Advanced Graphics

Beziers, B-splines, and NURBS

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

Bezier splines, B-Splines, and NURBS

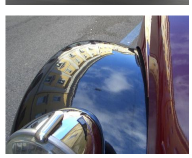
Expensive products are sleek and smooth.
→ Expensive products are C2 continuous.


Shiny, but reflections are warped


Shiny, and reflections are perfect

History

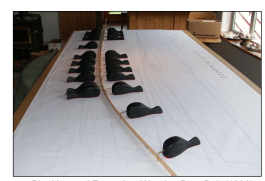
- *Continuity* (smooth curves) can be essential to the perception of *quality*.
- The automotive industry wanted to design cars which were aerodynamic, but also visibly of high quality.
- Bezier (Renault) and de Casteljau (Citroen) invented Bezier curves in the 1960s. de Boor (GM) generalized them to B-splines.



History

The term *spline* comes from the shipbuilding industry: long, thin strips of wood or metal would be bent and held in place by heavy 'ducks', lead weights which acted as control points of the curve.

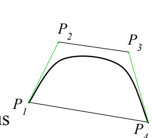
Wooden splines can be described by C_n-continuous Hermite polynomials which interpolate n+1 control points.



Top: Fig 3, P.7, Bray and Spectre, *Planking and Fastening*, Wooden Boat Pub (1996)
Bottom: http://www.pranos.com/boatsofwood/lofting%20ducks/lofting_ducks.htm

Beziers—a quick review

- A Bezier cubic is a function P(t) defined by four control points:
  - P_1 and P_4 are the endpoints of the curve
  - P_2 and P_3 define the other two corners of the bounding polygon.
- The curve fits entirely within the convex hull of $P_1 \ldots P_4$.
- A degree-*d* Bezier is infinitely continuous throughout its interior. However, when joining two Beziers, careful placement of the control points is required to ensure continuity.



Cubic: $P(t) = (1-t)^3P_1 + 3t(1-t)^2P_2 + 3t^2(1-t)P_3 + t^3P_4$

Beziers

Cubics are just one example of Bezier splines:

- Linear: $P(t) = (1-t)P_1 + tP_2$
- Quadratic: $P(t) = (1-t)^2P_1 + 2t(1-t)P_2 + t^2P_3$
- Cubic: $P(t) = (1-t)^3P_1 + 3t(1-t)^2P_2 + 3t^2(1-t)P_3 + t^3P_4$

...

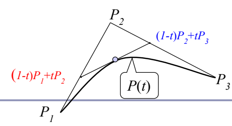
General:

"n choose i" = n! / i!(n-i)!

$$P(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i,\ 0 \le t \le 1$$

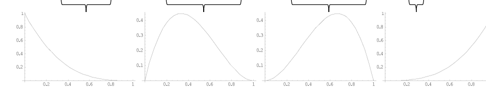
Beziers

- You can describe Beziers as *nested linear interpolations:*
  - The linear Bezier is a linear interpolation between two points:
    $P(t) = (1\text{-}t)\,(P_1) + (t)\,(P_2)$
  - The quadratic Bezier is a linear interpolation between two lines:
    $P(t) = (1\text{-}t)\,((1\text{-}t)P_1+tP_2) + (t)\,((1\text{-}t)P_2+tP_3)$
  - The cubic is a linear interpolation between linear interpolations between linear interpolations… etc.
- Another way to see Beziers is as a *weighted average* between the control points.



Bernstein polynomials

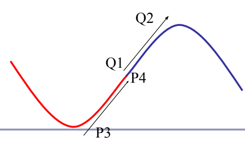
$$P(t) = (1\text{-}t)^3P_1 + 3t(1\text{-}t)^2P_2 + 3t^2(1\text{-}t)P_3 + t^3P_4$$



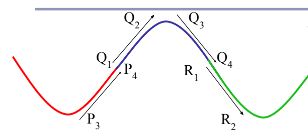
- The four control functions are the four *Bernstein polynomials* for $n=3$.
  - General form: $b_{v,n}(t) = \binom{n}{v} t^v (1-t)^{n-v}$
  - Bernstein polynomials in $0 \le t \le 1$ always sum to 1:
    $\sum_{v=1}^{n} \binom{n}{v} t^v (1-t)^{n-v} = (t + (1-t))^n = 1$

Joining Bezier splines

- To join two Bezier splines with C0 continuity, set $P_4=Q_1$.
- To join two Bezier splines with C1 continuity, require C0 and make the tangent vectors equal: set $P_4=Q_1$ and $P_4\text{-}P_3=Q_2\text{-}Q_1$.



What if we want to chain Beziers together?



Consider a chain of splines with many control points…
$P = \{P_0, P_1, P_2, P_3\}$
$Q = \{Q_0, Q_1, Q_2, Q_3\}$
$R = \{R_0, R_1, R_2, R_3\}$
…with C1 continuity…
$P3=Q_0$, $P_2\text{-}P_3=Q_0\text{-}Q_1$
$Q3=R_0$, $Q_2\text{-}Q_3=R_0\text{-}R_1$

We can parameterize this chain over *t* by saying that instead of going from 0 to 1, *t* moves smoothly through the intervals [0,1,2,3]

The curve $C(t)$ would be:
$C(t) = P(t) \bullet ((0 \le t < 1)\ ?\ 1 : 0) +$
$Q(t\text{-}1) \bullet ((1 \le t < 2)\ ?\ 1 : 0) +$
$R(t\text{-}2) \bullet ((2 \le t < 3)\ ?\ 1 : 0)$

[0,1,2,3] is a type of *knot vector*. 0, 1, 2, and 3 are the *knots*.

NURBS

- *NURBS* ("*Non-Uniform Rational B-Splines*") are a generalization of Beziers.
  - NU: *Non-Uniform*. The knots in the knot vector are not required to be uniformly spaced.
  - R: *Rational*. The spline may be defined by rational polynomials (homogeneous coordinates.)
  - BS: *B-Spline*. A generalization of Bezier splines with controllable degree.

B-Splines

- A Bezier cubic is a polynomial of degree three: it must have four control points, it must begin at the first and end at the fourth, and it assumes that all four control points are equally important.

- *B-spline* curves are a piecewise parameterization of a series of splines, that supports an arbitrary number of control points and lets you specify the degree of the polynomial which interpolates them.

B-Splines

We'll build our definition of a B-spline from:

- d, the *degree* of the curve
- $k = d+1$, called the *parameter* of the curve
- $\{P_1 \dots P_n\}$, a list of n *control points*
- $[t_1, \dots, t_{k+n}]$, a *knot vector* of (k+n) parameter values

- $d = k-1$ is the degree of the curve, so k is the number of control points which influence a single interval.
  - Ex: a cubic (d=3) has four control points (k=4).
- There are $k+n$ knots, and $t_i \le t_{i+1}$ for all t_i.
- Each B-spline is $C^{(k-2)}$ continuous: *continuity* is degree minus one, so a k=3 curve has d=2 and is C1.

B-Splines

- The equation for a B-spline curve is

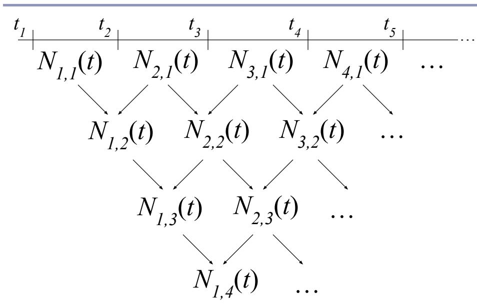
$$P(t) = \sum_{i=1}^{n} N_{i,k}(t) P_i, \; t_{min} \le t < t_{max}$$

- $N_{i,k}(t)$ is the *basis function* of control point P_i for parameter k. $N_{i,k}(t)$ is defined recursively:

$$N_{i,1}(t) = \begin{cases} 1, & t_i \le t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

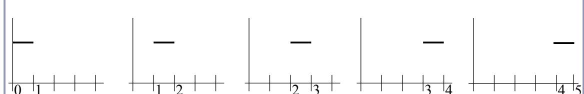
$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t)$$

B-Splines



B-Splines

$$N_{i,1}(t) = \begin{cases} 1, & t_i \le t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$



$N_{1,1}(t)$ $N_{2,1}(t)$ $N_{3,1}(t)$ $N_{4,1}(t)$ $N_{5,1}(t)$

$t_1 = 0.0$
$t_2 = 1.0$
$t_3 = 2.0$
$t_4 = 3.0$
$t_5 = 4.0$
$t_6 = 5.0$

$N_{1,1}(t)=1, 0 \le t < 1$

$N_{2,1}(t)=1, 1 \le t < 2$

$N_{3,1}(t)=1, 2 \le t < 3$

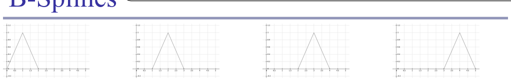
$N_{4,1}(t)=1, 3 \le t < 4$

$N_{5,1}(t)=1, 4 \le t < 5$

Knot vector = {0,1,2,3,4,5}, $k = 1 \rightarrow d = 0$ (degree = zero)

B-Splines

$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t)$$



$N_{1,2}(t)$ $N_{2,2}(t)$ $N_{3,2}(t)$ $N_{4,2}(t)$

$$N_{1,2}(t) = \frac{t-0}{1-0} N_{1,1}(t) + \frac{2-t}{2-1} N_{2,1}(t) = \begin{cases} t & 0 \le t < 1 \\ 2-t & 1 \le t < 2 \end{cases}$$

$$N_{2,2}(t) = \frac{t-1}{2-1} N_{2,1}(t) + \frac{3-t}{3-2} N_{3,1}(t) = \begin{cases} t-1 & 1 \le t < 2 \\ 3-t & 2 \le t < 3 \end{cases}$$

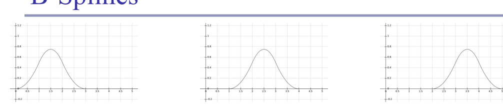
$$N_{3,2}(t) = \frac{t-2}{3-2} N_{3,1}(t) + \frac{4-t}{4-3} N_{4,1}(t) = \begin{cases} t-2 & 2 \le t < 3 \\ 4-t & 3 \le t < 4 \end{cases}$$

$$N_{4,2}(t) = \frac{t-3}{4-3} N_{4,1}(t) + \frac{5-t}{5-4} N_{5,1}(t) = \begin{cases} t-3 & 3 \le t < 4 \\ 5-t & 4 \le t < 5 \end{cases}$$

Knot vector = {0,1,2,3,4,5}, $k = 2 \rightarrow d = 1$ (degree = one)

B-Splines

$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t)$$



$N_{1,3}(t)$ $N_{2,3}(t)$ $N_{3,3}(t)$

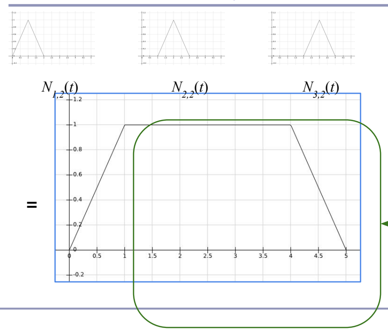
$$N_{1,3}(t) = \frac{t-0}{2-0} N_{1,2}(t) + \frac{3-t}{3-1} N_{2,2}(t) = \begin{cases} t^2/2 & 0 \le t < 1 \\ -t^2 + 3t - 3/2 & 1 \le t < 2 \\ (3-t)^2/2 & 2 \le t < 3 \end{cases}$$

$$N_{2,3}(t) = \frac{t-1}{3-1} N_{2,2}(t) + \frac{4-t}{4-2} N_{3,2}(t) = \begin{cases} (t-1)^2/2 & 1 \le t < 2 \\ -t^2 + 5t - 11/2 & 2 \le t < 3 \\ (4-t)^2/2 & 3 \le t < 4 \end{cases}$$

$$N_{3,3}(t) = \frac{t-2}{4-2} N_{3,2}(t) + \frac{5-t}{5-3} N_{4,2}(t) = \begin{cases} (t-2)^2/2 & 2 \le t < 3 \\ -t^2 + 7t - 23/2 & 3 \le t < 4 \\ (5-t)^2/2 & 4 \le t < 5 \end{cases}$$

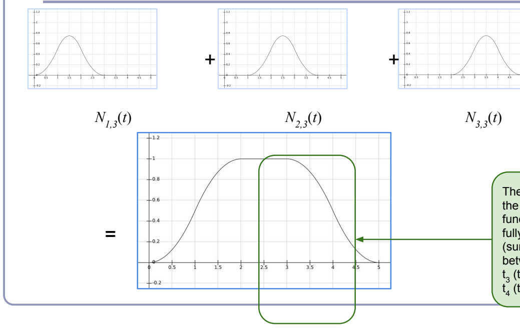
Knot vector = {0,1,2,3,4,5}, $k = 3 \rightarrow d = 2$ (degree = two)

Basis functions really sum to one (k=2)



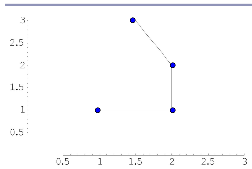
The sum of the four basis functions is fully defined (sums to one) between t_2 (t=1.0) and t_5 (t=4.0).

Basis functions really sum to one (k=3)

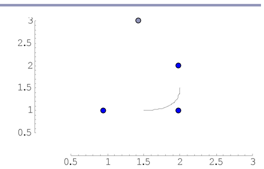


The sum of the three functions is fully defined (sums to one) between t_3 (t=2.0) and t_4 (t=3.0).

B-Splines



At k=2 the function is piecewise linear, depends on P_1, P_2, P_3, P_4, and is fully defined on $[t_2, t_5)$.



At k=3 the function is piecewise quadratic, depends on P_1, P_2, P_3, and is fully defined on $[t_3, t_4)$.

Each parameter-k basis function depends on $k+1$ knot values; $N_{i,k}$ depends on t_i through t_{i+k}, inclusive. So six knots → five discontinuous functions → four piecewise linear interpolations → three quadratics, interpolating three control points. n=3 control points, d=2 degree, k=3 parameter, n+k=6 knots.

Knot vector = {0,1,2,3,4,5}

Non-Uniform B-Splines

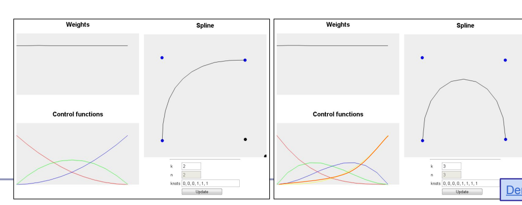
- The knot vector {0,1,2,3,4,5} is *uniform*:
  $t_{i+1}-t_i = t_{i+2}-t_{i+1}\ \forall t_i$.
- Varying the size of an interval changes the parametric-space distribution of the weights assigned to the control functions.
- Repeating a knot value reduces the continuity of the curve in the affected span by one degree.
- Repeating a knot k times will lead to a control function being influenced <u>only</u> by that knot value; the spline will pass through the corresponding control point with C0 continuity.

Open vs *Closed*

- A knot vector which repeats its first and last knot values k times is called *open*, otherwise *closed*.
  - Repeating the knots k times is the only way to force the curve to pass through the first or last control point.
  - Without this, the functions $N_{1,k}$ and $N_{n,k}$ which weight P_1 and P_n would still be 'ramping up' and not yet equal to one at the first and last t_i.

Open vs Closed

- Two examples you may recognize:
  - k=3, n=3 control points, knots={0,0,0,1,1,1}
  - k=4, n=4 control points, knots={0,0,0,0,1,1,1,1}



Demo

Non-Uniform *Rational* B-Splines

- Repeating knot values is a clumsy way to control the curve's proximity to the control point.
  - We want to be able to slide the curve nearer or farther without losing continuity or introducing new control points.
  - The solution: *homogeneous coordinates*.
  - Associate a 'weight' with each control point: ω_i.

Non-Uniform Rational B-Splines

- Recall: $[x, y, z, \omega]_H \rightarrow [x / \omega, y / \omega, z / \omega]$
  - Or: $[x, y, z, 1] \rightarrow [x\omega, y\omega, z\omega, \omega]_H$
- The control point

  $P_i = (x_i, y_i, z_i)$

  becomes the homogeneous control point

  $P_{iH} = (x_i\omega_i, y_i\omega_i, z_i\omega_i)$
- A NURBS in homogeneous coordinates is:

$$P_H(t) = \sum_{i=1}^{n} N_{i,k}(t) P_{iH}, \; t_{min} \le t < t_{max}$$

Non-Uniform Rational B-Splines

- To convert from homogeneous coords to normal coordinates:

$$\begin{aligned} x_H(t) &= \textstyle\sum_{i=1}^{n} (x_i \omega_i)(N_{i,k}(t)) \\ y_H(t) &= \textstyle\sum_{i=1}^{n} (y_i \omega_i)(N_{i,k}(t)) \\ z_H(t) &= \textstyle\sum_{i=1}^{n} (z_i \omega_i)(N_{i,k}(t)) \\ \omega(t) &= \textstyle\sum_{i=1}^{n} (\omega_i)(N_{i,k}(t)) \end{aligned}$$

$$\begin{aligned} x(t) &= x_H(t) / \omega(t) \\ y(t) &= y_H(t) / \omega(t) \\ z(t) &= z_H(t) / \omega(t) \end{aligned}$$

Non-Uniform Rational B-Splines

- A piecewise rational curve is thus defined by:

$$P(t) = \sum_{i=1}^{n} R_{i,k}(t) P_i, \; t_{min} t < t_{max}$$

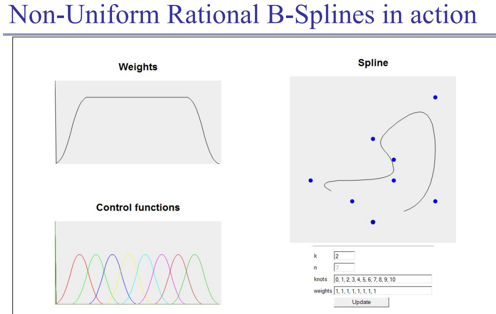
with supporting *rational basis functions*:

$$R_{i,k}(t) = \frac{\omega_i N_{i,k}(t)}{\sum_{j=1}^{n} \omega_j N_{j,k}(t)}$$

This is essentially an average re-weighted by the ω's.

- Such a curve can be made to pass arbitrarily far or near to a control point by changing the corresponding weight.

