates, we can use a single 4×4 matrix (actually a 3×4 is enough) to represent the transformation $\varphi(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$:

$$\begin{bmatrix} \mathbf{A} & \mathbf{b} \\ [0, 0, 0] & 1 \end{bmatrix} \in \mathbb{R}^{4 \times 4}$$



Affine transformations

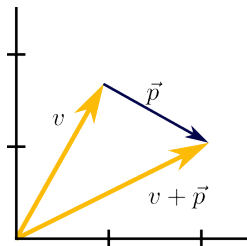
Barycentric coordinates are invariant under affine transformations.

Proof: let α_i be the barycentric coordinates of \mathbf{x} wrt. \mathbf{x}_i , then

$$\begin{aligned}\varphi(\mathbf{x}) &= \varphi\left(\sum_{i=0}^n \alpha_i \mathbf{x}_i\right) \\ &= \sum_{i=0}^n \varphi(\alpha_i \mathbf{x}_i) \\ &= \sum_{i=0}^n \alpha_i \varphi(\mathbf{x}_i)\end{aligned}$$



Translation



- ▶ Translate all points with the vector \mathbf{d} :

$$\mathbf{x}' = \mathbf{x} + \mathbf{d}$$

- ▶ We denote the matrix by $\mathbf{T}(d_x, d_y, d_z)$. Using homogeneous coordinates:

$$\mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Translation

- ▶ Since

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot x + 0 \cdot y + 0 \cdot z + 1 \cdot d_x \\ 0 \cdot x + 1 \cdot y + 0 \cdot z + 1 \cdot d_y \\ 0 \cdot x + 0 \cdot y + 1 \cdot z + 1 \cdot d_z \\ 1 \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{bmatrix}$$

- ▶ A commutative sub-group of the group of affine transformations
- ▶ The inverse of $\mathbf{T}(a, b, c)$ is $\mathbf{T}^{-1}(a, b, c) = \mathbf{T}(-a, -b, -c)$



Rotation

- ▶ Rotation in the XY plane by θ :

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta.$$

- ▶ In matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = x \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix} + y \begin{bmatrix} -\sin \theta \\ \cos \theta \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- ▶ Similar in the XZ and YZ planes so...



Rotations about cardinal axis

Z axis

$$\mathbf{R}_Z(\theta) = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

Y axis

$$\mathbf{R}_Y(\theta) = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

X axis

$$\mathbf{R}_X(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $c = \cos \theta$ and $s = \sin \theta$.

- ▶ The rotations along the same axis form a commutative sub-group
- ▶ They are linear transformations, 3×3 matrices suffice to represent them
- ▶ The inverse of rotation $\mathbf{R}_X(\theta)$ is $\mathbf{R}_X(-\theta)$
- ▶ Since $\mathbf{R}_X(\theta) \cdot \mathbf{R}_X(\varphi) = \mathbf{R}_X(\theta + \varphi)$



Yaw, pitch, roll

An arbitrary orientation can be attained by doing 3 rotations sequentially:

$$\mathbf{R}(\alpha, \beta, \gamma) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma & -\sin \gamma \\ 0 & \sin \gamma & \cos \gamma \end{bmatrix}$$

- ▶ Store the rotations about the three axes: Y (**yaw**), X (**pitch**), and Z (**roll**)
- ▶ Common in flight dynamics and robotics
- ▶ Same as the *three matrices multiplied* seen earlier
- ▶ Most of the graphics APIs have support for this

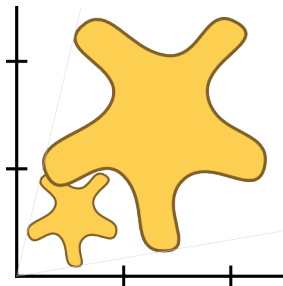


Rigid body transformations

- ▶ The translations and rotations are rigid body transformations
- ▶ They preserve distances and angles
- ▶ Their determinant is 1
- ▶ Important in physics simulations



Scaling



- ▶ Shrink and expand the object independently along the x, y, z axes
- ▶ Matrix:

$$\mathcal{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Special case: Mirroring

- ▶ If at least one of $s_x, s_y, s_z < 0$
 - ▶ if exactly 1 is negative: mirror the point to the plane spanned by the axes of the non-negative coefficients
 - ▶ if exactly 2 are negative: mirror to the third axis
 - ▶ if all 3 are negative: mirror to the origin
- ▶ Attention: if $s_x s_y s_z < 0$, the **winding** (handedness) of our coordinate system changes!

Winding

- ▶ Using the basis vectors $\mathbf{i}, \mathbf{j}, \mathbf{k}$ of the canonic base

$$\varphi(\mathbf{p}) = \varphi(x\mathbf{i} + y\mathbf{j} + z\mathbf{k}) = x\varphi(\mathbf{i}) + y\varphi(\mathbf{j}) + z\varphi(\mathbf{k})$$

- ▶ If the determinant of a transformation matrix is negative, then it changes the winding of the coordinate system (left handed \rightarrow right handed, and vice versa)



Special case: Orthographic projection

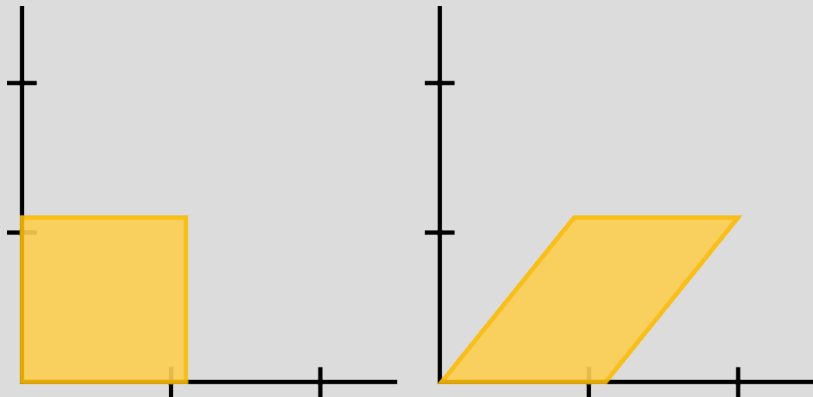
- ▶ If one of s_x, s_y, s_z is zero:
 - ▶ 1 zero: orthogonal projection to the plane spanned by the other two axes
 - ▶ 2 zeros: project onto the non-zero axis
 - ▶ all 3 are zeros: project to the 'origin'...
- ▶ Remark: the determinant is zero! \Rightarrow there is no inverse!



Shear

Example

Consider a deck of cards:



Shear

Shear of the x, y coordinates proportional to the z coordinates:

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In general:

$$\mathbf{N} = \begin{bmatrix} 1 & a & b & 0 \\ 0 & 1 & c & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Change of basis

- ▶ Let $\mathbf{i}, \mathbf{j}, \mathbf{k}$ be the canonical orthonormal base, and $\mathbf{u}, \mathbf{v}, \mathbf{w}$ the new orthonormal basis vectors (wrt. $\mathbf{i}, \mathbf{j}, \mathbf{k}$)
- ▶ What will be the new $\mathbf{x}' = [x', y', z']^T$ coordinates (in the new base) of the point $\mathbf{x} = [x, y, z]^T$ (in the old base)?

$$\mathbf{x} = [\mathbf{u}^T, \mathbf{v}^T, \mathbf{w}^T]\mathbf{x}' = B\mathbf{x}' \implies \mathbf{x}' = B^{-1}\mathbf{x} = B^T\mathbf{x}$$

- ▶ Since the bases are orthonormal $M = B^{-1}$ is the following: $M = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- ▶ If the origin has changed to \mathbf{c} , then the matrix of the change of basis is $M = B^{-1}T(-c_x, -c_y, -c_z)$

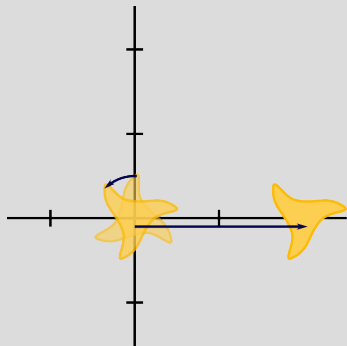


Commutativity counterexample

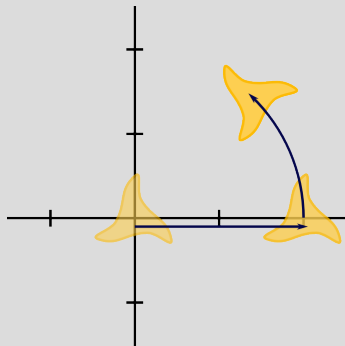
Matrix multiplication (and concatenation of transformations) is not commutative, i.e.

$$\mathbf{ABv} \neq \mathbf{BAv}$$

Rotation followed by translation

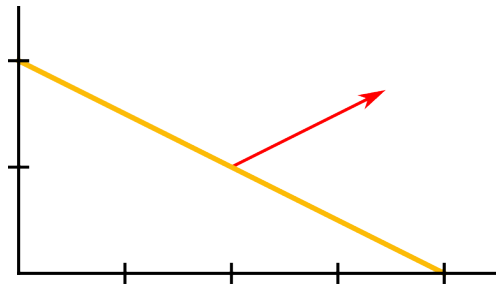
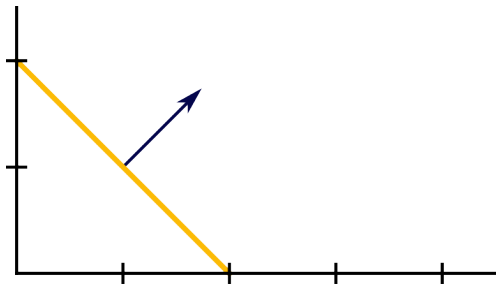


Translation followed by rotation



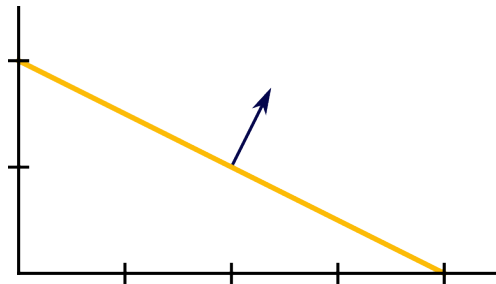
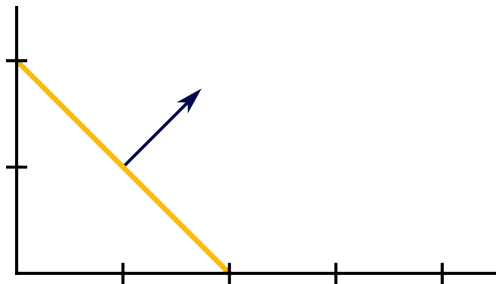
Transformation of normal vectors

- ▶ Consider the following example:



Transformation of normal vectors

- ▶ Consider the following example:



- ▶ Let us consider the implicit equation of the tangent plane:

$$\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle = 0$$

- ▶ Then for an arbitrary, invertible \mathbf{A} matrix:

$$\langle \mathbf{A}^{-1} \mathbf{A}(\mathbf{x} - \mathbf{p}), \mathbf{n} \rangle = 0$$

$$\left(\mathbf{A}^{-1} \mathbf{A}(\mathbf{x} - \mathbf{p}) \right)^T \cdot \mathbf{n} = 0$$

$$\left(\mathbf{A}(\mathbf{x} - \mathbf{p}) \right)^T \left(\mathbf{A}^{-1} \right)^T \mathbf{n} = 0$$

- ▶ Using the properties of the dot product and matrix multiplication

$$\langle \mathbf{A}(\mathbf{x} - \mathbf{p}), \left(\mathbf{A}^{-1} \right)^T \mathbf{n} \rangle = 0$$

- ▶ Thus, transform the normal vectors by the inverse transpose of \mathbf{A} (eg.: \mathbf{A}^{-T})!



General case

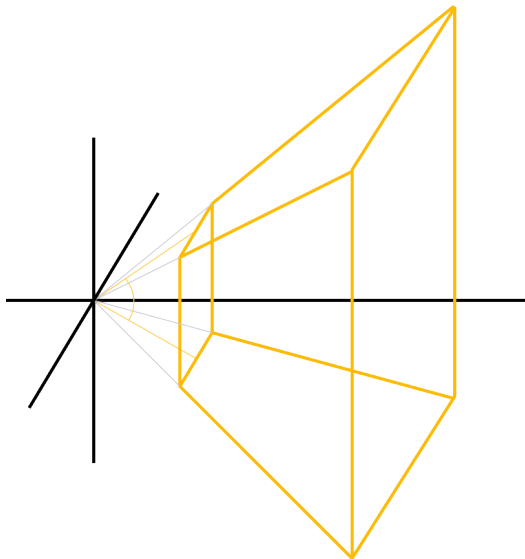
If the last row of the transformation matrix is not $[0, 0, 0, 1]$, then it is a **linear homogeneous transformation** (a transformation that is linear in the homogeneous space, but not necessarily linear in the Euclidean) This is what we need for perspective projections!

Perspective transform

- ▶ Projection from a point onto a plane (central projection)
- ▶ Let that point be the origin, and the plane be parallel to the XY plane at $z = d$
- ▶ Viewing frustum



Perspective transform



Homogeneous division

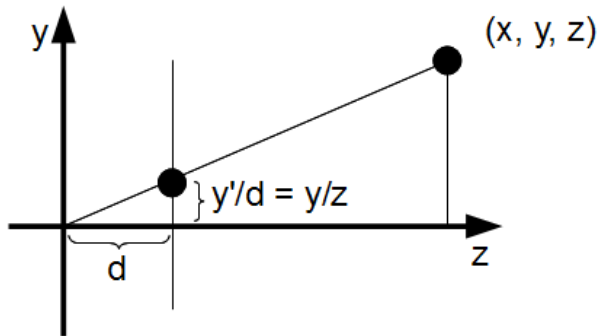
- ▶ If \mathbf{M} is a *real* projective transformation, its last row is not $[0, 0, 0, 1]^T$, i.e. after

$$[x, y, z, w] = \mathbf{M}\mathbf{v} \implies w \neq 1$$

- ▶ We need to divide all the coordinates by the last coordinate w (provided $w \neq 0$) – this is the homogeneous division



Central projection



$$x' = \frac{x}{z}d, \quad y' = \frac{y}{z}d, \quad z' = \frac{z}{z}d = d$$

Central projection

- ▶ Let the origin be the point of projection, the plane onto which we project is the $z = d$ plane. Then the matrix is:

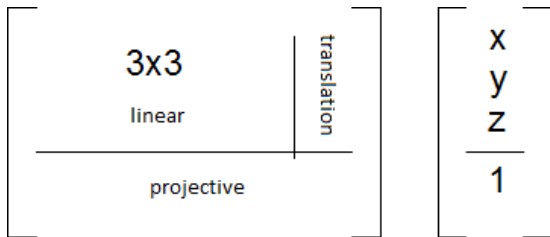
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

- ▶ We get the same formulas as before after we apply the homogeneous division with $\frac{z}{d}$



Transformation matrices

In summary, transformations will be 4×4 matrices:



Attention: points and vectors are column vectors, therefore the **vectors are on the right side of matrices** during this course.



Graphics Pipeline

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
- Surfaces
- ▼ Transformations
- ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
- ▼ Projective transformations
- ▼ Graphics Pipeline
 - Transformations
 - Clipping
- Shading
- Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
- ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB



Raytracing vs incremental image synthesis

Raycasting

For every pixel on the screen:

For every object (geometry) in the scene:

Does the ray of the pixel intersects the object?

Incremental image synthesis

For every object (geometry) in the scene:

For every pixel on the screen:

Does the projection of the object contain the pixel?



Raytracing vs incremental image synthesis

- ▶ Advantages of raytracing:
 - ▶ Wide range of geometries (almost the only restriction: carry out ray-surface intersection efficiently)
 - ▶ Easy to implement using recursion
 - ▶ It handles light as particles – effects related to the particle nature of light are simple to implement
- ▶ Disadvantages of raytracing:
 - ▶ Computationally expensive (intersection test with all objects, for every pixel!)
 - ▶ Global algorithm: in order to compute the color of a single pixel, you need access to the description of the entire scene
 - ▶ It is harder to simulate the wave properties of light
 - ▶ Slow for real-time applications



Graphics pipeline – Motivation

- ▶ Speed up image synthesis by changing the loop ordering in the raycast/raytrace pseudo-code
- ▶ In the new loop order the efficiency of the algorithms greatly depends on how easy it is to determine whether a pixel is inside the projection of a geometry or not
- ▶ Thus the range of appropriate geometries is much more narrow \Rightarrow in practice, this means linear geometries (e.g. lines, triangles), so called **primitives**
- ▶ Nonlinear geometries have to be approximated (**tessellated**) by linear elements



Real-time graphics

- ▶ Avoid unnecessary computations: pre-filter the geometries so that we cull (throw away) every geometry that is guaranteed to **not** end up on the screen (because e.g. they are behind us, etc.)
- ▶ Let us carry out every operation in a coordinate system that is a best fit to the problem
- ▶ Use the results of the previous computations to speed up things
- ▶ **Coherence**: instead of pixels, base computations on bigger elements – **primitives**
- ▶ Use **object space precision** (instead of pixel precision)
- ▶ **Clipping**: cull every part of the primitives that lie outside the screen
- ▶ **Incremental synthesis**: use the information computed for the bigger elements to resolve the shading and occlusion problem (e.g. the slope of the primitives w.r.t. the x and y coordinates)



Graphics Pipeline

1. **Transform** every single vertex into the **Normalized Device Coordinates (NDC)**
2. Assemble every primitive from the transformed coordinates
3. **Clip** every primitive to the viewport window
4. **Rasterize** all the primitives creating fragments
5. **Interpolate** the vertex attributes for every single fragment
6. Colorize every single fragment using textures and a **shading model**
7. Decide for each fragment whether it is visible and how, and blend the value accordingly



Graphics Pipeline

1. **Transform** every single vertex into the **Normalized Device Coordinates (NDC)**
2. Assemble every primitive from the transformed coordinates
3. **Clip** every primitive to the viewport window
4. **Rasterize** all the primitives creating fragments
5. **Interpolate** the vertex attributes for every single fragment
6. Colorize every single fragment using textures and a **shading model**
7. Decide for each fragment whether it is visible and how, and blend the value accordingly



Graphics Pipeline

1. **Transform** every single vertex into the **Normalized Device Coordinates (NDC)**
2. Assemble every primitive from the transformed coordinates
3. **Clip** every primitive to the viewport window
4. **Rasterize** all the primitives creating fragments
5. **Interpolate** the vertex attributes for every single fragment
6. Colorize every single fragment using textures and a **shading model**
7. Decide for each fragment whether it is visible and how, and blend the value accordingly



Graphics Pipeline

1. **Transform** every single vertex into the **Normalized Device Coordinates (NDC)**
2. Assemble every primitive from the transformed coordinates
3. **Clip** every primitive to the viewport window
4. **Rasterize** all the primitives creating fragments
5. **Interpolate** the vertex attributes for every single fragment
6. Colorize every single fragment using textures and a **shading model**
7. Decide for each fragment whether it is visible and how, and blend the value accordingly



Graphics Pipeline

1. **Transform** every single vertex into the **Normalized Device Coordinates (NDC)**
2. Assemble every primitive from the transformed coordinates
3. **Clip** every primitive to the viewport window
4. **Rasterize** all the primitives creating fragments
5. **Interpolate** the vertex attributes for every single fragment
6. Colorize every single fragment using textures and a **shading model**
7. Decide for each fragment whether it is visible and how, and blend the value accordingly



Graphics Pipeline

1. **Transform** every single vertex into the **Normalized Device Coordinates (NDC)**
2. Assemble every primitive from the transformed coordinates
3. **Clip** every primitive to the viewport window
4. **Rasterize** all the primitives creating fragments
5. **Interpolate** the vertex attributes for every single fragment
6. Colorize every single fragment using textures and a **shading model**
7. Decide for each fragment whether it is visible and how, and blend the value accordingly



Graphics Pipeline – How do we optimize?

1. Transform every single vertex into the Normalized Device Coordinates (NDC)
2. Assemble every primitive from the transformed coordinates
3. Clip every primitive to the viewport window
4. Rasterize all the primitives creating fragments
5. Interpolate the vertex attributes for every single fragment
6. Colorize every single fragment using textures and a shading model
7. Decide for each fragment whether it is visible and how, and blend the value accordingly



Graphics Pipeline – parallelization

1. Transform **every single vertex** into the Normalized Device Coordinates (NDC)
2. Assemble **every primitive** from the transformed coordinates
3. Clip **every primitive** to the viewport window
4. Rasterize **all the primitives** creating fragments
5. Interpolate the vertex attributes for **every single fragment**
6. Colorize **every single fragment** using textures and a shading model
7. Decide **for each fragment** whether it is visible and how, and blend the value accordingly



Graphics Pipeline – parallelization stages

1. Transform **every single vertex** into the Normalized Device Coordinates (NDC)
2. Assemble **every primitive** from the transformed coordinates
3. Clip **every primitive** to the viewport window
4. Rasterize **all the primitives** creating fragments
5. Interpolate the vertex attributes for **every single fragment**
6. Colorize **every single fragment** using textures and a shading model
7. Decide **for each fragment** whether it is visible and how, and blend the value accordingly



Comparison

Ray-tracing

- ▶ performed per pixel
- ▶ anything can be a geometry as long as it can be intersected by a ray
- ▶ reflection, refraction, shadows are inherently part of the computation
- ▶ occlusion resolution is trivial
- ▶ many rays per pixel: expensive

Incremental image synthesis

- ▶ performed per primitive
- ▶ anything that is not a primitive has to be approximated
- ▶ individual algorithms for each effect
- ▶ several algorithms for occlusion tests
- ▶ orders of magnitude less computation



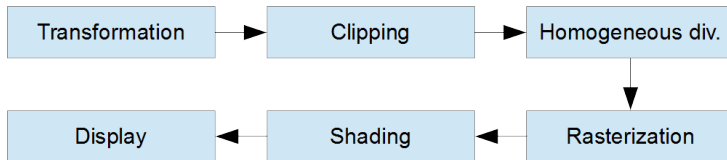
Graphics pipeline

- ▶ A *pipeline* is a chain of data processing elements, arranged such that the input of stage s_i is the output of the previous stage s_{i-1} , and the output of s_i is the input of s_{i+1}
- ▶ If one can decompose a problem into an n -stage pipeline (each stage taking roughly the same time), n elements can be processed per unit time – after the initial startup and final pass-through
- ▶ The *graphics pipeline* is a set of operations that we have to carry out on *primitives* in order to render them to the screen
- ▶ This is accompanied by several change of basis transformations so that every operations is carried out as efficiently, as possible



Graphics pipeline

- ▶ General overview of steps:
 - ▶ Transformations
 - ▶ Clipping
 - ▶ Homogeneous division
 - ▶ Rasterization and interpolation
 - ▶ Shading
 - ▶ Display



- ▶ The output of the graphics pipeline is a picture: a 2D array consisting of colors



Graphics pipeline inputs

- ▶ The **geometric model** of the scene (=list of primitives to be drawn)
- ▶ **Optical model** of the scene geometries
- ▶ The attributes of the **virtual camera** (point of view and viewing frustum)
- ▶ Screen boundaries



Transformations

- ▶ Transformations operate on the vertices of the primitives
- 1. **Model** coordinate system
⇒ **transformations**: move, rotate, ect. ⇒
- 2. **World** coordinate system
⇒ **'lookAt'** ⇒
- 3. **Camera** coordinate system
⇒ **projective** transformation ⇒
- 4. **Normalized device** coordinate system
⇒ **orthographic** projection ⇒
- 5. **Screen** coordinate system



Coordinate systems

- ▶ **Model CS:** The own, individual coordinate system of each object.
- ▶ **World CS:** The actual CS of the world (scene), where every object has its own position. In general, we think in terms of the world CS.
- ▶ **Camera CS:** A CS with origin at the camera position and axes aligned to the camera attributes.
- ▶ **Normalized device CS:** The GPU's internal CS, $[-1, 1] \times [-1, 1] \times [-1, 1]$ or $[-1, 1] \times [-1, 1] \times [0, 1]$.
- ▶ **Screen CS:** A left-handed CS, conforming to the current display properties. Units along the axes are pixels. The origin is at the top-left corner.



Model (world) transformation

- ▶ It should transform the object from its own CS into the common, world CS
- ▶ Generally, it's unique for every object
- ▶ Almost always a sequence of affine transformations, represented by a single 4×4 matrix
- ▶ We call these **world** or **model** in our code



View (camera) transformation

- ▶ Aligns the world CS with the camera
- ▶ This is translation followed by a change-of-basis transformation, represented by a single 4×4 matrix
- ▶ In our code: this is the **view** or **camera** matrix



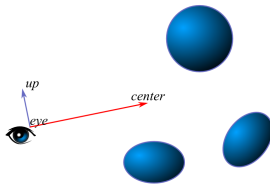
View (camera) transformation

- ▶ It can be specified similarly to the ray-casting case:
eye, center, up
- ▶ The camera CS X, Y, Z axes are then:

$$w = \frac{\mathit{eye} - \mathit{center}}{|\mathit{eye} - \mathit{center}|}$$

$$u = \frac{\mathit{up} \times w}{|\mathit{up} \times w|}$$

$$v = w \times u$$



View (camera) transformation

- Let the origin be at **eye**, with axes ***u***, ***v***, ***w***:

$$T_{View} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sidenote: Orthographic projection

A simple orthogonal projection to the *XY* plane would be

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

Instead, however, let's talk about perspective projection

Projection – perspective

- ▶ Actually, this is more than just a simple central projection
- ▶ First, we need the camera projection properties:
 - ▶ horizontal and vertical field-of-view (fov_x , fov_y) or either of them and the *aspect ratio* of the width and height of the display
 - ▶ distance of *near* and *far* planes

$$T_{Projection} = \begin{bmatrix} 1 / \tan \frac{fov_x}{2} & 0 & 0 & 0 \\ 0 & 1 / \tan \frac{fov_y}{2} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot near \cdot far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

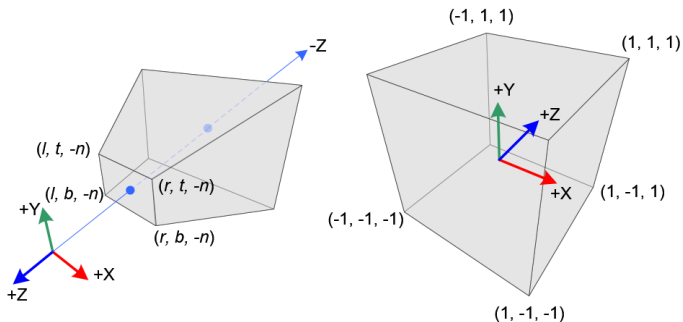
How do we get such a transformation?



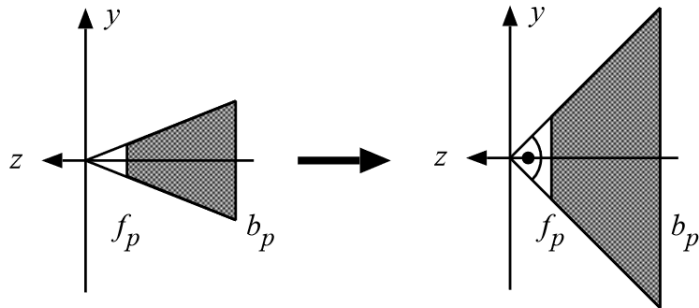
Projection – perspective

1. Transform the space such that the projection defined by $(\text{fov}_x, \text{fov}_y)$ turns into $(\frac{\pi}{2}, \frac{\pi}{2})$ (this is a simple scale)
2. Carry out the central projection (this is the actual projective transformation)
3. Map the *near* and *far* planes to $z = -1$ and $z = 1$, respectively

Reminder: at this stage the origin is the camera position!

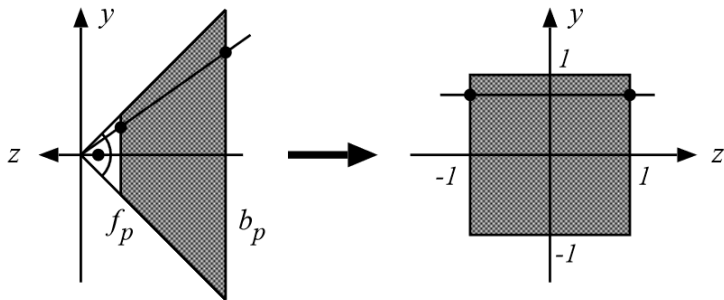


Projection step 1: normalize frustum



$$T_{Projection_1} = \begin{bmatrix} 1 / \tan \frac{fov_x}{2} & 0 & 0 & 0 \\ 0 & 1 / \tan \frac{fov_y}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection step 2: make lines of central projection parallel



$$T_{Projection_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot near \cdot far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Central projection through the origin

- ▶ Central projection along the Z axis, onto a plane parallel to XY , at distance d from the origin:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

- ▶ We project onto the $d = -1$ plane, so $z \in [-far, -near]$.

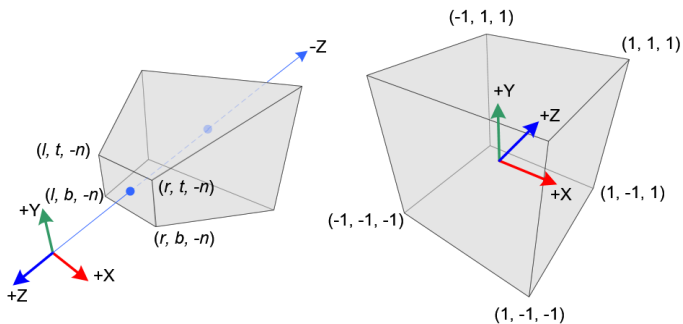


Transform to Normalized Device Coordinates (NDC) (OpenGL)

- ▶ Map the *near* and *far* planes to $z = -1$ and $z = 1$

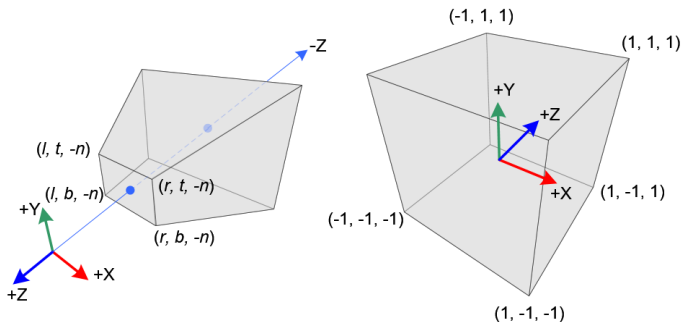
$$z \mapsto \frac{z \cdot -\frac{far+near}{far-near} - \frac{2 \cdot near \cdot far}{far-near}}{-1 \cdot z}$$

- ▶ The last division is the homogeneous division which is done by the GPU



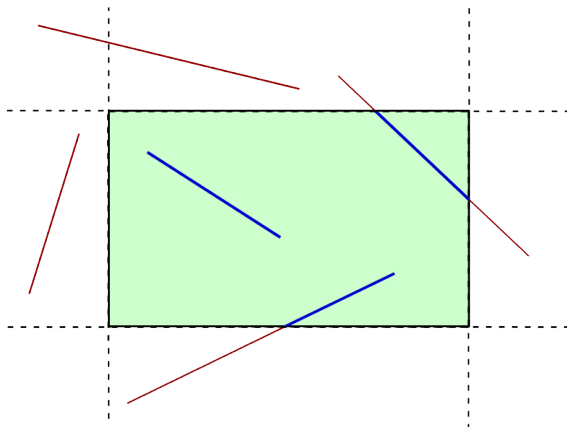
Transform to Normalized Device Coordinates (NDC) (OpenGL)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot near \cdot far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



Clipping

- ▶ Clipping has to be done before homogeneous division
- ▶ For example, consider a line segment starting in front and ending behind the camera
- ▶ The transformed (projected) line segment is **not** the connection of the projected endpoints!



Clipping using homogeneous coordinates

- ▶ Let $[x_h, y_h, z_h, w]^T = T_{projall} \cdot [x, y, z, 1]^T$
- ▶ If the projected coordinates should be within $[-1, 1]$, then for $w > 0$ we get

$$-w < x_h < w$$

$$-w < y_h < w$$

$$-w < z_h < w$$

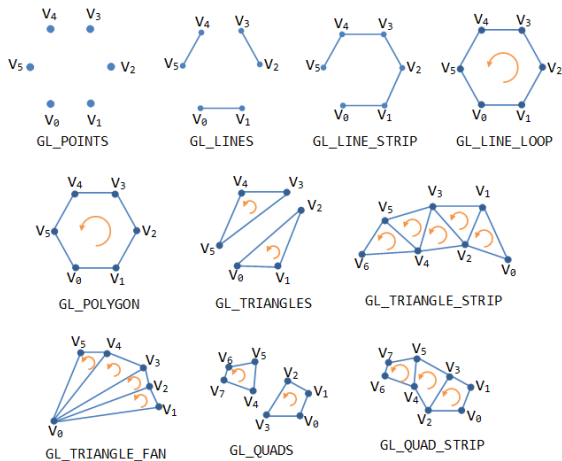
as a clipping condition.

- ▶ If any of the above containments do not hold, the primitive of the vertex should be clipped



Rasterization

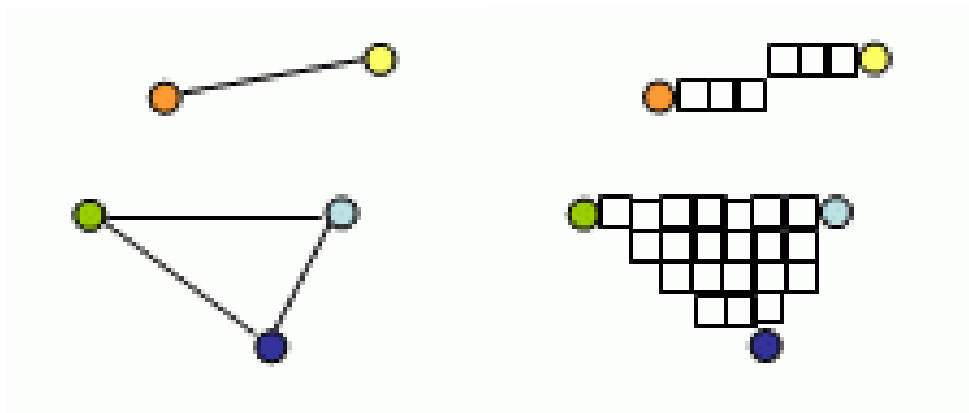
- ▶ For each clipped primitive: **discretize** the geometry
- ▶ Every pixel-worth chunk of the geometry becomes a **fragment**
- ▶ Using the barycentric coordinates of each fragment (with respect to the 3 vertices of the triangle), we **interpolate** every attribute for the fragments (not just position, but color, normals, etc. as well)



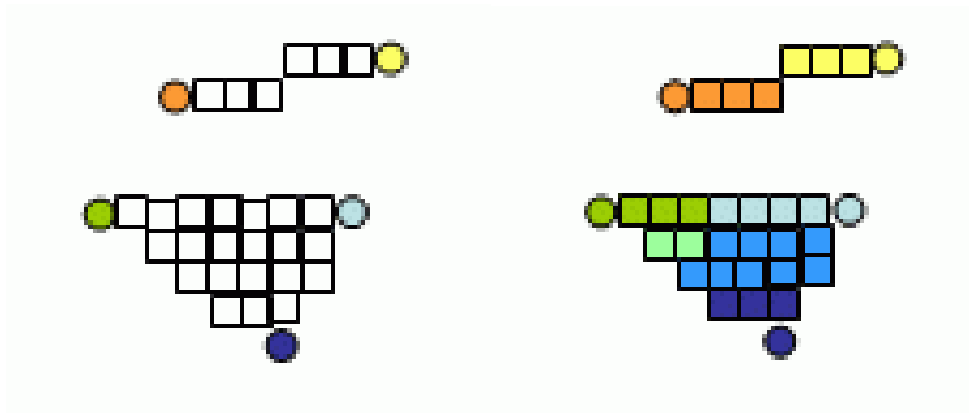
OpenGL Primitives



Rasterization of primitives

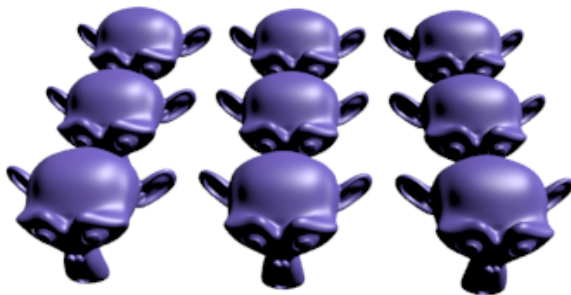


Rasterization of primitives



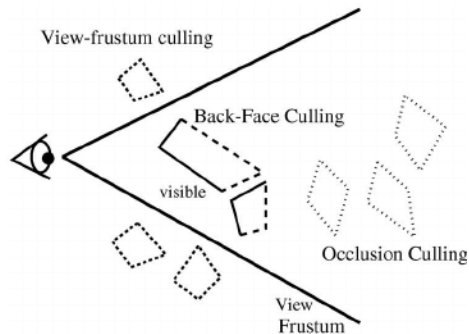
Shading

More on this later



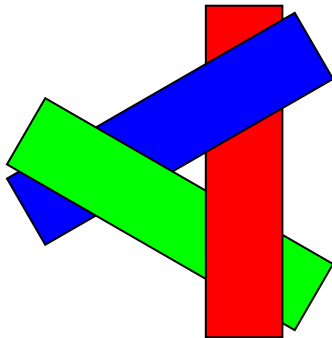
Occlusion

- ▶ Occlusion resolution is important
- ▶ The real-time industry went with the z-buffer
- ▶ But let us see some other approaches too



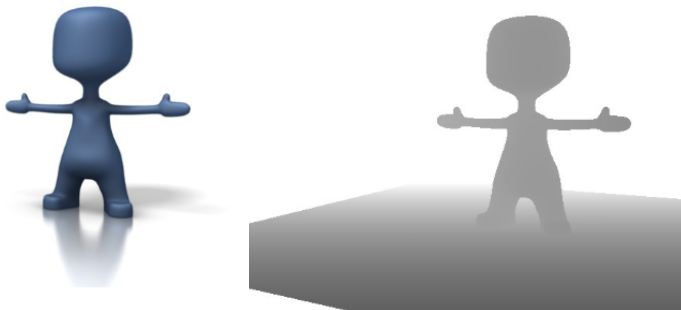
Painter algorithm

- ▶ Order the objects from farthest to closest
- ▶ Draw things starting from the farthest
- ▶ Problem: there is no guaranteed ordering



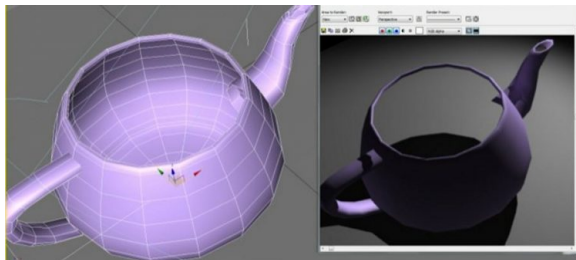
Z-buffer

- ▶ Allocate a secondary buffer (**depth buffer**) to store depth values for every pixel
- ▶ When determining **occlusion** for a fragment: check if the fragment's depth is closer than the distance stored for the pixel
- ▶ If closer: write the color to the framebuffer and update Z-buffer for the pixel (must be an atomic operation)
- ▶ Else: **discard** the fragment



Optimization – Backface culling

- ▶ May be used for the rendering of a single, closed convex object possessing a volume
- ▶ Do not render triangles that are not facing us
- ▶ Not facing us = its vertices are either clockwise or counter-clockwise (depends on convention)
- ▶ Used in general too: you can roughly shave off half of the triangles (but you need volume-like geometries!)

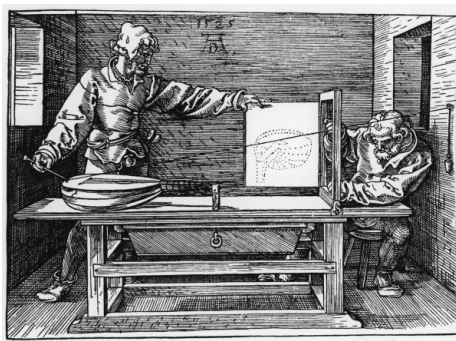


Raycasting

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
- Surfaces
- ▼ Transformations
 - ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
 - ▼ Projective transformations
 - ▼ Graphics Pipeline
 - Transformations
 - Clipping
- Shading
- Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
 - ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB

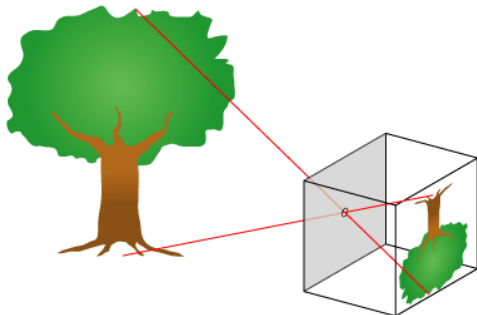


The Draughtsman of the Lute – Albrecht Dürer, 1525



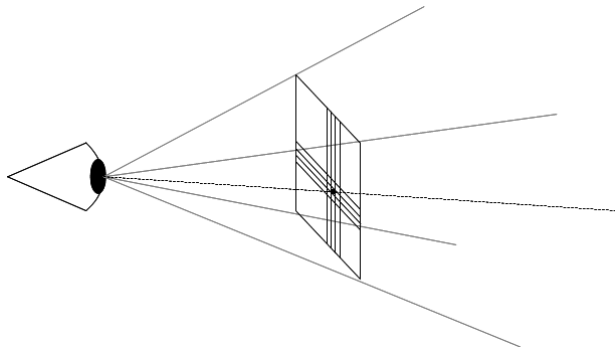
Albrecht Dürer, 1525

Pinhole camera



- ▶ Let us consider each pixel as a small window to the world

Pixel-sized hole



- ▶ What color shall we paint this window, so that we end up with an image that looks like reality?



Raycasting

For each pixel:

Construct a ray r from the pixel

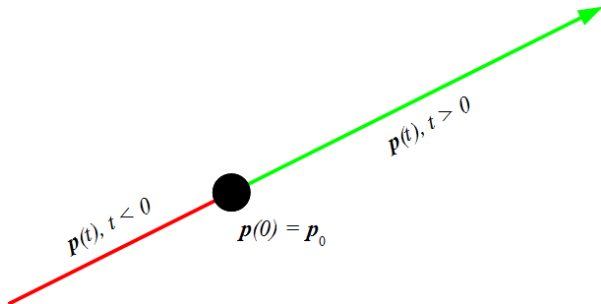
For each object o in the scene:

Compute intersection of r with o

Pixel color = closest intersected object's color



Ray



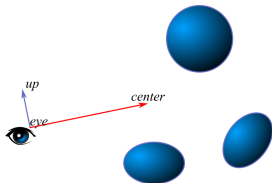
- ▶ A ray¹ has a p_0 **origin** and a v **direction**
- ▶ The parametric form of the ray is then

$$p(t) = p_0 + t \cdot v \quad (t > 0)$$

¹Not to be confused with Rey from Star Wars.

Ray creation

1. How to compute p_0 , v for a given pixel (i, j) ?
2. How to intersect anything with a ray?



Camera properties

- ▶ Camera position ($eye \in \mathbb{E}^3$),
- ▶ Look at point ($center \in \mathbb{E}^3$),
- ▶ Up direction ($up \in \mathbb{R}^3$),
- ▶ Horizontal and vertical field of view $fov_x, fov_y \in \mathbb{R}$.

Ray creation

Let us find the right-handed u, v, w **orthonormal** base of the camera!

- ▶ Let the camera face $-Z$:

$$w = \frac{\mathit{eye} - \mathit{center}}{\|\mathit{eye} - \mathit{center}\|_2}$$

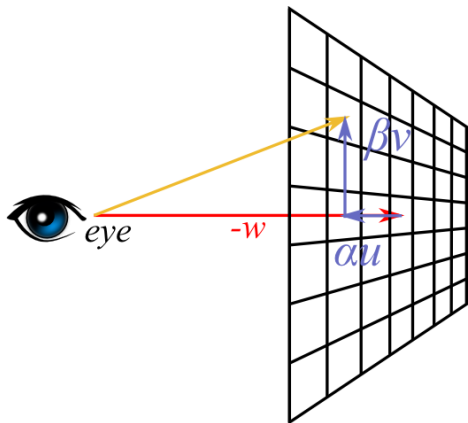
- ▶ Let the X axis be perpendicular to w and up :

$$u = \frac{up \times w}{\|up \times w\|_2}$$

- ▶ Let the Y axis be perpendicular to X and Z :

$$v = w \times u$$



Coordinates of the pixel at (i, j) 

Let the plane, onto which we are projecting, be 1 unit away from the origin (eye). The $p(t) = p_0 + tv$ ($t > 0$) ray for pixel (i, j) is:

$$p_0 = eye + (\alpha u + \beta v - w), \quad v = \frac{p(i, j) - eye}{\|p(i, j) - eye\|_2}$$

where

$$\alpha = \tan\left(\frac{fov_x}{2}\right) \cdot \frac{i - width/2}{width/2},$$

$$\beta = \tan\left(\frac{fov_y}{2}\right) \cdot \frac{height/2 - j}{height/2}.$$



Ray intersections

- ▼ Introduction
 - Course details
- ▼ Coordinate systems
 - Cartesian
 - Polar and spherical
 - Barycentric
 - Homogeneous coordinates
- ▼ Curves and surfaces
 - Lines and planes
 - Curves
- Surfaces
- ▼ Transformations
 - ▼ Affine transformations
 - Translation
 - Rotation
 - Scaling
 - Shearing
 - Change of basis
 - ▼ Projective transformations
 - ▼ Graphics Pipeline
 - Transformations
 - Clipping
- Shading
- Display
- ▼ Raycasting
 - Raycasting
 - Ray creation
- ▼ Ray intersections
 - Ray – plane
 - Ray – triangle
 - Ray – polygon
 - Ray – sphere
 - Transformed objects
 - Ray – AAB



Ray intersections



- ▶ Let us consider the parametric form of the ray: $p(t) = p_0 + tv$, where in the following we presume that $\|v\|_2 = 1$

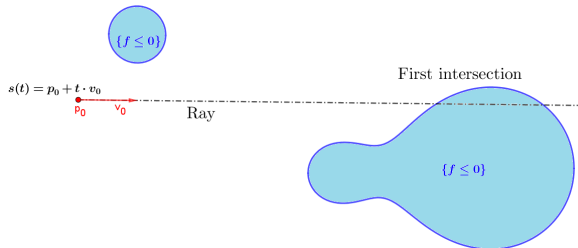


Parametric ray – implicit surface intersection

- ▶ Implicit surface: $f(\mathbf{x}) = 0$, where $\mathbf{x} \in \mathbb{E}^3$
- ▶ Let us plug the parametric ray's equation into the implicit equation of the surface!

$$f(\mathbf{p}(t)) = 0$$

- ▶ Let t be a root of $f(\mathbf{p}(t))$.
 - ▶ $t > 0$: the ray intersects the surface. (Object intersection is in front of the camera)
 - ▶ $t = 0$: the ray starts from the surface.
 - ▶ $t < 0$: the ray does not intersect the surface. (Intersection is behind the camera)



Parametric ray – parametric surface intersection

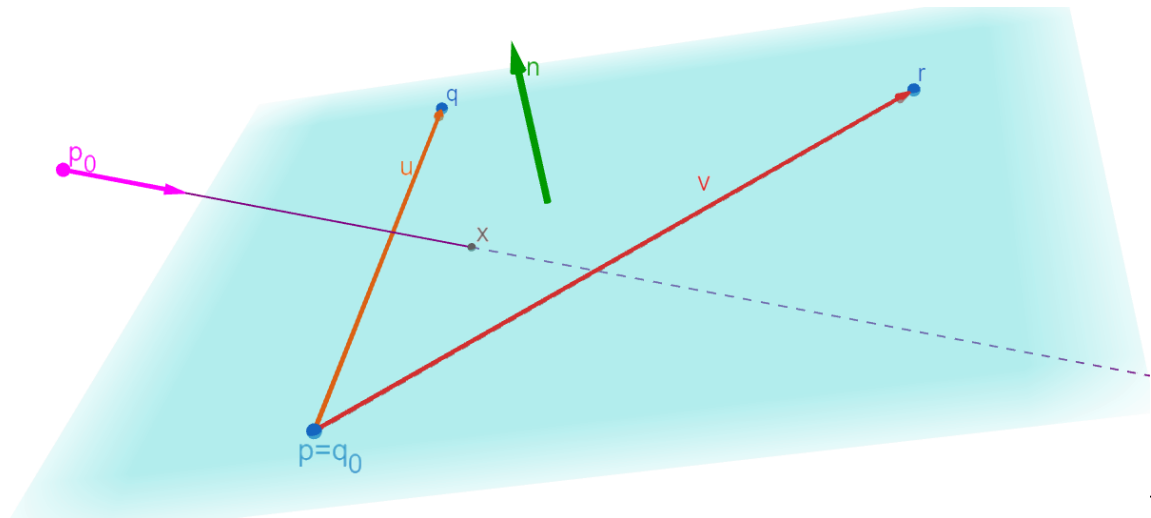
- ▶ Let $\mathbf{r}(u, v) = [r_x(u, v), r_y(u, v), r_z(u, v)]^T$ be a parametric surface, $(u, v) \in [a, b] \times [c, d]$
- ▶ Solve for t, u, v :

$$\mathbf{p}(t) = \mathbf{r}(u, v)$$

- ▶ If (t, u, v) is a solution, then $t > 0$, $(u, v) \in [a, b] \times [c, d]$ has to be verified as well.
- ▶ Parametric ray – parametric surface intersections are usually hard to solve
- ▶ If you do solve it, you get texture coordinates too.



Ray - implicit plane intersection



Ray - implicit plane intersection

- ▶ Implicit plane: $Ax + By + Cz + D = 0$
- ▶ The ray $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{v} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + t \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ intersects the plane for t -s such that

$$A(x_0 + tx) + B(y_0 + ty) + C(z_0 + tz) + D = 0$$

- ▶ Solve it for t :

$$t(Ax + By + Cz) + Ax_0 + By_0 + Cz_0 + D = 0$$

$$t = -\frac{Ax_0 + By_0 + Cz_0 + D}{Ax + By + Cz}$$

- ▶ The ray intersects the plane in front of the camera if $t > 0$



Ray - plane intersection, plane given by a point and normal

- ▶ Let q_0 be a point of the plane and n its normal
- ▶ The implicit equation of the plane:

$$\langle n, q - q_0 \rangle = 0 \quad (q \in \mathbb{R}^3)$$

- ▶ Plug $p(t)$ into q :

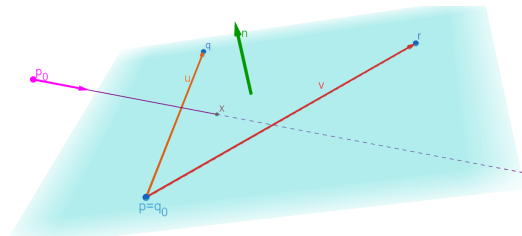
$$\langle n, p_0 + tv - q_0 \rangle = 0,$$

$$\langle n, p_0 \rangle + t \langle n, v \rangle - \langle n, q_0 \rangle = 0,$$

- ▶ Thus,

$$t = \frac{\langle n, q_0 \rangle - \langle n, p_0 \rangle}{\langle n, v \rangle} = \frac{\langle n, q_0 - p_0 \rangle}{\langle n, v \rangle},$$

- ▶ The ray's line intersects the plane in front of the camera if $t > 0$.
- ▶ $\langle n, v \rangle = 0$: the plane is parallel to the ray



Ray – parametric plane intersection

- ▶ Let the plane be given by one of its points, \mathbf{q} and two spanning vectors, \mathbf{i}, \mathbf{j} :

$$\mathbf{s}(u, v) = \mathbf{q} + u\mathbf{i} + v\mathbf{j}$$

- ▶ Intersection with the ray $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{v}$: find t and u, v such that

$$\mathbf{p}(t) = \mathbf{s}(u, v)$$

$$\mathbf{p}_0 + t\mathbf{v} = \mathbf{q} + u\mathbf{i} + v\mathbf{j} \implies \mathbf{p}_0 - \mathbf{q} = -t\mathbf{v} + u\mathbf{i} + v\mathbf{j}$$

- ▶ In matrix form:

$$\begin{bmatrix} p_{0x} - q_x \\ p_{0y} - q_y \\ p_{0z} - q_z \end{bmatrix} = \begin{bmatrix} -v_x & i_x & j_x \\ -v_y & i_y & j_y \\ -v_z & i_z & j_z \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix}$$

- ▶ The previous can be solved iff $\mathbf{v}, \mathbf{i}, \mathbf{j}$ are linearly independent
- ▶ Intersection in front of the camera iff $t > 0$



Ray - triangle intersection (Tomas Möller and Ben Trumbore, 1997)

- ▶ Triangle is given as a convex barycentric combination of its vertices \mathbf{a} , \mathbf{b} , \mathbf{c} :

$$\mathbf{r}(u, v) = (1 - u - v)\mathbf{a} + u\mathbf{b} + v\mathbf{c} \quad (u, v, u + v \in [0, 1])$$

- ▶ $\mathbf{p}_0 + t\mathbf{v} = \mathbf{r}(u, v)$ equation is to be solved for u, v, t , after rearranging:

$$\begin{bmatrix} -v & \mathbf{b} - \mathbf{a} & \mathbf{c} - \mathbf{a} \end{bmatrix} \cdot \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{p}_0 - \mathbf{a}$$

- ▶ Using the Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det \begin{bmatrix} -v & \mathbf{b} - \mathbf{a} & \mathbf{c} - \mathbf{a} \end{bmatrix}} \cdot \begin{bmatrix} \det \begin{bmatrix} \mathbf{p}_0 - \mathbf{a} & \mathbf{b} - \mathbf{a} & \mathbf{c} - \mathbf{a} \end{bmatrix} \\ \det \begin{bmatrix} -v & \mathbf{p}_0 - \mathbf{a} & \mathbf{c} - \mathbf{a} \end{bmatrix} \\ \det \begin{bmatrix} -v & \mathbf{b} - \mathbf{a} & \mathbf{p}_0 - \mathbf{a} \end{bmatrix} \end{bmatrix}$$



Ray - triangle intersection (Tomas Möller and Ben Trumbore, 1997)

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det \begin{bmatrix} -v & b-a & c-a \end{bmatrix}} \cdot \begin{bmatrix} \det \begin{bmatrix} p_0 - a & b-a & c-a \end{bmatrix} \\ \det \begin{bmatrix} -v & p_0 - a & c-a \end{bmatrix} \\ \det \begin{bmatrix} -v & b-a & p_0 - a \end{bmatrix} \end{bmatrix}$$

- Determinants expressed as triple products: $\det \begin{bmatrix} x & y & z \end{bmatrix} = \langle x \times y, z \rangle$, ($x, y, z \in \mathbb{R}^3$)

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\langle v \times (c-a), b-a \rangle} \cdot \begin{bmatrix} \langle (p_0 - a) \times (b-a), c-a \rangle \\ \langle v \times (c-a), p_0 - a \rangle \\ \langle (p_0 - a) \times (b-a), v \rangle \end{bmatrix} = \frac{1}{\langle f, ab \rangle} \begin{bmatrix} \langle g, ac \rangle \\ \langle f, ap \rangle \\ \langle g, v \rangle \end{bmatrix}$$

where

$$ab = b - a$$

$$ac = c - a$$

$$ap = p_0 - a$$

$$f = v \times ac$$

$$g = ap \times ab$$



Ray - triangle intersection (Tomas Möller and Ben Trumbore, 1997)

- ▶ Therefore, the final formula is

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\langle \mathbf{f}, \mathbf{ab} \rangle} \begin{bmatrix} \langle \mathbf{g}, \mathbf{ac} \rangle \\ \langle \mathbf{f}, \mathbf{ap} \rangle \\ \langle \mathbf{g}, \mathbf{v} \rangle \end{bmatrix}$$

where

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

$$\mathbf{ac} = \mathbf{c} - \mathbf{a}$$

$$\mathbf{ap} = \mathbf{p}_0 - \mathbf{a}$$

$$\mathbf{f} = \mathbf{v} \times \mathbf{ac}$$

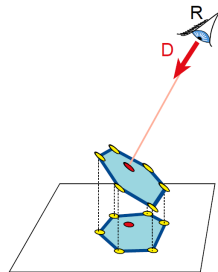
$$\mathbf{g} = \mathbf{ap} \times \mathbf{ab} .$$

- ▶ There is an intersection with the triangle if $t, u, v, 1 - u - v$ values are non-negative
- ▶ Solution resolves u, v texture coordinates as well



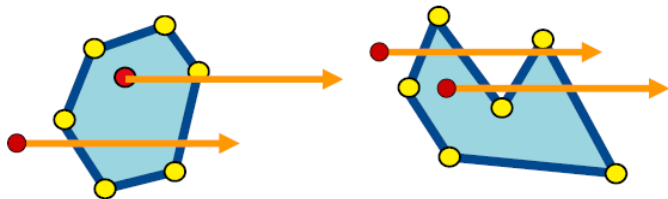
Ray - polygon intersection

- ▶ Let the polygon be a planar polygon. The intersection can be tested in two steps:
 - ▶ Intersect the ray with the polygon's plane
 - ▶ Test whether the intersection point is inside the polygon or not
- ▶ The second step once again can be done in a plane (either in the polygon's or a coordinate plane)



Point - polygon containment test

- ▶ A point is inside the polygon if an arbitrary half-line (**ray**) starting from that point has an odd amount of intersections with the edges of the polygon
- ▶ Other idea method via the **winding number**.



