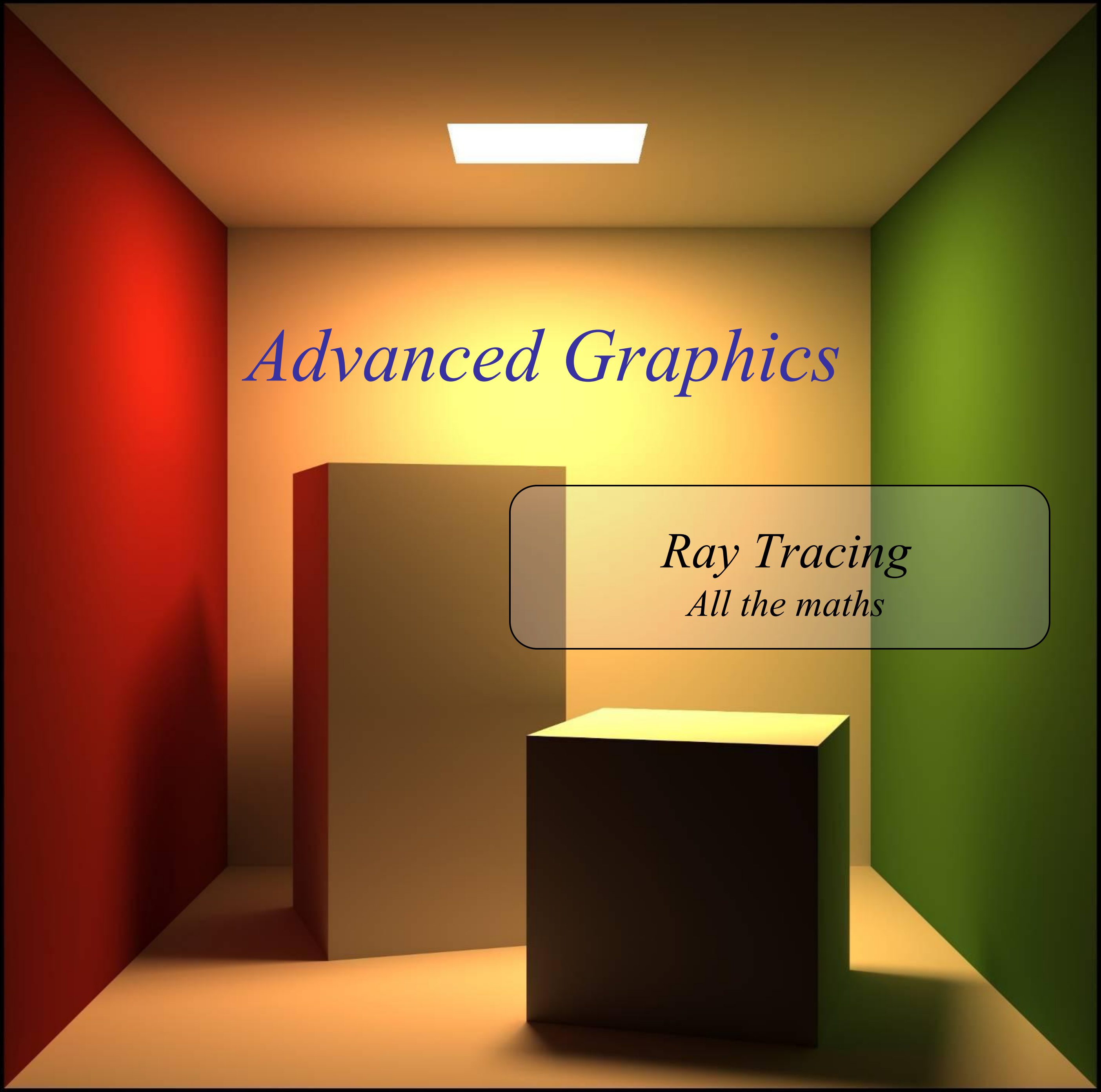


Advanced Graphics

Ray Tracing

All the maths

"*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.

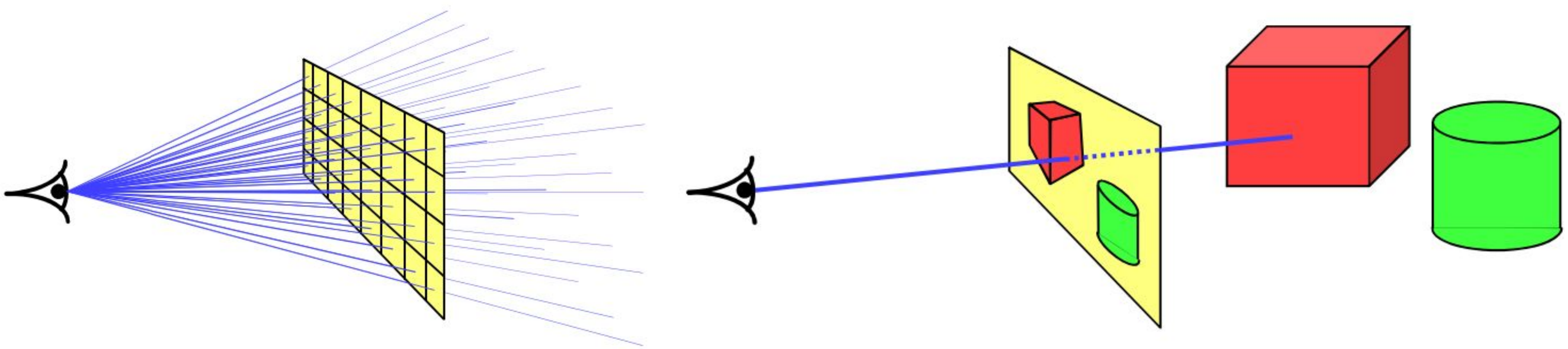
Alex Benton, University of Cambridge – alex@bentonian.com

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):

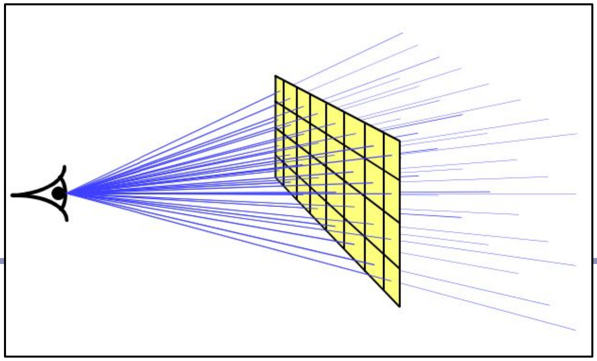
        if (the ray hits the object):

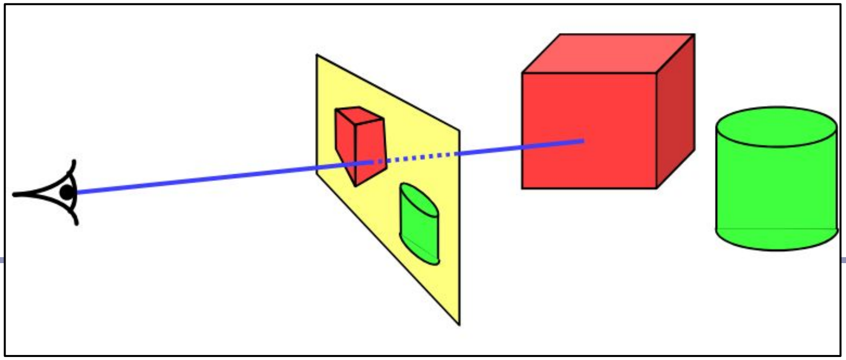
            if (the intersection is the nearest (so far) to the eye):

                Record the intersection point.

                Record the color of the object at that point.

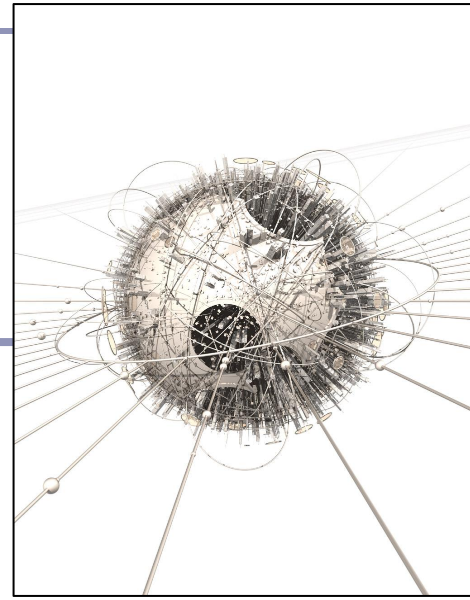
    Set the screen plane pixel to the nearest recorded color.



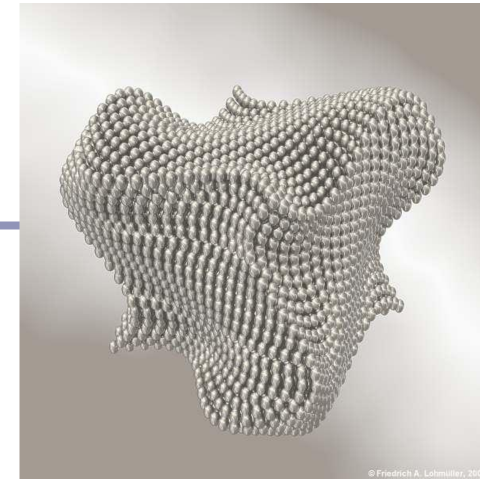


Examples

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



"POV Planet" by Casey Uhrig (2004)



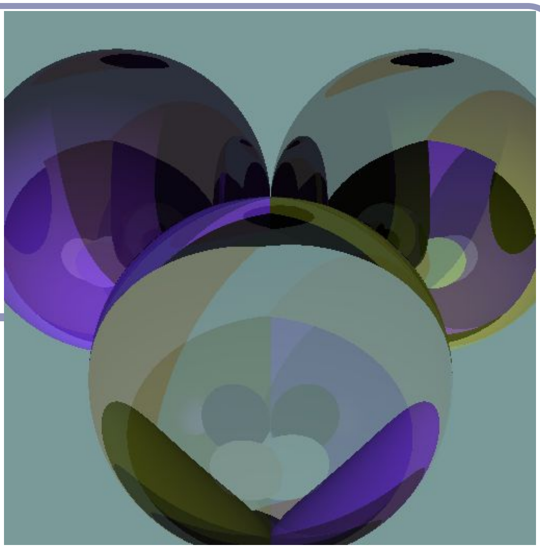
"Dancing Cube" by Friedrich A. Lohmueller (2003)

"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
- …

```c
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.

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):

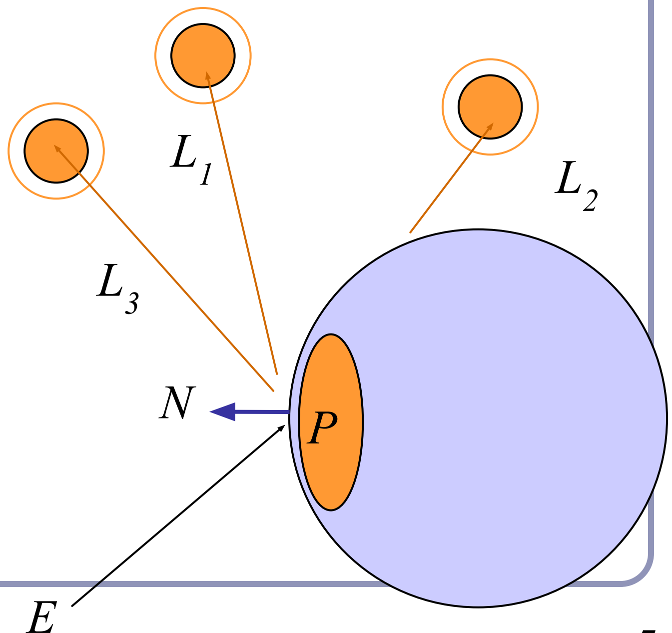
  if $(N \bullet L) > 0$:

  [Optionally: if (a ray from P to L_i can reach L_i):]

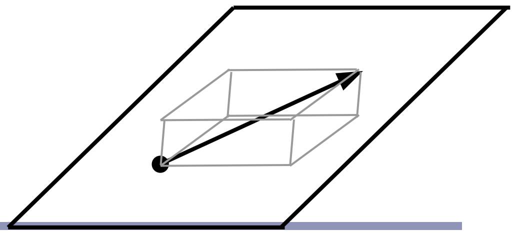
    $diffuse \mathrel{+}= k_D(N \bullet L)$

    $specular \mathrel{+}= k_S(R \bullet E)^n$

intensity at P = ambient + diffuse + specular



Hitting things with rays



A ray is defined parametrically as

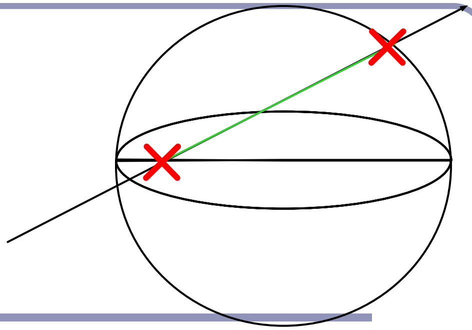
$$P(t) = E + tD,\ t \geq 0 \qquad (\alpha)$$

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:

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

Hitting things with rays: Sphere



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

$$x^2 + y^2 + z^2 = 1 \qquad (\gamma)$$

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

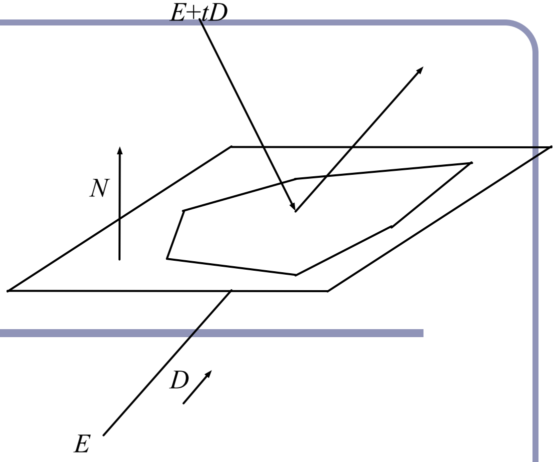
$$at^2+bt+c=0$$

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: 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

$$N \bullet (p - v_1) = 0 \qquad (\zeta)$$

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

$N \bullet (E + tD - v_1) = 0$

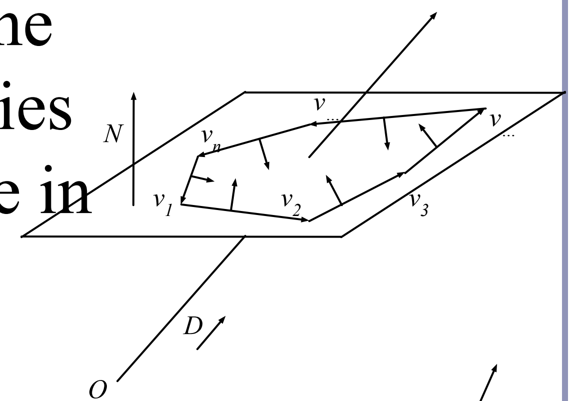
$x_N(x_E + tx_D - x_{v1}) + y_N(y_E + ty_D - y_{v1}) + z_N(z_E + tz_D - z_{v1}) = 0$

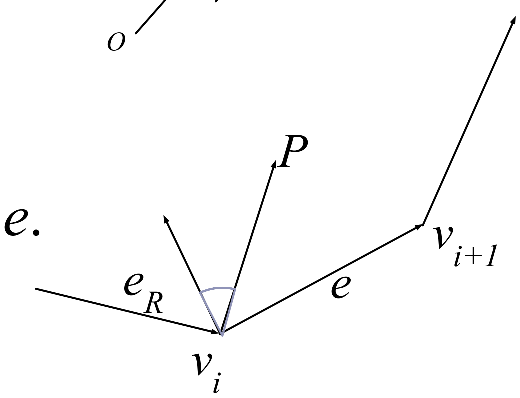
$$t = \frac{(N \bullet v1) - (N \bullet E)}{N \bullet D}$$

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.

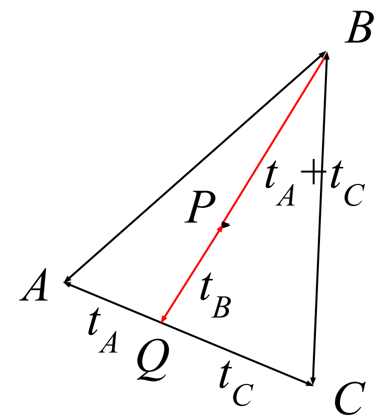


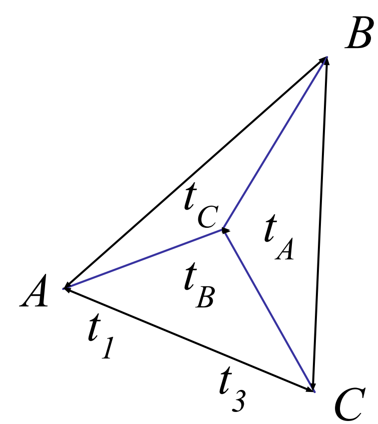


Barycentric coordinates

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.



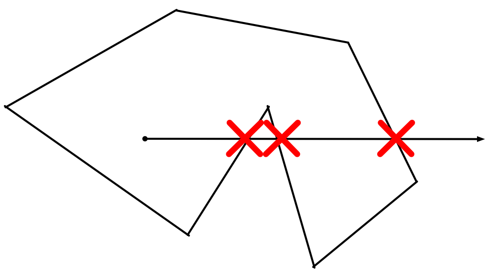


Barycentric coordinates