Non-Uniform Rational B-Splines in action



Demo

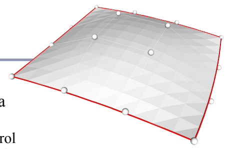
Tensor product

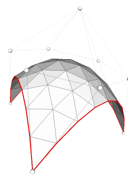
- The *tensor product* of two vectors is a matrix.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \otimes \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

- Can also take the tensor of two polynomials.
  - Each coefficient represents a piece of each of the two original expressions, to the cumulative polynomial represents both original polynomials completely.

NURBS patches





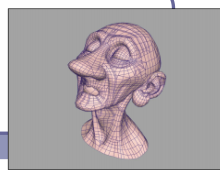
- The tensor product of the polynomial coefficients of two NURBS splines is a matrix of polynomial coefficients.
  - If curve A has parameter k and n control points and curve B has parameter j and m control points then $A \otimes B$ is an $(n)\times(m)$ matrix of polynomials of parameter $max(j,k)$.
- Multiply this matrix against an $(n)\times(m)$ matrix of control points and sum them all up and you've got a bivariate expression for a rectangular surface patch, in 3D
- This approach generalizes to triangles and arbitrary n-gons.

References

- Les Piegl and Wayne Tiller, *The NURBS Book*, Springer (1997)
- Alan Watt, *3D Computer Graphics*, Addison Wesley (2000)
- G. Farin, J. Hoschek, M.-S. Kim, *Handbook of Computer Aided Geometric Design*, North-Holland (2002)

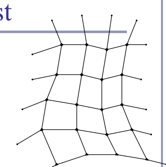
Advanced Graphics

Subdivision Surfaces



Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

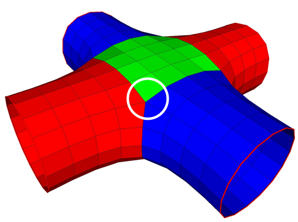
NURBS patches aren't the greatest



- NURBS patches are $n \times m$, forming a mesh of quadrilaterals.
  - What if you wanted triangles or pentagons?
    - A NURBS dodecahedron?
  - What if you wanted vertices of valence other than four?
- NURBS expressions for triangular patches, and more, do exist; but they're cumbersome.

Problems with NURBS patches

- Joining NURBS patches with C_n continuity across an edge is challenging.
- What happens to continuity at corners where the number of patches meeting isn't exactly four?
- Animation is tricky: bending and blending are doable, but not easy.



Sadly, the world isn't made up of shapes that can always be made from one smoothly-deformed rectangular surface.

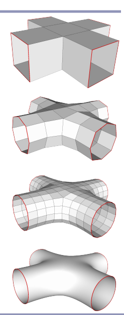
Subdivision surfaces

- Beyond shipbuilding: we want guaranteed continuity, without having to build everything out of rectangular patches.
  - Applications include CAD/CAM, 3D printing, museums and scanning, medicine, movies…
- The solution: *subdivision surfaces*.

Geri's Game, by Pixar (1997)

Subdivision surfaces

- Instead of ticking a parameter t along a parametric curve (or the parameters u,v over a parametric grid), subdivision surfaces repeatedly refine from a coarse set of *control points*.
- Each step of refinement adds new faces and vertices.
- The process converges to a smooth *limit surface*.



(Catmull-Clark in action)

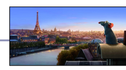
Subdivision surfaces – History

- de Rahm described a 2D (curve) subdivision scheme in 1947; rediscovered in 1974 by Chaikin
- Concept extended to 3D (surface) schemes by two separate groups during 1978:
  - Doo and Sabin found a biquadratic surface
  - Catmull and Clark found a bicubic surface
- Subsequent work in the 1980s (Loop, 1987; Dyn [Butterfly subdivision], 1990) led to tools suitable for CAD/CAM and animation

Subdivision surfaces and the movies

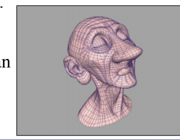
- Pixar first demonstrated subdivision surfaces in 1997 with Geri's Game.
  - Up until then they'd done everything in NURBS (Toy Story, A Bug's Life.)
  - From 1999 onwards everything they did was with subdivision surfaces (Toy Story 2, Monsters Inc, Finding Nemo...)
  - Two decades on, it's all heavily customized.
- It's not clear what Dreamworks uses, but they have recent patents on subdivision techniques.



Useful terms

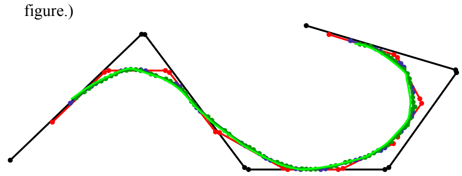
- A scheme which describes a 1D curve (even if that curve is travelling in 3D space, or higher) is called *univariate*, referring to the fact that the limit curve can be approximated by a polynomial in one variable (t).
- A scheme which describes a 2D surface is called *bivariate*, the limit surface can be approximated by a u,v parameterization.
- A scheme which retains and passes through its original control points is called an *interpolating* scheme.
- A scheme which moves away from its original control points, converging to a limit curve or surface nearby, is called an *approximating* scheme.



Control surface for Geri's head

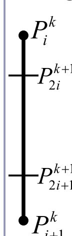
How it works

- Example: *Chaikin* curve subdivision (2D)
  - On each edge, insert new control points at ¼ and ¾ between old vertices; delete the old points
  - The *limit curve* is C1 everywhere (despite the poor figure.)



Notation

Chaikin can be written programmatically as:



$$P_{2i}^{k+1} = (3/4)P_i^k + (1/4)P_{i+1}^k \quad \leftarrow Even$$

$$P_{2i+1}^{k+1} = (1/4)P_i^k + (3/4)P_{i+1}^k \quad \leftarrow Odd$$

...where k is the 'generation'; each generation will have twice as many control points as before.

Notice the different treatment of generating odd and even control points.

Borders (terminal points) are a special case.

Notation

Chaikin can be written in vector notation as:

$$\begin{bmatrix} \vdots \\ P_{2i-2}^{k+1} \\ P_{2i-1}^{k+1} \\ P_{2i}^{k+1} \\ P_{2i+1}^{k+1} \\ P_{2i+2}^{k+1} \\ P_{2i+3}^{k+1} \\ \vdots \end{bmatrix} = \frac{1}{4} \begin{bmatrix} & & & \vdots & & & \\ \cdots & 0 & 3 & 1 & 0 & 0 & 0 & \cdots \\ & 0 & 1 & 3 & 0 & 0 & 0 & \\ & 0 & 0 & 3 & 1 & 0 & 0 & \\ & 0 & 0 & 1 & 3 & 0 & 0 & \\ & 0 & 0 & 0 & 3 & 1 & 0 & \\ & 0 & 0 & 0 & 1 & 3 & 0 & \\ & & & \vdots & & & \end{bmatrix} \begin{bmatrix} \vdots \\ P_{i-2}^{k} \\ P_{i-1}^{k} \\ P_{i}^{k} \\ P_{i+1}^{k} \\ P_{i+2}^{k} \\ P_{i+3}^{k} \\ \vdots \end{bmatrix}$$

Notation

- The standard notation compresses the scheme to a *kernel*:
  - $h = (1/4)[\ldots,0,0,\boxed{1},\boxed{3},\boxed{3},\boxed{1},0,0,\ldots]$
- The kernel interlaces the odd and even rules.
- It also makes matrix analysis possible: eigenanalysis of the matrix form can be used to prove the continuity of the subdivision limit surface.
  - The details of analysis are fascinating, lengthy, and sadly beyond the scope of this course
- The limit curve of Chaikin is a quadratic B-spline!

Reading the kernel

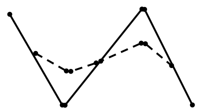
Consider the kernel

h=(1/8)[…,0,0,1,4,6,4,1,0,0,…]

You would read this as

$$P_{2i}^{k+1} = (1/8)(P_{i-1}^k + 6P_i^k + P_{i+1}^k)$$

$$P_{2i+1}^{k+1} = (1/8)(4P_i^k + 4P_{i+1}^k)$$



The limit curve is provably C2-continuous.

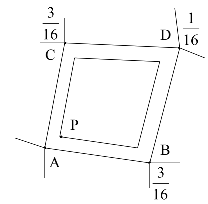
Making the jump to 3D: Doo-Sabin

Doo-Sabin takes Chaikin to 3D:

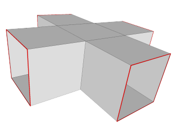
P =(9/16) A +
(3/16) B +
(3/16) C +
(1/16) D

This replaces every old vertex with four new vertices.

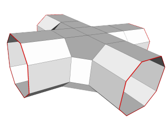
The limit surface is biquadratic, C1 continuous everywhere.



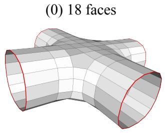
Doo-Sabin in action



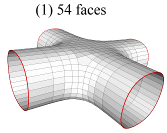
(0) 18 faces



(1) 54 faces



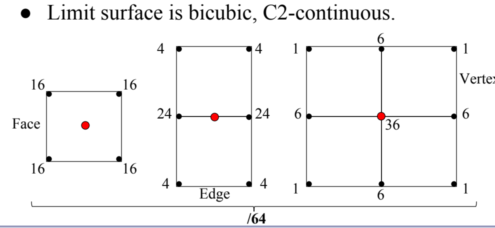
(2) 190 faces



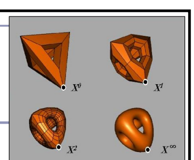
(3) 702 faces

Catmull-Clark

- *Catmull-Clark* is a bivariate approximating scheme with kernel h=(1/8)[1,4,6,4,1].
  - Limit surface is bicubic, C2-continuous.



Catmull-Clark

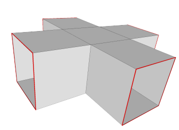


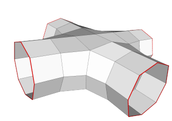
Getting tensor again:

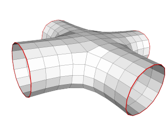
$$\frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix} \otimes \frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix} = \frac{1}{64}\begin{bmatrix}\boxed{1} & 4 & \boxed{6} & 4 & \boxed{1}\\ \boxed{4} & \boxed{16} & \boxed{24} & \boxed{16} & \boxed{4}\\ \boxed{6} & 24 & \boxed{36} & 24 & \boxed{6}\\ \boxed{4} & \boxed{16} & \boxed{24} & \boxed{16} & \boxed{4}\\ \boxed{1} & 4 & \boxed{6} & 4 & \boxed{1}\end{bmatrix}$$

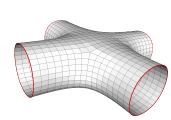
Vertex rule Face rule Edge rule

Catmull-Clark in action

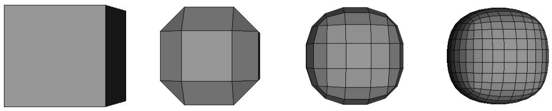




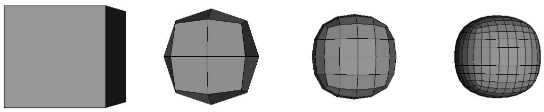




Catmull-Clark vs Doo-Sabin



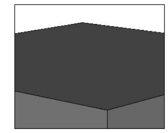
Doo-Sabin

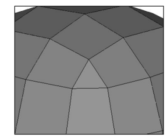


Catmull-Clark

Extraordinary vertices

- Catmull-Clark and Doo-Sabin both operate on quadrilateral meshes.
  - All faces have four boundary edges
  - All vertices have four incident edges
- What happens when the mesh contains *extraordinary* vertices or faces?
  - For many schemes, adaptive weights exist which can continue to guarantee at least some (non-zero) degree of continuity, but not always the best possible.
- CC replaces extraordinary faces with extraordinary vertices; DS replaces extraordinary vertices with extraordinary faces.





Detail of Doo-Sabin at cube corner

Extraordinary vertices: Catmull-Clark

Catmull-Clark vertex rules generalized for extraordinary vertices:

- Original vertex:
  $(4n-7) / 4n$
- Immediate neighbors in the one-ring:
  $3/2n^2$
- Interleaved neighbors in the one-ring:
  $1/4n^2$

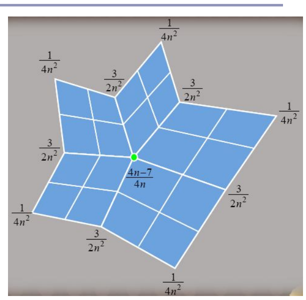
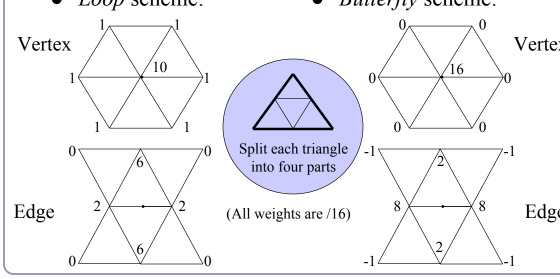


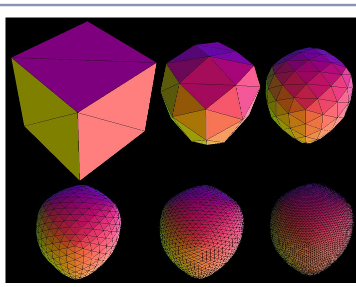
Image source: "*Next-Generation Rendering of Subdivision Surfaces*", Ignacio Castaño, SIGGRAPH 2008

Schemes for simplicial (triangular) meshes

- *Loop* scheme:
- *Butterfly* scheme:



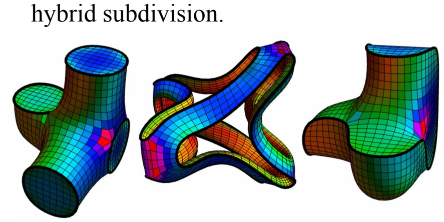
Loop subdivision



Loop subdivision in action. The asymmetry is due to the choice of face diagonals.
Image by Matt Fisher, http://www.its.caltech.edu/~matthewf/Chatter/Subdivision.html

Creases

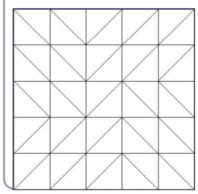
Extensions exist for most schemes to support *creases*, vertices and edges flagged for partial or hybrid subdivision.

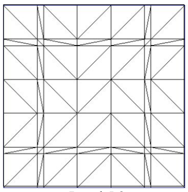


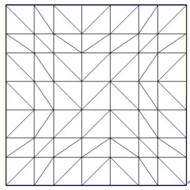
Still from "Volume Enclosed by Subdivision Surfaces with Sharp Creases" by Jan Hakenberg, Ulrich Reif, Scott Schaefer, Joe Warren http://vixra.org/pdf/1406.0060v1.pdf

Continuous level of detail

For live applications (e.g. games) can compute *continuous* level of detail, e.g. as a function of distance:


Level 5


Level 5.2

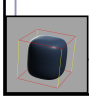

Level 5.8

Direct evaluation of the limit surface

- In the 1999 paper *Exact Evaluation Of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values*, Jos Stam (now at Alias|Wavefront) describes a method for finding the exact final positions of the CC limit surface.
  - His method is based on calculating the tangent and normal vectors to the limit surface and then shifting the control points out to their final positions.
  - What's particularly clever is that he gives exact evaluation at the extraordinary vertices. (Non-trivial.)

Bounding boxes and convex hulls for subdivision surfaces

- The limit surface is (the weighted average of (the weighted averages of (the weighted averages of (repeat for eternity…)))) the original control points.
- This implies that for any scheme where all weights are positive and sum to one, the limit surface lies entirely within the convex hull of the original control points.
- For schemes with negative weights:
  - Let $L=max_t \Sigma_i |N_i(t)|$ be the greatest sum throughout parameter space of the absolute values of the weights.
  - For a scheme with negative weights, L will exceed 1.
  - Then the limit surface must lie within the convex hull of the original control points, expanded unilaterally by a ratio of (L-1).



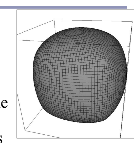
Splitting a subdivision surface

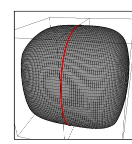
Many algorithms rely on subdividing a surface and examining the bounding boxes of smaller facets.

- Rendering, ray/surface intersections…

It's not enough just to delete half your control points: the limit surface will change (see right)

- Need to include all control points from the previous generation, which influence the limit surface in this smaller part.

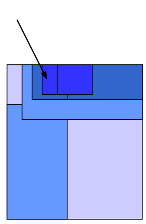




(Top) 5x Catmull-Clark subdivision of a cube
(Bottom) 5x Catmull-Clark subdivision of two halves of a cube; the limit surfaces are clearly different.

Ray/surface intersection

- To intersect a ray with a subdivision surface, we recursively split and split again, discarding all portions of the surface whose bounding boxes / convex hulls do not lie on the line of the ray.
- Any subsection of the surface which is 'close enough' to flat is treated as planar and the ray/plane intersection test is used.
- This is essentially a binary tree search for the nearest point of intersection.
  - You can optimize by sorting your list of subsurfaces in increasing order of distance from the origin of the ray.

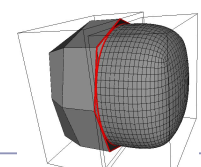


Rendering subdivision surfaces

- The algorithm to render any subdivision surface is exactly the same as for Bezier curves:

  "If the surface is simple enough, render it directly; otherwise split it and recurse."
- One fast test for "simple enough" is,

  "Is the convex hull of the limit surface sufficiently close to flat?"
- Caveat: splitting a surface and subdividing one half but not the other can lead to tears where the different resolutions meet. →



Rendering subdivision surfaces on the GPU

- Subdivision algorithms have been ported to the GPU, often using *geometry shaders*.
  - This subdivision can be done completely independently of geometry, imposing no demands on the CPU.
  - Uses a complex blend of precalculated weights and shader logic
  - Impressive effects in use at id, Valve, et al

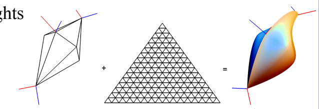


Figure from *Generic Mesh Renement on GPU,*
Tamy Boubekeur & Christophe Schlick (2005)
LaBRI INRIA CNRS University of Bordeaux, France

Subdivision Schemes—A partial list

- Approximating
  - Quadrilateral
    - (1/2)[1,2,1]
    - (1/4)[1,3,3,1] (Doo-Sabin)
    - (1/8)[1,4,6,4,1] (Catmull-Clark)
    - *Mid-Edge*
  - Triangles
    - Loop
- Interpolating
  - Quadrilateral
    - *Kobbelt*
  - Triangle
    - Butterfly
    - "√3" *Subdivision*

Many more exist, some much more complex

This is a major topic of ongoing research

References

Catmull, E., and J. Clark. "Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes." *Computer Aided Design*, 1978.