Ray-line segment intersection

- ▶ The line segment spanned between the two consecutive points of the polygon $\mathbf{d}_i = (x_i, y_i)$, $\mathbf{d}_{i+1} = (x_{i+1}, y_{i+1})$ has the parametric form

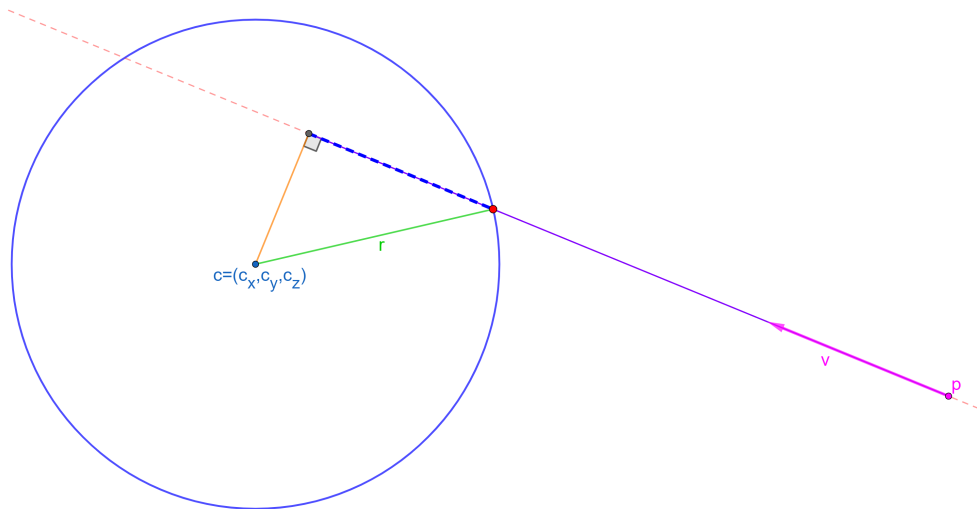
$$\mathbf{d}_{i,i+1}(s) = (1 - s)\mathbf{d}_i + s\mathbf{d}_{i+1} = \mathbf{d}_i + s(\mathbf{d}_{i+1} - \mathbf{d}_i), \quad s \in [0, 1]$$

- ▶ Let us intersect it with $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{v}$, where $\mathbf{p}_0 = (x_0, y_0)$ (where \mathbf{p}_0 is the intersection of the ray and the plane, and \mathbf{v} is the projection of the direction in the plane).
- ▶ Need to solve $\mathbf{p}(t) = \mathbf{d}_{i,i+1}(s)$ for $s \in [0, 1], t > 0$!
- ▶ Let $\mathbf{v} = (1, 0)$ since we may choose any direction.
- ▶ Solve for Y direction only: $(\mathbf{d}_{i,i+1})_y = y_i + s \cdot (y_{i+1} - y_i) = y_0$, that is:
- ▶ If $s \notin [0, 1]$ or $t < 0$ the ray does not intersect the line segment

$$s = \frac{y_0 - y_i}{y_{i+1} - y_i}$$



Ray - sphere intersection



Ray - sphere intersection

- ▶ The $\mathbf{c} = (c_x, c_y, c_z)$ centered, radius r sphere's equation is:

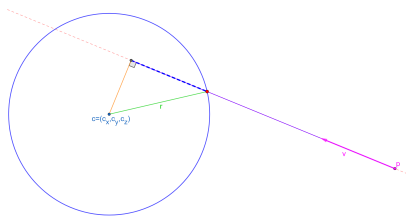
$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 = 0$$

$$\langle \mathbf{p} - \mathbf{c}, \mathbf{p} - \mathbf{c} \rangle - r^2 = 0, \quad \mathbf{p} = (x, y, z)$$

- ▶ Let \mathbf{p}_0 be the ray's origin and \mathbf{v} its direction
- ▶ Substituting the ray's equation into the plane's

$$\langle \mathbf{p}_0 + t\mathbf{v} - \mathbf{c}, \mathbf{p}_0 + t\mathbf{v} - \mathbf{c} \rangle - r^2 = 0$$

$$t^2 \langle \mathbf{v}, \mathbf{v} \rangle + 2t \langle \mathbf{v}, \mathbf{p}_0 - \mathbf{c} \rangle + \langle \mathbf{p}_0 - \mathbf{c}, \mathbf{p}_0 - \mathbf{c} \rangle - r^2 = 0$$



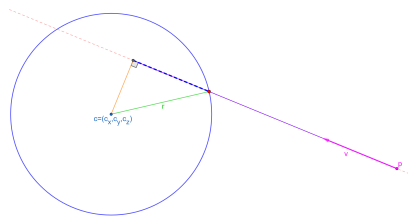
Ray - sphere intersection

$$t^2 \langle \mathbf{v}, \mathbf{v} \rangle + 2t \langle \mathbf{v}, \mathbf{p}_0 - \mathbf{c} \rangle + \langle \mathbf{p}_0 - \mathbf{c}, \mathbf{p}_0 - \mathbf{c} \rangle - r^2 = 0$$

- ▶ Quadratic equation in t with discriminant

$$D = (2 \langle \mathbf{v}, \mathbf{p}_0 - \mathbf{c} \rangle)^2 - 4 \langle \mathbf{v}, \mathbf{v} \rangle (\langle \mathbf{p}_0 - \mathbf{c}, \mathbf{p}_0 - \mathbf{c} \rangle - r^2)$$

- ▶ If $D > 0$: two solutions, the smallest positive is the intersection in front of the camera
- ▶ If $D = 0$: one solution, ray is tangential
- ▶ If $D < 0$: no intersection



Transformed objects

Theorem

The intersection of the ray \mathbf{r} and an object transformed by transformation \mathbf{M}

\equiv

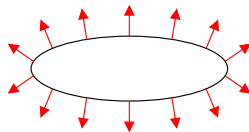
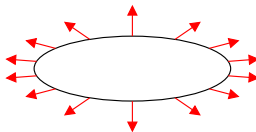
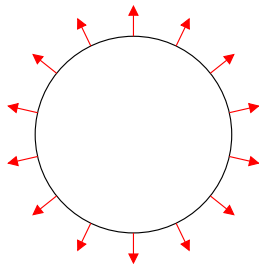
the intersection of the transformed ray $\mathbf{M}^{-1}\mathbf{r}$ and the non-transformed object transformed by \mathbf{M}

- ▶ $\mathbf{M} \in \mathbb{R}^{4 \times 4}$ homogeneous transformation
- ▶ Ray's origin: $\mathbf{p}_0 = (p_x, p_y, p_z) \rightarrow [p_x, p_y, p_z, 1]^T$
- ▶ Ray's direction: $\mathbf{v} = (v_x, v_y, v_z) \rightarrow [v_x, v_y, v_z, 0]^T$. Translation in \mathbf{M} does not affect it.
- ▶ Transformed ray $\mathbf{r}_M(t) = \mathbf{M}^{-1}\mathbf{p} + t \cdot \mathbf{M}^{-1}\mathbf{v}$



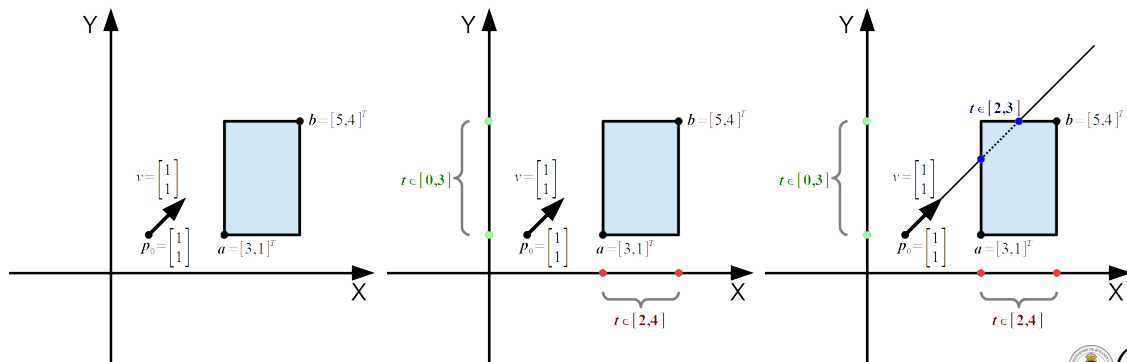
Transformed objects

- ▶ Intersection query: use $\mathbf{r}_M(t)$!
- ▶ If the intersection in the model space is \mathbf{q} , then in the world space it is $\mathbf{M} \cdot \mathbf{q}$.
- ▶ Distances change unless \mathbf{M} is a rigid transformation
- ▶ Pay attention to normal vectors: $\mathbf{M}^{-T} \cdot \mathbf{n}$



Ray - Axis aligned box (AAB) intersection

- ▶ AAB = axis aligned box, the sides of the box are parallel to the coordinate planes
- ▶ Let $p(t) = p_0 + tv$ be a ray, where $p_0 = (x_0, y_0)$ and $v = (v_x, v_y)$
- ▶ Let us represent the AAB by the endpoints of its diagonal, a and b , where $a < b$



Ray - Axis aligned box (AAB) intersection

- ▶ We check the intersection for all axis one by one, let us start with x
 - ▶ $v_x = 0$: the ray is vertical, there's no intersection if $x_0 \notin [a_x, b_x]$
 - ▶ $v_x \neq 0$: let $t_n := -\infty, t_f := +\infty$ and $t_1 := \frac{a_x - x_0}{v_x}, t_2 := \frac{b_x - x_0}{v_x}$
 - ▶ If $t_1 > t_2$: swap t_1 and t_2
 - ▶ If $t_n < t_1$: $t_n := t_1$
 - ▶ If $t_f > t_2$: $t_f := t_2$
- ▶ Then proceed to the y and then the z coordinates
- ▶ Finally, if
 - ▶ $t_n > t_f$: the ray misses the box
 - ▶ $t_f < 0$: the box is behind the ray
 - ▶ Else: t_n is the closer, t_f is the farther intersection point's ray parameter



Thank you for your attention!



Human Vision, Computer Colors, and Physics of Light

Csaba **Bálint**
first name family name

csabix@inf.elte.hu

Eötvös Loránd University,
Faculty of Informatics

Computer Graphics Lecture
Budapest 2022

Table of Contents

- ▼ Introduction
 - Course details
- ▼ Human vision
 - Motivation
 - Eye Anatomy
 - Receptors
- ▼ Computer Colors
 - Color spaces
 - RGB
 - XYZ
 - Perception-based
 - sRGB
- ▼ Physics of Light
 - Motivation
 - Properties of light
 - Speed of light
 - Light as a wave
 - Blackbody radiation
 - Photoelectric effect
 - Relativity



Course details

- ▶ Csaba Bálint csabix@inf.elte.hu (me). Room: 2-706.
- ▶ **Lecture**: Tuesday, 16:15-17:45
 - ▶ Exam: Explain 2 topics. You chose one, and I choose one.
 - ▶ Points from practice can help your course (but do not count on it).
- ▶ **Practice**: Tuesday, 18:00-19:30
 - ▶ Small assignment (≈ 30 points) and large assignment (100+ points)
 - ▶ Work during practice (≈ 15 points)
 - ▶ Scores above 100 points will count towards the lecture exam
- ▶ In both
 - ▶ **Grade boundaries: 40, 55, 70, 85.**



Further reading

1. *Edward Angel, Dave Shreiner*: Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL (6th Edition)
2. *Andrew Glassner*: [Principles of digital image synthesis](#)
3. *Pharr, Humphreys, Hanrahan*: [Physically Based Rendering \(From Theory to Implementation\)](#)
4. *Akenine-Möller, Haines, Hoffman*: [Real-Time Rendering \(4th edition\)](#)
5. *Tekla Tóth, Iván Eichhardt, Gábor Valasek*: [BSc Computer Graphics Lecture slides](#)



Human vision

- ▼ Introduction
 - Course details
- ▼ Human vision
 - Motivation
 - Eye Anatomy
 - Receptors
- ▼ Computer Colors
 - Color spaces
 - RGB
 - XYZ
- Perception-based
- sRGB
- ▼ Physics of Light
 - Motivation
 - Properties of light
 - Speed of light
 - Light as a wave
 - Blackbody radiation
 - Photoelectric effect
 - Relativity



Motivation

- ▶ Computer Graphics: **physics simulation** based on some postulates/assumptions.
- ▶ How do we **represent** our virtual world (scene)?
- ▶ What **algorithms** do we use to render the images?
- ▶ It depends on how we perceive the real world in the first place
- ▶ Sensors and psychological factors
- ▶ Generate real looking images. They must look real to **us**.



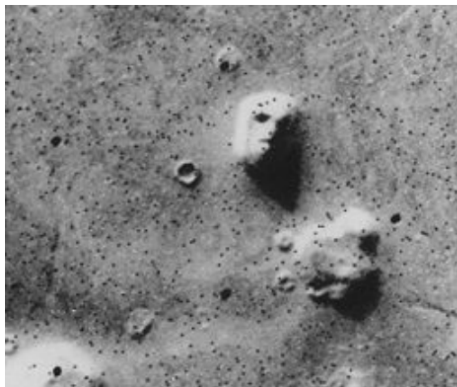
Motivation – fun facts¹

- ▶ Your eyes focus on 50 different objects every second.
- ▶ The only organ more complex than the eye is the brain.
- ▶ Your eyes can distinguish approximately 10 million different colors.
- ▶ It is impossible to sneeze with your eyes open.
- ▶ 80 percent of all learning comes through the eyes.
- ▶ Your eyes can detect a candle flame 2.7km away.
- ▶ Your iris (the colored part of your eye) has 256 unique characteristics; your fingerprint has just 40.
- ▶ Only $\frac{1}{6}$ of your eyeball is visible.
- ▶ The average person blinks 12 times a minute (bet you just blinked!).
- ▶ The shark cornea is nearly identical to the human cornea, and has even been used in human eye surgery!
- ▶ Your eye is the fastest contracting muscle in the body, contracting in less than $\frac{1}{100}$ of a second.

¹Source: <https://versanthealth.com/blog/15-facts-about-all-things-eyes/>



Cydonia (1987)

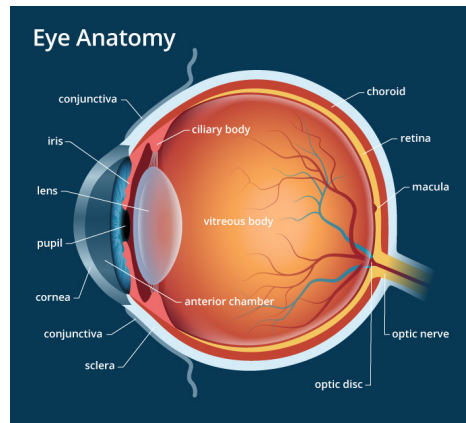


Cydonia (1997)

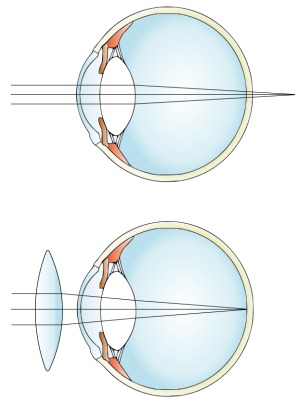
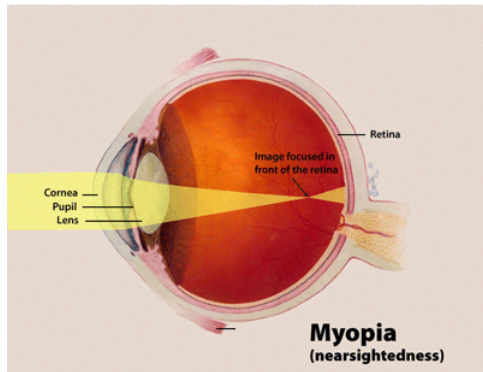


Human Eye

1. **Cornea** refracts light
 \implies strong gathering lens
2. **Iris** limits (filters) light intensity
3. **Lens** focuses light ray onto the retina
 \implies adjustable lens
4. **Retina** contains the receptors (rods & cones)
5. **Optic nerve** encode and transmit data to brain

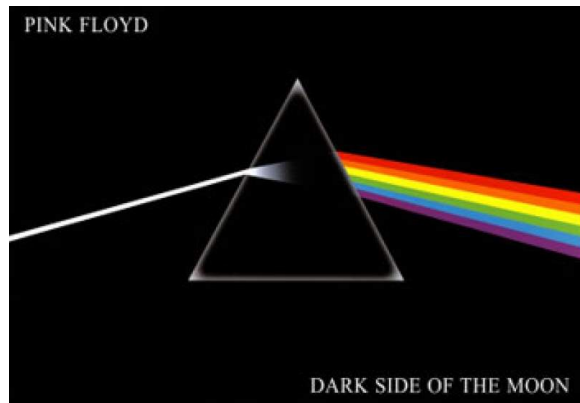


Near vs Farsightedness



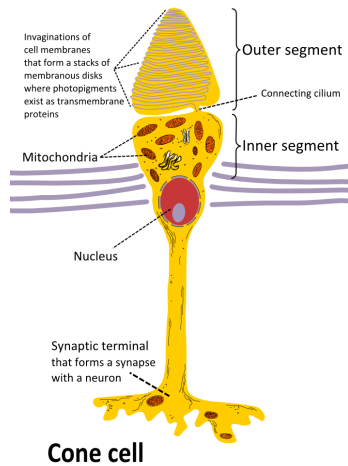
Lens

- ▶ Light coming from close objects need more refraction
- ▶ Refraction depends on the wavelength of the light (color)
- ▶ Muscles have to adjust the shape of the lens to correct
- ▶ A red room feels smaller than a blue room because the eye has to focus closer in a red room



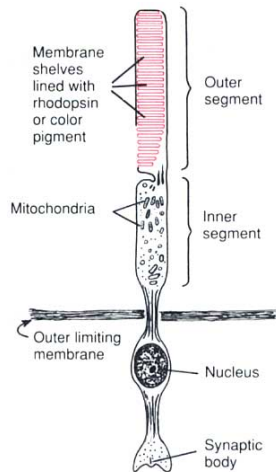
Photoreceptors – Cones

- ▶ **Color perception**
- ▶ 3 types, S:M:L = 1:4:8
- ▶ High accuracy at
- ▶ Central fovea 150000/mm²
- ▶ 6-7 million

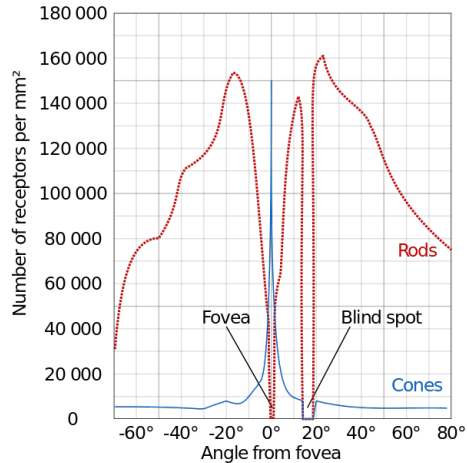
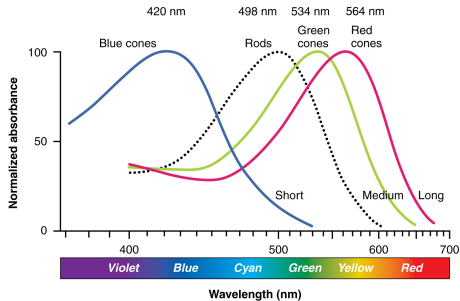


Photoreceptors – Rods

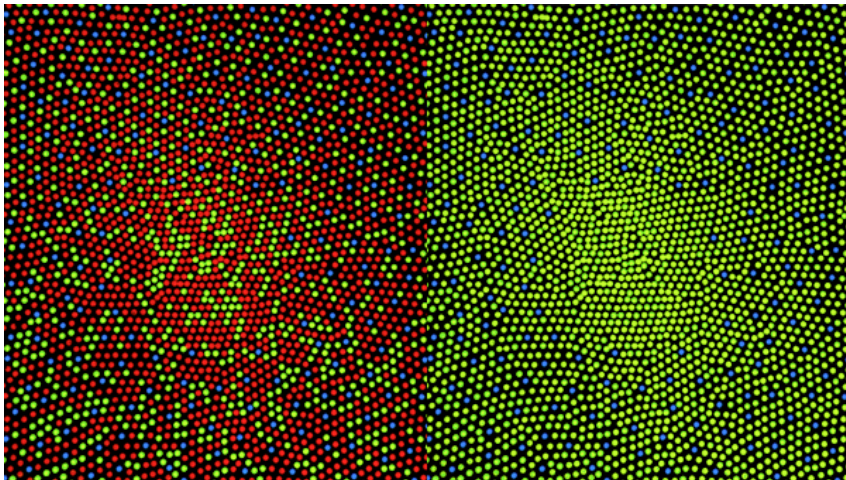
- ▶ **Light intensity**
- ▶ More sensitive to light
- ▶ No color information
- ▶ Lower accuracy
- ▶ Peripheral vision
- ▶ Slower reaction
- ▶ 90-120 million



Photoreceptors



Photoreceptors – normal vision vs color blind

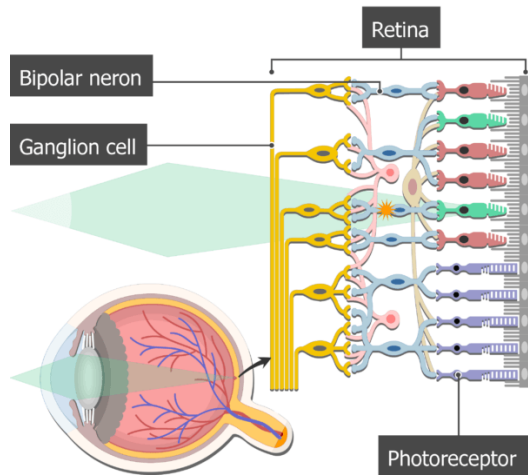


Blind spot – experiment

Look at the X with your right eye while having your left one closed. How far back do you have to sit to make the dot on the right disappear?

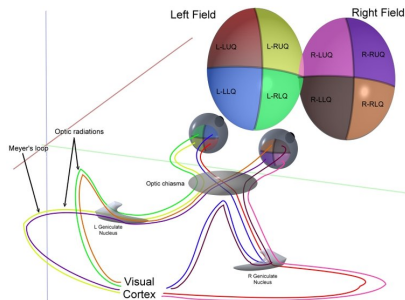


Blind spot



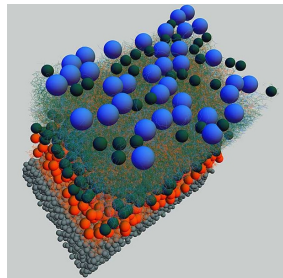
Signal frequency

- ▶ A single photon can cause a signal that lasts a few milliseconds
- ▶ Each of these are additive \Rightarrow average in time \Rightarrow low pass filter
- ▶ **Critical Flicker Frequency:** lowest frequency flicker seen as continuous
 - ▶ Depends on several human and environmental factors
 - ▶ About 60Hz
 - ▶ However, we can detect 500Hz anomalies!



Signal compression

- ▶ 120 million rods, 6 million cones \Rightarrow 1 million nerves
- ▶ Different channels:
 - ▶ $A = M + L$ achromatic channel (R + G) intensity
 - ▶ $R/G = M - L$ red minus green red-green
 - ▶ $B/Y = S - A$ blue minus achromatic blue-yellow
- ▶ **Neurons connect to multiple photoreceptors**
- ▶ JPEG compression does a similar transform!



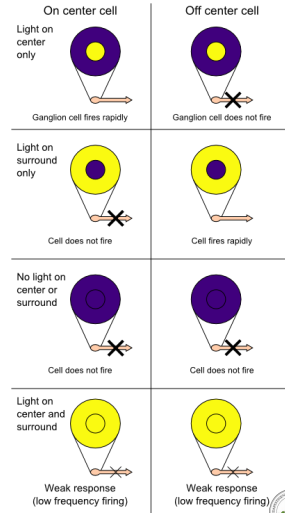
Source: MIT

Signal compression

- ▶ Neurons connect to multiple photoreceptors
- ▶ Two types of neurons: On-center and Off-center
- ▶ On-center neuron:
 - ▶ High positive weight on middle photoreceptors
 - ▶ Small negative weight on the rest (sum is not zero)
 - ▶ Enhanced resolution around edges \Rightarrow Edge detection
- ▶ Off-center neuron is opposite
- ▶ Similar to edge detection:

-1	-1	-1	0	-1	0
-1	8	-1	-1	4	-1
-1	-1	-1	0	-1	0

Common Laplacian edge detection filters

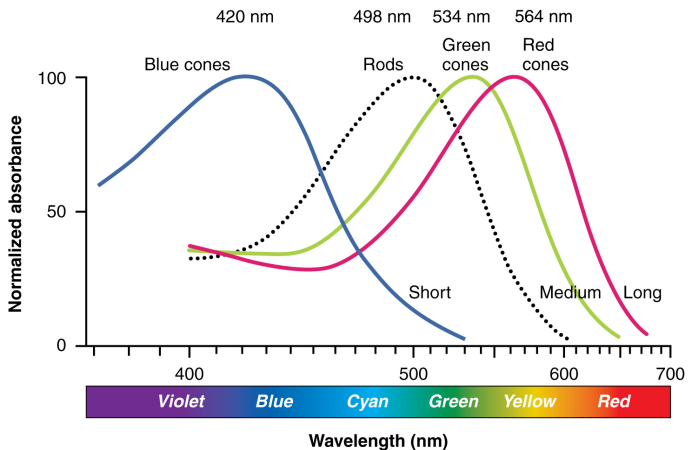


Computer Colors

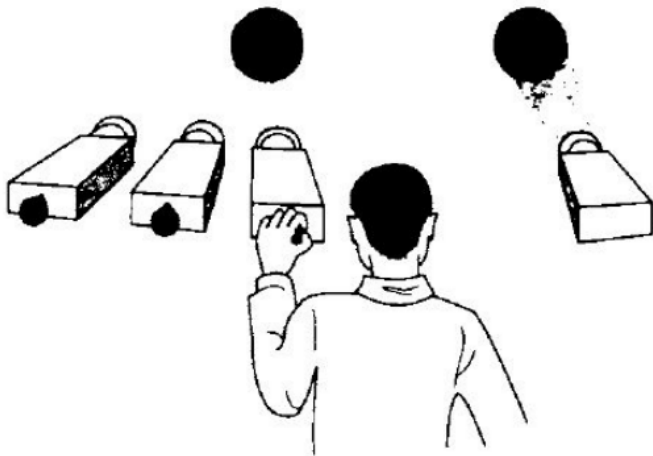
- ▼ Introduction
 - Course details
- ▼ Human vision
 - Motivation
 - Eye Anatomy
 - Receptors
- ▼ Computer Colors
 - Color spaces
 - RGB
 - XYZ
 - Perception-based
 - sRGB
- ▼ Physics of Light
 - Motivation
 - Properties of light
 - Speed of light
 - Light as a wave
 - Blackbody radiation
 - Photoelectric effect
 - Relativity



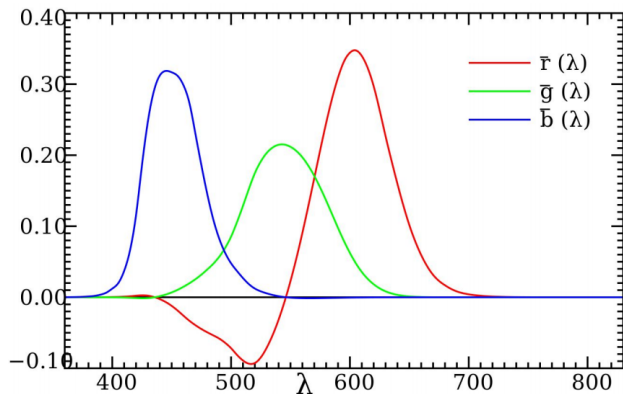
Reminder: Cone light absorption



CIE: Commission internationale de l'éclairage



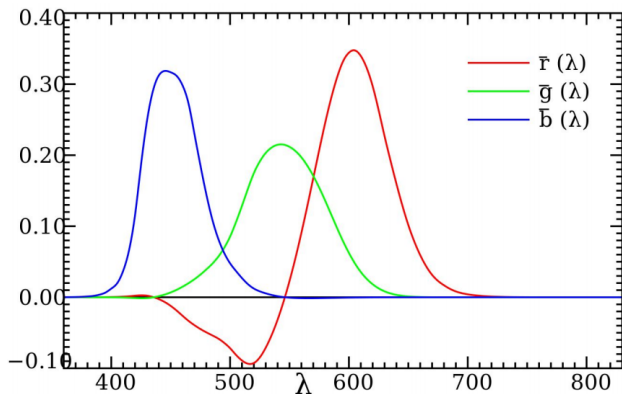
CIE RGB color space



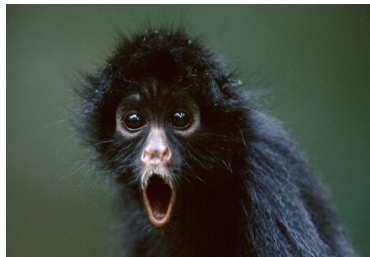
- ▶ Not the same as receptor absorption!
- ▶ Reflectors had specific wavelengths, but each cone has an absorption spectrum



CIE RGB color space



- ▶ Not the same as receptor absorption!
- ▶ Reflectors had specific wavelengths, but each cone has an absorption spectrum



Spectrum \rightarrow RGB

Given $S(\lambda)$ spectrum for a light, we can calculate the RGBs as **inner products**:

$$R = \int_0^{+\infty} S(\lambda) \cdot \bar{r}(\lambda) d\lambda$$

$$G = \int_0^{+\infty} S(\lambda) \cdot \bar{g}(\lambda) d\lambda$$

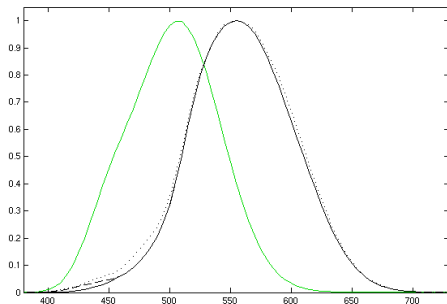
$$B = \int_0^{+\infty} S(\lambda) \cdot \bar{b}(\lambda) d\lambda$$



RGB \rightarrow XYZ

Another CIE color space: XYZ

- ▶ Average observer light intensity \Rightarrow
- ▶ RGB has negative weights
- ▶ RGB \rightarrow XYZ has to be linear
- ▶ Equal intensity point has to be $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$



$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{b_{21}} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49000 & 0.31000 & 0.20000 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00000 & 0.01000 & 0.99000 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$



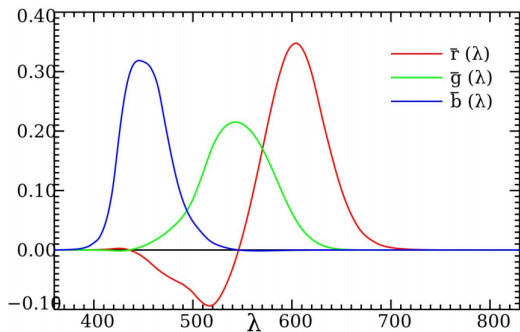
RGB \rightarrow XYZ

Figure: RGB

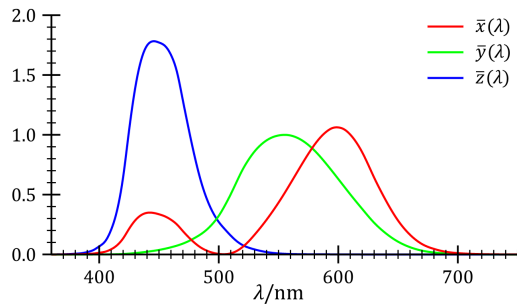
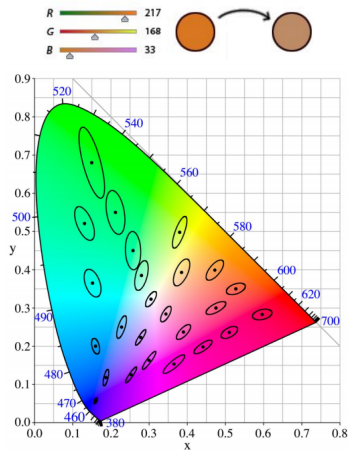


Figure: XYZ

Color spaces

- ▶ RGB and XYZ are not practical for creative work
- ▶ Changing the color slightly can be difficult
- ▶ What colors are similar?
 - ▶ **MacAdam ellipses**
 - ▶ Participants recorded when two colors appear different
 - ▶ Similar colors appear as ellipses
- ▶ If MacAdam ellipses are circles
⇒ Uniform color space
- ▶ If linear changes in color space appear linear
⇒ Linear color space



- ▶ HP and Microsoft created the standard for monitors, printers, and internet
- ▶ Linear transformation from XYZ
- ▶ Non-linear Gamma correction
 - ▶ for each channel

$$C_{\text{srgb}} = \begin{cases} 12.92C_{\text{linear}}, & C_{\text{linear}} \leq 0.0031308 \\ (1 + a)C_{\text{linear}}^{1/2.4} - a, & C_{\text{linear}} > 0.0031308 \end{cases} \quad a = 0.055$$

$$\begin{bmatrix} R_{\text{linear}} \\ G_{\text{linear}} \\ B_{\text{linear}} \end{bmatrix} = \begin{bmatrix} 3.2406 & -1.5372 & -0.4986 \\ -0.9689 & 1.8758 & 0.0415 \\ 0.0557 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$



sRGB \rightarrow XYZ

- ▶ Since sRGB is non-linear, vector operations are not allowed
 - ▶ because energy is non-linear too!
- ▶ Image processing usually requires linear space
- ▶ Data is often stored in sRGB
- ▶ Conversion:

$$C_{\text{linear}} = \begin{cases} \frac{C_{\text{srgb}}}{12.92}, & C_{\text{srgb}} \leq 0.04045 \\ \left(\frac{C_{\text{srgb}} + a}{1 + a}\right)^{2.4}, & C_{\text{srgb}} > 0.04045 \end{cases} \quad \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix} \begin{bmatrix} R_{\text{linear}} \\ G_{\text{linear}} \\ B_{\text{linear}} \end{bmatrix}$$



Physics of Light

- ▼ Introduction
 - Course details
- ▼ Human vision
 - Motivation
 - Eye Anatomy
 - Receptors
- ▼ Computer Colors
 - Color spaces
 - RGB
 - XYZ
- Perception-based
- sRGB
- ▼ Physics of Light
 - Motivation
 - Properties of light
 - Speed of light
 - Light as a wave
 - Blackbody radiation
 - Photoelectric effect
 - Relativity



Motivation

traditional shader content



pbr shader content



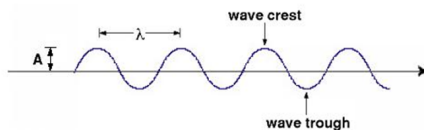
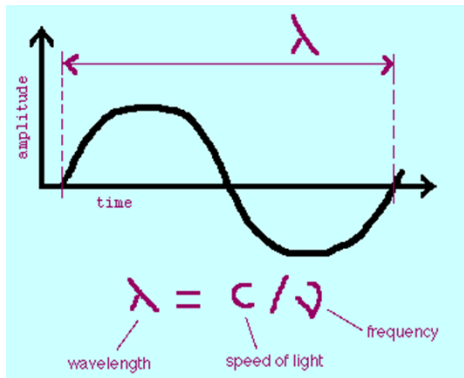
Motivation

- ▶ Goal: simulate how light behaves to render realistic images
- ▶ Build a mathematical model, create data structures, and algorithms
- ▶ We must understand how light behaves
- ▶ Capture core properties and disregard the rest
- ▶ 42



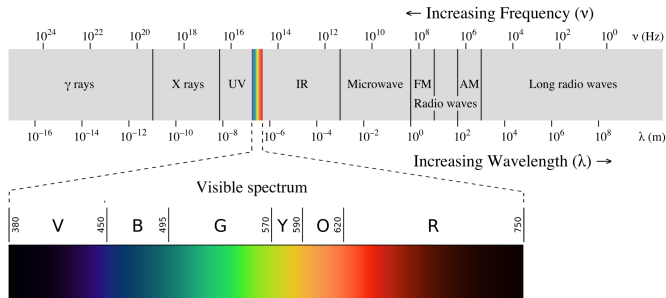
Light as a wave

Most of the properties of light can be explained with how waves behave.



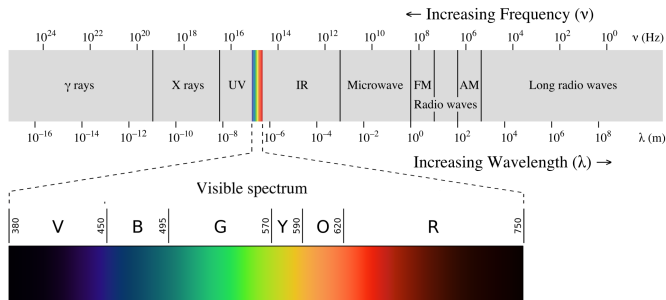
Light as a wave – wave properties

- ▶ Waves can do:
 - ▶ Absorption
 - ▶ Reflection
 - ▶ Refraction
 - ▶ Diffraction
 - ▶ Interference
 - ▶ Polarization
 - ▶ Dispersion
- ▶ Longitudinal and **transverse**
- ▶ Electromagnetic wave
- ▶ Propagation speed



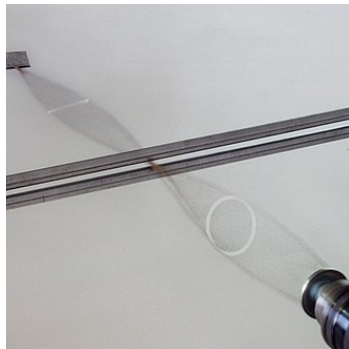
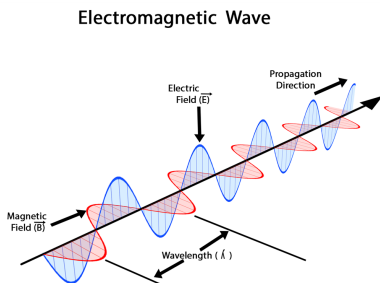
Light as a wave – monochromatic waves

- ▶ Frequency
 - ▶ color, speed / wavelength
- ▶ Amplitude (intensity)
- ▶ Direction of propagation
- ▶ Polarization
- ▶ Phase (relative)

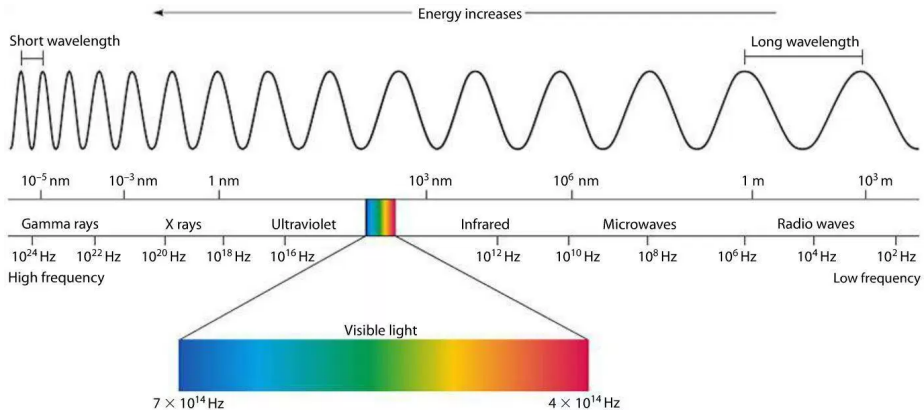


What is light?

- ▶ Electromagnetic wave
 - ▶ Wave in electric field induces wave in magnetic field and vice versa because of the Maxwell equations
- ▶ Transverse wave (not longitudinal)
- ▶ **Polarization**: direction of displacement of the Electric field



Electromagnetic spectrum



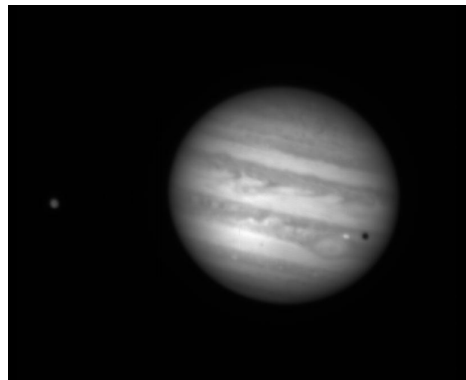
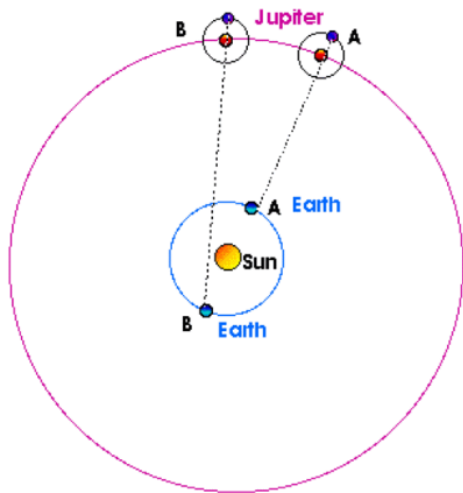
Speed of light

- ▶ Speed of light in vacuum: $c = 299792458$ m/s
- ▶ Low Earth Orbit (LEO) ≈ 7 km/s (0.002‰ c)
- ▶ Parker Solar Probe (PSP) ≈ 200 km/s (0.064‰ c)

- ▶ Defines the speed of every reaction (also the speed of time...)
- ▶ The definition of 1 meter:
 - ▶ The length of the path travelled by light in a vacuum in $\frac{1}{299792458}$ of a second.



Roemer: Io's Shadow on Jupiter



Measurements of the speed of light

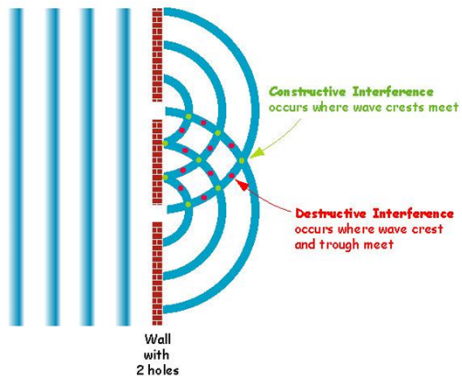
Date	Author	Method	Result (km/s)	Error
1676	Olaus Roemer	Jupiter's satellites	214,000	
1726	James Bradley	Stellar aberration	301,000	
1849	Armand Fizeau	Toothed wheel	315,000	
1862	Leon Foucault	Rotating mirror	298,000	+500
1879	Albert Michelson	Rotating mirror	299,910	+50
1907	Rosa, Dorsay	Electromagnetic constants	299,788	+30
1926	Albert Michelson	Rotating mirror	299,796	+4
1947	Essen, Gordon-Smith	Cavity resonator	299,792	+3
1958	K. D. Froome	Radio interferometer	299,792.5	+0.1
1973	Evanson <i>et al</i>	Lasers	299,792.4574	+0.001
1983		Adopted value	299,792.458	



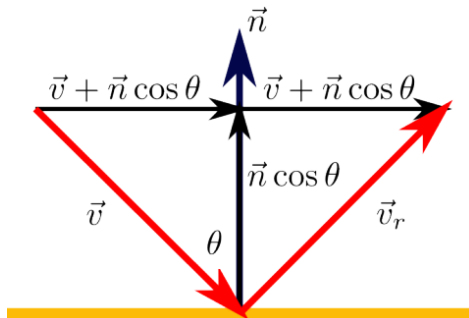
Diffraction



Interference

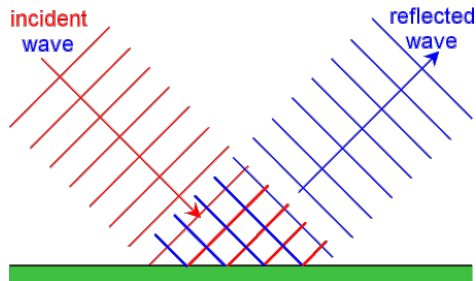


Reflection



$$\vec{v}_r = \vec{v} - 2\vec{n} \cdot \langle \vec{n}, \vec{v} \rangle$$

Householder transformation in linear algebra.



Refraction

- ▶ Snell's law:

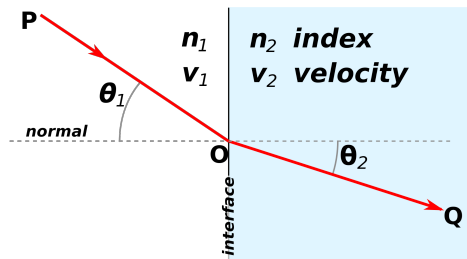
$$\frac{\sin \theta_1}{\sin \theta_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1}$$

- ▶ The refracted ray can be calculated as follows:

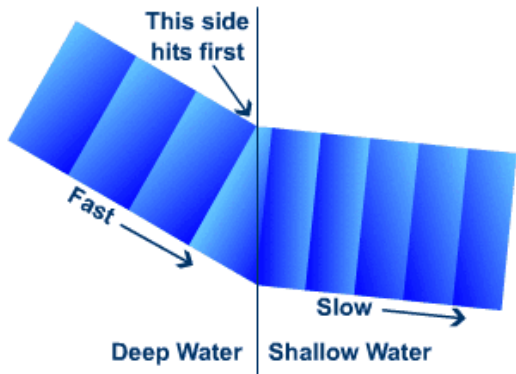
$$v_t = \frac{v}{\eta} + n \left(\frac{\cos \alpha}{\eta} - \sqrt{1 - \frac{\sin^2 \alpha}{\eta^2}} \right)$$

where $\eta = \frac{n_2}{n_1}$. See [refract](#) in GLSL.

- ▶ Index Of Reflection (IOR)
 - ▶ function of wavelength!



Refraction and mirage



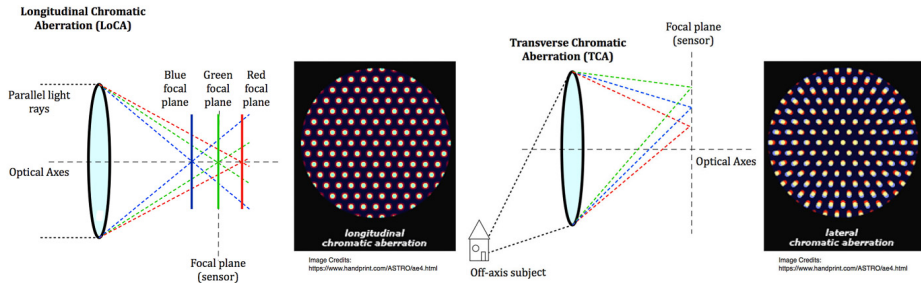
Mirage

Dispersion

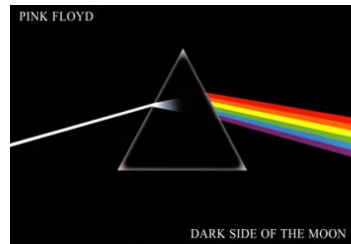
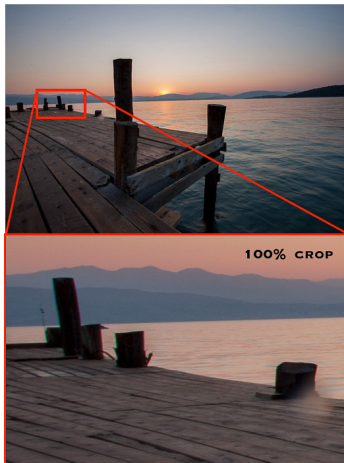
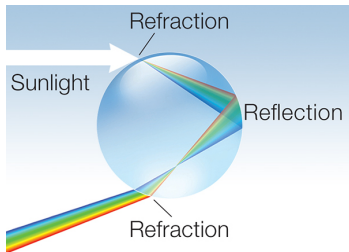
- ▶ Abbe number:

$$V_D = \frac{n_D - 1}{n_F - n_C}$$

- ▶ n_D , n_F and n_C are the refractive indices of the material at the wavelengths of the Fraunhofer C , D_1 , and F spectral lines (656.3 nm, 589.3 nm, and 486.1 nm respectively)
- ▶ Problem in cameras
- ▶ Chromatic aberration



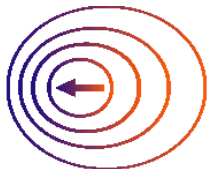
Dispersion



Polarization

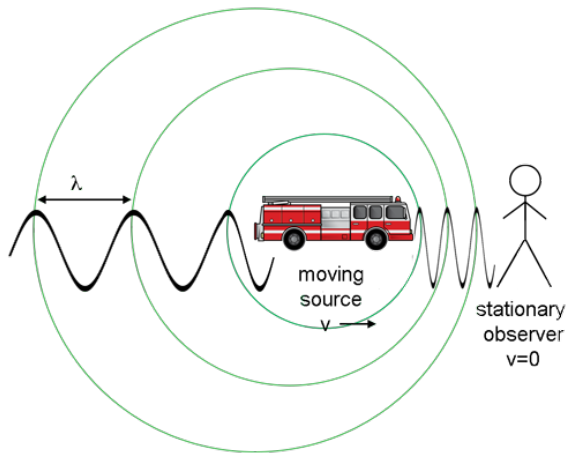


Doppler Shift



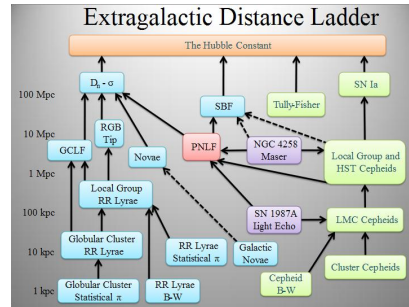
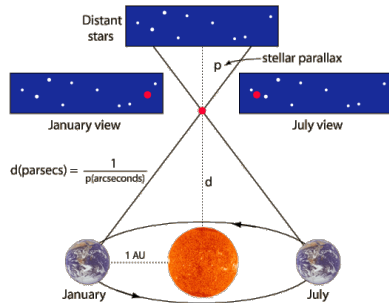
Source moving TOWARD observer
Wavelength decreasing,
Frequency increasing,
Observer experiencing BLUE shift.

Source moving AWAY from observer
Wavelength increasing,
Frequency decreasing,
Observer experiencing RED shift.

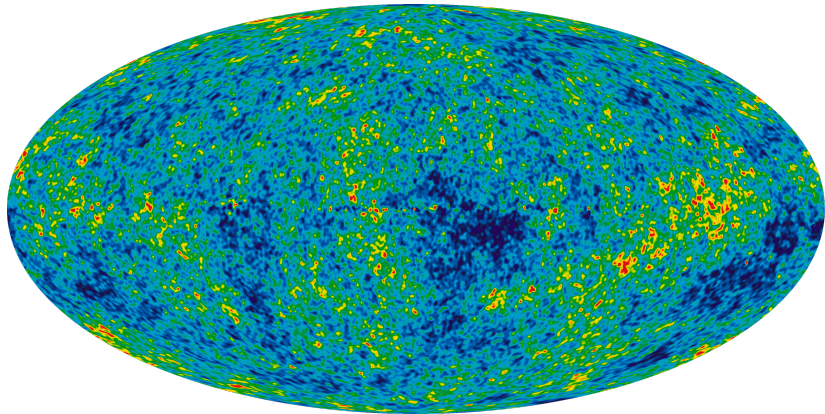


Doppler Shift – measuring astronomical distances

- ▶ We can measure speed accurately
- ▶ How do we measure distance? ⇒ **Parallax**
- ▶ Standard candle ⇒ magnitude → distance



Background radiation



- ▶ Measure Hubble's constant \Rightarrow **Size of the universe**
- ▶ CMB (cosmic microwave background) \Rightarrow Big Bang



Blackbody radiation

Some important questions:

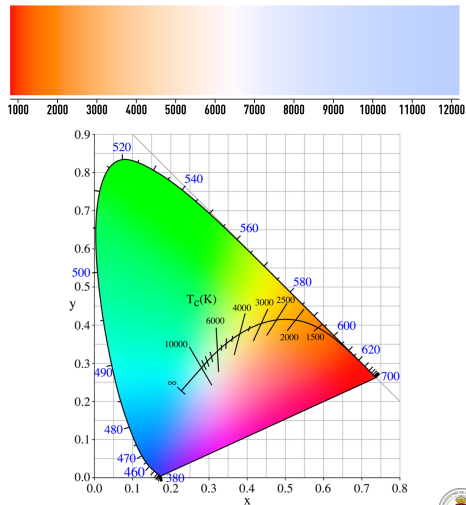
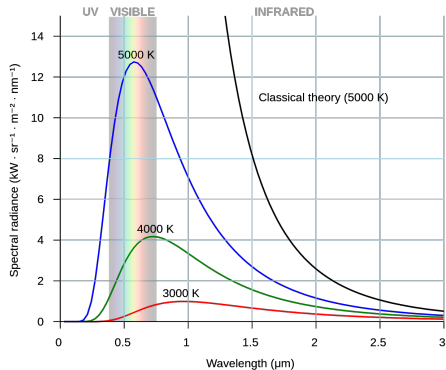
- ▶ Why does lava glow?
- ▶ Why is it red?
- ▶ What glows in blue?

$$B_\nu(\nu, T) = \frac{2h\nu^3}{c^2} \frac{1}{e^{h\nu/kT} - 1}$$

$$L = \frac{2\pi^5}{15} \frac{k^4 T^4}{c^2 h^3} \frac{1}{\pi} = \sigma T^4 \frac{1}{\pi}$$

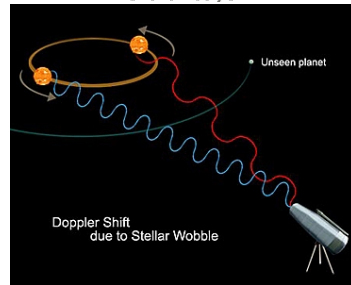
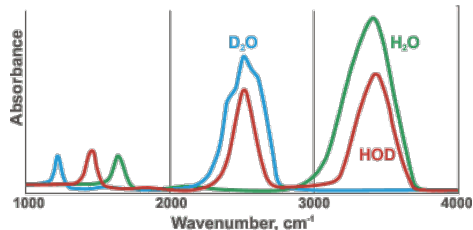
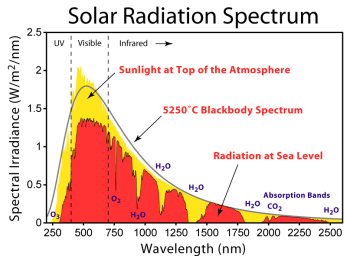


Blackbody radiation



Atmosphere

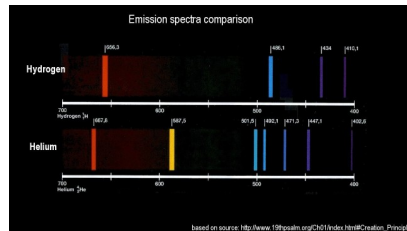
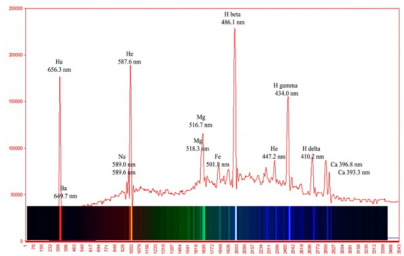
- ▶ Atmosphere blocks some of the blackbody radiation coming from the sun
- ▶ Each element has an absorption spectrum



Helios

- ▶ Subtracting the elements of the atmosphere, the spectrum of the sun is still not the blackbody spectrum, because each element in the sun radiates in different spectrums
- ▶ Helium was discovered in the sun's spectrum and named after it
- ▶ From a single dot in the sky corresponding to a binary star we can tell their mass from doppler shift and orbital mechanics, and each of their composition from the spectrum

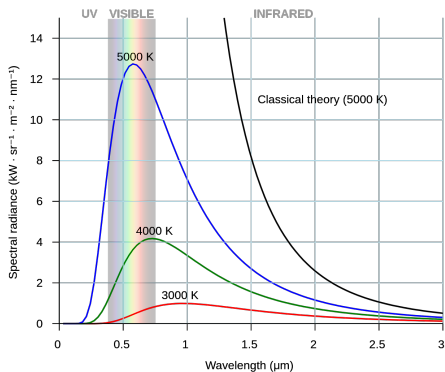
The Solar Chromosphere Spectrum (Flash Spectrum)



based on source: http://www.19thpsalm.org/Ch01/index.html#Creation_Principle



Planck's law (1900)



- ▶ Blackbody radiation curves are really weird
- ▶ Even though their formula was well known, it was lacking explanation until Max Planck.
- ▶ He quantized the amount of possible energy transfer in 1900 which explained the formula.
- ▶ The Planck constant is the smallest possible energy value for a given frequency:

$$h = 6.62607015 \times 10^{-34} \text{ J} \cdot \text{s}$$

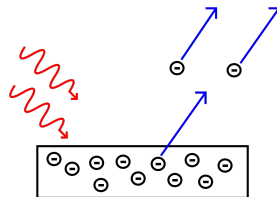
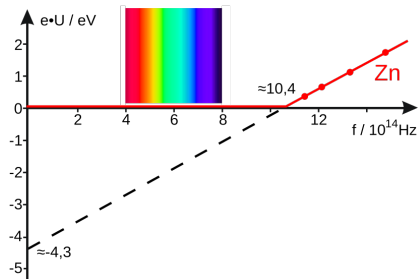
$$E = hf$$

$$B_{\lambda}(\lambda, T) = \frac{2hc^2}{\lambda^5} \frac{1}{e^{\frac{hc}{\lambda k_B T}} - 1}$$



Photoelectric effect

- ▶ Shining light on a metal plate can decouple electrons from the metal
- ▶ Higher frequency radiation has more energy, so it moves more electrons
- ▶ Photoelectric effect is that even high intensity but low frequency light does not generate electric charge in the metal plate
- ▶ This cannot be explained by the classical waveform of light

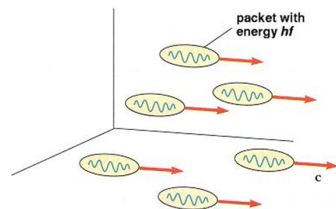


$$E = hf$$



Photoelectric effect

- ▶ Only large enough frequency produces current
- ▶ The effect is instantaneous
- ▶ Current is proportional to intensity
- ▶ Red light in photographic darkrooms
- ▶ Ultraviolet light causes sunburn
- ▶ Photons of different energies trigger different chemical reactions in retina cells



Particle-wave duality

Low frequency \Rightarrow more wave like

High frequency \Rightarrow particle like

Photon is the electromagnetic particle with $E = hf$ energy.



Photoelectric effect

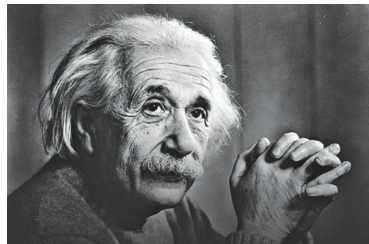
- ▶ Only large enough frequency produces current
- ▶ The effect is instantaneous
- ▶ Current is proportional to intensity
- ▶ Red light in photographic darkrooms
- ▶ Ultraviolet light causes sunburn
- ▶ Photons of different energies trigger different chemical reactions in retina cells

Particle-wave duality

Low frequency \Rightarrow more wave like

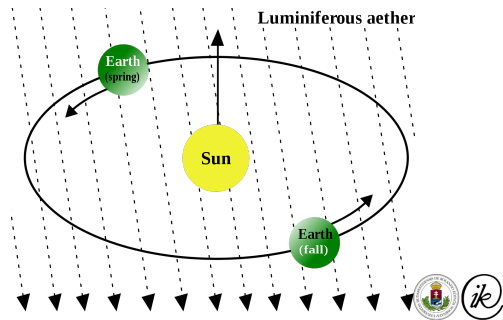
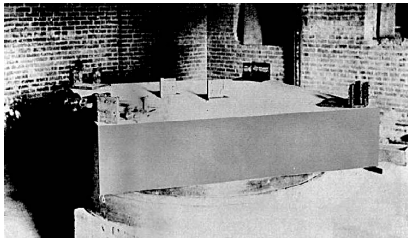
High frequency \Rightarrow particle like

Photon is the electromagnetic particle with $E = hf$ energy.



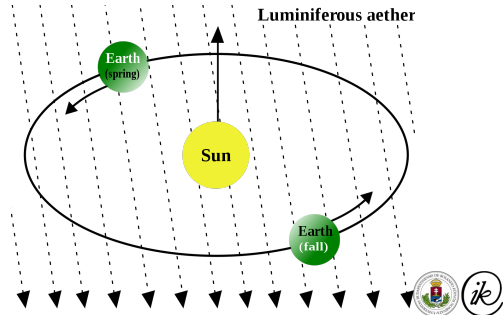
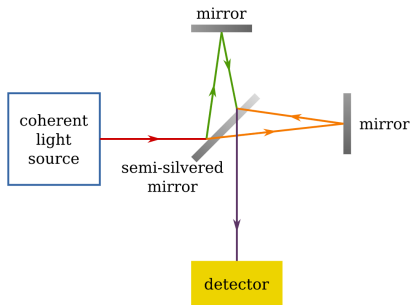
Michelson – Morley (1887)

- ▶ But if light is a wave, what does it propagate in? \implies **Aether?**
- ▶ The Michelson and Morley experiment:
 - ▶ Due the rotation of Earth around the sun and its axis we should be moving at a large speed relative to the cosmic Aether.
 - ▶ The light travelling in different direction and speed should destructively interfere with each other. They did not, and thus making the experiment the most famous null result.



Michelson – Morley (1887)

- ▶ But if light is a wave, what does it propagate in? \implies **Aether?**
- ▶ The Michelson and Morley experiment:
 - ▶ Due the rotation of Earth around the sun and its axis we should be moving at a large speed relative to the cosmic Aether.
 - ▶ The light travelling in different direction and speed should destructively interfere with each other. They did not, and thus making the experiment the most famous null result.



Special relativity

Derived from two postulates

1. The laws of physics are invariant in all inertial frames of reference
2. The speed of light in a vacuum is the same for all observers

It solves:

- ▶ Michelson - Morley null result
- ▶ Inconsistency of Newtonian mechanics and Maxwell's equations

Accurate if **gravitation** is negligible

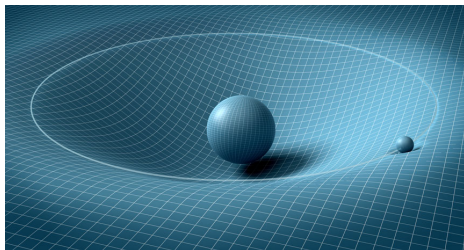


General relativity

Derived from three postulates

1. The laws of physics are invariant in all inertial frames of reference
2. Acceleration cannot be distinguished from gravity
3. The speed of light in a vacuum is the same for all observers

It solves **EVERYTHING**²!