```
// Compute barycentric coordinates (u, v, w) for
// point p with respect to triangle (a, b, c)
vec3 barycentric(vec3 p, vec3 a, vec3 b, vec3 c) {
  vec3 v0 = b - a, v1 = c - a, v2 = p - a;
  float d00 = dot(v0, v0);
  float d01 = dot(v0, v1);
  float d11 = dot(v1, v1);
  float d20 = dot(v2, v0);
  float d21 = dot(v2, v1);
  float denom = d00 * d11 - d01 * d01;
  float v = (d11 * d20 - d01 * d21) / denom;
  float w = (d00 * d21 - d01 * d20) / denom;
  float u = 1.0 - v - w;
  return vec3(u, v, w);
}
```

Code credit: Christer Ericson, *Real-Time Collision Detection* (2004)
(adapted to GLSL for this lecture)

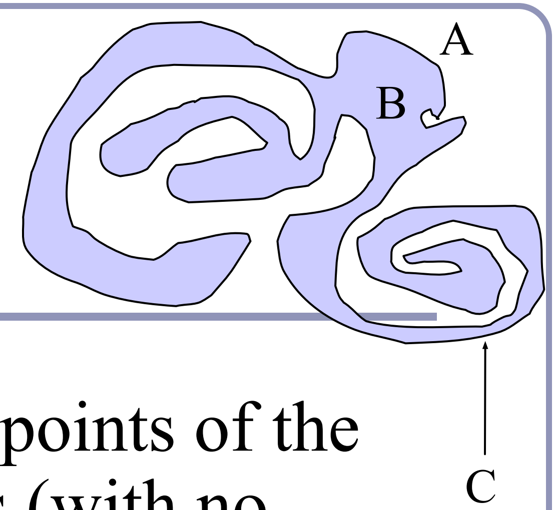
Point in nonconvex polygon



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*.

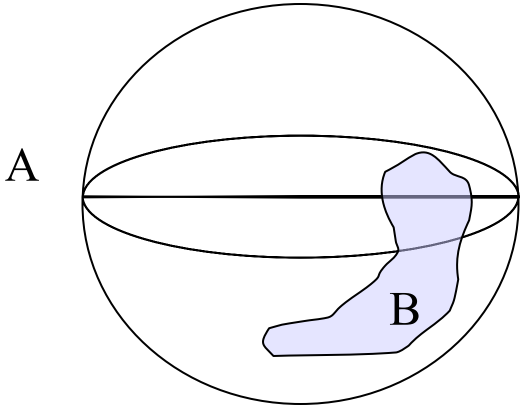
Sketch of proof : (For full proof see Courant & Robbins, 1941.)

- 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

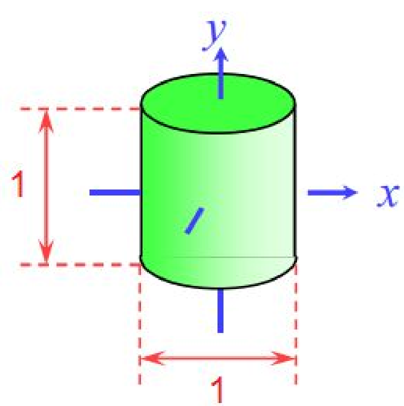
Note that the Jordan curve theorem can be extended to a curve on a sphere, or anything which is topologically equivalent to a sphere.

“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.



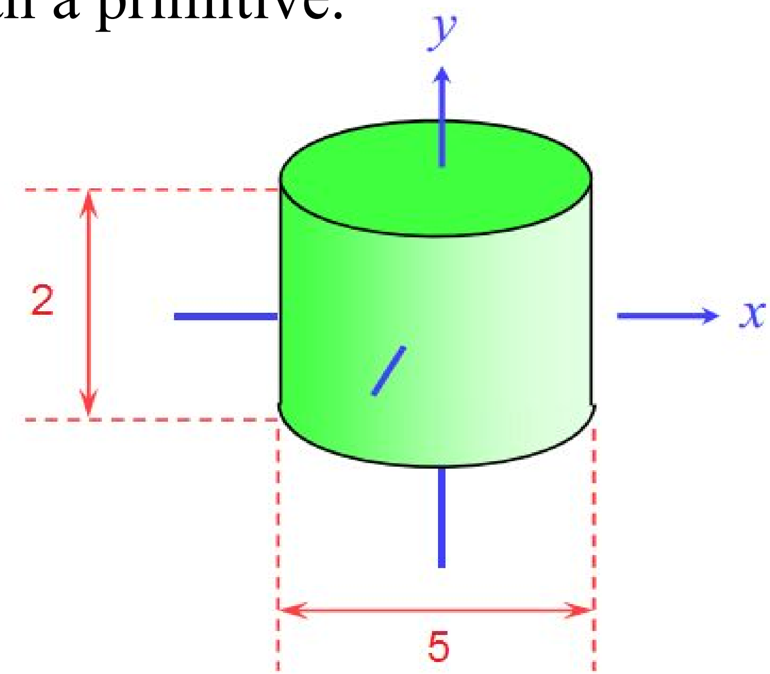
The cylinder "as it sees itself", in local coordinates

*

5	0	0	0
0	2	0	0
0	0	5	0
0	0	0	1

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

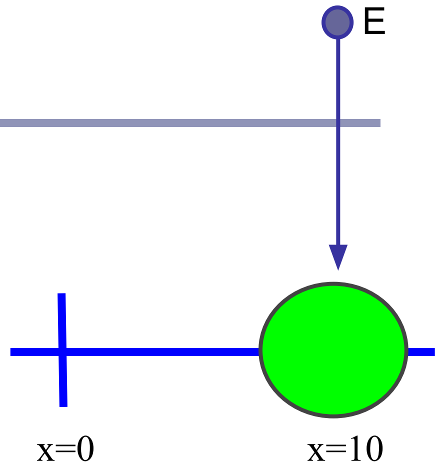
=



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*, $\mathtt{T}^{-1}$.



World coordinates

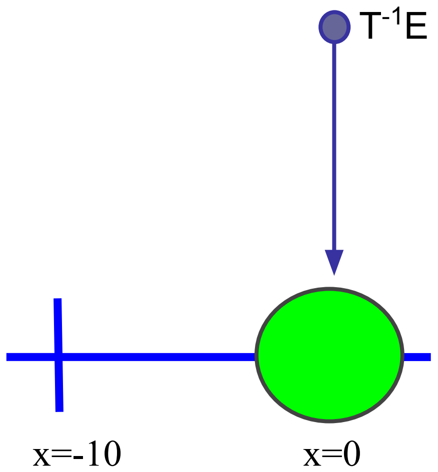
If the ray is defined by

$$\mathtt{P(t) = E + tD}$$

then the ray in local coordinates is defined by

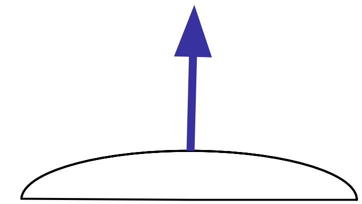
$$\mathtt{T^{-1}(P(t)) = T^{-1}(E) + t(T^{-1}{}_{3x3}D)}$$

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



Local coordinates

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*	$N = n$

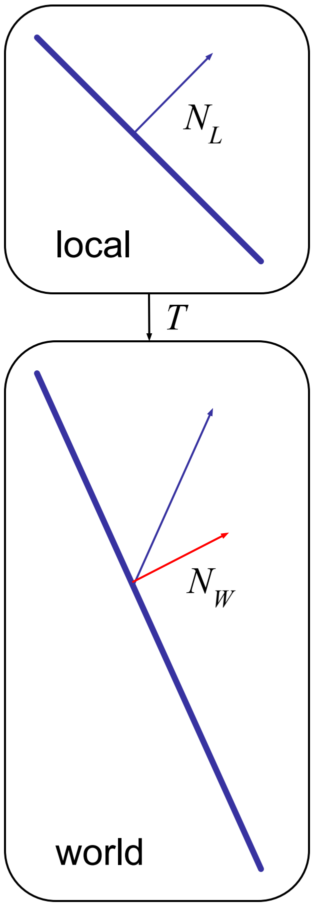
We use the normal for color, reflection, refraction, shadow rays...

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.

Your scene graph and you

Many 2D GUIs today favor an event model in which events ‘bubble up’ from child windows to parents. This is sometimes mirrored in a scene graph.

- Ex: a child changes size, changing the size of the parent’s bounding box
- Ex: the user drags a movable control in the scene, triggering an update event

If you do choose this approach, consider using the *Model View Controller* or *Model View Presenter* design pattern. 3D geometry objects are good for displaying data but they are not the proper place for control logic.

- For example, the class that stores the geometry of the rocket should not be the same class that stores the logic that moves the rocket.
- <u>Always separate logic from representation</u>.

Your scene graph and you

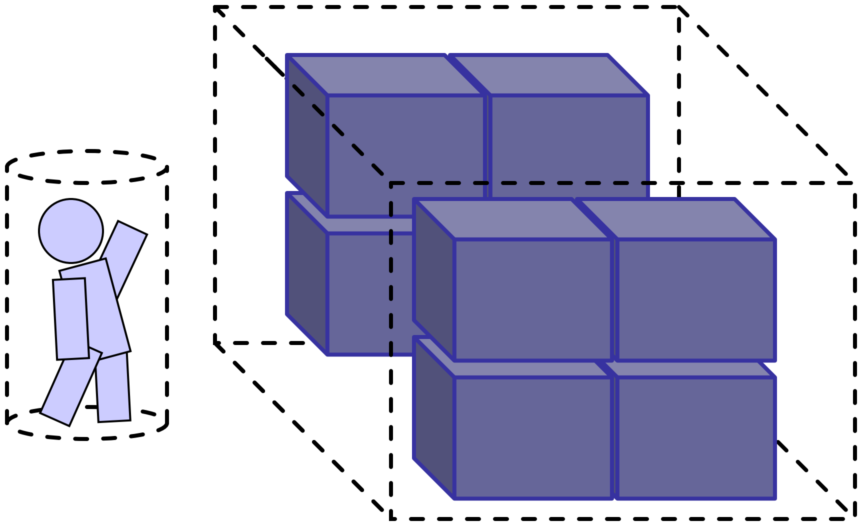
A common optimization derived from the scene graph is the propagation of *bounding volumes*.

Nested bounding volumes allow the rapid culling of large portions of geometry

- Test against the bounding volume of the top of the scene graph and then work down.

Great for…

- Collision detection between scene elements
- Culling before rendering
- Accelerating ray-tracing



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

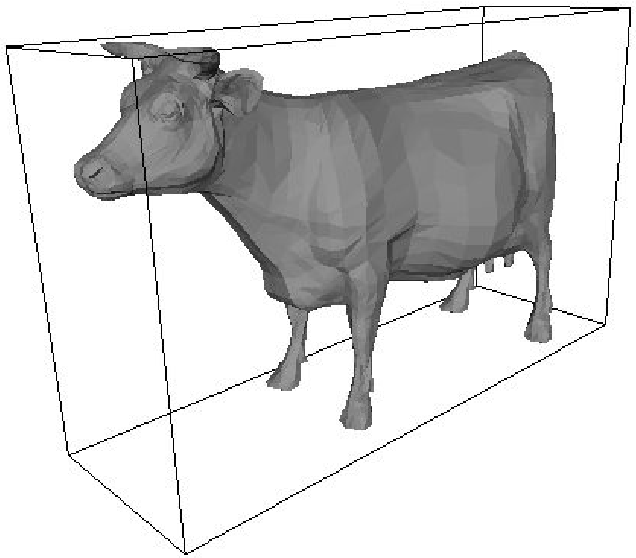
- max and min of x/y/z.

Bounding spheres

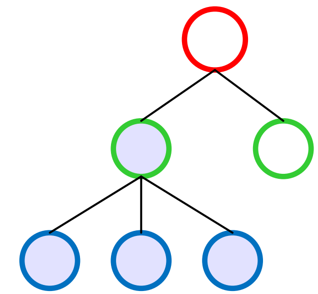
- max of radius from some rough center

Bounding cylinders

- common in early FPS games

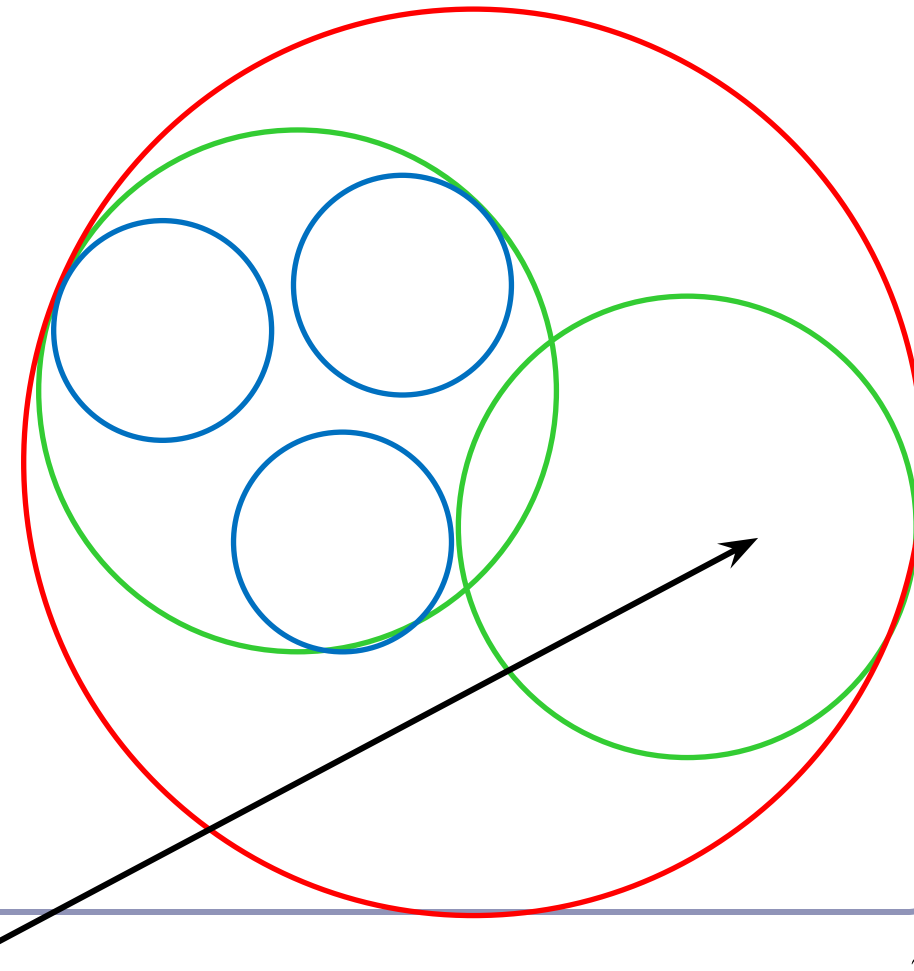


Bounding volumes in hierarchy



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

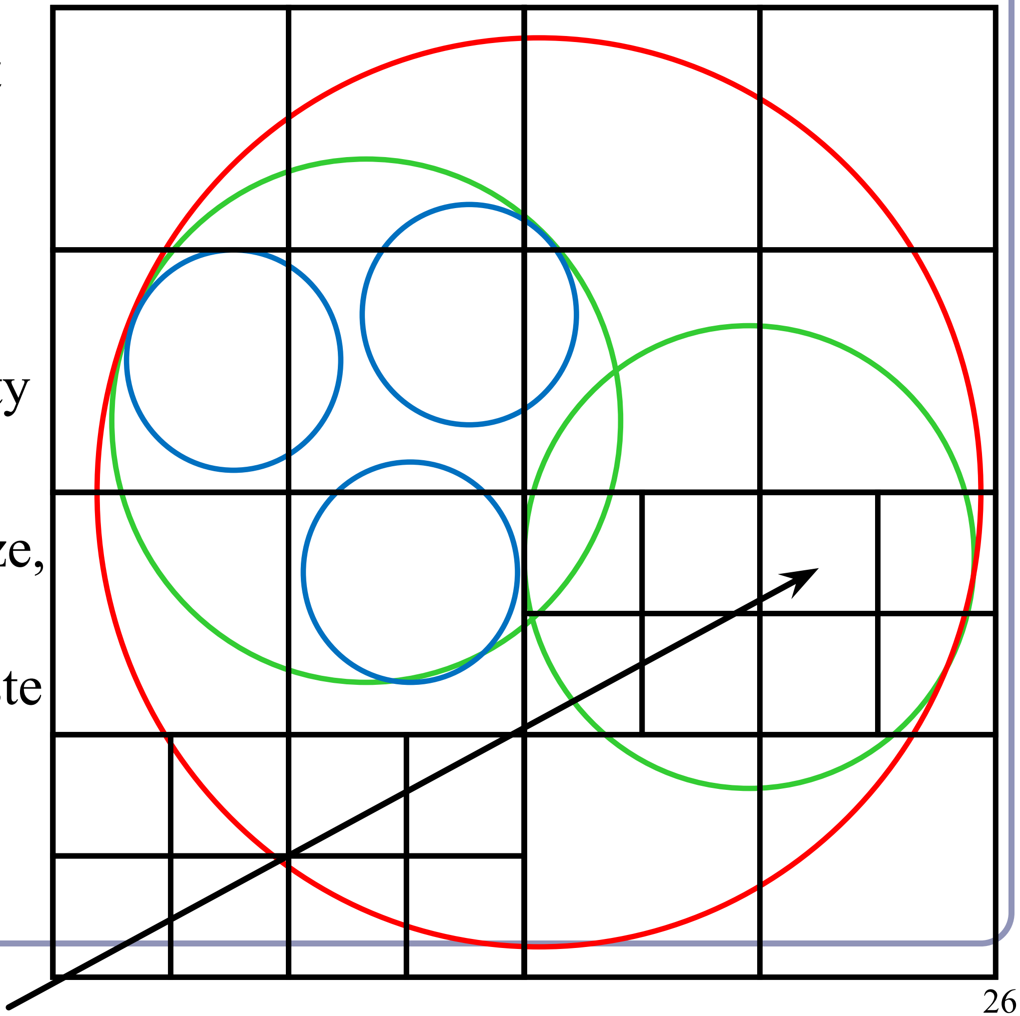
- Pro: Rays can skip subsections of the hierarchy
- Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



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



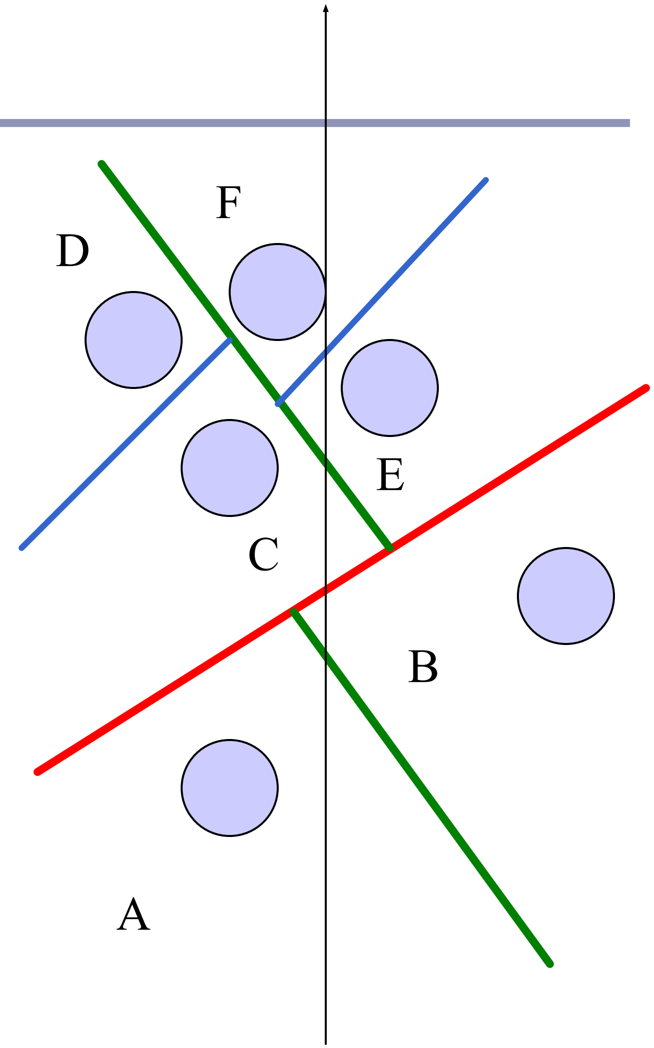
Popular acceleration structures: BSP Trees

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 *k*d-tree has O(*n* log *n*) insertion time (but this is very optimizable by domain knowledge) and $O(n^{2/3})$ search time.
- *k*d-trees don't suffer from the mathematical slowdowns of BSPs because their planes are always axis-aligned.

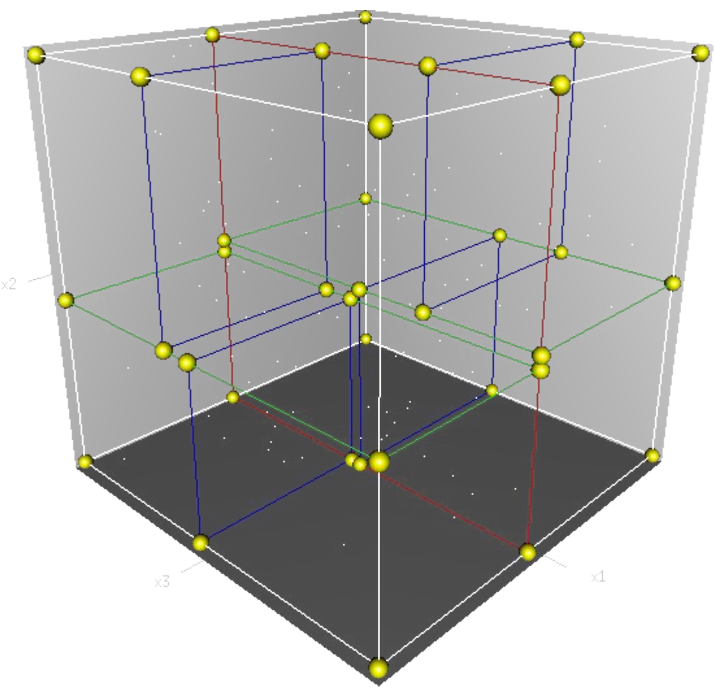


Image from Wikipedia, bless their hearts.

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

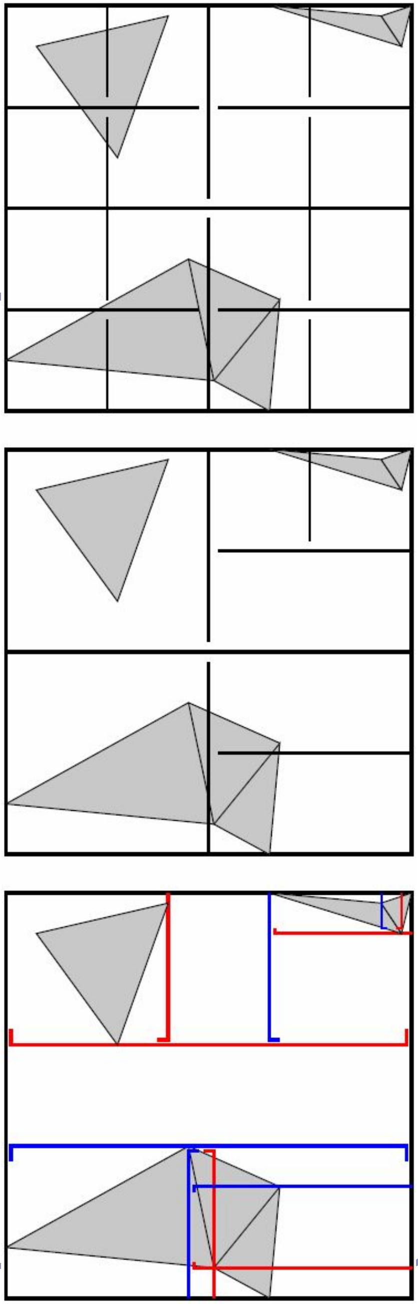


Image from Wächter and Keller’s paper, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics (2006)

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

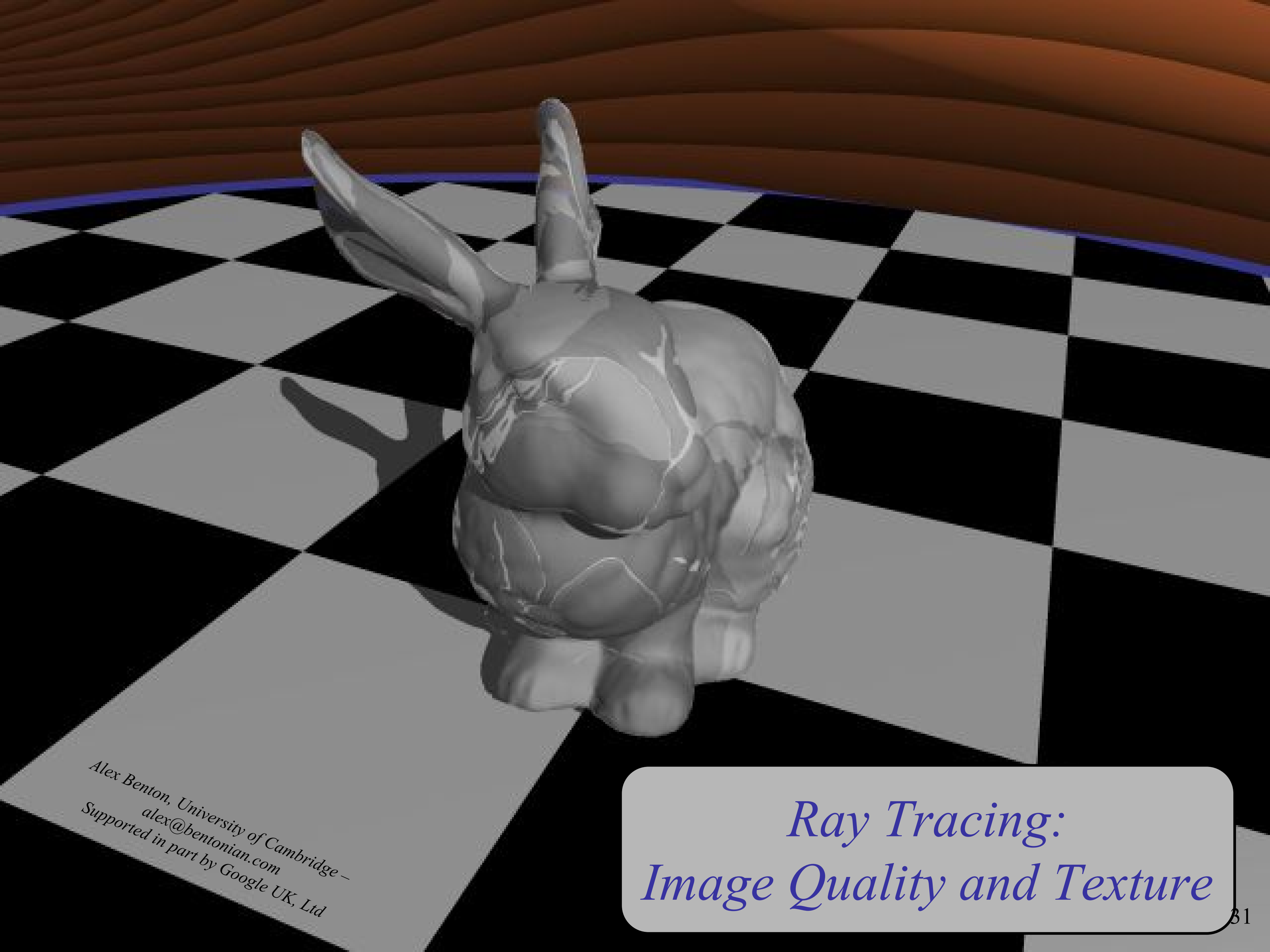
http://mathworld.wolfram.com/BarycentricCoordinates.html

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



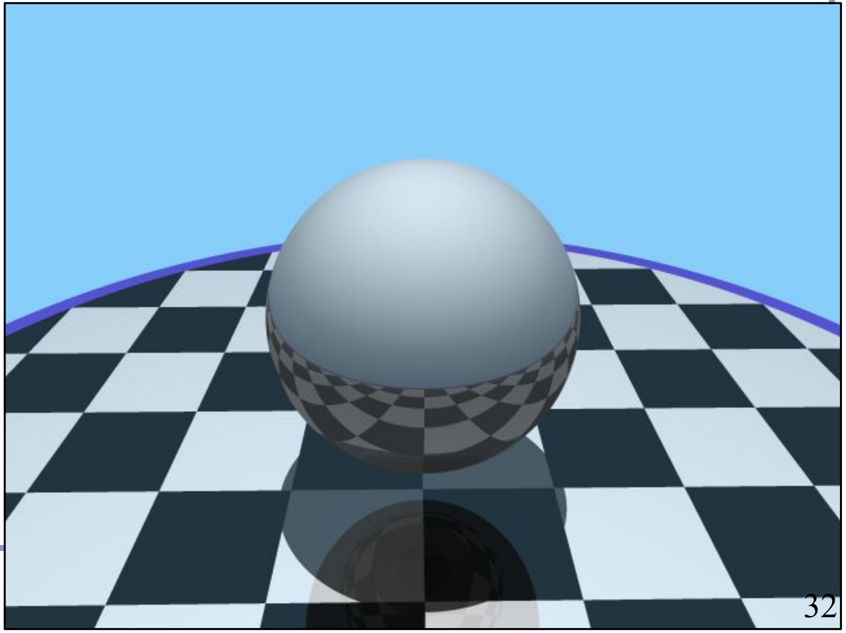
Ray Tracing: Image Quality and Texture