Dyn, N., J. A. Gregory, and D. A. Levin. "Butterfly Subdivision Scheme for Surface Interpolation with Tension Control." *ACM Transactions on Graphics.* Vol. 9, No. 2 (April 1990): pp. 160–169.

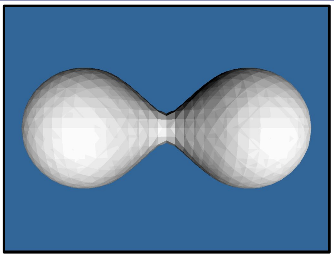
Halstead, M., M. Kass, and T. DeRose. "Efficient, Fair Interpolation Using Catmull-Clark Surfaces." *Siggraph '93*. p. 35.

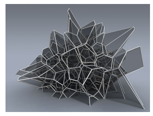
Zorin, D. "Stationary Subdivision and Multiresolution Surface Representations." Ph.D. diss., California Institute of Technology, 1997

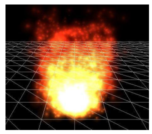
Ignacio Castano, "Next-Generation Rendering of Subdivision Surfaces." Siggraph '08, http://developer.nvidia.com/object/siggraph-2008-Subdiv.html

Dennis Zorin's SIGGRAPH course, "Subdivision for Modeling and Animation", http://www.mrl.nyu.edu/publications/subdiv-course2000/

Advanced Graphics







Implicit Surfaces, Voxels, Voronoi Diagrams and Particle Systems

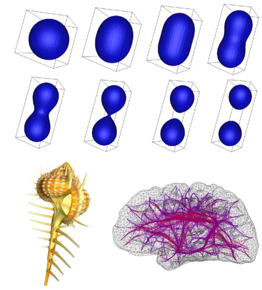
Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

Implicit surfaces

Implicit surface modeling(1) is a way to produce very 'organic' or 'bulbous' surfaces very quickly without subdivision or NURBS.

Uses of implicit surface modelling:

- Organic forms and nonlinear shapes
- Scientific modeling (electron orbitals, gravity shells in space, some medical imaging)
- Muscles and joints with skin
- Rapid prototyping
- CAD/CAM solid geometry

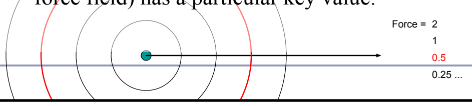


(1) AKA "*metaball modeling*", "*force functions*", "*blobby modeling*"…

How it works

The user controls a set of *control points*, like NURBS; each point in space generates a field of force, which drops off as a function of distance from the point (like gravity weakening with distance.)

This 3D field of forces defines an *implicit surface*: the set of all the points in space where some mathematical function (in this case, the value of the force field) has a particular key value.

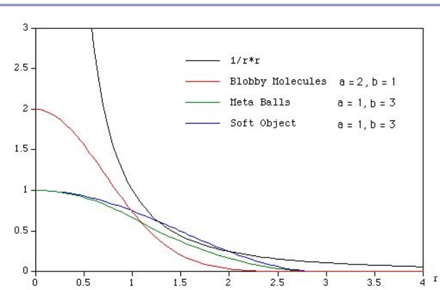


Force functions

A few popular force field functions:

- "Blobby Molecules" – Jim Blinn
  $F(r) = a\, e^{-br^2}$
- "Metaballs" – Jim Blinn
  $$F(r) = \begin{cases} a(1 - 3r^2 / b^2) & 0 \le r < b/3 \\ (3a/2)(1-r/b)^2 & b/3 \le r < b \\ 0 & b \le r \end{cases}$$
- "Soft Objects" – Wyvill & Wyvill
  $F(r) = a(1 - 4r^6/9b^6 + 17r^4/9b^4 - 22r^2 / 9b^2)$

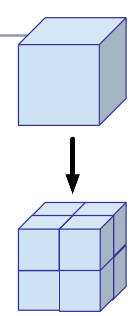
Comparison of force functions



Discovering the surface

An *octree* is a recursive subdivision of space which "homes in" on the surface, from larger to finer detail.

- An octree encloses a cubical volume in space. You evaluate the force function F(*v*) at each vertex *v* of the cube.
- As the octree subdivides and splits into smaller octrees, only the octrees which contain some of the surface are processed; empty octrees are discarded.



Polygonizing the surface

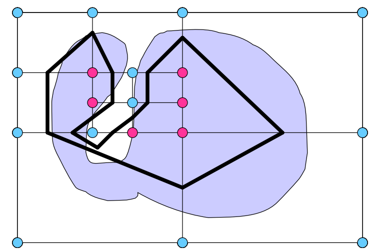
To display a set of octrees, convert the octrees into polygons.

- If some corners are “hot” (above the force limit) and others are “cold” (below the force limit) then the implicit surface crosses the cube edges in between.
- The set of midpoints of adjacent crossed edges forms one or more rings, which can be triangulated. The normal is known from the hot/cold direction on the edges.

To refine the polygonization, subdivide recursively; discard any child whose vertices are all hot or all cold.

Polygonizing the surface

Recursive subdivision (on a quadtree):

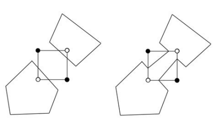


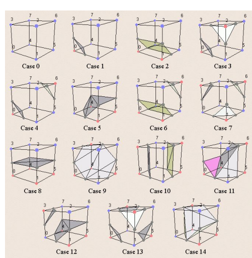
Polygonizing the surface

There are fifteen possible configurations (up to symmetry) of hot/cold vertices in the cube. →

- With rotations, that’s 256 cases.

Beware: there are *ambiguous cases* in the polygonization which must be addressed separately. ↓





Images courtesy of Diane Lingrand

Polygonizing the surface

One way to overcome the ambiguities that arise from the cube is to decompose the cube into tretrahedra.

- A common decomposition is into five tetrahedra. →
- Caveat: need to flip every other cube. (Why?)
- Can also split into six.

Another way is to do the subdivision itself on tetrahedra—no cubes at all.

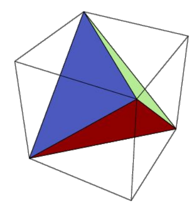


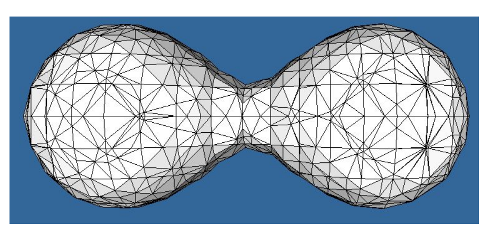
Image from the Open Problem Garden

Smoothing the surface

Improved edge vertices

- The naïve implementation builds polygons whose vertices are the midpoints of the edges which lie between hot and cold vertices.
- The vertices of the implicit surface can be more closely approximated by points linearly interpolated along the edges of the cube by the weights of the relative values of the force function.
  - t = (0.5 - F(P1)) / (F(P2) - F(P1))
  - P = P1 + t (P2 - P1)

Implicit surfaces -- demo



Marching cubes

An alternative to octrees if you only want to compute the final stage is the *marching cubes* algorithm (Lorensen & Cline, 1985):

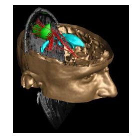
- Fire a ray from any point known to be inside the surface.
- Using Newton's method or binary search, find where the ray crosses the surface.
  - Newton: derivative estimated from discrete local sampling
  - There may be many crossings
- Drop a cube around the intersection point: it will have some vertices hot, some cold.
- While there exists a cube which has at least one hot vertex and at least one cold vertex on a side and no neighbor on that side, create a neighboring cube on that side. Repeat.

Marching cubes is common in medical imaging such as MRI scans. It was first demonstrated (and patented!) by researchers at GE in 1984, modeling a human spine.

Voxels and volume rendering

A *voxel* ("volume pixel") is a cube in space with a given color; like a 3D pixel.

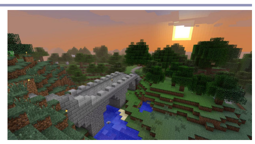
- Voxels are often used for medical imaging, terrain, scanning and model reconstruction, and other very large datasets.
- Voxels usually contain color but could contain other data as well—flow rates (in medical imaging), density functions (analogous to implicit surface modeling), lighting data, surface normals, 3D texture coordinates, etc.
- Often the goal is to render the voxel data directly, not to polygonize it.

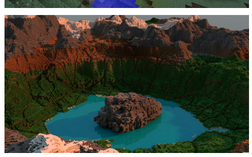


Voxels for deformable geometry

Voxels are uniquely well-suited to large-scale, dynamically deformable environments.

Geometry stored in a recursive data structure ("chunks", arrays of cubes containing arrays of cubes) can be locally edited in real time.



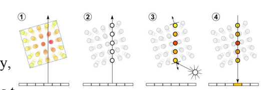


Fan art from the game Minecraft (from Deviantart.com, Wallchan.com)

Volume ray casting

If speed can be sacrificed for accuracy, render voxels with *volume ray casting*:

- Fire a ray through each pixel;
- Sample the voxel data along the ray, computing the weighted average (*trilinear* filter) of the contributions to the ray of each voxel it passes through;
- Compute surface gradient from of each voxel from local sampling; generate surface normals; compute lighting with the standard lighting equation;
- 'Paint' the ray from back to front, occluding more distant voxels with nearer voxels; this is the Painter's Algorithm for hidden-surface removal.

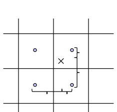


Top: the steps of volume rendering
Bottom: a volume ray-cast skull.
Images from wikipedia.

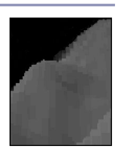
Sampling in voxel rendering

Why trilinear filtering?

- If we just show the color of the voxel we hit, we'll see the exact edges of every cube.
- Instead, choose the weighted average between adjacent voxels.
  - Trilinear: averaging across X, Y, and Z



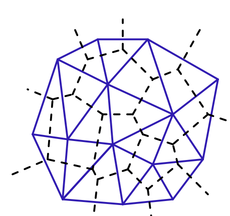
Your sample will fall somewhere between eight (in 3d) voxel centers. Weight the color of the sample by the inverse of its distance from the center of each voxel.



Voronoi diagrams

The *Voronoi diagram*[2] of a set of points P_i divides space into 'cells', where each cell C_i contains the points in space closer to P_i than any other P_j.

The *Delaunay triangulation* is the dual of the Voronoi diagram: a graph in which an edge connects every P_i which share a common edge in the Voronoi diagram.



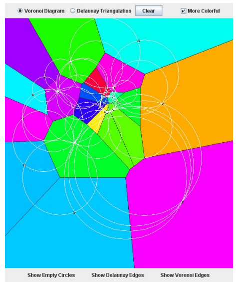
A Voronoi diagram (dotted lines) and its dual Delaunay triangulation (solid).

[2] AKA "Voronoi tesselation", "Dirichelet domain", "Thiessen polygons", "plesiohedra", "fundamental areas", "domain of action"…

Voronoi diagrams

Given a set $S=\{p_1,p_2,\ldots,p_n\}$, the formal definition of a Voronoi cell $C(S,p_i)$ is
$C(S,p_i)=\{p \in R^d \mid |p-p_i|<|p-p_j|, i \neq j\}$
The p_i are called the *generating points* of the diagram.

Where three or more boundary edges meet is a *Voronoi point*. Each Voronoi point is at the center of a circle (or sphere, or hypersphere…) which passes through the associated generating points and which is guaranteed to be empty of all other generating points.



Delaunay triangulation applet by Paul Chew ©1997—2007
http://www.cs.cornell.edu/home/chew/Delaunay.html

Delaunay triangulations and *equi-angularity*

The *equiangularity* of any triangulation of a set of points S is a sorted list of the angles ($\alpha_1\ldots\alpha_{3t}$) of the triangles.

- A triangulation is said to be *equiangular* if it possesses lexicographically largest equiangularity amongst all possible triangulations of S.
- The Delaunay triangulation is equiangular.

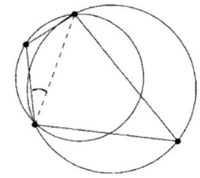


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

Delaunay triangulations and *empty circles*

Voronoi triangulations have the *empty circle* property*:* in any Voronoi triangulation of S, no point of S will lie inside the circle circumscribing any three points sharing a triangle in the Voronoi diagram.

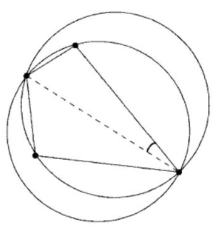


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

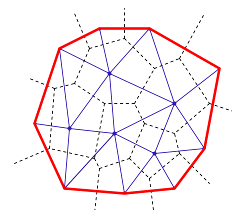
Delaunay triangulations and convex hulls

The border of the Delaunay triangulation of a set of points is always convex.

- This is true in 2D, 3D, 4D…

The Delaunay triangulation of a set of points in R^n is the planar projection of a convex hull in R^{n+1}.

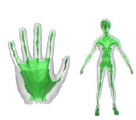
- Ex: from 2D ($P_i=\{x,y\}_i$), loft the points upwards, onto a parabola in 3D ($P'_i=\{x,y,x^2+y^2\}_i$). The resulting polyhedral mesh will still be convex in 3D.



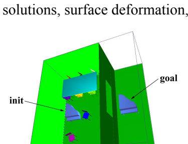
Voronoi diagrams and the *medial axis*

The *medial axis* of a surface is the set of all points within the surface equidistant to the two or more nearest points on the surface.

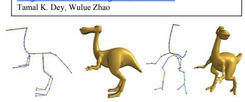
- This can be used to extract a skeleton of the surface, for (for example) path-planning solutions, surface deformation, and animation.



Approximating the Medial Axis from the Voronoi Diagram with a Convergence Guarantee
Tamal K. Dey, Wulue Zhao



A Voronoi-Based Hybrid Motion Planner for Rigid Bodies
M Foskey, M Garber, M Lin, DManocha

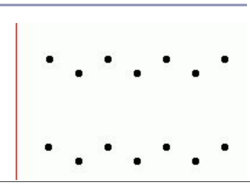


Shape Deformation using a Skeleton to Drive Simplex Transformations
IEEE Transaction on Visualization and Computer Graphics, Vol. 14, No. 3, May/June 2008, Page 693-706
Han-Bing Yan, Shi-Min Hu, Ralph R Martin, and Yong-Liang Yang

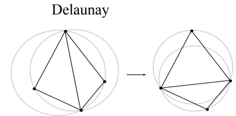
Finding the Voronoi diagram

There are four general classes of algorithm for computing the Delaunay triangulation:

- Divide-and-conquer
- Sweep plane
  - Ex: Fortune's algorithm →
- Incremental insertion
- "Flipping": repairing an existing triangulation until it becomes Delaunay



Fortune's Algorithm for the plane-sweep construction of the Voronoi diagram (Steve Fortune, 1986.)



This triangulation fails the circumcircle definition; we flip its inner edge and it becomes Delaunay. *(Image from Wikipedia.)*

Fortune's algorithm

1. The algorithm maintains a sweep line and a "beach line", a set of parabolas advancing left-to-right from each point. The beach line is the union of these parabolas.
   a. The intersection of each pair of parabolas is an edge of the voronoi diagram
   b. All data to the left of the beach line is "known"; nothing to the right can change it
   c. The beach line is stored in a binary tree
2. Maintain a queue of two classes of event: the addition of, or removal of, a parabola
3. There are O(n) such events, so Fortune's algorithm is O(n log n)

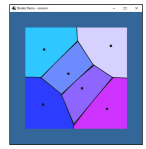
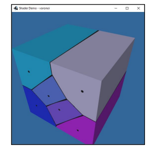
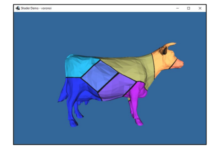
GPU-accelerated Voronoi Diagrams

Brute force:

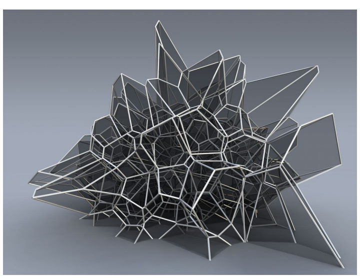
- For each pixel to be rendered on the GPU, search all points for the nearest point

Elegant:

- Render each point as a discrete cone in isometric projection, let z-buffering sort it out

Voronoi cells in 3D



Silvan Oesterle, Michael Knauss

Particle systems

Particle systems are a monte-carlo style technique which uses thousands (or millions) or tiny graphical artefacts to create large-scale visual effects.

Particle systems are used for hair, fire, smoke, water, spores, clouds, explosions, energy glows, in-game special effects and much more.

The basic idea:
"If lots of little things all do something the same way, our brains will see the thing they do and not the dots doing it."

Fallout 4 (Bethesda)

Command and Conquer 3 (Electronic Arts)

History of particle systems

- *1962*: Ships explode into pixel clouds in "Spacewar!", the 2nd video game *ever*.
- *1978*: Ships explode into broken lines in "Asteroid".
- *1982*: The Genesis Effect in "*Star Trek II: The Wrath of Khan*".


Fanboy note: You can play the original *Spacewar* at http://spacewar.oversigma.com/ -- the actual original game,running in a PDP-1 emulated in a Java applet.

Particle systems

How it works:

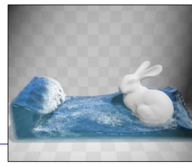
- Particles are generated an emitting source
  - Emitter position and orientation are specified discretely;
  - Emitter rate, direction, flow, etc are often specified as a bounded random range
  - This gives a Monte Carlo integration-style effect
- Time ticks; at each tick, particles move.
  - New particles are generated; expired particles are deleted
  - Forces (gravity, wind, etc) accelerate each particle
  - Acceleration changes velocity
  - Velocity changes position
- Particles are textured and rendered.


Transient vs persistent particles emitted to create a 'hair' effect (source: Wikipedia)

Particle systems—implementations

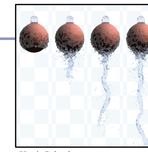
Closed-form function:
- Represent every particle as a parametric equation; store only the initial position p_0, initial velocity v_0, then apply fixed acceleration (such as gravity g.)
  - $p(t)=p_0+v_0t+½gt^2$
- No storage of state → small memory footprint
- *Very* limited possibility of interaction
- Best for fire, projectiles, etc—non-responsive particles.