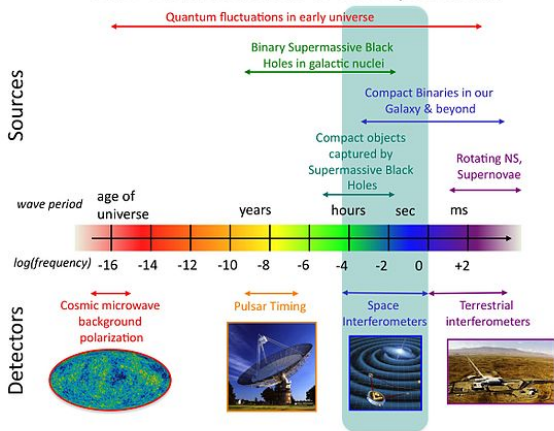


²Except quantum stuff and climate change.

Ripples in Spacetime

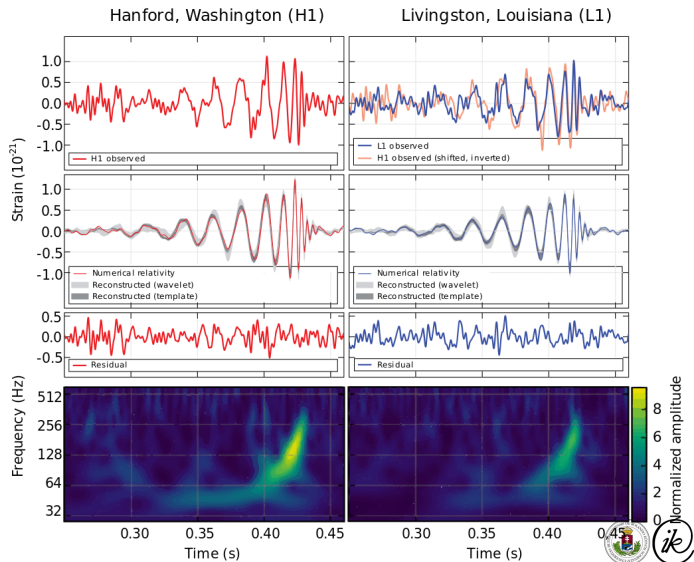
- ▶ Matter affects the shape of the spacetime
- ▶ Spacetime drives how mass can move
- ▶ Spacetime wanna be flat
- ▶ \Rightarrow Gravitational waves
- ▶ \Rightarrow Similar to Michelson - Morley experiment

The Gravitational Wave Spectrum



Ripples in Spacetime

- ▶ Matter affects the shape of the spacetime
- ▶ Spacetime drives how mass can move
- ▶ Spacetime wanna be flat
- ▶ \Rightarrow Gravitational waves
- ▶ \Rightarrow Similar to Michelson – Morley experiment



Thank you for your attention!



Rendering Equation

Csaba **Bálint**
first name family name

csabix@inf.elte.hu

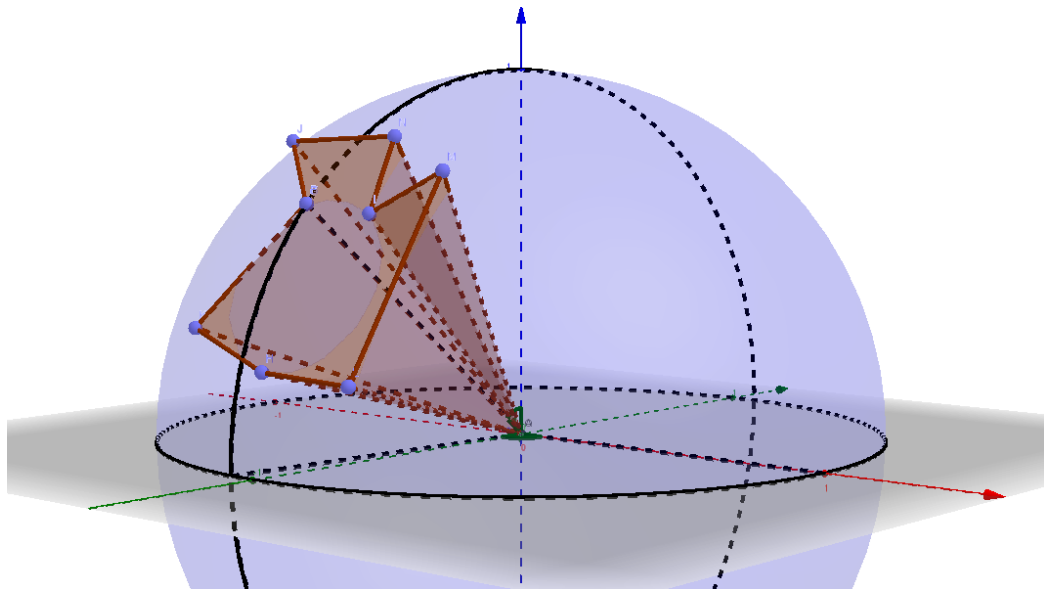
Eötvös Loránd University,
Faculty of Informatics

Computer Graphics Lecture
Budapest 2022

- ▼ Area on a sphere
 - Functions on the unit sphere
 - Spherical coordinates
 - Integrating functions
 - Integral transform theorem
- ▼ Visibility
 - of a differential surface
- ▼ Luminance
 - Flux
 - Radiance
 - Fundamental Law of Photometry
- ▼ Light–Surface interaction
 - Reflectance probability
 - Rendering Equation



Area on sphere (from theory to practice)



Area on sphere

Let us denote the **unit sphere** with

$$\mathbb{S} := \{\mathbf{p} \in \mathbb{R}^3 : \|\mathbf{p}\|_2 = 1\}.$$

How do we measure the area of a set $D \subseteq \mathbb{S}$?



Area on sphere

Let us denote the **unit sphere** with

$$\mathbb{S} := \{\mathbf{p} \in \mathbb{R}^3 : \|\mathbf{p}\|_2 = 1\}.$$

How do we measure the area of a set $D \subseteq \mathbb{S}$?

We denote the **characteristic function** as $\chi_D : \mathbb{S} \rightarrow \{0, 1\}$, such that

$$\chi_D(\omega) := \begin{cases} 0 & \text{if } \omega \notin D \\ 1 & \text{if } \omega \in D \end{cases} \quad (\omega \in \mathbb{S})$$



Area on sphere

Let us denote the **unit sphere** with

$$\mathbb{S} := \{\mathbf{p} \in \mathbb{R}^3 : \|\mathbf{p}\|_2 = 1\}.$$

How do we measure the area of a set $D \subseteq \mathbb{S}$?

We denote the **characteristic function** as $\chi_D : \mathbb{S} \rightarrow \{0, 1\}$, such that

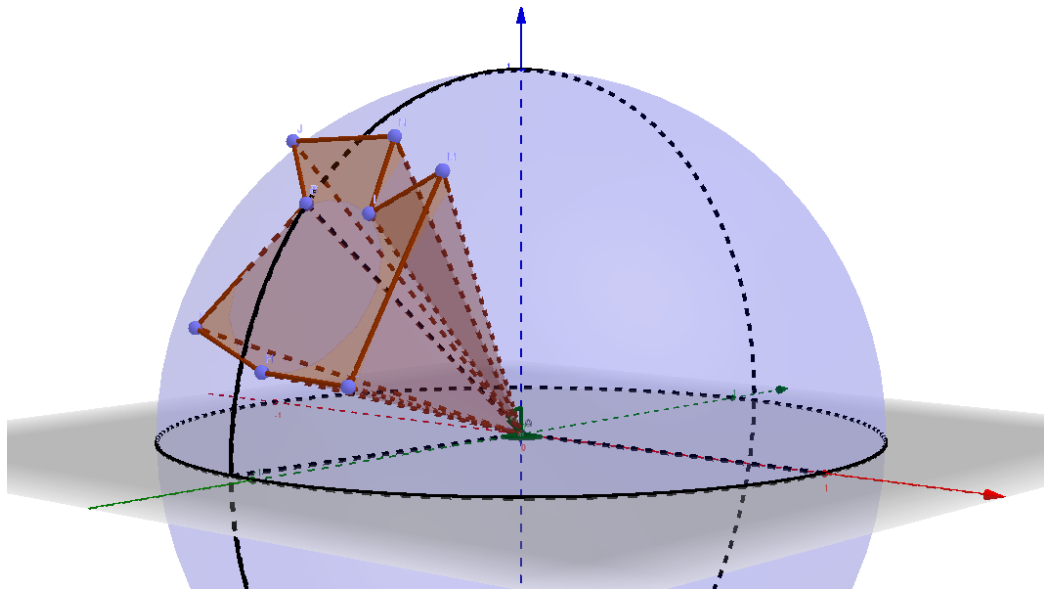
$$\chi_D(\omega) := \begin{cases} 0 & \text{if } \omega \notin D \\ 1 & \text{if } \omega \in D \end{cases} \quad (\omega \in \mathbb{S})$$

The area of the D is

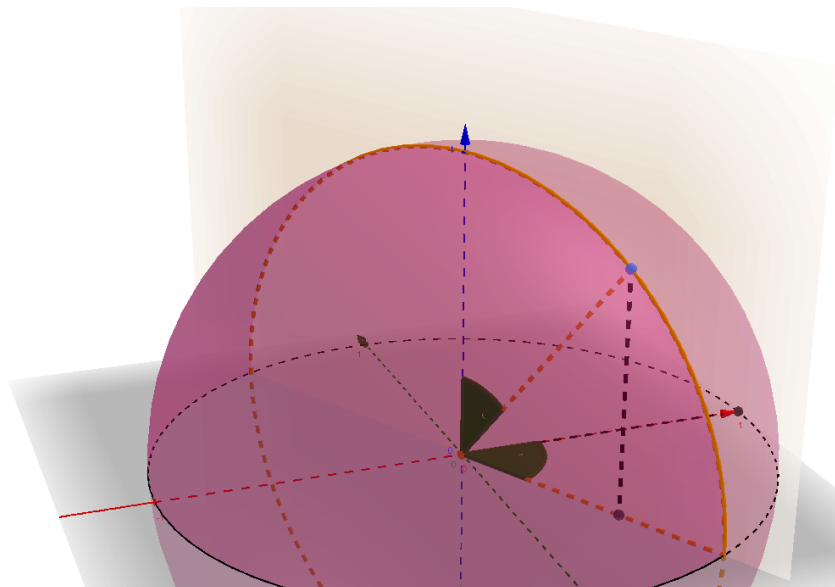
$$\int_{\mathbb{S}} \chi_D = \int_{\mathbb{S}} \chi_D(\omega) d\omega = ??$$



Represent surface area as a volume using ratios!



Spherical to Cartesian coordinate system I



Spherical to Cartesian coordinate system II

$$T(\varphi, \theta, r) := (r \cos(\varphi) \sin(\theta), r \sin(\varphi) \sin(\theta), r \cos(\theta))$$

- ▶ Homework: $|\det T'(\varphi, \theta, r)| = r^2 \sin(\theta)$
- ▶ We could use this transformation to calculate the surface area of D , but how?



Spherical to Cartesian coordinate system II

$$T(\varphi, \theta, r) := (r \cos(\varphi) \sin(\theta), r \sin(\varphi) \sin(\theta), r \cos(\theta))$$

- ▶ Homework: $|\det T'(\varphi, \theta, r)| = r^2 \sin(\theta)$
- ▶ We could use this transformation to calculate the surface area of D , but how?
- ▶ Let us calculate the **volume** from D to the origin, a cone-like shape.
- ▶ The area A of D can be calculated from the ratio of volume relative to the volume of the unit ball (so $r = 1$):

$$A = \frac{V}{\frac{4}{3}r^3\pi} \cdot 4r^2\pi = 3 \cdot V$$



Homework

$$\begin{aligned}
 \det T'(\varphi, \theta, r) &= \det \begin{bmatrix} r \cos \varphi \sin \theta \\ r \sin \varphi \sin \theta \\ r \cos \theta \end{bmatrix}' = \det \begin{bmatrix} -r \sin \varphi \sin \theta & r \cos \varphi \cos \theta & \cos \varphi \sin \theta \\ r \cos \varphi \sin \theta & r \sin \varphi \cos \theta & \sin \varphi \sin \theta \\ 0 & -r \sin \theta & \cos \theta \end{bmatrix} = \\
 &= r^2 \det \begin{bmatrix} -\sin \varphi \sin \theta & \cos \varphi \cos \theta & \cos \varphi \sin \theta \\ \cos \varphi \sin \theta & \sin \varphi \cos \theta & \sin \varphi \sin \theta \\ 0 & \boxed{-\sin \theta} & \boxed{\cos \theta} \end{bmatrix} = \\
 &= r^2 \cdot \left(\sin \theta \sin^2 \theta \cdot \det \begin{bmatrix} -\sin \varphi & \cos \varphi \\ \cos \varphi & \sin \varphi \end{bmatrix} + \cos \theta (\sin \theta \cos \theta) \cdot \det \begin{bmatrix} -\sin \varphi & \cos \varphi \\ \cos \varphi & \sin \varphi \end{bmatrix} \right) = \\
 &= r^2 \sin \theta (\sin^2 \theta + \cos^2 \theta) \cdot \det \begin{bmatrix} -\sin \varphi & \cos \varphi \\ \cos \varphi & \sin \varphi \end{bmatrix} = -r^2 \sin \theta
 \end{aligned}$$



Generalize to functions

Let us generalize \mathcal{X}_D to functions like $f : [0, 2\pi) \times [0, \pi) \rightarrow \mathbb{R}$ that **operate in spherical coordinates**.

- ▶ Let us create an $F : \mathbb{R}^3 \rightarrow \mathbb{R}$ function that takes values from this domain:

$$F(x, y, z) := f([T^{-1}(x, y, z)]_{1,2})$$

- ▶ So we evaluate f after transforming back to spherical coordinates without the radius
- ▶ Now, the volume can be expressed as:

$$V = \int_{\text{unit ball}} F(x, y, z) \, dx \, dy \, dz$$

- ▶ Remember integration by substitution? (Change of Variables Theorem)



Integral transform theorem

Assume that $U \subseteq \mathbb{R}^n$ is an open set, $T : U \rightarrow \mathbb{R}^n$, and $f \in \mathbb{R}^n \rightarrow \mathbb{R}$. If

Then

$$\int_{T(U)} f(\mathbf{v}) \, d\mathbf{v} = \int_U f(T(\mathbf{u})) \cdot |\det T'(\mathbf{u})| \, d\mathbf{u} .$$



Integral transform theorem

Assume that $U \subseteq \mathbb{R}^n$ is an open set, $T : U \rightarrow \mathbb{R}^n$, and $f \in \mathbb{R}^n \rightarrow \mathbb{R}$. If

Then

$$\int_{T(U)} f(\mathbf{v}) \, d\mathbf{v} = \int_U f(T(\mathbf{u})) \cdot |\det T'(\mathbf{u})| \, d\mathbf{u} .$$

-
- ▶ The transformed set is $T(U) = \{T(\mathbf{u}) : \mathbf{u} \in U\}$.
 - ▶ $T'(\mathbf{u}) \in \mathbb{R}^{n \times n}$ is the Jacobian matrix of the transformation at $\mathbf{u} \in \mathbb{R}^n$.
 - ▶ The $|\det T'(\mathbf{u})| \in \mathbb{R}$ is by how much a unit volume is stretched.



Integral transform theorem

Assume that $U \subseteq \mathbb{R}^n$ is an open set, $T : U \rightarrow \mathbb{R}^n$, and $f \in \mathbb{R}^n \rightarrow \mathbb{R}$. If

- ▶ $T : U \rightarrow \mathbb{R}^n$ is injective and continuously differentiable function such that

$$\forall \mathbf{u} \in U : \det T'(\mathbf{u}) \neq 0 .$$

- ▶ The function $f : T(U) \rightarrow \mathbb{R}$ is continuous and has a compact support.

Then

$$\int_{T(U)} f(\mathbf{v}) \, d\mathbf{v} = \int_U f(T(\mathbf{u})) \cdot |\det T'(\mathbf{u})| \, d\mathbf{u} .$$

-
- ▶ The transformed set is $T(U) = \{T(\mathbf{u}) : \mathbf{u} \in U\}$.
 - ▶ $T'(\mathbf{u}) \in \mathbb{R}^{n \times n}$ is the Jacobian matrix of the transformation at $\mathbf{u} \in \mathbb{R}^n$.
 - ▶ The $|\det T'(u)| \in \mathbb{R}$ is by how much a unit volume is stretched.



Integral transform theorem – applied

Let

$$U := (0, 2\pi) \times (0, \pi) \times (0, 1) \subseteq \mathbb{R}^n ,$$

and

$$T(\varphi, \theta, r) := (r \cos(\varphi) \sin(\theta), r \sin(\varphi) \sin(\theta), r \cos(\theta)) ,$$

then

$$T(U) = \{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 < 1\} = \text{open unit ball.}$$

Check theorem requirements!



Transforming the integral

Remember that $f : [0, 2\pi) \times [0, \pi] \rightarrow \mathbb{R}$, then $F(x, y, z) := f([T^{-1}(x, y, z)]_{1,2})$.

$$V = \int_{\text{unit ball}} F(x, y, z) \, dx \, dy \, dz = \int_U F(T(\varphi, \theta, r)) \cdot |\det T'(\varphi, \theta, r)| \, dr \, d\theta \, d\varphi$$

- ▶ Where $F(T(\varphi, \theta, r)) = f([T^{-1}(T(\varphi, \theta, r))]_{1,2}) = f(\varphi, \theta)$
- ▶ Note that $|\det T'(\varphi, \theta, r)| = r^2 \sin(\theta)$

$$V = \int_{[0,2\pi] \times [0,\pi] \times [0,1]} r^2 \cdot \sin(\theta) \cdot f(\varphi, \theta) \, dr \, d\theta \, d\varphi = \int_0^1 r^2 \, dr \cdot \int_{[0,2\pi] \times [0,\pi]} f(\varphi, \theta) \sin(\theta) \, d\theta \, d\varphi$$



The new measure

- ▶ Since $\int_0^1 r^2 dr = \frac{1}{3}$,

$$A = 3V = \int_{[0,2\pi] \times [0,\pi]} f(\varphi, \theta) \sin(\theta) d\theta d\varphi$$

- ▶ For a range of directions we often write $\omega = (\varphi, \theta)$ instead of the pair of angles.
- ▶ However, the meaning of $d\omega$ is different now:

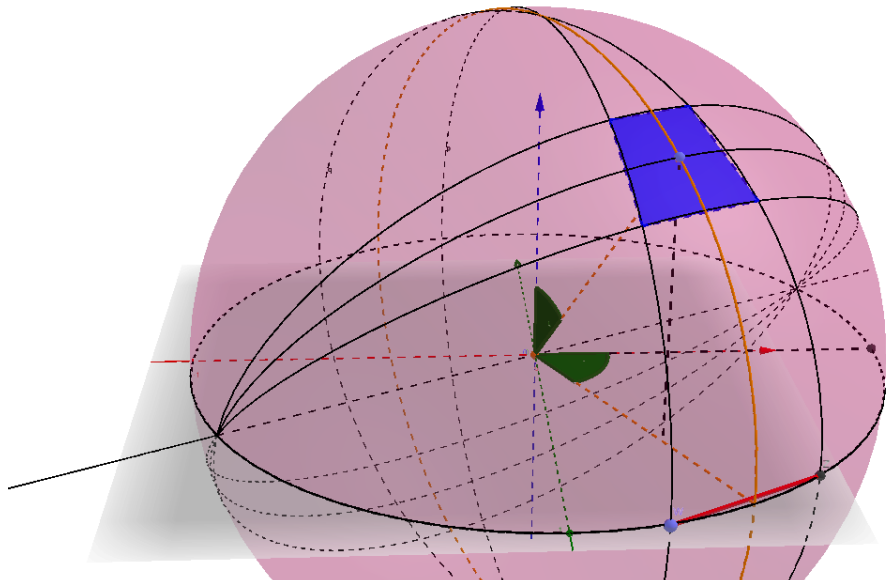
$$d\omega = \sin(\theta) d\theta d\varphi$$

- ▶ Ultimately, we can express integrals over a range of directions as

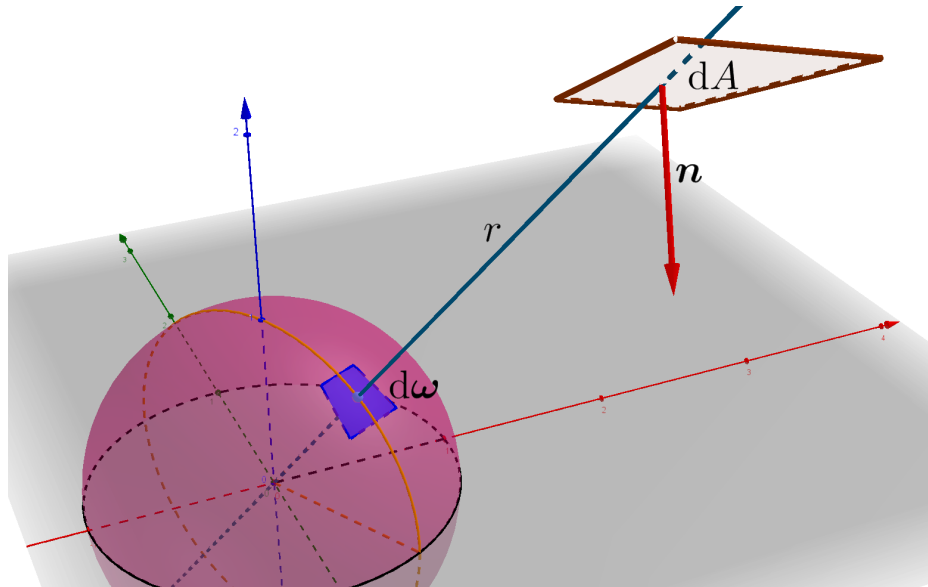
$$\int_D f(\omega) d\omega$$



Another explanation



Visibility of a differential surface



Visibility of a differential surface

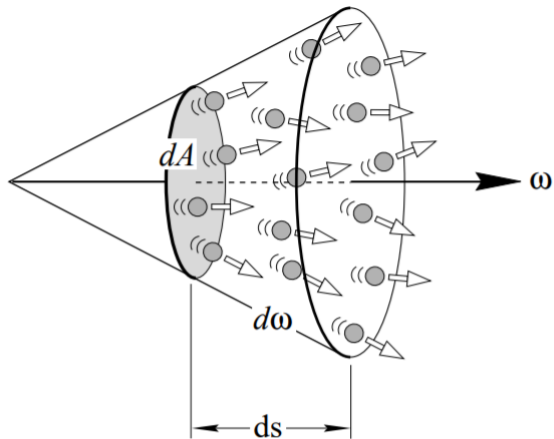
The **solid angle** at which the surface point \mathbf{x} with surface normal \mathbf{n} and differential area dA is visible from **the origin** is:

$$\boxed{d\omega = \frac{\cos \theta_n}{r^2} dA} = \frac{\langle \mathbf{n}, \mathbf{x} \rangle}{\|\mathbf{x}\|^3} dA$$

This means we can transform an arbitrary surface integral onto a sphere, and vice versa.



Phase space flux



Phase space flux

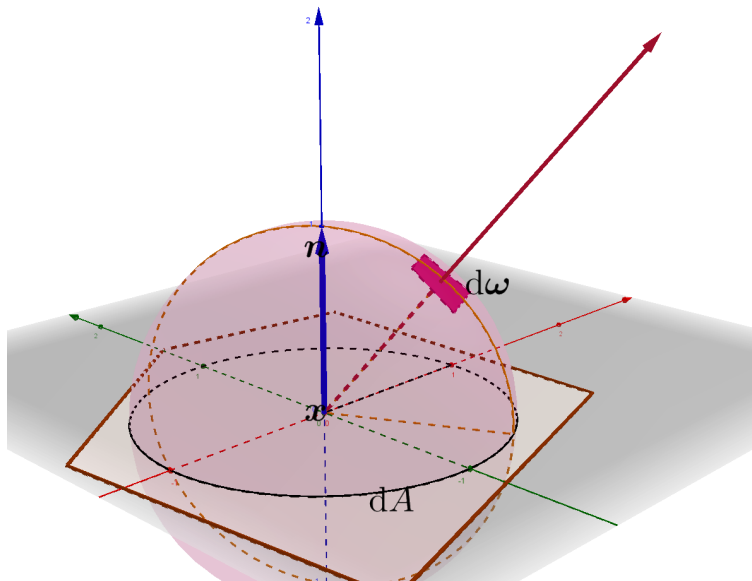
Spectral flux is the light energy emitted per unit time and wavelength by given surface point

$$\Phi_{\lambda,t}(\mathbf{x}) := \frac{\partial^2 Q_{\partial\mathbf{x}}(\mathbf{x}, t, \lambda)}{\partial t \partial \lambda} = \frac{\partial^3 Q(\mathbf{x}, t, \lambda)}{\partial \mathbf{x} \partial t \partial \lambda}$$

- ▶ that describes light energy density with respect to time and wavelength
- ▶ $Q_{\partial\mathbf{x}}(\mathbf{x}, t)$ is the light energy emitted at a given position and time towards $\partial\mathbf{x}$ direction
- ▶ λ is the wavelength of the light emitted
- ▶ Spectral because of the $\partial\lambda$, otherwise radial flux



Radiance or Luminance



Radiance or Luminance

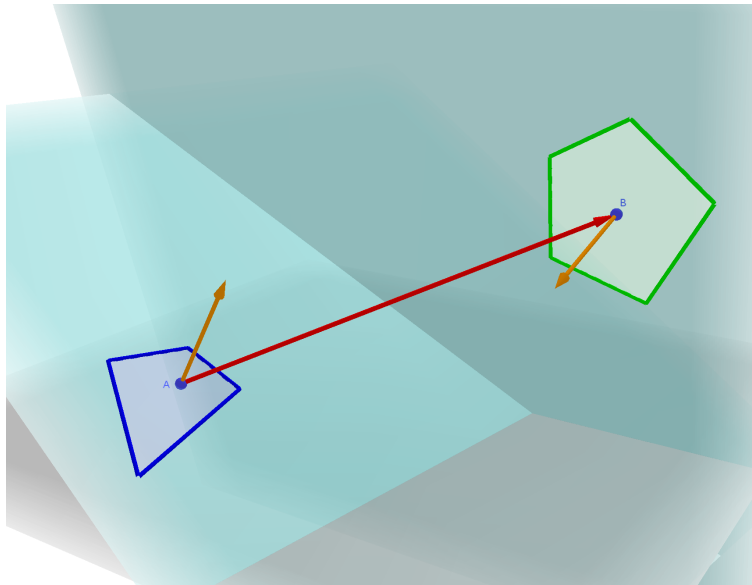
Radiance or **Luminance** is the flux emitted by a unit area of the surface under unit solid angle

$$L_{\lambda,t}(\mathbf{x}, \boldsymbol{\omega}) := \frac{d^2\Phi_{\lambda,t}}{d\boldsymbol{\omega} dA \cos \theta_n}$$

- ▶ $\Phi_{\lambda,t}(\mathbf{x})$ is the spectral flux
- ▶ $dA \cos \theta_n$ is the area we project the flux to
- ▶ First quantity that makes physical and mathematical sense by itself



Fundamental Law of Photometry



Fundamental Law of Photometry

Definition of luminance:

$$L_{\lambda,t}(\mathbf{x}, \boldsymbol{\omega}) = \frac{d^2\Phi_{\lambda,t}}{d\boldsymbol{\omega} dA \cos \theta_n}$$

Visibility of the other differential area from in a solid angle: (see Frame 16)

$$d\boldsymbol{\omega} = \frac{\cos \theta'_n}{r^2} dA'$$

implies

$$L_{\lambda,t}(\mathbf{x}, \boldsymbol{\omega}) = \frac{r^2 \cdot d^2\Phi_{\lambda,t}}{dA' \cos \theta'_n \cdot dA \cos \theta_n} = L_{\lambda,t}(\mathbf{x}', \boldsymbol{\omega}')$$



Fundamental Law of Photometry

Definition of luminance:

$$L_{\lambda,t}(\mathbf{x}, \boldsymbol{\omega}) = \frac{d^2\Phi_{\lambda,t}}{d\boldsymbol{\omega} dA \cos \theta_n}$$

Visibility of the other differential area from in a solid angle: (see Frame 16)

$$d\boldsymbol{\omega} = \frac{\cos \theta'_n}{r^2} dA'$$

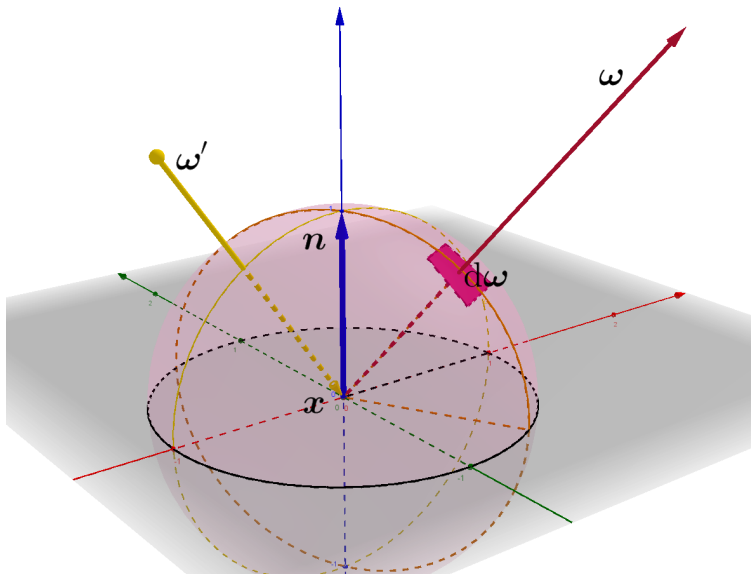
implies

$$L_{\lambda,t}(\mathbf{x}, \boldsymbol{\omega}) = \frac{r^2 \cdot d^2\Phi_{\lambda,t}}{dA' \cos \theta'_n \cdot dA \cos \theta_n} = L_{\lambda,t}(\mathbf{x}', \boldsymbol{\omega}')$$

Transmitted radiance (or luminance) from A to B is the same as from B to A!



Light-Surface interaction



Light-Surface interaction

Reflectance probability density function

$$w_{t,\lambda}(\omega', \mathbf{x}, \omega) = \Pr\{\text{photon hits } \omega \pm d\omega \text{ solid angle} \mid \text{coming from } \omega'\} \in [0, 1]$$

- ▶ Probability of a photon going towards ω and its ω vicinity if it comes from ω' direction.
- ▶ The distribution depends on the material properties at position \mathbf{x}
- ▶ Energy conservation: $\int_{\mathbb{S}} w_{\lambda,t}(\omega', \mathbf{x}, \omega) d\omega \leq 1$
- ▶ However, this is possible: $\int_{\mathbb{S}} w_{\lambda,t}(\omega', \mathbf{x}, \omega) d\omega' \geq 1$



Reflected Flux

$$\Phi_{t,\lambda}^{\mathbf{x} \rightarrow \omega} = \Phi_{t,\lambda}^{em} + \int_{\mathbb{S}} w_{t,\lambda}(\omega', \mathbf{x}, \omega) \cdot \Phi_{t,\lambda}^{\omega' \rightarrow \mathbf{x}} d\omega'$$

- ▶ Total light coming from the surface is the emitted light plus the total reflected light.
- ▶ $\Phi_{t,\lambda}^{\mathbf{x} \rightarrow \omega} = L_{t,\lambda}(\mathbf{x}, \omega) dA \cos \theta d\omega$
- ▶ $\Phi_{t,\lambda}^{em} = L_{t,\lambda}^{em}(\mathbf{x}, \omega) dA \cos \theta d\omega$
- ▶ $\Phi_{t,\lambda}^{\omega' \rightarrow \mathbf{x}} = L_{t,\lambda}(V(\mathbf{x}, -\omega'), \omega') dA \cos \theta' d\omega'$
- ▶ The function $V(\mathbf{x}, -\omega') \in \mathbb{R}^3$ is the **visibility function**, it queries the closest point towards $-\omega'$ from point \mathbf{x} .



Rendering Equation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_{\mathbb{S}} L_{t,\lambda}(V(\mathbf{x}, -\boldsymbol{\omega}'), \boldsymbol{\omega}') \cdot \frac{w_{t,\lambda}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega})}{\cos \theta} \cdot \cos \theta' d\boldsymbol{\omega}'$$

The **B**idirectional **R**eflectance **D**istribution **F**unction is the following

$$f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) := \frac{w_{t,\lambda}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega})}{\cos \theta}$$

The rendering equation can be summarized as

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_{\mathbb{S}} L_{t,\lambda}(V(\mathbf{x}, -\boldsymbol{\omega}'), \boldsymbol{\omega}') \cdot f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot \cos \theta' d\boldsymbol{\omega}'$$



Rendering Equation

- ▶ Recursive integral equation

$$L = L^{em} + \tau L$$

- ▶ Very hard to solve
- ▶ Algorithms provide approximate solution
- ▶ Error? Convergence? Artifacts?

To be continued...



Bidirectional Reflectance Distribution Function

Csaba **Bálint**
first name family name

csabix@inf.elte.hu

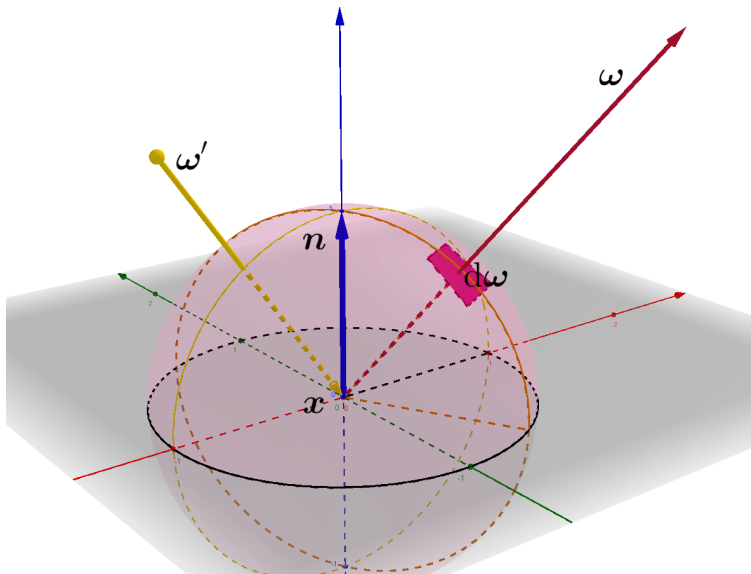
Eötvös Loránd University,
Faculty of Informatics

Computer Graphics Lecture
Budapest 2022

- ▼ Rendering equation revised
 - Reflectance probability density function
 - Rendering equation
- ▼ Bidirectional Reflectance Distribution Function
 - Definition and modelling
 - Diffuse surface – Lambert's law
- ▼ Specular reflectance
 - Phong model
 - Blinn-Phong model
- ▼ Fresnel refraction
 - Refraction and IOR
 - Fresnel equations and approximation
- ▼ Cook-Torrance model
 - Roughness models
 - Geometric Attenuation Factor



Reflectance probability density function



Reflectance probability density function

Reflectance probability density function

$$w_{t,\lambda}(\omega', \mathbf{x}, \omega) = \Pr\{\text{photon hits } \omega \pm d\omega \text{ solid angle} \mid \text{coming from } \omega'\} \in [0, 1]$$

- ▶ Probability of a photon going towards ω and its ω vicinity if it comes from ω' direction.
- ▶ The distribution depends on the material properties at position \mathbf{x}
- ▶ Energy conservation: $\int_{\mathbb{S}} w_{\lambda,t}(\omega', \mathbf{x}, \omega) d\omega \leq 1$
- ▶ However, this is possible: $\int_{\mathbb{S}} w_{\lambda,t}(\omega', \mathbf{x}, \omega) d\omega' \geq 1$



From flux to rendering equation

$$\Phi_{t,\lambda}^{\mathbf{x} \rightarrow \omega} = \Phi_{t,\lambda}^{em} + \int_{\mathbb{S}} w_{t,\lambda}(\omega', \mathbf{x}, \omega) \cdot \Phi_{t,\lambda}^{\omega' \rightarrow \mathbf{x}} d\omega'$$

- ▶ Total light coming from the surface is the emitted light plus the total reflected light.
- ▶ **Visibility function** $V(\mathbf{x}, -\omega') \in \mathbb{R}^3$: queries the closest point towards $-\omega'$ from point \mathbf{x} .
- ▶ Divide the equation by $dA \cos \theta d\omega$ to get the rendering equation

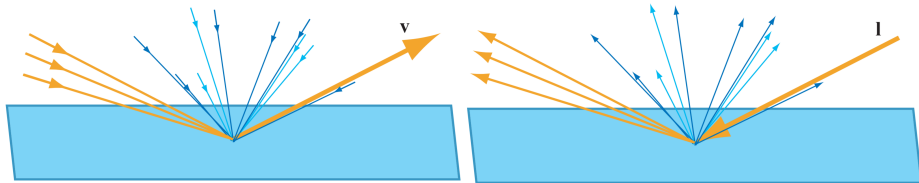
$$L_{t,\lambda}(\mathbf{x}, \omega) = L_{t,\lambda}^{em}(\mathbf{x}, \omega) + \int_{\mathbb{S}} L_{t,\lambda}(V(\mathbf{x}, -\omega'), \omega') \cdot \frac{w_{t,\lambda}(\omega', \mathbf{x}, \omega)}{\cos \theta} \cdot \cos \theta' d\omega'$$

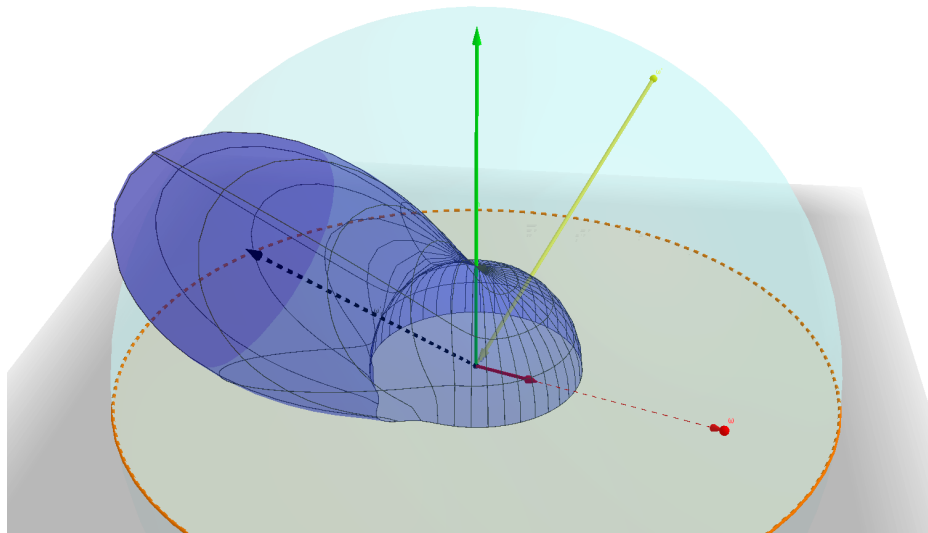


The Bidirectional Reflectance Distribution Function

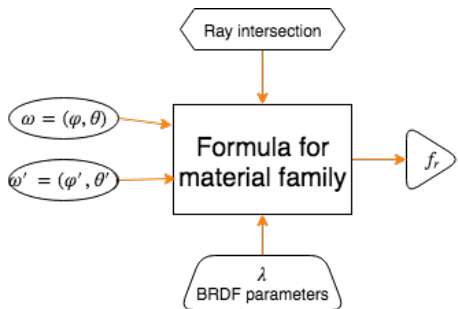
$$f_{r_{t,\lambda}}(\omega', \mathbf{x}, \omega) := \frac{w_{t,\lambda}(\omega', \mathbf{x}, \omega)}{\cos \theta} = \frac{w_{t,\lambda}(\omega', \mathbf{x}, \omega)}{\langle \mathbf{n}, \omega \rangle}$$

- ▶ The BRDF is an input function to the rendering equation solver engine that describes **materials properties**
- ▶ \mathbf{x} characterizes the position on the surface with surface normal \mathbf{n} .
- ▶ \mathbf{x} will tell us which object is \mathbf{x} on, and where are we on it (texture fetch).
- ▶ ω, ω' directions describe a direction of a possible bounce of photons





Modelling BRDF



- ▶ We can measure the BRDF as a function of $\varphi, \theta, \varphi', \theta', \lambda$
- ▶ Memory issues: $100 \times 100 \times 100 \times 100 \times 10$ data points.
- ▶ **Isotropic material**: invariant under rotating both ω and ω' around surface normal n
- ▶ **Anisotropic material**: CD, polished metal



Gonioreflectometer

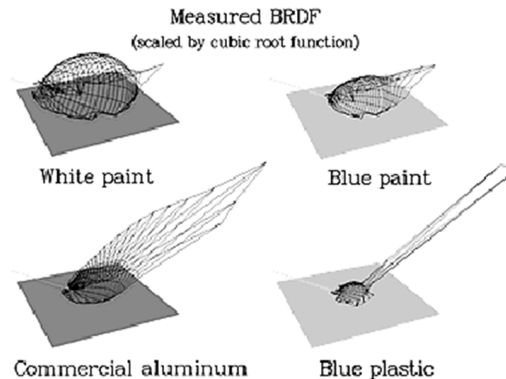
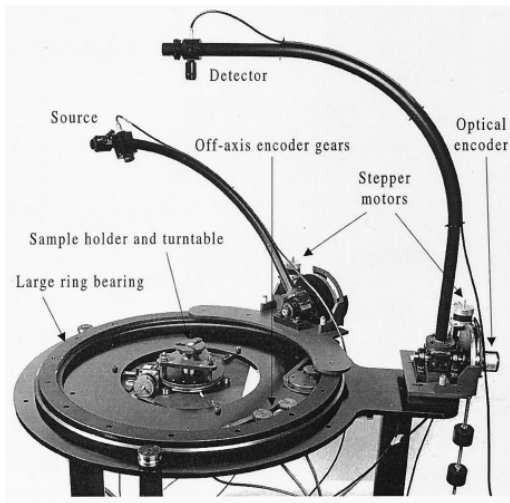


Figure 8: Measured BRDF for four isotropic materials.

Figure 8 taken from "A Framework for Realistic Image Synthesis" by Greenberg, et. al. (Cornell)



Properties of BRDF

1. Positive: $f_{r_{t,\lambda}}(\omega', \mathbf{x}, \omega) \geq 0$
2. Symmetric – Helmholtz law: $f_{r_{t,\lambda}}(\omega', \mathbf{x}, \omega) = f_{r_{t,\lambda}}(\omega, \mathbf{x}, \omega')$
3. Energy conservation: $a(\omega, \mathbf{x}) \leq 1$

Albedo: Probability of a photon being reflected (to anywhere)

$$a(\omega, \mathbf{x}) := \int_{\mathbb{S}} w_{\lambda,t}(\omega, \mathbf{x}, \omega') d\omega' = \int_{\mathbb{S}} f_{r_{t,\lambda}}(\omega, \mathbf{x}, \omega') \cos \theta' d\omega'$$

- ▶ Symmetry implies that viewed from ω direction a homogeneous skylight with radiance of 1 will produce $a(\omega, \mathbf{x})$ radiance along the ray.
- ▶ $a(\mathbf{x}, \omega) = \tau 1 = \int_{\mathbb{S}} f_{r_{t,\lambda}}(\omega, \mathbf{x}, \omega') \cos \theta' d\omega'$



Diffuse reflectance

- ▶ Material is opaque, i.e. $f_{rt,\lambda}((\varphi, \theta), \mathbf{x}, (\varphi', \theta')) = 0$ if $(\theta - \frac{\pi}{2}) \cdot (\theta' - \frac{\pi}{2}) < 0$
- ▶ Reflected radiance is invariant on viewing angle ω
- ▶ Helmholtz law implies that it is invariant on illumination angle ω'
- ▶ Thus, it is a constant: $f_{rt,\lambda}(\omega, \mathbf{x}, \omega') = k_d(\lambda)$ if $(\theta - \frac{\pi}{2}) \cdot (\theta' - \frac{\pi}{2}) > 0$

1. Positive: $0 < k_d(\lambda)$
2. Symmetric
3. Energy conservation:

$$a(\omega, \mathbf{x}) = \int_{\mathbb{S}} f_{rt,\lambda}(\omega, \mathbf{x}, \omega') \cos \theta' d\omega' = \int_0^{\frac{\pi}{2}} \int_0^{2\pi} k_d(\lambda) \cos \theta \sin \theta d\varphi d\theta \leq 1$$



Diffuse energy conservation

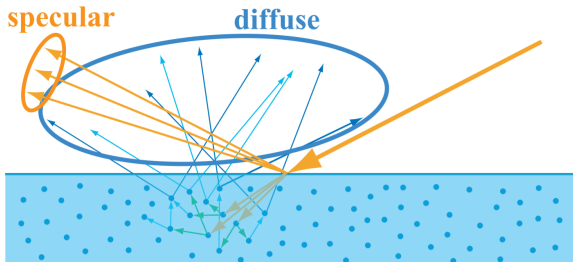
$$f_{r_{t,\lambda}}((\varphi, \theta), \mathbf{x}, (\varphi', \theta')) = \begin{cases} k_d(\lambda) & \text{if } (\theta - \frac{\pi}{2}) \cdot (\theta' - \frac{\pi}{2}) \geq 0 \\ 0 & \text{if } (\theta - \frac{\pi}{2}) \cdot (\theta' - \frac{\pi}{2}) < 0 \end{cases}$$

$$\begin{aligned} a(\boldsymbol{\omega}, \mathbf{x}) &= \int_{\mathbb{S}} f_{r_{t,\lambda}}(\boldsymbol{\omega}, \mathbf{x}, \boldsymbol{\omega}') \cos \theta' d\boldsymbol{\omega}' = \int_0^{\frac{\pi}{2}} \int_0^{2\pi} k_d(\lambda) \cos \theta \sin \theta d\varphi d\theta = \\ &= \int_0^{2\pi} k_d(\lambda) d\varphi \cdot \int_0^{\frac{\pi}{2}} \cos \theta \sin \theta d\theta = 2\pi k_d(\lambda) \cdot \left[\frac{1}{2} \sin^2(\theta) \right]_0^{\frac{\pi}{2}} = \\ &= \pi k_d(\lambda) \leq 1 \end{aligned}$$



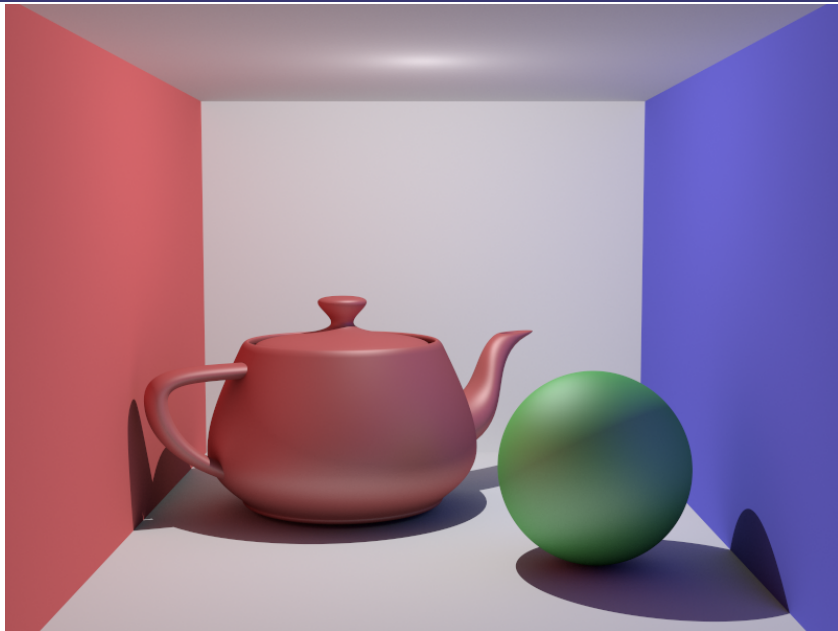
Lambertian reflectance

$$f_{r_{t,\lambda}}((\varphi, \theta), \mathbf{x}, (\varphi', \theta')) = \begin{cases} k_d(\lambda) & \text{if } (\theta - \frac{\pi}{2}) \cdot (\theta' - \frac{\pi}{2}) \geq 0 \\ 0 & \text{if } (\theta - \frac{\pi}{2}) \cdot (\theta' - \frac{\pi}{2}) < 0 \end{cases}$$



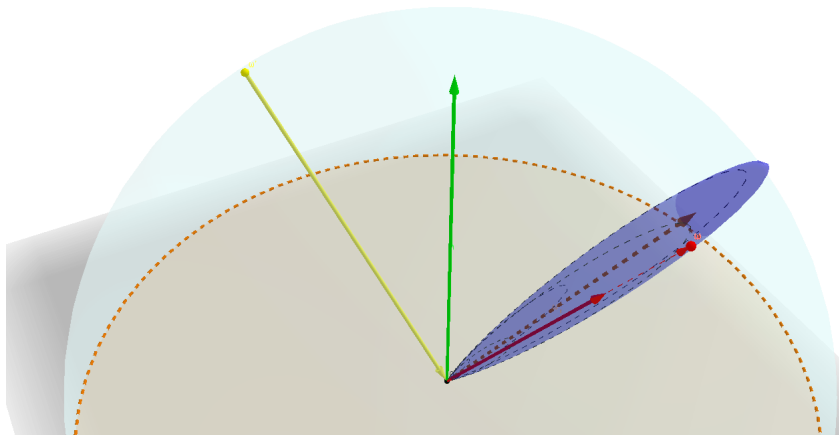
1. Positive: $0 < k_d(\lambda)$
2. Symmetric
3. Energy conservation: $k_d(\lambda) \leq \frac{1}{\pi} \iff a(\omega, \mathbf{x}) \leq 1$





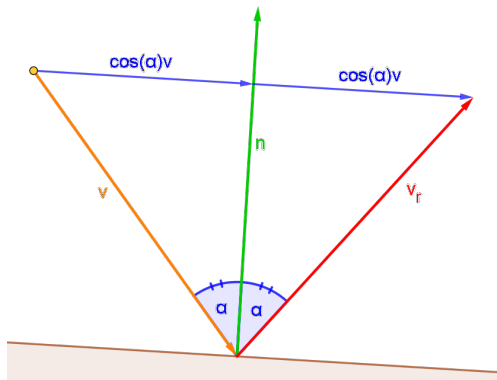
Mirror

$$f_{r_t, \lambda}(\mathbf{l}, \mathbf{v}) = k_r(\lambda) \cdot \frac{\delta(\mathbf{v} - \mathbf{l}_r)}{\langle \mathbf{n}, \mathbf{l} \rangle}$$



Reflect – Householder transformation

$$\cos \alpha = -\langle \mathbf{v}, \mathbf{n} \rangle \quad \mathbf{v}_r = \mathbf{v} + 2 \cos \alpha \cdot \mathbf{n}$$

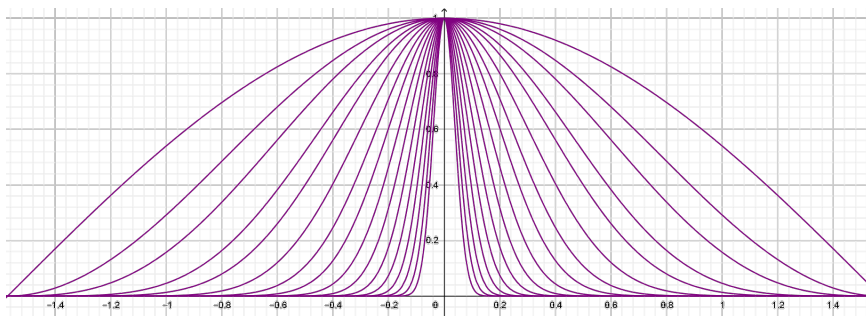


$$\mathbf{v}_r = \mathbf{v} - 2\langle \mathbf{v}, \mathbf{n} \rangle \cdot \mathbf{n}$$



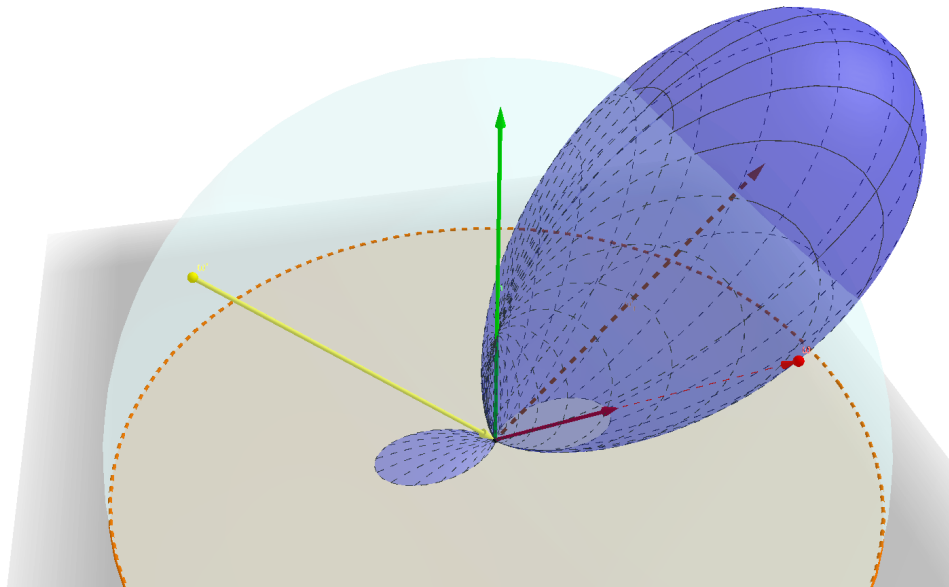
Phong model

$$\delta(x) \approx \cos^N(x) \quad (x \in [-\pi, \pi])$$



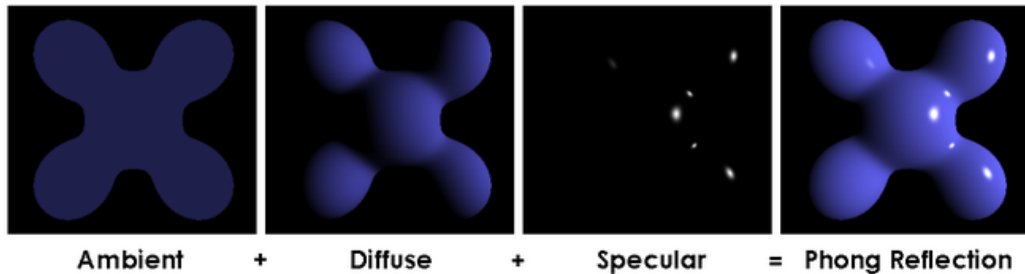
$$f_{r_t, \lambda}(\mathbf{l}, \mathbf{v}) = k_r(\lambda) \cdot \langle \mathbf{v}, \mathbf{l} - 2\langle \mathbf{n}, \mathbf{l} \rangle \mathbf{n} \rangle^N$$

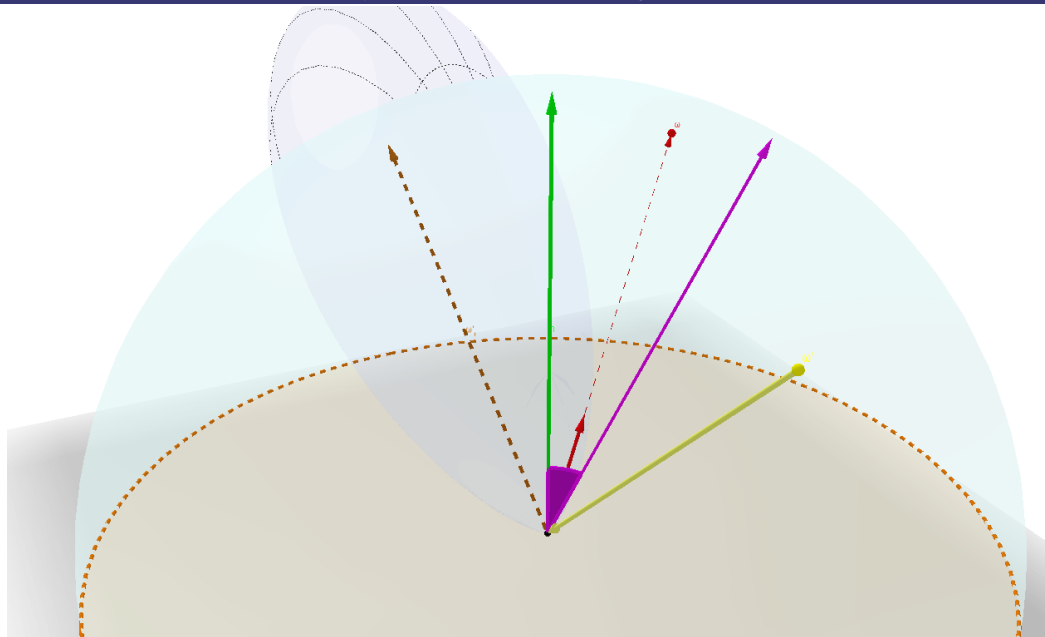




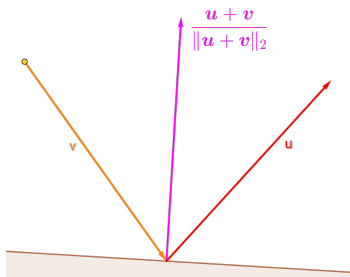
The Phong model

- ▶ Produces specular highlights
- ▶ Very fast to compute
- ▶ Asymmetric :(





Blinn-Phong model

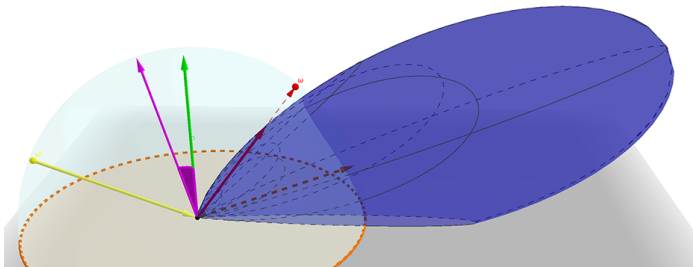
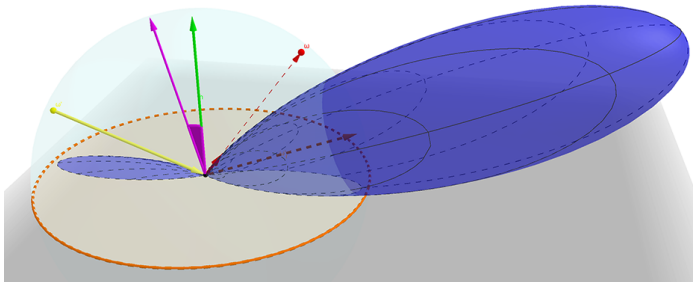


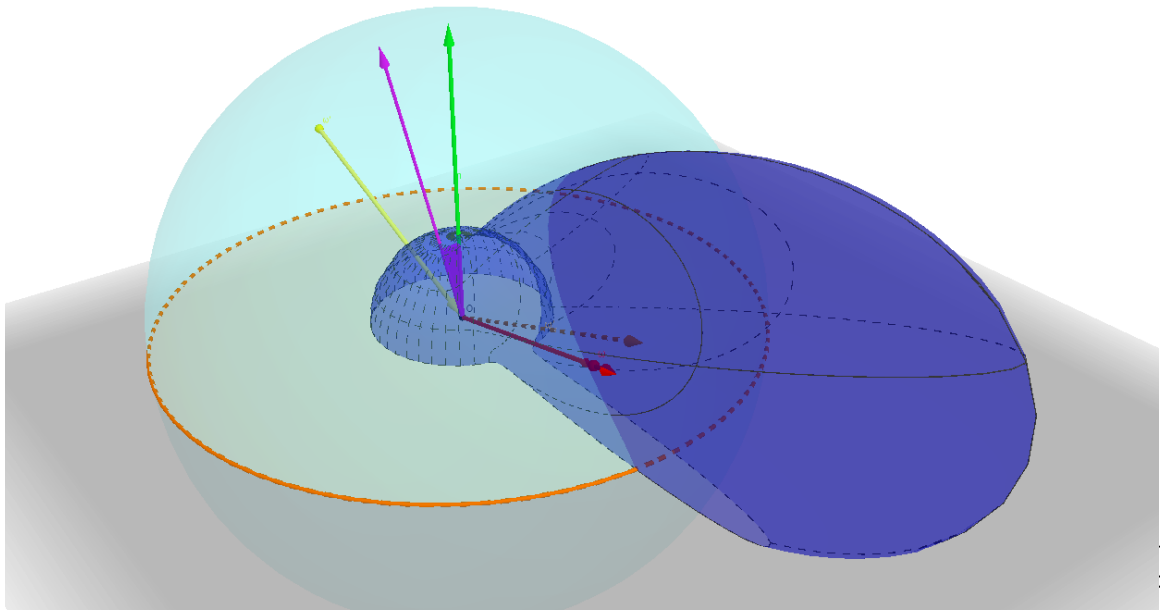
$$f_{r_t, \lambda}(\mathbf{l}, \mathbf{v}) = k_r(\lambda) \cdot \langle \mathbf{n}, \mathbf{h} \rangle^N$$

- ▶ Where $\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|_2}$
- ▶ Very fast
- ▶ Symmetric



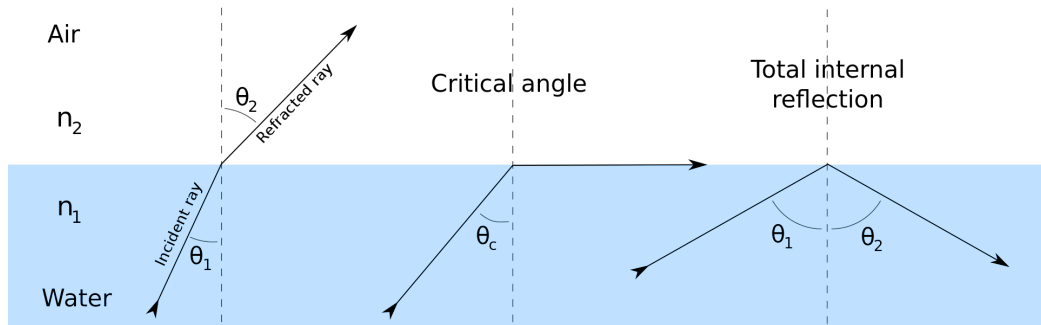
Phong vs Blinn-Phong





Refraction – Snell–Descartes law

plane of incidence: plane of incoming ray, outgoing ray, and normal



$$n_2 \sin(\theta_2) = n_1 \sin(\theta_1)$$



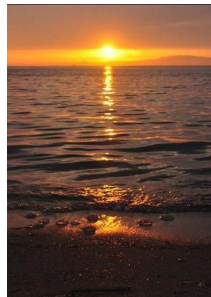
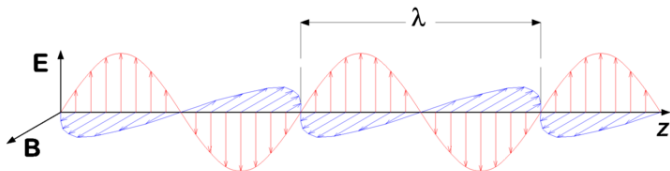
IOR: Index of Reflection

$$n_1 = \frac{\sqrt{\frac{i \frac{2\pi}{\lambda} \mu}{\sigma + i \frac{2\pi}{\lambda} \epsilon}}}{Z_0}$$

- ▶ Speed of the light relative to vacuum
- ▶ μ magnetic permeability: how hard is it to magnetize the material
- ▶ ϵ electric permeability: how hard is it to charge it with electrons
- ▶ $\sigma = \frac{1}{\rho}$ electric conductivity is the reciprocal of electrical resistivity: large for conductors
- ▶ Metals produce complex numbers. The nominator is the wave impedance.
- ▶ $Z_0 \approx 120\pi \Omega$ is the wave impedance of the vacuum
- ▶ λ is the wavelength of the light



Fresnel reflection



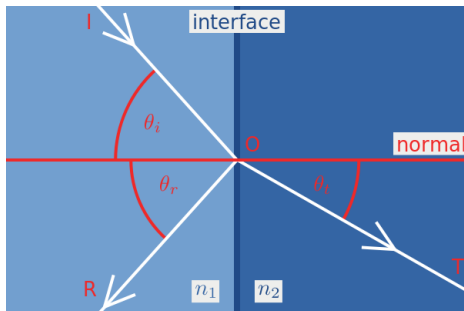
- ▶ **s-polarized**: wave in the electric field is parallel with the **plane of incidence**
- ▶ **p-polarized** is the perpendicular component
- ▶ At high incidence angles, the s-polarized light is reflected more
- ▶ Polarized sunglasses reduce glare



Fresnel equations

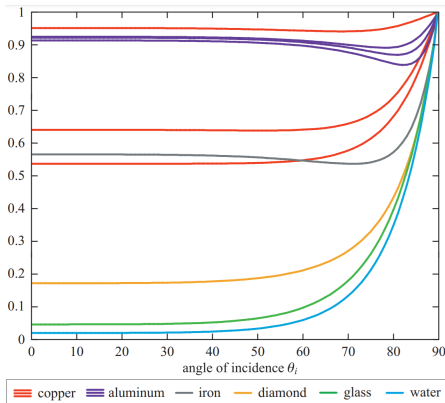
$$R_s = \left| \frac{n_2 \cos \theta_i - n_1 \cos \theta_t}{n_2 \cos \theta_i + n_1 \cos \theta_t} \right| \quad R_p = \left| \frac{n_2 \cos \theta_t - n_1 \cos \theta_i}{n_2 \cos \theta_t + n_1 \cos \theta_i} \right|$$

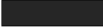









- ▶ Each polarized component of the light is reduced according R_s and R_p factors.
- ▶ Conservation of energy means $R_s^2 + R_p^2 \leq 1$.