Alex Benton, University of Cambridge –
alex@bentonian.com
Supported in part by Google UK, Ltd

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

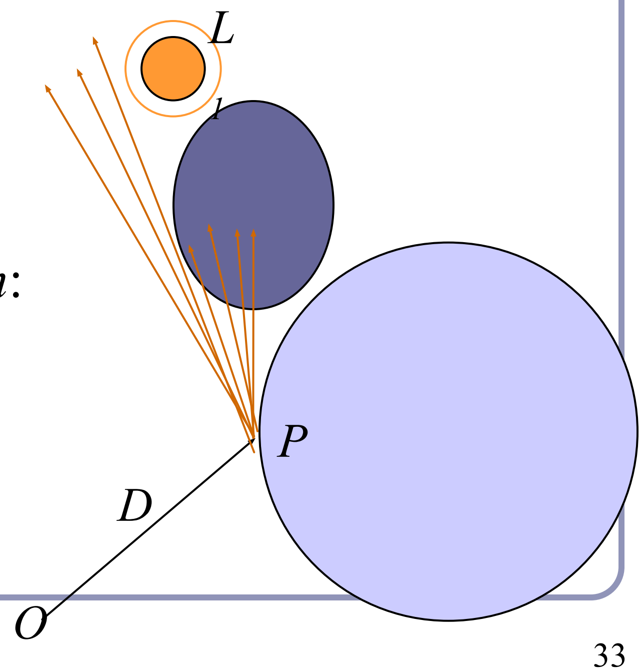
Shadows in nature are not sharp because light sources are not infinitely small.

- Also because light scatters, etc.

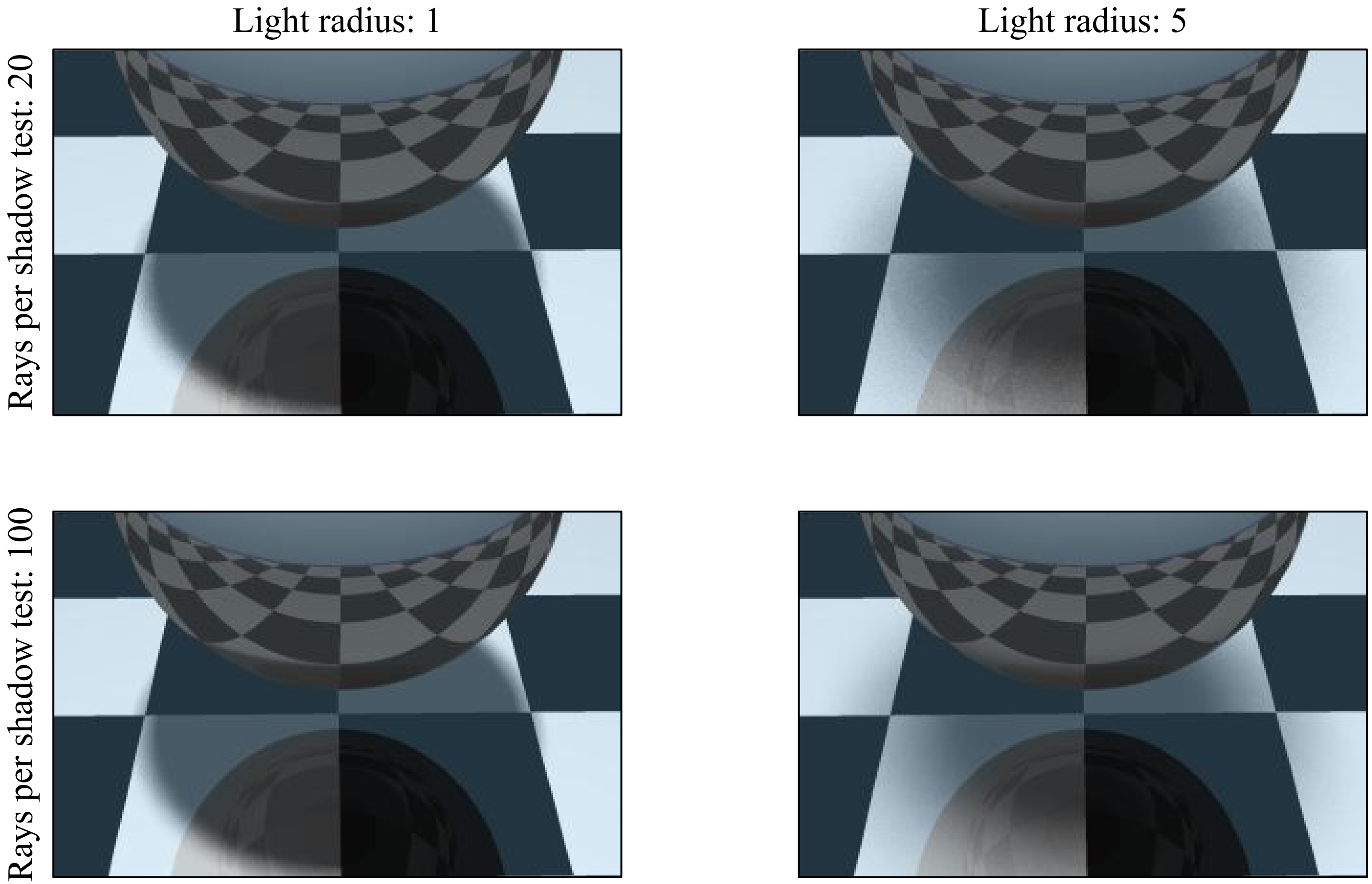
For lights with volume, fire many rays, covering the cross-section of your illuminated space.

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

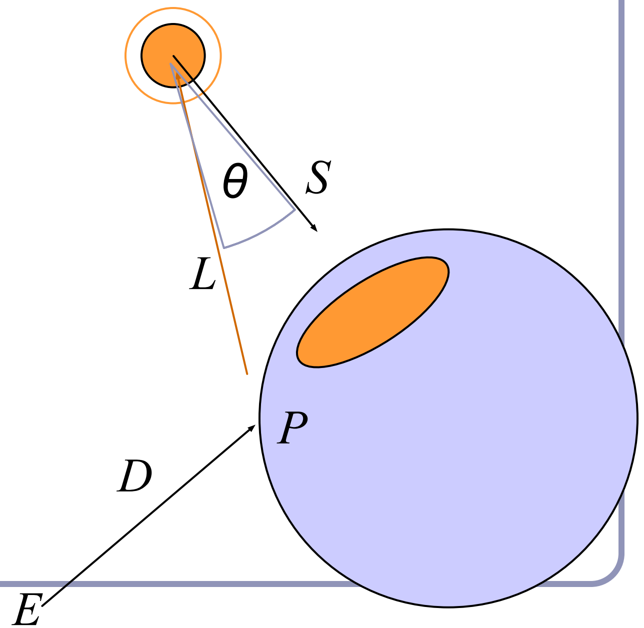


All images anti-aliased with 4x supersampling
Distance to light in all images: 20 units

Raytraced spotlights

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

- 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. $((L \cdot S > \cos(15^\circ))\ ?\ 1 : 0)$.



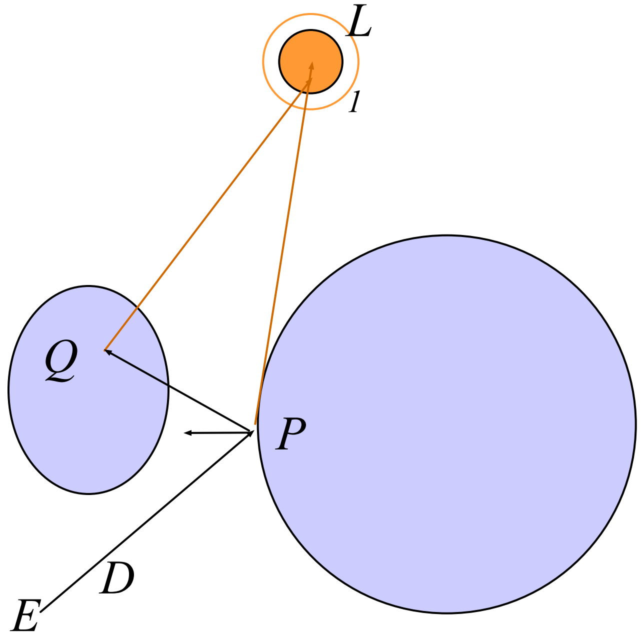
Reflection

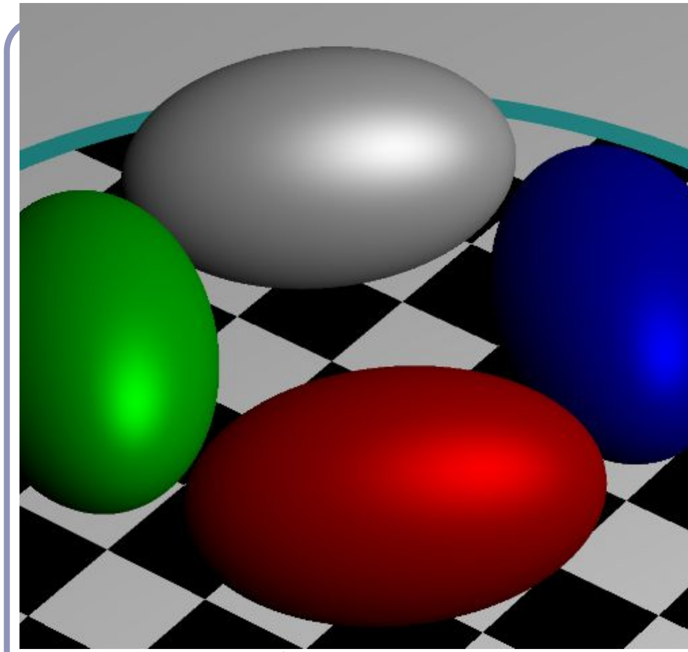
Reflection rays are calculated as:

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

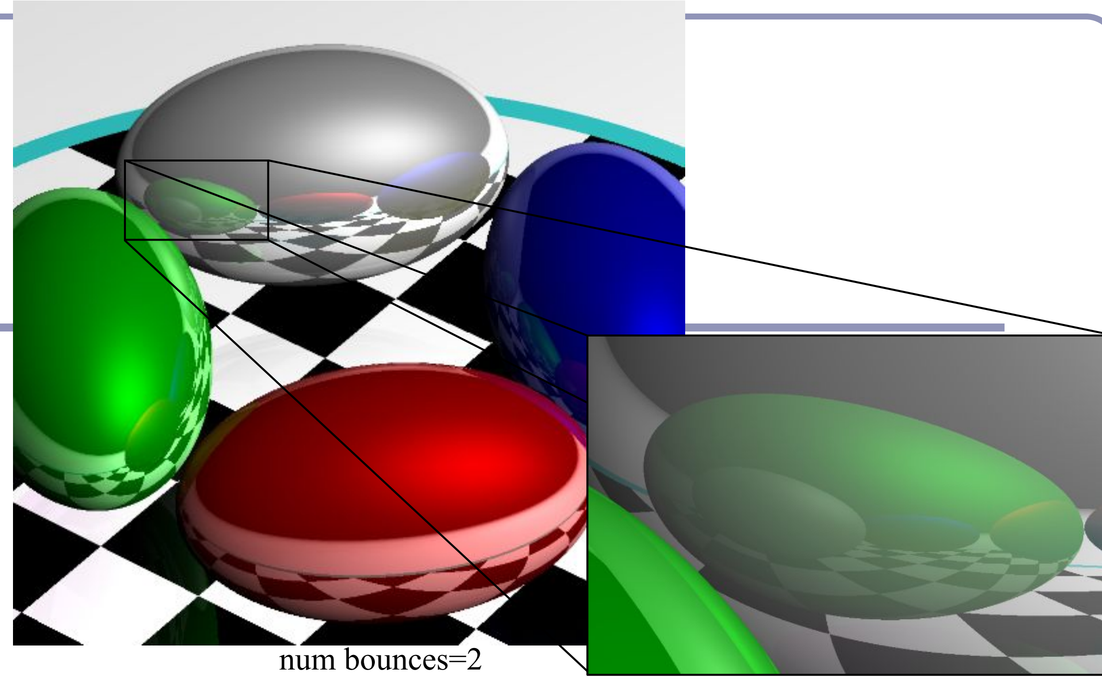
- Finding the reflected color is a recursive raycast.
- Reflection has *scene-dependant* performance impact.

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

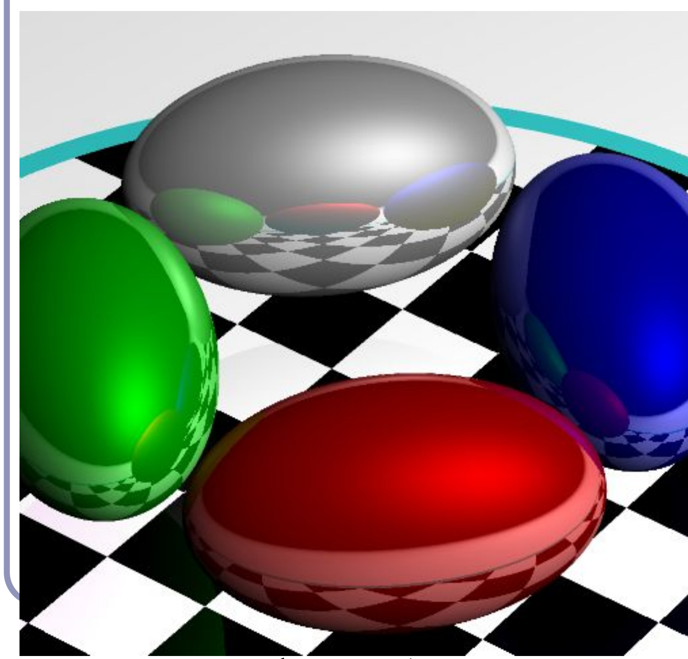




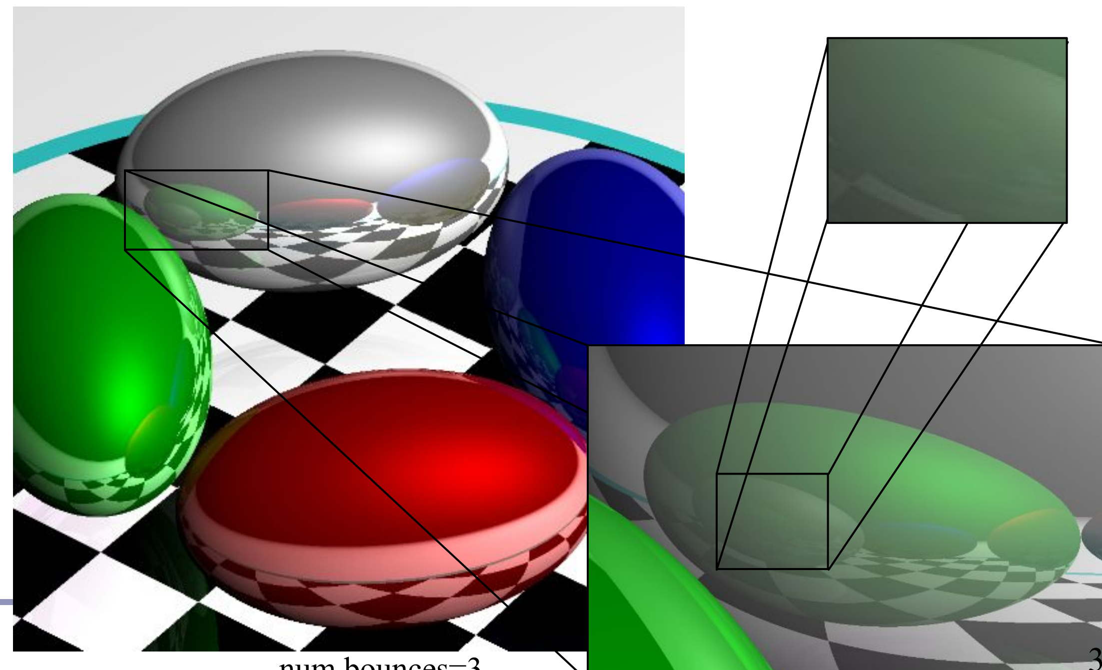
num bounces=0



num bounces=2



num bounces=1

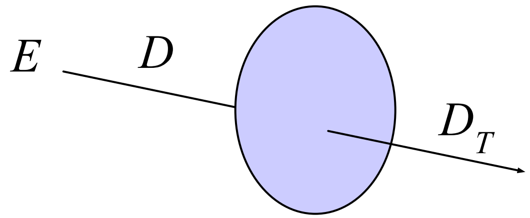


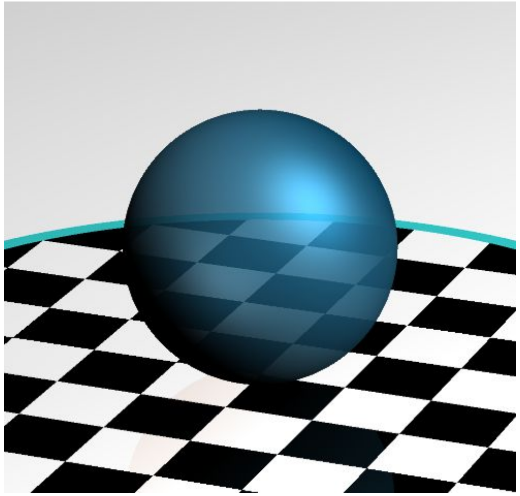
num bounces=3

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 1x<u>4</u> 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."

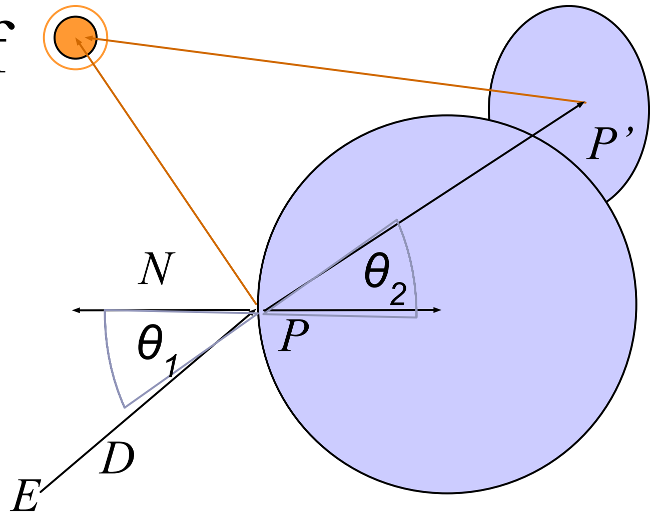
Historical note: this formula has been attributed to Willebrord Snell (1591-1626) and René Descartes (1596-1650) but first discovery goes to Ibn Sahl (940-1000) of Baghdad.

Refraction in ray tracing

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

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

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.



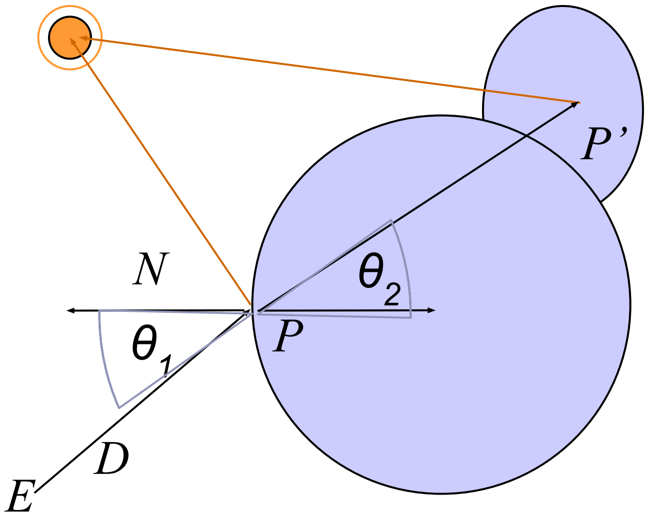
Refraction in ray tracing

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



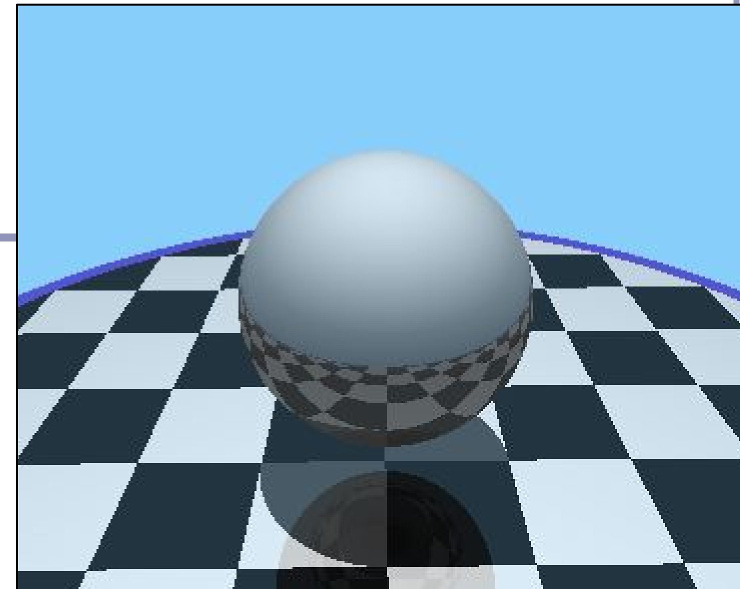
Aliasing

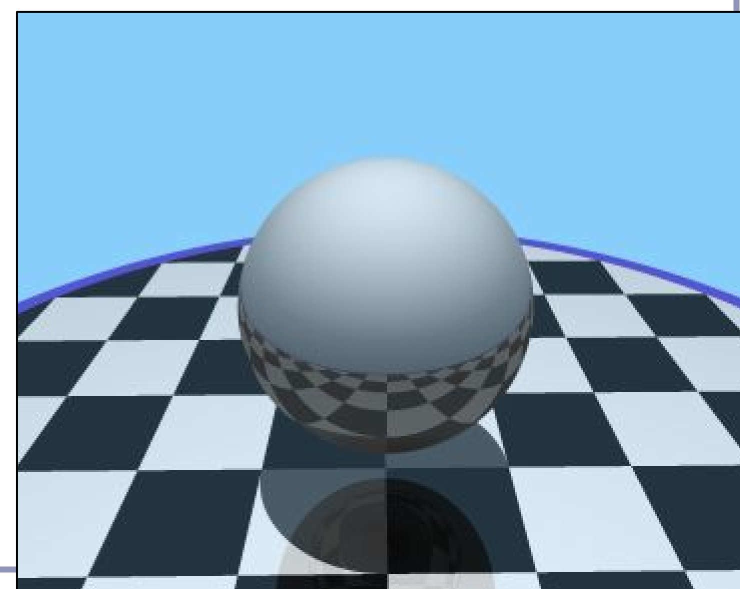
aliasing
/ˈeɪlɪəsɪŋ/
noun: **aliasing**
1. PHYSICS / TELECOMMUNICATIONS

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

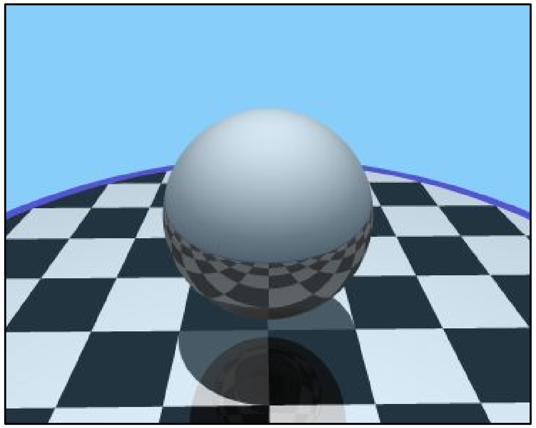
"high-frequency sounds are prone to aliasing"
2. COMPUTING

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.

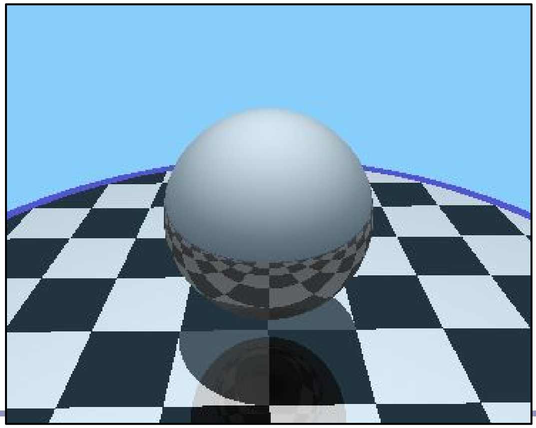




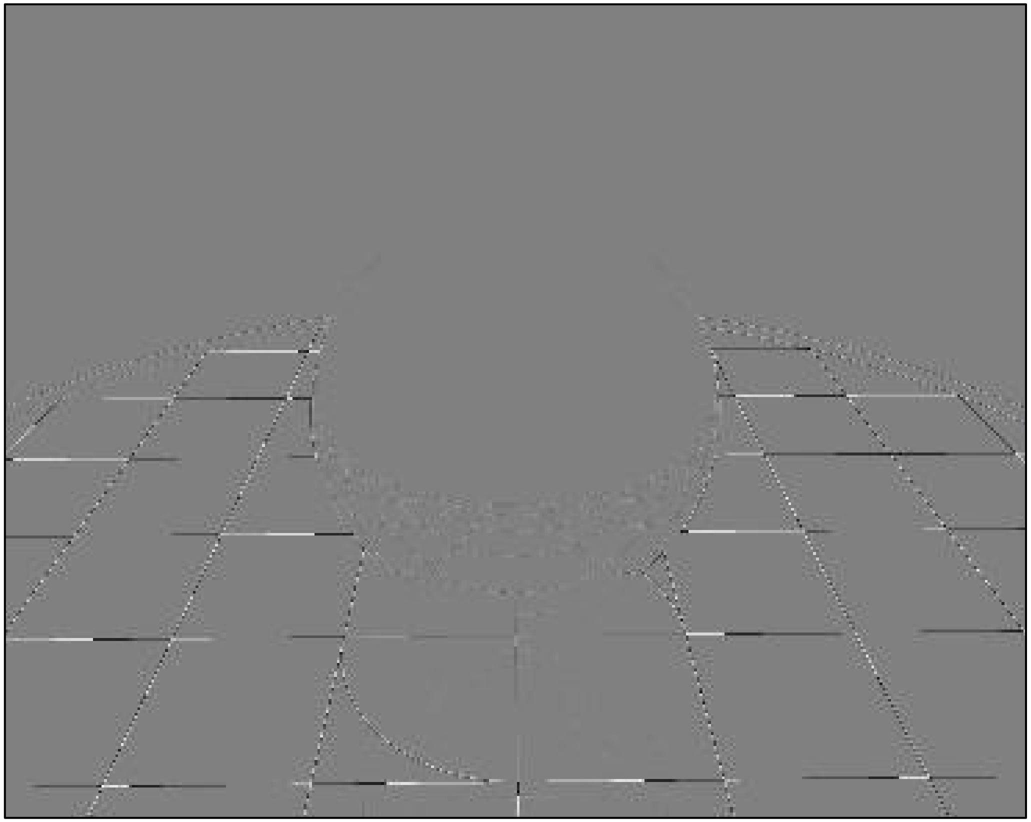
Aliasing



-



=



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.

To avoid heavy computational load And also avoid sub-super-sampling artifacts, consider using *jittered super-sampling*.

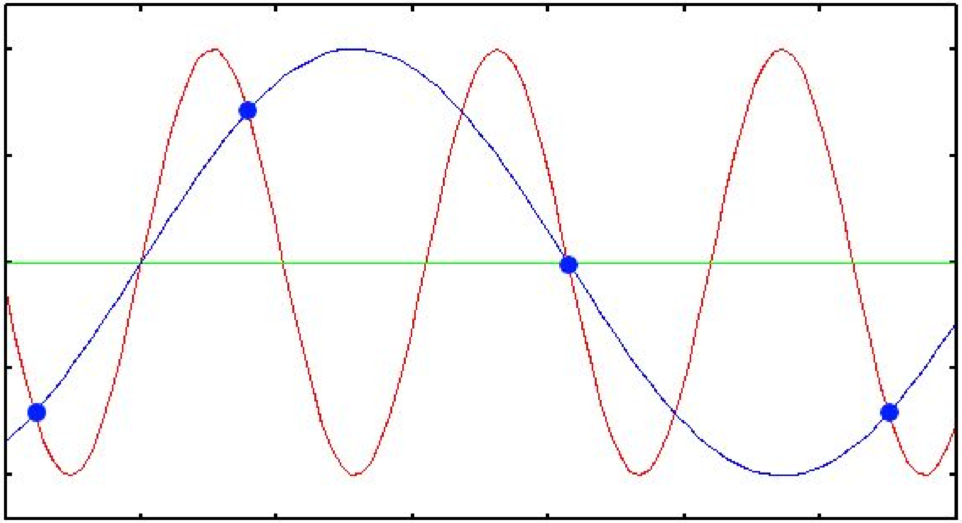
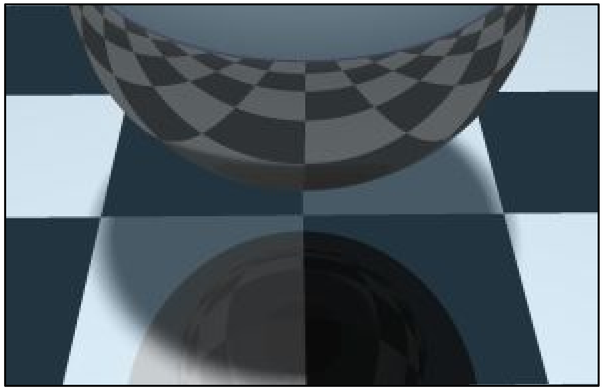


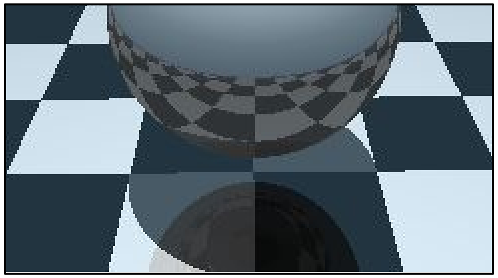
Image source: www.svi.nl

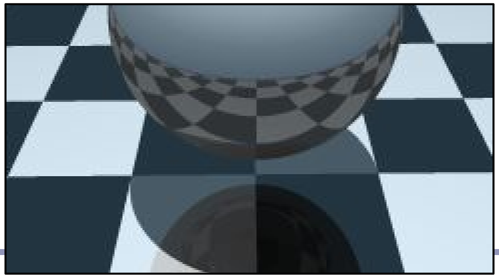
Lecture note: Four printed slides removed here, reviewing antialiasing from last year's notes.

Applications of super-sampling

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







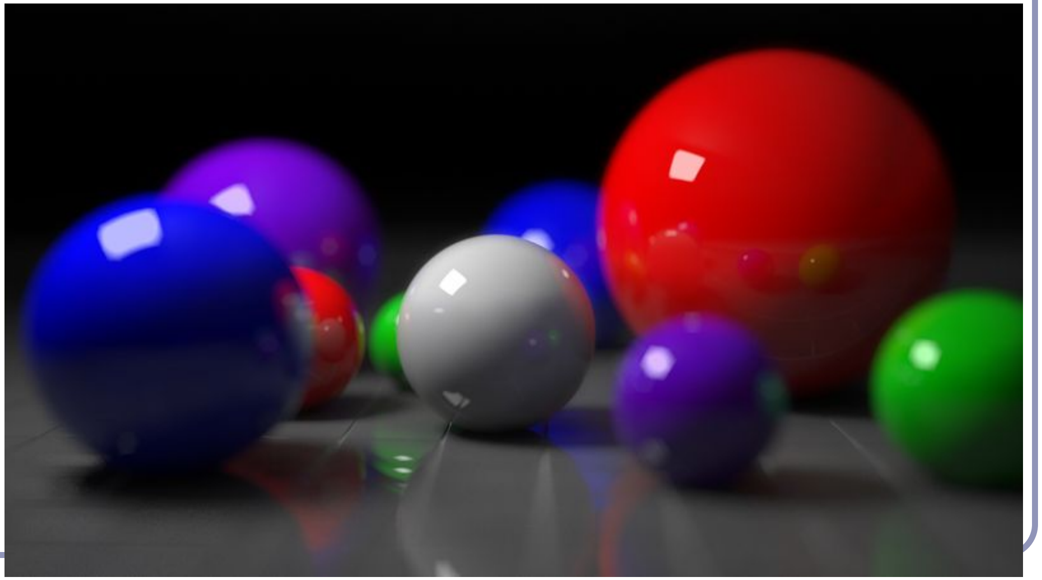
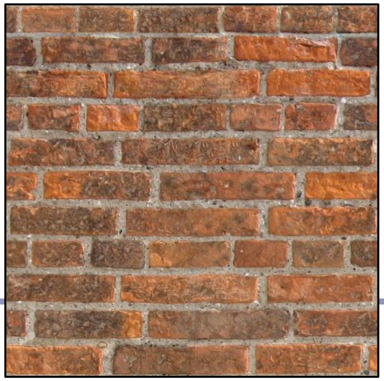
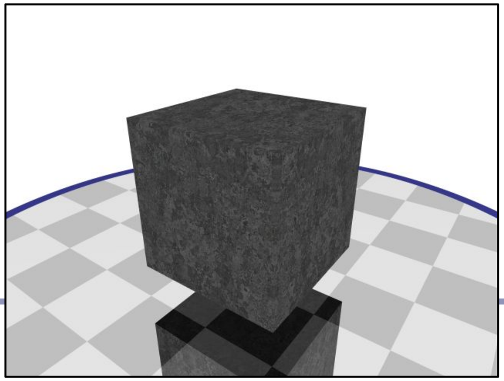
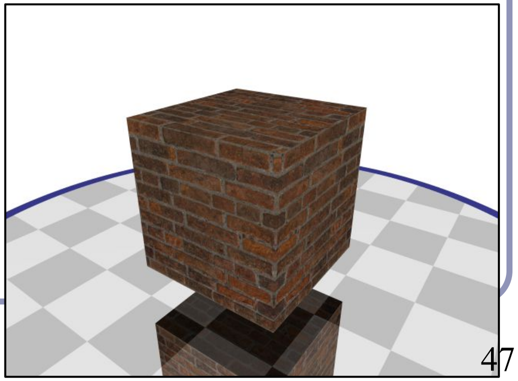


Image credit:
http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

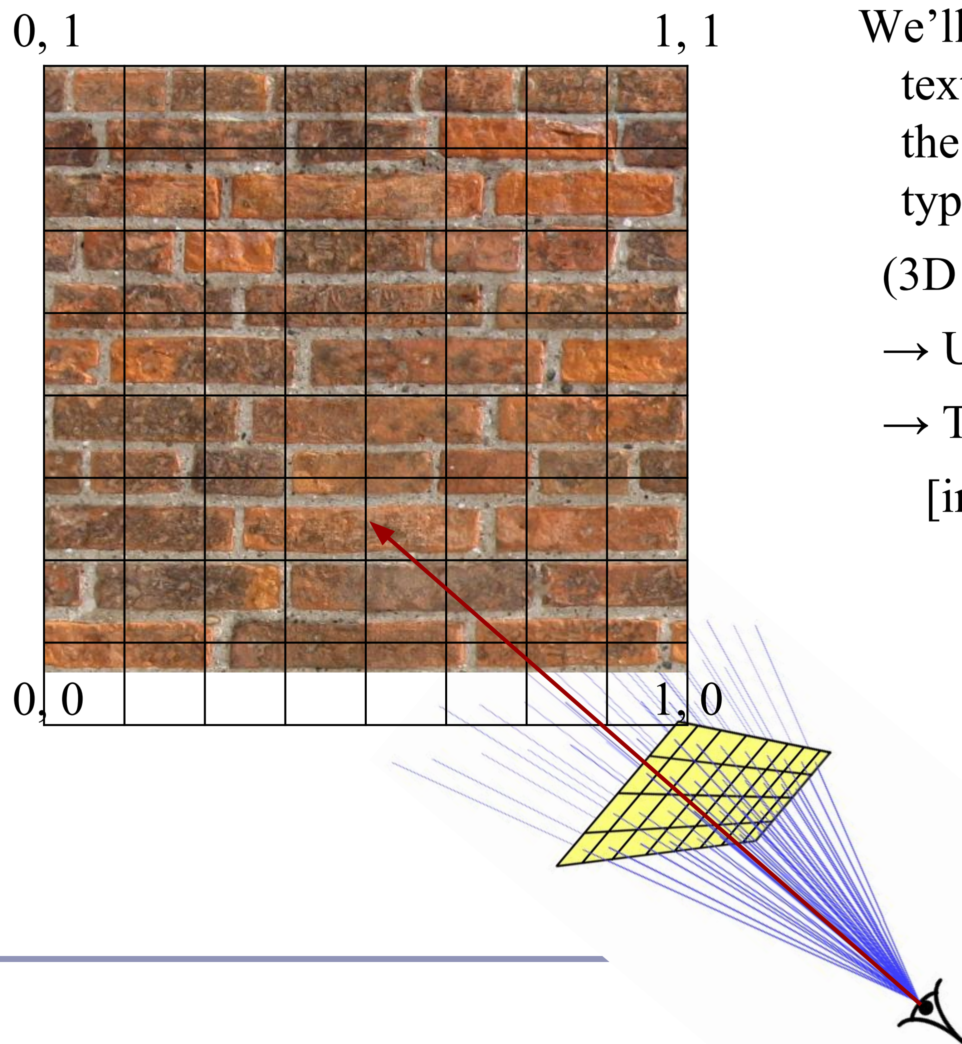
Texture mapping

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

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.

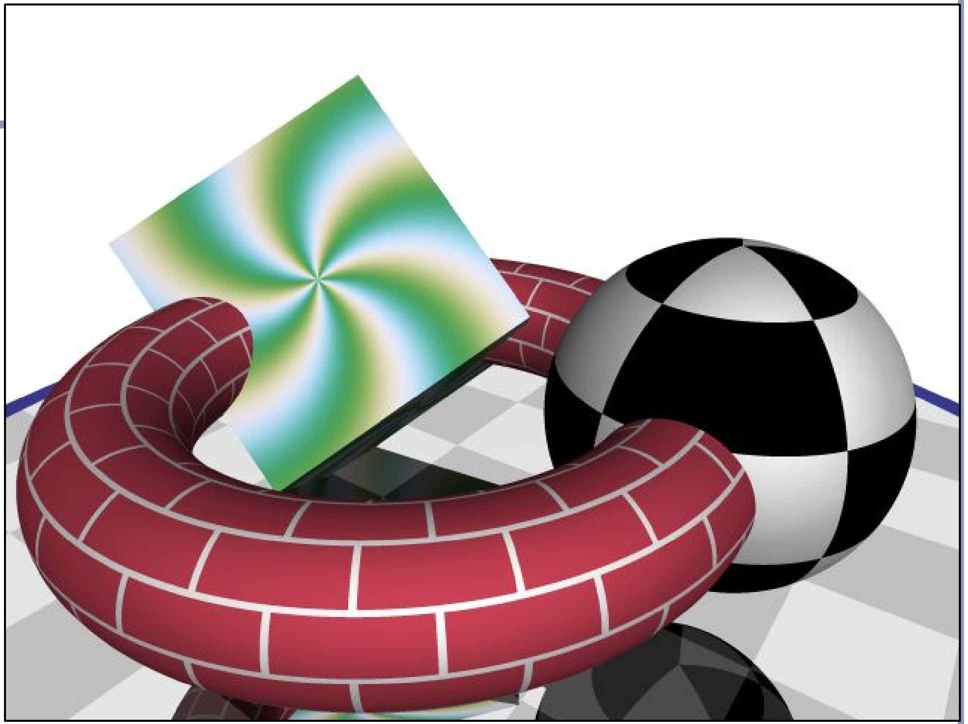
Procedural texture

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):

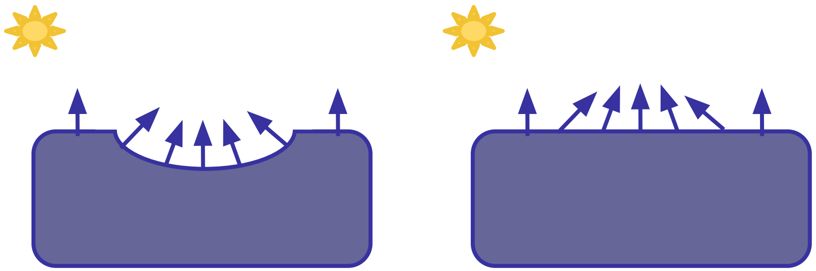
```
tx = (int) 10 * u
ty = (int) 10 * v
oddity = (tx & 0x01) == (ty & 0x01)
edge = ((10 * u - tx < 0.1) && oddity) || (10 * v - ty < 0.1)
return edge ? WHITE : RED
```



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

Non-color textures: 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.

Non-color textures: normal mapping

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/

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):

$$f(P) = (X_P^2+Z_P^2 + noise(P)) \bmod 1$$

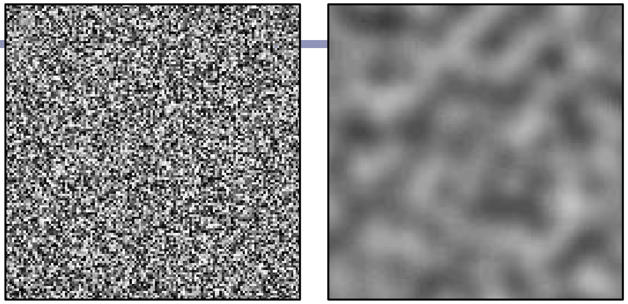
where *noise(P)* is a function that maps 3D coordinates in space to scalar values chosen at random.

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

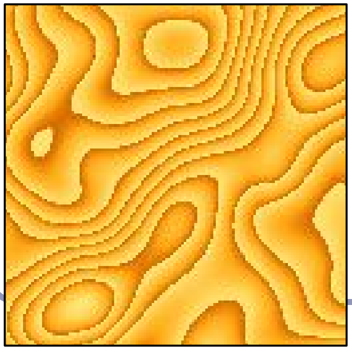
Perlin noise

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

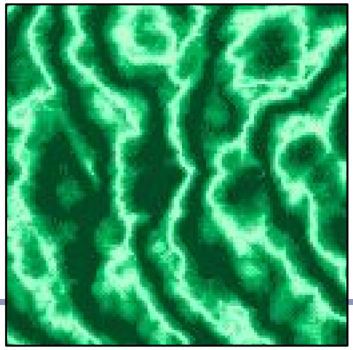
- It is a *band-limited repeatable pseudorandom* function (in the words of its author, Ken Perlin)
- It is bounded within a range close [-1, 1]
- It varies continuously, without discontinuity