Discrete integration:
- Update every particle separately; this can be expressed as a loop over a list, or as a mutation of a texture (if using a GPU), or as a massive matrix multiplication operation (if using CUDA)



NVIDIA

Particle systems—rendering



Hagit Schechter
http://www.cs.ubc.ca/~hagitsch/Research/

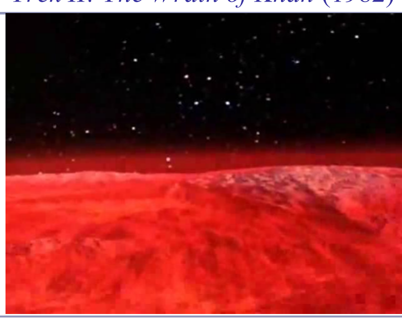
Can render particles as points, textured polys, or primitive geometry
- Minimize the data sent down the pipe!
- Polygons with alpha-blended images make pretty good fire, smoke, etc

Transitioning one particle type to another creates realistic interactive effects
- Ex: a 'rain' particle becomes an emitter for 'splash' particles on impact

Particles can be the force sources for a blobby model implicit surface
- Nice for simulating liquids

nvidia

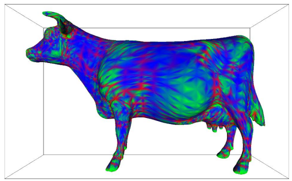
"The Genesis Effect" – William Reeves
Star Trek II: The Wrath of Khan (1982)



References

Implicit modelling:
D. Ricci, *A Constructive Geometry for Computer Graphics,* Computer Journal, May 1973
J Bloomenthal, *Polygonization of Implicit Surfaces,* Computer Aided Geometric Design, Issue 5, 1988
B Wyvill, C McPheeters, G Wyvill, *Soft Objects*, Advanced Computer Graphics (Proc. CG Tokyo 1986)
B Wyvill, C McPheeters, G Wyvill, *Animating Soft Objects,* The Visual Computer, Issue 4 1986
http://astronomy.swin.edu.au/~pbourke/modelling/implicitsurf/
http://www.cs.berkeley.edu/~job/Papers/turk-2002-MIS.pdf
http://www.unchainedgeometry.com/jbloom/papers/interactive.pdf
http://www-courses.cs.uiuc.edu/~cs319/polygonization.pdf
Voxels:
J. Wilhelms and A. Van Gelder, *A Coherent Projection Approach for Direct Volume Rendering*, Computer Graphics, 35(4):275-284, July 1991.
Voronoi diagrams
M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, "Computational Geometry: Algorithms and Applications", Springer-Verlag, http://www.cs.uu.nl/geobook/
http://www.ics.uci.edu/~eppstein/junkyard/nn.html
http://www.iquilezles.org/www/articles/voronoilines/voronoilines.htm // Voronois on GPU
Particle Systems:
William T. Reeves, "*Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*", Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83).
Lutz Latta, *Building a Million Particle System*, http://www.2ld.de/gdc2004/MegaParticlesPaper.pdf , 2004
http://www.darwin3d.com/gamedev/articles/col0798.pdf
http://mmacklin.com/pbf_sig_preprint.pdf

Advanced Graphics



"Nobody expects the geometric inquisition"

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Supported in part by Google UK, Ltd

Querying your geometry

Given a polygonal model, how might you find…

- the normal at each vertex?
- the curvature at each vertex?
- the convex hull?
- the bounding box?
- the center of mass?

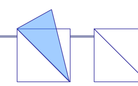
Querying your geometry

"Here's some geometry. What can we know?"

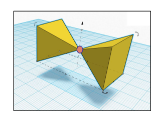
- A recurring theme here will be,
  "The polygons are not the shape: the polygons *approximate* the *surface* of the shape."
- Some questions from we could ask (e.g. ray-polygon intersection) are about the actual polygons.
- But other questions, like the normal at a vertex, are really about approximating the underlying surface as closely as possible.

Terminology

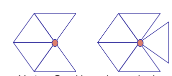
- We'll be focusing on *discrete* (as opposed to continuous) representation of geometry; i.e., polygon meshes
  - Many rendering systems limit themselves to triangle meshes
  - Many require that the mesh be *manifold*
- In a *closed manifold* polygon mesh:
  - Exactly two triangles meet at each edge
  - The faces meeting at each vertex belong to a single, connected loop of faces
- In a *manifold with boundary*:
  - At most two triangles meet at each edge
  - The faces meeting at each vertex belong to a single, connected strip of faces



Edge: Non-manifold vs manifold



Non-manifold vertex

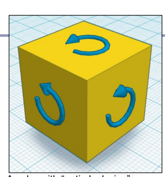


Vertex: Good boundary vs bad

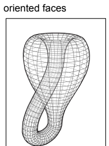
This slide draws much inspiration from Shirley and Marschner's *Fundamentals of Computer Graphics*, pp. 262-263

Terminology

- We say that a surface is *oriented* if:
  a. the vertices of every face are stored in a fixed order
  b. if vertices *i*, *j* appear in both faces *f1* and *f2*, then the vertices appear in order *i*, *j* in one and *j*, *i* in the other
- We say that a surface is *embedded* if, informally, "nothing pokes through":
  a. No vertex, edge or face shares any point in space with any other vertex, edge or face except where dictated by the data structure of the polygon mesh
- A closed, embedded surface must separate 3-space into two parts: a bounded *interior* and an unbounded *exterior*.



A cube with "anti-clockwise" oriented faces



Klein bottle: not an embedded surface.

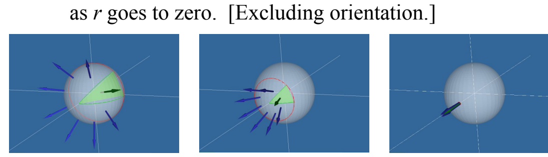
Also, terrible for holding drinks.

This slide draws much inspiration from Hughes and Van Dam's *Computer Graphics: Principles and Practice*, pp. 637-642

Normal at a vertex

Expressed as a limit,

The *normal of surface S at point P* is the limit of the cross-product between two (non-collinear) vectors from *P* to the set of points in *S* at a distance *r* from *P* as *r* goes to zero. [Excluding orientation.]



Normal at a vertex

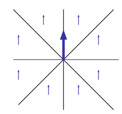
Using the limit definition, is the ‘normal’ to a discrete surface necessarily a vector?

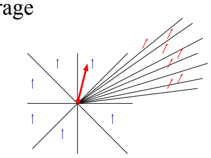
- The normal to the surface at any point on a face is a constant vector.
- The ‘normal’ to the surface at any edge is an arc swept out on a unit sphere between the two normals of the two faces.
- The ‘normal’ to the surface at a vertex is a space swept out on the unit sphere between the normals of all of the adjacent faces.

Finding the normal at a vertex

Method 1: Take the average of the normals of surrounding polygons

Problem: splitting one adjacent face into 10,000 shards would skew the average





Finding the normal at a vertex

Method 2: Take the weighted average of the normals of surrounding polygons, weighted by the area of each face

- 2a: Weight each face normal by the area of the face divided by the total number of vertices in the face

Problem: Introducing new edges into a neighboring face (and thereby reducing its area) should not change the normal.

Should making a face larger affect the normal to the surface near its corners?

- Argument for yes: If the vertices interpolate the ‘true’ surface, then stretching the surface at a distance could still change the local normals.

Finding the normal at a vertex

Method 3: Take the weighted average of the normals of surrounding polygons, weighted by each polygon’s *face angle* at the vertex

Face angle: the angle α formed at the vertex *v* by the vectors to the next and previous vertices in the face *F*

$$\alpha(F,v_i) = cos^{-1}\left(\frac{v_{i+1}-v_i}{|v_{i+1}-v_i|} \bullet \frac{v_{i-1}-v_i}{|v_{i-1}-v_i|}\right)$$

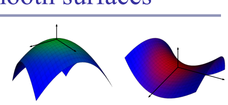
$$N(v) = \frac{\sum_F \alpha(F,v)\, N_F}{\left|\sum_F \alpha(F,v)\right|}$$

Note: In this equation, *arccos* implies a convex polygon. Why?

Gaussian curvature on smooth surfaces

Informally speaking, the *curvature* of a surface expresses “how flat the surface isn’t”.

- One can measure the directions in which the surface is curving *most*; these are the directions of *principal curvature*, k_1 and k_2.
- The product of k_1 and k_2 is the scalar *Gaussian curvature*.



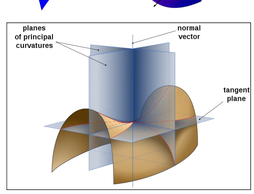
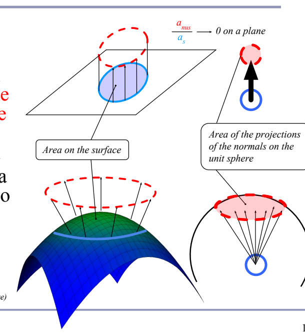


Image by Eric Gaba, from Wikipedia

Gaussian curvature on smooth surfaces

Formally, the *Gaussian curvature of a region* on a surface is the ratio between the area of the surface of the unit sphere swept out by the normals of that region and the area of the region itself.

The Gaussian curvature of a point is the limit of this ratio as the region tends to zero area.



$\frac{a_{nus}}{a_s} \longrightarrow r^2$ *on a sphere of radius r*
(please pretend that this is a sphere)

Gaussian curvature on discrete surfaces

On a discrete surface, normals do not vary smoothly: the normal to a face is constant on the face, and at edges and vertices the normal is—strictly speaking—undefined.

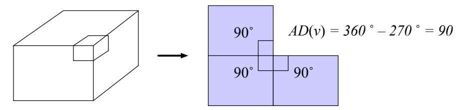
- Normals change instantaneously (as one's point of view travels across an edge from one face to another) or not at all (as one's point of view travels within a face.)

The Gaussian curvature of the surface of any polyhedral mesh is **zero** everywhere except at the vertices, where it is **infinite**.

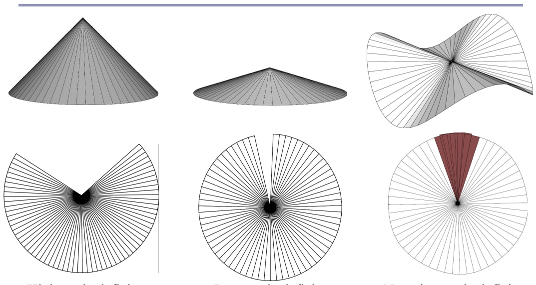
Angle deficit – a better solution for measuring discrete curvature

The *angle deficit AD(v)* of a vertex *v* is defined to be two π minus the sum of the face angles of the adjacent faces.

$$AD(v) = 2\pi - \sum_F \alpha(F,v)$$

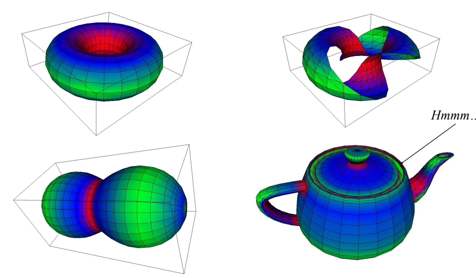


Angle deficit



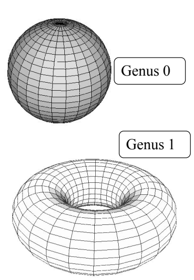
High angle deficit | Low angle deficit | Negative angle deficit

Angle deficit



Genus, Poincaré and the Euler Characteristic

- Formally, the *genus g* of a closed surface is
  - ..."a topologically invariant property of a surface defined as the largest number of nonintersecting simple closed curves that can be drawn on the surface without separating it."
    --*mathworld.com*
- Informally, it's the number of coffee cup handles in the surface.



Genus, Poincaré and the Euler Characteristic

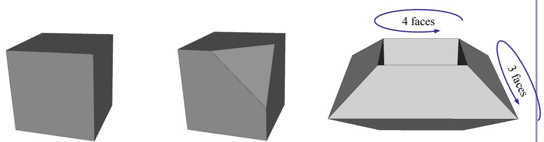
Given a polyhedral surface *S* without border where:

- V = the number of vertices of S,
- E = the number of edges between those vertices,
- F = the number of faces between those edges,
- χ is the *Euler Characteristic* of the surface,

the Poincaré Formula states that:

$$V - E + F = 2 - 2g = \chi$$

Genus, Poincaré and the Euler Characteristic



$g = 0$
$E = 12$
$F = 6$
$V = 8$
$V\text{-}E\text{+}F = 2\text{-}2g = 2$

$g = 0$
$E = 15$
$F = 7$
$V = 10$
$V\text{-}E\text{+}F = 2\text{-}2g = 2$

$g = 1$
$E = 24$
$F = 12$
$V = 12$
$V\text{-}E\text{+}F = 2\text{-}2g = 0$

The Euler Characteristic and angle deficit

Descartes' *Theorem of Total Angle Deficit* states that on a surface S with Euler characteristic χ, the sum of the angle deficits of the vertices is $2\pi\chi$:

$$\sum_S AD(v) = 2\pi\chi$$

Cube:
- $\chi = 2\text{-}2g = 2$
- $AD(v) = \pi/2$
- $8(\pi/2) = 4\pi = 2\pi\chi$

Tetrahedron:
- $\chi = 2\text{-}2g = 2$
- $AD(v) = \pi$
- $4(\pi) = 4\pi = 2\pi\chi$

Convex hull

The *convex hull* of a set of points is the unique surface of least area which contains the set.

- If a set of infinite half-planes have a finite non-empty intersection, then the surface of their intersection is a convex polyhedron.
- If a polyhedron is convex then for any two faces A and B in the polyhedron, all points in B which are not in A lie to the same side of the plane containing A.

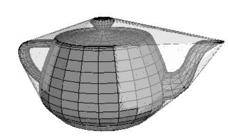
Every point on a convex hull has non-negative angle deficit.

The faces of a convex hull are always convex.

Finding the convex hull of a set of points

Method 1: For every triple of points in the set, define a plane P. If all other points in the set lie to the same side of P (dot-product test) then add P to the hull; else discard.

Problem 1: this works but it's $O(n^4)$.



Finding the convex hull of a set of points

Method 2:
- Initialize C with a tetrahedron from any four non-colinear points in the set. Orient the faces of C by taking the dot product of the center of each face with the average of the vertices of C.
- For each vertex v,
  - For each face f of C,
    - If the dot product of the normal of f with the vector from the center of f to v is positive then v is 'above' f.
    - If v is above f then delete f and update a (sorted) list of all new border vertices.
  - Create a new triangular face from v to each pair of border vertices.

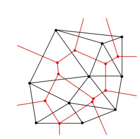
Problem 2:

This is $O(n^2)$ at best.

Finding the convex hull of a set of points

Method 3:

The exterior boundary of the union of the cells of the Delaunay triangulation of a set of points is its convex hull.



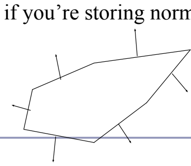
The exterior border of the Delaunay triangulation is the convex hull of the point set.

Algorithm:
- Find the Voronoi diagram of your point set
- Compute the Delaunay triangulation (2D) or tetrahedralization (3D)
- Delete all faces of the simplices which aren't on the exterior border

Testing if a point is inside a convex hull

We can generalize Method 2 to test whether a point is inside any convex polyhedron.

- For each face, test the dot product of the normal of the face with a vector from the face to the point. If the dot is ever positive, the point lies outside.
- The same logic applies if you're storing normals at vertices.

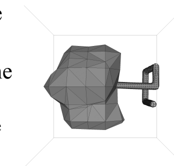


Centroids

The *centroid* of a surface is the center of mass of the volume enclosed by the surface.

This is *not* the same as the center of the bounding box.

- We'll assume that the 'material' within the surface is of uniform density.
- We'll also assume that we have a closed surface (without border.)

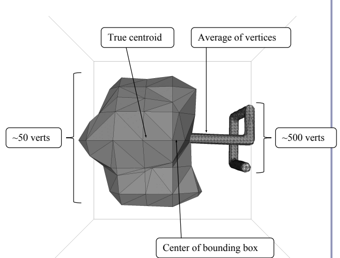


Centroids

Method 1: Take the average of all vertices.

C = $(\Sigma_{\{v\}}(v)) / \|\{v\}\|$

Problem 1: as with normals, an area of bizarre density would skew the average.

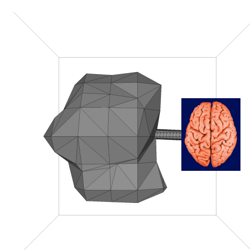


Centroids

Method 2: Take the average of the centers of the faces of the surface, weighting each by the area of the face.

- This method works well for convex polyhedra.

Problem 2: This is vulnerable to dense 'wrinkles' of many polygons packed into a small volume.



The average adult human brain has a surface area of approximately 2,500 cm^2, a volume of roughly 1200 cm^3, and weighs about 1400g. For comparison, a sphere of similar volume would have a surface area of 546 cm^2. Brain image courtesy of Moprhonix.com.

Centroids

Method 3a: Use "Monte Carlo" integration. Find the bounding box of the surface and then choose *billions* of points at random inside the box; take the average of all those points which fall inside the surface.

Problem 3a: Testing for 'inside' is time-consuming (although it can be accelerated; try BSP trees.) Also, this lacks precision. And, frankly, finesse.

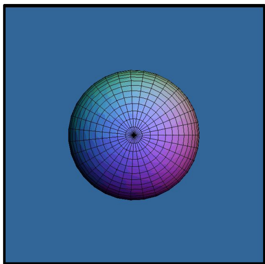
Method 3b: Decompose the polyhedron into convex polyhedra, then use method 2 to find the center of each. Average the centers, weighting each point by the volume of its convex polyhedron.

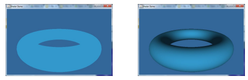
Problem 3b: Convex decomposition is solved, but it's not trivial.

- Convex regions decompose rapidly to tetrahedra.
- Nonconvex regions can be tricky: tetrahedra may cross.