Metals and Non-metals

- ▶ F_0 is the reflected light at 0 incident angle.
- ▶ Non-metals typically range between 2%-5% gray.
- ▶ Metals reflect 50%-100% of light in various colours.

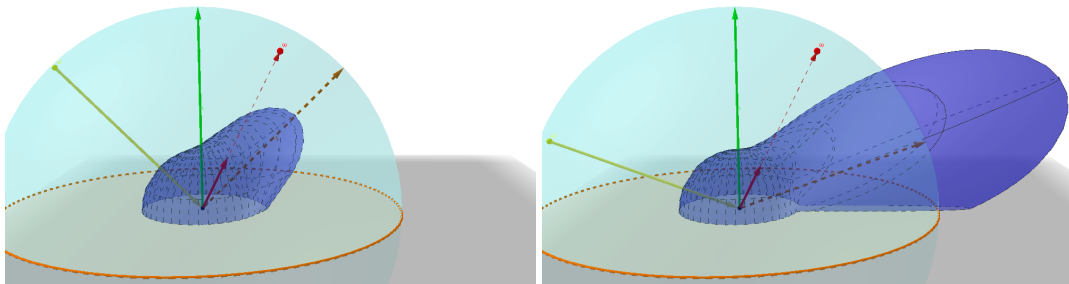


Material	F_0 (Linear)	F_0 (sRGB)	Color
Water	0.02,0.02,0.02	0.15,0.15,0.15	
Plastic / Glass (Low)	0.03,0.03,0.03	0.21,0.21,0.21	
Plastic High	0.05,0.05,0.05	0.24,0.24,0.24	
Glass (High) / Ruby	0.08,0.08,0.08	0.31,0.31,0.31	
Diamond	0.17,0.17,0.17	0.45,0.45,0.45	
Iron	0.56,0.57,0.58	0.77,0.78,0.78	
Copper	0.95,0.64,0.54	0.98,0.82,0.76	
Gold	1.00,0.71,0.29	1.00,0.86,0.57	
Aluminum	0.91,0.92,0.92	0.96,0.96,0.97	
Silver	0.95,0.93,0.88	0.98,0.97,0.95	



Schlick's approximation

- ▶ For natural unpolarized light $R_0 \approx \sqrt{\frac{R_s^2 + R_p^2}{2}}$
- ▶ Incorporate it into Blinn-Phong as a multiplier for specular highlight:

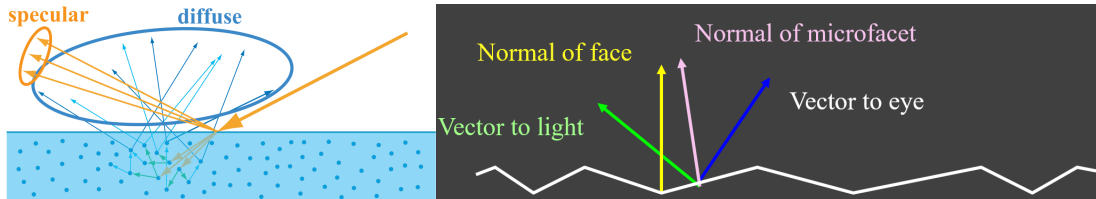


- ▶ Schlick simplified and approximated this with
- ▶ $F_0 := \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$ $F := F_0 + (1 - F_0)(1 - \langle \mathbf{n}, \omega' \rangle)^5$

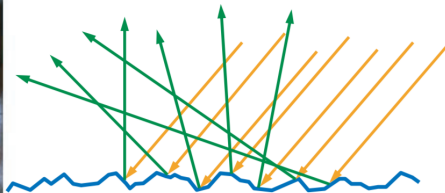
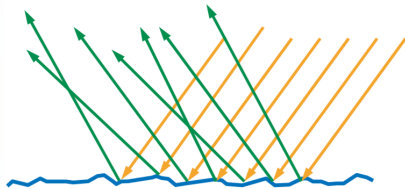


Cook-Torrance model 1983

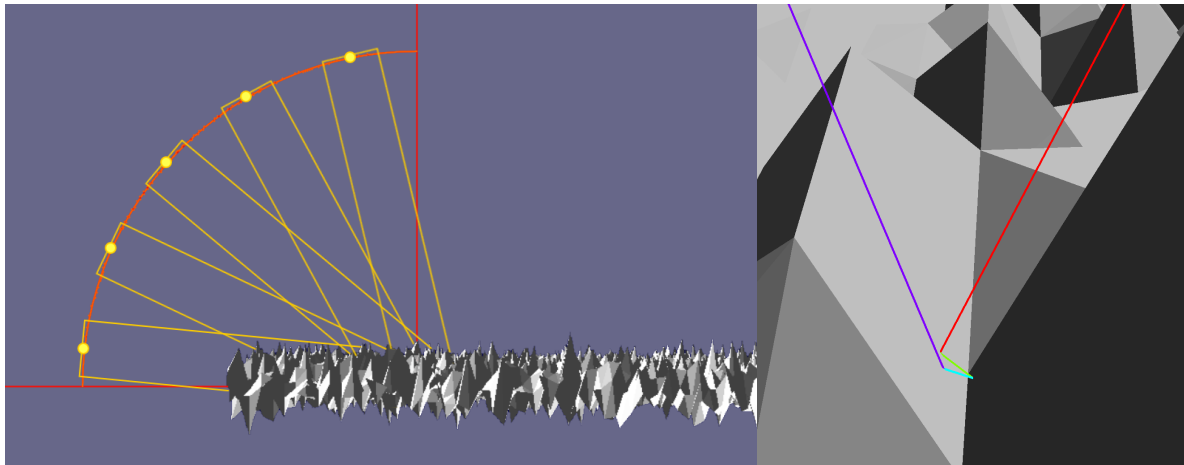
- ▶ Physically based: the surface is modelled as perfectly reflecting microfacets
- ▶ Distribution of microfacets is given (roughness)
- ▶ Calculates masking and self-shadowing effects between microfacets
- ▶ Inputs are physical properties of the material, can be measured (unlike the specular power)



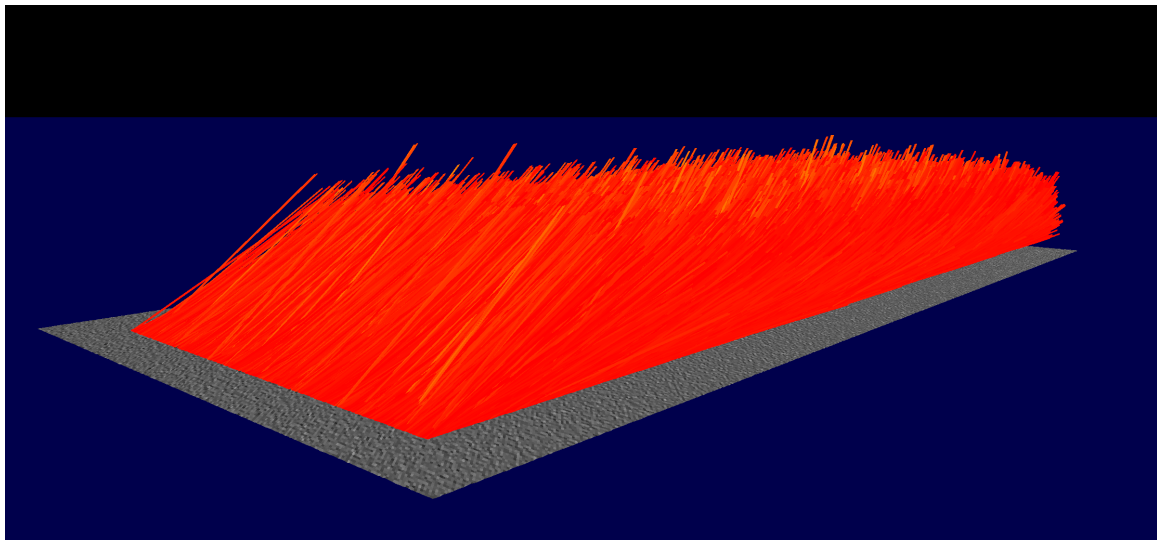
Microfacet theory



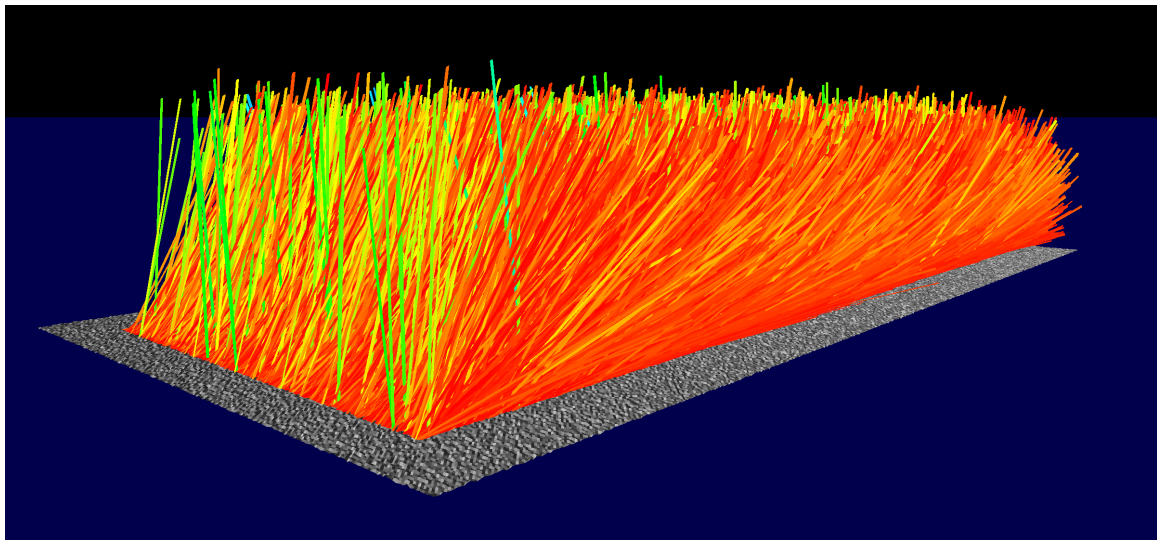
Demo: generated random microfacet surface with ray intersections



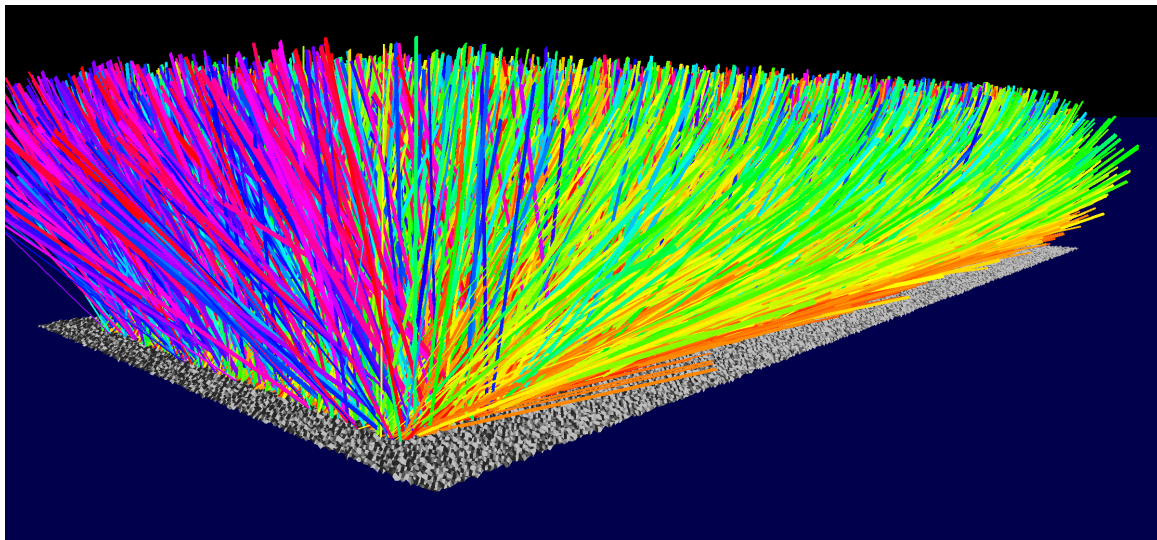
Demo: reflected rays from surface with 0.05 normal variance



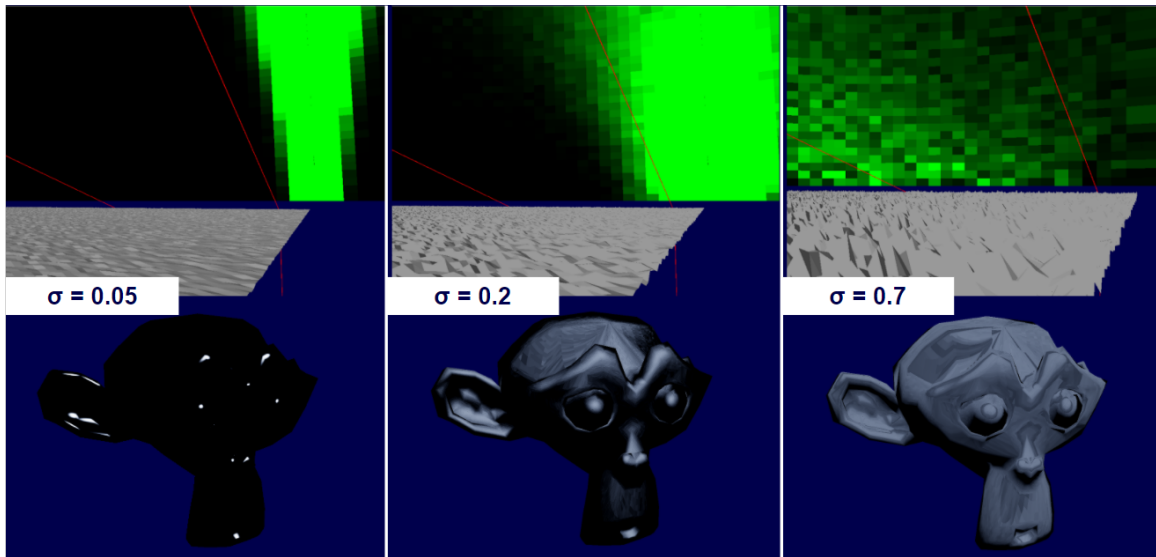
Demo: reflected rays from surface with 0.15 normal variance

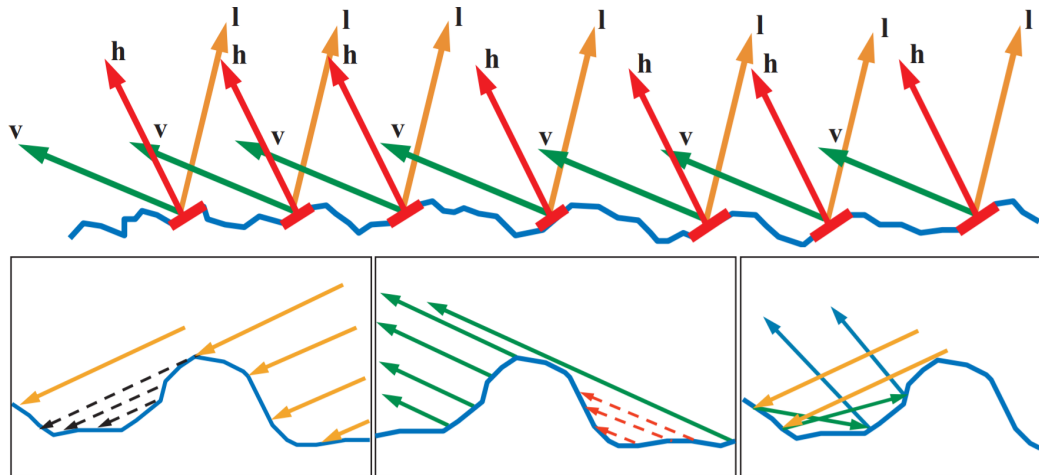


Demo: reflected rays from surface with 0.40 normal variance



Demo: resulting BRDF



Bisector angle h 

Roughness models

Microgeometry density from a given angle is modelled with a distribution.

$D_m(\mathbf{h})$ "counts" the microfacets that would reflect \mathbf{l} to \mathbf{v} .

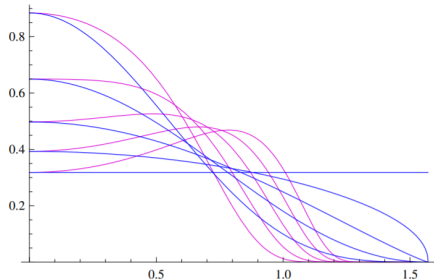
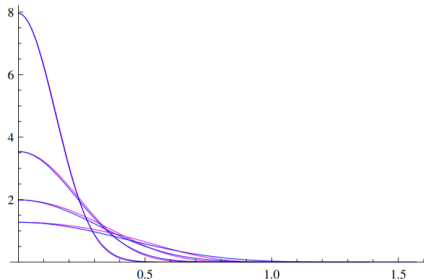
1. Blinn: Gaussian distribution
$$D_m(\mathbf{h}) = \frac{1}{\pi m^2} \cdot \langle \mathbf{h}, \mathbf{n} \rangle^{\frac{2}{m^2} - 2}$$

2. Backman distribution:
$$D_m(\mathbf{h}) = \frac{1}{\pi m^2 \langle \mathbf{h}, \mathbf{n} \rangle^4} \cdot e^{\frac{\langle \mathbf{h}, \mathbf{n} \rangle^2 - 1}{m^2 \langle \mathbf{h}, \mathbf{n} \rangle^2}}$$

- ▶ \mathbf{h} is the viewing angle for which $D_m(\mathbf{h})$ returns the relative microfacet density.
- ▶ $D_m(\mathbf{h})$ is normally evaluated at $\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|_2}$ angular bisector
- ▶ The roughness parameter m is the root mean square – quadratic mean of the slope of the microfacets



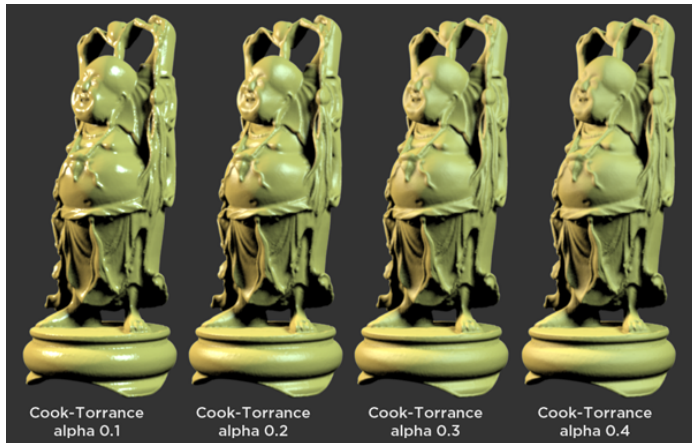
Blinn Phong vs Backman



Blinn-Phong distribution (blue) and Backman distribution (purple) for roughness values of 0.2-0.5 (right) and 0.6-1.0 (left).

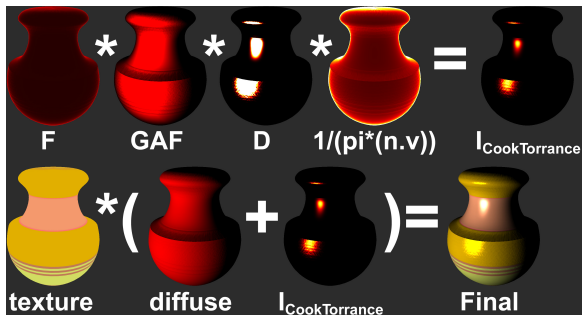
Properties of D_m and the result of varying roughness

$$\forall v \in \Omega, \quad \langle v, n \rangle = \int_{\Omega} D_m(h) \langle v, h \rangle dh \quad \xRightarrow{v:=n} \quad 1 = \int_{\Omega} D_m(h) \langle n, h \rangle dh$$



Cook-Torrance model

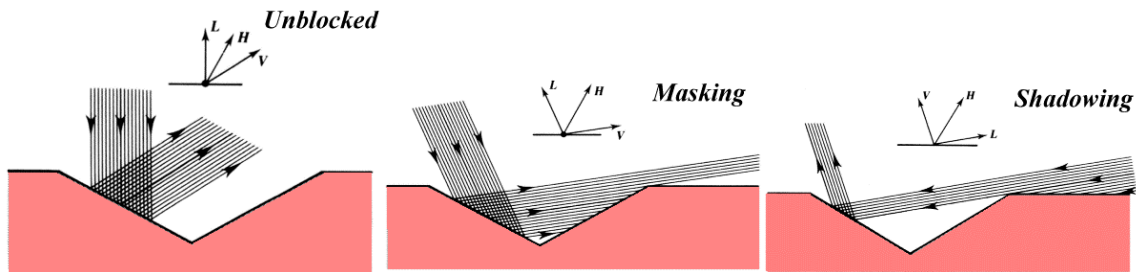
$$f_{rt,\lambda}(l, v) = \frac{F_\lambda(\mathbf{h}) \cdot G(\mathbf{n}, \mathbf{v}, \mathbf{l}) \cdot D_m(\mathbf{h})}{\langle \mathbf{n}, \mathbf{v} \rangle}$$



- ▶ where $\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|_2}$
- ▶ $D_m(\mathbf{h})$ is the microfacet density towards \mathbf{h}' . $F_\lambda(\mathbf{h})$ is the Fresnel reflectance ratio.
- ▶ $G(\mathbf{n}, \mathbf{v}, \mathbf{l})$ **Geometric attenuation factor**: self shadowing and masking



Geometric Attenuation Factor

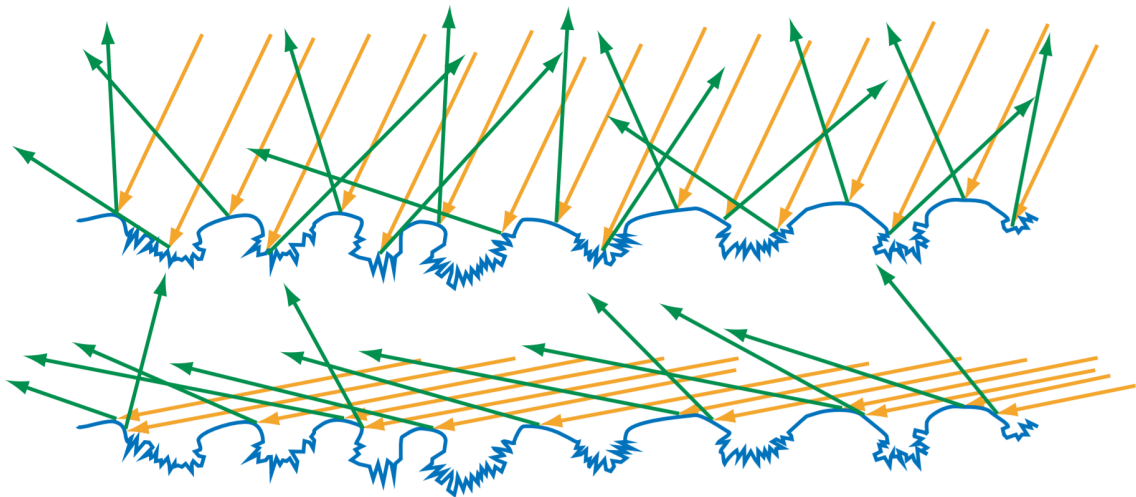


$$G_{unblocked} = 1, \quad G_{mask} = \frac{2\langle \mathbf{h}, \mathbf{n} \rangle \langle \mathbf{v}, \mathbf{n} \rangle}{\langle \mathbf{v}, \mathbf{h} \rangle}, \quad G_{shadow} = \frac{2\langle \mathbf{h}, \mathbf{n} \rangle \langle \mathbf{l}, \mathbf{n} \rangle}{\langle \mathbf{v}, \mathbf{h} \rangle}$$

$$G = 1 - \frac{\text{blocked}}{\text{facet}} = \min\{G_{unblocked}, G_{mask}, G_{shadow}\}$$



Limitations



Code for Fresnel

$$R_s = \left| \frac{n_2 \cos \theta_i - n_1 \cos \theta_t}{n_2 \cos \theta_i + n_1 \cos \theta_t} \right| \quad R_p = \left| \frac{n_2 \cos \theta_t - n_1 \cos \theta_i}{n_2 \cos \theta_t + n_1 \cos \theta_i} \right|$$

```
float Fresnel(vec3 v, vec3 n)
{
    vec3 t = normalize(refract(v, n, n1/n2));
    float ci = max(dot(v,n),0.), ct= max(-dot(t,n),0.);
    float Rs = (n1*ci-n2*ct)/(n1*ci+n2*ct); Rs *= Rs;
    float Rp = (n1*ct-n2*ci)/(n1*ct+n2*ci); Rp *= Rp;
    return 0.45*0.5*(Rp+Rs)+0.55*sqrt(0.5*(Rs*Rs+Rp*Rp)); //approx
}

vec3 radiance
(
    vec3 n, // normal
    vec3 l, // direction from x point to light
    vec3 v // direction from x point to view
)
```

...



Code for Radiance

```

{
float m = sin(iTime)*0.5+0.5; //between 0 and 1
vec3 h = normalize( l + v ); // half vector
float hn = max(dot(h, n), 0.0), vn = max(dot(v, n), 0.0);
float ln = max(dot(l, n), 0.0), vh = max(dot(v, h), 0.0);

// Geometric Attenuation Factor
float G = min( min( 2.*hn*vn/vh, 2.*hn*ln/vh ), 1.0 );

// Microfacet density, normal distribution
float hn2 = hn*hn, m2= m*m; //tan(acos(hn)) optimized away
float D = exp((hn2-1.)/(hn2*m2)) / (m2*hn2*hn2*pi);

float R_spec = Fresnel(l, n); //Fresnel reflectance ratio
//float R_spec = R0_spec + (1.-R0_spec)*pow(1.-ln,5.); //Schlick approx

// Cook-Torrance BRDF
vec3 brdf_spec = vec3(R_spec * D * G / (vn*ln));

// Lambertian BRDF
vec3 brdf_diff = k_d * ( 1.0 - R_spec );

// Punctual Light Source
return ( brdf_spec + brdf_diff ) * light_int * ln;
}

```

$$D(m, \alpha) = \frac{e^{-\frac{\tan^2 \alpha}{m^2}}}{\pi m^2 \cos^4 \alpha}$$



Radiosity

Csaba **Bálint**
first name family name

csabix@inf.elte.hu

Eötvös Loránd University,
Faculty of Informatics

Computer Graphics Lecture
Budapest 2022

- ▼ Rendering Equation
- ▼ Towards Radiosity
 - Integral transformation
 - Geometric term
 - Form factor
- ▼ Radiosity equation
 - Most important assumption
 - Discrete form
 - Numerical Performance



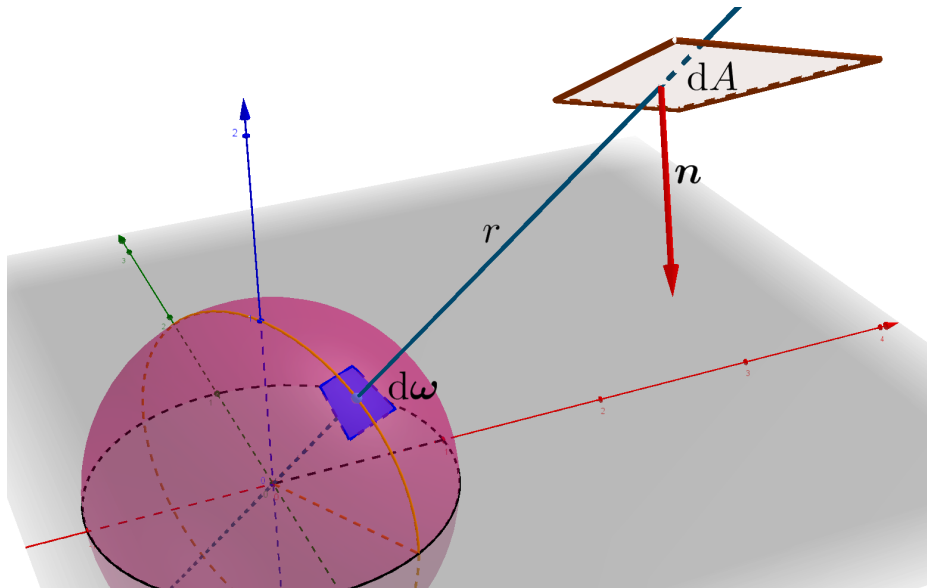
Rendering Equation

$$L_{t,\lambda}(\mathbf{x}, \omega) = L_{t,\lambda}^{em}(\mathbf{x}, \omega) + \int_{\mathbb{S}} L_{t,\lambda}(V(\mathbf{x}, -\omega'), \omega') \cdot f_{r_{t,\lambda}}(\omega', \mathbf{x}, \omega) \cdot \cos \theta' d\omega'$$

- ▶ $L_{t,\lambda}(\mathbf{x}, \omega)$ is the radiance from a point on a surface \mathbf{x} in a given direction ω , at time t in wavelength λ .
- ▶ $L_{t,\lambda}^{em}(\mathbf{x}, \omega)$ is the emitted radiance from \mathbf{x} to ω .
- ▶ $\int_{\mathbb{S}} L_{t,\lambda}(V(\mathbf{x}, -\omega'), \omega') \cdot f_{r_{t,\lambda}}(\omega', \mathbf{x}, \omega) \cdot \cos \theta' d\omega'$ Radiance contribution from all of the other surfaces in the scene
- ▶ $L_{t,\lambda}(V(\mathbf{x}, -\omega'), \omega')$ incoming radiance – recursion
- ▶ $V(\mathbf{x}, -\omega') =: \mathbf{x}'$ closest object in direction $-\omega'$
- ▶ $f_{r_{t,\lambda}}(\omega', \mathbf{x}, \omega)$ Bidirectional Reflectance Distribution Function, $\omega' = (\varphi', \theta')$.



Remember: visibility of a differential surface



Using the visibility function

Given that: (without proof)

Let $F : I_1 \times I_2 \rightarrow \mathbb{R}^3$ be a regular surface with normal $\mathbf{m}_F(\mathbf{w})$ such that the

$$V^{-1}(u, v) := [T^{-1}(F(u, v))]_{1,2} \in \Omega \quad (u \in I_1, v \in I_2)$$

function is bijective. Let $f : I_1 \times I_2 \rightarrow \mathbb{R}$ be a continuous function. Then

$$\int_{\Omega} f(V(\boldsymbol{\omega})) \, d\boldsymbol{\omega} = \int_{I_1 \times I_2} f(\mathbf{w}) \frac{\langle \mathbf{m}_F(\mathbf{w}), F(\mathbf{w}) \rangle}{\|F(\mathbf{w})\|^3} \, d\mathbf{w}$$



Using the visibility function

Given that: (without proof)

Let $F : I_1 \times I_2 \rightarrow \mathbb{R}^3$ be a regular surface with normal $\mathbf{m}_F(\mathbf{w})$ such that the

$$V^{-1}(u, v) := [T^{-1}(F(u, v))]_{1,2} \in \Omega \quad (u \in I_1, v \in I_2)$$

function is bijective. Let $f : I_1 \times I_2 \rightarrow \mathbb{R}$ be a continuous function. Then

$$\int_{\Omega} f(V(\boldsymbol{\omega})) \, d\boldsymbol{\omega} = \int_{I_1 \times I_2} f(\mathbf{w}) \frac{\langle \mathbf{m}_F(\mathbf{w}), F(\mathbf{w}) \rangle}{\|F(\mathbf{w})\|^3} \, d\mathbf{w}$$

In the rendering equation we have

$$\int_{\mathbb{S}} L_{t,\lambda}(V(\mathbf{x}, -\boldsymbol{\omega}'), \boldsymbol{\omega}') \cdot f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot \cos \theta' \, d\boldsymbol{\omega}' =$$



Transforming the integral

$$\int_{\mathbb{S}} L_{t,\lambda}(V(\mathbf{x}, -\boldsymbol{\omega}'), \boldsymbol{\omega}') \cdot f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot \cos \theta' \, d\boldsymbol{\omega}' =$$

Integrate on the visible surfaces instead of the sphere using the transformation $\mathbf{x}' = V(\mathbf{x}, -\boldsymbol{\omega}')$, $\boldsymbol{\omega}' = [T^{-1}(\mathbf{x}' - \mathbf{x})]_{1,2}$.

$$= \int_{V(\mathbf{x}, \mathbb{S})} L_{t,\lambda}(\mathbf{x}', \boldsymbol{\omega}') \cdot f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot \frac{\langle \mathbf{n}, \mathbf{x}' - \mathbf{x} \rangle \langle \mathbf{n}', \mathbf{x} - \mathbf{x}' \rangle}{\|\mathbf{x}' - \mathbf{x}\|^4} \, d\mathbf{x}' =$$

Notice that $V(\mathbf{x}, \mathbb{S}) \subseteq X$, where $X \subseteq \mathbb{R}^3$ the set of all surfaces.

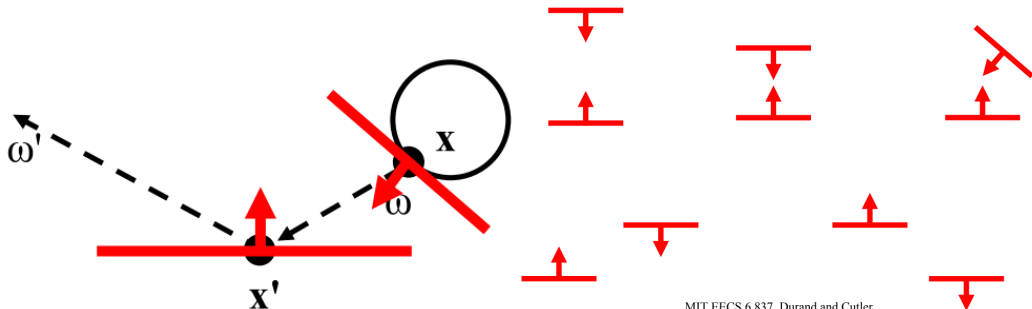
$$= \int_X f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot L_{t,\lambda}(\mathbf{x}', \boldsymbol{\omega}') \cdot \chi_{V(\mathbf{x}, \mathbb{S})}(\mathbf{x}') \cdot \frac{\langle \mathbf{n}, \mathbf{x}' - \mathbf{x} \rangle \langle \mathbf{n}', \mathbf{x} - \mathbf{x}' \rangle}{\|\mathbf{x}' - \mathbf{x}\|^4} \, d\mathbf{x}'$$



Geometric term and visibility function

$$= \int_X f_{r,t,\lambda}(\omega', \mathbf{x}, \omega) \cdot L_{t,\lambda}(\mathbf{x}', \omega') \cdot \mathcal{X}_{V(\mathbf{x}, \mathbb{S})}(\mathbf{x}') \cdot \frac{\langle \mathbf{n}, \mathbf{x}' - \mathbf{x} \rangle \langle \mathbf{n}', \mathbf{x} - \mathbf{x}' \rangle}{\|\mathbf{x}' - \mathbf{x}\|^4} d\mathbf{x}' =$$

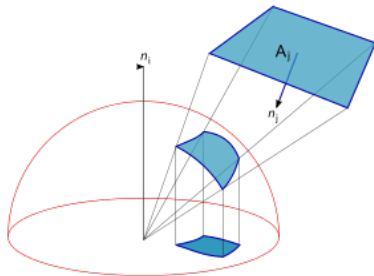
- ▶ $(\mathbf{x}, \mathbf{x}') \rightarrow \mathcal{X}_{V(\mathbf{x}, \mathbb{S})}(\mathbf{x}')$ is the **visibility function** between two points.
- ▶ $G(\mathbf{x}, \mathbf{x}') := \frac{\langle \mathbf{n}, \mathbf{x}' - \mathbf{x} \rangle \langle \mathbf{n}', \mathbf{x} - \mathbf{x}' \rangle}{\|\mathbf{x}' - \mathbf{x}\|^4}$ describes the geometric relationship between \mathbf{x} and \mathbf{x}' .
 - ▶ Symmetric function (fundamental law of photometry).



Form factor (view factor)

$$= \int_X f_{r_t, \lambda}(\omega', \mathbf{x}, \omega) \cdot L_{t, \lambda}(\mathbf{x}, \mathbf{x}') \cdot \underbrace{\mathcal{X}_{V(\mathbf{x}, \mathbb{S})}(\mathbf{x}') \cdot G(\mathbf{x}, \mathbf{x}')}_{F(\mathbf{x}, \mathbf{x}')} d\mathbf{x}'$$

- ▶ The geometric term multiplied by visibility is the **form factor** $F(\mathbf{x}, \mathbf{x}')$.
- ▶ Also a symmetric function.
- ▶ Only depends on \mathbf{x}, \mathbf{x}' , and not on ω . (Note that ω' is the direction of $\mathbf{x}' - \mathbf{x}$.)



Radiosity equation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_X f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot L_{t,\lambda}(\mathbf{x}, \mathbf{x}') \cdot F(\mathbf{x}, \mathbf{x}') d\mathbf{x}'$$

What do we want?



Radiosity equation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_X f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot L_{t,\lambda}(\mathbf{x}, \mathbf{x}') \cdot F(\mathbf{x}, \mathbf{x}') d\mathbf{x}'$$

What do we want? Precalculate the integral for all $x \in X$.

What's the problem?



Radiosity equation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_X f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot L_{t,\lambda}(\mathbf{x}, \mathbf{x}') \cdot F(\mathbf{x}, \mathbf{x}') d\mathbf{x}'$$

What do we want? Precalculate the integral for all $x \in X$.

What's the problem? Integral depends on $\boldsymbol{\omega} \in \mathbb{S}$ viewing angle.

What do we do?



Radiosity equation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_X f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot L_{t,\lambda}(\mathbf{x}, \mathbf{x}') \cdot F(\mathbf{x}, \mathbf{x}') d\mathbf{x}'$$

What do we want? Precalculate the integral for all $x \in X$.

What's the problem? Integral depends on $\boldsymbol{\omega} \in \mathbb{S}$ viewing angle.

What do we do? Assume it does not.

What does that mean?



Radiosity equation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_X f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cdot L_{t,\lambda}(\mathbf{x}, \mathbf{x}') \cdot F(\mathbf{x}, \mathbf{x}') d\mathbf{x}'$$

What do we want? Precalculate the integral for all $x \in X$.

What's the problem? Integral depends on $\boldsymbol{\omega} \in \mathbb{S}$ viewing angle.

What do we do? Assume it does not.

What does that mean? Diffuse BRDF

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + k_d(\mathbf{x}) \cdot \int_X L_{t,\lambda}(\mathbf{x}, \mathbf{x}') \cdot F(\mathbf{x}, \mathbf{x}') d\mathbf{x}'$$



Discretisation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + k_d(\mathbf{x}) \cdot \int_X L_{t,\lambda}(\mathbf{x}, \mathbf{x}') \cdot F(\mathbf{x}, \mathbf{x}') d\mathbf{x}'$$

Discretize X into n patches over which the radiances are constant: $X = \bigcup_{i=1}^n X_i$.

- ▶ $F(\mathbf{x}, \mathbf{x}') = F_{ij} \in [0, +\infty)$ ($\mathbf{x} \in X_i$) discretized form factors
- ▶ $L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) = E_i \in [0, +\infty)$ ($\mathbf{x} \in X_i$) emitted radiance
- ▶ $k_d(\mathbf{x}) = \rho_i \in [0, \frac{1}{\pi})$ ($\mathbf{x} \in X_i$) is the reflectivity of the patch
- ▶ $L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_i \in [0, +\infty)$ ($\mathbf{x} \in X_i$) this is the **unknown radiance**

This leads to the discrete radiosity equation

$$L_i = E_i + \rho_i \sum_{j=1}^n F_{i,j} \cdot L_j .$$



Discrete Radiosity Equation

$$L_i = E_i + \rho \sum_{j=1}^n F_{i,j} \cdot L_j =$$

In matrix form:

$$(I - \rho \cdot F) \cdot L = E$$

$$\begin{bmatrix} 1 - \rho_1 \cdot F_{11} & -\rho_1 \cdot F_{12} & \dots & -\rho_1 \cdot F_{1n} \\ -\rho_2 \cdot F_{21} & 1 - \rho_2 \cdot F_{22} & \dots & -\rho_2 \cdot F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n \cdot F_{n1} & -\rho_n \cdot F_{n2} & \dots & 1 - \rho_n \cdot F_{nn} \end{bmatrix} \cdot \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

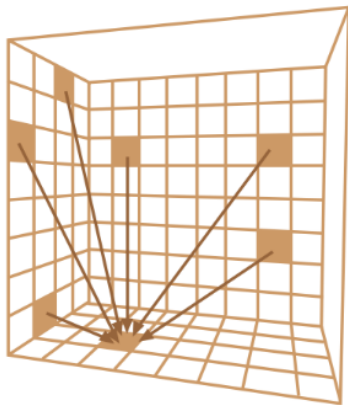
What can we see in this matrix?



Jacobi method

Since the diagonal part was I , the Jacobi iteration becomes

$$L^{(k+1)} = E + \rho_i \cdot F \cdot L^{(k)}$$



But does it converge?



Convergence condition

If $I - \rho F$ is strictly diagonally dominant, so when

$$|1 - \rho_i F_{ii}| > \sum_{j=1, i \neq j}^n |\rho_i F_{ij}| \iff 1 > \rho_i \cdot \sum_{j=1}^n F_{ij}$$

Using that all values are positive. Moreover,

- ▶ $\sum_{j=1}^n F_{ij} \leq 1$ because the ratio of visible surfaces cannot exceed 1 together.
- ▶ $\rho_i \leq \frac{1}{\pi}$ due to energy conservation discussed earlier.

Hence, the spectral radius is $\max_i |\lambda_i| \leq \frac{1}{\pi}$, so we can bound the convergence error with

$$\|L^{(k)} - L^*\|_{\infty} \leq 1.47 \cdot 0.32^k \cdot \|L^{(1)} - L^{(0)}\|_{\infty} \quad (k \in \mathbb{N})$$



Global Illumination

Csaba Bálint

first name family name

csabix@inf.elte.hu

Eötvös Loránd University,
Faculty of Informatics

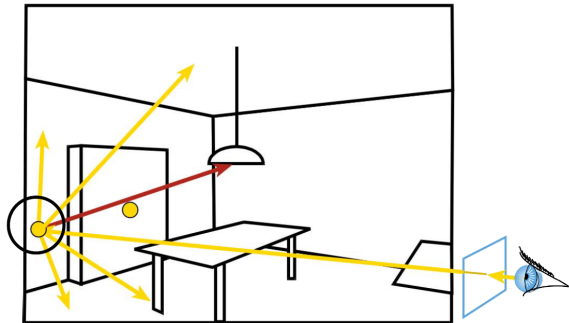
Computer Graphics Lecture
Budapest 2022

- ▼ Motivation
- ▼ Recursive ray tracing
 - Monte Carlo
 - Ray tracing
- ▼ Path tracing
 - Rendering equation
 - Path tracing
- ▼ Further methods
 - Irradiance caching
 - Photon mapping
 - Subsurface scattering

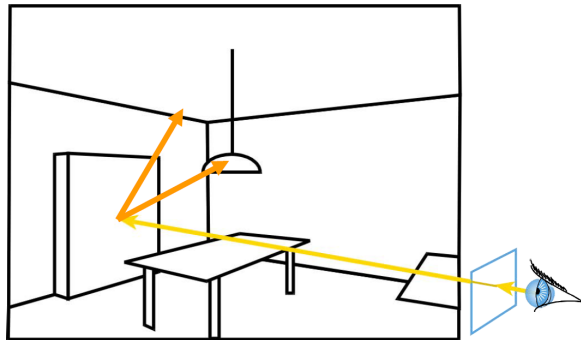
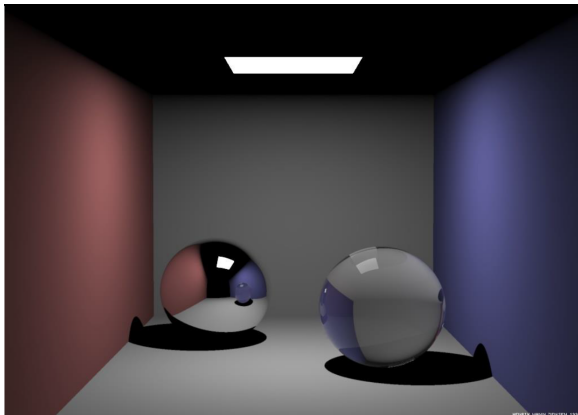


Motivation: Raycast

- ▶ Simulate light rays – can only simulate finite amount
- ▶ How to choose from a continuum set of rays?
- ▶ How to increase performance and quality?
- ▶ E-g.: choose the red ray: direct lighting
- ▶ Need all other directions too



Motivation: Direct Illumination

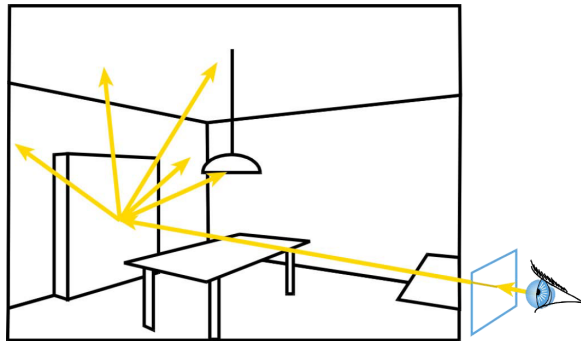
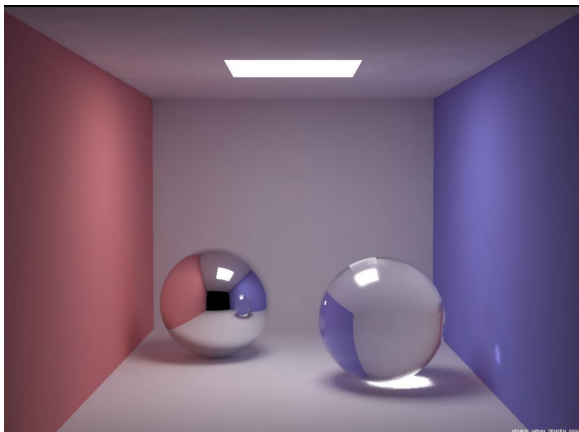


Monte Carlo ray tracing

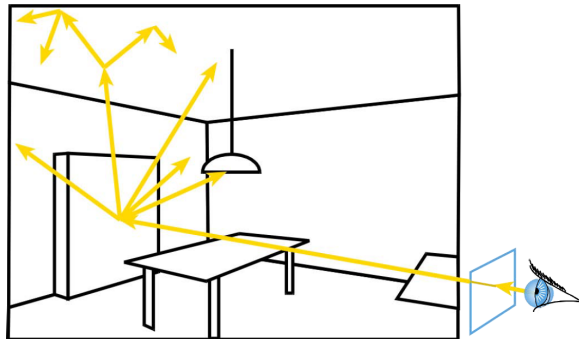
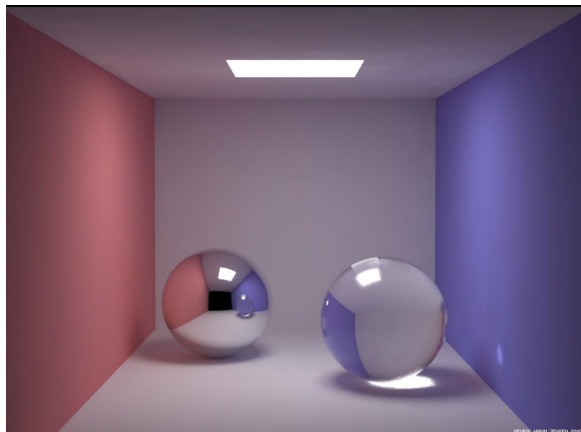
[See supplemental material](#)



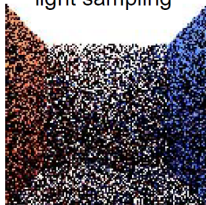
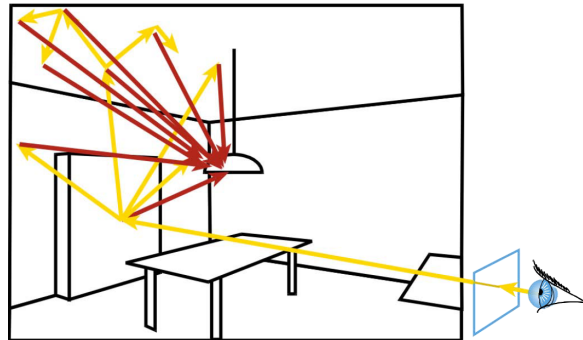
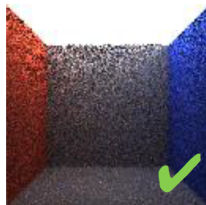
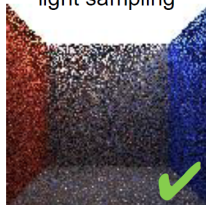
Random sample directions - one bounce



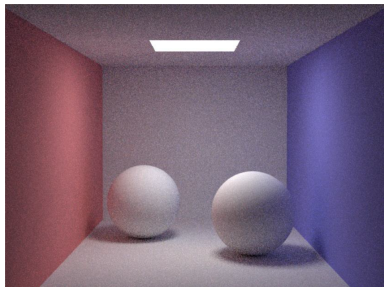
Ray trace recursively



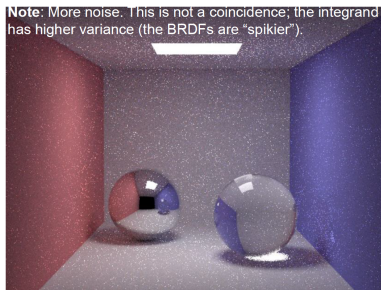
Direct light sampling

Without explicit
light sampling1 path
per pixel4 paths
per pixelWith explicit
light sampling

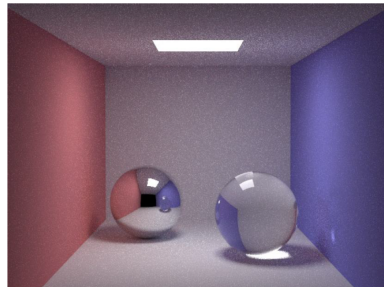
BRDF variance impact on performance



Diffuse 10 path per pixel



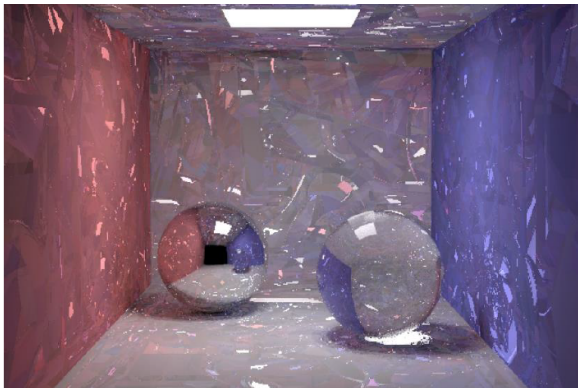
Glossy 10 path per pixel



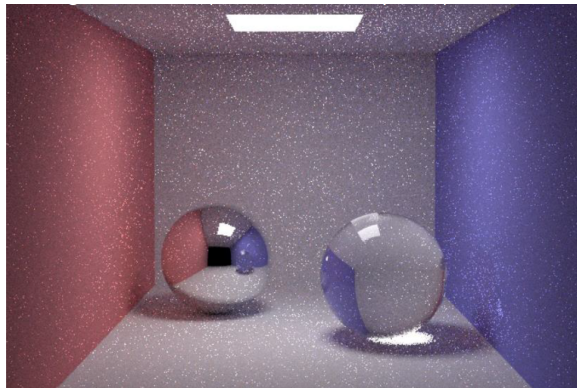
Glossy 100 path per pixel

Why random?

- ▶ What happens if we use a fixed random sequence?
- ▶ Structured error is worse than random noise.