- It has regions of relative stability
- It can be initialized with random values, extended arbitrarily in space,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



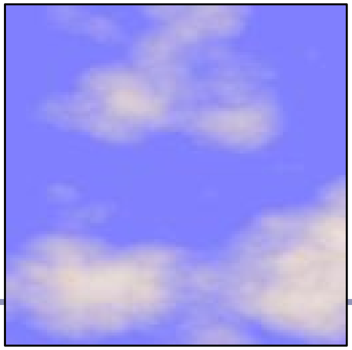
Non-coherent noise (left) and Perlin noise (right)
Image credit: Matt Zucker



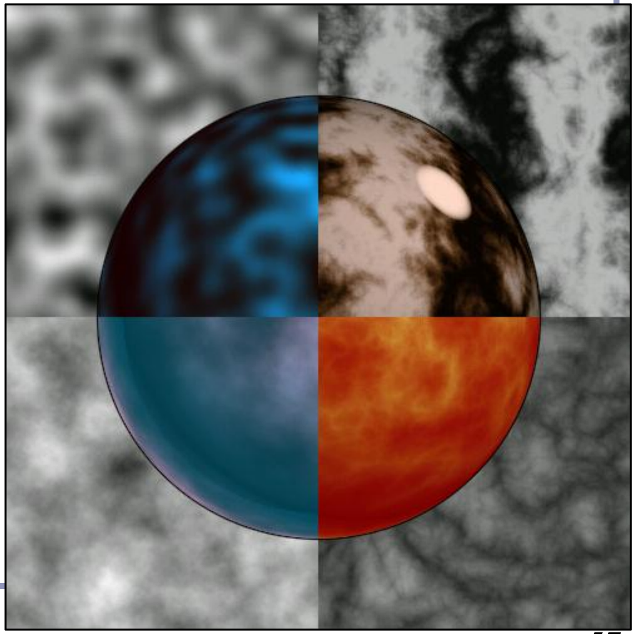
Matt Zucker



Matt Zucker



Matt Zucker

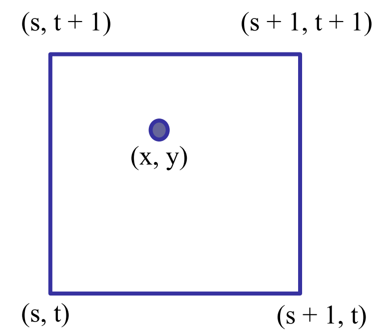


Ken Perlin

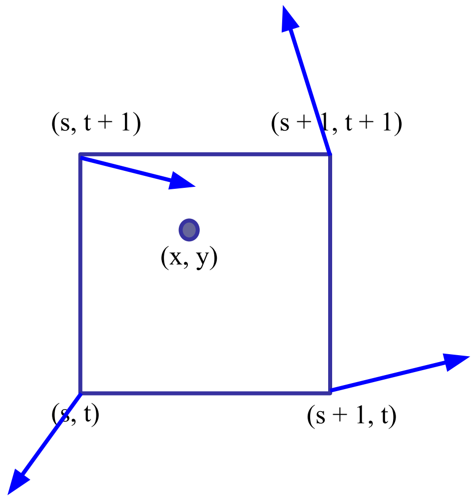
Perlin noise 1

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.

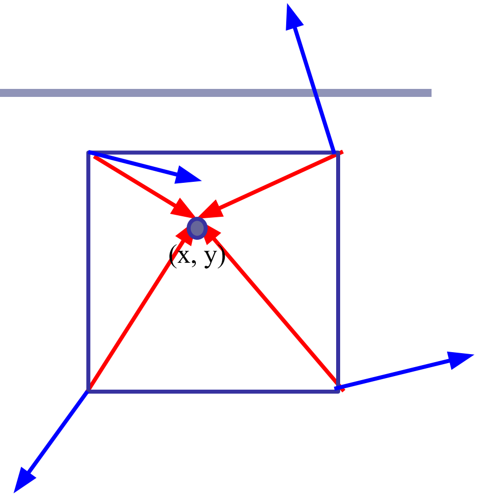


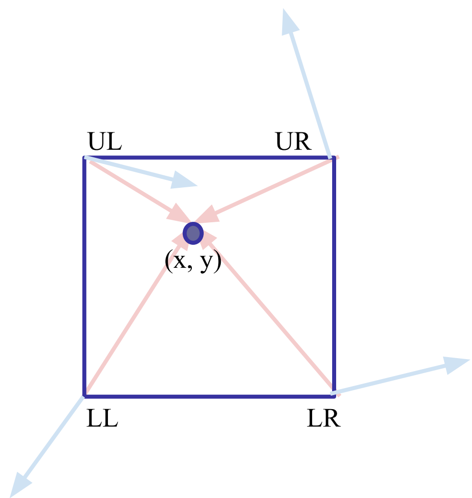
These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Perlin noise 2

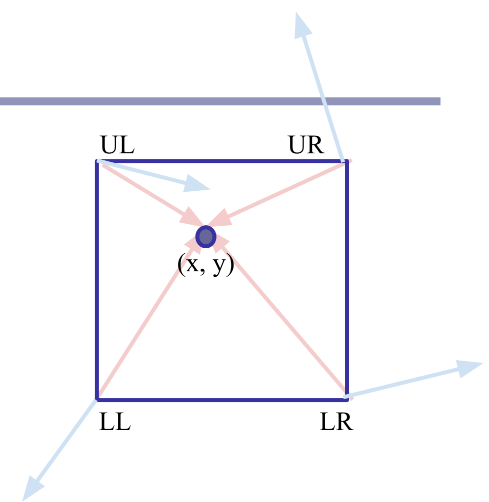
For each of the four corners, take the dot product of the random seed vector with the vector from that corner to *(x, y)*. This gives you a unique scalar value per corner.

- 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].





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:

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

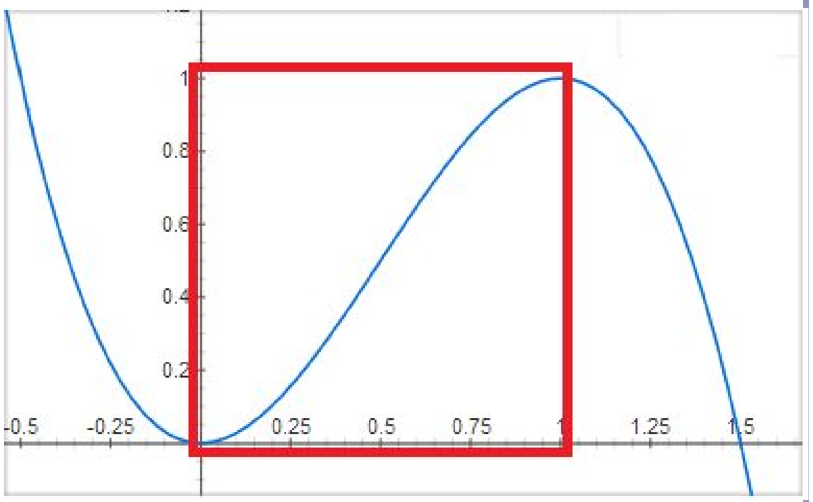
$$U(x, y) = UL + S(x - floor(x))(UR - UL)$$

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

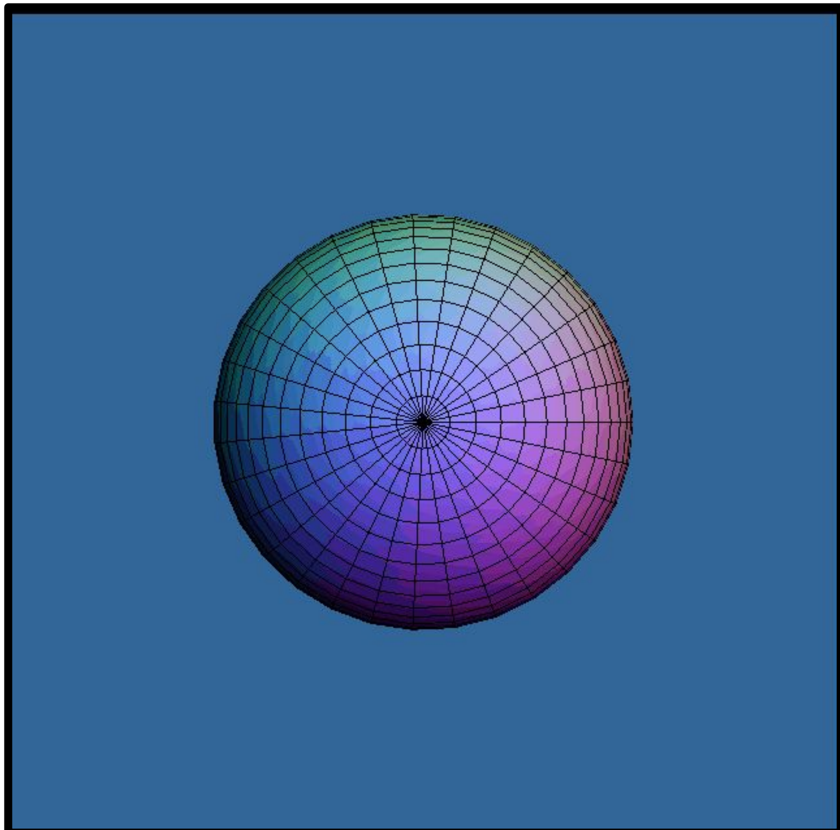
Noise amplitude
1 → 3

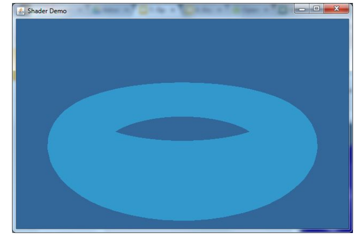
References

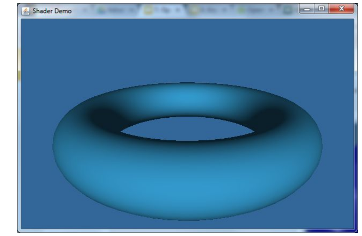
Ray tracing
Peter Shirley, Steve Marschner. *Fundamentals of Computer Graphics*. Taylor & Francis, 21 Jul 2009
Hughes, Van Dam et al. *Computer Graphics: Principles and Practice*. Addison Wesley, 3rd edition (10 July 2013)

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

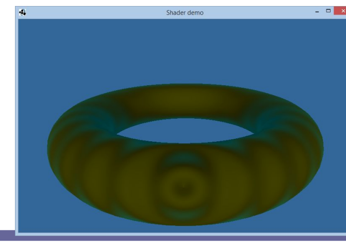
Perlin noise
http://www.noisemachine.com/talk1/
http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

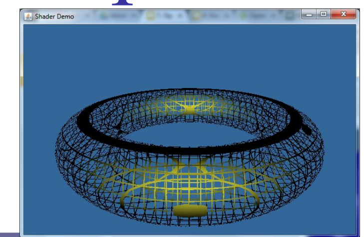




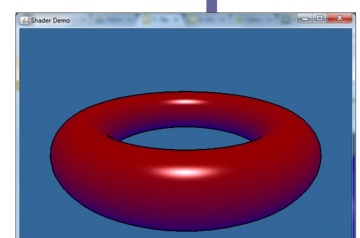


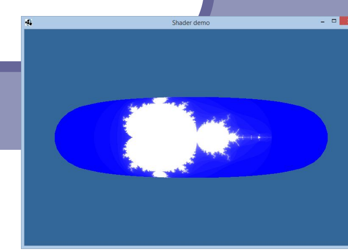
Advanced Graphics





OpenGL and Shaders I





Alex Benton, University of Cambridge – alex@bentonian.com

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)

Mobile GPUs

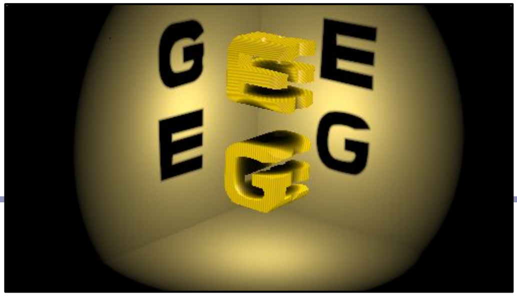
- OpenGL ES 1.0-3.2
  - A stripped-down version of OpenGL
  - Removes functionality that is not strictly necessary on mobile devices (like recursion!)
- Devices
  - iOS: iPad, iPhone, iPod Touch
  - Android phones
  - PlayStation 3, Nintendo 3DS, and more

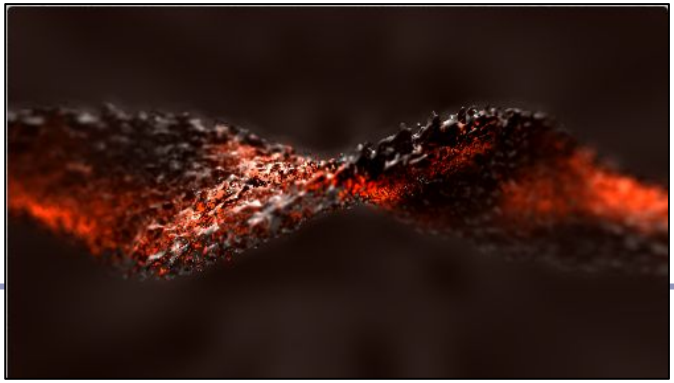
OpenGL ES 2.0 rendering (iOS)

WebGL

- JavaScript library for 3D rendering in a web browser
  - Based on OpenGL ES 2.0
  - Many supporting JS libraries
  - Even gwt, angular, dart...
- Most modern browsers support WebGL, even mobile browsers
  - Enables in-browser 3D games
  - Enables realtime experimentation with glsl shader code

Samples from Shadertoy.com





Vulkan

Vulkan is the next generation of OpenGL: a cross-platform open standard aimed at pure performance on modern hardware

Compared to OpenGL, Vulkan--

- Reduces CPU load
- Has better support of multi-CPU core architectures
- Gives finer control of the GPU

The Talos Principle running on Vulkan (via www.geforce.com)

--but--

- Drawing a few primitives can take 1000s of lines of code
- Intended for game engines and code that must be very well optimized

OpenGL in Java - choices

***JOGL*: "Java bindings for OpenGL"**

jogamp.org/jogl

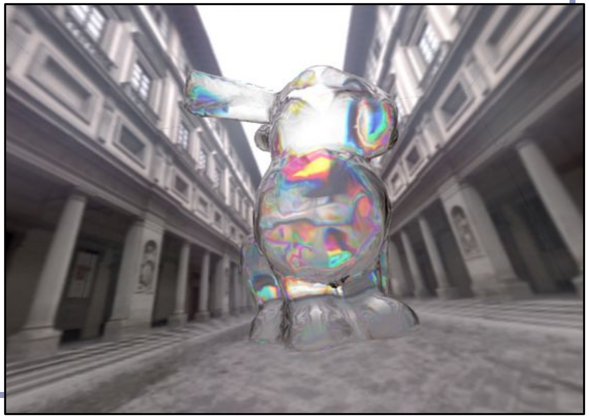
JOGL apps can be deployed as applications or as *applets*, making it suitable for educational web demos and cross-platform applications.

- 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"**

www.lwjgl.org

LWJGL is targeted at game developers, so it's got a 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.

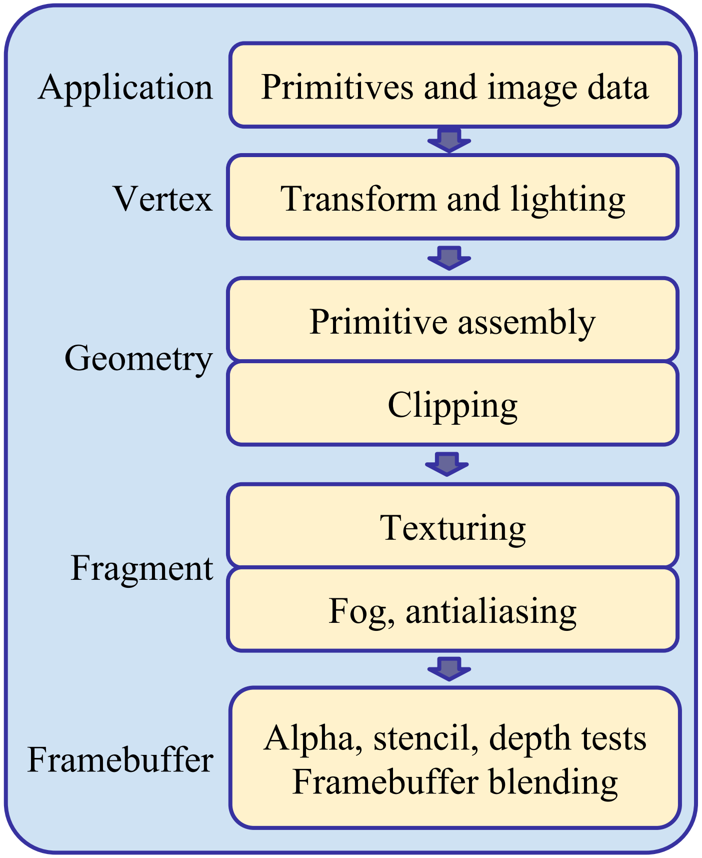
You'll write shaders in the OpenGL shader language, GLSL.

You'll write *vertex* and *fragment* shaders. (And maybe others.)

The OpenGL rendering pipeline

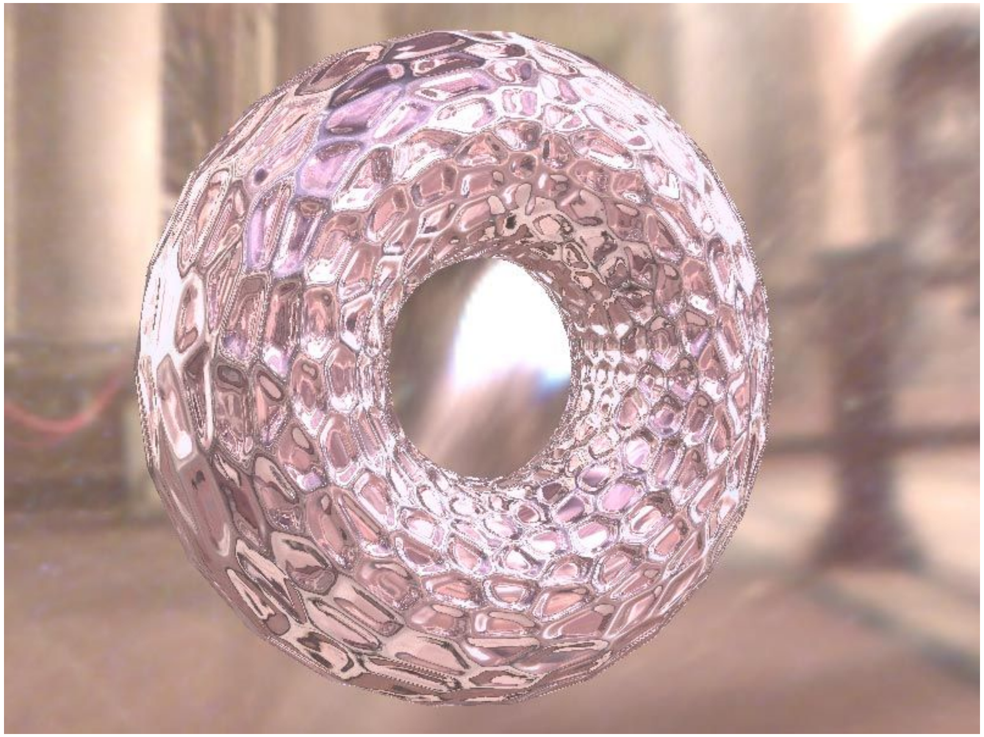
An OpenGL application assembles sets of *primitives*, *transforms* and *image data*, which it passes to OpenGL's GLSL shaders.

- *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*
(a massively simplified view)

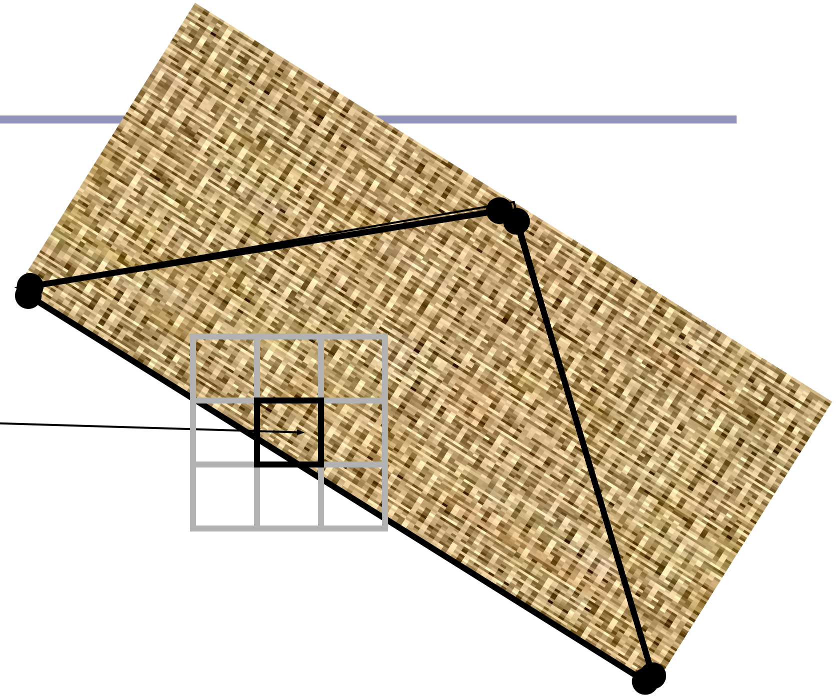
Shader gallery I



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

OpenGL: **Shaders**

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
- Vulkan's SPIR-V lets developers code in high-level GLSL but tune at the machine code level

GPUs typically have multiple processing units

That means that multiple shaders execute in parallel

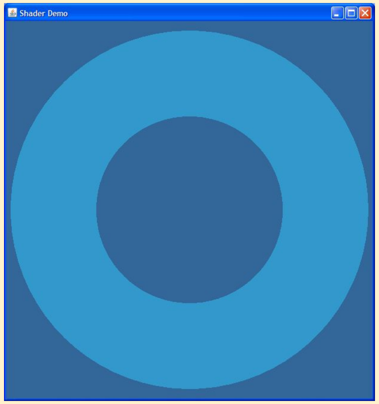
- We're moving away from the purely-linear flow of early "C" programming models

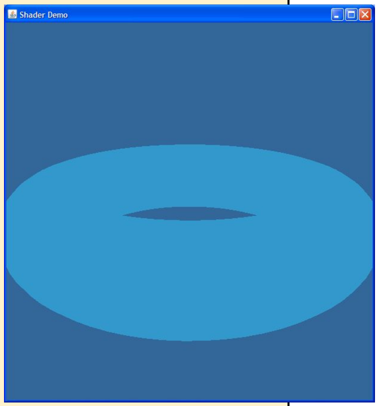
Shader example one – ambient lighting