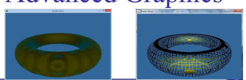
References

Gaussian Curvature
http://en.wikipedia.org/wiki/Gaussian_curvature
http://mathworld.wolfram.com/GaussianCurvature.html
The Poincaré Formula
http://mathworld.wolfram.com/PoincareFormula.html
Convex Hulls
Tim Lambert's Java demos: http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html
Wolfram: http://demonstrations.wolfram.com/ConvexHullAndDelaunayTriangulation/
Bounding volumes
http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/CG-Applets/Center/centercli.htm
M. Dyer and N. Megiddo, "Linear Programming in Low Dimensions." Ch. 38 in *Handbook of Discrete and Computational Geometry* (Ed. J. E. Goodman and J. O'Rourke). Boca Raton, FL: CRC Press, pp. 669-710, 1997.
J. O'Rourke, *Finding minimal enclosing boxes*, Springer Netherlands, 1985
Centroids
B. Mirtich, "Fast and Accurate Computation of Polyhedral Mass Properties", Journal of Graphics Tools v.1 n.2, 1996.
Kim et al, "Fast GPU Computation of the Mass Properties of a General Shape and its Application to Buoyancy Simulation", The Visual Computer v.2 n.9-11, 2006 (Adapts Mirtich's method to use GPU hardware acceleration)

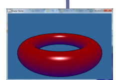


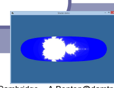


Advanced Graphics



OpenGL and Shaders





Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Supported in part by Google UK, Ltd

3D technologies today

Java
- Common, re-usable language; well-designed
- Steadily increasing popularity in industry
- Weak but evolving 3D support

C++
- Long-established language
- Long history with OpenGL
- Long history with DirectX
- Losing popularity in some fields (finance, web) but still strong in others (games, medical)

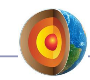
JavaScript
- WebGL is surprisingly popular

OpenGL
- Open source with many implementations
- Well-designed, old, and still evolving
- Fairly cross-platform

DirectX/Direct3d (Microsoft)
- Microsoft™ only
- Dependable updates

Mantle (AMD)
- Targeted at game developers
- AMD-specific

Higher-level commercial libraries
- RenderMan
- AutoDesk / SoftImage



OpenGL

OpenGL is…
- Hardware-independent
- Operating system independent
- Vendor neutral

On many platforms
- Great support on Windows, Mac, linux, etc
- Support for mobile devices with OpenGL ES
  - Android, iOS (but not Windows Phone)
  - Android Wear watches!
- Web support with WebGL

A state-based renderer
- many settings are configured before passing in data; rendering behavior is modified by existing state

Accelerates common 3D graphics operations
- Clipping (for primitives)
- Hidden-surface removal (Z-buffering)
- Texturing, alpha blending NURBS and other advanced primitives (GLUT)

OpenGL in Java

- *JOGL*: "Java bindings for OpenGL"

http://jogamp.org/jogl/

JOGL apps can be deployed as applications or as *applets*, making it suitable for educational web demos and cross-platform applications.
- If the user has installed the latest Java, of course.
- And if you jump through Oracle's authentication hoops.
- And… let's be honest, 1998 called, it wants its applets back.

- *LWJGL*: "Lightweight Java Games Library"

http://www.lwjgl.org/

LWJGL is targeted at game developers, so it's got a really solid threading model and good support for new input methods like joysticks, gaming mice, and the Oculus Rift.

JOGL shaders in action. Image from Wikipedia

OpenGL architecture

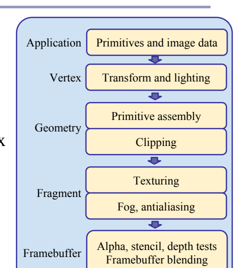
The CPU (your processor and friend) delivers data to the GPU (Graphical Processing Unit).
- The GPU takes in streams of vertices, colors, texture coordinates and other data; constructs polygons and other primitives; then uses *shaders* to draw the primitives to the screen pixel-by-pixel.
- The GPU processes the vertices according to the *state* set by the CPU; for example, "every trio of vertices describes a triangle".

This process is called the *rendering pipeline*. Implementing the rendering pipeline is a joint effort between you and the GPU.

You'll write shaders in the OpenGL shader language, GLSL.
You'll write *vertex* and *fragment* shaders. (And maybe others.)

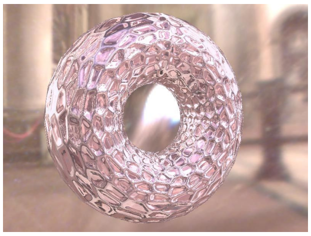
The OpenGL rendering pipeline

An OpenGL application assembles sets of *primitives*, *transforms* and *image data*, which it passes to OpenGL's GLSL shaders.
- *Vertex shaders* process every vertex in the primitives, computing info such as position of each one.
- *Fragment shaders* compute the color of every fragment of every pixel covered by every primitive.



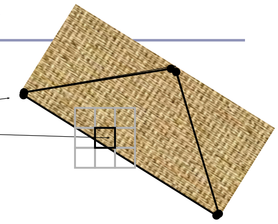
The OpenGL *rendering pipeline (simplified view)*

Shader gallery I



Above: Demo of Microsoft's XNA game platform
Right: Product demos by nvidia (top) and ATI (bottom)

What are we targeting?



OpenGL shaders give the user control over each *vertex* and each *fragment* (each pixel or partial pixel) interpolated between vertices.

After vertices are processed, polygons are *rasterized*. During rasterization, values like position, color, depth, and others are interpolated across the polygon. The interpolated values are passed to each pixel fragment.

Think parallel

Shaders are compiled from within your code
- They used to be written in assembler
- Today they're written in high-level languages

They execute on the GPU

GPUs typically have multiple processing units

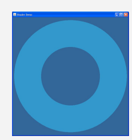
That means that multiple shaders execute in parallel
- We're moving away from the purely-linear flow of early "C" programming models

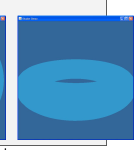
Shader example one – ambient lighting

```
#version 330

uniform mat4 mvp;

in vec4 vPosition;

void main() {
  gl_Position = mvp *
vPosition;
}
```

// Vertex Shader

```
#version 330

out vec4 fragmentColor;

void main() {
  fragmentColor =
      vec4(0.2, 0.6, 0.8, 1);
}
```





// Fragment Shader

GLSL

Notice the C-style syntax
`void main() { … }`

The vertex shader uses two inputs, one four-element `vec4` and one four-by-four `mat4` matrix; and one standard output, `gl_Position`.

The line
`gl_Position = mvp * gl_Vertex;`
applies our model-view-projection matrix to calculate the correct vertex position in perspective coordinates.

This fragment shader implements the most basic ambient lighting by setting its one output, `col`, to a fixed value.

GLSL

The language design in GLSL is strongly based on ANSI C, with some C++ added.
- There is a preprocessor--**#define**, etc
- Basic types: int, float, bool
  - No double-precision float
- Vectors and matrices are standard: **vec2**, **mat2** = 2x2; **vec3**, **mat3** = 3x3; **vec4**, **mat4** = 4x4
- Texture samplers: **sampler1D**, **sampler2D**, etc are used to sample multidemensional textures
- New instances are built with *constructors*, a la C++
- Functions can be declared before they are defined, and operator overloading is supported.

GLSL

Some differences from C/C++:

- No pointers, strings, chars; no unions, enums; no bytes, shorts, longs; no unsigned. No switch() statements.
- There is no implicit casting (type promotion):

```
float foo = 1;
```

fails because you can't implicitly cast **int** to **float**.

- Explicit type casts are done by constructor:

```
vec3 foo = vec3(1.0, 2.0, 3.0);
vec2 bar = vec2(foo);  // Drops foo.z
```

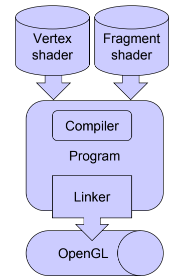
Function parameters are labeled as **in**, **out**, or **uniform**.

- Functions are called by *value-return*, meaning that values are copied into and out of parameters at the start and end of calls.

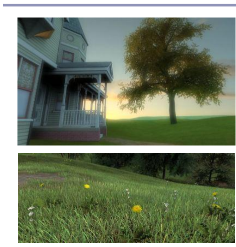
OpenGL / GLSL API - setup

To install and use a shader in OpenGL:

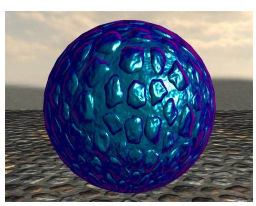
1. Create one or more empty *shader objects* with **glCreateShader**.
2. Load source code, in text, into the shader with **glShaderSource**.
3. Compile the shader with **glCompileShader**.
4. Create an empty *program object* with **glCreateProgram**.
5. Bind your shaders to the program with **glAttachShader**.
6. Link the program (ahh, the ghost of C!) with **glLinkProgram**.
7. Activate your program with **glUseProgram**.



Shader gallery II



Above: Kevin Boulanger (PhD thesis, "*Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*", 2005)



Above: Ben Cloward ("Car paint shader")

What will you have to write?

It's up to you to implement perspective and lighting.

1. Pass geometry to the GPU
2. Implement perspective on the GPU
3. Calculate lighting on the GPU

1. OpenGL / GLSL API - variables

GLSL shaders use named parameters which can be looked up from OpenGL.

GLSL:

```
uniform mat4 modelToScreen;
in vec4 vPosition;
...
```

The OpenGL API looks up the location integers of these parameters and uses the location as an address:

OpenGL:

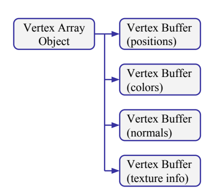
```
int attributeId = glGetAttribLocation(program,
"vPosition");
glEnableVertexAttribArray(attributeId);
glVertexAttribPointer(attributeId, ...);
```


Passing geometry to OpenGL

Vertex buffer objects store arrays of vertex data--positional or descriptive. With a vertex buffer object ("VBO") you can compute all vertices at once, pack them into a VBO, and pass them to OpenGL *en masse* to let the GPU processes all the vertices together.

To group different kinds of vertex data together, you can serialize your buffers into a single VBO, or you bind and attach them to a *Vertex Array Objects*. Each vertex array object ("VAO") can contain multiple VBOs.

Although not required, VAOs help you to organize and isolate the data in your VBOs.



Vertex arrays contain vertex buffers

First, we allocate a *vertex array*:

```
private void createAndBindVertexBuffer() {
  int vertexArrayId = glGenVertexArrays();
  glBindVertexArray(vertexArrayId);
}
```

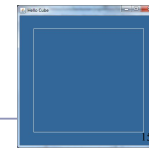
Then we fill attach a *vertex buffer* with vertex coordinates:

```
private void addVertexBuffer(String name, FloatBuffer data) {
  int BufferId = glGenBuffers();
  glBindBuffer(GL_ARRAY_BUFFER, bufferId);
  glBufferData(GL_ARRAY_BUFFER, data, GL_STATIC_DRAW);
  int attributeId = glGetAttribLocation(program, name);
  glEnableVertexAttribArray(attributeId);
  glVertexAttribPointer(attributeId, 3, GL11.GL_FLOAT, false, 0, 0);
}
```


Vertex buffers contain vertex data

In Java, vertex data is typically packed into a FloatBuffer:

```
static final float[][] CORNERS = {
  {-0.8f, 0.8f, 0.8f}, { 0.8f, 0.8f, 0.8f}, { 0.8f, 0.8f,-0.8f}, {-0.8f, 0.8f,-0.8f},
  {-0.8f,-0.8f, 0.8f}, { 0.8f,-0.8f, 0.8f}, { 0.8f,-0.8f,-0.8f}, {-0.8f,-0.8f,-0.8f},
};
static final int[] INDICES = { 0, 1, 2, 3, 0, 4, 5, 1, 5, 6, 2, 6, 7, 3, 7, 4 };
private void drawCube() {
  FloatBuffer vertices = Buffers.newDirectFloatBuffer(INDICES.length * 3);
  for (int index : INDICES) { vertices.put(CORNERS[index]); }
  vertices.rewind();
  fillCurrentVertexBuffer("vPosition", vertices);
  // ...
  glDrawArrays(GL_LINE_STRIP, 0, INDICES.length);
}
```



...and it's boring, because we have no 3D.

Binding multiple buffers in a VAO

Need more info? We can pass more than just *coordinate* data--we can create as many buffer objects as we want for different types of per-vertex data.

To bind two arrays of floats together, we build a *vertex array object* as before:

```
int vertexArrayId = glGenVertexArrays();
glBindVertexArray(vertexArrayId);
```

We bind a vertex buffer object for coordinate data, then another for normals:

```
addVertexBuffer("vPosition", vertices);
addVertexBuffer("vNormal", normals);
```

Later, to render, we'll unbind the buffers and work only with the vertex array:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glDrawArrays(GL_LINE_STRIP, 0, INDICES.length);
```


Memory management: Lifespan of an OpenGL object

Most objects in OpenGL are created and deleted explicitly. Because these entities live in the GPU, they're outside the scope of Java's garbage collection.

The typical creation and deletion of an OpenGL object look like this:

```
int createAndBindVBO() {
  int name = glGenBuffers();
  glBindBuffer(GL_ARRAY_BUFFER, name);
  return name;
}

// Work with your object

void deleteVBO(int vboName) {
  glDeleteBuffers(vboName);
}
```

2. Getting some perspective

To add *3D perspective* to our flat model, we face three challenges:

- Compute a 3D perspective matrix
- Pass it to OpenGL, and on to the GPU
- Apply it to each vertex

To do so we're going to need to apply our perspective matrix in the shader, which means we'll need to build our own 4x4 perspective transform.

4x4 perspective matrix transform

Every OpenGL package provides utilities to build a perspective matrix. You'll usually find a method named something like *glGetFrustum()* which will assemble a 4x4 grid of floats suitable for passing to OpenGL.

Or you can build your own:

$$P = \begin{pmatrix} \frac{1}{ar \cdot \tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-NearZ - FarZ}{NearZ - FarZ} & \frac{2 \cdot FarZ \cdot NearZ}{NearZ - FarZ} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

α: Field of view, typically 50°

ar: Aspect ratio of width over height

NearZ: Near clip plane

FarZ: Far clip plane

Passing uniform data to GLSL

The method glGetUniformLocation() will look up the location of a uniform parameter in a shader program.

(This is analogous to the attribute lookup seen earlier.)

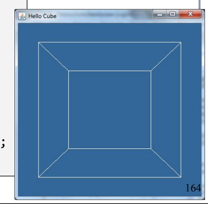
```
private void updateM4x4(String name, M4x4 T) {
  int uniform = glGetUniformLocation(program, name);
  if (uniform != -1) {
    glUniformMatrix4(uniform, false, T.asFloats());
  }
}
```


Reading uniform data in GLSL

Next we need to modify our shader to transform our vertices by our perspective matrix.

This shader takes a matrix and applies it to each vertex:

```
#version 330

uniform mat4 modelToScreen;

in vec4 vPosition;

void main() {
   gl_Position = modelToScreen * vPosition;
}
```



Multiple uniforms

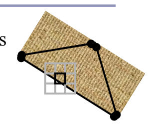
```
#version 330

uniform mat4 modelToScreen;
uniform mat4 modelToWorld;
uniform mat3 normalToWorld;

in vec4 vPosition;
in vec3 vNormal;

void main() {
  vec3 p = (modelToWorld * vPosition).xyz;
  vec3 n = normalize(normalToWorld * vNormal);
  // ...
```

Use multiple uniforms for different fields that are constant throughout the rendering pass, such as transform matrices and lighting coordinates.

3. Lighting and Shading

- Vertex shader outputs are interpolated across fragments.

This makes the implementation of classic illumination models like *Gouraud shading* very straightforward.



Diffuse lighting $d = k_D(N \cdot L)$ expressed as a shader

```
// ...
out vec4 color;
void main() {
  vec3 N = // ...
  vec3 L = // ...
  float diffuse = Kd * clamp(0, dot(N, L), 1);
  color = vec4(PURPLE * diffuse, 1.0);
}
```

// Vertex Shader

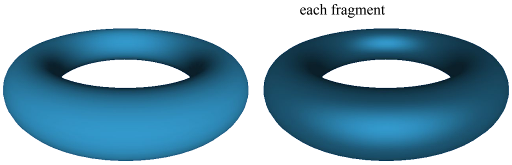
```
// ...
in vec4 color;
out vec4 fragmentColor;

void main() {
  fragmentColor = color;
}
```

// Fragment Shader

Gouraud and Phong

Gouraud shading
- Compute color in vertex shader
- Let OpenGL interpolate color across fragments
- Output interpolated color

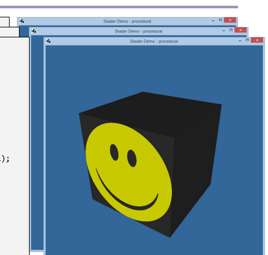
Phong shading
- Compute normal in vertex shader
- Let OpenGL interpolate normal across fragments
- Compute color separately for each fragment



GLSL includes handy helper methods for illumination, such as a reflect() method that reflects one vector across another--perfect for specular highlighting. For a few examples, check out the demo source code on github.

Procedural texturing in the fragment shader

```
// ...
const vec3 CENTER = vec3(0, 0, 1);
const vec3 LEFT_EYE = vec3(-0.2, 0.25, 0);
const vec3 RIGHT_EYE = vec3(0.2, 0.25, 0);
// ...