10 samples from fixed list



10 random samples



Rendering equation

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) + \int_{\mathcal{K}} L_{t,\lambda}(V(\mathbf{x}, -\boldsymbol{\omega}'), \boldsymbol{\omega}') f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cos \theta' d\boldsymbol{\omega}'$$

Let $\mathbf{x}' = V(\mathbf{x}, -\boldsymbol{\omega}')$ and $f_{\mathbf{x}}(\boldsymbol{\omega}', \boldsymbol{\omega}) = f_{r_{t,\lambda}}(\boldsymbol{\omega}', \mathbf{x}, \boldsymbol{\omega}) \cos \theta'$ for brevity.

$$L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega}) = L_{t,\lambda}^{em}(\mathbf{x}, \boldsymbol{\omega}) \int_{\mathcal{K}} L_{t,\lambda}(\mathbf{x}', \boldsymbol{\omega}') f_{\mathbf{x}}(\boldsymbol{\omega}', \boldsymbol{\omega}) d\boldsymbol{\omega}'$$

Also, let $L_{\mathbf{x}}(\boldsymbol{\omega}) = L_{t,\lambda}(\mathbf{x}, \boldsymbol{\omega})$.

$$L_{\mathbf{x}}(\boldsymbol{\omega}) = L_{\mathbf{x}}^{em}(\boldsymbol{\omega}) + \int_{\mathcal{K}} L_{\mathbf{x}'}(\boldsymbol{\omega}') f_{\mathbf{x}}(\boldsymbol{\omega}', \boldsymbol{\omega}) d\boldsymbol{\omega}'$$



Recursion

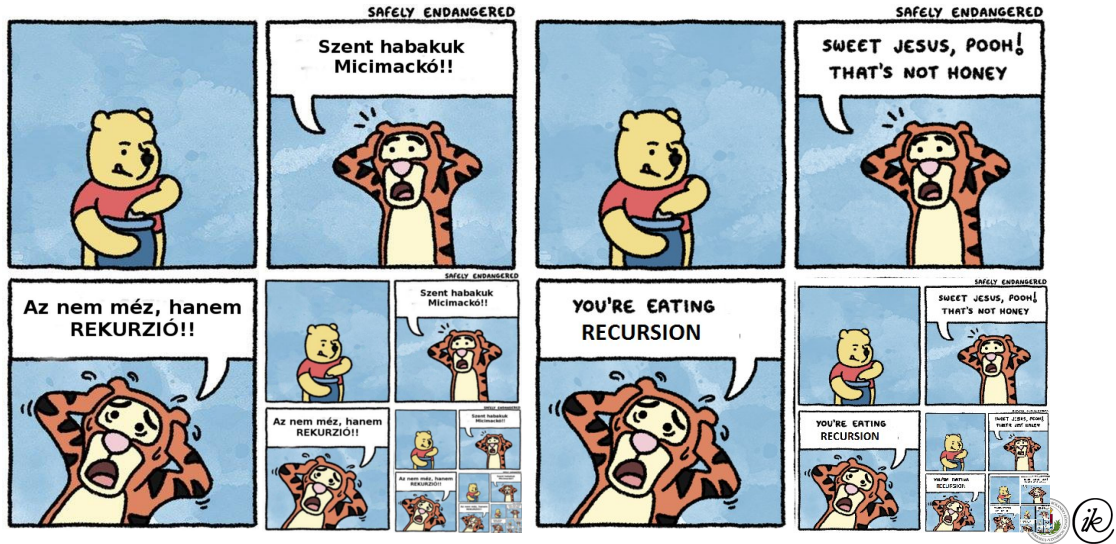
$$L_x(\omega) = L_x^{em}(\omega) + \int_{\mathcal{K}} L_{x'}(\omega') f_x(\omega', \omega) d\omega'$$

$$L_x(\omega) = L_x^{em}(\omega) + \int_{\mathcal{K}} \left(L_{x'}^{em}(\omega') + \int_{\mathcal{K}} L_{x''}(\omega'') f_{x'}(\omega'', \omega') d\omega'' \right) f_x(\omega', \omega) d\omega'$$

$$L_x(\omega) = L_x^{em}(\omega) + \int_{\mathcal{K}} \left(L_{x'}^{em}(\omega') + \int_{\mathcal{K}} \left(L_{x''}^{em}(\omega'') + \int_{\mathcal{K}} L_{x'''}(\omega''') f_{x''}(\omega''', \omega'') d\omega''' \right) f_{x'}(\omega'', \omega') d\omega'' \right) f_x(\omega', \omega) d\omega'$$



Micimackó



$$L_{\mathbf{x}}(\omega) = L_{\mathbf{x}}^{em}(\omega) + \int_{\mathcal{K}} L_{\mathbf{x}'}(\omega') f_{\mathbf{x}}(\omega', \omega) d\omega'$$

$$L_{\mathbf{x}}(\omega) = L_{\mathbf{x}}^{em}(\omega) + \int_0^{2\pi} \int_0^{\pi} L_{\mathbf{x}'}(\omega') f_{\mathbf{x}}(\omega', \omega) \sin \theta' d\theta' d\varphi'$$

$$L_{\mathbf{x}}(\omega) = \int_0^{2\pi} \left(\frac{L_{\mathbf{x}}^{em}(\omega)}{2\pi} + \int_0^{\pi} L_{\mathbf{x}'}(\omega') f_{\mathbf{x}}(\omega', \omega) \sin \theta' d\theta' \right) d\varphi'$$

$$L_{\mathbf{x}}(\omega) = \int_0^{2\pi} \int_0^{\pi} \left(\frac{L_{\mathbf{x}}^{em}(\omega)}{2\pi^2 \sin \theta'} + L_{\mathbf{x}'}(\omega') f_{\mathbf{x}}(\omega', \omega) \right) \sin \theta' d\theta' d\varphi'$$

$$L_{\mathbf{x}}(\omega) = \int_{\mathcal{K}} \left(\frac{L_{\mathbf{x}}^{em}(\omega)}{2\pi^2 \sin \theta'} + L_{\mathbf{x}'}(\omega') f_{r_{t,\lambda}}(\omega', \mathbf{x}, \omega) \cos \theta' \right) d\omega'$$



Recursion

$$L_{\mathbf{x}}(\omega) = L_{\mathbf{x}}^{em}(\omega) + \int_{\mathcal{K}} L_{\mathbf{x}'}(\omega') f_{\mathbf{x}}(\omega', \omega) d\omega'$$

$$L_{\mathbf{x}}(\omega) = \int_{\mathcal{K}} \left(\frac{L_{\mathbf{x}}^{em}(\omega)}{2\pi^2 \sin \theta'} + L_{\mathbf{x}'}(\omega') f_{\mathbf{x}}(\omega', \omega) \right) d\omega'$$

$$L_{\mathbf{x}}(\omega) = \int_{\mathcal{K}} \left(\frac{L_{\mathbf{x}}^{em}(\omega)}{2\pi^2 \sin \theta'} + \int_{\mathcal{K}} \left(\frac{L_{\mathbf{x}'}^{em}(\omega')}{2\pi^2 \sin \theta''} + L_{\mathbf{x}''}(\omega'') f_{\mathbf{x}'}(\omega'', \omega') \right) d\omega'' f_{\mathbf{x}}(\omega', \omega) \right) d\omega'$$

$$L_{\mathbf{x}}(\omega) = \int_{\mathcal{K}} \int_{\mathcal{K}} \frac{L_{\mathbf{x}}^{em}(\omega)}{(2\pi^2)^2 \sin \theta' \sin \theta''} + \left(\frac{L_{\mathbf{x}'}^{em}(\omega')}{2\pi^2 \sin \theta''} + L_{\mathbf{x}''}(\omega'') f_{\mathbf{x}'}(\omega'', \omega') \right) f_{\mathbf{x}}(\omega', \omega) d\omega'' d\omega'$$



Path tracing

$$L_{\mathbf{x}}(\omega) = \int_{\mathcal{K}} \int_{\mathcal{K}} \frac{L_{\mathbf{x}}^{em}(\omega)}{(2\pi^2)^2 \sin \theta' \sin \theta''} + \left(\frac{L_{\mathbf{x}'}^{em}(\omega')}{2\pi^2 \sin \theta''} + L_{\mathbf{x}''}(\omega'') f_{\mathbf{x}'}(\omega'', \omega') \right) f_{\mathbf{x}}(\omega', \omega) d\omega'' d\omega'$$

$$L_{\mathbf{x}}(\omega) = \int_{\mathcal{K}} \dots \int_{\mathcal{K}} \sum_{i=1}^n \frac{L_{\mathbf{x}^{(i-1)}}^{em}(\omega^{(i-1)})}{(2\pi^2)^{n-i+1} \prod_{j=i}^n \sin \theta^{(j)}} \prod_{j=1}^{i-1} f_{\mathbf{x}^{(j)}}(\omega^{(j)}, \omega^{(j-1)}) \\ + L_{\mathbf{x}^{(n)}}(\omega^{(n)}) \prod_{i=1}^n f_{\mathbf{x}^{(i)}}(\omega^{(i)}, \omega^{(i-1)}) d\omega^{(n+1)} \dots d\omega'$$

$$L_{\mathbf{x}}(\omega) = \lim_{n \rightarrow \infty} \int_{\mathcal{K}} \dots \int_{\mathcal{K}} \sum_{i=1}^n \frac{L_{\mathbf{x}^{(i-1)}}^{em}(\omega^{(i-1)})}{(2\pi^2)^{n-i+1} \prod_{j=i}^n \sin \theta^{(j)}} \prod_{j=1}^{i-1} f_{\mathbf{x}^{(j)}}(\omega^{(j)}, \omega^{(j-1)}) d\omega^{(n+1)} \dots d\omega'$$



Path tracing

$$L_{\mathbf{x}}(\boldsymbol{\omega}) = \lim_{n \rightarrow \infty} \int_{\mathcal{K}} \dots \int_{\mathcal{K}} \sum_{i=1}^n \frac{L_{\mathbf{x}^{(i-1)}}^{em}(\boldsymbol{\omega}^{(i-1)})}{(2\pi^2)^{n-i+1} \prod_{j=i}^n \sin \theta^{(j)}} \prod_{j=1}^{i-1} f_{\mathbf{x}^{(j)}}(\boldsymbol{\omega}^{(j)}, \boldsymbol{\omega}^{(j-1)}) d\boldsymbol{\omega}^{(n+1)} \dots d\boldsymbol{\omega}'$$

Input: $\mathbf{x}_0 \in \mathbb{R}^3$

Input: $\boldsymbol{\omega}[0] = (\varphi[0], \theta[0])$

Output: $L := [0, 0, 0]$

$factor := [1, 1, 1]$

for $i := 1 \dots n$ **do**

$\boldsymbol{\omega}[i] := RandomDir()$

$\mathbf{x}_i := RayTrace(\mathbf{x}_{i-1}, -\boldsymbol{\omega}_i)$

$L := \frac{L}{2\pi^2 \sin(\theta_i)} + factor \cdot Emission(\mathbf{x}_{i-1}, \boldsymbol{\omega}_{i-1})$

$factor := factor \cdot BRDF(\boldsymbol{\omega}_i, \mathbf{x}_i, \boldsymbol{\omega}_{i-1}) \cdot \cos(\theta_i)$

end for

return L

▷ Surface position

▷ Direction

▷ linear RGB color value

▷ BRDFs multiplied together

▷ Generate random direction vector

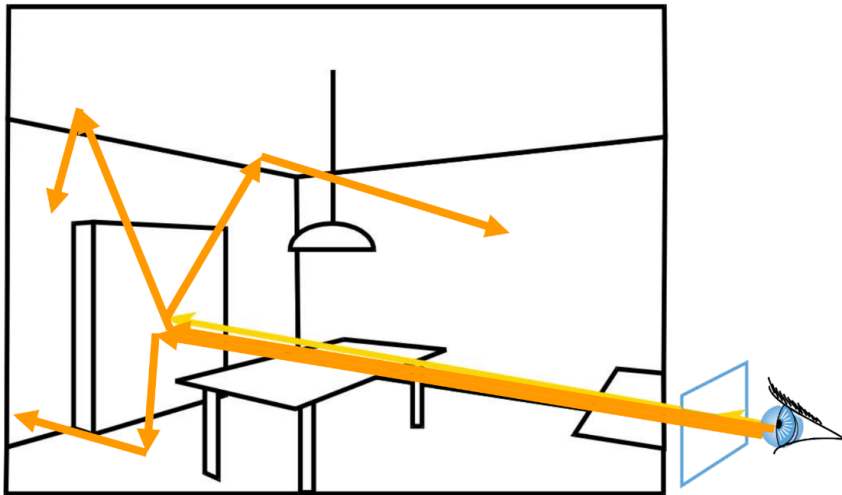
▷ Trace next position

▷ Update summation

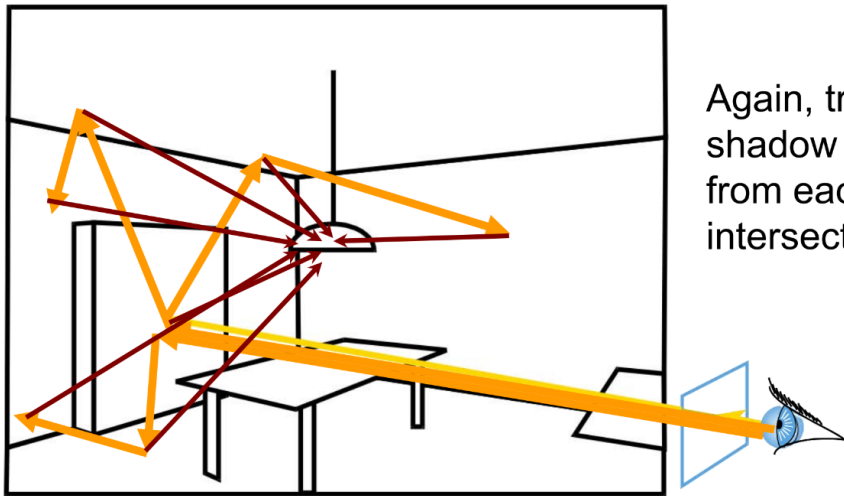
▷ Update product



Path tracing with anti-aliasing



Path tracing with direct light sampling



[See supplemental presentation](#)



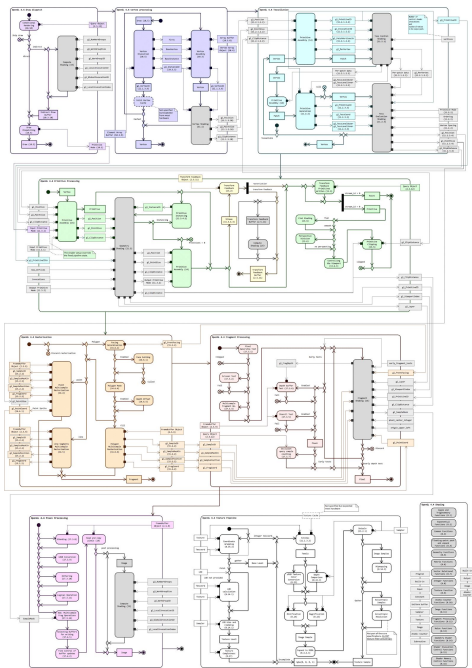
Computer Graphics Lecture

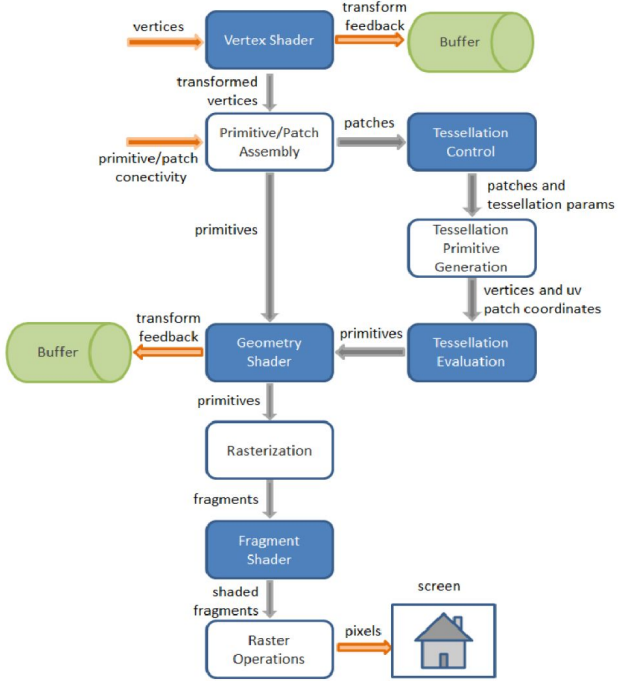
OpenGL overview

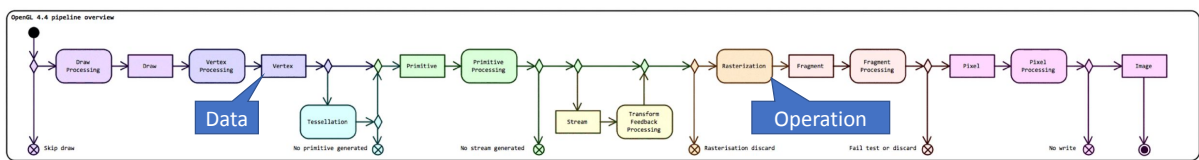
Csaba Bálint

Part I:

The OpenGL Pipeline







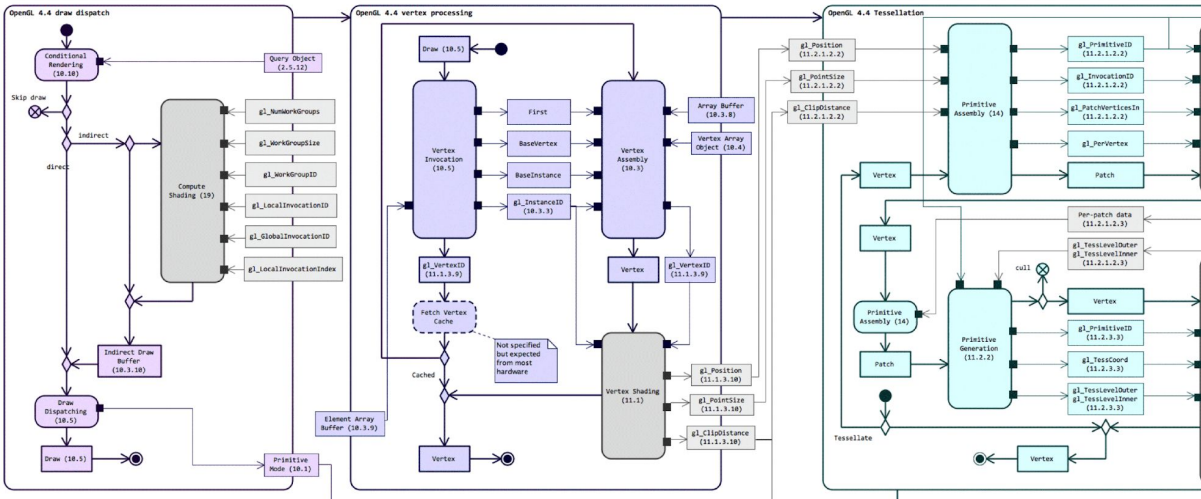
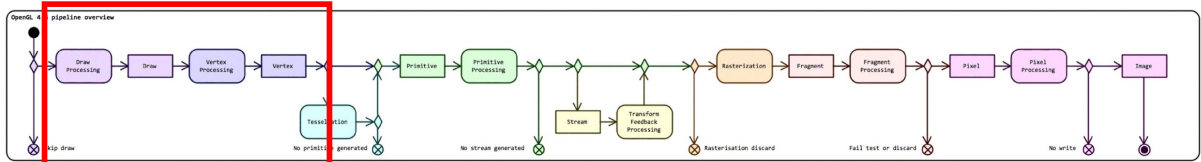
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

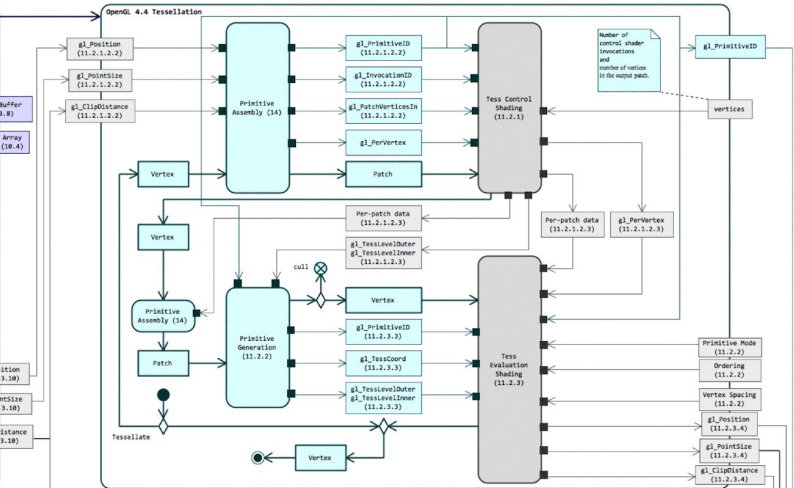
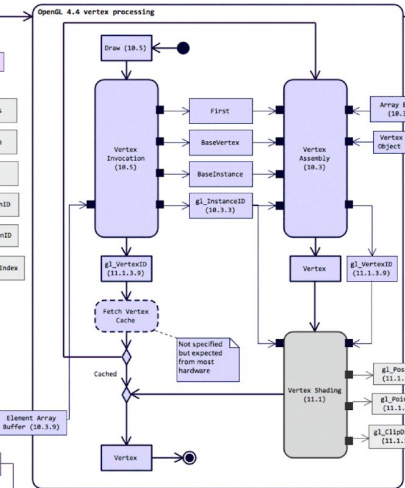
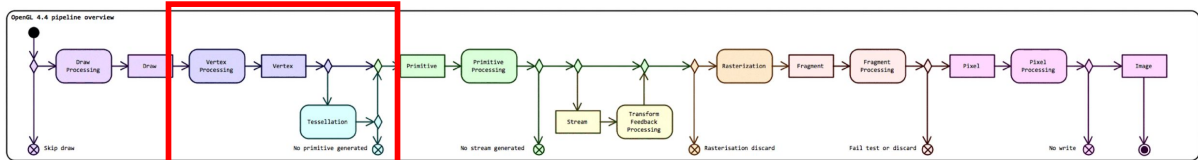
OpenGL shaders

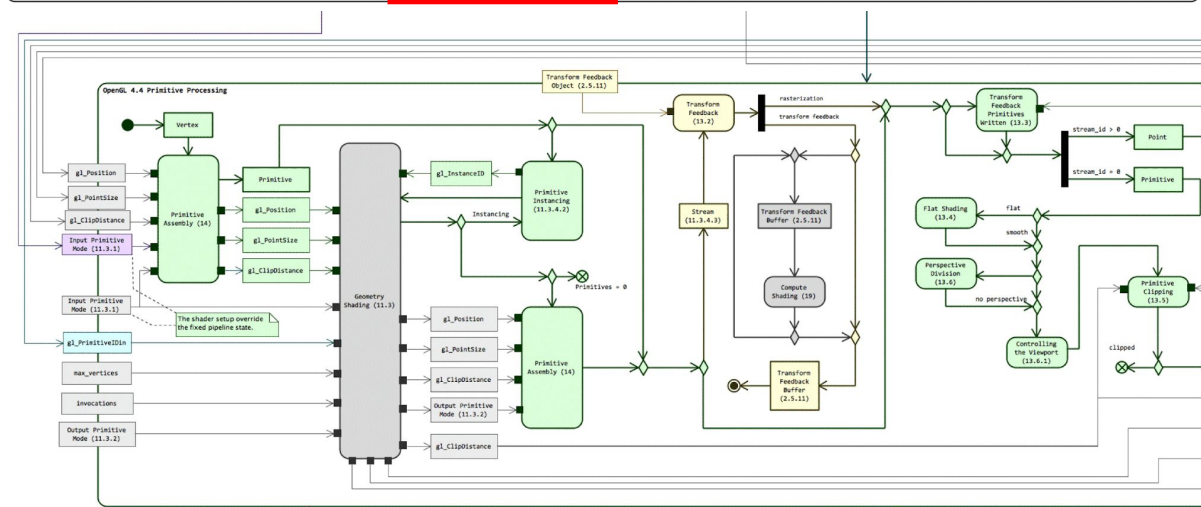
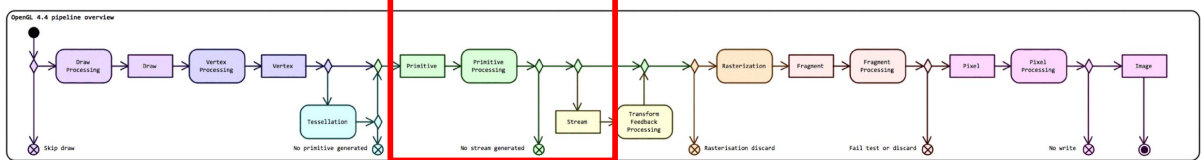
- [Vertex shader](#)
- [Tessellation control shader](#)
- [Tessellation evaluation shader](#)
- [Geometry shader](#)
- [Fragment shader](#)

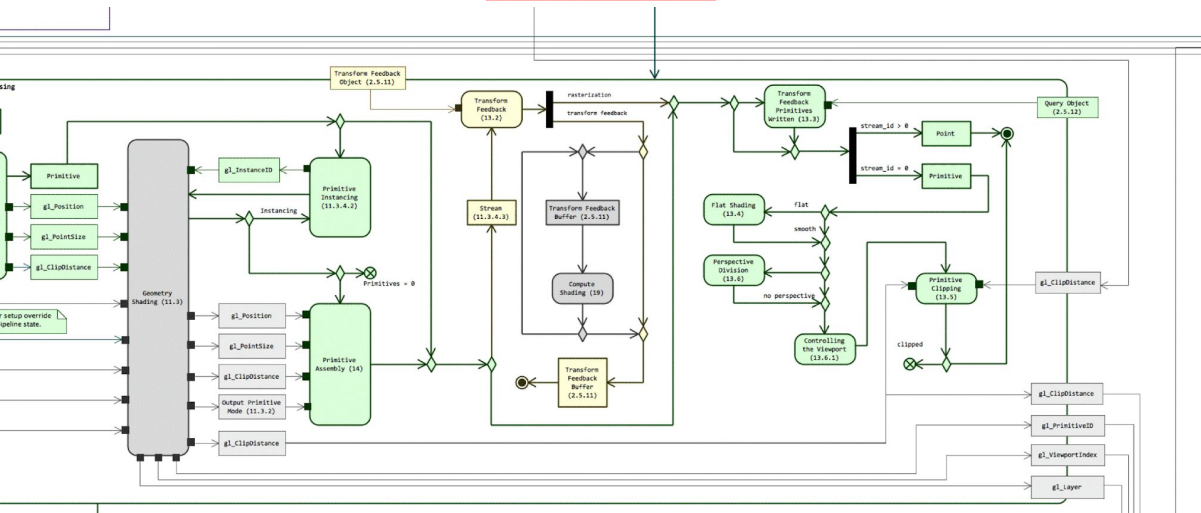
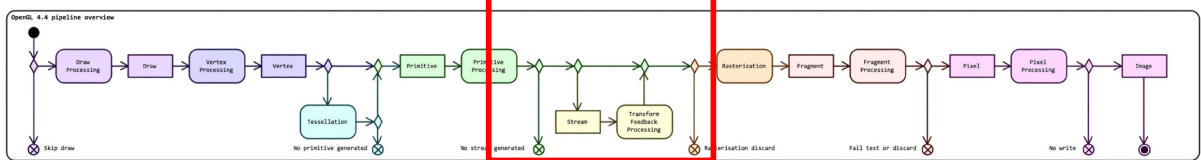
Three optimization opportunities:

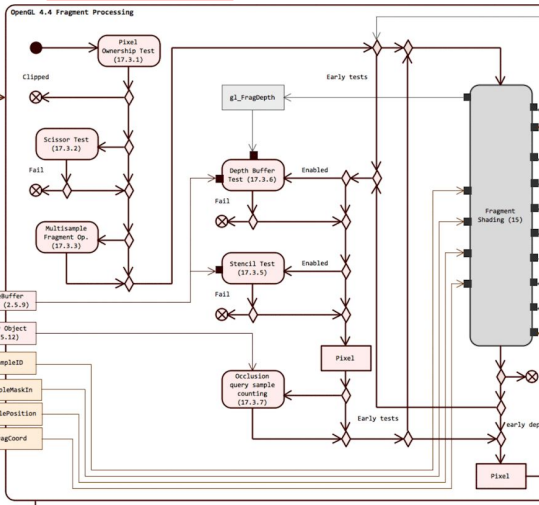
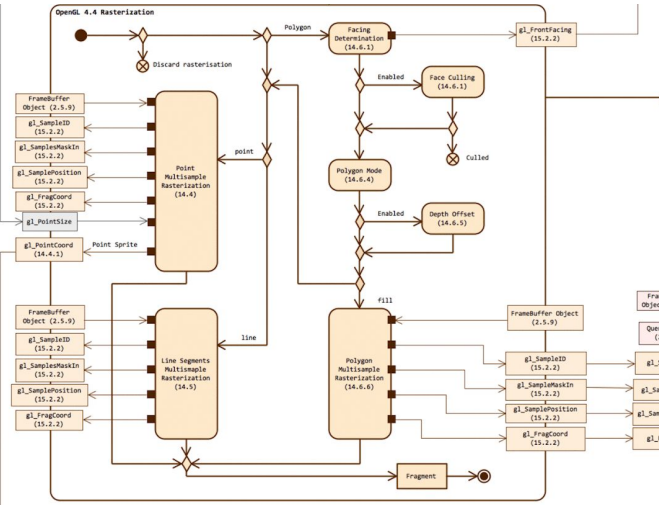
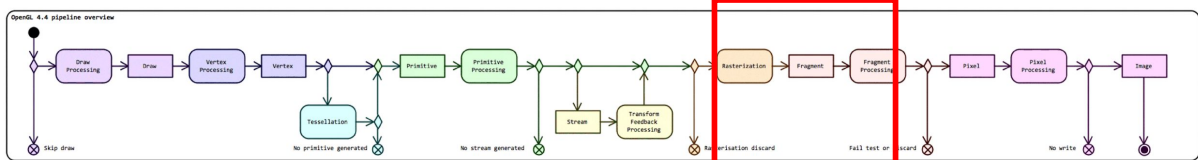
- **Parallelisation** (vertex, fragment, ect.) \Rightarrow Work item, work group, stb. \Rightarrow GPGPU
- **Pipeline** \Rightarrow data streams
- **Memory waits spent computing** \Rightarrow GPGPU

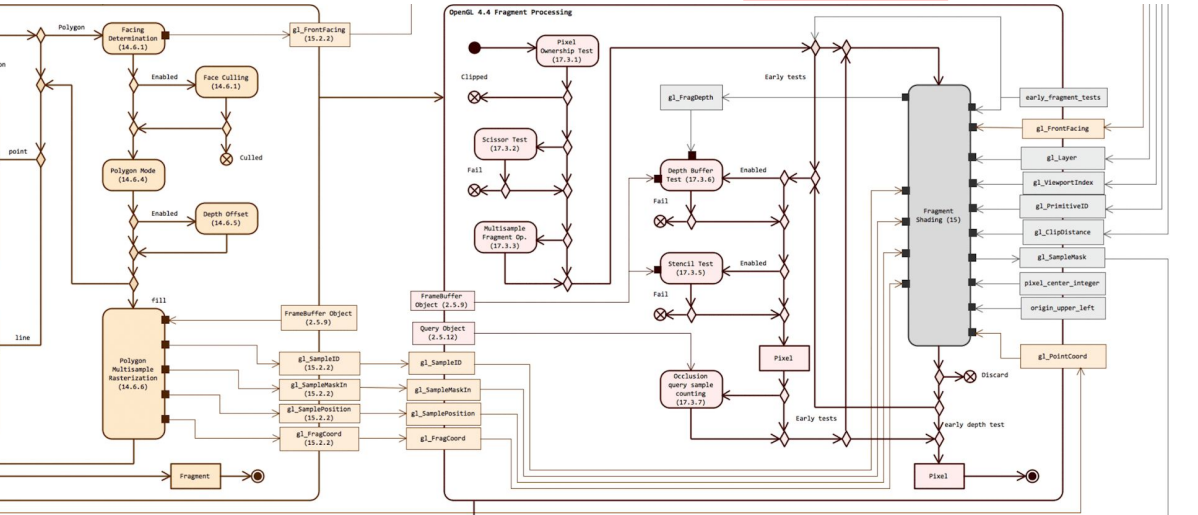
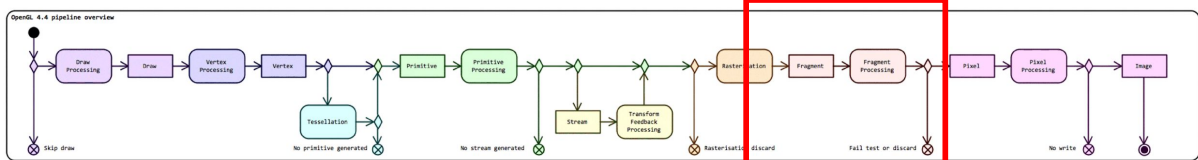


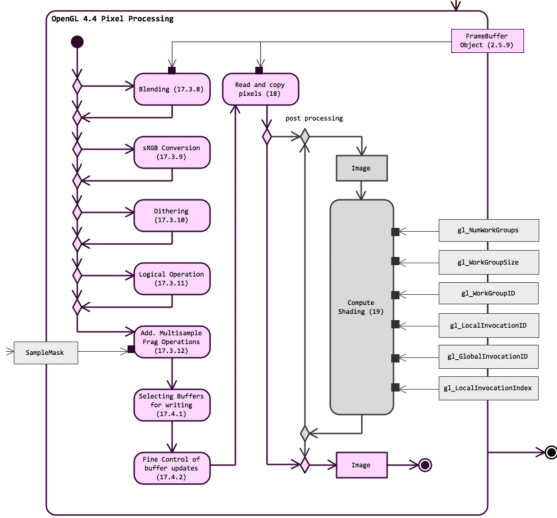
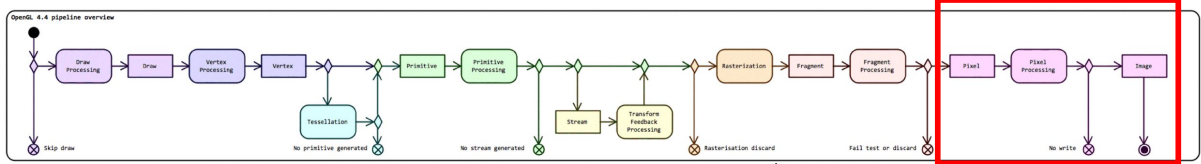






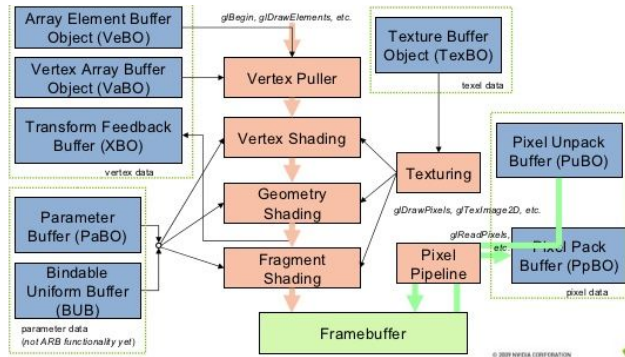






Part II:

OpenGL Objects



VAO – Vertex Array Object

- Draw commands: `gl [Multi] Draw {Arrays, *Elements} [Instanced] *`
- Draw command \Rightarrow Uses active VAO \Rightarrow **VAO defines how the GPU reads the geometry from the VBO** (later)
- Array of `AttribArray`-s. A single AttribArray:
 - **Active or inactive**
 - If inactive, constant value at every location (`glVertexAttrib`)
 - If active, **points** to a memory location (type, size, offset, ect)
 - Reads new pointer after each Vertex, unless set otherwise

Eg.: Position, normal, texture coordinate.

VAO does not hold geometric information, it only holds the layout.

Setting up VAO – old method I'm still popular!

1. Creation of VAO: `glGenVertexArrays (1, &vaoid);`
2. Binding the context: `glBindVertexArray (vaoid)`
3. Setting VBO ptr to VAO: `glBindBuffer (GL_ARRAY_BUFFER, vbooid);`
4. Activation: `glEnableVertexAttribArray(idx);`
5. Setting layout & type: `glVertexAttribPointer*(idx, <type, size>);`
6. Repeat 3-5 for every attribute.
7. Setting up Index buffer : `glBindBuffer (GL_ELEMENT_ARRAY_BUFFER);`

Setting up VAO– „new” method

OpenGL 4.3

1. `glGenVertexArrays`, `glBindVertexArray`

Separate memory & geometry structure:

2. Setting up Buffer Binding Point:

How to read the raw buffer?

- `glBindVertexBuffer` (bindingindex, buffer, offset, stride)
- `glVertexBindingDivisor` (bindingindex, divisor)

How big is the data,
and is it interleaved?

3. Setting up Vertex Attribute Format:

How to interpret the fetched data?

- `glVertexAttribFormat` (attribindex, size, type, normalized, *relativeoffset*)

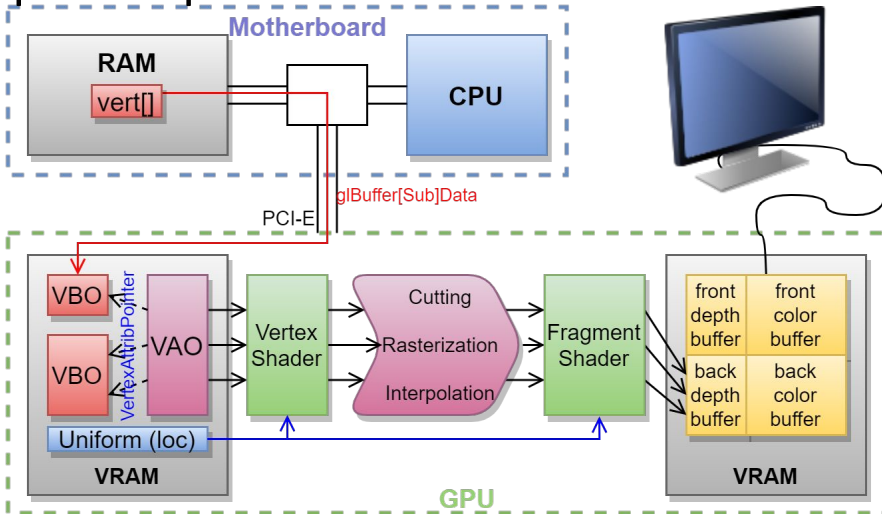
Where does
it start?

Where does the
geometry start?

Which buffer holds
that attribute?

4. Binding the two: `glVertexAttribBinding` (attribindex, bindingindex)

Graphics Pipeline and VAO



Drawing commands: **gl*Draw***

Optimizing memory access:

- ***Elements***: Usage of index buffer (***Arrays*** otherwise).
- ***Range***: The range of the indices are given so the driver can optimize.

Paralellisation of Drawing:

- ***Multi***: Iterate through draw calls efficiently.
- ***Instancing***: Draw same geometry multiple times.

Omitting the GPU \Rightarrow CPU \Rightarrow GPU turnaround:

- ***TransformFeedback***: Send geometry back to be drawn „again”
- ***Indirect***: Read draw parameters from buffer
- ***gl{Begin,End}ConditionalRender***: Rendering object based on condition evaluated on the GPU ([Occlusion queries](#))

OpenGL objects : Name

- Type: **GLuint**. **glGen***, **glCreate***, **glDelete***.
- Zero is the default value. If **glGen*** returns 0 \Rightarrow Error
- Each context has its own namespace (until they are connected: context sharing).
- Typical usage of OpenGL objects:
 1. **Generate** name (declare GLuint, **glGen***, **glCreate***)
 2. **Activate** resource **glBind***
 3. **Call functions & use implicitly** \Rightarrow Internal state changes
 4. **Deactivate** resource (by binding another or **0**)
 5. **Delete** resource (**glDelete***)

OpenGL objects : Binding

- Binding = Coupling to active context
- If coupling is to another object, it is called **attachment!**
 1. First parameter is the **target**. If no index parameter present, only a single resource can be attached to that point.
 2. Optional parameter: **index** can have multiple resources at the same target with different indices. (Multibind)
 3. Second or third parameter is the object's name to be binded.
- Sometimes there are more parameters
- OpenGL can only free up a resource if nothing points to it anymore
⇒ **Including attachments!** ⇒ Deletion orphaning

OpenGL Buffers

- OpenGL Objects

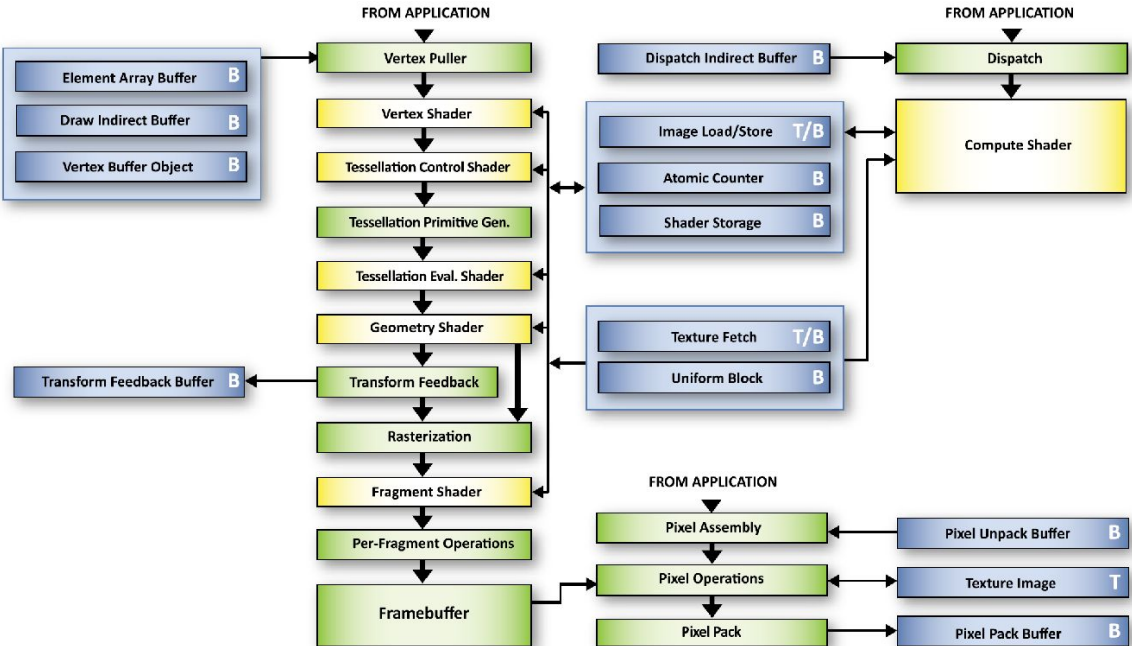
- `glGenBuffers` (1, &bufferName)
- `glBindBuffer` (GLenum target, GLuint bufferName)

- **Unformatted continuous memory**

- OpenGL context allocates it (on the GPU).

- Examples:

- **Vertex data** (VBO = `GL_ARRAY_BUFFER`, index buffer = `GL_ELEMENT_ARRAY_BUFFER`)
- **Arrays for in shader use**(UBO, SSBO)
- **Counters** (atomic counter), **Queries** (`GL_QUERY_BUFFER`, eg. runtime)
- Transform feedback buffer, indirect buffer
- Special textures: texture-, pixel unpack-, pixel pack buffer.



OpenGL Buffer types

Mutable Storage

- Can be allocated any number of times
- `glBufferData`(target, size, data, usage)
 - **Allocates AND uploads!**
 - `GL_{STATIC, DYNAMIC, STREAM}`
 - `_{DRAW, READ, COPY}`
- Hard to read and write data from CPU
- Cannot map memory persistently
- Cacheing can be difficult for the driver
- Inneficient synchronization

Immutable Storage

OpenGL 4.4

- Allocate once
- `glBufferStorage`(target, size, data, flags)
 - `GL_MAP_{READ, WRITE}_BIT`
 - `GL_DYNAMIC_STORAGE_BIT`
 - `GL_{COHERENT, PERSISTENT}_BIT`
 - `GL_CLIENT_STORAGE_BIT`
- All „**server-side**” operations are allowed:
 - Pipeline can write to it
 - Clearable and invalidatable
 - Copy to another
 - `glGetBufferSubData` ⇒ odd one out

OpenGL Buffer operations

- Clear: `glClearBuffer[Sub]Data`

OpenGL 4.3

OpenGL 1.5

- Overwrite: `glBufferSubData` (does not allocate) often slow

- Read: `glGetBufferSubData`

OpenGL 2.0

- Copy: `GL_COPY_READ_BUFFER` ⇒ `GL_COPY_WRITE_BUFFER`
`glCopyBufferSubData`

OpenGL 3.1

- Mapping: `glMapBuffer[Range]` returns a readable/writable ptr.

- `glUnmapBuffer` ends operation and flushes updates.
- If `GL_MAP_PERSISTENT_BIT` is not set, then memory locked when not mapped.
- Flush early by using `glFlushMappedBufferRange`.
- Can be much faster than `glBufferSubData`.

OpenGL 1.9!

A Taste of Types

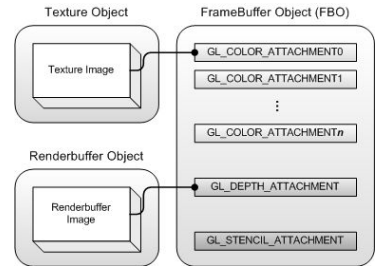
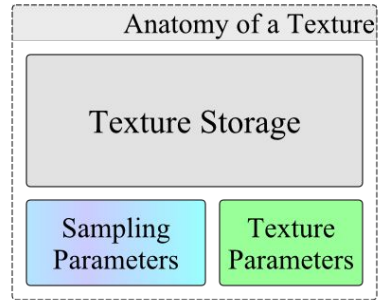
On the CPU side

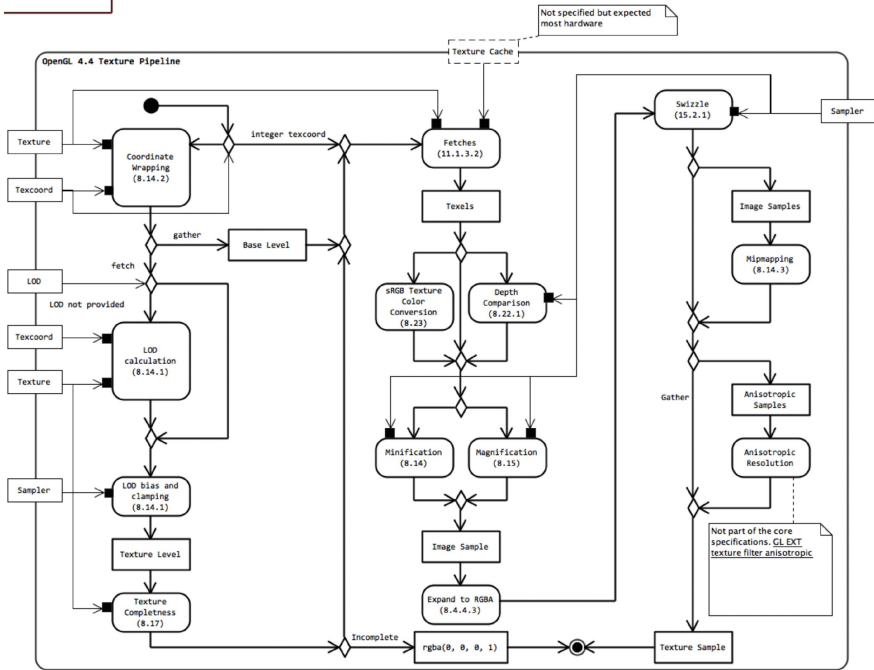
Bitdepth	8	16	32	64
Integers	GLbyte	GLshort	GLint	GLint64
Naturals	GLubyte	GLushort	GLuint	GLuint64
Floating point		GLhalf	GLfloat	GLdouble

Other types: **GLboolean**, **GLfixed**, **GLbitfield**, **GLenum**, **GLsizei**, **GLclampf**, **GLintptr**, **GLsizeiptr**, **GLclampd**.

Part III:

Textures and Framebuffers





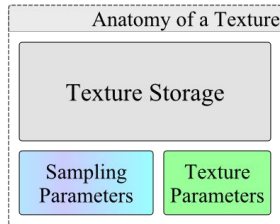
OpenGL Textures

- OpenGL Objects

- `glGenTextures` (1, &textureName)
- `glBindTexture` (GLenum target, GLuint textureName)

- **Fast arrays of data with specified pixel formats**

- OpenGL context allocates (on the GPU)



Texture type	Mipmaps	Array Layers	Cubemap Faces	Image dimensionality
GL_TEXTURE_1D	Yes			1D
GL_TEXTURE_2D	Yes			2D
GL_TEXTURE_3D	Yes			3D
GL_TEXTURE_1D_ARRAY	Yes	Yes		1D
GL_TEXTURE_2D_ARRAY	Yes	Yes		2D
GL_TEXTURE_CUBE_MAP	Yes		Yes	2D
GL_TEXTURE_CUBE_MAP_ARRAY	Yes	Yes	Yes	2D
GL_TEXTURE_RECTANGLE				2D
GL_TEXTURE_BUFFER				1D
GL_TEXTURE_2D_MULTISAMPLE				2D
GL_TEXTURE_2D_MULTISAMPLE_ARRAY		Yes		2D

Texture usage

GPU (GLSL)

```
uniform sampler2d mytex;  
//samplerID (int)  
  
vec4 col = texture(mytex, uv);  
//read and interpolate
```

CPU (C++)

```
glActiveTexture(GL_TEXTURE0 +  
               mSamplerID);  
//integers: 0-31  
  
glBindTexture(GL_TEXTURE_2D,  
             mTexName);  
//mSampler points to mTexName  
  
glUniform1i(mLoc, mSamplerID);  
//"mytex" uniform is mSampler
```

Texture semantics

- **Image**: a single {1,2,3}D array of pixels
- **Image format**: type description how a single pixel is stored
- **Sampler**: stores sampling parameters for shaders to access a **texture**
- **Texture**: Contains some number of **images** and a **sampler**
 - same image format, but can have different sizes via mip-maps
- **Texture completeness**
 - Mipmap completeness: $\text{width/height/depth} = \text{floor}(\text{base level} / 2^k)$
 - Cubemap completeness: square images only!
 - Image format completeness: no interpolation for stencil textures
 - Sampler objects and completeness: \rightarrow Sampler object

```
// Create Texture2D:
glGenTextures(1, &mTexName);
glBindTexture(GL_TEXTURE_2D, mTexName);

// Set the texture sampling parameters:
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_MAX_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

// Allocate immutable storage:
glTexStorage2D(GL_TEXTURE_2D, levels, GL_RGB8, w, h);
// Write uninitialized memory. First mip-map layer:
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGB, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D); //fill in the rest
```

OpenGL Texture Storage

Mutable Storage

- Can be allocated any number of times
- `glTexImage{1,2,3}D(...)`
 - **Allocates AND uploads!**
 - Texture can easily be incomplete!
- Hard to read and write data from CPU
- Cacheing can be difficult for the driver
- Inneficient synchronization

Immutable Storage

OpenGL 4.2

- Allocate once!
- `glTexStorage{1,2,3}D(...
internalformat...)`
 - **Base:** `GL_{RED, RG[B[A]]}`
 - Sized: **Base** `{8,16[F],32F}[_SNORM], I, UI`
 - PI: `GL_RG16_SNORM`
- All „**server-side**” operations are allowed:
 - Pipeline can write to it
 - Clearable and copyable
 - `glTexSubImage{1,2,3}D`
 - `glClearTex[Sub]Image`

Sized Internal Format	Base Internal Format	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits
GL_R8	GL_RED	8				
GL_R8_SNORM	GL_RED	s8				
GL_R16	GL_RED	16				
GL_R16_SNORM	GL_RED	s16				
GL_RG8	GL_RG	8	8			
GL_RG8_SNORM	GL_RG	s8	s8			
GL_RG16	GL_RG	16	16			
GL_RG16_SNORM	GL_RG	s16	s16			
GL_R3_G3_B2	GL_RGB	3	3	2		
GL_RGB4	GL_RGB	4	4	4		
GL_RGB5	GL_RGB	5	5	5		
GL_RGB8	GL_RGB	8	8	8		
GL_RGB8_SNORM	GL_RGB	s8	s8	s8		
GL_RGB10	GL_RGB	10	10	10		
GL_RGB12	GL_RGB	12	12	12		
GL_RGB16_SNORM	GL_RGB	16	16	16		
GL_RGBA2	GL_RGBA	2	2	2	2	
GL_RGBA4	GL_RGBA	4	4	4	4	
GL_RGB5_A1	GL_RGBA	5	5	5	1	
GL_RGBA8	GL_RGBA	8	8	8	8	
GL_RGBA8_SNORM	GL_RGBA	s8	s8	s8	s8	
GL_RGB10_A2	GL_RGBA	10	10	10	2	
GL_RGB10_A2UI	GL_RGBA	ui10	ui10	ui10	ui2	
GL_RGBA12	GL_RGBA	12	12	12	12	
GL_RGBA16	GL_RGBA	16	16	16	16	
GL_SRGB8	GL_RGB	8	8	8		
GL_SRGB8_ALPHA8	GL_RGBA	8	8	8	8	
GL_R16F	GL_RED	f16				
GL_RG16F	GL_RG	f16	f16			
GL_RGB16F	GL_RGB	f16	f16	f16		
GL_RGBA16F	GL_RGBA	f16	f16	f16	f16	

GL_R32F	GL_RED	f32				
GL_RG32F	GL_RG	f32	f32			
GL_RGB32F	GL_RGB	f32	f32	f32		
GL_RGBA32F	GL_RGBA	f32	f32	f32	f32	
GL_R11F_G11F_B10F	GL_RGB	f11	f11	f10		
GL_RGB9_E5	GL_RGB	9	9	9		5
GL_R8I	GL_RED	i8				
GL_R8UI	GL_RED	ui8				
GL_R16I	GL_RED	i16				
GL_R16UI	GL_RED	ui16				
GL_R32I	GL_RED	i32				
GL_R32UI	GL_RED	ui32				
GL_RG8I	GL_RG	i8	i8			
GL_RG8UI	GL_RG	ui8	ui8			
GL_RG16I	GL_RG	i16	i16			
GL_RG16UI	GL_RG	ui16	ui16			
GL_RG32I	GL_RG	i32	i32			
GL_RG32UI	GL_RG	ui32	ui32			
GL_RGB8I	GL_RGB	i8	i8	i8		
GL_RGB8UI	GL_RGB	ui8	ui8	ui8		
GL_RGB16I	GL_RGB	i16	i16	i16		
GL_RGB16UI	GL_RGB	ui16	ui16	ui16		
GL_RGB32I	GL_RGB	i32	i32	i32		
GL_RGB32UI	GL_RGB	ui32	ui32	ui32		
GL_RGBA8I	GL_RGBA	i8	i8	i8	i8	
GL_RGBA8UI	GL_RGBA	ui8	ui8	ui8	ui8	
GL_RGBA16I	GL_RGBA	i16	i16	i16	i16	
GL_RGBA16UI	GL_RGBA	ui16	ui16	ui16	ui16	
GL_RGBA32I	GL_RGBA	i32	i32	i32	i32	
GL_RGBA32UI	GL_RGBA	ui32	ui32	ui32	ui32	

Image Load/Store

- Allows **direct** access!

- No interpolation

- Read and **write** within shaders!

- Image uniforms

- Atomic operations!