```
#version 330

uniform mat4 mvp;

in vec4 vPos;

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

// Vertex Shader

```
#version 330

out vec4 fragmentColor;

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





// Fragment Shader

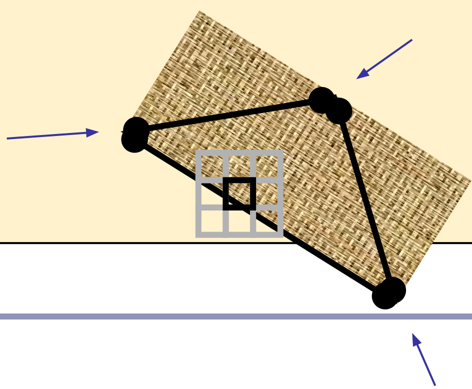
Vertex outputs become fragment inputs

Input

```
#version 330

uniform mat4 mvp;

in vec4 vPos;
out vec3 c;

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

Output

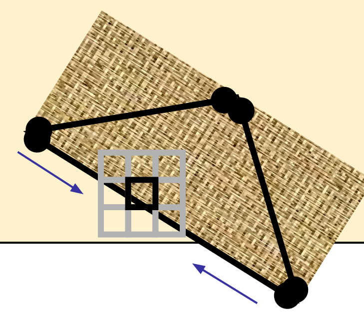
interpolation



Input

Color output

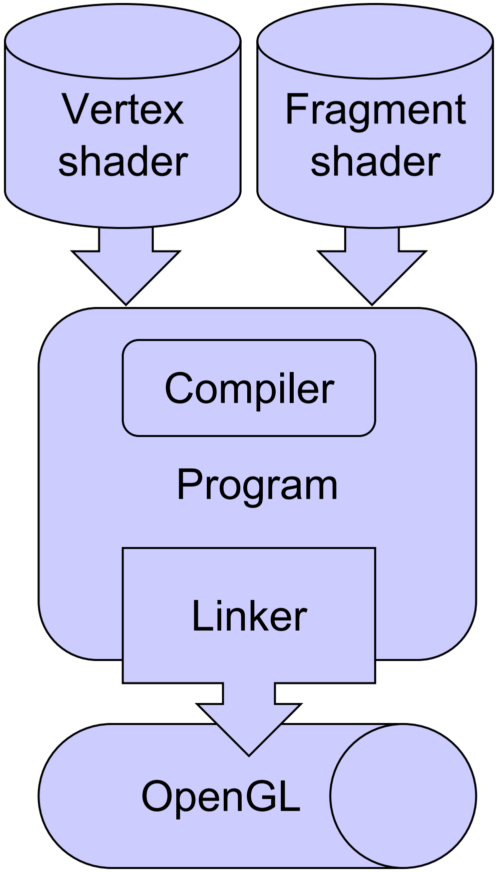
```
#version 330

in vec3 c;
out vec4 fragmentColor;

void main() {
  fragmentColor = vec4(c, 1);
}
```



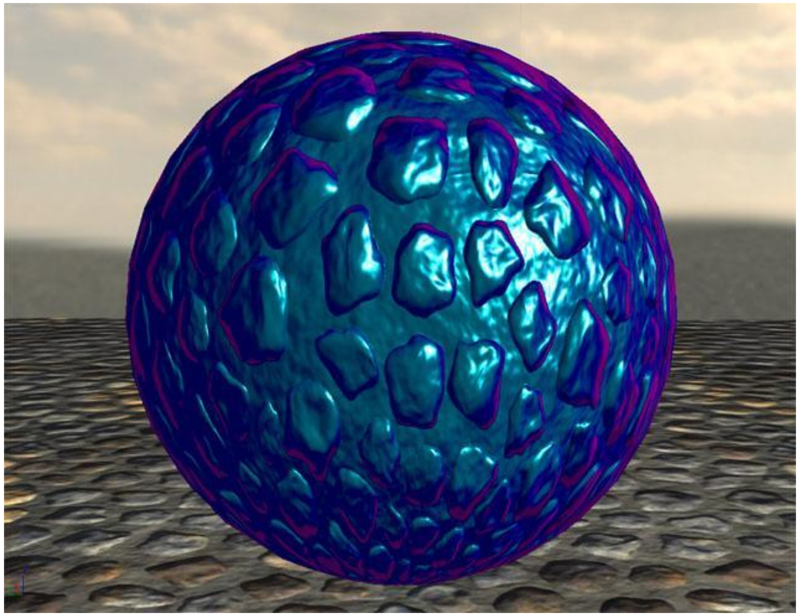
OpenGL / GLSL API - setup

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



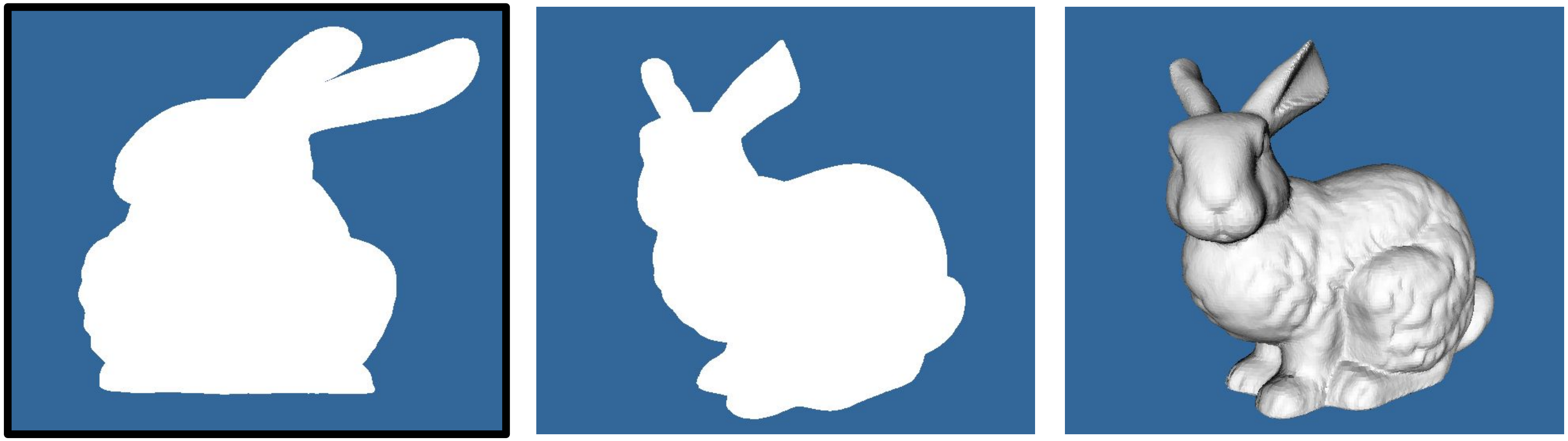
Above: Kevin Boulanger (PhD thesis, "*Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*", 2005)

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



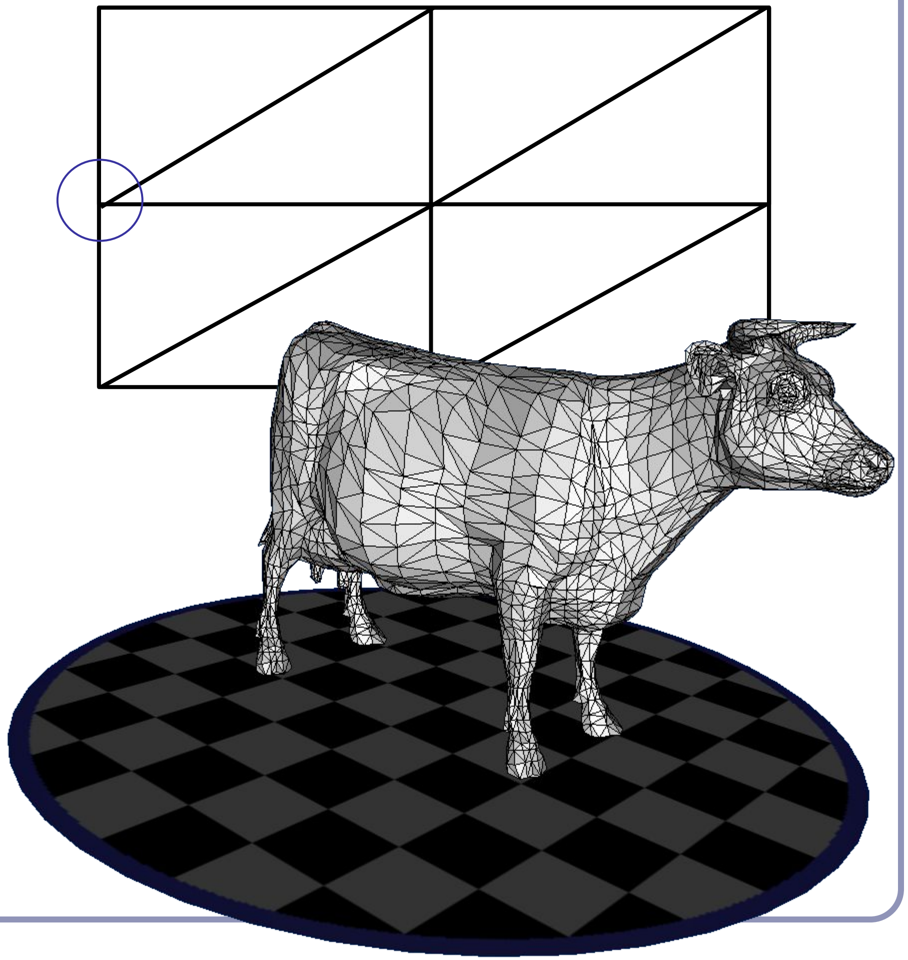
Geometry in OpenGL

The atomic datum of OpenGL is a **vertex**.

- 2d or 3d
- Specify arbitrary details

The fundamental primitives in OpenGL are the **line segment** and **triangle**.

- Very hard to get wrong
- {vertices} + {ordering} = surface

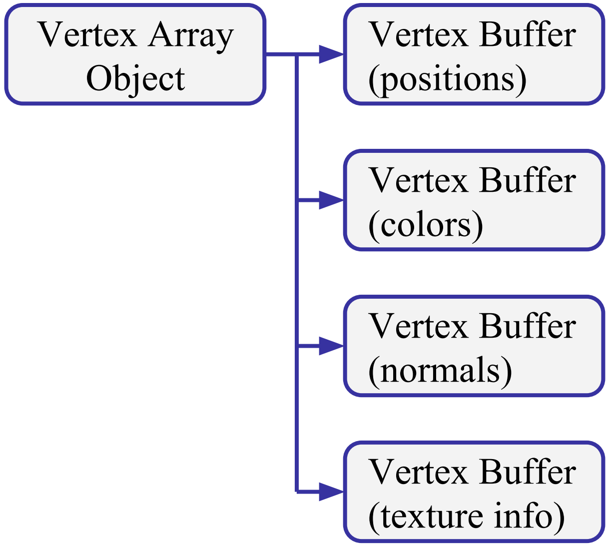


Geometry in OpenGL

V*ertex 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 can bind and attach them to *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.



HelloGL.java [1/4]

```
//////////////////////////////////////////////////////////////////////////////
// Set up GLFW window

GLFWErrorCallback errorCallback = GLFWErrorCallback.createPrint(System.err);
GLFW.glfwSetErrorCallback(errorCallback);
GLFW.glfwInit();
GLFW.glfwWindowHint(GLFW.GLFW_CONTEXT_VERSION_MAJOR, 3);
GLFW.glfwWindowHint(GLFW.GLFW_CONTEXT_VERSION_MINOR, 3);
GLFW.glfwWindowHint(GLFW.GLFW_OPENGL_PROFILE, GLFW.GLFW_OPENGL_CORE_PROFILE);
GLFW.glfwWindowHint(
    GLFW.GLFW_OPENGL_FORWARD_COMPAT, GLFW.GLFW_TRUE);
long window = GLFW.glfwCreateWindow(
    800 /* width */, 600 /* height */, "HelloGL", 0, 0);
GLFW.glfwMakeContextCurrent(window);
GLFW.glfwSwapInterval(1);
GLFW.glfwShowWindow(window);

//////////////////////////////////////////////////////////////////////////////
// Set up OpenGL

GL.createCapabilities();
GL11.glClearColor(0.2f, 0.4f, 0.6f, 0.0f);
GL11.glClearDepth(1.0f);
```

“HelloGL.java” - github.com/AlexBenton/AdvancedGraphics

HelloGL.java [2/4]

```
////////////////////////////////////
// Set up minimal shader programs

// Vertex shader source
String[] vertex_shader = {
  "#version 330\n",
  "in vec3 v;",
  "void main() {",
  "  gl_Position = ",
  "      vec4(v, 1.0);",
  "}"
};

// Fragment shader source
String[] fragment_shader = {
    "#version 330\n",
    "out vec4 frag_colour;",
    "void main() {",
    "  frag_colour = ",
    "      vec4(1.0);",
    "}"
};
```

```
// Compile vertex shader
int vs = GL20.glCreateShader(
    GL20.GL_VERTEX_SHADER);
GL20.glShaderSource(
    vs, vertex_shader);
GL20.glCompileShader(vs);

// Compile fragment shader
int fs = GL20.glCreateShader(
    GL20.GL_FRAGMENT_SHADER);
GL20.glShaderSource(
    fs, fragment_shader);
GL20.glCompileShader(fs);

// Link vertex and fragment
// shaders into active program
int program =
    GL20.glCreateProgram();
GL20.glAttachShader(program, vs);
GL20.glAttachShader(program, fs);
GL20.glLinkProgram(program);
GL20.glUseProgram(program);
```

"HelloGL.java" - github.com/AlexBenton/AdvancedGraphics

HelloGL.java [3/4]

```
//////////////////////////////////////////////////////////////////////////
// Set up data

// Fill a Java FloatBuffer object with memory-friendly floats
float[] coords = new float[] { -0.5f, -0.5f, 0,  0, 0.5f, 0,  0.5f, -0.5f, 0 };
FloatBuffer fbo = BufferUtils.createFloatBuffer(coords.length);
fbo.put(coords);                                // Copy the vertex coords into the
floatbuffer
fbo.flip();                                     // Mark the floatbuffer ready for reads

// Store the FloatBuffer's contents in a Vertex Buffer Object
int vbo = GL15.glGenBuffers();                  // Get an OGL name for the VBO
GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo);   // Activate the VBO
GL15.glBufferData(GL15.GL_ARRAY_BUFFER, fbo, GL15.GL_STATIC_DRAW);  // Send VBO data to GPU

// Bind the VBO in a Vertex Array Object
int vao = GL30.glGenVertexArrays();             // Get an OGL name for the VAO
GL30.glBindVertexArray(vao);                    // Activate the VAO
GL20.glEnableVertexAttribArray(0);              // Enable the VAO's first attribute (0)
GL20.glVertexAttribPointer(0, 3, GL11.GL_FLOAT, false, 0, 0);  // Link VBO to VAO attrib 0
```


"HelloGL.java" - github.com/AlexBenton/AdvancedGraphics

HelloGL.java [4/4]

```
//////////////////////////////////////////////////////////////////////////////
// Loop until window is closed

while (!GLFW.glfwWindowShouldClose(window)) {
  GLFW.glfwPollEvents();

  GL11.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
  GL30.glBindVertexArray(vao);
  GL11.glDrawArrays(GL11.GL_TRIANGLES, 0 /* start */, 3 /* num vertices */);

  GLFW.glfwSwapBuffers(window);
}

//////////////////////////////////////////////////////////////////////////////
// Clean up

GL15.glDeleteBuffers(vbo);
GL30.glDeleteVertexArrays(vao);
GLFW.glfwDestroyWindow(window);
GLFW.glfwTerminate();
GLFW.glfwSetErrorCallback(null).free();
```

"HelloGL.java" - github.com/AlexBenton/AdvancedGraphics

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. This lets us bind vertices with **normals**, **colors**, **texture coordinates**, etc...

Here we bind a vertex buffer object for position data and another for normals:

```
int vao = glGenVertexArrays();
glBindVertexArray(vao);
GL20.glEnableVertexAttribArray(0);
GL20.glEnableVertexAttribArray(1);
GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo_0);
GL20.glVertexAttribPointer( 0, 3, GL11.GL_FLOAT, false, 0, 0);
GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo_1);
GL20.glVertexAttribPointer( 1, 3, GL11.GL_FLOAT, false, 0, 0);
```

Later, to render, we work only with the vertex array:

```
glBindVertexArray(vao);
glDrawArrays(GL_LINE_STRIP, 0, data.length);
```

Caution--all VBOs in a VAO must describe the same number of vertices!

Accessing named GLSL attributes from Java

```
// Vertex shader
// …

#version 330

in vec3 v;
void main() {
  gl_Position =
      vec4(v, 1.0);
}

// …
```

```
// …

glEnableVertexAttribArray(0);
glVertexAttribPointer(0,
    3, GL_FLOAT, false, 0, 0);

// …
```

The HelloGL sample code hardcodes the assumption that the vertex shader input field 'v' is the zeroeth input (position 0).

That's unstable: never rely on a fixed ordering.

Instead, fetch the attrib location:

```
int vLoc =
  GL20.glGetAttribLocation(program, "v");
GL20.glEnableVertexAttribArray( vLoc);
GL20.glVertexAttribPointer( vLoc,
    3, GL_FLOAT, false, 0, 0);
```

This enables greater flexibility and Java code that can adapt to dynamically-changing vertex and fragment shaders.

Improving data throughput

You configure how OpenGL interprets the vertex buffer. Vertices can be interpreted directly, or *indexed* with a separate integer indexing buffer. By re-using vertices and choosing ordering / indexing carefully, you can reduce the number of raw floats sent from the CPU to the GPU dramatically.

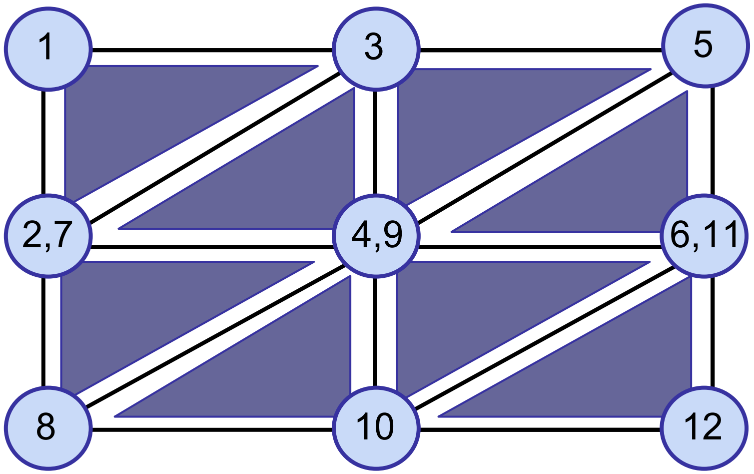
Options include line primitives--

- `GL_LINES`
- `GL_LINE_STRIP`
- `GL_LINE_LOOP`

--triangle primitives--

- `GL_TRIANGLES`
- `GL_TRIANGLE_STRIP`
- `GL_TRIANGLE_FAN`

--and more. OpenGL also offers *backface culling* and other optimizations.



Triangle-strip vertex indexing (counter-clockwise ordering)

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.

This means that **you must handle your own memory cleanup**.

```
// create and bind buffer object
int name = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, name);

// work with your object
// …

// delete buffer object, free memory
glDeleteBuffers(name);
```

Emulating classic OpenGL1.1 direct-mode rendering in modern GL

The original OpenGL API allowed you to use *direct mode* to send data for immediate output:

```
glBegin(GL_QUADS);
  glColor3f(0, 1, 0);
  glNormal3f(0, 0, 1);
  glVertex3f(1, -1, 0);
  glVertex3f(1, 1, 0);
  glVertex3f(-1, 1, 0);
  glVertex3f(-1, -1, 0);
glEnd();
```

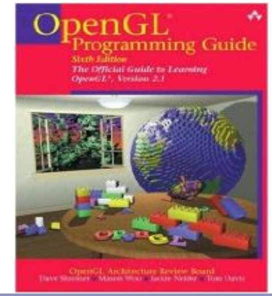
Direct mode was very inefficient: the GPU was throttled by the CPU.

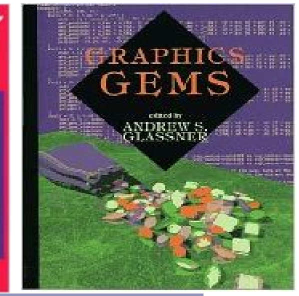
You can emulate the GL1.1 API:

```
class GLVertexData {
  void begin(mode) { … }
  void color(color) { … }
  void normal(normal) { … }
  void vertex(vertex) { … }
  …
  void compile() { … }
}
```

The method compile() can encapsulate all the vertex buffer logic, making each instance a self-contained buffer object.

Check out a working example in the class `framework.GLVertexData` on the course github repo.





Recommended reading

Course source code on Github -- many demos
(https://github.com/AlexBenton/AdvancedGraphics)

The OpenGL Programming Guide (2013), by Shreiner, Sellers, Kessenich and Licea-Kane

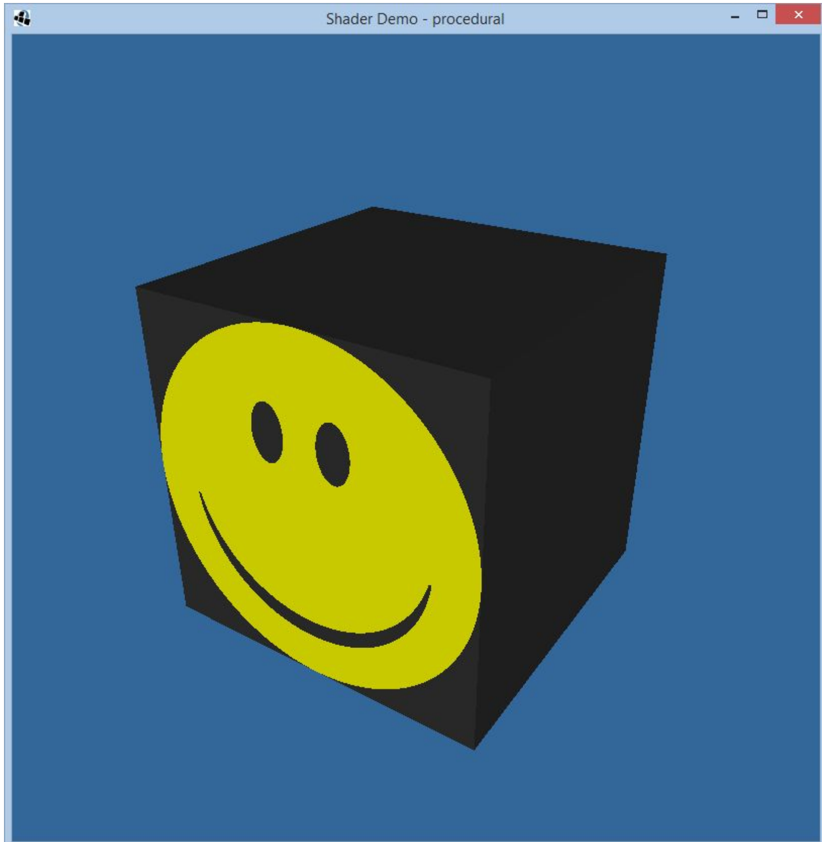
  - Some also favor *The OpenGL Superbible* for code samples and demos
  - There's also an OpenGL-ES reference, same series

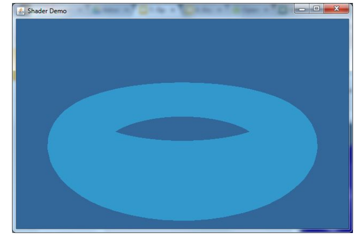
OpenGL Insights (2012), by Cozzi and Riccio

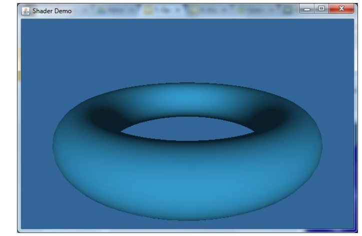
OpenGL Shading Language (2009), by Rost, Licea-Kane, Ginsburg et al

The *Graphics Gems* series from Glassner

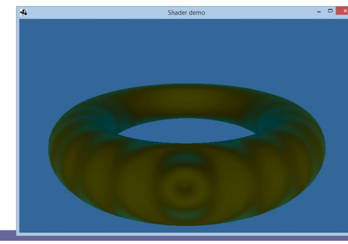
ShaderToy.com, a web site by Inigo Quilez (Pixar) dedicated to amazing shader tricks and raycast scenes

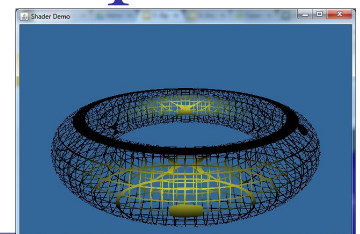




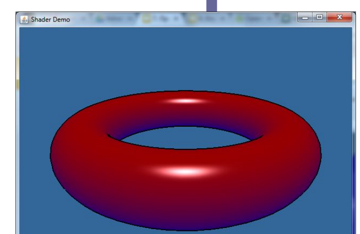


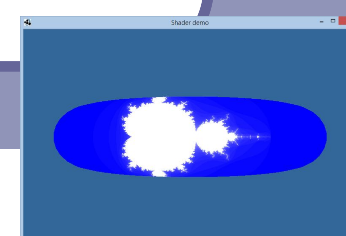
Advanced Graphics





OpenGL and Shaders II





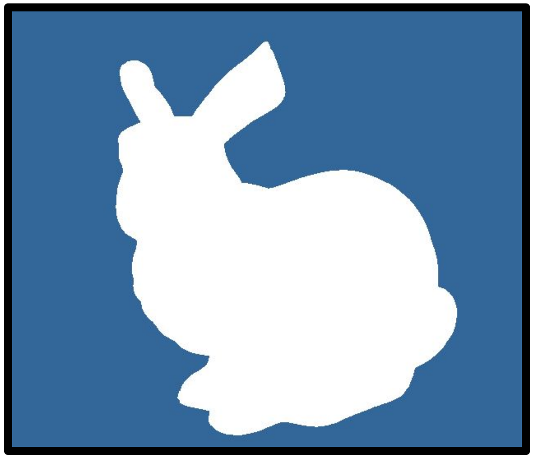
Alex Benton, University of Cambridge – alex@bentonian.com

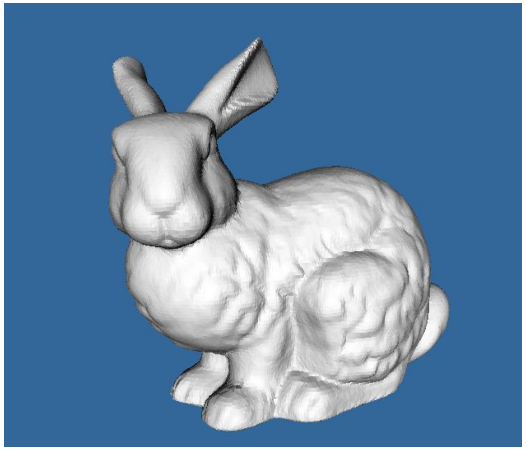
Supported in part by Google UK, Ltd

2.Perspective and Camera Control

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





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:

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

α: Field of view, typically 50°

ar: Aspect ratio of width over height

NearZ: Near clip plane

FarZ: Far clip plane

Writing uniform data from Java

Once you have your perspective matrix, the next step is to copy it out to the GPU as a **`Mat4`**, GLSL's 4x4 matrix type.