void main() {
  bool isOutsideFace = (length(position - CENTER) > 1);
  bool isEye = (length(position - LEFT_EYE) < 0.1)
      || (length(position - RIGHT_EYE) < 0.1);
  bool isMouth = (length(position - CENTER) < 0.75)
      && (position.y <= -0.1);

  vec3 color = (isMouth || isEye || isOutsideFace)
      ? BLACK : YELLOW;

  fragmentColor = vec4(color, 1.0);
}
```



(Code truncated for brevity--again, check out the source on github for how I did the curved mouth and oval eyes.)

Voronoi diagrams in the fragment shader

For a limited set of generating points, can compute the *Voronoi Diagram* in the fragment shader.

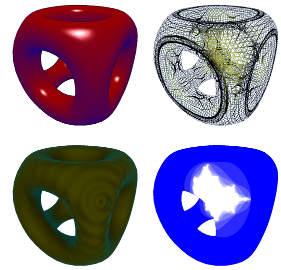
Simple version: "F2-F1": find the nearest two generating points by iteration, render the isolines where their forces = 0.

Better: With a two-pass solution, can generate the isolines *within* the cell as well (see link)

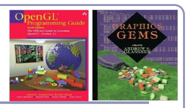
Iñigo Quilez (Pixar, Oculus)
http://www.iquilezles.org/www/articles/voronoilines/voronoilines.htm

More advanced surface effects

- Specular highlighting
- Non-photorealistic illumination
- Volumetric textures
- Bump-mapping
- Interactive surface effects
- Ray-casting in the shader
- Higher-order math in the shader
- ...much, much more!



Recommended reading



Course source code on Github -- many sample shaders (https://github.com/AlexBenton/AdvancedGraphics/tree/master/AdvGraph1415)

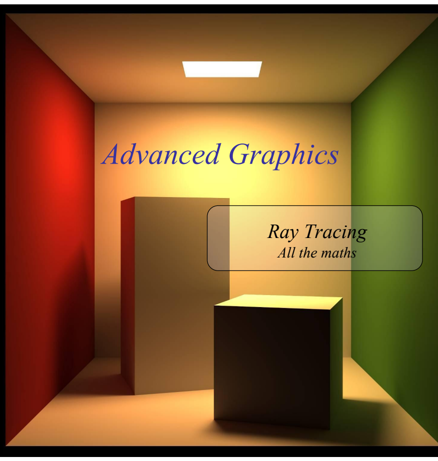
The OpenGL Programming Guide (2013), by Shreiner, Sellers, Kessenich and Licea-Kane
 Some also favor *The OpenGL Superbible* for code samples and demos
 There's also an OpenGL-ES reference, same series

OpenGL Insights (2012), by Cozzi and Riccio

OpenGL Shading Language (2009), by Rost, Licea-Kane, Ginsburg et al

The *Graphics Gems* series from Glassner

ShaderToy.com, a web site by Inigo Quilez (Pixar) dedicated to amazing shader tricks and raycast scenes



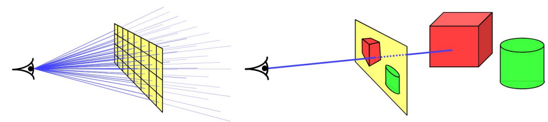
"Cornell Box" by Steven Parker, University of Utah. A tera-ray monte-carlo rendering of the Cornell Box, generated in 2 CPU years on an Origin 2000. The full image contains 2048 x 2048 pixels with over 100,000 primary rays per pixel (317 x 317 jittered samples). Over one trillion rays were traced in the generation of this image.

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

Ray tracing

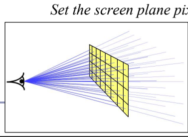
- A powerful alternative to polygon scan-conversion techniques
- An elegantly simple algorithm:

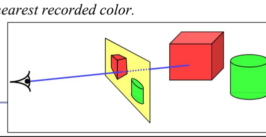
Given a set of 3D objects, shoot a ray from the eye through the center of every pixel and see what it hits.



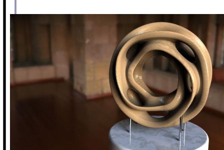
The algorithm

Select an eye point and a screen plane.
for (every pixel in the screen plane):
  Find the ray from the eye through the pixel's center.
  for (each object in the scene):
    if (the ray hits the object):
      if (the intersection is the nearest (so far) to the eye):
        Record the intersection point.
        Record the color of the object at that point.
  Set the screen plane pixel to the nearest recorded color.

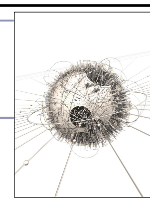




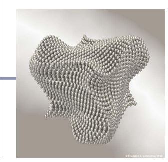
Examples



"Scherk-Collins sculpture" by Trevor G. Quayle (2008)



"POV Planet" by Casey Uhrig (2004)



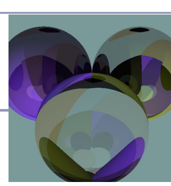
"Dancing Cube" by Friedrich A. Lohmueller (2003)

"Villarceau Circles" by Tor Olav Kristensen (2004)

'Glasses" by Gilles Tran (2006)

All images are from the POV-Ray Hall of Fame: hof.povray.org

It doesn't take much code



The basic algorithm is straightforward, but there's much room for subtlety

- Refraction
- Reflection
- Shadows
- Anti-aliasing
- Blurred edges
- Depth-of-field effects
- …