GPU (GLSL):

```
layout(binding = 0)
[coherent] [volatile] [restrict] [readonly, writeonly]
uniform [iu]image{1,2,3}D myImage;
```

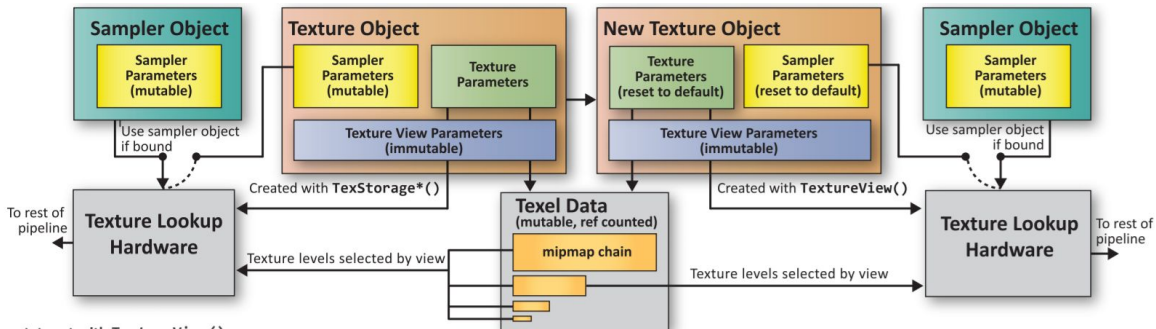
```
vec4 col = imageLoad(myImage, coords);
imageStore(myImage, coords, col);
```

CPU (C++):

```
void glBindImageTexture(GLuint unit, GLuint texture,
    GLint level, GLboolean layered, GLint layer,
    GLenum access, GLenum format);
```

Texture View

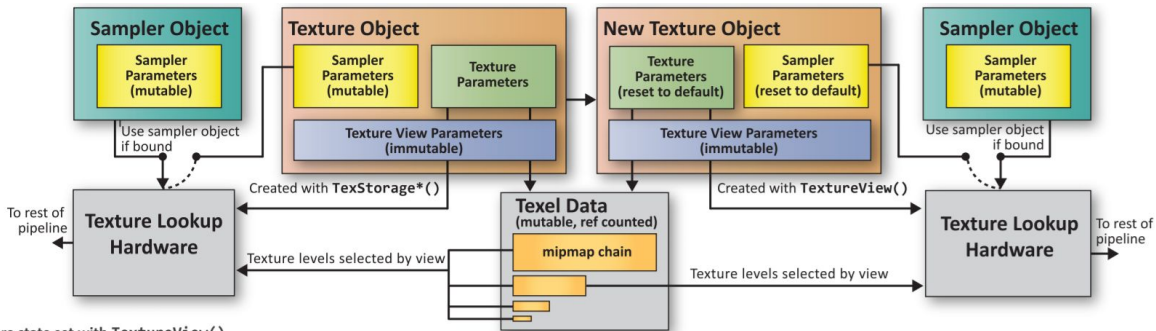
- “Texel Data” is reference counted. \Rightarrow Simple to use!
- Texture Views are normal textures which point to existing data
- They can have different pixel format or layers (immutable)



Texture state set with `TextureView()`

Texture View

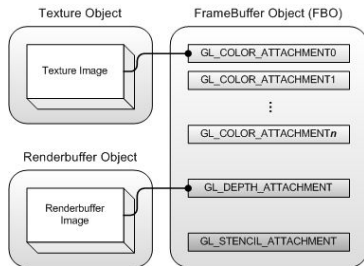
```
void glTextureView(GLuint texture, GLenum target,  
                  GLuint origtexture, GLenum internalformat, GLuint minlevel,  
                  GLuint numlevels, GLuint minlayer, GLuint numlayers);
```



Texture state set with `TextureView()`

Framebuffer (FBO) semantics

- **Renderbuffer**: a single 2D image only needed during rendering to FBO
 - Cannot read nor write to it from shaders.
- **Attach**: To connect one object to another (as opposed to binding).
- **Attachment point**: indexed location within the FBO where an **image** can be attached
- **Framebuffer completeness**:
 - Each texture must be **complete** and **FBO compatible**
 - Draw buffer must be set correctly
 - Ect. Use `glCheckFramebufferStatus`



Framebuffer Object

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GLenum attachment,  
    GLenum textarget, GLuint texture, GLint level);
```

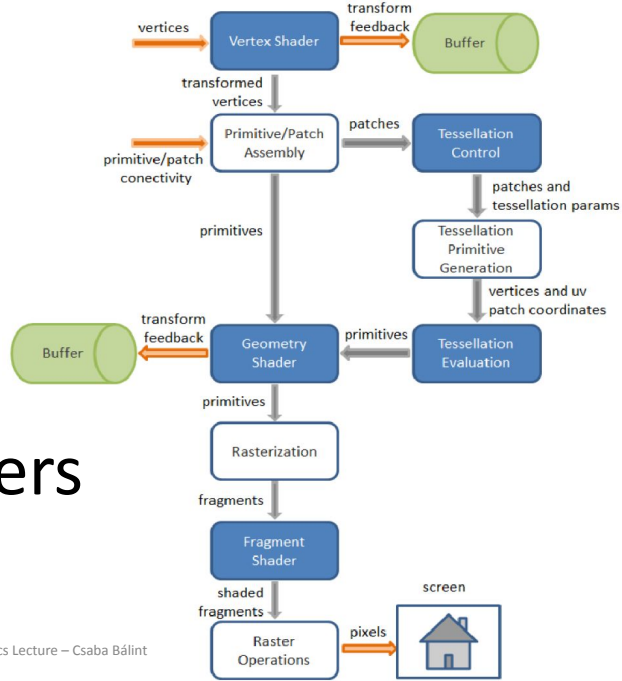
- `GL_COLOR_ATTACHMENTi`, (i=0..8)
 - `GL_DEPTH_ATTACHMENT`,
 - `GL_STENCIL_ATTACHMENT`,
 - `GL_DEPTH_STENCIL_ATTACHMENT`
-
- **Implicit synchronisation** in pipeline. Simple and effective!
 - Do not create **feedback loops**. Undefined behaviour!
 - Connection between the FBO and texture images is like between VAO and VBO-s.

More details

- <https://www.khronos.org/opengl/wiki/Texture>
- https://www.khronos.org/opengl/wiki/Sampler_Object
- https://www.khronos.org/opengl/wiki/Image_Format
- https://www.khronos.org/opengl/wiki/Texture_Storage
- https://www.khronos.org/opengl/wiki/Image_Load_Store
- https://www.khronos.org/opengl/wiki/Framebuffer_Object

Part IV:

Shaders

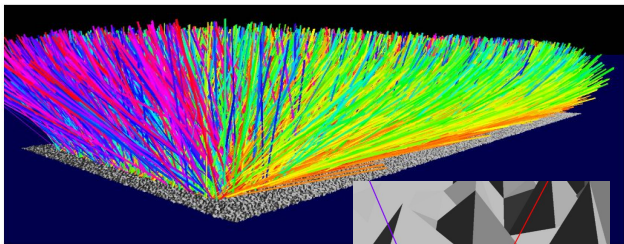


Vertex Shader

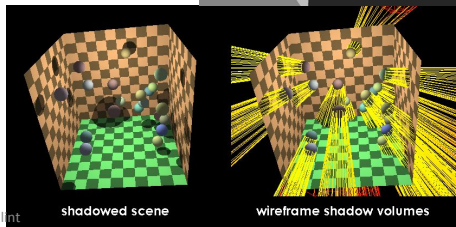
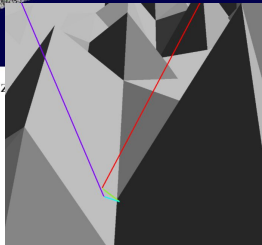
- 1:1 mapping of vertices or `gl_VertexID` to output
- Transformations should go here if further geometry operations are invariant or unaffected by it
- Examples
 - MVP, world, worldIT matrix multiplications
 - Texture read to get geometry, eg. hightmap
 - Patch coefficient calculation for (eg.) tessellation
 - SSBO operations
- Additional clipping with `gl_ClipDistance`

Geometry shader

- Per primitive create more
→ Geometry generation
- 1:N (with predefined maximum)
- Much more sequential than other stages
- Less efficient but allows more
- Examples
 - Every tessellation can be done here
 - For each triangle create an arrow with the normal
 - Shadow geometry generation -- shadow volumes
 - Pointcloud visualization

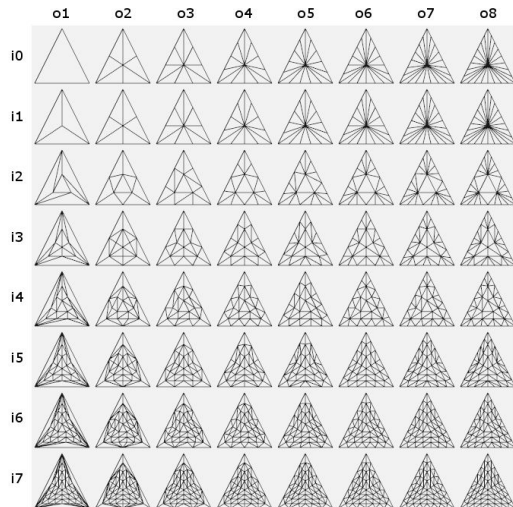


(c) $\sigma = 0.4$ sz



Tessellation Control Shader

- TCS
- Runs for each vertex of the patch
- Outputs an array for the evaluation shader
- `layout(vertices = patch_size) out;`
- Defines the tessellation amount and shape
- `if(gl_InvocationID == 0) {...}`
- `float gl_TessLevelInner[4], gl_TessLevelOuter[3]`

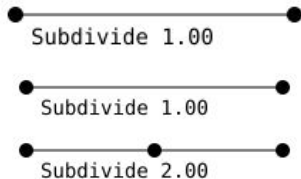


Tessellation Evaluation Shader

- TES
- Parametric evaluation
 - u,v (square) or
 - $u,v, w=1-u-v$ (triangle)
- Very efficient
- The tessellation levels may vary dynamically
- **equal_spacing,**

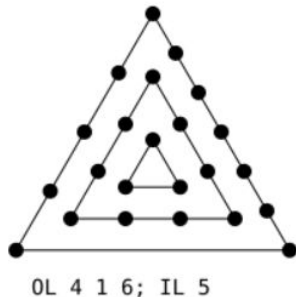
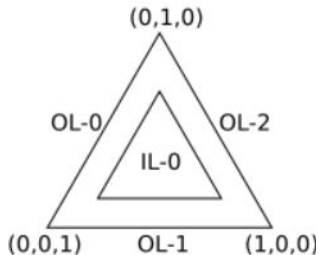
fractional_even_spacing,

fractional_odd_spacing



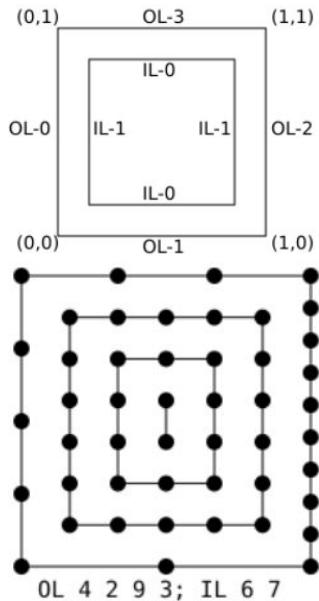
Triangle tessellation

- `gl_TessCoord.xyz` barycentric coordinates, $0 \leq u, v, w \leq 1$, $u+v+w=1$
- `patch out float gl_TessLevelOuter[3];`
- `patch out float gl_TessLevelInner[1];`
- Triangularization is up to vendor



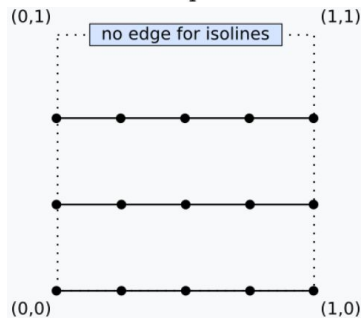
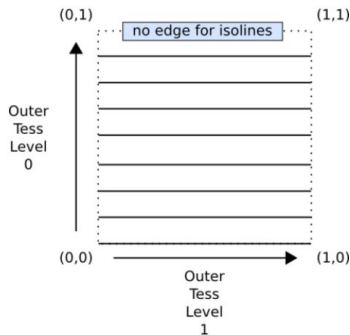
Quad tessellation

- `gl_TessCoord.xy` normalized patch coordinates in $[0,1]^2$
- `patch out float gl_TessLevelOuter[4];`
- `patch out float gl_TessLevelInner[2];`
- Triangularization is up to vendor



Isolines tessellation

- `gl_TessCoord.xy` normalized patch coordinates in $[0,1]^2$
- `patch out float gl_TessLevelOuter[2];`
- `patch out float gl_TessLevelInner[0];`
- Produces lines



Fragment shader

Inputs:

- in vec4 **gl_FragCoord**; in bool **gl_FrontFacing**; in vec2 **gl_PointCoord**;
- in int **gl_SampleID**; in vec2 **gl_SamplePosition**; in int **gl_SampleMaskIn**[];
- in float **gl_ClipDistance**[]; in int **gl_PrimitiveID**;
- in int **gl_Layer**; in int **gl_ViewportIndex**;

Outputs:

glBindFragDataLocation

- Color: layout(location = 3) out vec4 **diffuseColor**;
- Depth: layout (depth_{any,greater,less,unchanged}) out float **gl_FragDepth**;
- Sample: out int **gl_SampleMask**[];

Textures, mappings, sampling

Csaba Bálint
with selected slides of
Gábor Valasek

Motivation

VP MATERIAL MANAGER LITE EDITION (User Library) Attributes

Material Type	Stone Type	Surface	Style	Colors	Renderer
19 Stone	4 Granite	15 Clean	3 Smooth	6 Black	111 Scanline
3 Wood	7 Sandstone	9 Dirty	5 Ragged	18 White	28 MentalR
Metal	6 Marble	6 Reflective	14 Round	9 Red	128 VRay
Plastic	6 Limestone	7 Rough	19 Flat	6 Green	111 Corona
Glass	4 Clay	17 Light	56 Regular	4 Blue	Octane R
Liquid	1 Concrete	4 Dark	29 Irregular	4 Yellow	
2 Ceramic	2 Travertine	4 Glazed	7 Shiny	11 Brown	
9 Nature	Slate	7 Transparent	Illuminated	14 Grey	

Walls & Tiles Mosaic Omnitiles Crossmap Color Extract Automap Favorites Sort by Last Update

Rocky Stonefloor

My Rating ★★★★★

134 of 136 Total (1 Selected)

Copyright © VIZPARK.com

20 Smart Material Mix

<https://www.artstation.com/polygon>

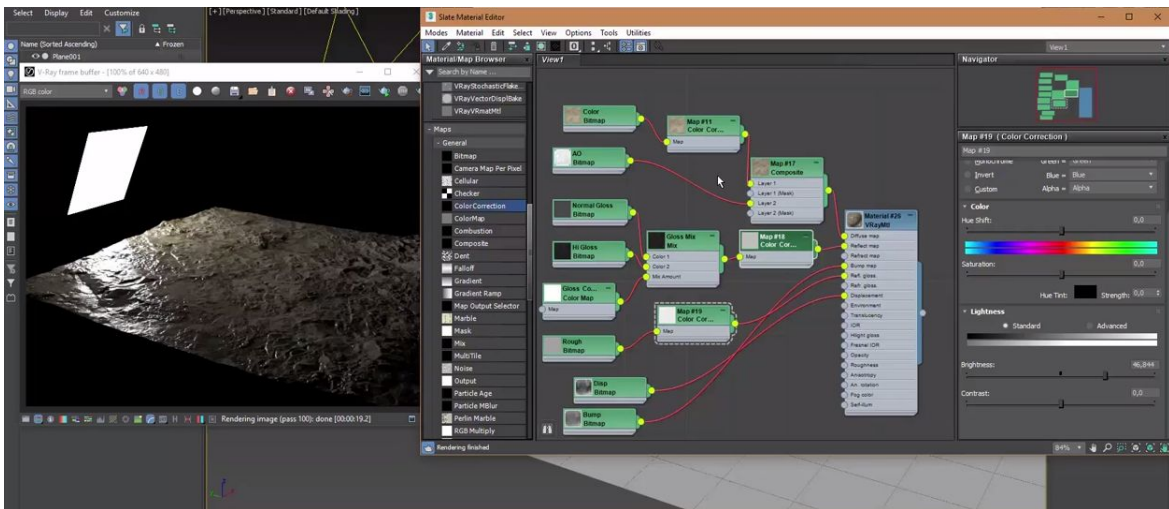
- Materials are commented and set up for easy editing.
- Contains anchor points systems.
- These materials can only be used in substance painter.

SUBSTANCE PAINTER

POLIGONE GEMSTONE SMART MATERIAL

www.artstation.com/polygon

Motivation



References and recommended reading

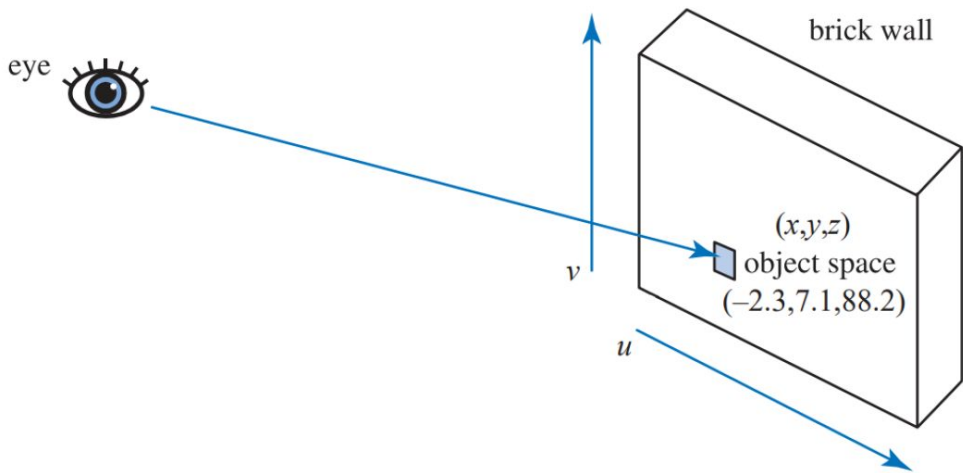
- [Real time rendering](#), 4th edition
- <http://www.reedbeta.com/blog/understanding-bcn-texture-compression-formats/>
- A quick summary: <http://acko.net/files/gltalks/pixelfactory/online.html#0>

Overview

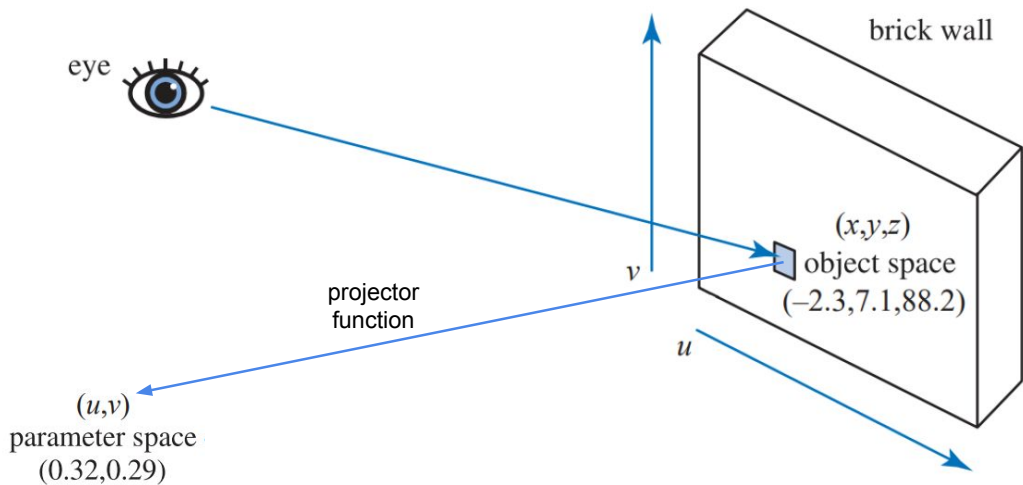
1. **Texture mapping**
2. **Texture filtering**
3. **Texture representations**
4. **Texture mapping techniques**
5. **3D Textures and procedural textures**
6. **Normal and bump mapping**
7. **Per pixel displacement mapping**

Texture mappings

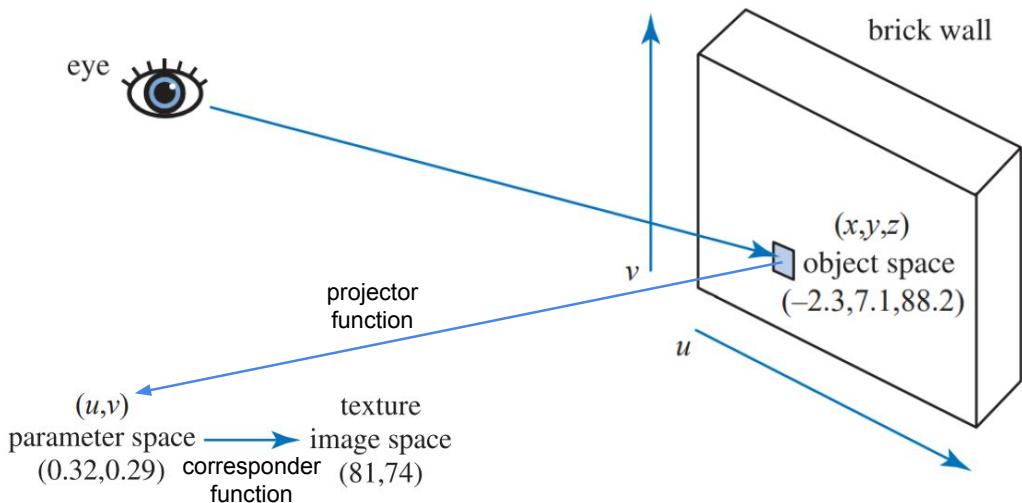
Texturing pipeline



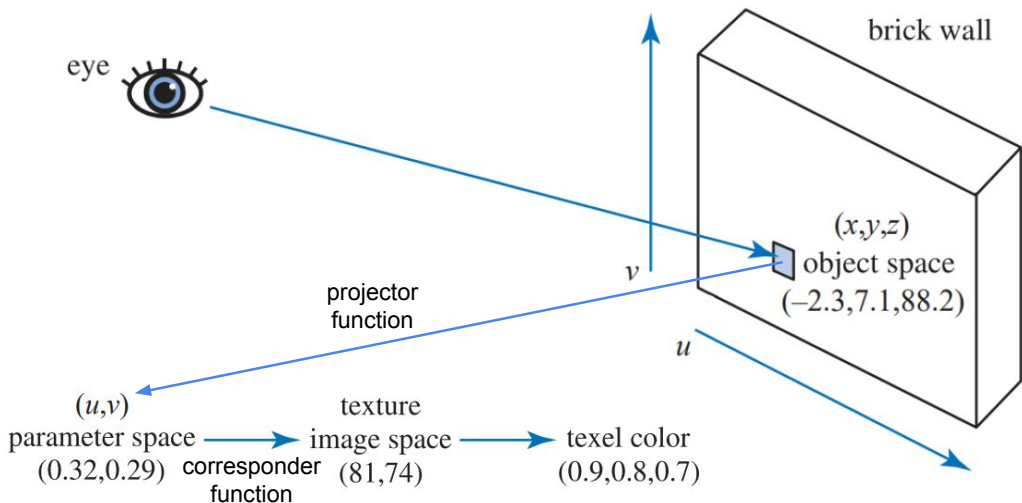
Texturing pipeline



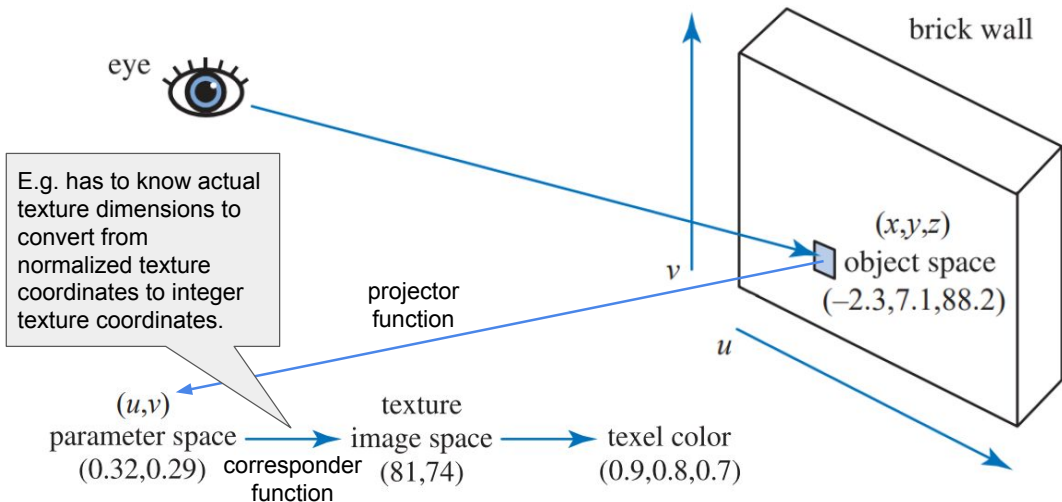
Texturing pipeline



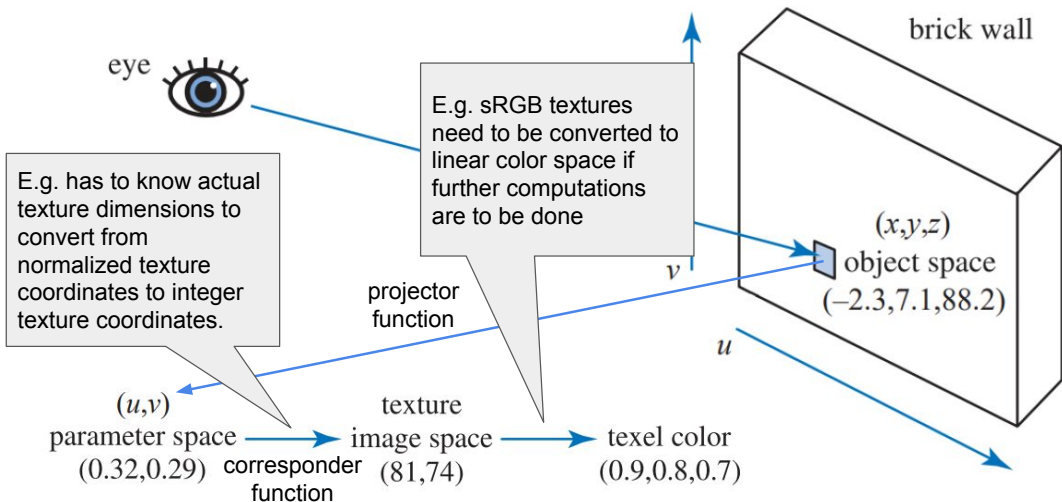
Texturing pipeline



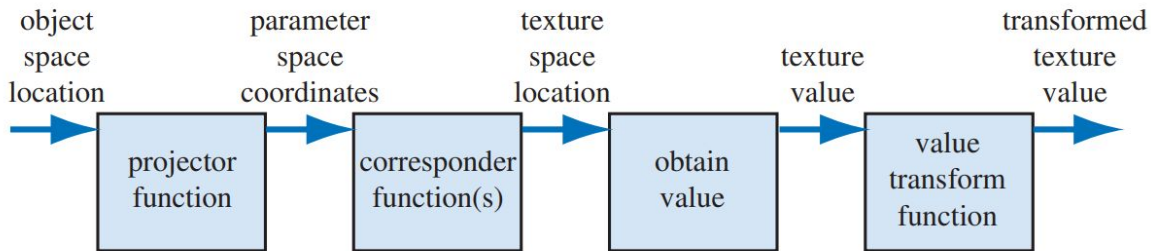
Texturing pipeline



Texturing pipeline



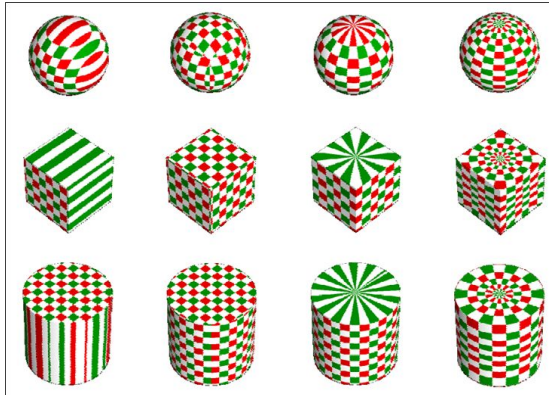
Texturing pipeline



Projector function

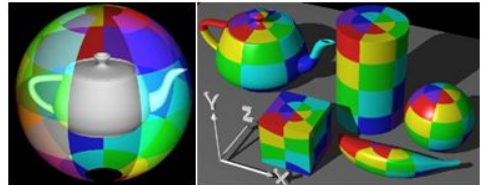
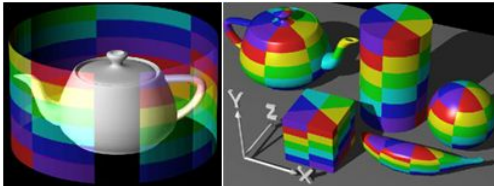
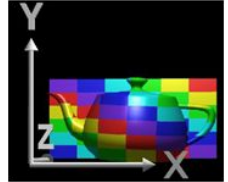
- Project the currently shaded world space position $\mathbf{x} \in \mathbb{E}^3$ to normalized texture coordinates $(u, v) \in [0, 1]^2$
- Done either via
 - **Primitive projector functions:** it is easy to derive the inverse of a planar, cylindrical or spherical projection (actually, you've already done the grunt work for the latter during BSc and you called it Cartesian to spherical conversion:
$$\phi = \text{atan2}(y, x), \theta = \arccos \frac{z}{r}$$
All there's left is to convert $(\phi, \theta) \in [0, 2\pi) \times [0, \pi]$ to $[0, 1] : (\frac{\phi}{2\pi}, \frac{\theta}{\pi})$
 - **Unwrapping:** artist provide (u,v) coordinates per vertex and we interpolate them for the rasterized fragment (in perspective correct way, if needed)

Two part mapping

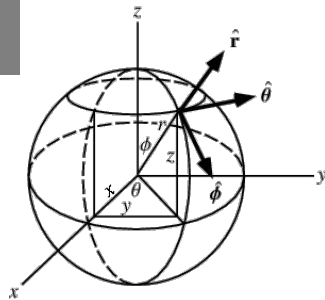
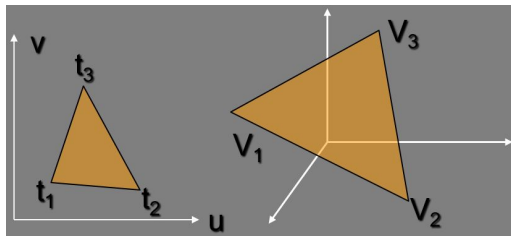
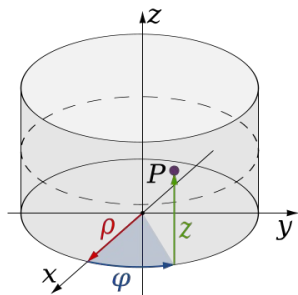


Two part mapping

- Spheres and cylinders are rare.
- Tedious work to write down every texel coordinate
- **Solution:** introduce an **intermediary surface**
 - Simple textured and parameterized intermediary surface
 - **O-mapping:** Object point is mapped to a surface point on intermediary surface
 - **S-mapping:** Surface points of the intermediary surface are mapped to texture space

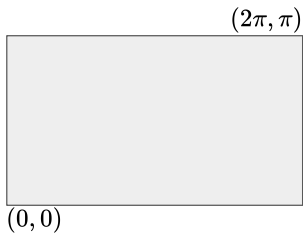


Cylinder, Sphere, and Triangle (**S**-Mapping)

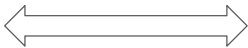


Projector functions: sphere

An origin centered sphere can be parametrized as



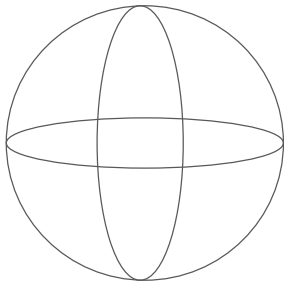
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = r \cdot \begin{bmatrix} \cos \phi \cdot \sin \theta \\ \sin \phi \cdot \sin \theta \\ \cos \theta \end{bmatrix}$$



$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\phi = \text{atan2}(y, x)$$

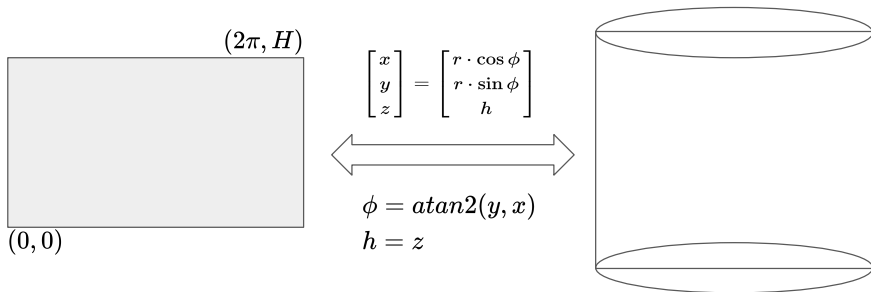
$$\theta = \text{acos} \frac{z}{r}$$



You obtain normalized texture coordinates as

$$(u, v) = \left(\frac{\phi}{2\pi}, \frac{\theta}{\pi} \right)$$

Projector functions: cylinder



You obtain normalized texture coordinates as

$$(u, v) = \left(\frac{\phi}{2\pi}, \frac{h}{H} \right)$$

Projector functions: triangle

- Use 3 vertices and 3 corresponding UV coordinates: $\mathbf{x}_i \in \mathbb{E}^2, \mathbf{u}_i \in \mathbb{R}^2$
- Compute a 3x3 transformation mapping that maps the Euclidean space to the texture space
- I.e. you need to solve

$$A \cdot [\mathbf{x}_1^T, \mathbf{x}_2^T, \mathbf{x}_3^T] = \left[\begin{bmatrix} \mathbf{u}_1 \\ 1 \end{bmatrix}, \begin{bmatrix} \mathbf{u}_2 \\ 1 \end{bmatrix}, \begin{bmatrix} \mathbf{u}_3 \\ 1 \end{bmatrix} \right]$$

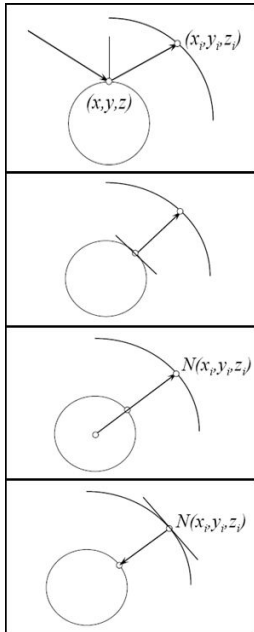
- If the points do not lie on a line, the solution is

$$A = [\mathbf{x}_1^T, \mathbf{x}_2^T, \mathbf{x}_3^T]^{-1} \cdot \left[\begin{bmatrix} \mathbf{u}_1 \\ 1 \end{bmatrix}, \begin{bmatrix} \mathbf{u}_2 \\ 1 \end{bmatrix}, \begin{bmatrix} \mathbf{u}_3 \\ 1 \end{bmatrix} \right]$$

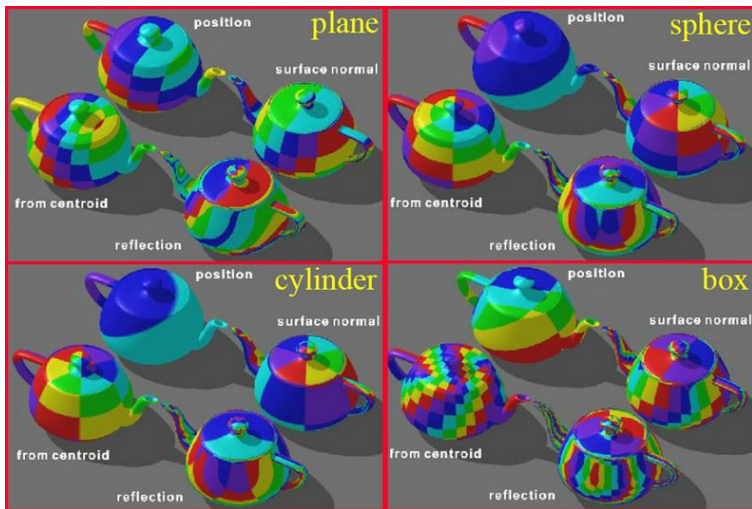
- \Rightarrow Screen-space interpolation without explicitly computing the matrix

Projector functions: O-map

- **Reflection vector:** intersection between the reflected eye ray and the proxy geometry that we use for texture coordinate extraction
- **Object normal:** intersection between the ray from the shaded point towards the surface normal
- **Centroid:** intersection between the line formed by the object barycenter and the vertex and the simpler geometry
- **Proxy's normal:** above which point of the proxy is our query point directly above?



Projector functions: two-part mapping - S-map

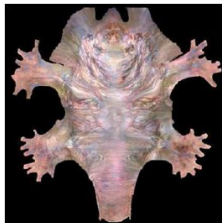
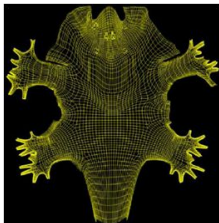
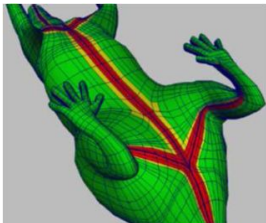
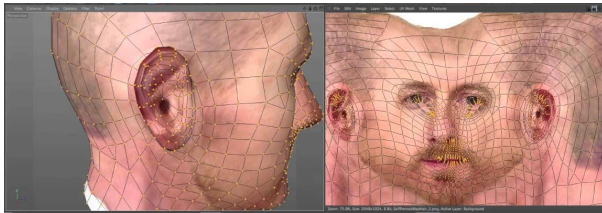


Projector functions: two-part mapping - S-map

	Plane	Cylinder	Sphere	Cube
Reflected ray	View dependent (environment mapping)			
Object normal	Redundant	Low quality	Adequate	Adequate
Centroid	Redundant	Low quality	Good	Good
Proxy's normal	Projector effect	Shrinking effect	Redundant	Good

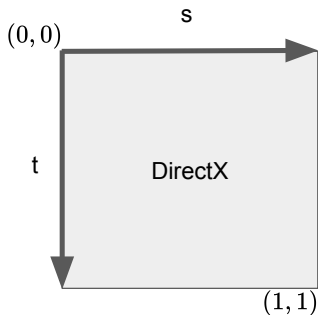
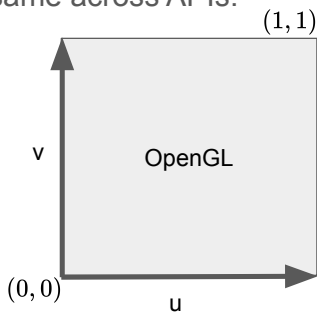
Projector functions: unwrapping

- Bijection needed \Rightarrow problems:
 - Distorted unwrapped geometry or
 - A lot of cuts in unwrapped geometry
 - Unused texture space
- Optimization problem
 - Still semi-automatic!



Corresponder function

- Convert continuous texture coordinates to texture space locations (i.e. normalized texture coordinates)
- There are some 'texture space' funkiness, not even the handedness is the same across APIs:



Corresponder function

- You can also use transformations on the incoming normalized texture coordinates
- And determining how you compute a normalized texture coordinate from an arbitrary continuous coordinate is also part of corresponders, i.e. [addressing](#)

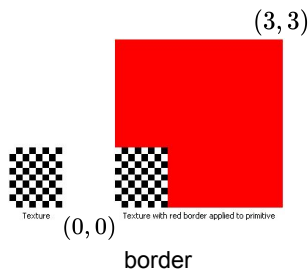
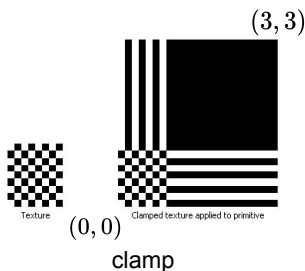
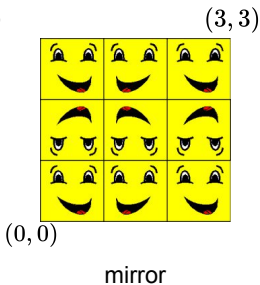
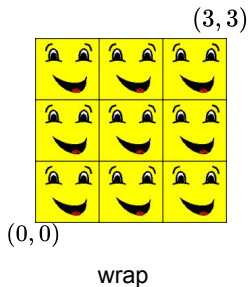


Image texturing

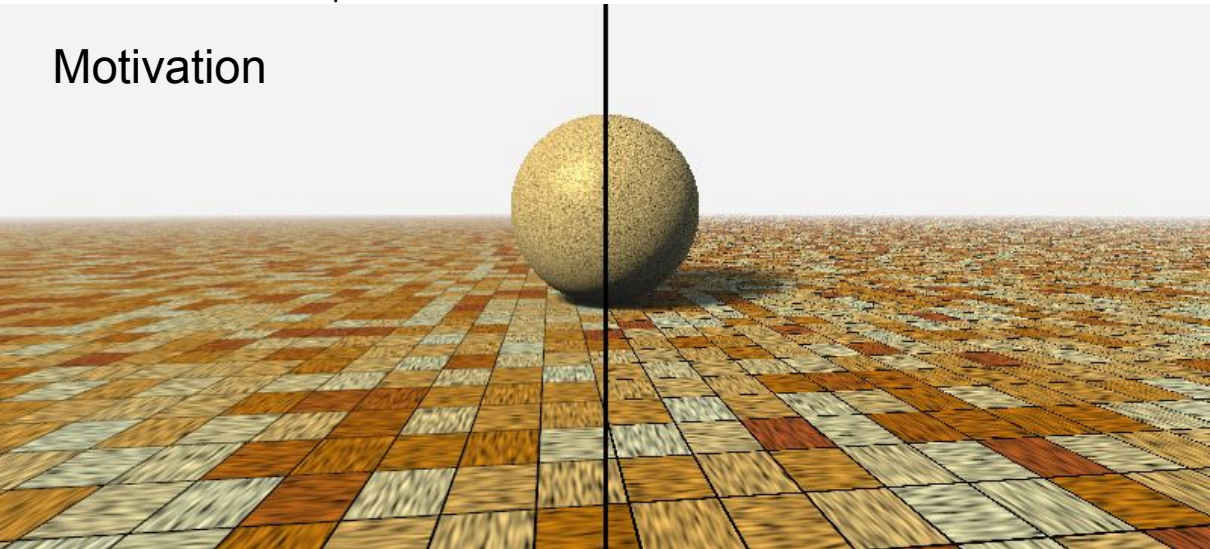
- Given a normalized texture coordinate, query the texture to return a value
- Remove the abstraction of normalized coordinates: multiply (u,v) by width and height
- Convert these to array indices
- Remember: a face of the texture is a collection of discrete samples
- Here, we are given a continuous (*) coordinate tuple and in turn, we should return a continuous texture function value for each input
- Inferring a continuous function from a discrete set of samples is often referred to as **filtering** or **reconstruction filtering** in the literature

*: sans discretization due to floating point representations

Texture sampling

A must see presentation: <http://acko.net/files/gltalks/pixelfactory/online.html#0>

Motivation

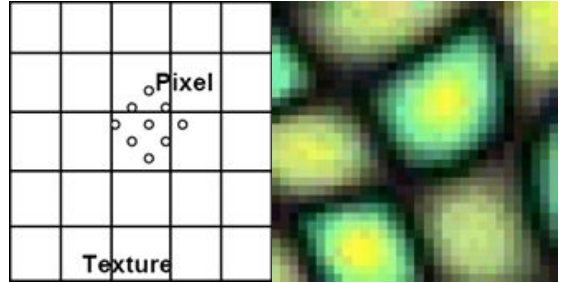


<https://www.iquilezles.org/www/articles/filtering/filtering.htm>

Sampling problems

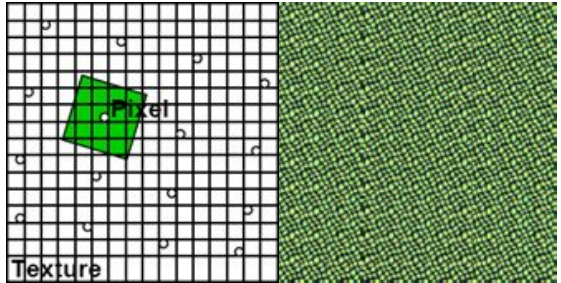
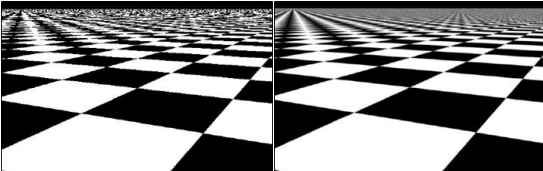
- **Magnification problem**

- Less than one texel per pixel
- Blocky



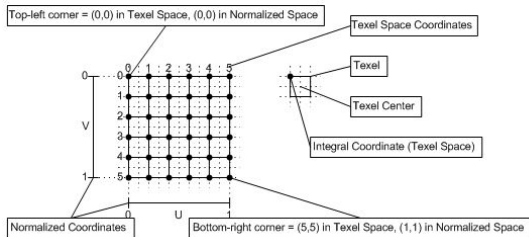
- **Minification problem**

- More than one texel per pixel
- Aliasing



Dimension-upscaled coordinates of samples

- Two conventions to convert dimension-upscaled texture coordinates to integer array indices:
 - Truncating: now in both DX and OpenGL
 - Rounding: used by DX previously
- By now, both DX and OpenGL uses the same convention (truncate) to convert from upscaled texture space coordinates to integer indices



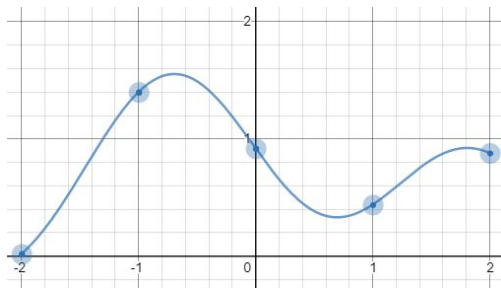
Dimension-upscaled coordinates of samples

- The upscaled coordinates of texel (i, j) are (i+0.5, j+0.5)
- In turn, before normalization, if you want to access texel (i, j), regardless of filtering, you have to sample using the normalized texture coordinates of
$$u_i = (i + 0.5) / W;$$
$$v_i = (j + 0.5) / H;$$
- Among other things, that's why power of two texture dimensions are useful: so that you can exactly and efficiently represent the results of the division
- If you perturb your texture coordinates, you can use some types of non-linearly weighted filterings with the GPU's bilinear filtering acceleration; knowing the above is useful for that
- (If you know what texel to fetch, just use a Load instruction - that's faster)

Periodic band-limited signals

- The Whittaker–Shannon theorem states that band-limited periodic signals can be exactly represented by a discrete set of samples given proper samples as

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] \operatorname{sinc}\left(\frac{t - nT}{T}\right)$$

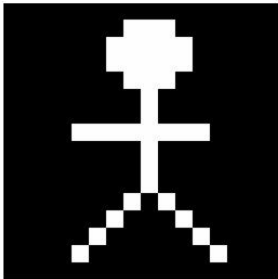


<https://www.desmos.com/calculator/gxocp9afaq>

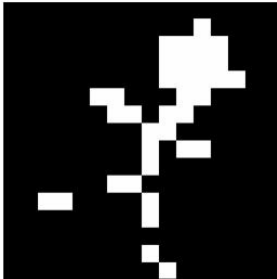
Reconstruction filtering

- Sinc has infinite support: not feasible for arbitrary input
- The simplest thing to do is to do nearest neighbor sampling
- Hardware also supports bilinear and trilinear interpolation (which are actually 2 and 3 dimensional tensor product linear interpolations)

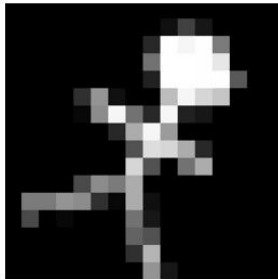
Original Image



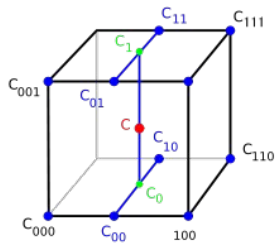
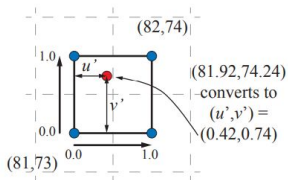
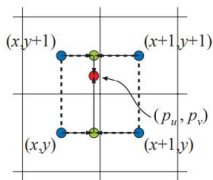
Nearest Neighbor



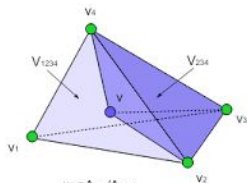
Bilinear Interpolation



Tensor product filtering

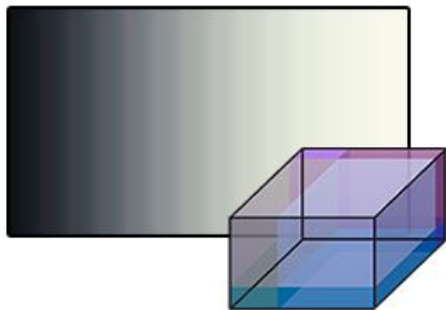


Simplex filtering

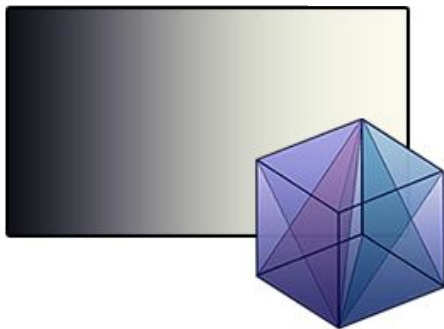


$$w_1 = A_{234} / A_{1234}$$
$$A_{234} = \det(V_1 - V, V_3 - V, V_2 - V) / 2$$
$$A_{1234} = \det(V_4 - V_1, V_3 - V_1, V_2 - V_1) / 2$$

Trilinear



Tetrahedral



Tensor product VS simplex interpolation

- In D dimensions
 - Need 2^D samples for tensor product interpolation
 - $D+1$ samples for simplex interpolation
- What are the trade-offs?
 - Have to manually interpolate and compute the barycentric weights
 - Certain continuity conditions may be more easily shown for tensor product constructs
 - The precision itself is the same, at least upper bound-wise: according to Taylor's theorem, your error is roughly the same either way inside the convex hull of the closest samples:

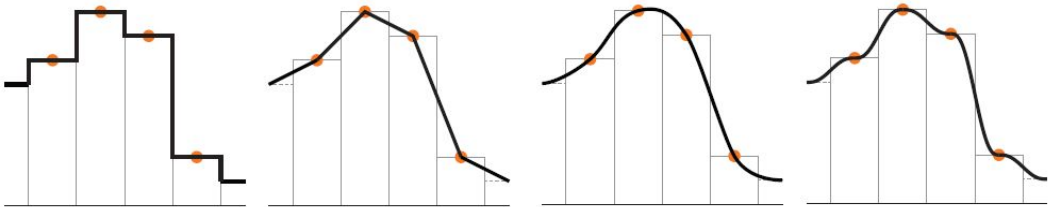
$$\begin{aligned} |f(\mathbf{x}) - \sum \lambda_i(\mathbf{x}) T_i(\mathbf{x})| &\leq \sum \lambda_i(\mathbf{x}) \cdot |R_i(\mathbf{x})| \\ &\leq \max_i |R_i(\mathbf{x})| \end{aligned}$$

where

$$T_{k, \mathbf{x}_0}(\mathbf{x}) = \sum_{|\boldsymbol{\alpha}| \leq k} \frac{\partial^{\boldsymbol{\alpha}} f(\mathbf{x}_0)}{\boldsymbol{\alpha}!} (\mathbf{x} - \mathbf{x}_0)^{\boldsymbol{\alpha}}$$

Magnification

- When a texel covers several pixels
- Hardware supports nearest neighbor and bilinear interpolation
- Sometimes you need cubic filters (basically: convolution, i.e. linear filtering)
 - And there are some tricks to make it use GPU too:
<https://developer.nvidia.com/gpugems/gpugems2/part-iii-high-quality-rendering/chapter-20-fast-third-order-texture-filtering>
 - Balázs Csébfalvi: <https://dl.acm.org/doi/10.1145/3306346.3323032>



Magnification: nearest VS bilinear VS bicubic



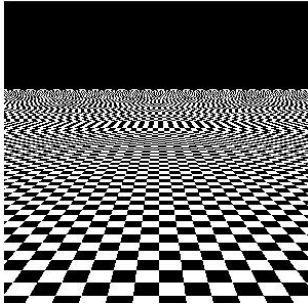
Magnification: Detail textures

- Detail textures can help to avoid the blurry look
- It is a carefully crafted texture that adds high frequency detail as we get closer

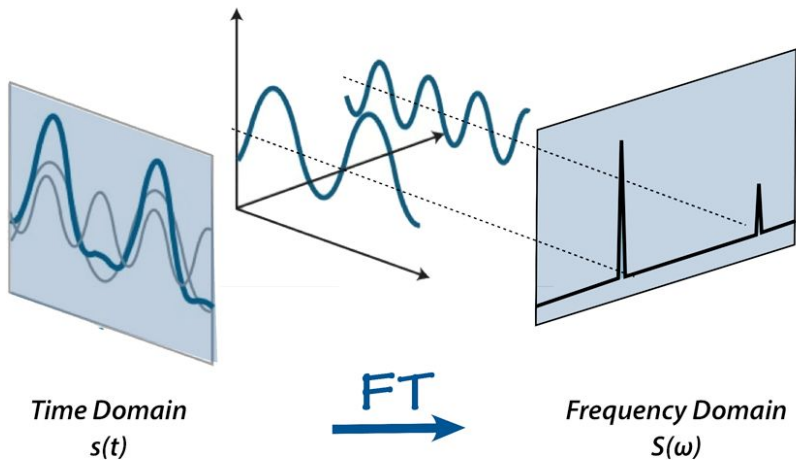


Minification

- When a single pixel contains several texels
- We have to somehow average their contribution
- Much harder than magnification: the the Nyquist frequency becomes closer and closer so we are getting a ton of aliasing



Fourier transform

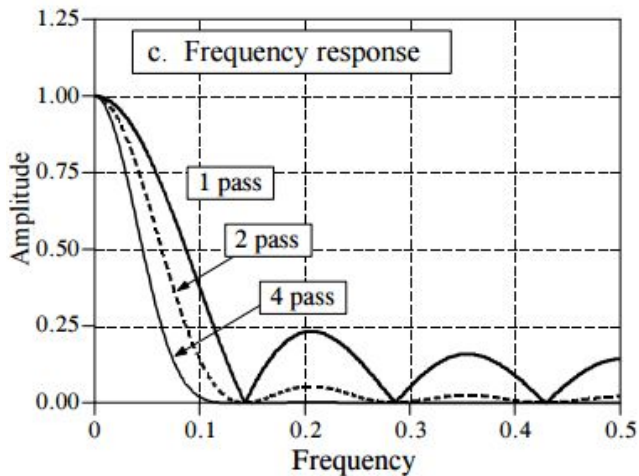


Minification

Theorem: If a function $x(t)$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced $1/(2B)$ seconds apart.

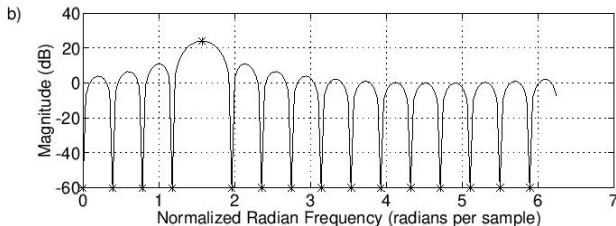
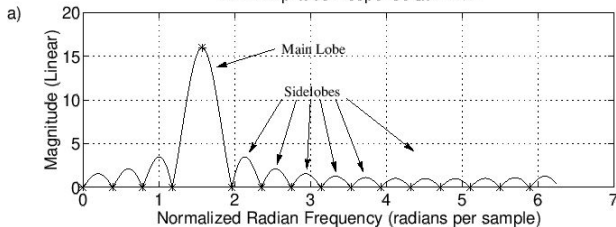
- Theoretically, if we can get rid of the high frequency details from the texture, we can avoid aliasing - we just have to replace 'seconds' by 'pixels'
- So any detail that 'changes' at least as fast as two pixels should be removed
- How do we remove high frequency details in 1D? Ideal low-pass filter!
 - Unfortunately, it's ideal in the sense as that it cannot be realized for finite signals
- Averaging is supposed to make things less high frequency, right? Maybe that will do?

Low pass filtering with averaging



Lost battle from the get go

DFT Amplitude Response at $k=N/4$



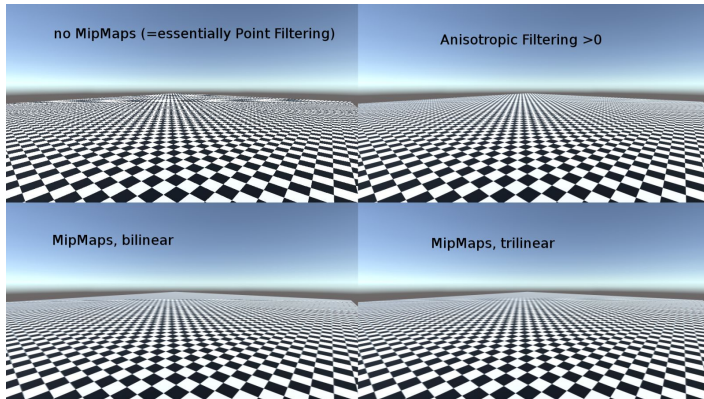
$$\begin{aligned} X(\omega_k) &\triangleq \langle x, s_k \rangle \triangleq \sum_{n=0}^{N-1} x(n) \overline{s_k(n)} \\ &= \sum_{n=0}^{N-1} e^{j\omega_x nT} e^{-j\omega_k nT} = \sum_{n=0}^{N-1} e^{j(\omega_x - \omega_k)nT} = \frac{1 - e^{j(\omega_x - \omega_k)NT}}{1 - e^{j(\omega_x - \omega_k)T}} \\ &= e^{j(\omega_x - \omega_k)(N-1)T/2} \frac{\sin[(\omega_x - \omega_k)NT/2]}{\sin[(\omega_x - \omega_k)T/2]}, \end{aligned}$$