1. Convert your floats to a FloatBuffer:

```
float data[][] = /* your 4x4 matrix here */
FloatBuffer buffer = BufferUtils.createFloatBuffer(16);
for (int col = 0; col < 4; col++) {
  for (int row = 0; row < 4; row++) {
    buffer.put((float) (data[row][col]));
  }
}
buffer.flip();
```

2. Write the FloatBuffer to the named uniform:

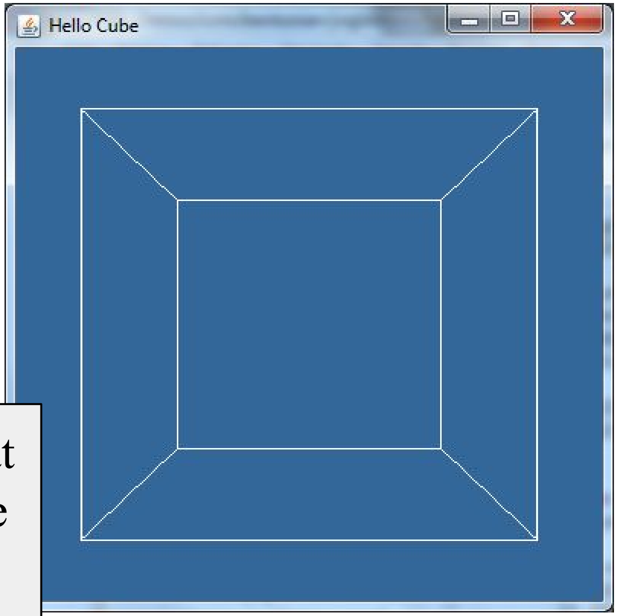
```
int uniformLoc = GL20.glGetUniformLocation(
    program, "name");
if (uniformLoc != -1) {
  GL20.glUniformMatrix4fv(uniformLoc, false, buffer);
}
```

Reading uniform data in GLSL

The **FloatBuffer** output is received in the shader as a *uniform* input of type **Mat4**.

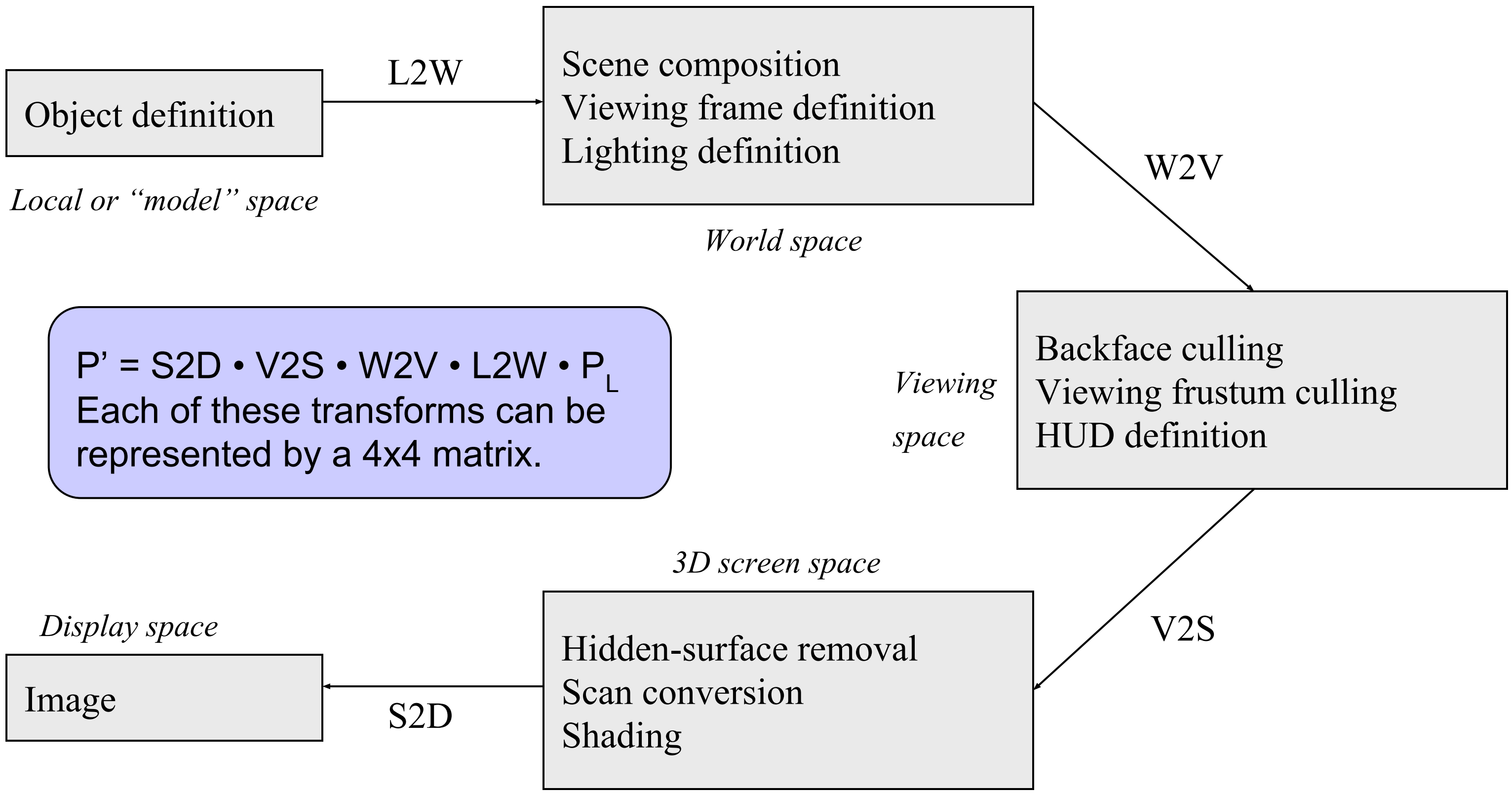
This shader takes a matrix and applies it to each vertex:

```
#version 330

uniform mat4 modelToScreen;

in vec4 vPosition;

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



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

Object position and camera position: a 'pipeline' model of matrix transforms



$P' = S2D \cdot V2S \cdot W2V \cdot L2W \cdot P_L$
Each of these transforms can be represented by a 4x4 matrix.

The pipeline model in OpenGL & GLSL

A flexible 3D graphics framework will track each transform:

- The object's current transform
- The camera's transform
- The viewing perspective transform

These matrices are all "constants" for the duration of a single frame of rendering. Each can be written to a 16-float buffer and sent to the GPU with `glUniformMatrix4fv`.

Remember to fetch uniform names with `glGetUniformLocation`, never assume ordering.

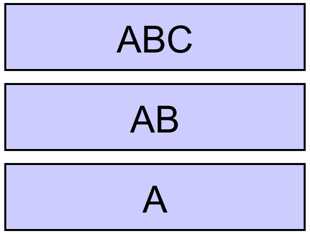
```
#version 330

uniform mat4 modelToWorld;
uniform mat4 worldToCamera;
uniform mat4 cameraToScreen;

in vec3 v;

void main() {
  gl_Position = cameraToScreen
      * worldToCamera
      * modelToWorld
      * vec4(v, 1.0);
}
```

The pipeline model in software: The *matrix stack* design pattern

A common design pattern in 3D graphics, especially when objects can contain other objects, is to use *matrix stacks* to store stacks of matrices. The topmost matrix is the product of all matrices below.

- This allows you to build a local frame of reference—local space—and apply transforms within that space.
- Remember: matrix multiplication is associative but not commutative.
  - $ABC = A(BC) = (AB)C \neq ACB \neq BCA$

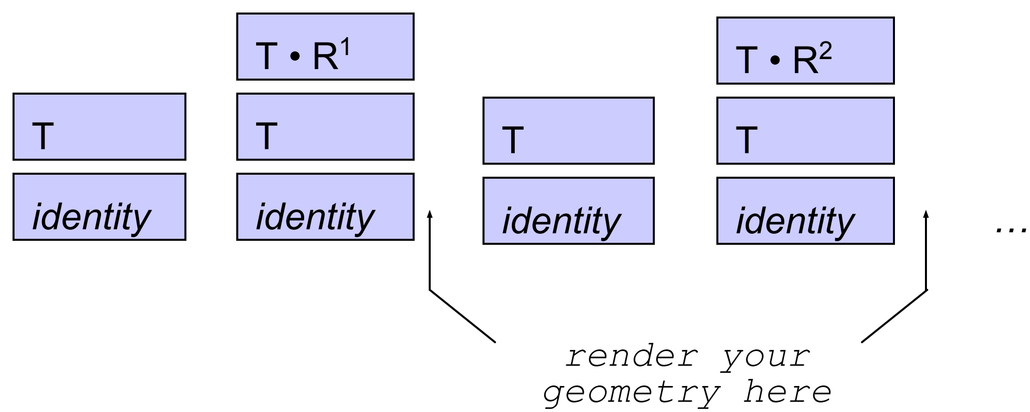
Pre-multiplying matrices that will be used more than once is faster than multiplying many matrices every time you render a primitive.



Matrix stacks

Matrix stacks are designed for nested relative transforms.

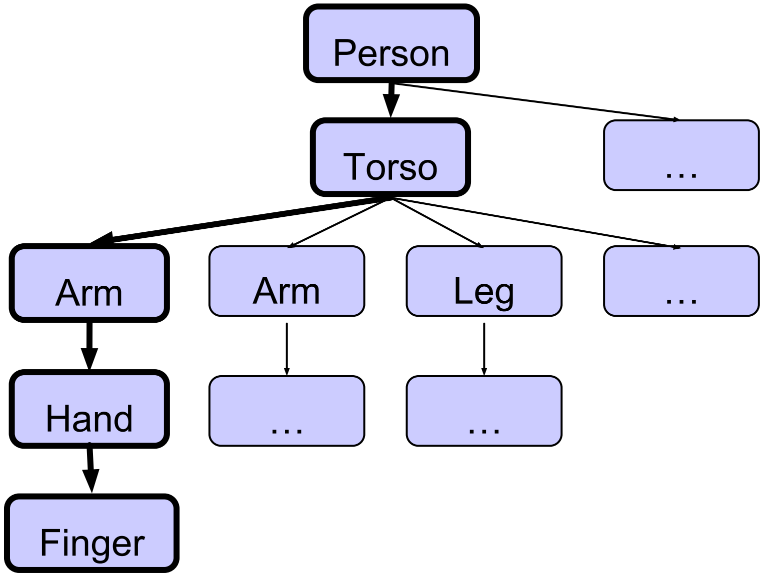
```
pushMatrix();
  translate(0,0,-5);
  pushMatrix();
    rotate(45,0,1,0);
    render();
  popMatrix();
  pushMatrix();
    rotate(-45,0,1,0);
    render();
  popMatrix();
popMatrix();
```



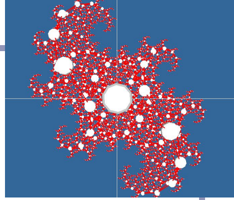
Scene graphs

A *scene graph* is a tree of scene elements where a child's transform is relative to its parent.

The final transform of the child is the ordered product of all of its ancestors in the tree.

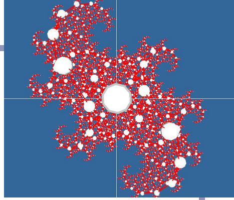


$$M_{fingerToWorld} = (M_{person} \bullet M_{torso} \bullet M_{arm} \bullet M_{hand} \bullet M_{finger})$$

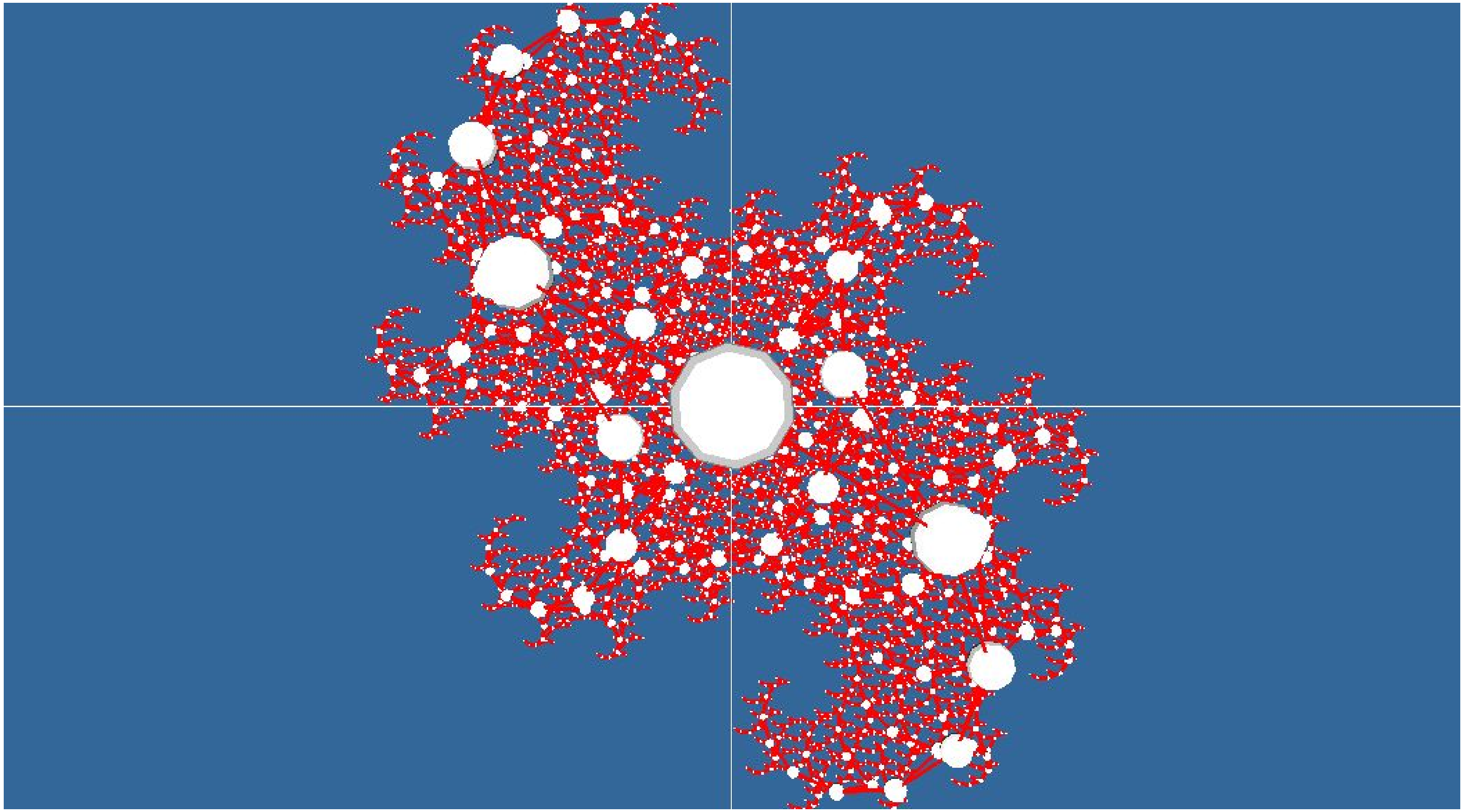


Hierarchical modeling in action

```
void renderLevel(GL gl, int level, float t) {
  pushMatrix();
  rotate(t, 0, 1, 0);
  renderSphere(gl);
  if (level > 0) {
    scale(0.75f, 0.75f, 0.75f);
    pushMatrix();
      translate(1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    popMatrix();
    pushMatrix();
      translate(-1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    popMatrix();
  }
  popMatrix();
}
```



Hierarchical modeling in action

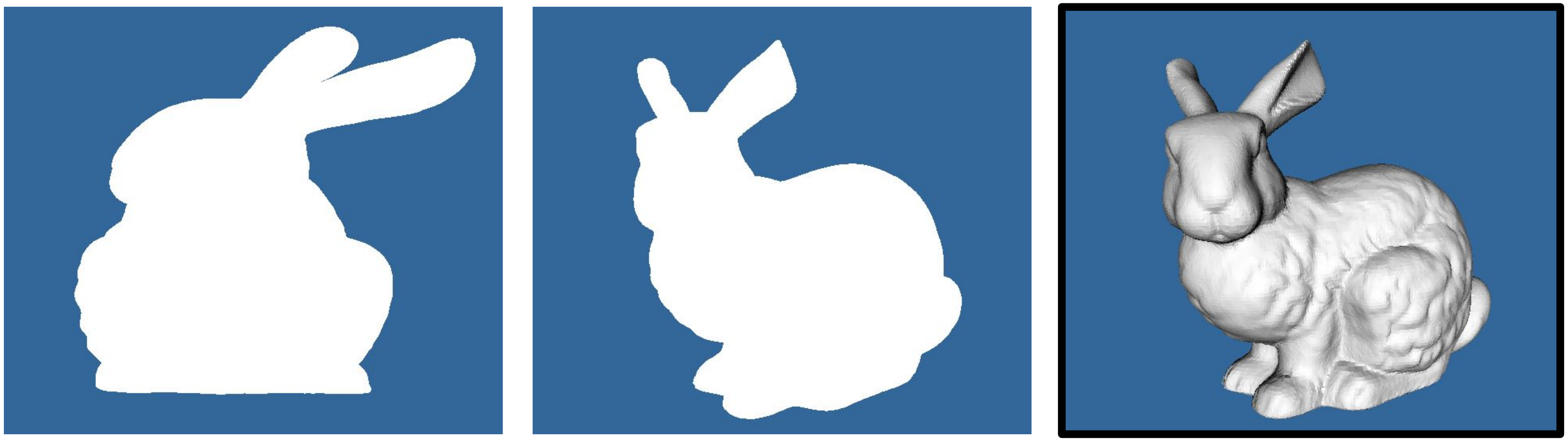


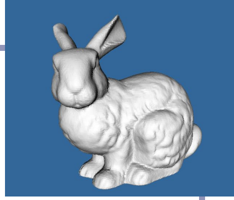
“HierarchyDemo.java” - github.com/AlexBenton/AdvancedGraphics

3. Lighting and Shading

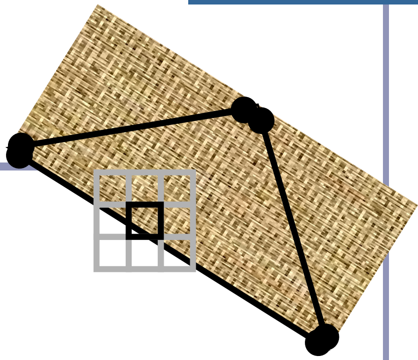
It’s up to you to implement perspective and lighting.

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





Lighting and Shading (a quick refresher)



Recall the classic **lighting equation**:

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

where…

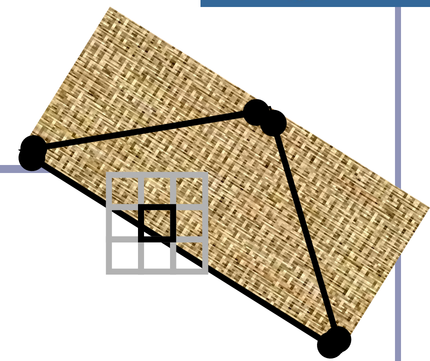
- k_A is the *ambient lighting coefficient* of the object or scene
- $k_D(N \cdot L)$ is the *diffuse component* of surface illumination ('matte')
- $k_S(E \cdot R)^n$ is the *specular component* of surface illumination ('shiny')
  where $R = L - 2(L \cdot N)N$

We compute color by vertex or by polygon fragment:

- Color at the vertex: **Gouraud shading**
- Color at the polygon fragment: **Phong shading**

Vertex shader outputs are interpolated across fragments, so code is clean whether we're interpolating colors or normals.

Lighting and Shading: required data



Shading means we need extra data about vertices.

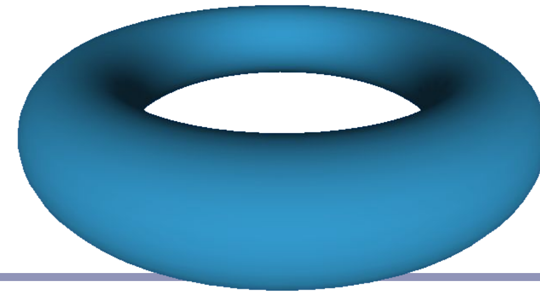
For each vertex our Java code will need to provide:

- Vertex position
- Vertex normal
- [Optional] Vertex color, k_A / k_D / k_S, reflectance, transparency…

We also need global state:

- Camera perspective transform
- Camera position and orientation, represented as a transform
- Object position and orientation, to modify the vertex positions above
- A list of light positions, ideally in world coordinates

Shader sample – Gouraud shading

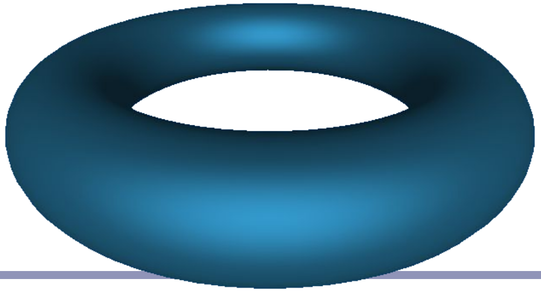


```
#version 330

uniform mat4 modelToScreen;
uniform mat4 modelToWorld;
uniform mat3 normalToWorld;
uniform vec3 lightPosition;

in vec4 v;
in vec3 n;

out vec4 color;

const vec3 purple = vec3(0.2, 0.6, 0.8);

void main() {
  vec3 p = (modelToWorld * v).xyz;
  vec3 n = normalize(normalToWorld * n);
  vec3 l = normalize(lightPosition - p);
  float ambient = 0.2;
  float diffuse = 0.8 * clamp(0, dot(n, l), 1);

  color = vec4(purple
      * (ambient + diffuse), 1.0);
  gl_Position = modelToScreen * v;
}
```

```
#version 330

in vec4 color;

out vec4 fragmentColor;

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

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

Shader sample – Phong shading



```
#version 330

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

in vec4 v;
in vec3 n;

out vec3 position;
out vec3 normal;

void main() {
  normal = normalize(
      normalToWorld * n);
  position =
      (modelToWorld * v).xyz;
  gl_Position =
      modelToScreen * v;
}
```

GLSL includes handy helper methods for illumination such as reflect()--perfect for specular highlights.

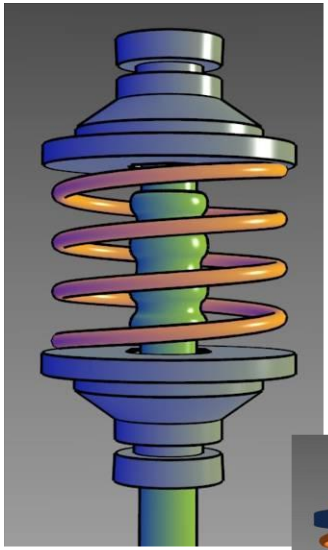
```
#version 330

uniform vec3 eyePosition;
uniform vec3 lightPosition;

in vec3 position;
in vec3 normal;

out vec4 fragmentColor;

const vec3 purple = vec3(0.2, 0.6, 0.8);

void main() {
  vec3 n = normalize(normal);
  vec3 l = normalize(lightPosition - position);
  vec3 e = normalize(position - eyePosition);
  vec3 r = reflect(l, n);

  float ambient = 0.2;
  float diffuse = 0.4 * clamp(0, dot(n, l), 1);
  float specular = 0.4 *
      pow(clamp(0, dot(e, r), 1), 2);

  fragmentColor = vec4(purple *
      (ambient + diffuse + specular), 1.0);
}
```

$a = k_A$
$d = k_D(N \bullet L)$
$s = k_S(\mathrm{E} \bullet \mathrm{R})^{\mathrm{n}}$

Shader sample – Gooch shading

Gooch shading is an example of *non-realistic rendering*. It was designed by Amy and Bruce Gooch to replace photorealistic lighting with a lighting model that highlights structural and contextual data.

- They use the term of the conventional lighting equation to choose a map between 'cool' and 'warm' colors.
  - This is in contrast to conventional illumination where lighting simply scales the underlying surface color.
- This, combined with edge-highlighting through a second renderer pass, creates models which look more like engineering schematic diagrams.



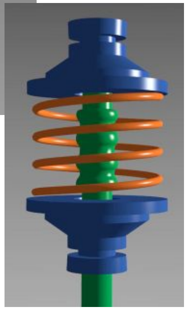
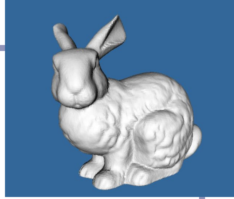


Image source: "A Non-Photorealistic Lighting Model For Automatic Technical Illustration", Gooch, Gooch, Shirley and Cohen (1998). Compare the Gooch shader, above, to the Phong shader (right).