```
typedef struct{double x,y,z;}vec;vec U,black,amb={.02,.02,.02};
struct sphere{vec cen,color;double rad,kd,ks,kt,kl,ir;}*s,*best
,sph[]={0.,6.,.5,1.,1.,1.,.9,.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5
,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,1.,.3,.7,0.,0.,1.2,3
.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,1.,5.,0
.,0.,0.,.5,1.5,};int yx;double u,b,tmin,sqrt(),tan();double
vdot(vec A,vec B){return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(
double a,vec A,vec B){B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return
B;}vec vunit(vec A){return vcomb(1./sqrt(vdot(A,A)),A,black);}
struct sphere*intersect(vec P,vec D){best=0;tmin=10000;s=sph+5;
while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+
s->rad*s->rad,u=u>0?sqrt(u):10000,u=b-u>0.000001?b-u:b+u,tmin=
u>0.00001&&u<tmin?best=s,u:tmin;return best;}vec trace(int
level,vec P,vec D){double d,eta,e;vec N,color;struct sphere*s,
*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s->ir;d=-vdot(D,N=vunit(vcomb(-1.,P=vcomb(
tmin,D,P),s->cen)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d=
-d;l=sph+5;while(l-->sph)if((e=l->kl*vdot(N,U=vunit(vcomb(-1.,P
,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e,l->color,color);
U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*eta*(
1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(
eta*d-sqrt(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(
2*d,N,D)),vcomb(s->kd,color,vcomb(s->kl,U,black))));}main(){int
d=512;printf("%d %d\n",d,d);while(yx<d*d){U.x=yx%d-d/2;U.z=d/2-
yx++/d;U.y=d/2/tan(25/114.5915590261);U=vcomb(255.,trace(3,
black,vunit(U)),black);printf("%0.f %0.f %0.f\n",U.x,U.y,U.z);}
}/*minray!*/
```

Paul Heckbert's 'minray' ray tracer, which fit on the back of his business card. (circa 1983)

Running time

The ray tracing time for a scene is a function of
(num rays cast) x
(num lights) x
(num objects in scene) x
(num reflective surfaces) x
(num transparent surfaces) x
(num shadow rays) x
(ray reflection depth) x …

Image by nVidia

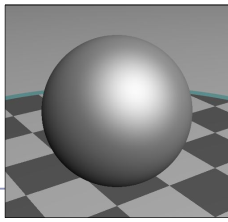
Contrast this to polygon rasterization: time is a function of the number of elements in the scene times the number of lights.

Recall: illumination

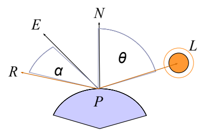
The *total illumination at P* is:

$I(P) = k_A + k_D(N \bullet L) + k_S(R \bullet E)^n$

summed over all lights L.



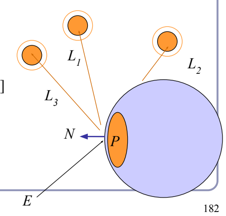
Ambient light: k_A
Diffuse light: $k_D(N \bullet L)$
Specular light: $k_S(R \bullet E)^n$
where R = L - 2(L•N)N



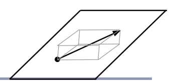
Ray-traced illumination

Once you have the point P (the intersection of the ray with the nearest object) you'll compute how much each of the lights in the scene illuminates P.

diffuse = 0
specular = 0
for (each light L_i in the scene):
  if $(N \bullet L) > 0$:
  [Optionally: if (a ray from P to L_i can reach L_i):]
    diffuse += $k_D(N \bullet L)$
    specular += $k_S(R \bullet E)^n$
intensity at P = ambient + diffuse + specular



Hitting things with rays



A ray is defined parametrically as

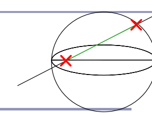
$P(t) = E + tD,\ t \geq 0$ (α)

where E is the ray's origin (our eye position) and D is the ray's direction, a unit-length vector.

We expand this equation to three dimensions, x, y and z:

$$\left.\begin{array}{l} x(t) = x_E + tx_D \\ y(t) = y_E + ty_D \\ z(t) = z_E + tz_D \end{array}\right\}\ t \geq 0 \qquad (\beta)$$

Hitting things with rays: Sphere



The unit sphere, centered at the origin, has the implicit equation

$x^2 + y^2 + z^2 = 1$ (γ)

Substituting equation (β) into (γ) gives

$(x_E+tx_D)^2 + (y_E+ty_D)^2 + (z_E+tz_D)^2 = 1$

which expands to

$t^2(x_D^2+y_D^2+z_D^2) + t(2x_Ex_D+2y_Ey_D+2z_Ez_D) + (x_E^2+y_E^2+z_E^2-1) = 0$

which is of the form

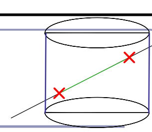
$at^2+bt+c=0$

which can be solved for t:

$$t = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

...giving us two points of intersection.

Hitting things with rays: Cylinder



The infinite unit cylinder, centered at the origin, has the implicit equation

$x^2 + y^2 = 1$ (δ)

Substituting equation (β) into (δ) gives

$(x_E+tx_D)^2 + (y_E+ty_D)^2 = 1$

which expands to

$t^2(x_D^2+y_D^2) + t(2x_Ex_D+2y_Ey_D) + (x_E^2+y_E^2-1) = 0$

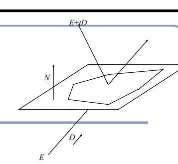
which is of the form

$at^2+bt+c=0$

which can be solved for t as before, giving us two points of intersection.

The cylinder is infinite; there is no z term.

Hitting things with rays: Planes and polygons



A planar polygon P can be defined as

Polygon $P = \{v^1, \ldots, v^n\}$

which gives us the normal to P as

$N = (v^n - v^1) \times (v^2 - v^1)$

The equation for the plane of P is

$N \bullet (p - v^1) = 0$ (ζ)

Substituting equation (α) into (ζ) for p yields

$N \bullet (E + tD - v^1) = 0$

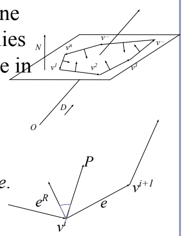
$x_N(x_E+tx_D-x_v^1) + y_N(y_E+ty_D-y_v^1) + z_N(z_E+tz_D-z_v^1) = 0$

$$t = \frac{(N \bullet v^1) - (N \bullet E)}{N \bullet D}$$

Point in convex polygon

Half-planes method

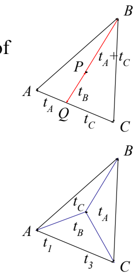
- Each edge defines an infinite half-plane covering the polygon. If the point P lies in all of the half-planes then it must be in the polygon.
- For each edge $e = v^i \rightarrow v^{i+1}$:
  - Rotate e by 90˚ CCW around N.
    - Do this quickly by crossing N with e.
  - If $e^R \bullet (P - v^i) < 0$ then the point is outside e.
- Fastest known method.



Barycentric coordinates

Barycentric coordinates (t_A, t_B, t_C) are a coordinate system for describing the location of a point P inside a triangle (A, B, C).

- You can think of (t_A, t_B, t_C) as 'masses' placed at (A, B, C) respectively so that the center of gravity of the triangle lies at P.
- (t_A, t_B, t_C) are also proportional to the subtriangle areas.
  - The area of a triangle is ½ the length of the cross product of two of its sides.



Point in nonconvex polygon

Winding number

- The *winding number* of a point P in a curve C is the number of times that the curve wraps around the point.
- For a simple closed curve (as any well-behaved polygon should be) this will be zero if the point is outside the curve, non-zero of it's inside.
- The winding number is the sum of the angles from v^i to P to v^{i+1}.
  - Caveat: This method is elegant but slow.

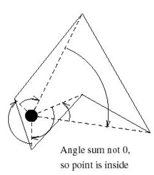
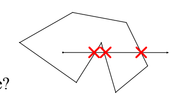


Figure from Eric Haines' "Point in Polygon Strategies", *Graphics Gems IV*, 1994

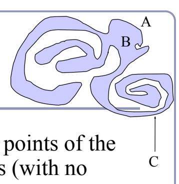
Point in nonconvex polygon

Ray casting (1974)

- Odd number of crossings = inside
- Issues:
  - How to find a point that you *know* is inside?
  - What if the ray hits a vertex?
  - Best accelerated by working in 2D
    - You could transform all vertices such that the coordinate system of the polygon has normal = Z axis…
    - Or, you could observe that crossings are invariant under scaling transforms and just project along any axis by ignoring (for example) the Z component.
- Validity proved by the *Jordan curve* theorem



The *Jordan curve theorem*



"Any simple closed curve C divides the points of the plane not on C into two distinct domains (with no points in common) of which C is the common boundary."

- First stated (but proved incorrectly) by Camille Jordan (1838 -1922) in his *Cours d'Analyse*.

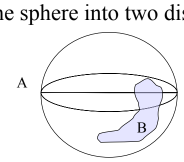
Sketch of proof : (For full proof see Courant & Robbins, 1941.)

- Show that any point in A can be joined to any other point in A by a path which does not cross C, and likewise for B.
- Show that any path connecting a point in A to a point in B *must* cross C.

The Jordan curve theorem on a sphere

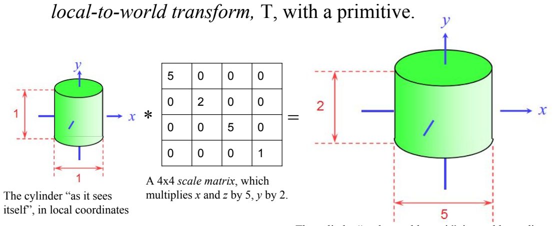
Note that the Jordan curve theorem can be extended to a curve on a sphere, or anything which is topologically equivalent to a sphere.

"Any simple closed curve on a sphere separates the surface of the sphere into two distinct regions."



Local coordinates, world coordinates

A very common technique in graphics is to associate a *local-to-world transform,* T, with a primitive.



The cylinder "as it sees itself", in local coordinates

A 4x4 *scale matrix*, which multiplies *x* and *z* by 5, *y* by 2.

The cylinder "as the world sees it", in world coordinates

Local coordinates, world coordinates: Transforming the ray

In order to test whether a ray hits a transformed object, we need to describe the ray in the object's *local coordinates*. We transform the ray by the *inverse of the local to world matrix*, T^{-1}.

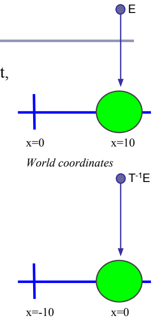
If the ray is defined by

`P(t) = E + tD`

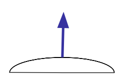
then the ray in local coordinates is defined by

$T^{-1}(P(t)) = T^{-1}(E) + t(T^{-1}{}_{3x3}D)$

where $T^{-1}{}_{3x3}$ is the top left 3x3 submatrix of T^{-1}.



Finding the normal



We often need to know *N*, the *normal to the surface* at the point where a ray hits a primitive.

- If the ray R hits the primitive P at point X then N is…

Primitive type	Equation for N
Unit Sphere centered at the origin	$N = X$
Infinite Unit Cylinder centered at the origin	$N = [\,x_X, y_X, 0\,]$
Infinite Double Cone centered at the origin	$N = X \times (X \times [\,0, 0, z_X\,])$
Plane with normal *n*	$N = n$

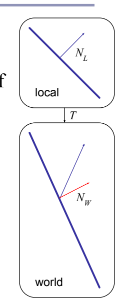
We use the normal for color, reflection, refraction, shadow rays...

Converting the normal from local to world coordinates

To find the world-coordinates normal *N* from the local-coordinates N_L, multiply N_L by the transpose of the inverse of the top left-hand 3x3 submatrix of *T*:

$$N = ((T_{3x3})^{-1})^T N_L$$

- We want the top left 3x3 to discard translations
- For any rotation Q, $(Q^{-1})^T = Q$
- Scaling is unaffected by transpose, and a scale of (a,b,c) becomes $(1/a, 1/b, 1/c)$ when inverted



Local coordinates, world coordinates Summary

To compute the intersection of a ray R=E+tD with an object transformed by local-to-world transform T:

1. Compute R', the ray R in local coordinates, as
   $P'(t) = T^{-1}(P(t)) = T^{-1}(E) + t(T^{-1}{}_{3x3}(D))$
2. Perform your hit test in local coordinates.
3. Convert all hit points from local coordinates back to world coordinates by multiplying them by T.
4. Convert all hit normals from local coordinates back to world coordinates by multiplying them by $((T^{3x3})^{-1})^T$.

This will allow you to efficiently and quickly fire rays at arbitrarily-transformed primitive objects.

Speed up ray-tracing with *bounding volumes*

Bounding volumes help to quickly accelerate volumetric tests, such as "does the ray hit the cow?"

- choose fast hit testing over accuracy
- 'bboxes' don't have to be tight

Axis-aligned bounding boxes

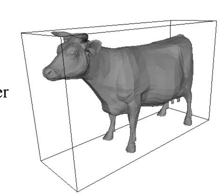
- max and min of x/y/z.

Bounding spheres

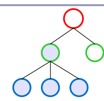
- max of radius from some rough center

Bounding cylinders

- common in early FPS games

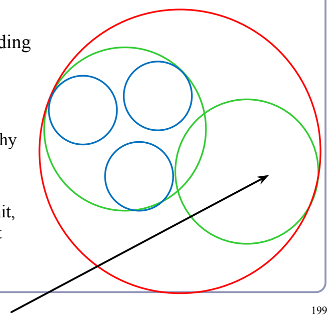


Bounding volumes in hierarchy



Hierarchies of bounding volumes allow early discarding of rays that won't hit large parts of the scene.

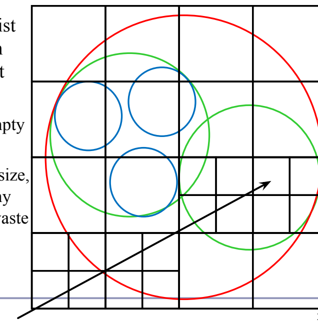
- Pro: Rays can skip subsections of the hierarchy
- Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



Subdivision of space

Split space into cells and list in each cell every object in the scene that overlaps that cell.

- Pro: The ray can skip empty cells
- Con: Depending on cell size, objects may overlap many filled cells or you may waste memory on many empty cells



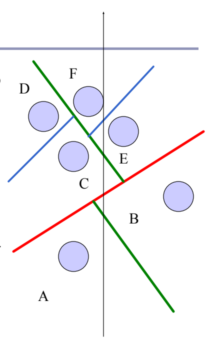
Popular acceleration structures: BSP Trees

The *BSP tree* partitions the scene into objects in front of, on, and behind a tree of planes.

- When you fire a ray into the scene, you test all near-side objects before testing far-side objects.

Problems:

- choice of planes is not obvious
- computation is slow
- plane intersection tests are heavy on floating-point math.



Popular acceleration structures: *kd-trees*

The *kd-tree* is a simplification of the BSP Tree data structure

- Space is recursively subdivided by axis-aligned planes and points on either side of each plane are separated in the tree.
- The *kd*-tree has O(n log n) insertion time (but this is very optimizable by domain knowledge) and O($n^{2/3}$) search time.
- *kd*-trees don't suffer from the mathematical slowdowns of BSPs because their planes are always axis-aligned.

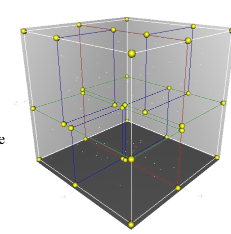


Image from Wikipedia, bless their hearts.

Popular acceleration structures: *Bounding Interval Hierarchies*

The *Bounding Interval Hierarchy* subdivides space around the volumes of objects and shrinks each volume to remove unused space.

- Think of this as a "best-fit" *k*d-tree
- Can be built dynamically as each ray is fired into the scene

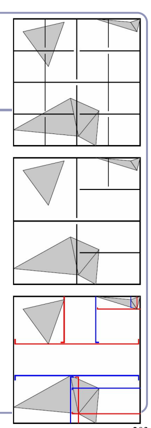
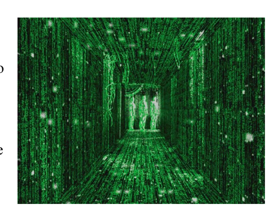


Image from Wächter and Keller's paper, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics (2006)

Using OpenGL to accelerate ray-tracing

To accelerate first raycast, don't raycast: use existing hardware.

- Use hardware rendering (eg OpenGL) to write to an offscreen buffer.
- Set the color of each primitive equal to a pointer to that primitive.
- Render your scene in gl with z-buffering and no lighting.
- The 'color' value at each pixel in the buffer is now a pointer to the primitive under that pixel.

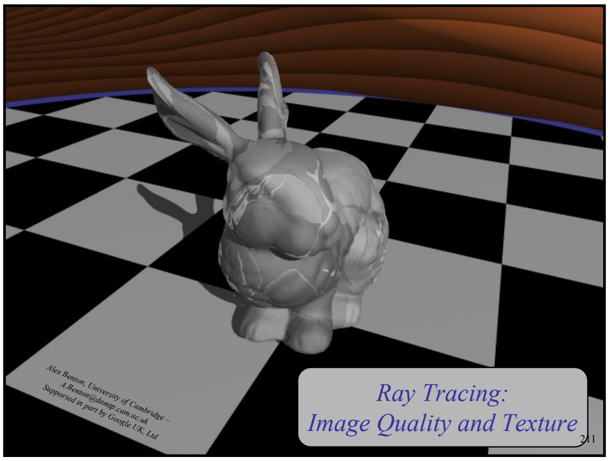


References

Jordan curves
R. Courant, H. Robbins, *What is Mathematics?*, Oxford University Press, 1941
http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/97/Octavian/compgeom.html

Intersection testing
http://www.realtimerendering.com/intersections.html
http://tog.acm.org/editors/erich/ptinpoly/
http://mathworld.wolfram.com/BarycentricCoordinates.html

Ray tracing
Foley & van Dam, *Computer Graphics* (1995)
Jon Genetti and Dan Gordon, *Ray Tracing With Adaptive Supersampling in Object Space*, http://www.cs.uaf.edu/~genetti/Research/Papers/GI93/GI.html (1993)
Zack Waters, "Realistic Raytracing", http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/realistic_raytracing.html

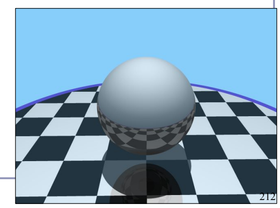


Ray Tracing: *Image Quality and Texture*

Shadows

To simulate shadows in ray tracing, fire a ray from P towards each light L_i. If the ray hits another object before the light, then discard L_i in the sum.

- This is a boolean removal, so it will give hard-edged shadows.
- Hard-edged shadows suggest a pinpoint light source.



Softer shadows

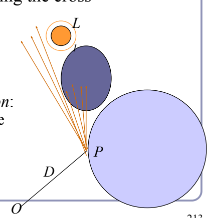
Shadows in nature are not sharp because light sources are not infinitely small.

- Also because light scatters, etc.

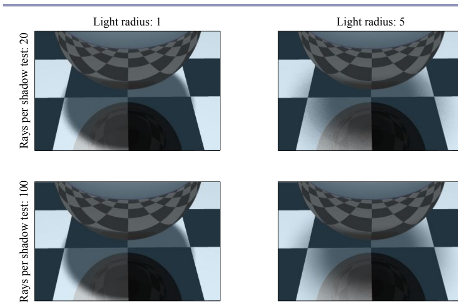
For lights with volume, fire many rays, covering the cross-section of your illuminated space.

Illumination is scaled by (the total number of rays that aren't blocked) divided by (the total number of rays fired).

- This is an example of *Monte-Carlo integration*: a coarse simulation of an integral over a space by randomly sampling it with many rays.
- The more rays fired, the smoother the result.



Softer shadows

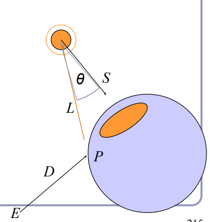


All images anti-aliased with 4x supersampling
Distance to light in all images: 20 units

Raytraced spotlights

To create a spotlight shining along axis S, you can multiply the (diffuse+specular) term by $(\max(L \bullet S, 0))^m$.

- Raising m will tighten the spotlight, but leave the edges soft.
- If you'd prefer a hard-edged spotlight of uniform internal intensity, you can use a conditional, e.g. $((L \bullet S > \cos(15^\circ))\ ?\ 1 : 0)$.



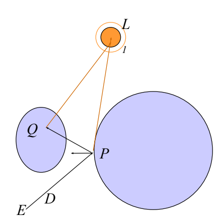
Reflection

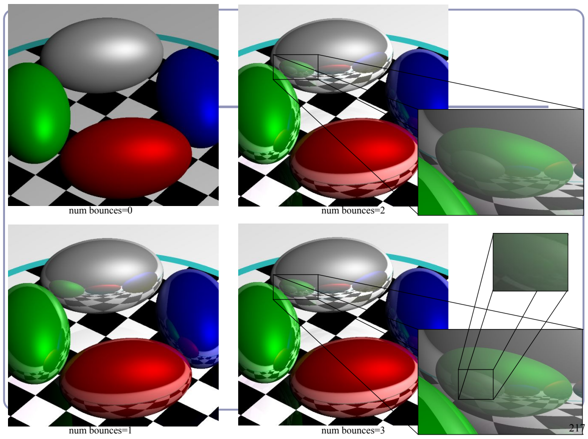
Reflection rays are calculated as:

$$R = 2(-D \bullet N)N + D$$

- Finding the reflected color is a recursive raycast.
- Reflection has *scene-dependant* performance impact.

- If you're using the GPU, GLSL supports `reflect()` as a built-in function.

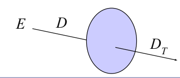


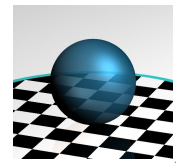


Transparency

To add transparency, generate and trace a new *transparency ray* with $E_T=P$, $D_T=D$.

To support this in software, make color a 1x4 vector where the fourth component, 'alpha', determines the weight of the recursed transparency ray.





Refraction

The *angle of incidence* of a ray of light where it strikes a surface is the acute angle between the ray and the surface normal.

The *refractive index* of a material is a measure of how much the speed of light[1] is reduced inside the material.

- The refractive index of air is about 1.003.
- The refractive index of water is about 1.33.

[1] Or sound waves or other waves

Refraction

Snell's Law:

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

"The ratio of the sines of the *angles of incidence* of a ray of light at the interface between two materials is equal to the inverse ratio of the *refractive indices* of the materials is equal to the ratio of the speeds of light in the materials."

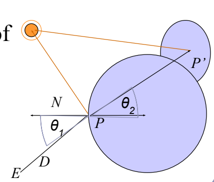
Historical note: this formula has been attributed to Willebrord Snell (1591-1626) and René Descartes (1596-1650) but first discovery goes to Ibn Sahl (940-1000) of Baghdad.

Refraction in ray tracing

$$\theta_1=\cos^{-1}(N\bullet D)$$

$$\frac{\sin\theta_1}{\sin\theta_2}=\frac{n_2}{n_1}\ \rightarrow\ \theta_2=\sin^{-1}\left(\frac{n_1}{n_2}\sin\theta_1\right)$$

Using Snell's Law and the angle of incidence of the incoming ray, we can calculate the angle from the negative normal to the outbound ray.



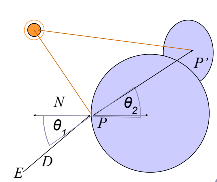
Refraction in ray tracing

What if the arcsin parameter is > 1?

- Remember, arcsin is defined in [-1,1].
- We call this the *angle of total internal reflection*: light is trapped completely inside the surface.

$$\theta_2=\sin^{-1}\left(\frac{n_1}{n_2}\sin\theta_1\right)$$

Total internal reflection



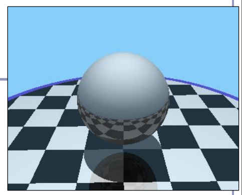
Aliasing

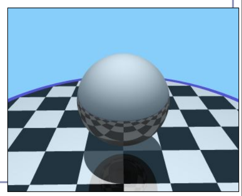
aliasing
/ˈeɪlɪəsɪŋ/
noun: **aliasing**
1. PHYSICS / TELECOMMUNICATIONS

the misidentification of a signal frequency, introducing distortion or error.

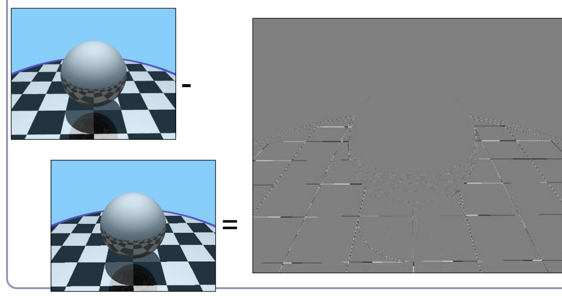
"high-frequency sounds are prone to aliasing"
2. COMPUTING

the distortion of a reproduced image so that curved or inclined lines appear inappropriately jagged, caused by the mapping of a number of points to the same pixel.





Aliasing



Anti-aliasing

Fundamentally, the problem with aliasing is that we're sampling an infinitely continuous function (the color of the scene) with a finite, discrete function (the pixels of the image).

One solution to this is *super-sampling*. If we fire multiple rays through each pixel, we can average the colors computed for every ray together to a single blended color.

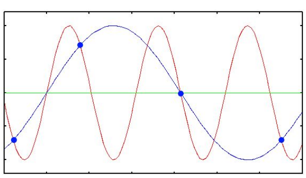
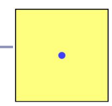


Image source: www.svi.nl

Anti-aliasing

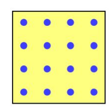
Single point

- Fire a single ray through the pixel's center



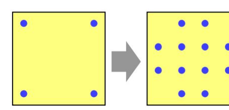
Super-sampling

- Fire multiple rays through the pixel and average the result
- Regular grid, random, jittered, Poisson disks



Adaptive super-sampling

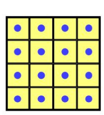
- Fire a few rays through the pixel, check the variance of the resulting values, if similar enough then stop else fire more rays

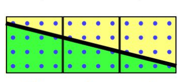


Types of super-sampling

Regular grid

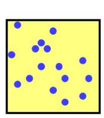
- Divide the pixel into a number of sub-pixels and fire a ray through the center of each
- This can still lead to noticeable aliasing unless a very high resolution of sub-pixel grid is used





Random

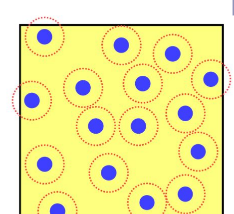
- Fire N rays at random points in the pixel
- Replaces aliasing artifacts with noise artifacts
  - But the human eye is much less sensitive to noise than to aliasing
- Requires special treatment for animation

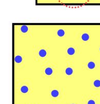


Types of super-sampling

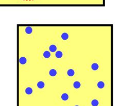
Poisson disk

- Fire N rays at random points in the pixel, with the proviso that no two rays shall pass through the pixel closer than ε to one another
- For N rays this produces a better looking image than pure random sampling
- However, can be very hard to implement correctly / quickly





Poisson disk

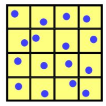


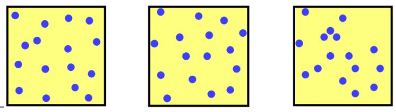
pure random

Types of super-sampling

Jittered

- Divide the pixel into N sub-pixels and fire one ray at a random point in each sub-pixel
- Approximates the Poisson disk behavior
- Better than pure random sampling, easier (and significantly faster) to implement than Poisson

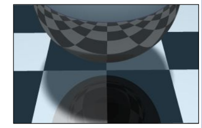


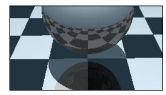


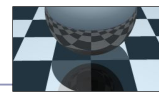
jittered Poisson disk pure random

Applications of super-sampling

- Anti-aliasing
- Soft shadows
- Depth-of-field camera effects (fixed focal depth, finite aperture)







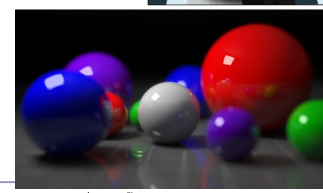


Image credit: http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Anisotropic shading

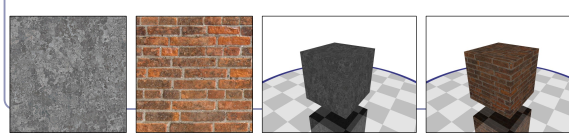
Anisotropic shading occurs in nature when light reflects off a surface differently in one direction from another, as a function of the surface itself. The specular component is modified by the direction of the light.

http://www.blenderguru.com/videos/introduction-to-anisotropic-shading/

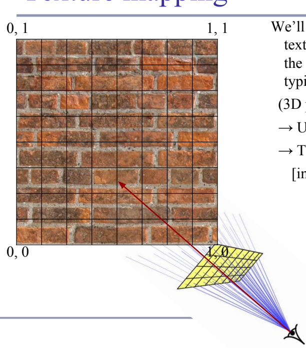
Texture mapping

As observed in last year's course, real-life objects rarely consist of perfectly smooth, uniformly colored surfaces.

Texture mapping is the art of applying an image to a surface, like a decal. Coordinates on the surface are mapped to coordinates in the texture.



Texture mapping

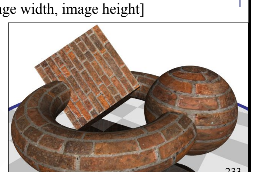


We'll need to query the color of the texture at the point in 3D space where the ray hits our surface. This is typically done by mapping

(3D point in local coordinates)

→ U,V coordinates bounded [0-1, 0-1]

→ Texture coordinates bounded by [image width, image height]



UV mapping the primitives

UV mapping of a unit cube	UV mapping of a unit sphere
if $\|x\| == 1$: $u = (z + 1) / 2$ $v = (y + 1) / 2$ *elif* $\|y\| == 1$: $u = (x + 1) / 2$ $v = (z + 1) / 2$ *else:* $u = (x + 1) / 2$ $v = (y + 1) / 2$	$u = 0.5 + atan2(z, x) / 2\pi$ $v = 0.5 - asin(y) / \pi$ **UV mapping of a torus of major radius R** $u = 0.5 + atan2(z, x) / 2\pi$ $v = 0.5 + atan2(y, ((x^2 + z^2)^{½} - R) / 2\pi$

UV mapping is easy for primitives but can be very difficult for arbitrary shapes.

Texture mapping

One constraint on using images for texture is that images have a finite resolution, and a virtual (ray-traced) camera can get quite near to the surface of an object.

This can lead to a single image pixel covering multiple ray-traced pixels (or vice-versa), leading to blurry or aliased pixels in your texture.

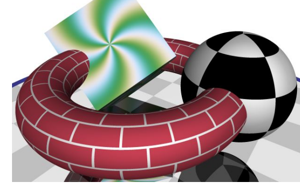
Procedural texture

Instead of relying on discrete pixels, you can get infinitely more precise results with procedurally generated textures.

Procedural textures compute the color directly from the U,V coordinate without an image lookup.

For example, here's the code for the torus' brick pattern (right):