https://ccrma.stanford.edu/~jos/mdft/Frequencies_Cracks.html

Low pass filtering

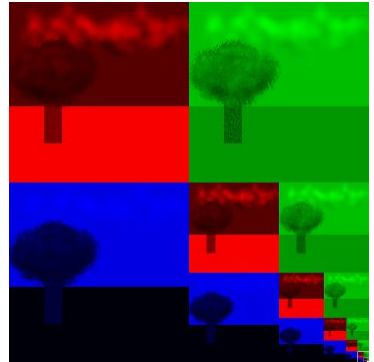
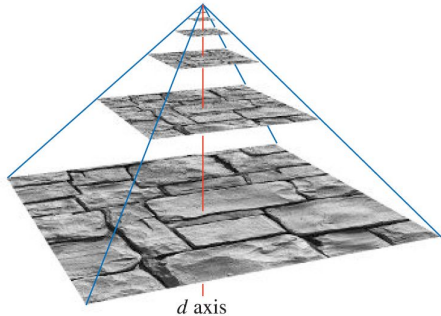
- In audio, there are established techniques to decompose a signal into sines + noise or even sines + noise + transients
- These parametric representations can be correctly (~ideally) filtered
- Not much luck for images
- In practice, many applications use averaging
- But even that is computationally infeasible if done in real-time

Mipmaps



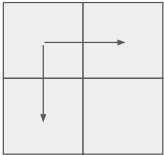
Mipmapping

- Multum in parvo = 'many things in a small place'
- Assume that minification happens equally along the axes and precompute the half, quarter, etc. resolution lowpass filtered images
- Can be done with only +33% storage

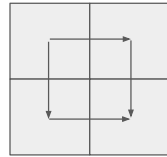


Mipmapping

- All you need during lookup is the ratio of the pixel and texel sizes in screen
- For this, we use the derivatives of the texture coordinates
- These are available because fragment shaders are grouped into quads, i.e. 2x2 fragments
- Actually, you can compute a forward differences approximation to all your variables using [dFdx and dFdy](#)



coarse



fine

Mipmapping

- We can use the the partial derivatives of the (u, v) texture coordinates w.r.t. the screen space X and Y axes to compute a level-of-detail value (LOD):

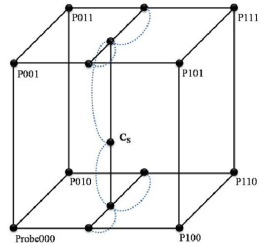
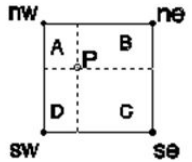
$$L = \log_2 \left(\max \left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2} \right) \right)$$

- Remember that averaging sub-par (it keeps too many high frequency things) so do experiment with other filters (Gaussian, Lanczos, etc.)
- Also, be careful about your gamma
- And not all data are equal: some things need different mipmapping...



Mipmapping

- Trilinear interpolation combines the result of the enclosing two bilinearly filtered mipmap faces' results
- Some bit image formats are slower in trilinear interpolation!
- Also, you cannot do automatic LOD selection under flow-control, you have to compute it beforehand (or manually) - as the fragments of your quad may diverge
- Still, no end-of-all solution: it tends to overblur



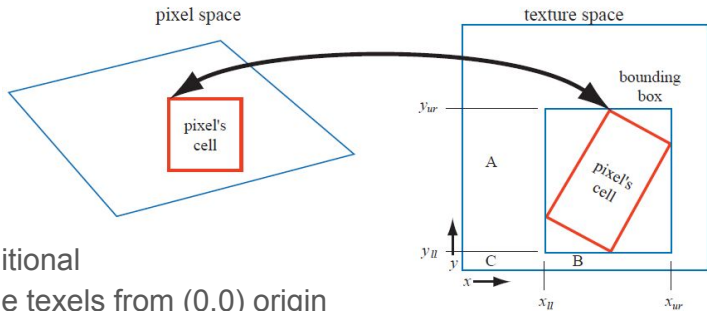
Alpha and mipmapping

- Averaging the alpha values doesn't make sense
- Instead, mipmap the signed distance measured from the cutoff level-set
- Or just mipmap the result of pass-ratio as the alpha value in a mip-level

$$c_k = \frac{1}{n_k} \sum_i (\alpha(k, i) > \alpha_t)$$



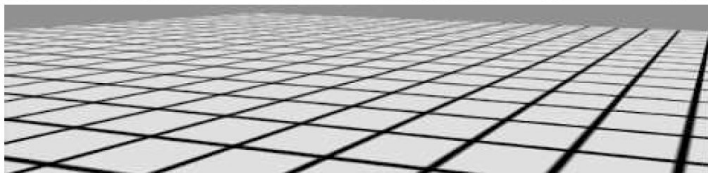
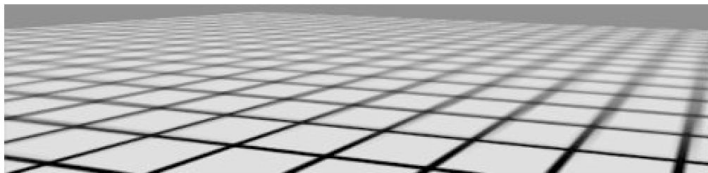
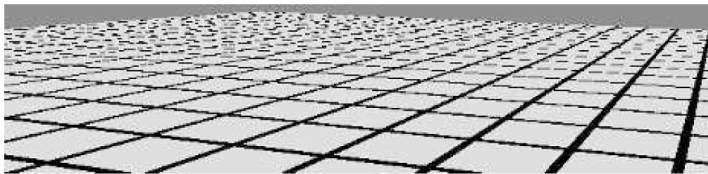
Summed-area table



- For each texel, store an additional color that is the sum of all the texels from (0,0) origin
- This extra data has to be in a higher bit format so that the sum isn't overflow
- Upon query, compute the texture space bounding box of the pixel rectangle and use the SAT to return an average
- Takes up at least twice as much memory and cannot handle general anisotropy

$$\mathbf{c} = \frac{\mathbf{s}[x_{ur}, y_{ur}] - \mathbf{s}[x_{ur}, y_{ll}] - \mathbf{s}[x_{ll}, y_{ur}] + \mathbf{s}[x_{ll}, y_{ll}]}{(x_{ur} - x_{ll})(y_{ur} - y_{ll})}$$

Nearest VS mipmapping VS SAT

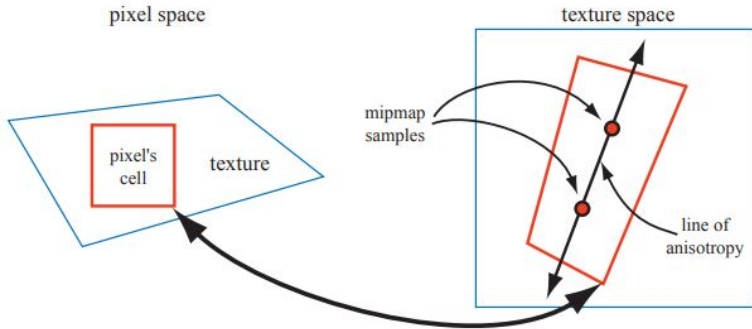


Anisotropic filtering



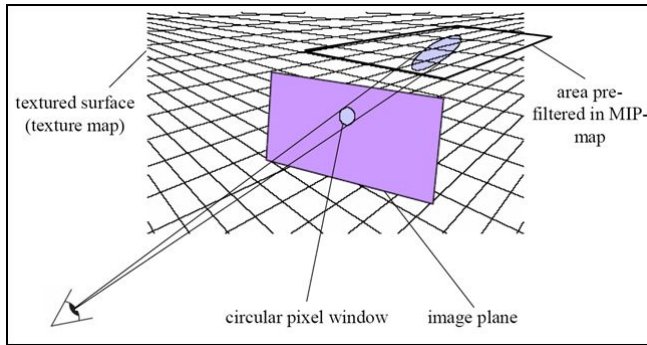
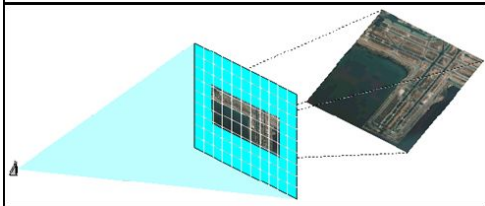
Anisotropic filtering

- Use the derivative min to determine the LOD level
- And fetch a couple of samples from this LOD to infer the filtered result



Anisotropic mipmapping

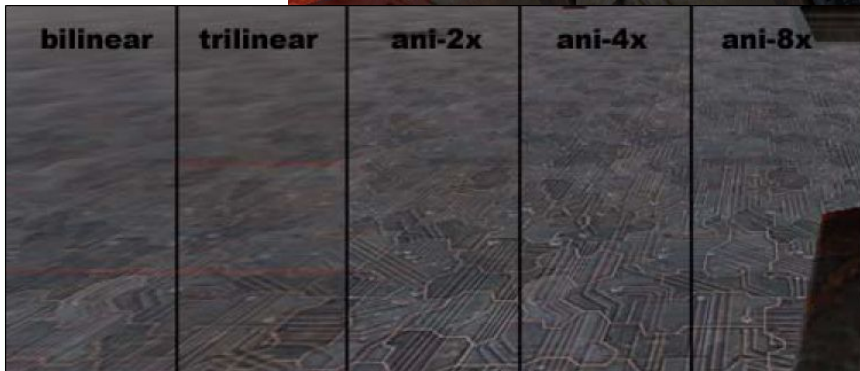
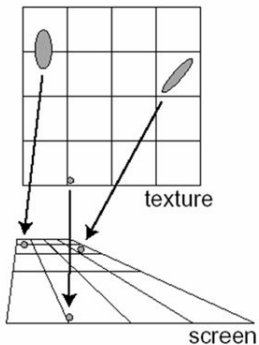
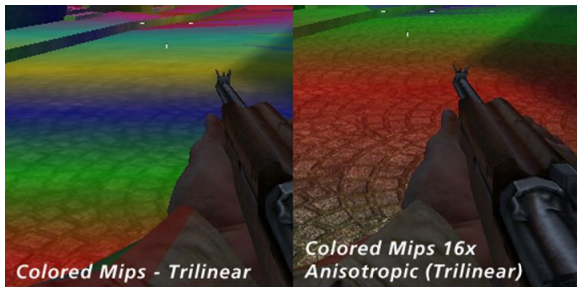
In reality, x and y distortion does not equal.



128×128	64×128	...
128×64	64×64	...
128×32	64×32	...

Anisotropic filtering

Calculating elongated average requires more samples.



Texture representation

Storage

- A **texture** is collection of **faces**
 - For example, many texture types have array variants [like in OpenGL](#)
 - The collection may refer to the MIP hierarchy too
 - And the combination of the above
- A **face** is a 1, 2, or 3 dimensional array of **texels**
- A **texel** consists of 1-4 channels
- A **texture format** determines how many bits you use to represent a texel and how you partition them to represent the channels
 - Keep in mind: hardware usually only supports multiplies of 4 bytes = 32 bits per texel formats natively, the rest may not work or come at a performance or storage cost
 - Larger formats also incur performance penalties for e.g. filtering

Color spaces

- For color textures, the most common is to store 3 or 4 components:
 - Color is represented as a triplet, often referred to as RGB
 - An alpha component can be used to encode transparency, referred to as A
- Usually, R8G8B8A8 or R11G11B10 are used as backbuffer formats
- For 8 bits, it is very important to use your available bits where human vision is sensitive; as such, storing the color primaries intensities is not the most efficient use of your bitwidth if you take humans into account:

Linear gradient

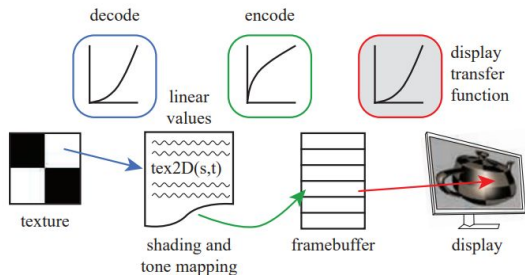


sRGB gradient



Color spaces - sRGB

- Display transfer function (DTF or EOTF): the mapping from digital values to emitted radiance
- It is done by your display



modern displays are still trying to roughly replicate the CRT response curve, which in turn is a pretty nice approximation to the HVS perceptually linear encoding

Color spaces - sRGB

- If we are encoding linear colors, i.e. a twice as large value should emit twice as much radiance, we need to take the inverse of the transfer function for each x channel of our linear RGB output too:

$$y = f_{\text{sRGB}}^{-1}(x) = \begin{cases} 1.055x^{1/2.4} - 0.055, & \text{where } x > 0.0031308, \\ 12.92x, & \text{where } x \leq 0.0031308, \end{cases}$$

- From display encoding to linear, we use

$$x = f_{\text{sRGB}}(y) = \begin{cases} \left(\frac{y + 0.055}{1.055} \right)^{2.4}, & \text{where } y > 0.04045, \\ \frac{y}{12.92}, & \text{where } y \leq 0.04045, \end{cases}$$

Color spaces - sRGB

- Usually, the above two are simplified as

$$y = f_{\text{display}}^{-1}(x) = x^{1/\gamma}$$

$$x = f_{\text{display}}(y) = y^\gamma.$$

- You don't have to do these manually all the times, there are cases when the GPU does the encoding upon reads and decoding upon writes - consult your API documentation on this
- If you forget encoding, you usually end up with a darker image

Color spaces

- Even if you are in linear space, there are several options
- RGB, especially with uneven bit-distributions can cause artifacts, hue shifts, etc. when you apply post-processing
- There are many other color spaces that try to separate the intensity information from the chroma (e.g HSV, HSL)
- One popular, but approximate one in real-time is [YCoCg](#):

$$\begin{bmatrix} Y \\ Co \\ Cg \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} Y \\ Co \\ Cg \end{bmatrix}$$

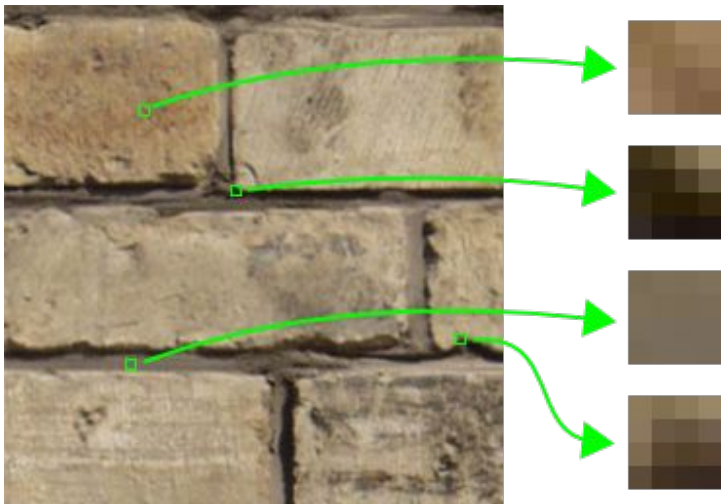
- Simplified LMS (long, medium, short):
(used by JPEG XL)

$$\begin{bmatrix} X \\ Y \\ B \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} L \\ M \\ S \end{bmatrix}$$

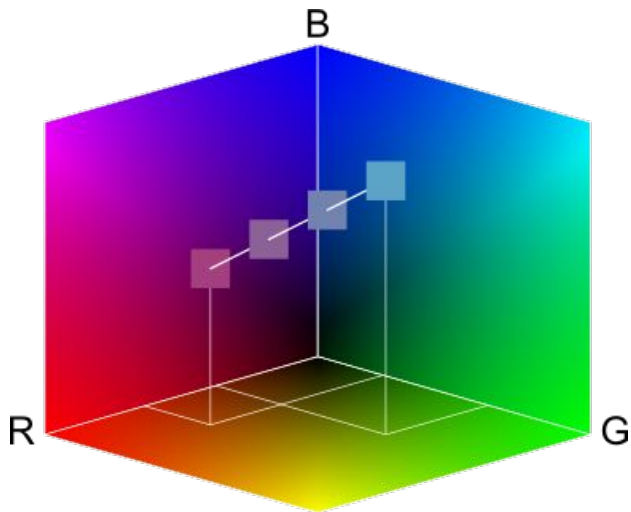
Color spaces

- Conversion between color spaces if often necessary
- You want to do your computations in a linear space
 - Not necessarily linear RGB though. For example in the case of linear filtering:
<http://staff.fh-hagenberg.at/burger/publications/pdf/aapr2010.pdf>
- In the presence of lower bit depth backbuffers, you want to output the final result such that you are not wasting bits where they don't matter
- A nice overview can be found at
<http://www.babelcolor.com/download/A%20review%20of%20RGB%20color%20spaces.pdf>

Block compression



Block compression



Block compression - BC1

- Take a 4x4 tile of pixels
- Store two colors to determine a line in a given color space
- And store 4x4 parameters for each pixel to store the closest color on the line to the original pixel color
- BC1:
 - $2 \times R5G6B5 = 32 \text{ bits} = 4 \text{ bytes}$ for the two endpoints
 - $16 \times 2 = 32 \text{ bits} = 4 \text{ bytes}$ for the $\{0, 1, 2, 3\}$ parameter values to represent the pixel colors

Uncompressed



BC1



Block compression: BC1

- The two endpoints can be of arbitrary order: use this redundancy to alpha!
- If $\text{asuint}(A) > \text{asuint}(B)$: as before
- Else: reduce the parameter range to 3 and use the fourth as a transparent black (RGBA = 0.0.rrrr)



Plain bilinear filtering



With alpha correction

BC6H

- Intended for lossy-compressing 16 bits float per channel HDR images
- It also uses partitioning: depending on partitioning mode, it can use up to two different color lines

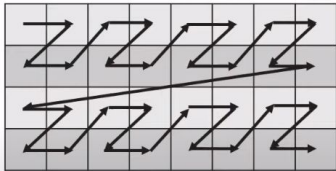
0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1,	0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,	0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1,	0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1,	0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,	0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,	0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1,	0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,	0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1,
0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,	0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,	0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1,	0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,	0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0,	0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0,
0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,	0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0,	0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0,	0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1,
0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,	0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0,	0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0,	0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0,
0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0,	0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,	0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0,	0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,

Source data	Minimum required data compression resolution	Format	Minimum feature level
Three-channel color with alpha channel	Three color channels (5 bits:6 bits:5 bits), with 0 or 1 bit(s) of alpha	BC1	Direct3D 9.1
Three-channel color with alpha channel	Three color channels (5 bits:6 bits:5 bits), with 4 bits of alpha	BC2	Direct3D 9.1
Three-channel color with alpha channel	Three color channels (5 bits:6 bits:5 bits) with 8 bits of alpha	BC3	Direct3D 9.1
One-channel color	One color channel (8 bits)	BC4	Direct3D 10
Two-channel color	Two color channels (8 bits:8 bits)	BC5	Direct3D 10
Three-channel high dynamic range (HDR) color	Three color channels (16 bits:16 bits:16 bits) in "half" floating point*	BC6H	Direct3D 11
Three-channel color, alpha channel optional	Three color channels (4 to 7 bits per channel) with 0 to 8 bits of alpha	BC7	Direct3D 11

Memory layout

- Textures are not necessarily stored row-major or column-major in memory due to how fragments usually address them, instead a Moreton z-ordering like layout is preferred

https://docs.microsoft.com/en-us/windows/win32/api/d3d12/ne-d3d12-d3d12_texture_layout



- However, you can specify that you want a linear layout, i.e. row-major

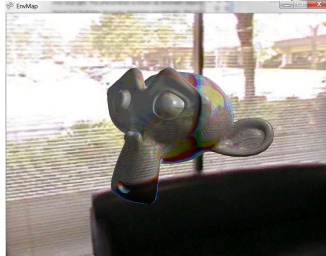
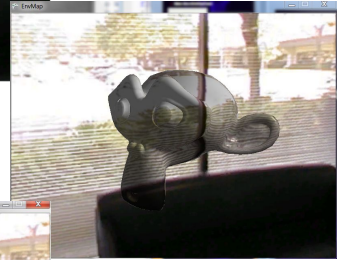
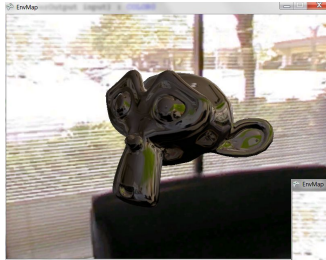
Texture mapping techniques

Environment mapping



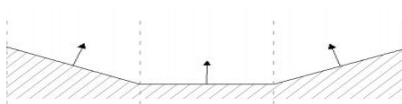
Environment mapping

- Supported effects:
 - Reflection & Refraction
 - Chromatic aberration
 - Fresnel effect
- Spherical texture has high distortion
⇒ Cube texture ⇒ Skybox
- Can also be a render target
⇒ Real-time reflection



Surface detail classification

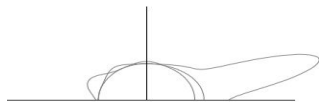
- **Macrogeometry:** features covering many pixels. Usually represented by API geometric primitives such as triangles
- **Mesogeomerty:** a couple of pixels-wide details. These are inefficient to explicitly represent as geometry (as that would result in rendering microgeometries, i.e. triangles that are only a few pixels large, which are a performance bottleneck) like wrinkles on skin or cloth
- **Microgeometry:** details that are smaller than a pixel. The various BRDF models try to represent these with various user parameters.



Macrogeometry

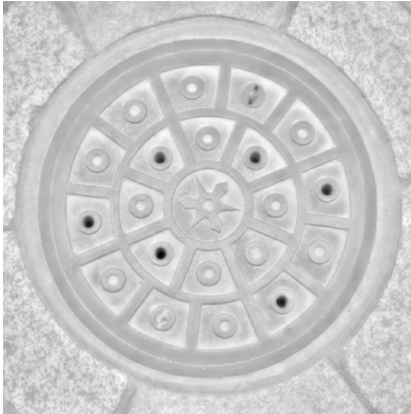


Macro + mesogeometry



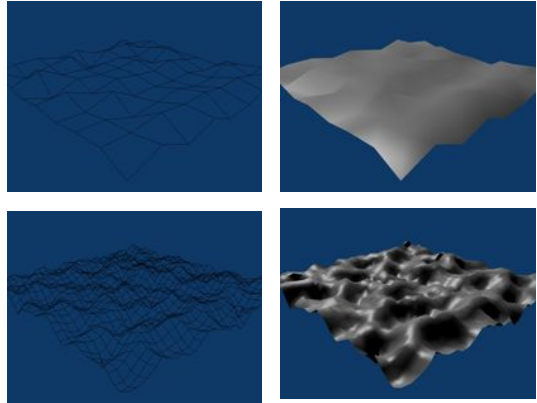
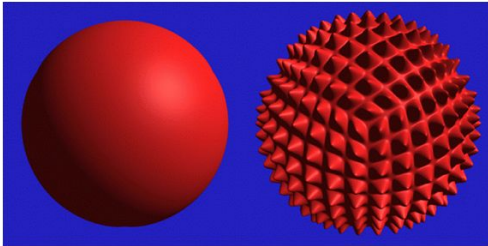
Microgeometry

Displacement mapping

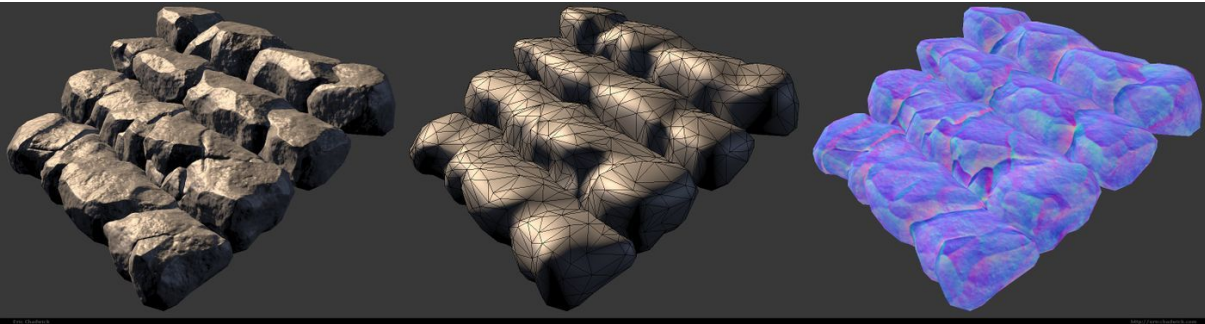


Displacement mapping

- Move each vertex towards normal vector by texture value
- Changes geometry
- Resolution depends on geometry resolution

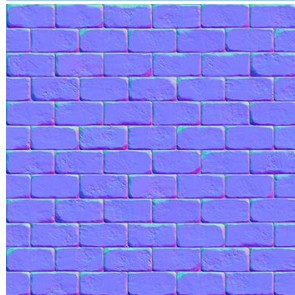
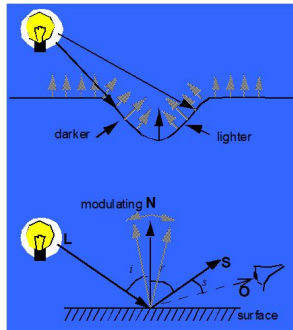


Normal mapping

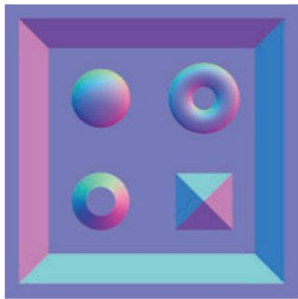


Normal map

- Problem:
 - Does not change geometry
 - Modulate normal vectors from texture
- RGB texture holds a normal vector at each texel
 - $\text{norm} = 2 * \text{texCol} - 1$
 - Normal vectors are unit length, values are between -1 and 1



Normal mapping



- These methods store the *shading normal* itself in the texture
- Nowadays, these normals are given in a tangent frame
 - At each vertex, store 3 linearly independent vectors that form 3D basis (may not be orthogonal)
 - Usually, store a tangent and bitangent vector pair that span the tangent plane (in the direction of u and v texture coordinates)
 - There are other efficient encodings, but these need not necessarily be interpolate-able

Normal mapping

Problem:

- From local normal \Rightarrow word normal ?
- Normal vector is not enough
- Tangent & bitangent defines transformation
- Tangent often can be exported

$$\begin{aligned} B &= N \times T \\ N &= T \times B \\ T &= B \times N \end{aligned} \quad \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$



Normal mapping

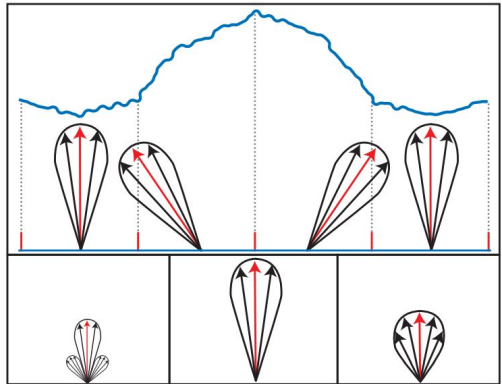
- Originally, normals were proposed to be stored in world space or object space
- However, that'd tie the normal map to a specific geometry or orientation
- The tangent frame formulation allows more trivial normal map reuse
- Nevertheless, there are other approaches, such as [Morten Mikkelson's surface gradient](#) based approach to bump mapping

- Implicitly define the fine normals by providing a heightfield as a variable-radius offset

$$\mathbf{d}^*(u, v) = \mathbf{d}(u, v) + h(u, v)\mathbf{m}(u, v) \quad \mathbf{m} = \frac{\partial_u \mathbf{d} \times \partial_v \mathbf{d}}{|\partial_u \mathbf{d} \times \partial_v \mathbf{d}|}$$

- The gradient of this offset surface is then used for the modification of the geometric normal
- Combination of several normal maps is more intuitive in this formulation because that's just the superposition of two heightfield displacements

Normal mipmapping



- For diffuse shading, averaging is not *that* bad since

$$\mathbf{l} \cdot \left(\frac{\sum_{j=1}^n \mathbf{n}_j}{n} \right) = \frac{\sum_{j=1}^n (\mathbf{l} \cdot \mathbf{n}_j)}{n}$$

holds, although imprecise since even Lambertian is nonlinear due to $(\mathbf{l} \cdot \mathbf{n}_j)_+$

- In general, however, we need to be smarter about filtering and mipmapping

Normal and normal distribution filtering

- It is more precise to think in terms of filtering normal distributions, not just individual normals (i.e. we need microscale information, such as the variance)
- One way is to take both the normal and roughness maps and generate a pair of normal and roughness mip values that correspond to a distribution that fits the entire region under the mip level's footprint
- Toksvig: the length of the averaged normal is inverse proportional to the width of the normal distribution. For Blinn-Phong NDF, modify roughness as

$$\alpha'_p = \frac{\|\bar{\mathbf{n}}\| \alpha_p}{\|\bar{\mathbf{n}}\| + \alpha_p (1 - \|\bar{\mathbf{n}}\|)}$$

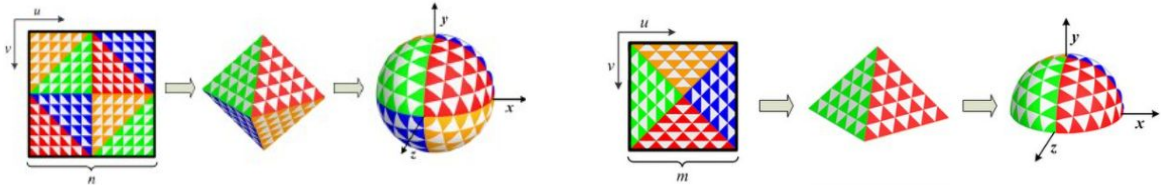
The diagram shows the equation for the modified roughness parameter α'_p . A callout box labeled "Averaged normal" points to the term $\|\bar{\mathbf{n}}\|$ in the numerator and denominator. Another callout box labeled "Original Blinn-Phong roughness" points to the term α_p in the numerator and denominator.

Normal and normal distribution filtering

- There are several advanced approaches to do this filtering (e.g. LEAN maps the covariance matrix of the normal distribution)
- For example in case of specular this is even more problematic and it is still an active area of research (see e.g. [Improved Geometric Specular Antialiasing](#))

Normal representation

- Usually, stored as tangent-space XY projections; after interpolation, the third coordinate of the normal is reconstructed as $z = \sqrt{x^2 + y^2}$
- This interpolates reasonably, but certain applications (such as deferred G buffers) don't need interpolation
- For these cases, there are more efficient encodings; a recent survey on unit vector representations: <http://jcgt.org/published/0003/02/01/>

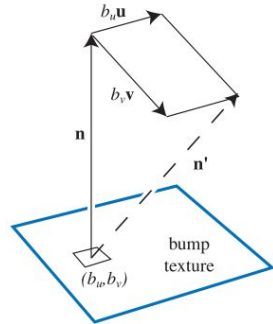


Bump mapping

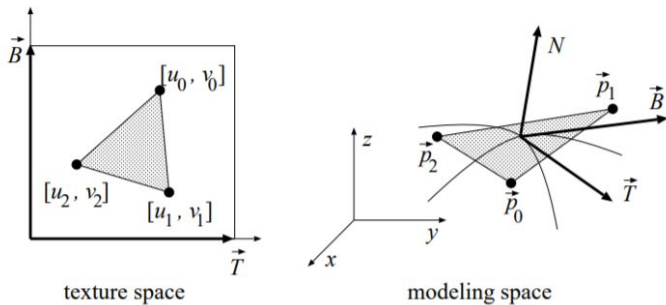


Bump mapping

- Blinn, 1978: use a texture to modify the *geometric normal* (the normal interpolated from the vertices) with a *shading normal* (from the texture)
- In Blinn's formulation, the texture stored an offset to the geometric normal
- This offset was given in a (\mathbf{u}, \mathbf{v}) basis
- Alternative formulation can be derived from taking a heightfield input
- Recall that many texture formats can only store unsigned values - so you will have to convert the range of b_u, b_v values to $[0, 1]$



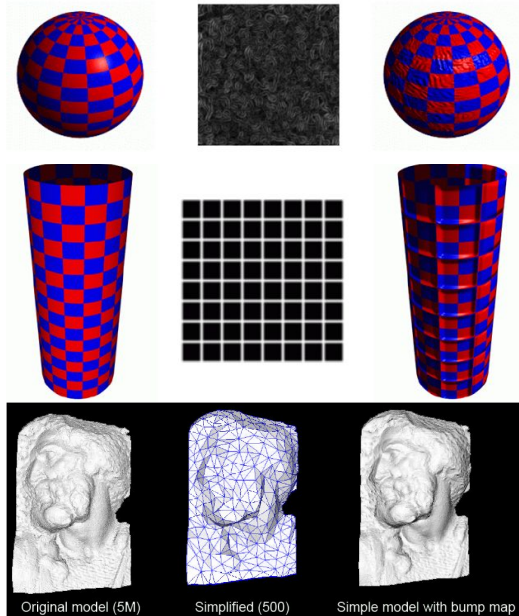
Tangent-space to world-space



$$\vec{p}(u, v) = \vec{p}_0 + [u, v, h] \cdot \begin{bmatrix} \vec{T} \\ \vec{B} \\ \vec{N} \end{bmatrix} = \vec{p}_0 + [u, v, h] \cdot \mathbf{M}$$

Bump map

- Same problem, modulate the normals
- The bump map can be created from high resolution model or displacement map
- Normals are modulated by the derivatives of the bumpmap

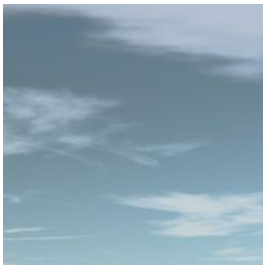
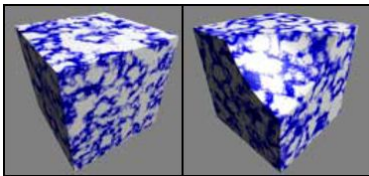


3D Textures

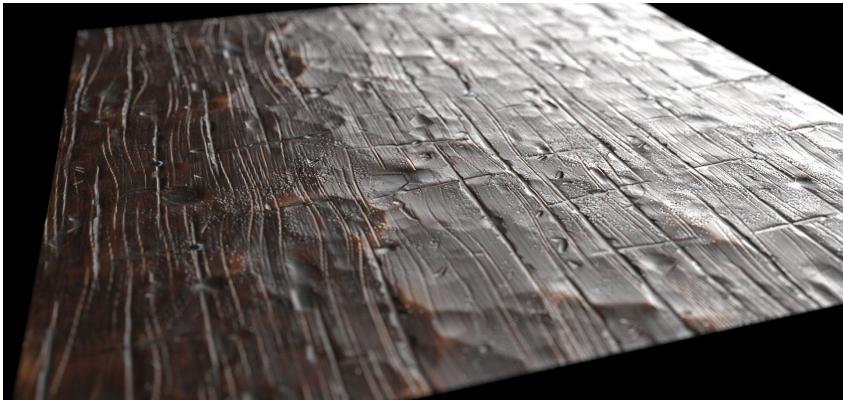


Volume textures

- texture mapping can be identity
- $256 \times 256 \times 256 = 64\text{Mbyte!}$
- Sometimes simulation result

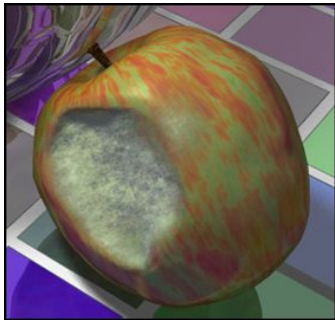


Procedural textures



Procedural texture

- Mathematical function
 - No repetition and easy to fit
 - Best for natural materials (stone, wood)
 - Can be zoomed in indefinitely.
- Only the function has to be stored
 - 3D texture in a few bytes!
 - Time can be a parameter
 - Can be slow to compute

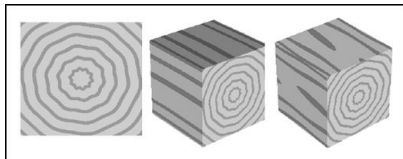
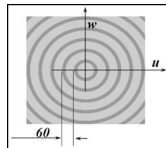


Creating procedural textures

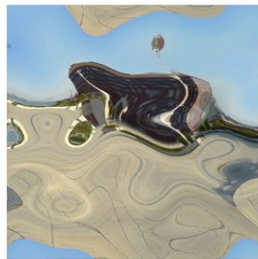
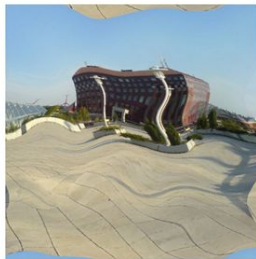
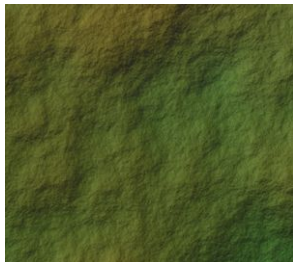
- Define mathematical structure
- Distort result
- Add noise