```
tx = (int) 10 * u
ty = (int) 10 * v
oddity = (tx & 0x01) == (ty & 0x01)
edge = ((10 * u - tx < 0.1) && oddity) || (10 * v - ty < 0.1)
return edge ? WHITE : RED
```



Confession: I cheated slightly and multiplied the u coordinate by 4 to repeat the brick texture four times around the torus.

Procedural volumetric texture

By mapping 3D coordinates to colors, we can create *volumetric texture*. The input to the texture is local model coordinates; the output is color and surface characteristics.

For example, to produce wood-grain texture, trees grow rings, with darker wood from earlier in the year and lighter wood from later in the year.

- Choose shades of early and late wood
- $f(P) = (X_p^2+Z_p^2) \bmod 1$
- $color(P) = earlyWood + f(P) * (lateWood - earlyWood)$

$f(P)=0$ $f(P)=1$

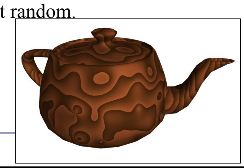
Adding realism

The teapot on the previous slide doesn't look very wooden, because it's perfectly uniform. One way to make the surface look more natural is to add a randomized noise field to f(P):

$f(P) = (X_p^2+Z_p^2 + noise(P)) \bmod 1$

where *noise(P)* is a function that maps 3D coordinates in space to scalar values chosen at random.

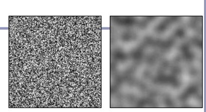
For natural-looking results, use *Perlin noise*, which interpolates smoothly between noise values.



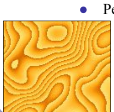
Perlin noise

Perlin noise (invented by Ken Perlin) is a method for generating noise which has some useful traits:

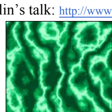
- It is a *band-limited repeatable pseudorandom* function (in the words of its author, Ken Perlin)
- It is bounded within a range close [-1, 1]
- It varies continuously, without discontinuity
- It has regions of relative stability
- It can be initialized with random values, extended arbitrarily in space, yet cached deterministically
  - Perlin's talk: http://www.noisemachine.com/talk1/



Non-coherent noise (left) and Perlin noise (right)
Image credit: Matt Zucker

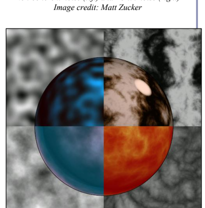


Matt Zucker



Matt Zucker

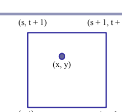
Matt Zucker



Ken Perlin

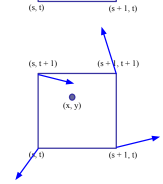
Perlin noise 1

Perlin noise caches 'seed' random values on a grid at integer intervals. You'll look up noise values at arbitrary points in the plane, and they'll be determined by the four nearest seed randoms on the grid.



Given point *(x, y)*, let *(s, t) = (floor(x), floor(y))*.

For each grid vertex in *{(s, t), (s+1, t), (s+1, t+1), (s, t+1)}* choose and cache a random vector of length one.

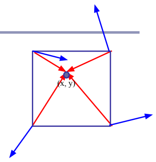


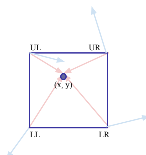
These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Perlin noise 2

For each of the four corners, take the dot product of the random seed vector with the vector from that corner to *(x, y)*. This gives you a unique scalar value per corner.

- As *(x, y)* moves across this cell of the grid, the values of the dot products will change smoothly, with no discontinuity.
- As *(x, y)* approaches a grid point, the contribution from that point will approach zero.
- The values of *LL, LR, UL, UR* are clamped to a range close to [-1, 1].





These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Perlin noise 3

Now we take a weighted average of *LL, LR, UL, UR*. Perlin noise uses a weighted averaging function chosen such that values close to zero and one are moved closer to zero and one, called the *ease curve*:

$S(t) = 3t^2 - 2t^3$

We interpolate along one axis first:

$L(x, y) = LL + S(x - floor(x))(LR-LL)$

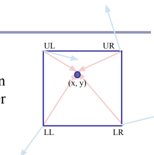
$U(x, y) = UL + S(x - floor(x))(UR-UL)$

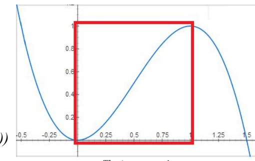
Then we interpolate again to merge the two upper and lower functions:

$noise(x, y) =$

$L(x, y) + S(y - floor(y))(U(x, y) - L(x, y))$

Voila!





The 'ease curve'

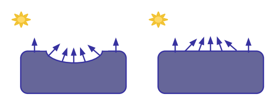
These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Tuning noise

Texture frequency 1 → 3 | Noise frequency 1 → 3 | Noise amplitude 1 → 3

Normal mapping

Normal mapping applies the principles of texture mapping to the surface normal instead of surface color.



The specular and diffuse shading of the surface varies with the normals in a dent on the surface.

If we duplicate the normals, we don't have to duplicate the dent.

In a sense, the ray tracer computes a trompe-l'oeuil image on the fly and 'paints' the surface with more detail than is actually present in the geometry.

Normal mapping

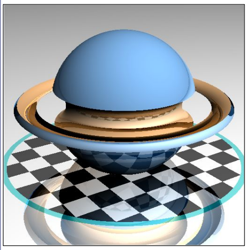
References

Ray tracing
Peter Shirley, Steve Marschner. *Fundamentals of Computer Graphics*. Taylor & Francis, 21 Jul 2009
Hughes, Van Dam et al. *Computer Graphics: Principles and Practice*. Addison Wesley, 3rd edition (10 July 2013)

Anisotropic shading
Greg Ward, "Measuring and Modeling Anisotropic Reflection", Computer Graphics (SIGGRAPH '92 Proceedings), pp. 265–272, July 1992 (http://radsite.lbl.gov/radiance/papers/sg92/paper.html)
https://en.wikibooks.org/wiki/GLSL_Programming/Unity/Brushed_Metal

Perlin noise
http://www.noisemachine.com/talk1/
http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Advanced Graphics



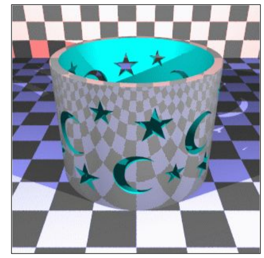
Advanced Scenes and Global Illumination

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

Constructive Solid Geometry

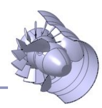
Constructive Solid Geometry (CSG) builds complicated forms out of simple primitives.

These primitives are combined with basic boolean operations: add, subtract, intersect.



CSG figure by Neil Dodgson

Constructive Solid Geometry



CSG models are easy to ray-trace but difficult to polygonalize

- Issues include choosing polygon boundaries at edges; converting adequately from pure smooth primitives to discrete (flat) faces; handling 'infinitely thin' sheet surfaces; and others.
- This is an ongoing research topic.

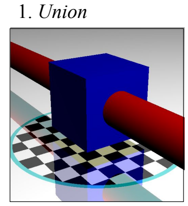
CSG models are well-suited to machine milling, automated manufacture, etc

- Great for 3D printers!

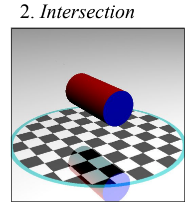
Constructive Solid Geometry

Three operations:

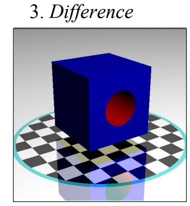
1. *Union*



2. *Intersection*



3. *Difference*



Constructive Solid Geometry

CSG surfaces can be described by a binary tree, where each leaf node is a primitive and each non-leaf node is a boolean operation.

(What would the *not* of a surface look like?)

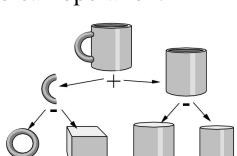
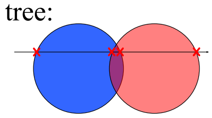


Figure from Wyvill (1995) part two, p. 4

Constructive Solid Geometry

For each node of the binary tree:



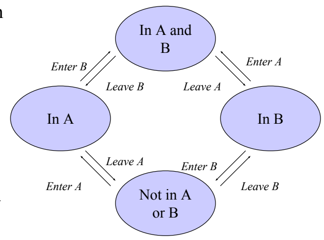
- Fire ray *r* at *A* and *B*.
- List in *t*-order all points where *r* enters of leaves *A* or *B*.
  - You can think of each intersection as a quad of booleans-- (*wasInA, isInA, wasInB, isInB*)
- Discard from the list all intersections which don't matter to the current boolean operation.
- Pass the list up to the parent node and recurse.

Ray-tracing CSG models

Each boolean operation can be modeled as a state machine.

For each operation, retain those intersections that transition into or out of the critical state(s).

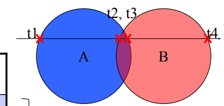
- Union: `{In A | In B | In A and B}`
- Intersection: `{In A and B}`
- Difference: `{In A}`



Ray-tracing CSG models

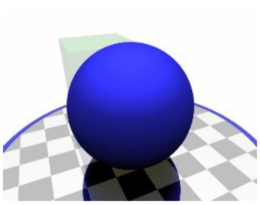
Example: Difference (A-B)

A-B	Was In A	Is In A	Was In B	Is In B
t1	No	Yes	No	No
t2	Yes	Yes	No	Yes
t3	Yes	No	Yes	Yes
t4	No	No	Yes	No

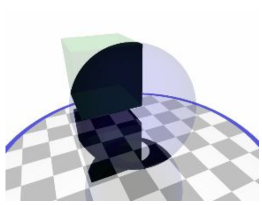



```
difference =
((wasInA != isInA) &&
 (!isInB)&&(!wasInB))
||
((wasInB != isInB) &&
 (wasInA || isInA))
```


CSG in action



Difference



Intersection

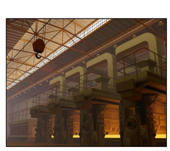
What's wrong with raytracing?

- Soft shadows are expensive
- Shadows of transparent objects require further coding or hacks
- Lighting off reflective objects follows different shadow rules from normal lighting
- Hard to implement diffuse reflection (color bleeding, such as in the Cornell Box—notice how the sides of the inner cubes are shaded red and green.)
- Fundamentally, the ambient term is a hack and the diffuse term is only one step in what should be a recursive, self-reinforcing series.

The *Cornell Box* is a test for rendering Software, developed at Cornell University in 1984 by Don Greenberg. An actual box is built and photographed; an identical scene is then rendered in software and the two images are compared.

Radiosity

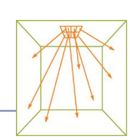
- *Radiosity* is an illumination method which simulates the global dispersion and reflection of diffuse light.
  - First developed for describing spectral heat transfer (1950s)
  - Adapted to graphics in the 1980s at Cornell University
- Radiosity is a finite-element approach to global illumination: it breaks the scene into many small elements ('*patches*') and calculates the energy transfer between them.

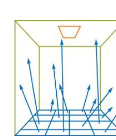


Images from Cornell University's graphics group
http://www.graphics.cornell.edu/online/research/

Radiosity—algorithm

- Surfaces in the scene are divided into *form factors* (also called *patches*), small subsections of each polygon or object.
- For every pair of form factors A, B, compute a *view factor* describing how much energy from patch A reaches patch B.
  - The further apart two patches are in space or orientation, the less light they shed on each other, giving lower view factors.
- Calculate the lighting of all directly-lit patches.
- Bounce the light from all lit patches to all those they light, carrying more light to patches with higher relative view factors. Repeating this step will distribute the total light across the scene, producing a total illumination model.





Radiosity—mathematical support

The 'radiosity' of a single patch is the amount of energy leaving the patch per discrete time interval.

This energy is the total light being emitted directly from the patch combined with the total light being reflected by the patch:

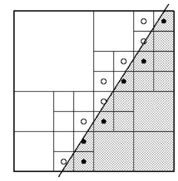
$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

where…

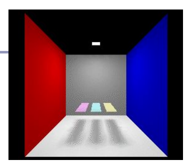
B_i is the radiosity of patch i;
B_j is the cumulative radiosity of all other patches ($j \neq i$)
E_i is the emitted energy of the patch
R_i is the reflectivity of the patch
F_{ij} is the view factor of energy from patch i to patch j.

Radiosity—form factors

- Finding form factors can be done procedurally or dynamically
  - Can subdivide every surface into small patches of similar size
  - Can dynamically subdivide wherever the 1st derivative of calculated intensity rises above some threshold.
- Computing cost for a general radiosity solution goes up as the square of the number of patches, so try to keep patches down.
  - Subdividing a large flat white wall could be a waste.
- Patches should ideally closely align with lines of shadow.



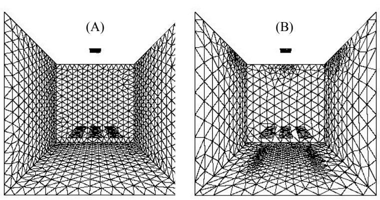
Radiosity—implementation



(A) Simple patch triangulation

(B) Adaptive patch generation: the floor and walls of the room are dynamically subdivided to produce more patches where shadow detail is higher.

Images from "Automatic generation of node spacing function", IBM (1998) http://www.trl.ibm.com/projects/meshing/nsp/nspE.htm

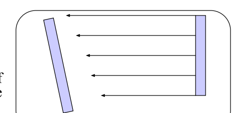


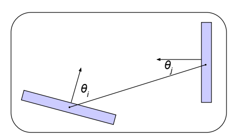
Radiosity—view factors

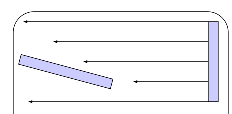
One equation for the view factor between patches i, j is:

$$Fi \rightarrow j = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} V(i,j)$$

…where θ_i is the angle between the normal of patch i and the line to patch j, r is the distance and $V(i,j)$ is the visibility from i to j (0 for occluded, 1 for clear line of sight.)

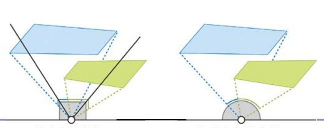






Radiosity—calculating visibility

- Calculating $V(i,j)$ can be slow.
- One method is the *hemicube*, in which each form factor is encased in a half-cube. The scene is then 'rendered' from the point of view of the patch, through the walls of the hemicube; $V(i,j)$ is computed for each patch based on which patches it can see (and at what percentage) in its hemicube.
- A purer method, but more computationally expensive, uses hemispheres.



Note: This method can be accelerated using modern graphics hardware to render the scene. The scene is 'rendered' with flat lighting, setting the 'color' of each object to be a pointer to the object in memory.

Radiosity gallery

Image from *A Two Pass Solution to the Rendering Equation: a Synthesis of Ray Tracing and Radiosity Methods*, John R. Wallace, Michael F. Cohen and Donald P. Greenberg (Cornell University, 1987)

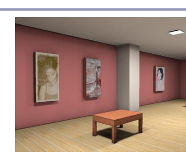


Image from *GPU Gems II*, nVidia

Teapot (wikipedia)

Shadows, refraction and caustics

- Problem: shadow ray strikes transparent, refractive object.
  - Refracted shadow ray will now miss the light.
  - This destroys the validity of the boolean shadow test.
- Problem: light passing through a refractive object will sometimes form *caustics* (right), artifacts where the envelope of a collection of rays falling on the surface is bright enough to be visible.

This is a photo of a real pepper-shaker.
Note the caustics to the left of the shaker, in and outside of its shadow.
Photo credit: Jan Zankowski

Shadows, refraction and caustics

- Solutions for shadows of transparent objects:
  - Backwards ray tracing (Arvo)
    - *Very* computationally heavy
    - Improved by stencil mapping (Shenya et al)
  - Shadow attenuation (Pierce)
    - Low refraction, no caustics
- More general solution:
  - Photon mapping (Jensen)→

Image from http://graphics.ucsd.edu/~henrik/
Generated with photon mapping

Photon mapping

Photon mapping is the process of emitting photons into a scene and tracing their paths probabilistically to build a *photon map*, a data structure which describes the illumination of the scene independently of its geometry.

This data is then combined with ray tracing to compute the global illumination of the scene.

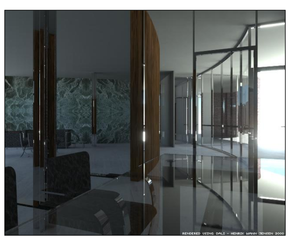


Image by Henrik Jensen (2000)

Photon mapping—algorithm (1/2)

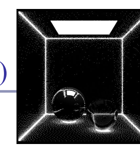


Image by Zack Waters

Photon mapping is a two-pass algorithm:

1. Photon scattering
   A. Photons are fired from each light source, scattered in randomly-chosen directions. The number of photons per light is a function of its surface area and brightness.
   B. Photons fire through the scene (re-use that raytracer, folks.) Where they strike a surface they are either absorbed, reflected or refracted.
   C. Wherever energy is absorbed, cache the location, direction and energy of the photon in the *photon map*. The photon map data structure must support fast insertion and fast nearest-neighbor lookup; a *kd-tree*[1] is often used.

Photon mapping—algorithm (2/2)

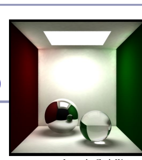


Image by Zack Waters

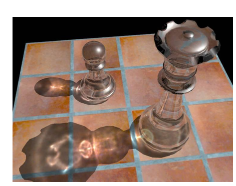
Photon mapping is a two-pass algorithm:

2. Rendering
   A. Ray trace the scene from the point of view of the camera.
   B. For each first contact point P use the ray tracer for specular but compute diffuse from the photon map and do away with ambient completely.
   C. Compute radiant illumination by summing the contribution along the eye ray of all photons within a sphere of radius r of P.
   D. Caustics can be calculated directly here from the photon map. For speed, the caustic map is usually distinct from the radiance map.

Photon mapping is probabilistic

This method is a great example of *Monte Carlo integration*, in which a difficult integral (the lighting equation) is simulated by randomly sampling values from within the integral's domain until enough samples average out to about the right answer.

- This means that you're going to be firing *millions* of photons. Your data structure is going to have to be very space-efficient.



http://www.okino.com/conv/imp_jt.htm

Photon mapping is probabilistic

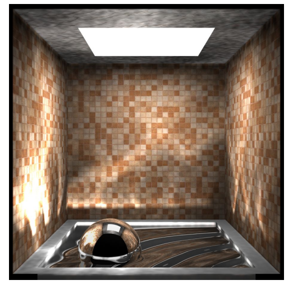
- Initial photon direction is random. Constrained by light shape, but random.
- What exactly happens each time a photon hits a solid also has a random component:
  - Based on the diffuse reflectance, specular reflectance and transparency of the surface, compute probabilities p_d, p_s and p_t where $(p_d+p_s+p_t) \leq 1$. This gives a probability map:

0 p_d p_s p_t 1

This surface would have minimal specular highlight.

  - Choose a random value $p \in [0,1]$. Where p falls in the probability map of the surface determines whether the photon is reflected, refracted or absorbed.

Photon mapping gallery



http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html

http://graphics.ucsd.edu/~henrik/images/global.html

http://www.pbrt.org/gallery.php

References

Shirley and Marschner, "Fundamentals of Computer Graphics", Chapter 24 (2009)

Radiosity
- nVidia: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter39.html
- Cornell: http://www.graphics.cornell.edu/online/research/
- Wallace, J. R., K. A. Elmquist, and E. A. Haines. 1989, "A Ray Tracing Algorithm for Progressive Radiosity." In *Computer Graphics (Proceedings of SIGGRAPH 89)* 23(4), pp. 315–324.
- Buss, "3-D Computer Graphics: A Mathematical Introduction with OpenGL" (Chapter XI), Cambridge University Press (2003)

Photon mapping
- Henrik Jenson, "Global Illumination using Photon Maps": http://graphics.ucsd.edu/~henrik/
- Zack Waters, "Photon Mapping": http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html