```
radius += 2*sin(a*tan(u/w) + v/b);
```

```
radius = sqrt(u*u + w*w);  
grain = radius%60;  
if (grain<40) return  
light_color;  
else return dark_color;
```



Noise can be used to color or to distort



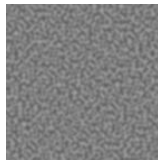
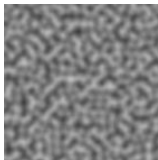
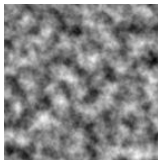
Noise

- Problems:

- Smooth enough to avoid aliasing
- Pseudorandom: same result twice

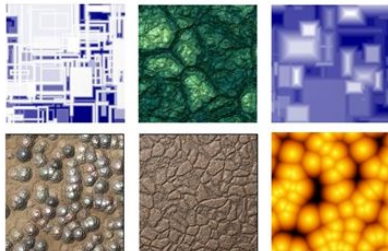
- Fractal noise:

- $\text{fractal_noise} = \text{noise}(x) + \frac{1}{2} \cdot \text{noise}(2 \cdot x) + \frac{1}{4} \cdot \text{noise}(4 \cdot x) + \dots$



- Cellular texturing

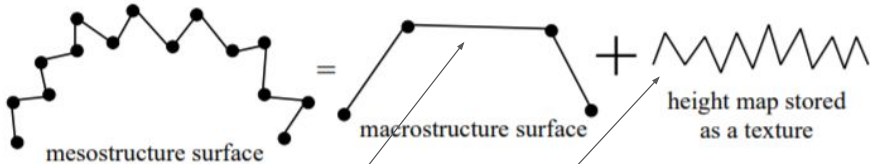
- Color is determined by the distance to some randomly scattered points
- Closest point \Rightarrow Voronoi (\Leftrightarrow Delaunay triangulation)



Bonus content:
Per-pixel displacement mapping

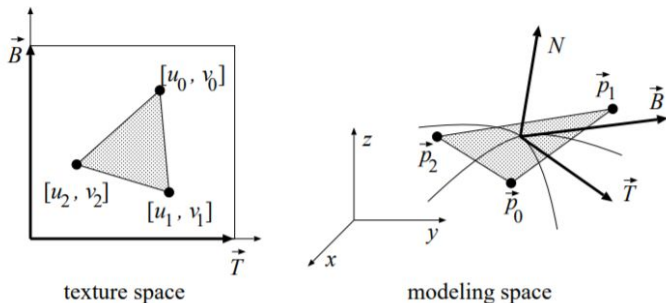
Literature

- Still interesting results
- A good survey to understand the basic (confusing) notations:
http://page.mi.fu-berlin.de/block/htw-lehre/wise2012_2013/bel_und_rend/skripte/szirmay2006.pdf



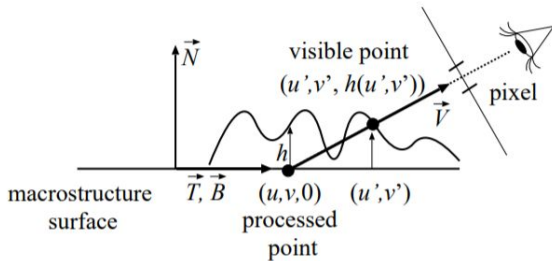
$$\vec{r}(u, v) = \vec{p}(u, v) + \vec{N}^0(u, v)h(u, v)$$

Tangent space to world space

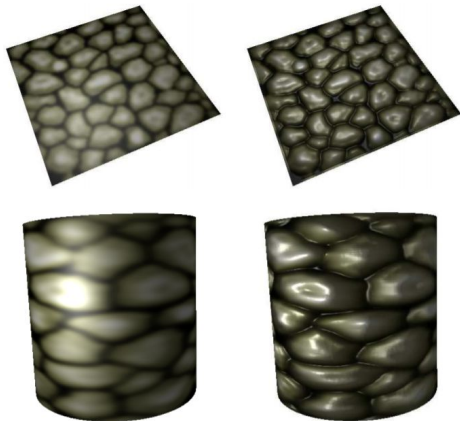


$$\vec{p}(u, v) = \vec{p}_0 + [u, v, h] \cdot \begin{bmatrix} \vec{T} \\ \vec{B} \\ \vec{N} \end{bmatrix} = \vec{p}_0 + [u, v, h] \cdot \mathbf{M}$$

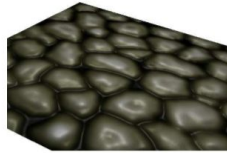
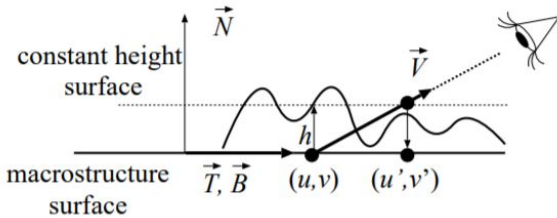
Motivation: Bump mapping



- It treats $(u, v, 0)$ as the intersection point
- Only changes the normal used for shading



Single-step parallax mapping



FPS = 695

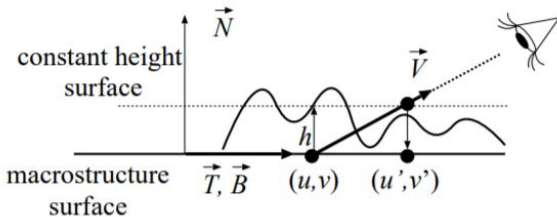


FPS = 675

Figure 10: Comparison of bump mapping (left) and parallax mapping (right) setting $BIAS = -0.06$ and $SCALE = 0.08$.

- It modifies the final (u, v) coordinates by solving $(u', v', h(u, v)) = (u, v, 0) + \vec{V}t$
- The solution is $(u', v') = (u, v) + h(u, v) \left(\frac{V_x}{V_z}, \frac{V_y}{V_z} \right)$

Single-step parallax mapping with offset limiting



FPS = 675

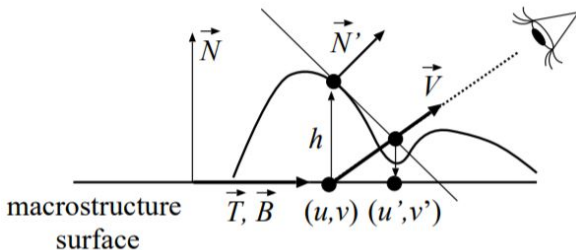


FPS = 680

Figure 12: Comparison of parallax mapping (left) and parallax mapping with offset limiting (right) setting $BIAS = -0.14$ and $SCALE = 0.16$.

- It modifies the final (u, v) coordinates by solving $(u', v', h(u, v)) = (u, v, 0) + \vec{V}t$
- The solution is $(u', v') = (u, v) + h(u, v) \left(\frac{V_x}{V_z}, \frac{V_y}{V_z} \right)$
- In practice, it's best to limit the offset which in this case is even better: $(u', v') = (u, v) + h(u, v) \left(V_x, V_y \right)$

Single-step parallax with slope



FPS = 675



FPS = 680

Figure 14: Comparison of parallax mapping with offset limiting (left) and parallax mapping with slope information (right) using $BIAS = -0.04$ and $SCALE = 0.12$.

- Move the ray-heightfield intersection estimate to the intersection between the plane at (u, v, h) and normal N' and the ray

- From $\vec{N}' \cdot ((u, v, 0) + \vec{V}t) = \vec{N}' \cdot (u, v, h)$ $(u', v') = (u, v) + h \frac{N'_z}{(\vec{N}' \cdot \vec{V})} (V_x, V_y)$

the unbounded and limited solutions are

$$(u', v') \approx (u, v) + hN'_z(V_x, V_y)$$

Single-step parallax mapping

- The above methods tried to find the intersection with a single step
- It makes sense to try to find a solution via iteration
- Note that at every iteration, you have to query potentially multiple textures



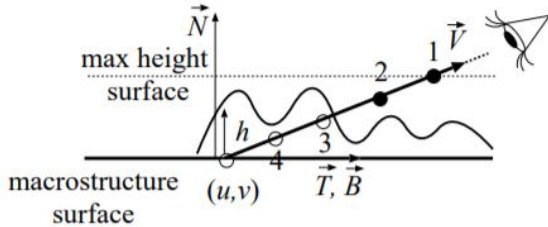
FPS = 600



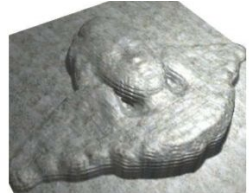
FPS = 570

Figure 15: Comparison of parallax mapping with slope (left) and iterative parallax mapping (right) setting $BIAS = -0.04$, $SCALE = 0.12$, and $PAR_ITER = 2$.

Raymarch parallax map



FPS = 410
LIN_ITER = 4



FPS = 340
LIN_ITER = 8

Figure 20: Linear search. Note the stair-stepping artifacts.

- Use constant sized steps
- Stop when below hightfield

Binary search in parallax map

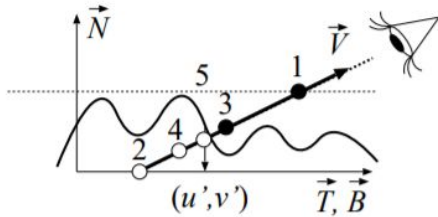
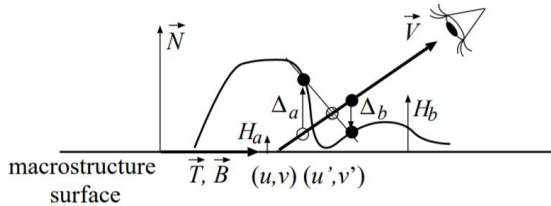


Figure 17: Binary search using 5 iteration steps. The rendering speed is 455 FPS.

- You have no guarantee if you will find the first intersection
- But you'll converge to one, eventually

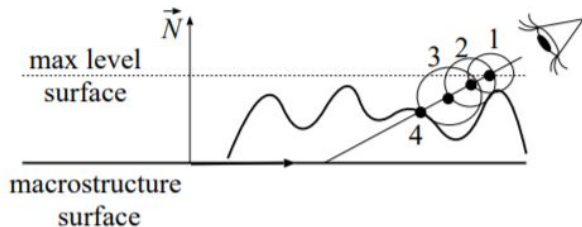
Secant method in parallax map



- The new guess at every iteration is

$$H_{new} = H_b + (H_a - H_b) \frac{\Delta_a}{\Delta_a - \Delta_b}$$

Sphere tracing a parallax map



depths = 32
FPS = 460

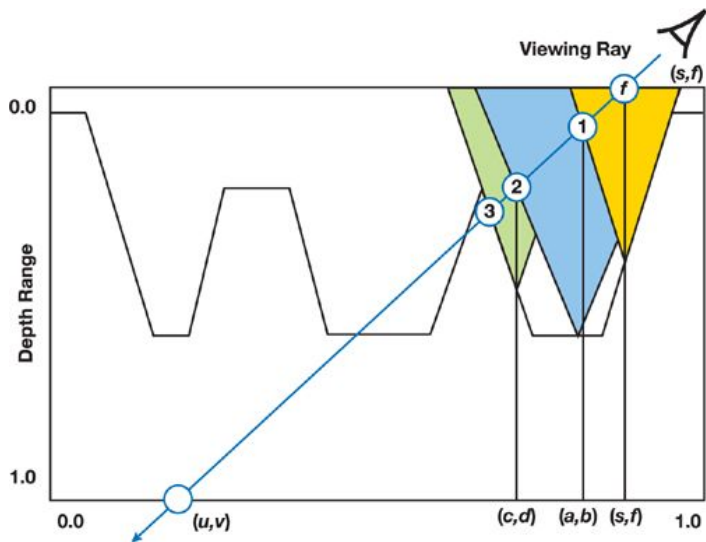


depths = 64
FPS = 460

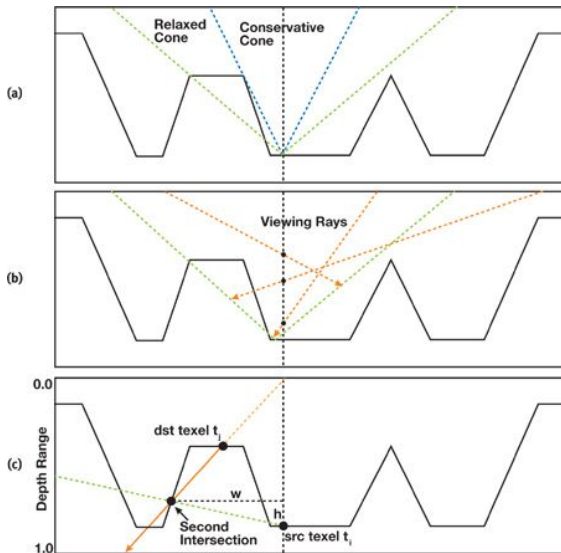
Figure 23: Sphere tracing results with different distance field texture resolutions. The left image was rendered with a 3D texture of 32 depth layers, while the right image with 64 depth layers.

- You need a heightfield to signed distance field conversion
- Otherwise, use your favorite sphere tracing algorithm

Parallax cone map



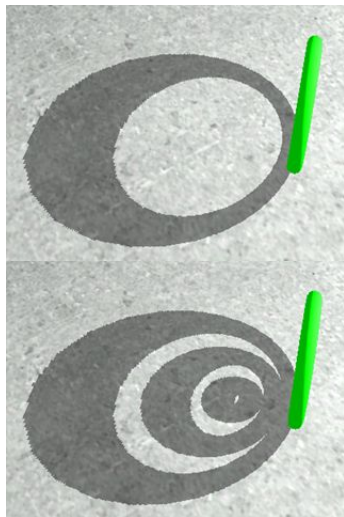
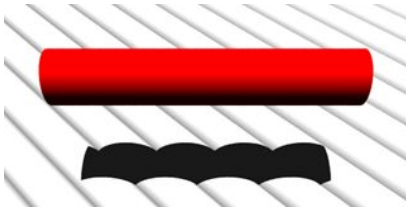
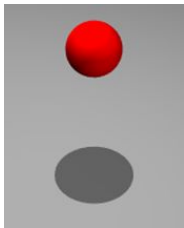
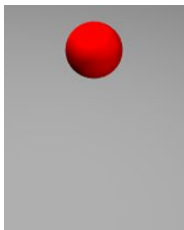
Relaxed parallax cone map



Real-time shadows

Csaba Bálint

Motivation



Real-time shadow algorithms

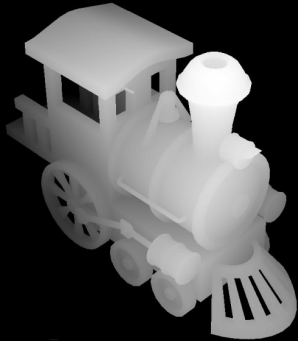
- **Shadow Map**
 - Fixed precision can cause blocky artifacts
 - Cascade shadows
- Shadow Volume
 - Precise, but computationally demanding
 - zpass and zfail
- Projected planar shadows: works best with flat surfaces
- Lightmaps: the scene must be static
- New: ray tracing: soft shadows, very demanding

Shadow Map



Idea

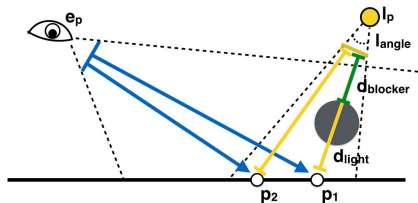
- Lance Williams - 1978
- Pixar's RenderMan, Toy Story
- **Idea: render from the light source**
 - Independent of scene geometry (apart from rendering)
 - May have sampling artifacts
- Two steps:
 1. Calculate distances to light \Rightarrow **Shadow map**
 2. Render final image from camera \Rightarrow **Shade**



Shadow Mapping

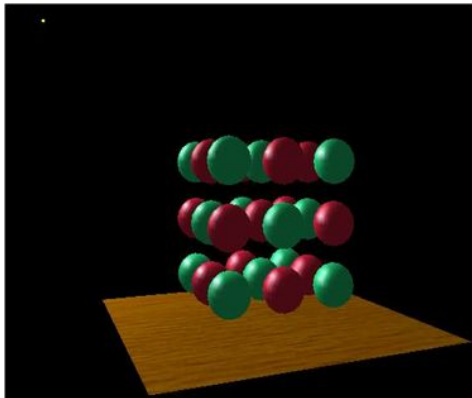
Algorithm

1. Calculate distances to light
 - a. Render scene from light source
 - b. Render depth buffer to a texture = **shadow map**
 - c. Calculate camera view to light view transformation
 - d. No need to shade, or look up textures \Rightarrow Fast
2. Render final image from camera
 - a. Transform pixel position to light coordinate system
 - b. Compare measured distance to light to value in shadow map
 - c. Equal values mean surface is lit
 - d. If the shadow map value is less, the pixel is in shadow
 - e. **Shade** accordingly

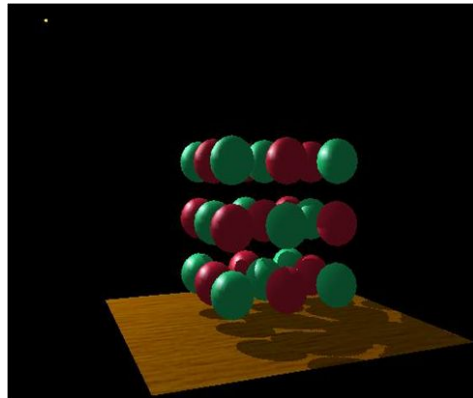


$$\text{shadow} = d_{\text{light}} < d_{\text{blocker}} + \text{bias}$$

Example: Goal

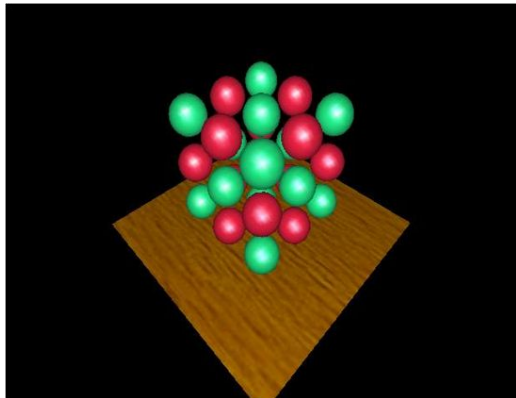


No shadow

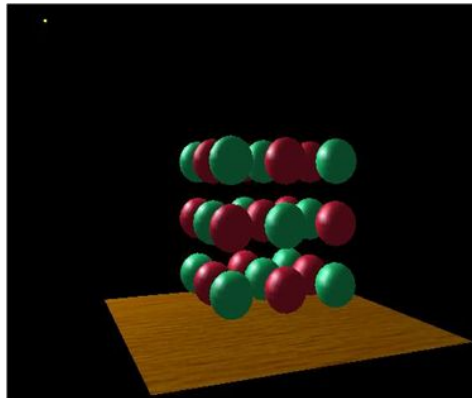


Real time shadow

Example: Two views

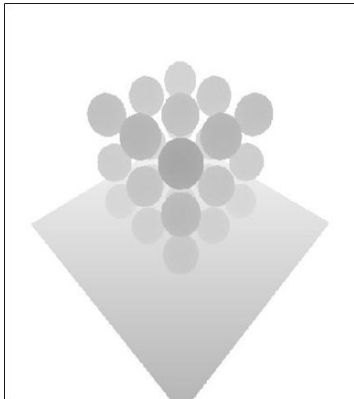


Light's view

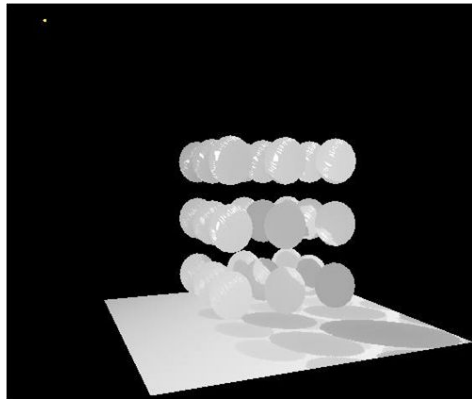


Camera's view

Example: Shadow Map

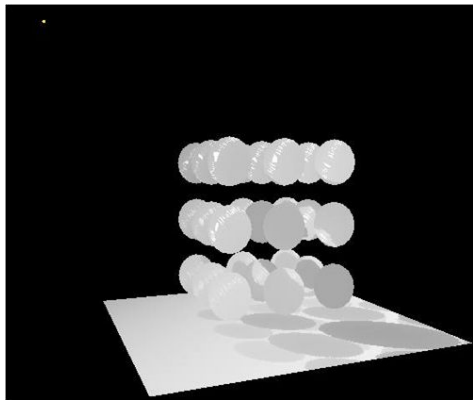


Light's view
Shadow map

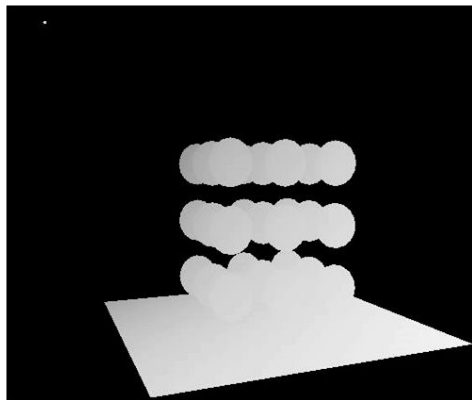


Camera's view
Shadow map

Example: Compare distances

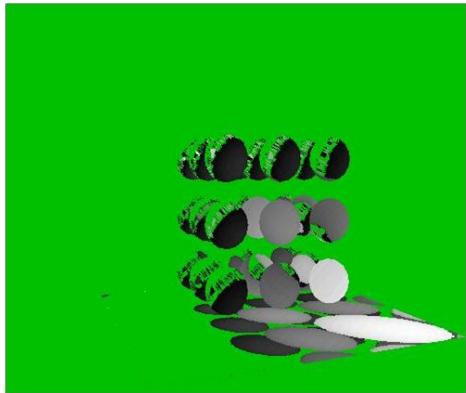


Camera's view
Shadow map

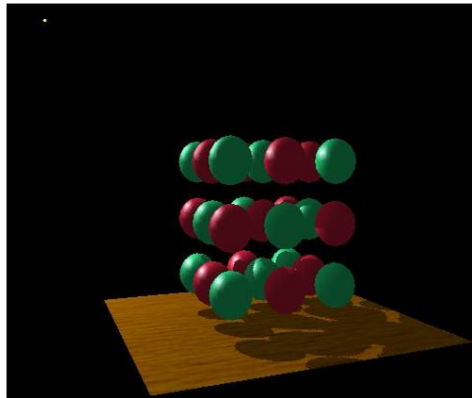


Camera's view
Calculated light distance

Example: Shadowing

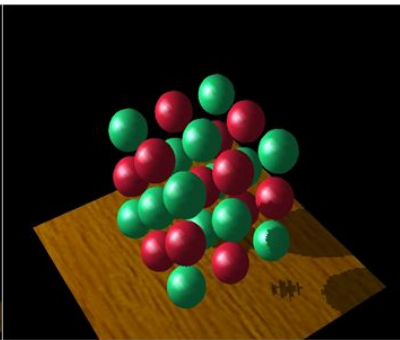
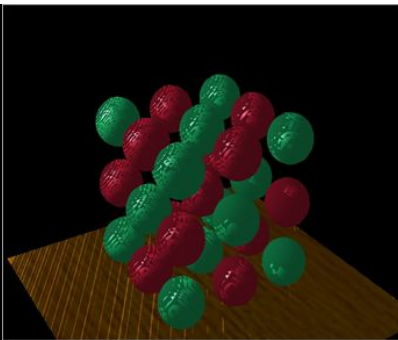
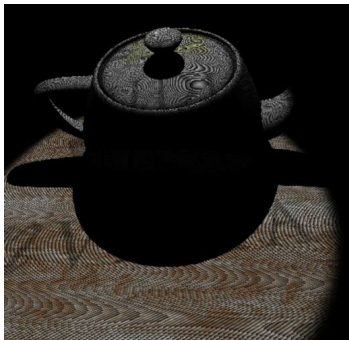


Camera's view
Difference



Camera's view
Shaded

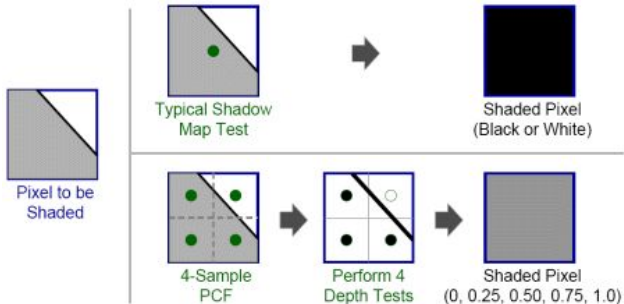
Sampling



Percentage-Closer Filtering (PCF)

- Almost parallel light rays \Rightarrow Extreme aliasing
- OpenGL shadow samplers (`sampler2DShadow`)
- `texture()` returns comparison result in $[0, 1]$
- LINEAR filtering is 4x PCF
- 16x PCF
- Sparse lookup kernes

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



Soft shadows with shadow maps

- PCF \neq soft shadows \Leftarrow area light
- Shadow map solutions must cheat
- Usually by varying the PCF kernel size as a function of the ratio between the distance to light and distance to occluder in the surrounding shadow map texels
- Shadow map is often processed and compressed



Omni light

- Render into a cubemap
 - ⇒ Multi-layer FBO attachment!
- Geometry shader!
 - ⇒ Single pass!
- **gl_Layer** : specifies FBO layer
 - Cubemap side

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices=18) out;

uniform mat4 shadowVPs[6];
out vec4 FragPos;

void main()
{
    for(int face = 0; face < 6; ++face)
    {
        gl_Layer = face;
        for(int i = 0; i < 3; ++i)
        {
            FragPos = gl_in[i].gl_Position;
            gl_Position = shadowVPs[face] *
FragPos;

            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Shadow Map LOD problems

- Cover the view-frustum with the SM
- Remap the domain in geom. shader
- Split up the SM into multiple parts

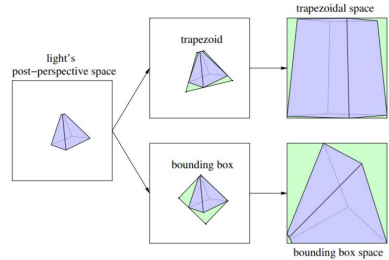


Figure 3: An example of the trapezoidal approximation (middle, top) and the smallest bounding box approximation (middle, bottom) of the eye's frustum as seen from the light (left). The wastage in the shadow map generated by the former is much smaller than that by the latter in this case (right).

Cascade shadows

- Calculate shadow map towards camera viewing frustum
- Subdivide frustum into multiple segments

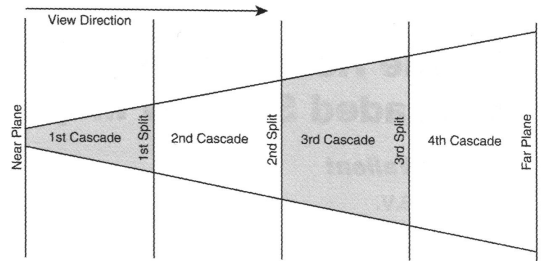
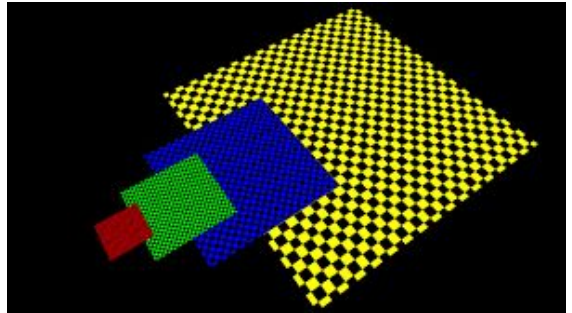
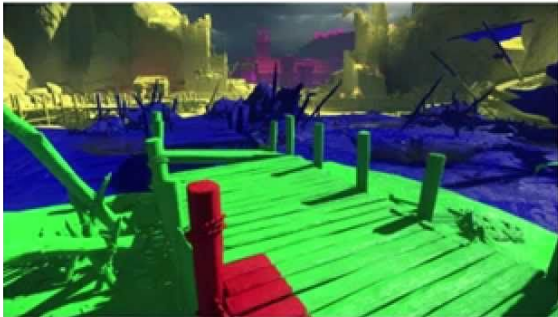


FIGURE 4.1.1 2D visualization of view frustum split (uniformly) into separate cascade frustums.



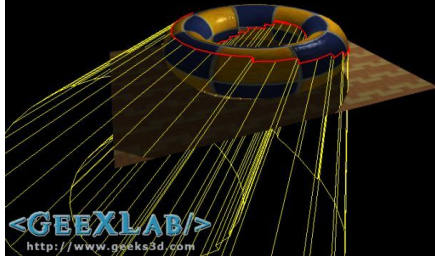
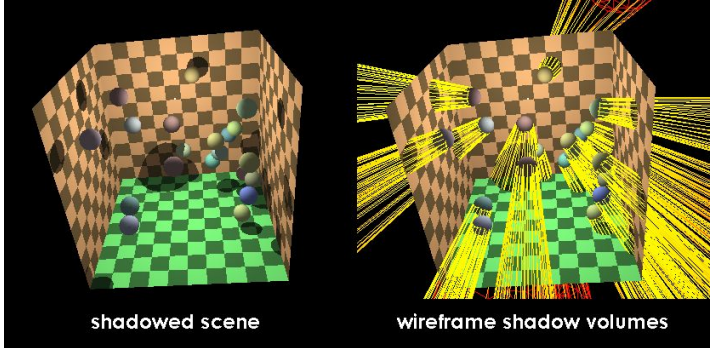
Shadow Volume



Shadow volumes

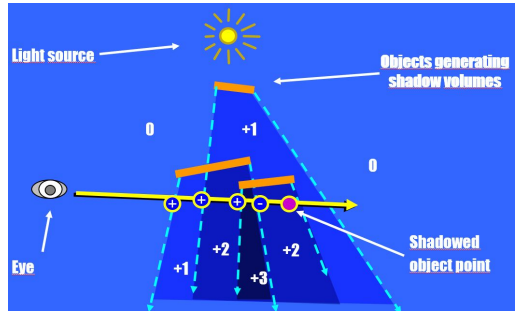
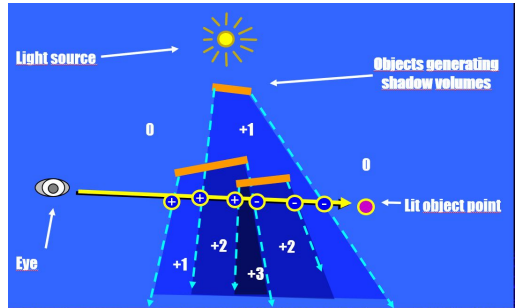
Franklin C. Crown

1. Identify contours of objects
2. Extruding the contours define shadow volumes
3. A surface point is in shadow when it is inside a shadow volume



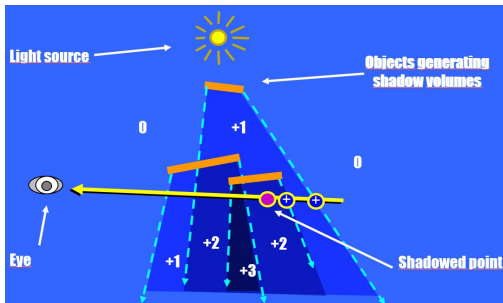
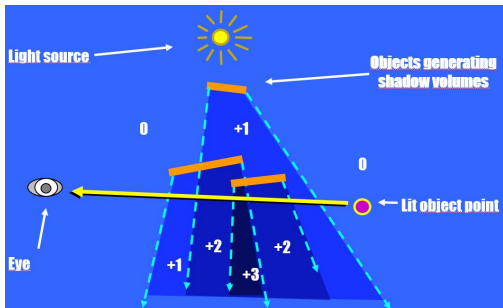
Algorithm

1. Render the shadow volumes
 - a. During rendering we keep track of the number of the shadow volumes entered and exited between camera eye and object point
 - b. Shadow volume faces facing the camera increase the counter by one
 - c. Back-faces decrease the (**stencil**) counter
2. If the stencil buffer is positive, the object in question is in shadow
3. If it is zero, it is lit



Problems & Solutions

1. Camera might be in shadow
 - a. **Solution: Start from infinity!** \Rightarrow zfail
2. Shadow volume may not be bounded
 - a. **Enclose the volumes!**
 - b. Volume bound can be at infinity
3. Stencil buffer updates:
 - a. First, render front-facing triangles of the shadow volumes, add values to stencil buffer
 - b. Second, render back-facing triangles, decrease stencil buffer



Calculating contours

Several algorithms exist. For example, for every triangle:

1. If triangle is back-facing the light source skip it
2. Add edges of the triangle to a list (vertex pairs)
3. If the list contained the edge, remove both the new and the old
4. Process next triangle

Pros and cons

Advantages

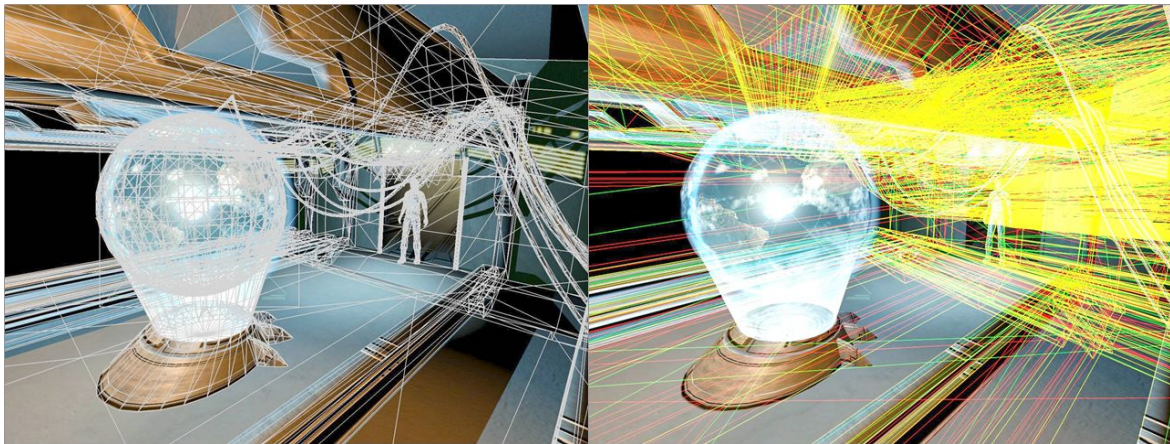
- Works with omnidirectional lights
- Lights are precise
- Rendering is GPU accelerated

Disadvantages

- Performance heavily depends on scene complexity
- Contour calculation is slow



Example from Abducted

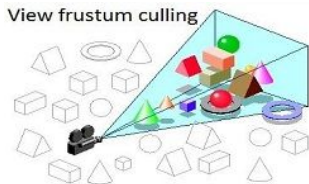


Spatial Data Structures

Csaba Bálint

Visibility Problem

- Classical methods
 - *Backface culling, Frustum culling*
 - **Painter, Z-Buffer, Warnock**
 - **BVH: Bounding Volume Hierarchy**
 - **Grids, Quadtree, Octree, KD-tree, BSP-tree**
- Modern methods
 - Cells and portals (portal culling)
 - PVS (Potentially visible set), Virtual covering objects
 - Hierarchical Z-buffer/occlusion maps (similar to Warnock)
- [More in this article](#)



Motivation

- Reduce redundant operations
- **Idea:** Do not draw non-visible objects
- Solutions in object-space:
 - **Backface culling:** throw away back-facing triangles ($\sim 2x$)
 - **Frustum culling:** only draw objects that are in the camera cone ($\sim 8x$)
- Solutions in screen-space:
 - **Z-buffer:** decide per pixel if the fragment is visible
 - **Early-Z:** fragment shader does not have to run if fragment will fail depth test

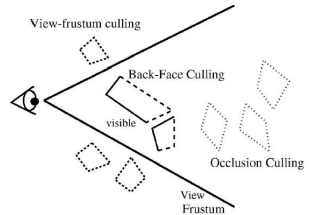


Fig. 1. Three types of visibility culling techniques: 1) View-Frustum Culling, 2) back-face culling, and 3) occlusion culling.

Why do we need more algorithms?

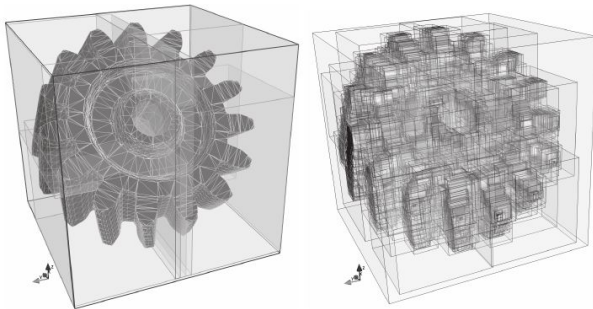
Runtime is still linear measured in the number of primitives in the scene.

Classification of methods

Algorithms can be classified based on efficiency and accuracy

- **Exact methods**
 - Classifies visible exactly those that are
- **Conservative methods**
 - May classify non-visible objects as visible. Usually good results.
- **Aggressive methods**
 - Visible objects might disappear, results in artifacts.
 - Useful when conservative methods are too slow and the error is acceptable.
- **Approximate methods**
 - Imprecise classification of both visible and invisible objects.
 - May be useful when geometry cannot be preprocessed

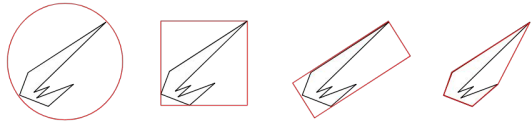
Bounding Volumes Hierarchy



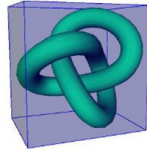
Bounding volumes

- Sphere
- **AABB: Axis Aligned Bounding Box**
- OBB : Oriented Bounding Box
- K-DOP: Discrete Oriented Polytopes
- Convex Hull

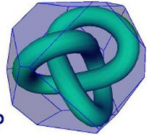
If a ray (or any connected set) does not intersect the bounding volume, it neither intersects its interior.



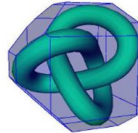
6-DOP
(AABB)



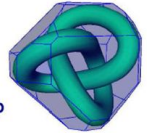
14-DOP



18-DOP



26-DOP

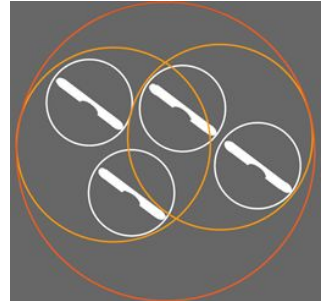
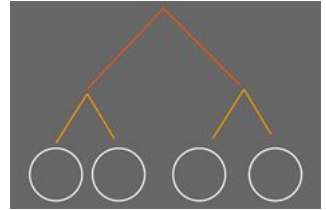


Bounding Volume Hierarchy

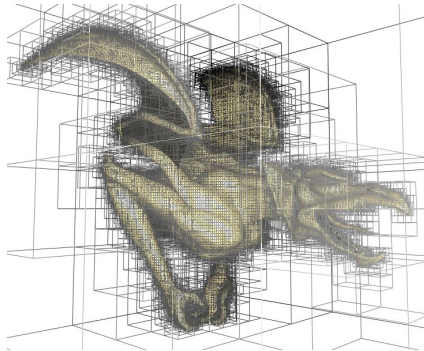
- Create a hierarchy of the bounding volumes
- If a BV is not intersected do not check its children
- Significantly decreases intersection tests
- Requires preprocessing and updates
- Conservative optimization

Use cases:

- Raytracing: huge speedup!
- Frustum culling
- Collision detection, ragged dolls

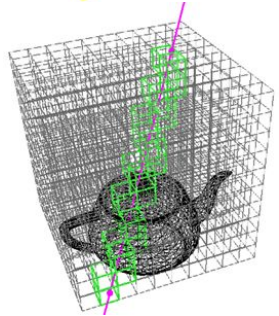
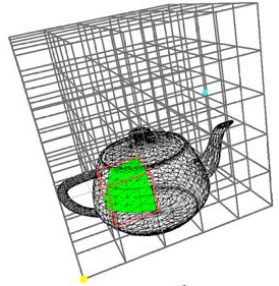


Space Partitioning I



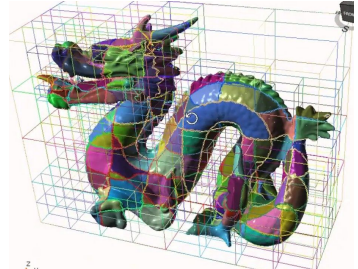
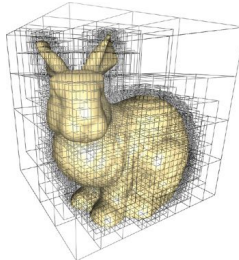
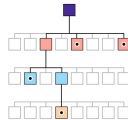
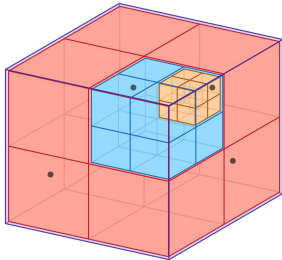
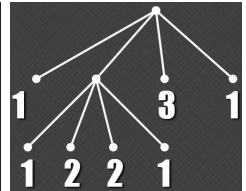
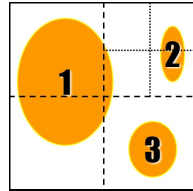
Uniform Grid

- Pros
 - Each cell holds the list of the objects within
 - Trivial neighbour lookup
 - Easy to implement
 - Ray-grid intersection is trivial:
“Voxelize” line \Rightarrow 3D Bresenham algorithm
- Cons
 - Subdivision is independent of scene geometry
 - May be slow or use too much memory



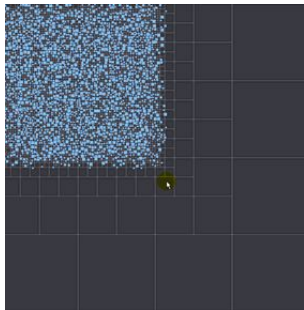
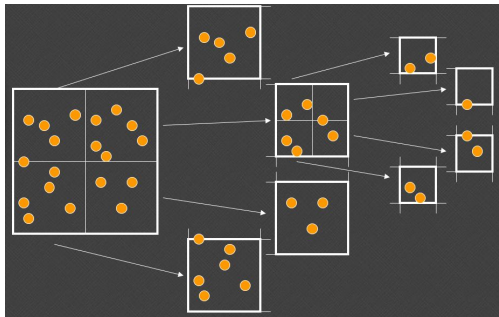
Quadtree / Octree

- Root node is the AABB of the whole scene
- If there are more objects in a node than a predefined number, then subdivide cell to 8 / 4 equal cells.
- Recursively continue the previous step until it stops or a maximum depth is reached



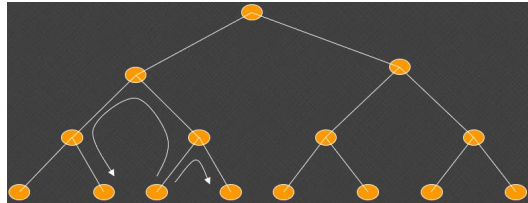
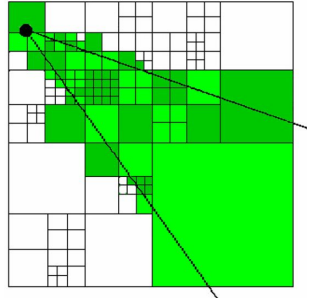
Tree construction

- A node contains:
 - Pointers to children
 - Pointer to parent: useful for neighbour lookup
 - Extent of the cell \Rightarrow Can be calculated from the tree
 - List of objects in the cell
- Recursive “buildNode” algorithm:
 - Input is the node and the list of objects that belong to it
 - If there aren't enough objects, set node to leaf and return
 - Distribute objects among created nodes
 - Call “buildNode” for each

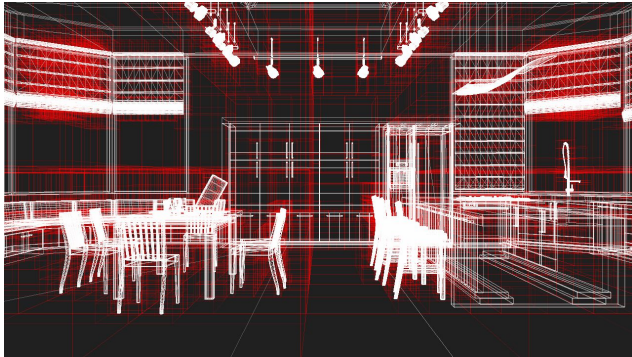


Tree navigation

- What happens if an object is in multiple cells?
 - Only pointer duplication \Rightarrow Memory efficient
 - Intersection tests can happen multiple times
 - *Solution*: set flag to object, eg. frame number
- “A” is neighbour of cell “B” if
 - They have a common side-plane
 - Cell “A” is inside a cell that’s the same size of “B” and is touching it, or vice versa.
 - “A” nor “B” has any children with the above properties
- Neighbour algorithm
 - Similar to binary tree \rightarrow

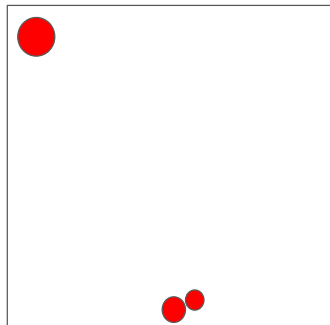
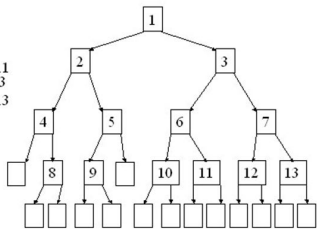
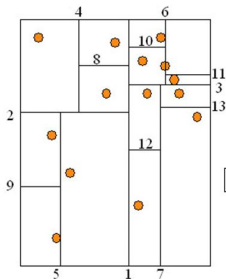


Space Partitioning II



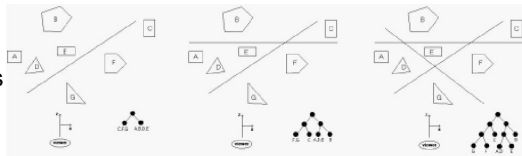
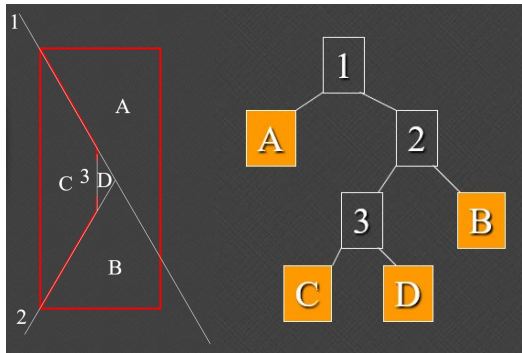
KD-tree

- Quadtree can become unbalanced!
- Each node represents a subdivision
 - A node has 2 children AABB-s
 - subdivided along the longest dimension
 - storing the position of that cut-plane.
 - Also store AABB positions, stored objects, parent pointer.
- Neighborhood lookup is harder
- During KD-tree construction, where to draw the new plane?
 - Cut along median point
 - Equalize surface area in cells



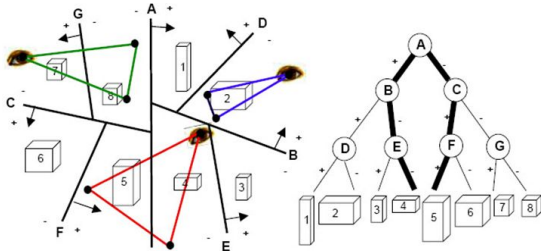
BSP-tree

- Similar to KD-tree
 - KD-tree is an axis align BSP-tree
- Arbitrary subdivision planes
 - All cells are convex
- Choosing cut-plane goals:
 - Minimize cell count and divided objects
 - Cut-planes should coincide with polygons
 - Heuristics:
 - choose cut-planes from scene polygons
 - start with the largest polygon
 - random choice is not bad
- Widely used algorithm!



BSP-tree Ordering

- Traverse the tree from a given point of view
 - Objects on “this” side appear in front of those on the “other” side of the cut-plane
- Back-to-front ordering
 - May not need Z-buffer!
 - Perfect for rendering transparent objects!
- Front-to-back ordering
 - Fewer Z-buffer overrides \Rightarrow Early-Z!

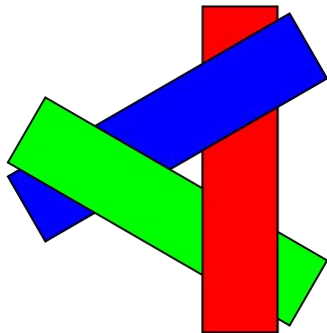


Other Methods



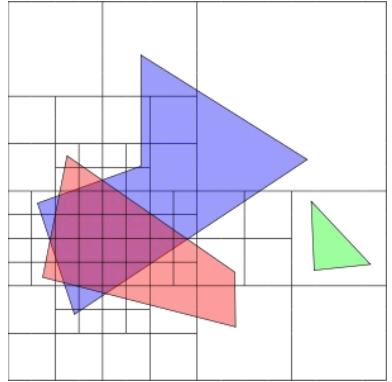
Painter algorithm - Newell's algorithm

- Front-to-back drawing. Can lead to overlaps:
- Need to cut: Test overlap
 - If the AABB of P and Q polygons overlap
 - Test for Z overlap from the sorting list
 - Test minmax overlap in X and Y directions
 - Test the ordering of the planes
 - All vertices of P lie deeper than Q
 - All vertices of Q lie closer than P
 - Finally, test if the rasterization of P and Q overlap
- If they overlap, swap them and try again.
- If a pair overlaps again
 - Cut them along the intersection
 - Put both back into the sorting list
- Continue until all pairs pass the test



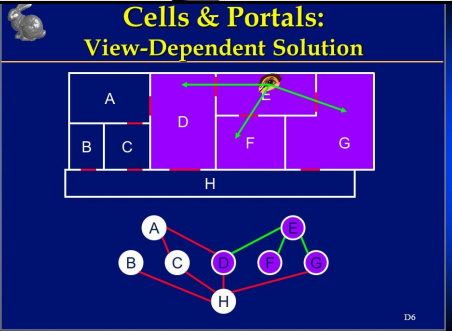
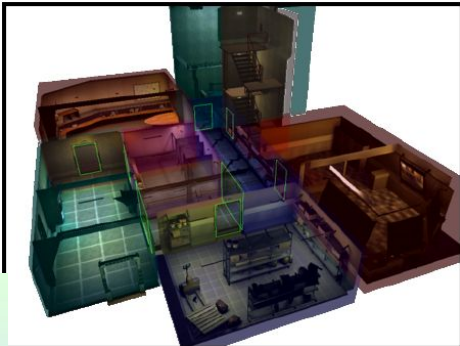
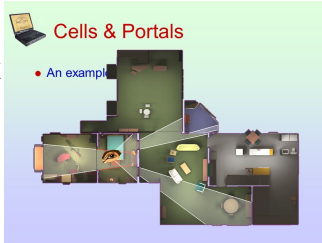
Warnock algorithm

- Input: viewport and list of polygons
- Simple cases:
 - Viewport is 1x1 pixel in size
 - Empty list or only a single polygon is in the list
- Otherwise: divide and conquer
 - Divide viewports into 4
 - Split list into 4
 - Call recursively
- $O(\text{polygons} * \text{pixels})$



Cells and Portals

- Start with current cell (room) within frustum
- For all portals (doors)
 - Cut frustum cone to portal
 - Draw next cell
 - Continue recursively
- Portals must be convex
- Cons:
 - A lot of cuts
 - Only interiors
- Pros:
 - Fast
 - Works on dynamic scenes
- Potentially **Visible Set**: Similar...



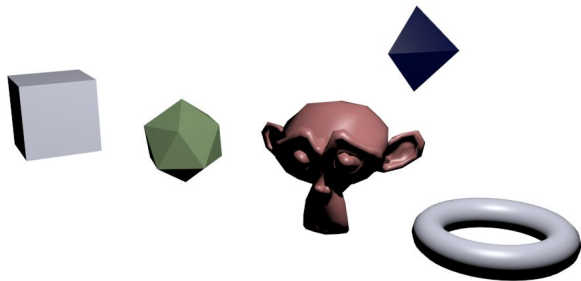
Ray Tracing APIs

Mesh shaders

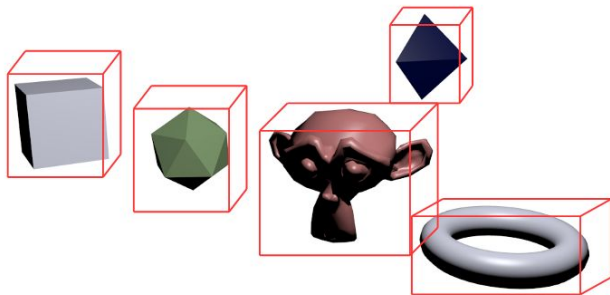
Csaba Bálint
with selected slides of
Gábor Valasek



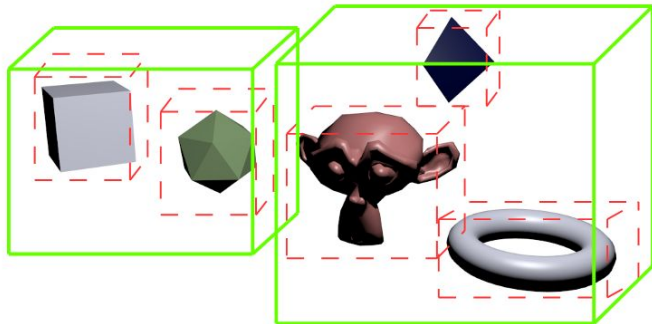
Acceleration structures



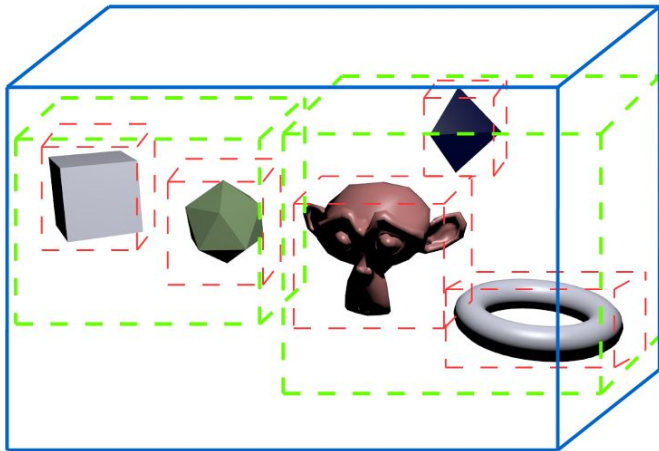
Acceleration structures



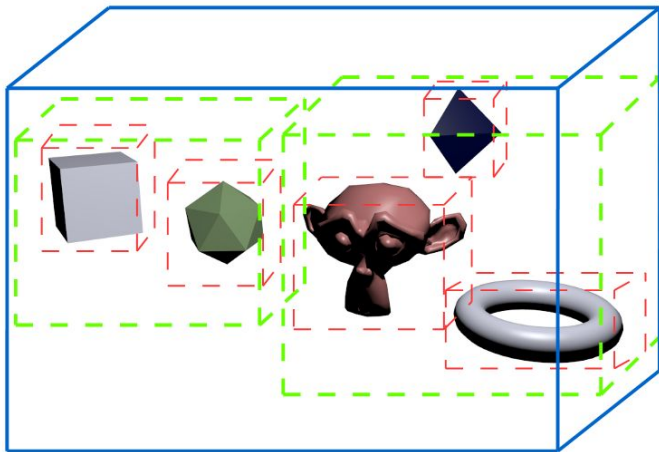
Acceleration structures



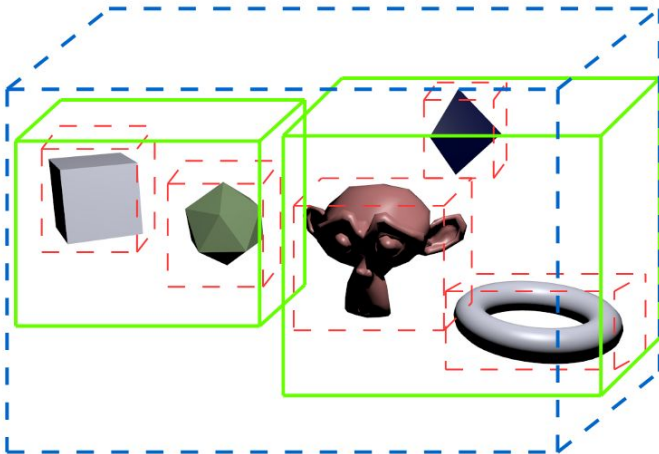
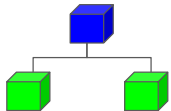
Acceleration structures



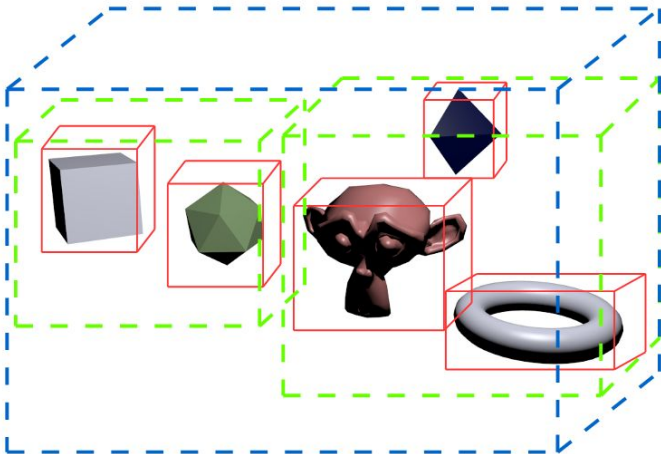
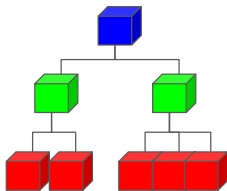
Acceleration structures



Acceleration structures



Acceleration structures



Part I:



with selected slides of:
Gábor Valasek

DirectX 12 Raytracing

— API support

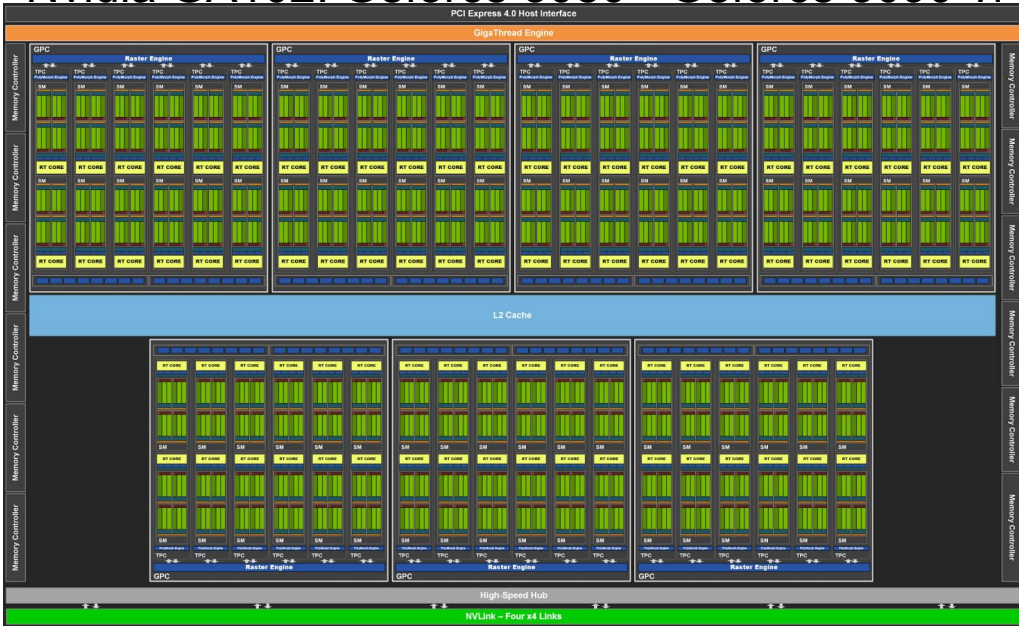
DirectX Ray Tracing:

- 0.01: initial draft in September, 2017
- 1.0: initial release in October, 2018 (preview since March, 2018)
- [1.1](#): major update in May, 2020

Vulkan Ray Tracing:

- Provisional extension: March 2020
- Final version: December 2020

Nvidia GA102: Geforce 3080 - Geforce 3090 Ti



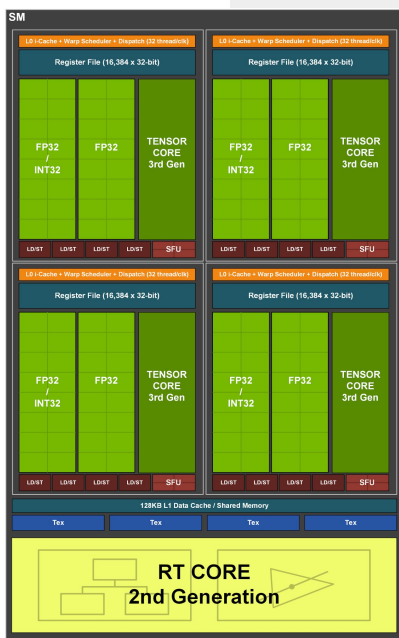
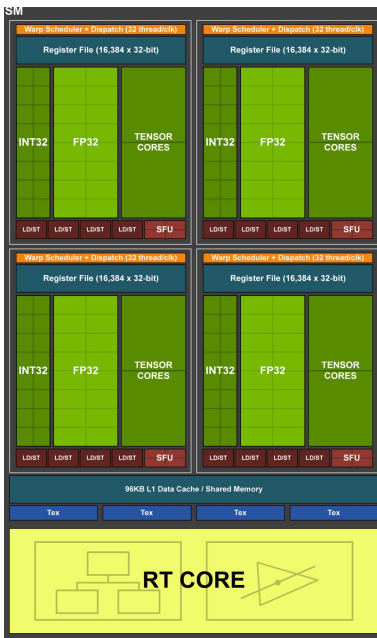
Hardware support for ray tracing

NVIDIA:

- 2018: Volta architecture (Titan V/V100, a professional card - not used in any consumer level GPUs)
- 2018: Turing (GeForce 20 series, consumer level, T4 for pros)
- 2020: Ampere (GeForce 30 series, consumer level, A100 et al. for professionals)

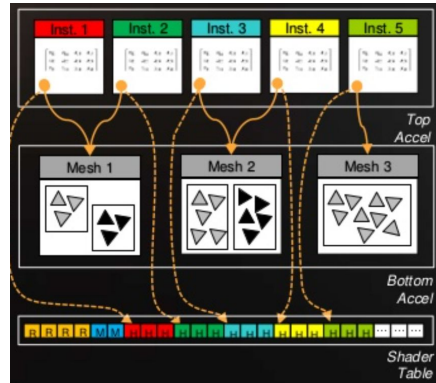
AMD:

- 2020: November, AMD RDNA 2 (Radeon RX 6000 desktop GPUs, also used in PS5, XBX, and even new Model S and X Tesla cars)

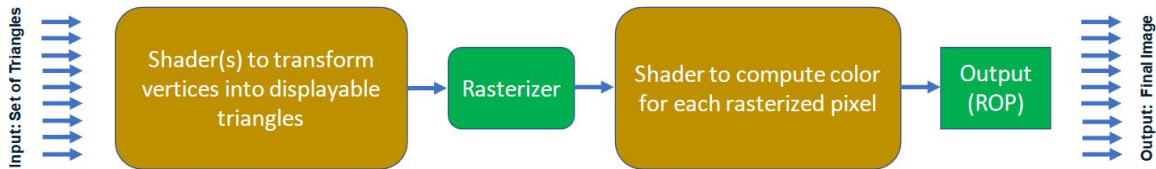


Acceleration Structure

- Bottom Level Acceleration Structure
 - “per object” acceleration structure
 - triangle mesh or procedural shape
 - procedural → define AABB and intersection shader
- Top Level Acceleration Structure
 - “scene” acceleration structure is a list
 - bottom level structures and instance data (eg. transformation matrix)
- Generated during runtime and must be updated regularly

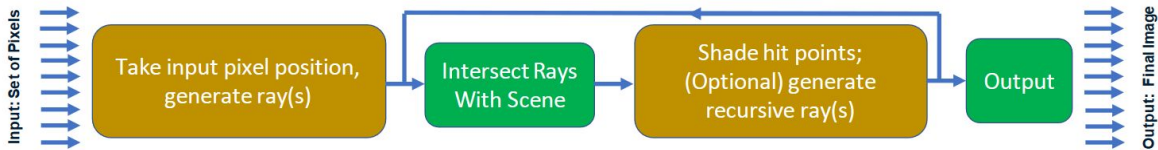


Ray tracing VS rasterization



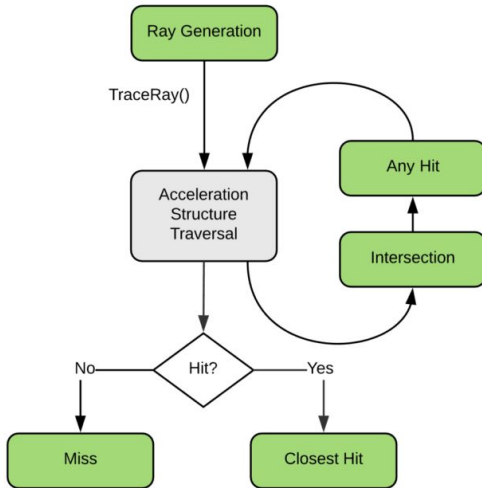
http://intro-to-dxr.cwyman.org/presentations/IntroDXR_RaytracingShaders.pdf

Ray tracing VS rasterization



http://intro-to-dxr.cwyman.org/presentations/IntroDXR_RaytracingShaders.pdf

DXR ray tracing pipeline



———— DXR ray tracing pipeline

The DXR pipeline is composed of five new shader types:

- Ray generation shaders
- Intersection shaders
- Closest-hit shaders
- Any-hit shaders
- Miss shaders

Plus a new shader class:

- Callable shader

DXR ray tracing

The DXR pipeline is composed of five new

- Ray generation shaders
- Intersection shaders
- Closest-hit shaders
- Any-hit shaders
- Miss shaders

Plus a new shader class:

- Callable shader



```
struct SceneConstantStructure { ... };

ConstantBuffer<SceneConstantStructure> SceneConstants;
RaytracingAccelerationStructure MyAccelerationStructure : register(t3);
struct MyPayload { ... };

[shader("raygeneration")]
void raygen_main() {
    ...
    RayDesc myRay = {
        <ray origin>,
        <TMin>,
        <ray direction>,
        <TMax> };
    MyPayload payload = { ... }; // init payload
    TraceRay(
        MyAccelerationStructure,
        <trace flags>,
        <optional flag to mask out instances>,
        <RayContributionToHitGroupIndex>,
        <MultiplierForGeometryContributionToHitGroupIndex>,
        <MissShaderIndex>,
        myRay,
        payload);
    ...
    WriteFinalPixel(DispatchRaysIndex(), payload);
}
```

DXR ray tracing

The DXR pipeline is composed of five new

- Ray generation shaders
- Intersection shaders
- Closest-hit shaders
- Any-hit shaders
- Miss shaders

Plus a new shader class:

- Callable shader



```
struct CustomPrimitiveDef { ... };
struct MyAttributes { ... };
struct CustomIntersectionIterator {...};

void InitCustomIntersectionIterator(CustomIntersectionIterator it) {...}
bool IntersectCustomPrimitiveFrontToBack(
    CustomPrimitiveDef prim,
    inout CustomIntersectionIterator it,
    float3 origin, float3 dir,
    float rayTMin, inout float curT,
    out MyAttributes attr) {...}

[shader("intersection")]
void intersection_main() {
    float THit = RayTCurrent();
    MyAttributes attr;
    CustomIntersectionIterator it;
    InitCustomIntersectionIterator(it);

    while(IntersectCustomPrimitiveFrontToBack(
        CustomPrimitiveDefinitions[LocalConstants.PrimitiveIndex],
        it, ObjectRayOrigin(), ObjectRayDirection(),
        RayTMin(), THit, attr)) {
        if (ReportHit(THit, /*hitKind*/ 0, attr) &&
            RayFlags() & RAY_FLAG_FORCE_OPAQUE)
            break;
    }
}
```

DXR ray tr

The DXR pipeline is composed of

- Ray generation shaders
- Intersection shaders
- **Closest-hit shaders**
- Any-hit shaders
- Miss shaders

Plus a new shader class:

- Callable shader

```
[shader("closesthit")]
void closesthit_main(inout MyPayload payload, in MyAttributes attr)
{
    CallShader( ... ); // maybe needed to shade

    float3 worldRayOrigin =
        WorldRayOrigin() + WorldRayDirection() * RayTCurrent();
    ...
    float3 worldNormal = mul(attr.normal, float3x3)ObjectToWorld3x4();
    RayDesc reflectedRay = { worldRayOrigin, SceneConstants.Epsilon,
                            ReflectRay(WorldRayDirection(), worldNormal),
                            SceneConstants.TMax };

    TraceRay(MyAccelerationStructure,
             SceneConstants.RayFlags,
             SceneConstants.InstanceInclusionMask,
             SceneConstants.RayContributionToHitGroupIndex,
             SceneConstants.MultiplierForGeometryContributionToHitGroupIndex,
             SceneConstants.MissShaderIndex,
             reflectedRay,
             payload);
    ...
}
```

DXR ray tracing pipeline

The DXR pipeline is composed of

- Ray generation shaders
- Intersection shaders
- Closest-hit shaders
- **Any-hit shaders**
- Miss shaders

Plus a new shader class:

- Callable shader

```
[shader("anyhit")]
void anyhit_main( inout MyPayload payload, in MyAttributes attr ) {
    float3 hitLocation =
        ObjectRayOrigin() + ObjectRayDirection() * RayTCurrent();

    float alpha = computeAlpha(hitLocation, attr, ...);

    // Processing shadow and only care if a hit is registered?
    if (TerminateShadowRay(alpha)) AcceptHitAndEndSearch();

    // Save alpha contribution and ignoring hit?
    if (SaveAndIgnore(payload, RayTCurrent(), alpha, attr, ...))
        IgnoreHit();
    ...
}
```

DXR ray tracing pipeline

The DXR pipeline is composed of five new

- Ray generation shaders
- Intersection shaders
- Closest-hit shaders
- Any-hit shaders
- Miss shaders

Plus a new shader class:

- Callable shader

```
[shader("miss")]  
void miss_main(inout MyPayload payload) {  
    // Use ray system values to compute contributions of  
    // background, sky, etc.  
  
    // Combine contributions into ray payload  
    CallShader( ... ); // maybe  
  
    TraceRay( ... ); // maybe  
  
    // this ray query is now complete  
}
```

DXR ray tracing pipeline

The DXR pipeline is composed of five new shader types:

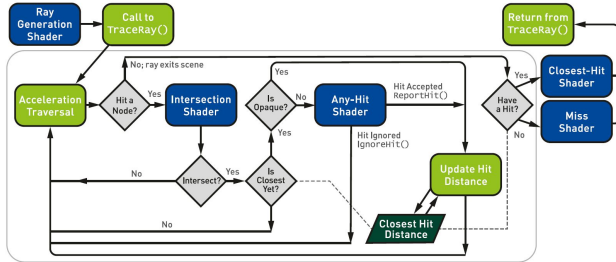
- Ray generation shaders
- Intersection shaders
- Closest-hit shaders
- Any-hit shaders
- Miss shaders

Plus a new shader class:

- Callable shader

```
[shader("callable")]  
void callable_main(inout MyParams params)  
{  
    // Perform some common operations and update params  
    CallShader( ... ); // maybe  
}
```

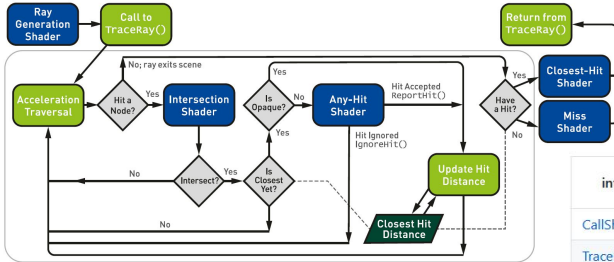

DirectX BVH (Bounding Volume Hierarchy)



```
1 RaytracingAccelerationStructure scene;           // Scene BVH from C++
2 RayDesc ray = { rayOrigin, minHitDist, rayDirection, maxHitDist };
3 UserDefinedPayloadStruct payload = { ... <initialize here>... };
4
5 TraceRay( scene, RAY_FLAG_NONE, instancesToQuery, // what geometry?
6          hitGroup, numHitGroups, missShader,     // which shaders?
7          ray,                                     // what ray to trace?
8          payload );                               // what data to use?
```

```
for  $x, y \in \text{image.dims}()$  do
  [1] ray = computeRay(x, y);
  closestHit = null;
  while
    leaf = findBvhLeafNode(ray, scene)
  do
    [2] hit = intersectGeometry(ray, leaf);
    if isCloser(hit, closestHit) then
      if [3] isOpaque(hit) then
        closestHit = hit;
  if closestHit then
    [4] image[x,y] = shade(ray, closestHit);
  else
    [5] image[x,y] = miss(ray);
```

DirectX BVH and intrinsics



intrinsic \ shader	ray generation	intersection	any hit	closest hit	miss	callable
CallShader()	*			*	*	*
TraceRay()	*			*	*	
ReportHit()		*				
IgnoreHit()			*			
AcceptHitAndEndSearch()			*			

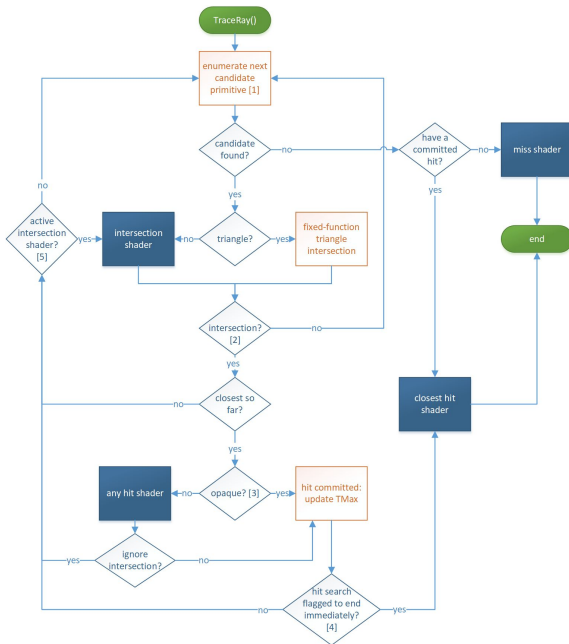
```

1 RaytracingAccelerationStructure scene; // Scene BVH from C+
2 RayDesc ray = { rayOrigin, minHitDist, rayDirection, maxHitDist };
3 UserDefinedPayloadStruct payload = { ... <initialize here>... };
4
5 TraceRay( scene, RAY_FLAG_NONE, instancesToQuery, // what geometry?
6           hitGroup, numHitGroups, missShader, // which shaders?
7           ray, // what ray to trace?
8           payload ); // what data to use?
  
```

DXR intrinsics

values \ shaders	ray generation	intersection	any hit	closest hit	miss	callable
<i>Ray dispatch system values:</i>						
uint3 DispatchRaysIndex()	*	*	*	*	*	*
uint3 DispatchRaysDimensions()	*	*	*	*	*	*
<i>Ray system values:</i>						
float3 WorldRayOrigin()		*	*	*	*	
float3 WorldRayDirection()		*	*	*	*	
float RayTMin()		*	*	*	*	
float RayTCurrent()		*	*	*	*	
uint RayFlags()		*	*	*	*	
<i>Primitive/object space system values:</i>						
uint InstanceIndex()		*	*	*		
uint InstanceID()		*	*	*		
uint GeometryIndex() (requires Tier 1.1 implementation)		*	*	*		
uint PrimitiveIndex()		*	*	*		
float3 ObjectRayOrigin()		*	*	*		
float3 ObjectRayDirection()		*	*	*		
float3x4 ObjectToWorld3x4()		*	*	*		
float4x3 ObjectToWorld4x3()		*	*	*		
float3x4 WorldToObject3x4()		*	*	*		
float4x3 WorldToObject4x3()		*	*	*		
<i>Hit specific system values:</i>						
uint HitKind()			*	*		

intrinsics \ shaders	ray generation	intersection	any hit	closest hit	miss	callable
CallShader()	*			*	*	*
TraceRay()	*			*	*	
ReportHit()		*				
IgnoreHit()			*			
AcceptHitAndEndSearch()			*			



HLSL Support

- Ray traversal functions
- Launch introspection
 - launch data(which pixel, ray)
- Ray introspection
 - ray payload data
- Object introspection
 - bottom instance data
- Hit introspection
 - user defined

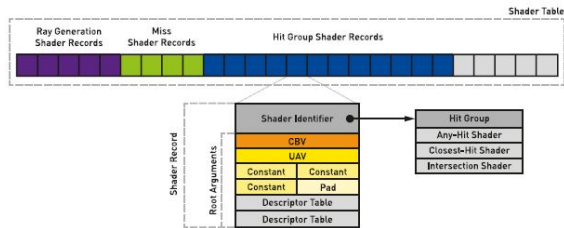
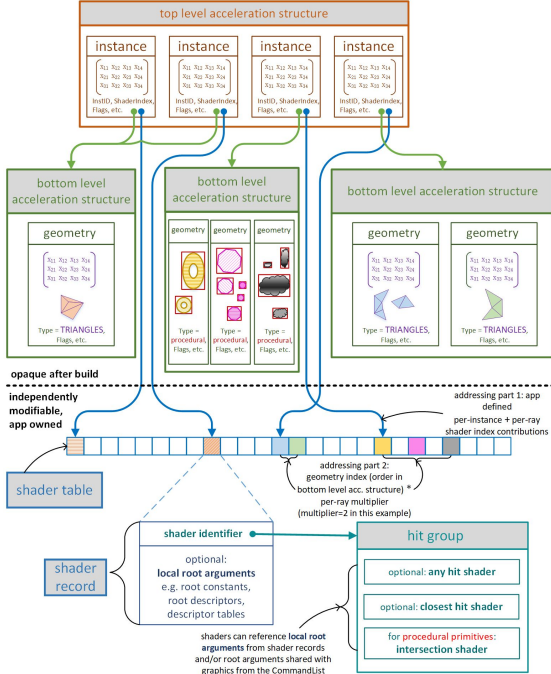


Figure 3-3. A visualization of a DXR shader table and its shader records. Shader Records contain a shader identifier and root arguments used to look up resources.



Part II:

KHRONOS[®]
GROUP

**Vulkan Ray
Tracing Launch**



Vulkan raytracing vs DXR

	Vulkan Ray Tracing	DX12 / DXR
Ray Tracing Pipelines	At least one must be available	Yes
Ray Queries		DXR Tier 1.1 Inline ray tracing
Language for Ray Tracing Shaders	GLSL or HLSL	HLSL
Pipeline Libraries	Yes	DXR Tier 1.1 AddToStateObject()
Build Acceleration Structure on Host	Optional	No
Deferred Host Operations	Optional	No
Capture/Replay Support for Tools (e.g. RenderDoc)	Optional	No

Figure 5: Comparing Vulkan Ray Tracing and DXR. It is straightforward to port code between the two APIs including re-use of ray tracing shaders written in HLSL

Vulkan ray tracing extension

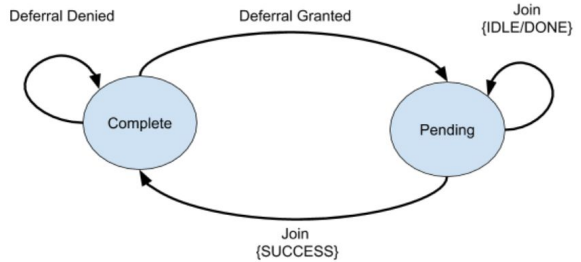
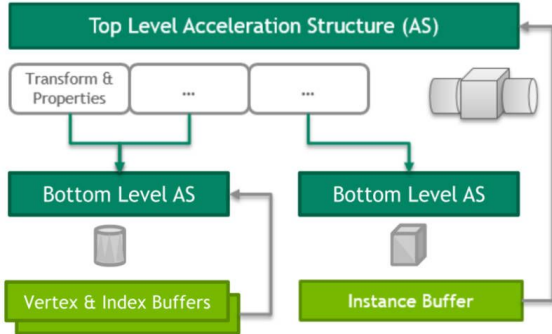


Figure 4: Deferred Operation State Diagram

Load balancing

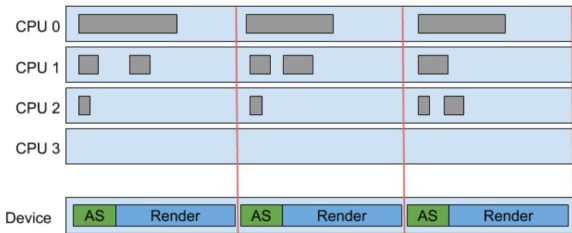


Figure 5: Load balancing: No Host Build

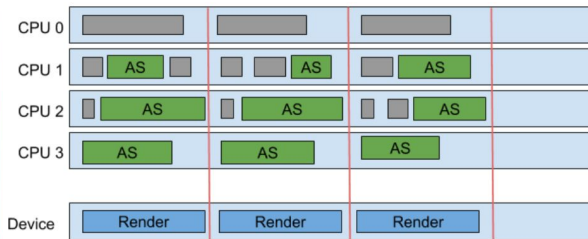
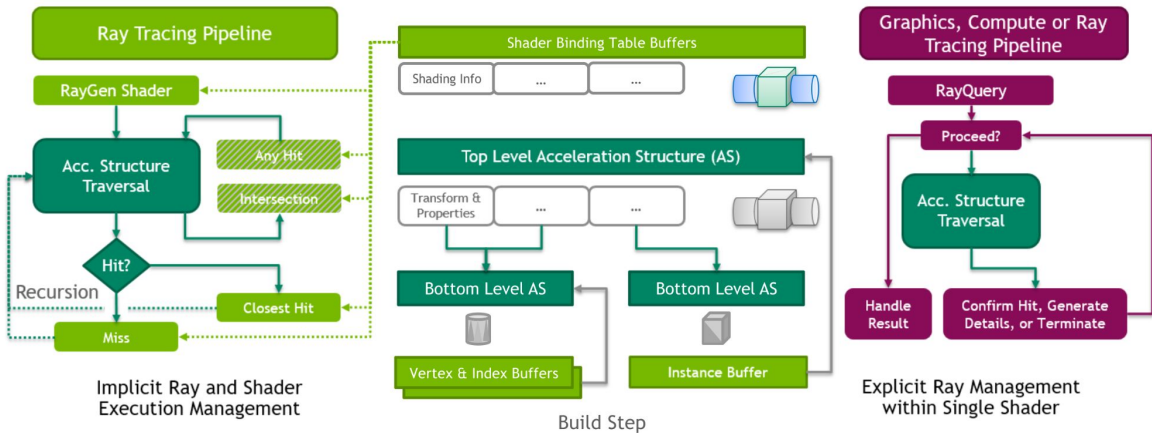
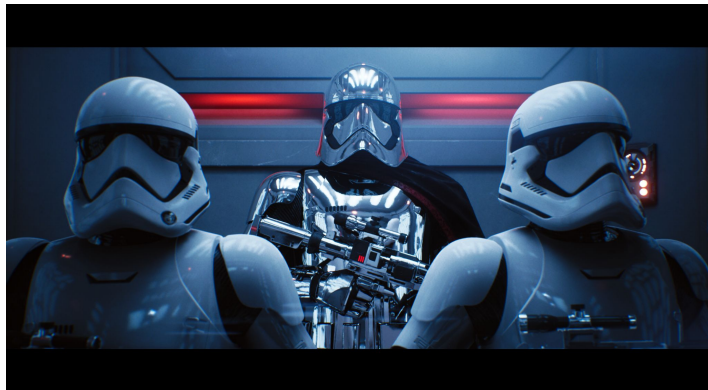


Figure 6: Load Balancing: Host Build Enabled

Ray Tracing Architecture



Part III: Applications



<https://www.youtube.com/watch?v=J3ue35ago3Y>

— Things to look out for

- On NVIDIA, RT Cores are (partially) independent pipelines - if you are not using RTX, you do waste some GPU resources
- Ray tracing is a 2-in-1 deal: you get bounding volume-ray and primitive-ray intersection capabilities - think about how you can use this beyond graphics
- Even with optimizations, you have a severely limited ray budget per frame (1-2 rays per pixel) - you need to tackle variance
- If doing recursions, mind your stack

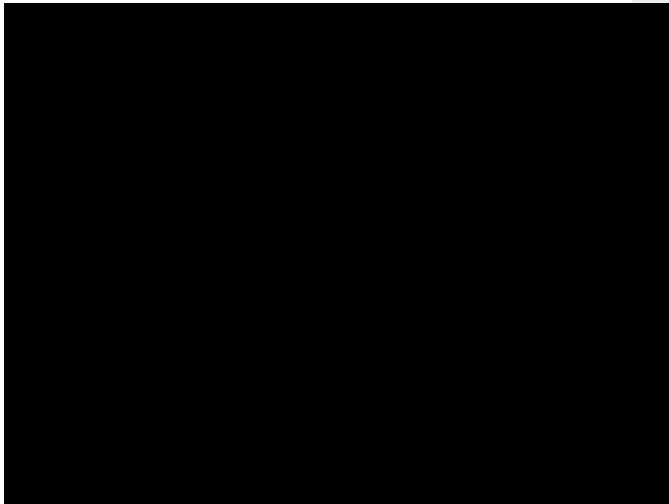
No recursion

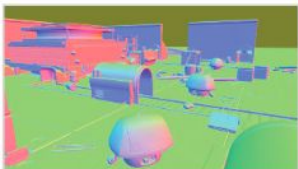


— Recursion depth = 1



— **Recursion depth = 2 + temporal accumulation**





G-Buffer
(Raster)



Direct Shadows
(Ray Trace or Raster)



Direct Lighting
(Compute)



Reflections
(Ray Trace or Compute)



Global Illumination
(Ray Trace and Compute)



Ambient Occlusion
(Ray trace or Compute)



**Transparency
& Translucency**
(Ray Trace and Compute)



Post-Processing
(Compute)

Part IV:

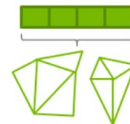
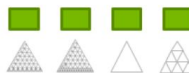
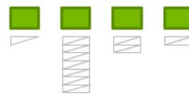


Mesh Shaders



Mesh Shaders

Shader		Thread Mapping	Topology
Vertex Shader	No access to connectivity	1 Vertex	No influence
Geometry Shader	Variable output doesn't fit HW well	1 Primitive / 1 Output Strip	Triangle Strips
Tessellation Shader	Fixed-function topology	1 Patch / 1 Evaluated Vertex	Fast Patterns
Mesh Shader	Compute shader features	Flexible	Flexible within work group allocation



Meshlets

Detail & Clutter Objects

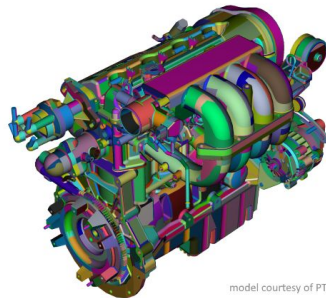
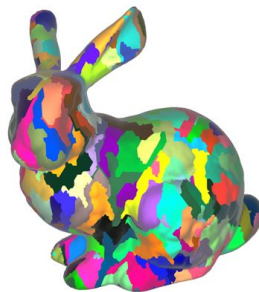


© ART BY RENS [1]

High Geometric Complexity



© photo by Chris Christian - model by Russell Berkoff [2]



model courtesy of PTC

LOD

0

1

3

5

7

9



Triangles

20

80

1,280

20,480

327,680

5,242,880

Task and Mesh shader

MESHLETS

TRADITIONAL PIPELINE



TASK/MESH PIPELINE



Compute mesh modell

- **Task shader** : a programmable unit that operates in workgroups and allows each to emit (or not) mesh shader workgroups
- **Mesh shader** : a programmable unit that operates in workgroups and allows each to generate primitives

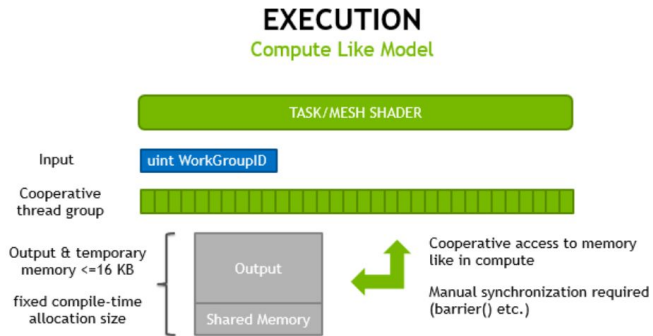
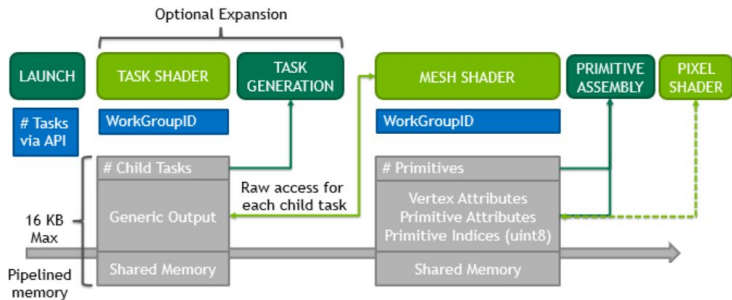


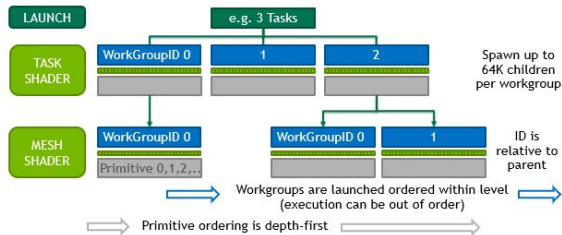
Figure 5. Mesh shaders behave similarly to compute shaders in using a cooperative thread model.

Compute mesh model



- **Higher scalability:** less fixed-function, but scalable
- **Bandwidth-reduction:** vertex re-use, index buffers, own VBO compression
- **Felxibility:** replaces both geometry and tessellation without their drawbacks

Tree Expansion



Thank you for your attention!

Mesh shader references

- <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/>
- <https://blog.siggraph.org/2021/04/mesh-shaders-release-the-intrinsic-power-of-a-gpu.html/>
- <https://microsoft.github.io/DirectX-Specs/d3d/MeshShader.html>
- <https://www.geeks3d.com/20200519/introduction-to-mesh-shaders-opengl-and-vulkan/>

Ray tracing references

- **Ray Tracing Gems:** <https://www.realtimerendering.com/raytracinggems/>
- **Microsoft:** [DXR functional specification](#)
- **Khronos:** [Vulkan Ray Tracing specification](#)
- **Chris Wyman:** [Introduction to ray tracing \(Siggraph 2018\)](#)
- **Peter Shirley:** [Ray Tracing in One Weekend](#)
[Ray Tracing: The Next Week](#)
[Ray Tracing: The Rest of Your Life](#)