Shader sample – Gooch shading

```
#version 330

// Original author: Randi Rost
// Copyright (c) 2002-2005 3Dlabs Inc. Ltd.

uniform mat4 modelToCamera;
uniform mat4 modelToScreen;
uniform mat3 normalToCamera;

vec3 LightPosition = vec3(0, 10, 4);

in vec4 vPosition;
in vec3 vNormal;

out float NdotL;
out vec3  ReflectVec;
out vec3  ViewVec;

void main()
{
  vec3 ecPos      = vec3(modelToCamera * vPosition);
  vec3 tnorm      = normalize(normalToCamera * vNormal);
  vec3 lightVec   = normalize(LightPosition - ecPos);
  ReflectVec      = normalize(reflect(-lightVec, tnorm));
  ViewVec         = normalize(-ecPos);
  NdotL           = (dot(lightVec, tnorm) + 1.0) * 0.5;
  gl_Position     = modelToScreen * vPosition;
}
```

```
#version 330

// Original author: Randi Rost
// Copyright (c) 2002-2005 3Dlabs Inc. Ltd.

uniform vec3 vColor;

float DiffuseCool = 0.3;
float DiffuseWarm = 0.3;
vec3 Cool = vec3(0, 0, 0.6);
vec3 Warm = vec3(0.6, 0, 0);

in float NdotL;
in vec3  ReflectVec;
in vec3  ViewVec;

out vec4 result;

void main()
{
  vec3 kcool  = min(Cool + DiffuseCool * vColor, 1.0);
  vec3 kwarm  = min(Warm + DiffuseWarm * vColor, 1.0);
  vec3 kfinal = mix(kcool, kwarm, NdotL);

  vec3 nRefl  = normalize(ReflectVec);
  vec3 nview  = normalize(ViewVec);
  float spec  = pow(max(dot(nRefl, nview), 0.0), 32.0);

  if (gl_FrontFacing) {
    result = vec4(min(kfinal + spec, 1.0), 1.0);
  } else {
    result = vec4(0, 0, 0, 1);
  }
}
```

Shader sample – Gooch shading

In the vertex shader source, notice the use of the built-in ability to distinguish front faces from back faces:

```
if (gl_FrontFacing) {...
```

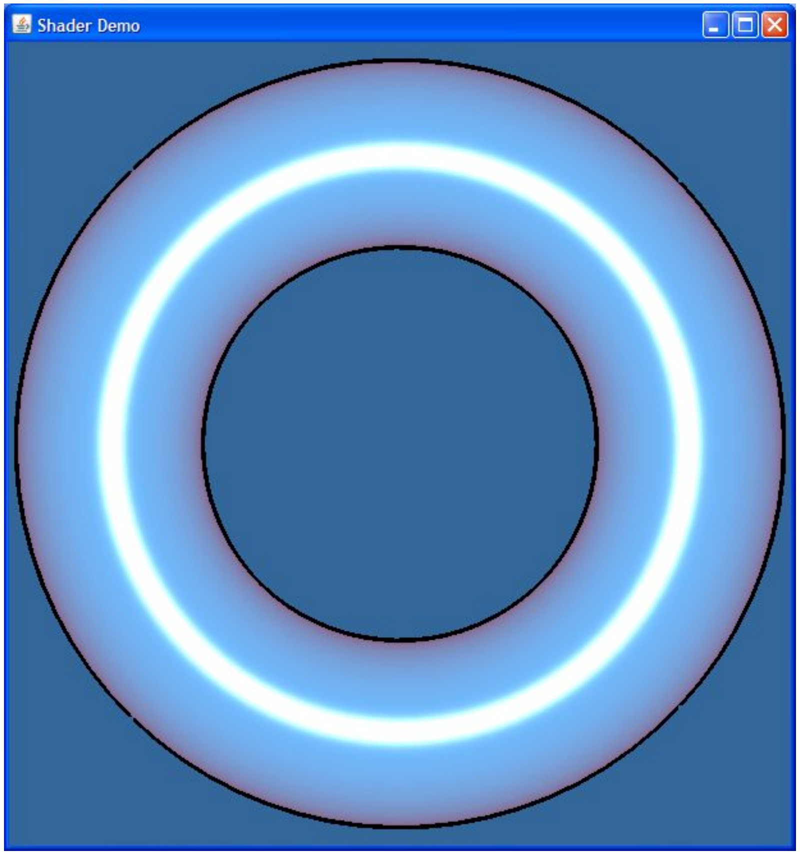
This supports distinguishing front faces (which should be shaded smoothly) from the edges of back faces (which will be drawn in heavy black.)

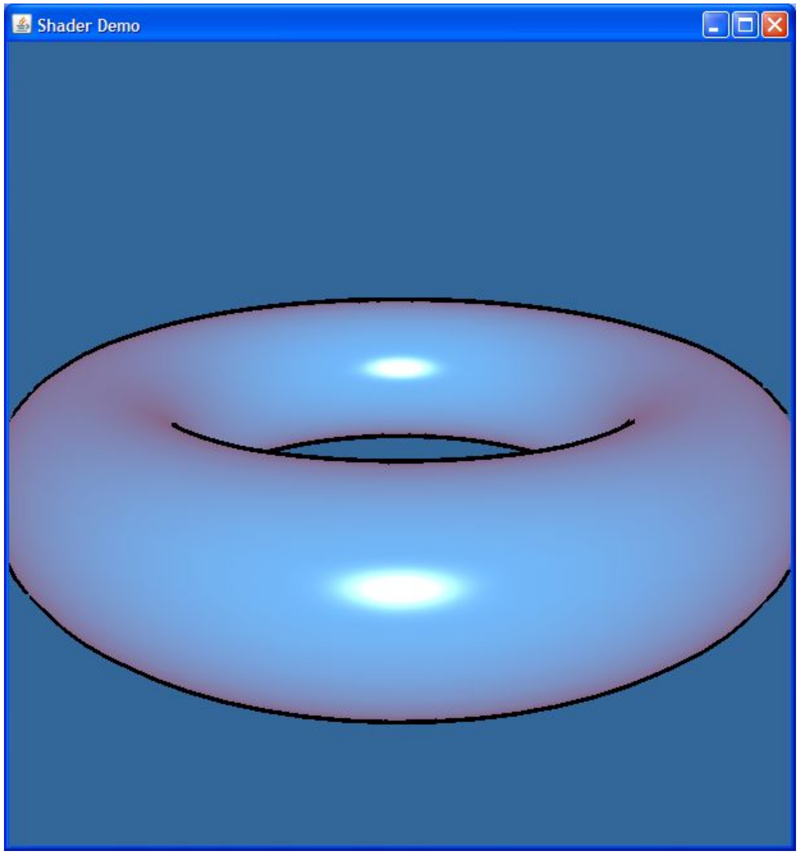
In the fragment shader source, this is used to choose the weighted color by clipping with the `a` component:

```
vec3 kfinal = mix(kcool, kwarm, NdotL);
```

Here `mix()` is a GLSL method which returns the linear interpolation between `kcool` and `kwarm`. The weighting factor is `NdotL`, the lighting value.

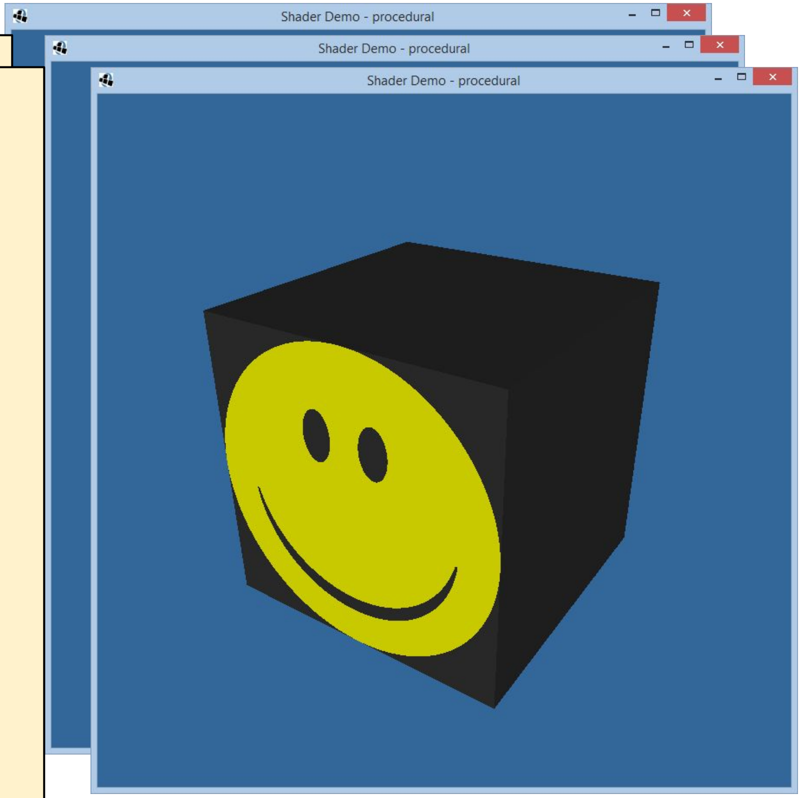
Shader sample – Gooch shading





Procedural texturing in the fragment shader

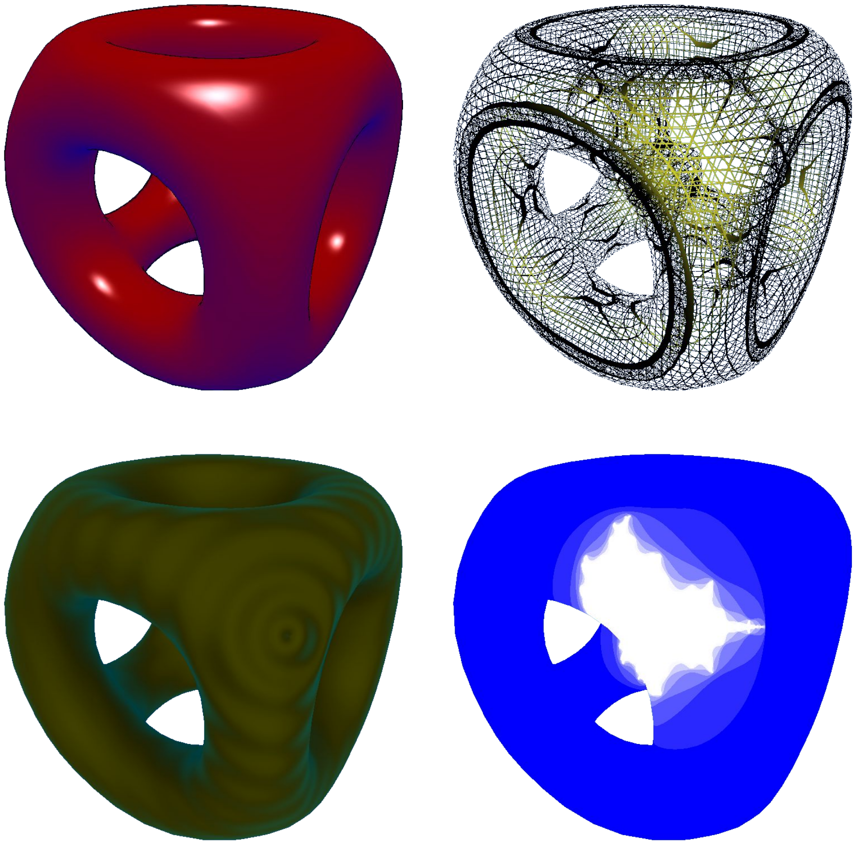
```
// ...
const vec3 CENTER = vec3(0, 0, 1);
const vec3 LEFT_EYE = vec3(-0.2, 0.25, 0);
const vec3 RIGHT_EYE = vec3(0.2, 0.25, 0);
// ...

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.)

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!



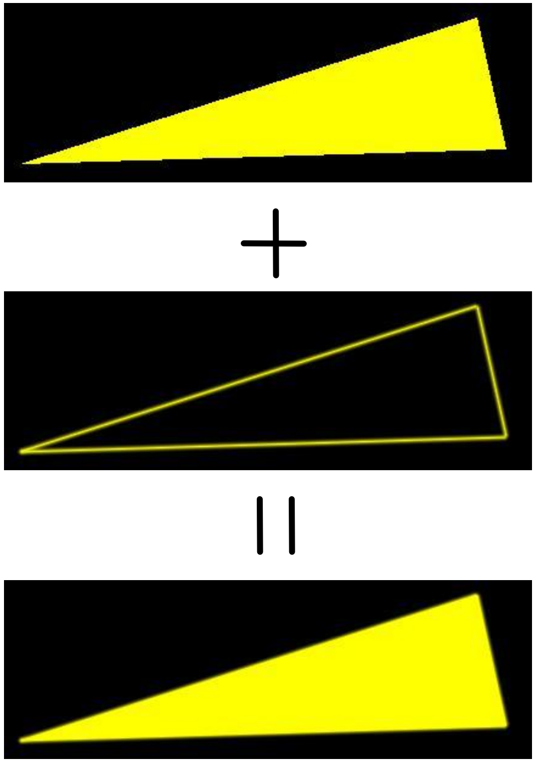
Antialiasing on the GPU

Hardware antialiasing can dramatically improve image quality.

- The naïve approach is to supersample the image
- This is easier in shaders than it is in standard software
- But it really just postpones the problem.

Several GPU-based antialiasing solutions have been found.

- Eric Chan published an elegant polygon-based antialiasing approach in 2004 which uses the GPU to prefilter the edges of a model and then blends the filtered edges into the original polygonal surface. (See figures at right.)



Antialiasing on the GPU

One clever form of antialiasing is *adaptive analytic prefiltering*.

- The precision with which an edge is rendered to the screen is dynamically refined based on the rate at which the function defining the edge is changing with respect to the surrounding pixels on the screen.

This is supported in the shader language by the methods `dFdx(F)` and `dFdy(F)`.

- These methods return the derivative with respect to X and Y of some variable F.
- These are commonly used in choosing the filter width for antialiasing procedural textures.

(A)

(B)

(C)

(A) Jagged lines visible in the box function of the procedural stripe texture
(B) Fixed-width averaging blends adjacent samples in texture space; aliasing still occurs at the top, where adjacency in texture space does not align with adjacency in pixel space.
(C) Adaptive analytic prefiltering smoothly samples both areas.
Image source: Figure 17.4, p. 440, *OpenGL Shading Language, Second Edition*, Randi Rost, Addison Wesley, 2006. Digital image scanned by Google Books.
Original image by Bert Freudenberg, University of Magdeburg, 2002.

Particle systems on the GPU

Shaders extend the use of *texture memory* dramatically. Shaders can write to texture memory, and textures are no longer limited to being two-dimensional planes of RGB(A).

- A particle systems can be represented by storing a position and velocity for every particle.
- A fragment shader can render a particle system entirely in hardware by using texture memory to store and evolve particle data.

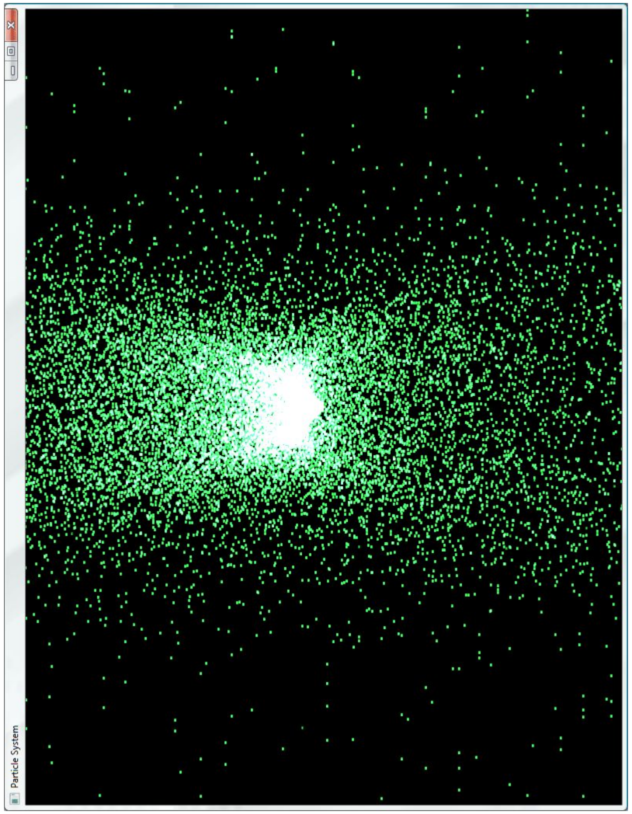


Image by Michael Short

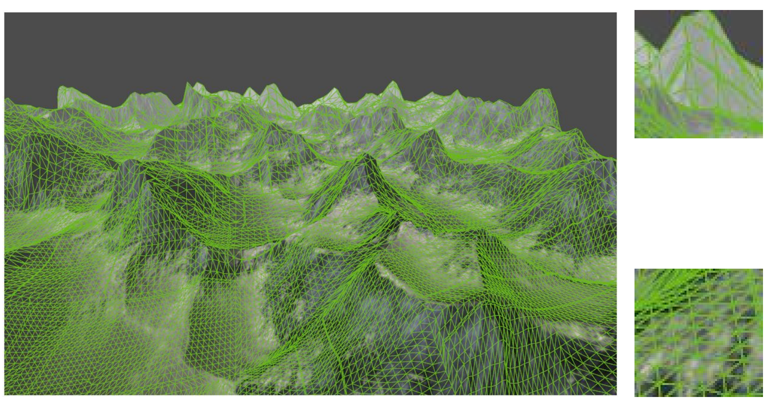
Tessellation shaders

Tesselation is a new shader type introduced in OpenGL 4.x. Tesselation shaders generate new vertices within *patches*, transforming a small number of vertices describing triangles or quads into a large number of vertices which can be positioned individually.

One use of tessellation is in rendering geometry such as game models or terrain with view-dependent *Levels of Detail* ("LOD").

Another is to do with geometry what ray-tracing did with bump-mapping: high-precision realtime geometric deformation.

Note how triangles are small and detailed close to the camera, but become very large and coarse in the distance.



Florian Boesch's LOD terrain demo
http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/

jabtunes.com*'s WebGL tessellation demo*

Tessellation shaders

How it works:

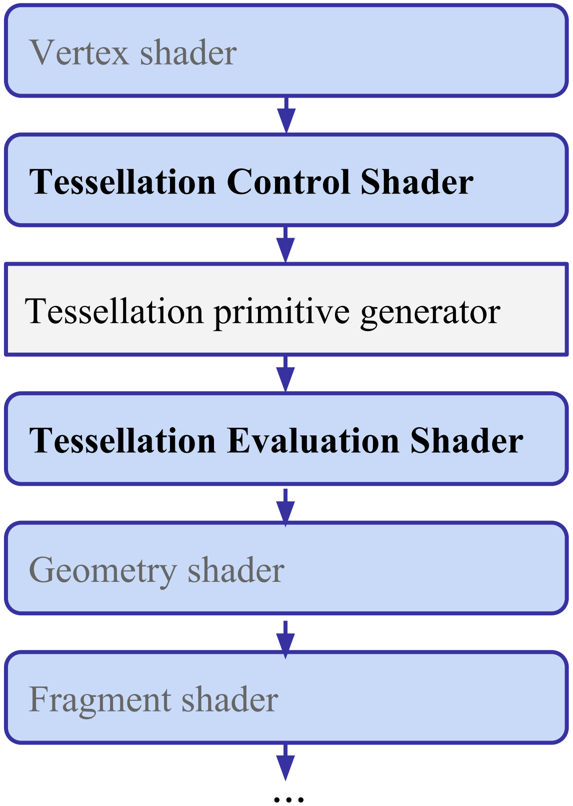
- You tell OpenGL how many vertices a single *patch* will have:

```
glPatchParameteri(GL_PATCH_VERTICES, 4);
```

- You tell OpenGL to render your patches:

```
glDrawArrays(GL_PATCHES, first, numVerts);
```

- The *Tessellation Control Shader* specifies output parameters defining how a patch is split up: `gl_TessLevelOuter[]` and `gl_TessLevelInner[]`. These control the number of vertices per primitive edge and the number of nested inner levels, respectively.

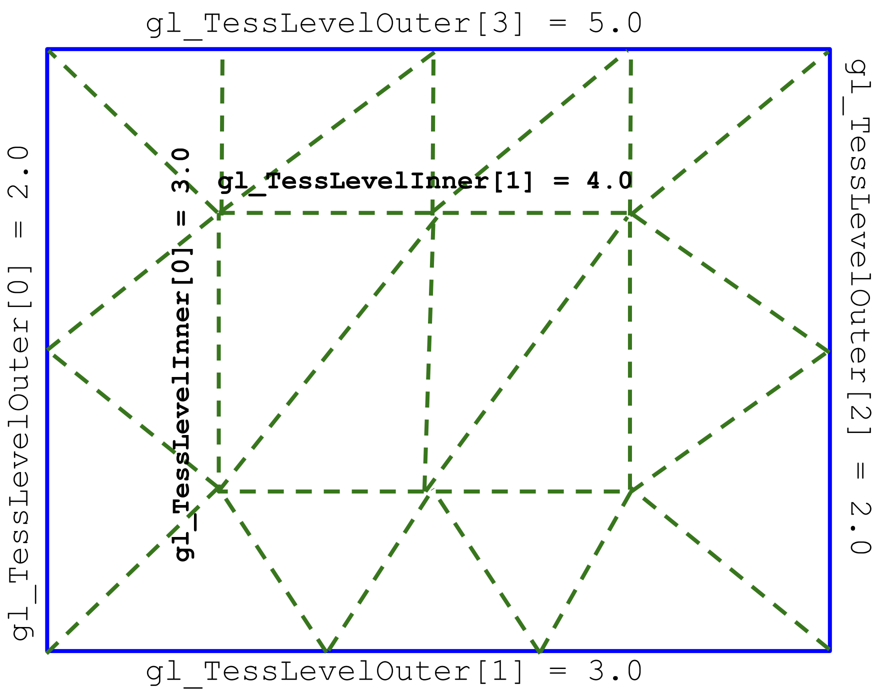


Tessellation shaders

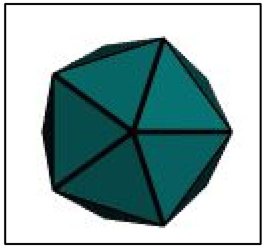
- The *tessellation primitive generator* generates new vertices along the outer edge and inside the patch, as specified by `gl_TessLevelOuter[]` and `gl_TessLevelInner[]`.

Each field is an array. Within the array, each value sets the number of intervals to generate during subprimitive generation.

Triangles are indexed similarly, but only use the first three `Outer` and the first `Inner` array field.



Tessellation shaders



- The generated vertices are then passed to the *Tesselation Evaluation Shader*, which can update vertex position, color, normal, and all other per-vertex data.
- Ultimately the complete set of new vertices is passed to the geometry and fragment shaders.

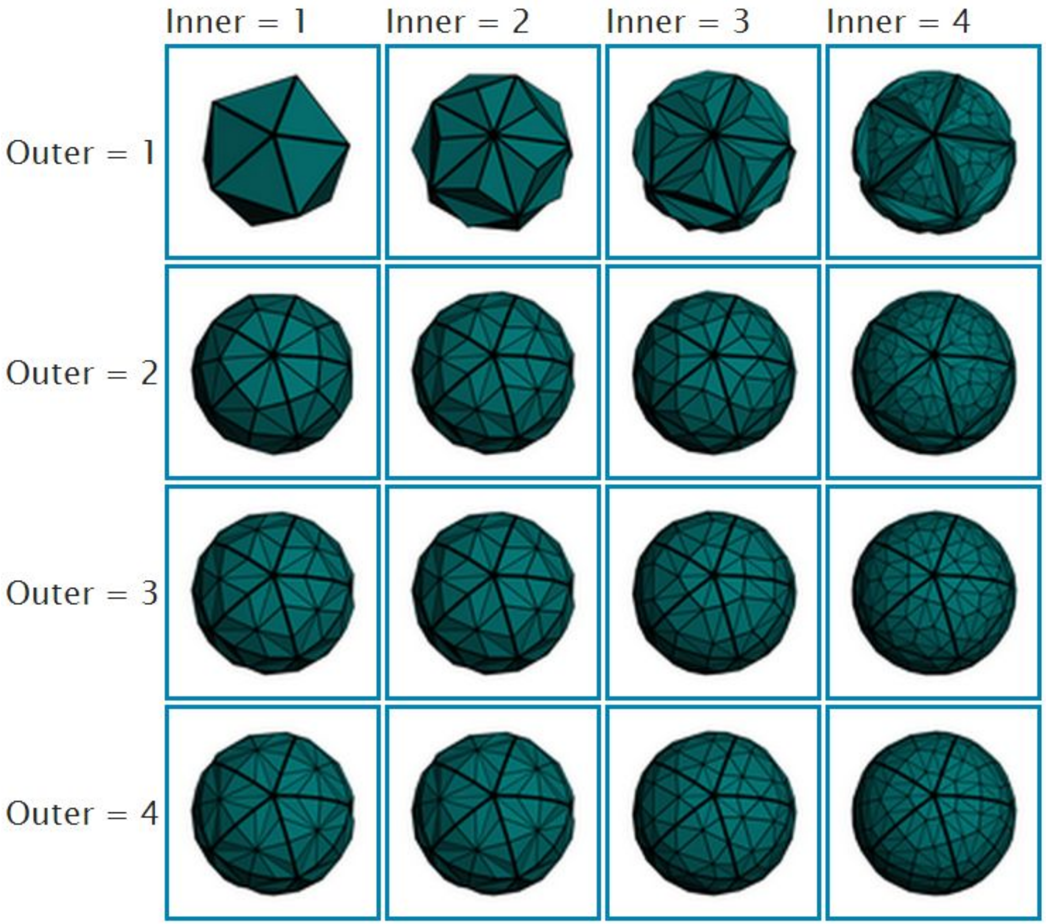
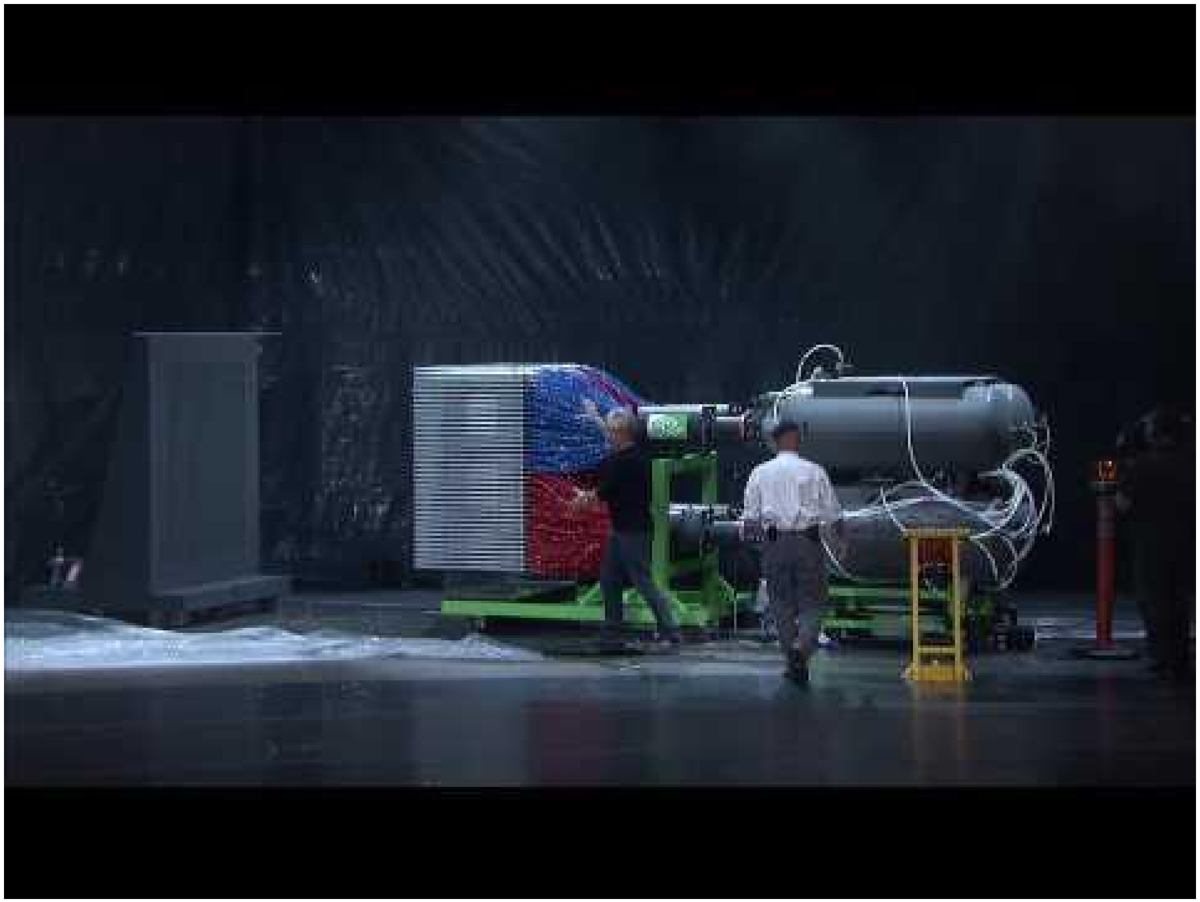


Image credit: Philip Rideout
http://prideout.net/blog/?p=48

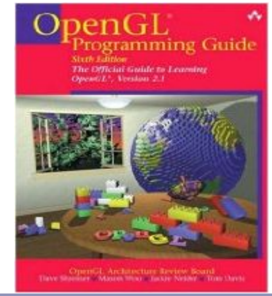
CPU vs GPU – an object demonstration

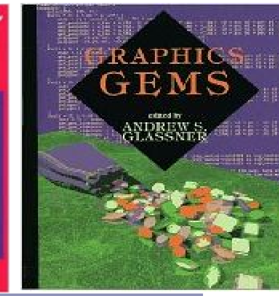


"NVIDIA: Mythbusters - CPU vs GPU"
https://www.youtube.com/watch?v=-P28LKWTzrI

Redux: http://www.youtube.com/watch?v=fKK933KK6Gg

Recommended reading





Course source code on Github -- many demos
(https://github.com/AlexBenton/AdvancedGraphics)

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

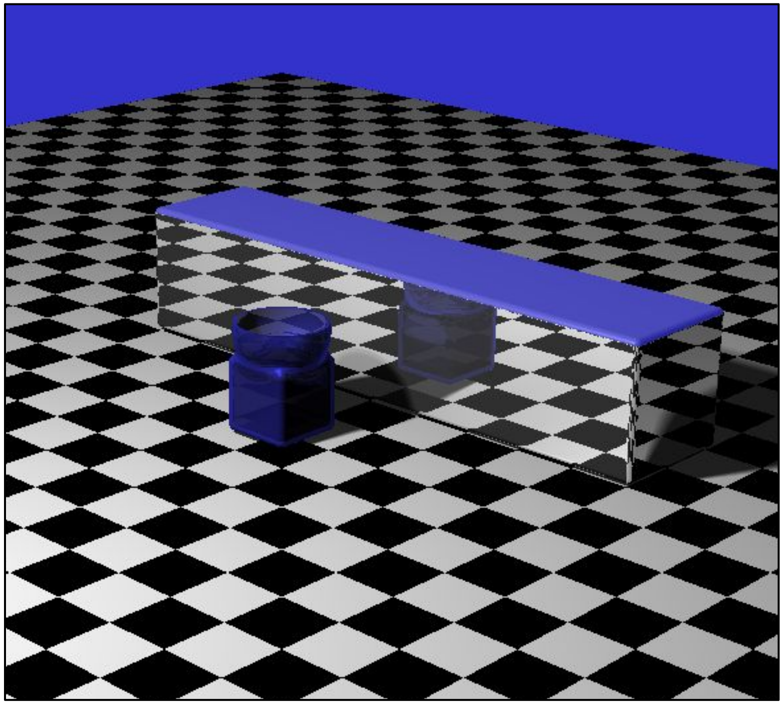
OpenGL Insights (2012), by Cozzi and Riccio

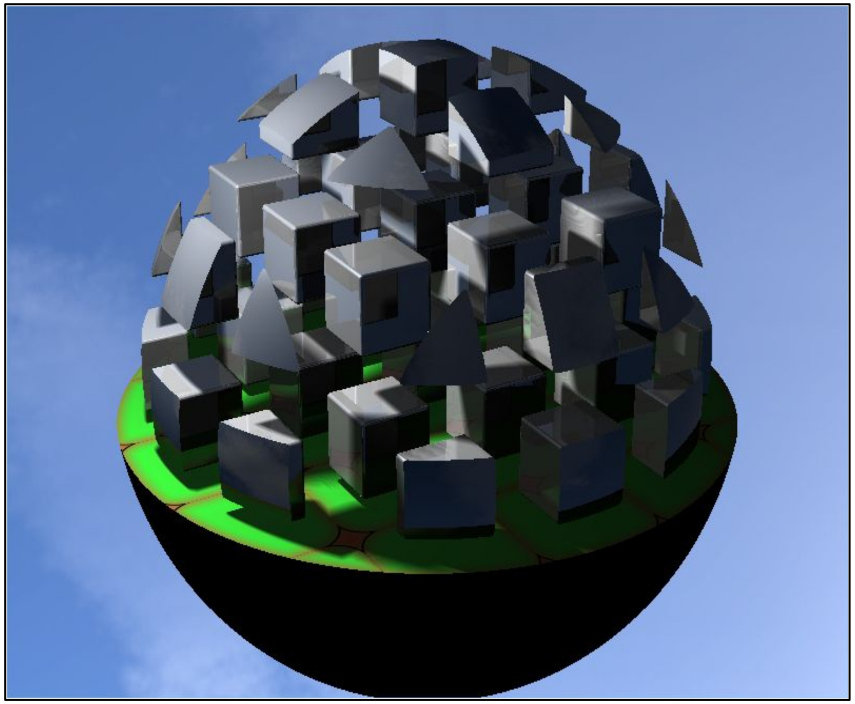
OpenGL Shading Language (2009), by Rost, Licea-Kane, Ginsburg et al

The *Graphics Gems* series from Glassner

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

Advanced Graphics





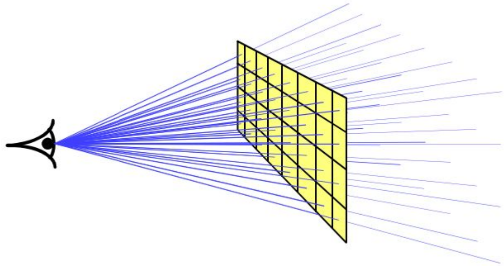
GPU Ray Marching

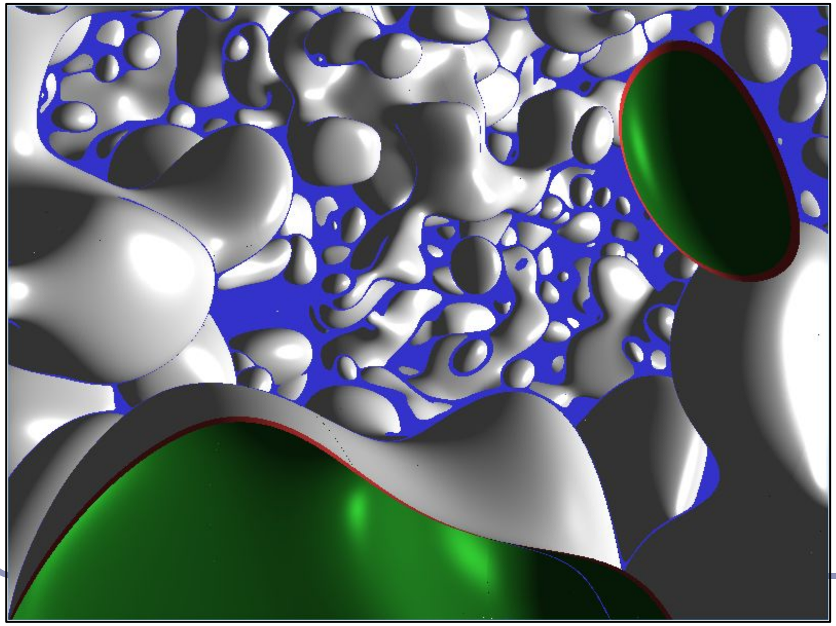
Alex Benton, University of Cambridge – alex@bentonian.com

Supported in part by Google UK, Ltd

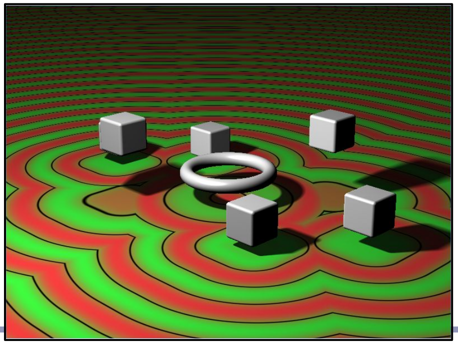
GPU Ray-tracing

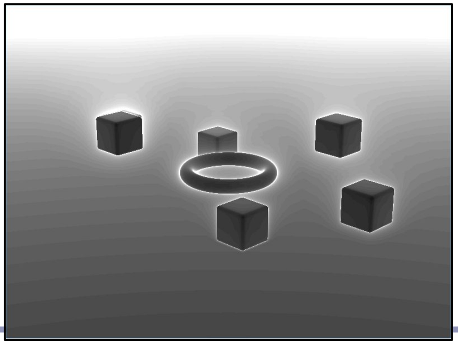
Ray tracing 101: “Choose the color of the pixel by firing a ray through and seeing what it hits.”





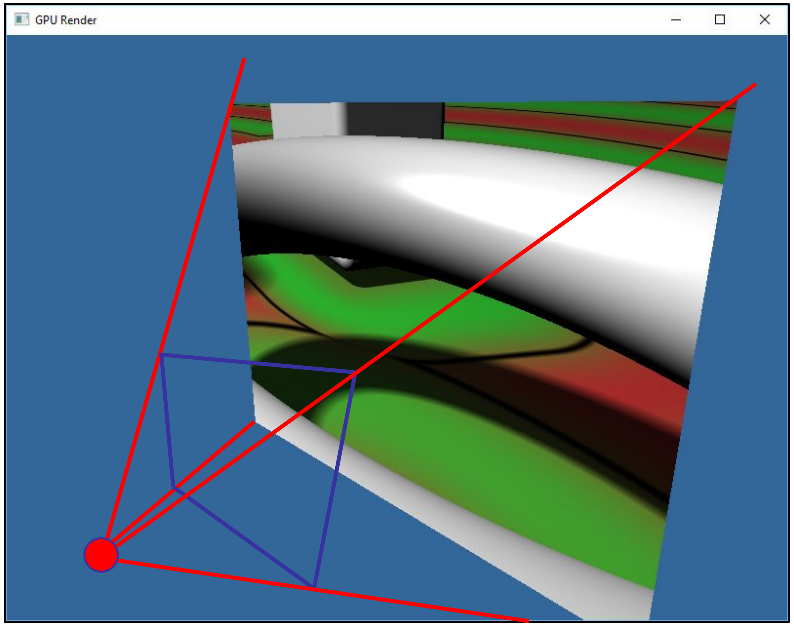
Ray tracing 102: “Let the pixel make up its own mind.”





GPU Ray-tracing

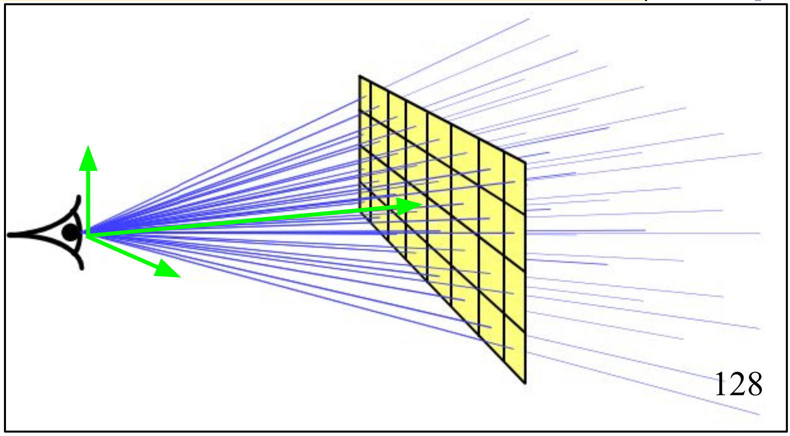
1. Use a minimal fragment shader (no transforms)
2. Set up OpenGL with minimal geometry, a single quad
3. Bind a `vec2` to each vertex specifying 'texture' coordinates
4. Implement raytracing in GLSL per pixel:
    a. For each pixel, compute the ray from the eye through the pixel, using the interpolated texture coordinate to identify the pixel
    b. Run the ray tracing algorithm for every ray



GPU Ray-tracing

```
// Window dimensions
uniform vec2 iResolution;

// Camera position
uniform vec3 iRayOrigin;

// Camera facing direction
uniform vec3 iRayDir;

// Camera up direction
uniform vec3 iRayUp;

// Distance to viewing plane
uniform float iPlaneDist;

// 'Texture' coordinate of each
// vertex, interpolated across
// fragments (0,0) → (1,1)
in vec2 texCoord;
```

```
vec3 getRayDir(
    vec3 camDir,
    vec3 camUp,
    vec2 texCoord) {
  vec3 camSide = normalize(
      cross(camDir, camUp));
  vec2 p = 2.0 * texCoord - 1.0;
  p.x *= iResolution.x
      / iResolution.y;
  return normalize(
      p.x * camSide
      + p.y * camUp
      + iPlaneDist * camDir);
}
```



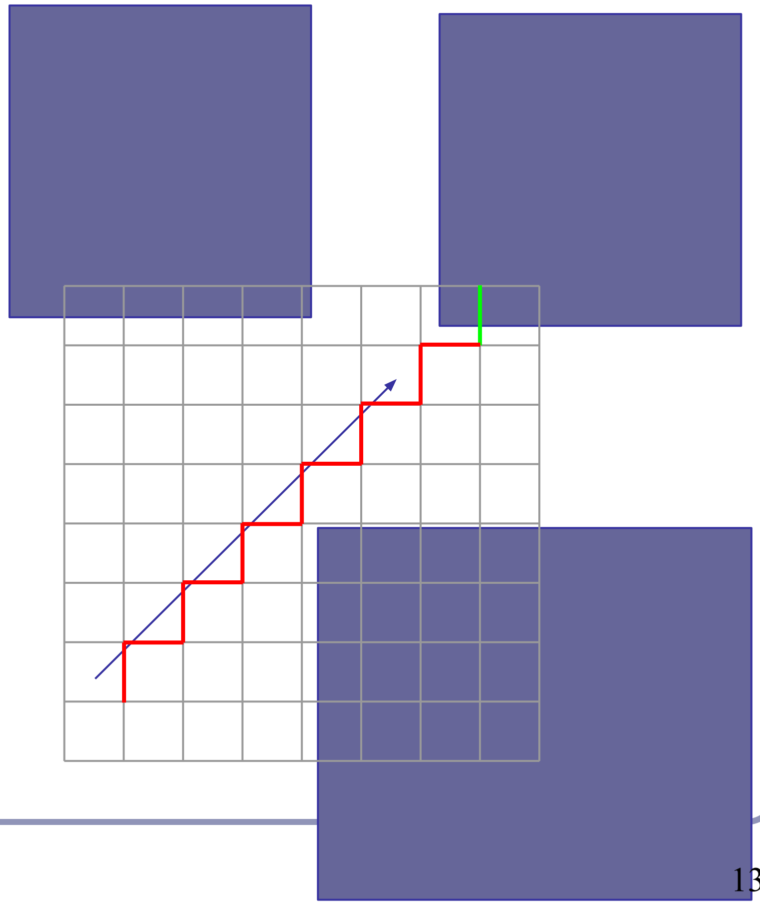
GPU Ray-tracing

```
Hit traceSphere(vec3 rayorig, vec3 raydir, vec3 pos, float radius) {
  float OdotD = dot(rayorig - pos, raydir);
  float OdotO = dot(rayorig - pos, rayorig - pos);
  float base = OdotD * OdotD - OdotO + radius * radius;

  if (base >= 0) {
    float root = sqrt(base);
    float t1 = -OdotD + root;
    float t2 = -OdotD - root;
    if (t1 >= 0 || t2 >= 0) {
      float t = (t1 < t2 && t1 >= 0) ? t1 : t2;
      vec3 pt = rayorig + raydir * t;
      vec3 normal = normalize(pt - pos);
      return Hit(pt, normal, t);
    }
  }
  return Hit(vec3(0), vec3(0), -1);
}
```

An alternative to raytracing: *Ray-marching*

An alternative to classic ray-tracing is ray-marching, in which we take a series of finite steps along the ray until we strike an object or exceed the number of permitted steps.

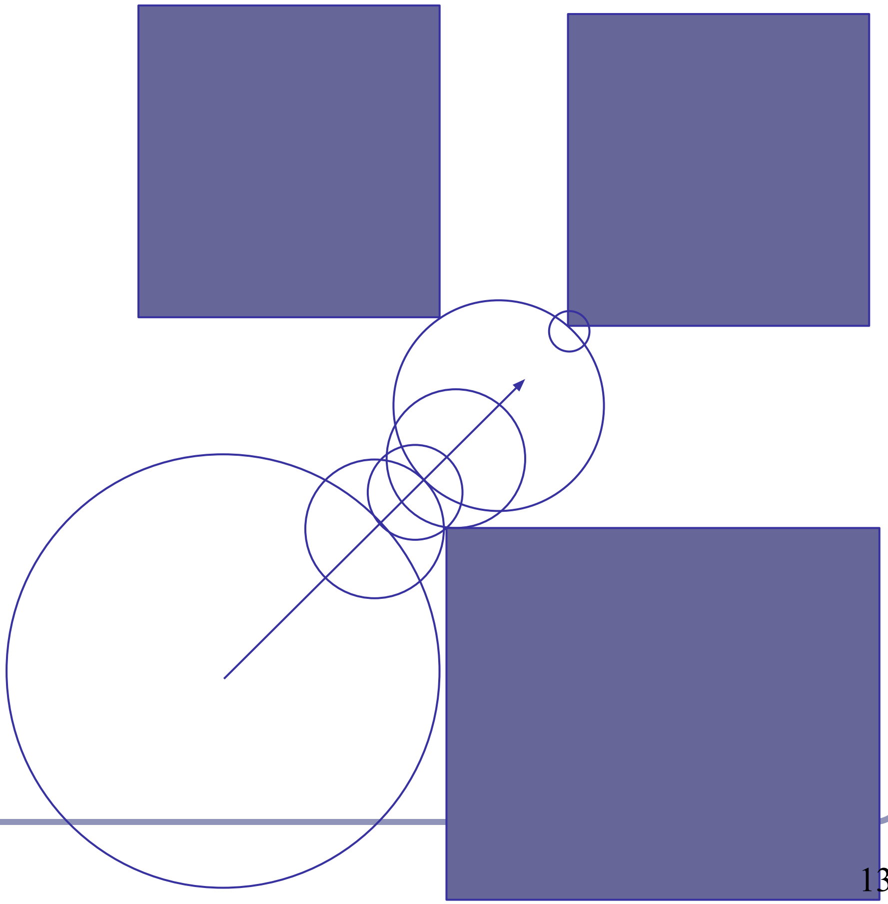
- Also sometimes called ray casting
- Scene objects only need to answer, *"has this ray hit you? y/n"*
- Great solution for data like height fields
- Unfortunately…
  - often involves many steps
  - too large a step size can lead to lost intersections (step over the object)
  - an `if()` test in the heart of a `for()` loop is very hard for the GPU to optimize



GPU Ray-marching: Signed Distance Fields

Ray-marching can be dramatically improved, to impressive realtime GPU performance, using *signed distance fields*:

1. Fire ray into scene
2. At each step, measure distance field function: *d(p)* = [distance to nearest object in scene]
3. Advance ray along ray heading by distance *d*, because the nearest intersection can be no closer than *d*



This is also sometimes called 'sphere tracing'. Early paper:

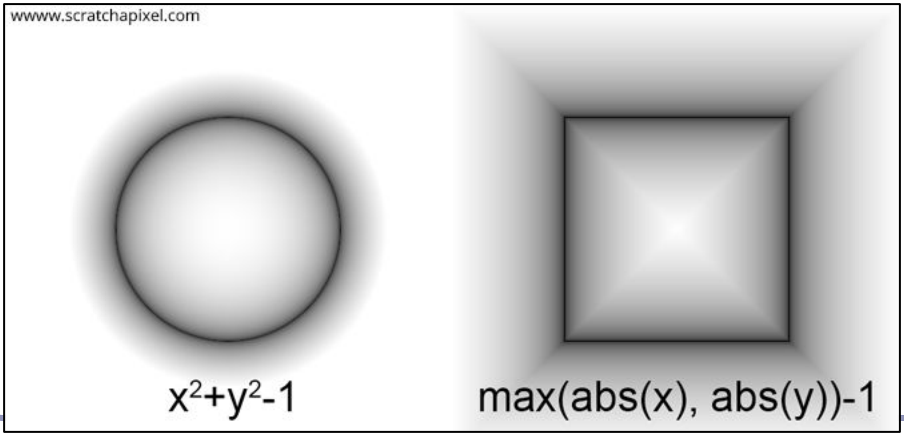
http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf

Signed distance functions

An *SDF* returns the minimum possible distance from point *p* to the surface it describes.

The sphere, for instance, is the distance from *p* to the center of the sphere, minus the radius.

Negative values indicate a sample inside the surface, and still express absolute distance to the surface.




```
float sphere(vec3 p, float r) {
  return length(p) - r;
}

float cube(vec3 p, vec3 dim) {
  vec3 d = abs(p) - dim;
  return min(max(d.x,
      max(d.y, d.z)), 0.0)
      + length(max(d, 0.0));
}

float cylinder(vec3 p, vec3 dim)
{
  return length(p.xz - dim.xy)
      - dim.z;
}

float torus(vec3 p, vec2 t) {
  vec2 q = vec2(
      length(p.xz) - t.x, p.y);
  return length(q) - t.y;
}
```

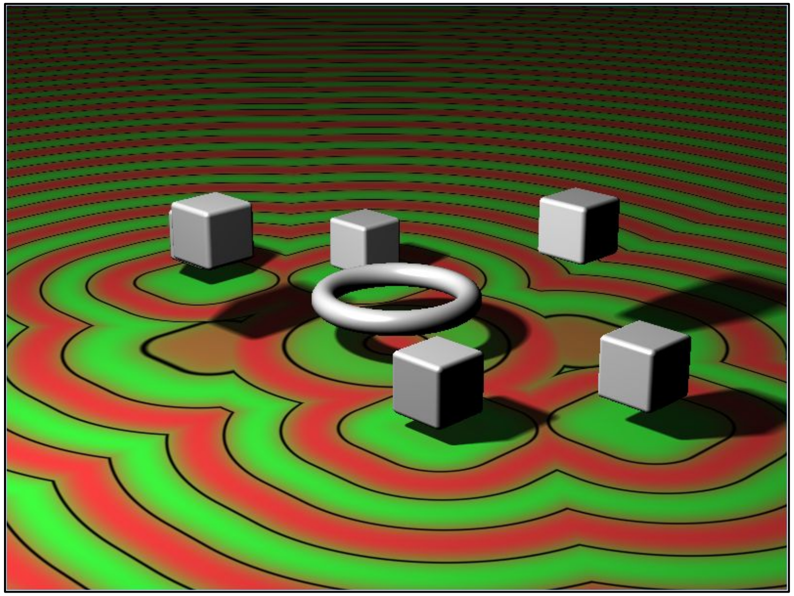
https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields

Raymarching signed distance fields

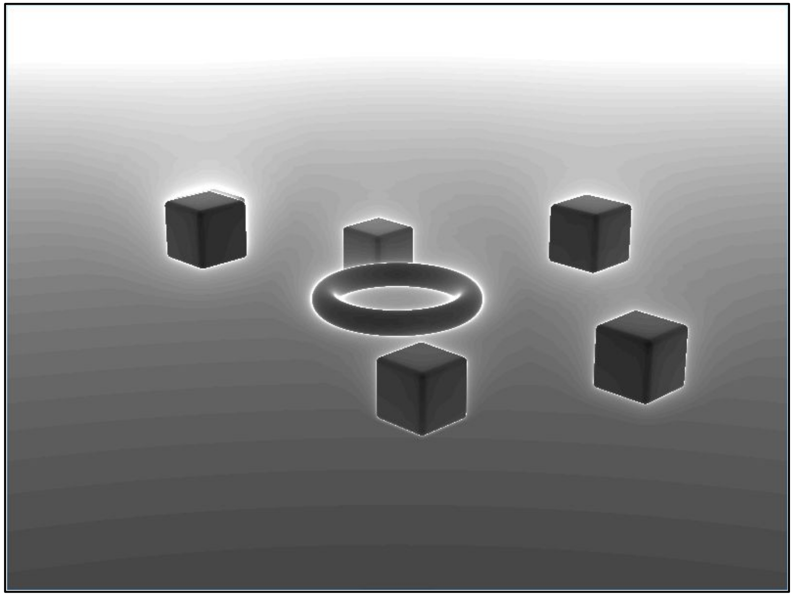
```
vec3 raymarch(vec3 pos, vec3 raydir) {
  int step = 0;
  float d = getSdf(pos);

  while (abs(d) > 0.001 && step < 50) {
    pos = pos + raydir * d;
    d = getSdf(pos);  // Return sphere(pos) or any other
    step++;
  }

  return
      (step < 50) ? illuminate(pos, rayorig) : background;
}
```

Visualizing step count

Final image

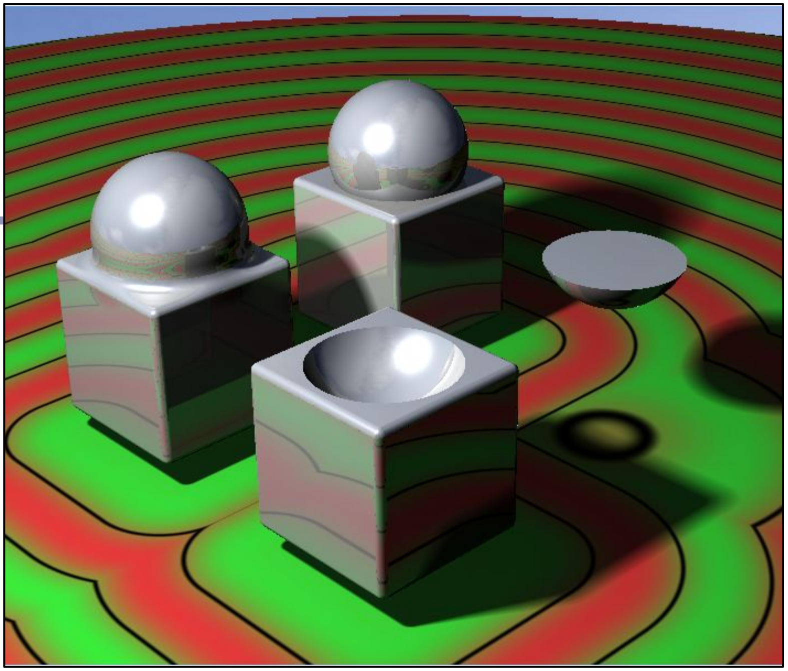


Distance field



Brighter = more steps, up to 50

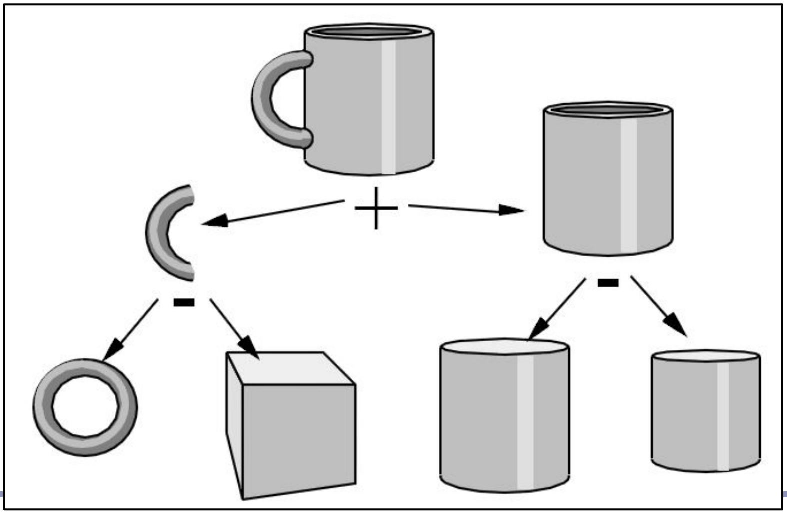
Combining SDFs



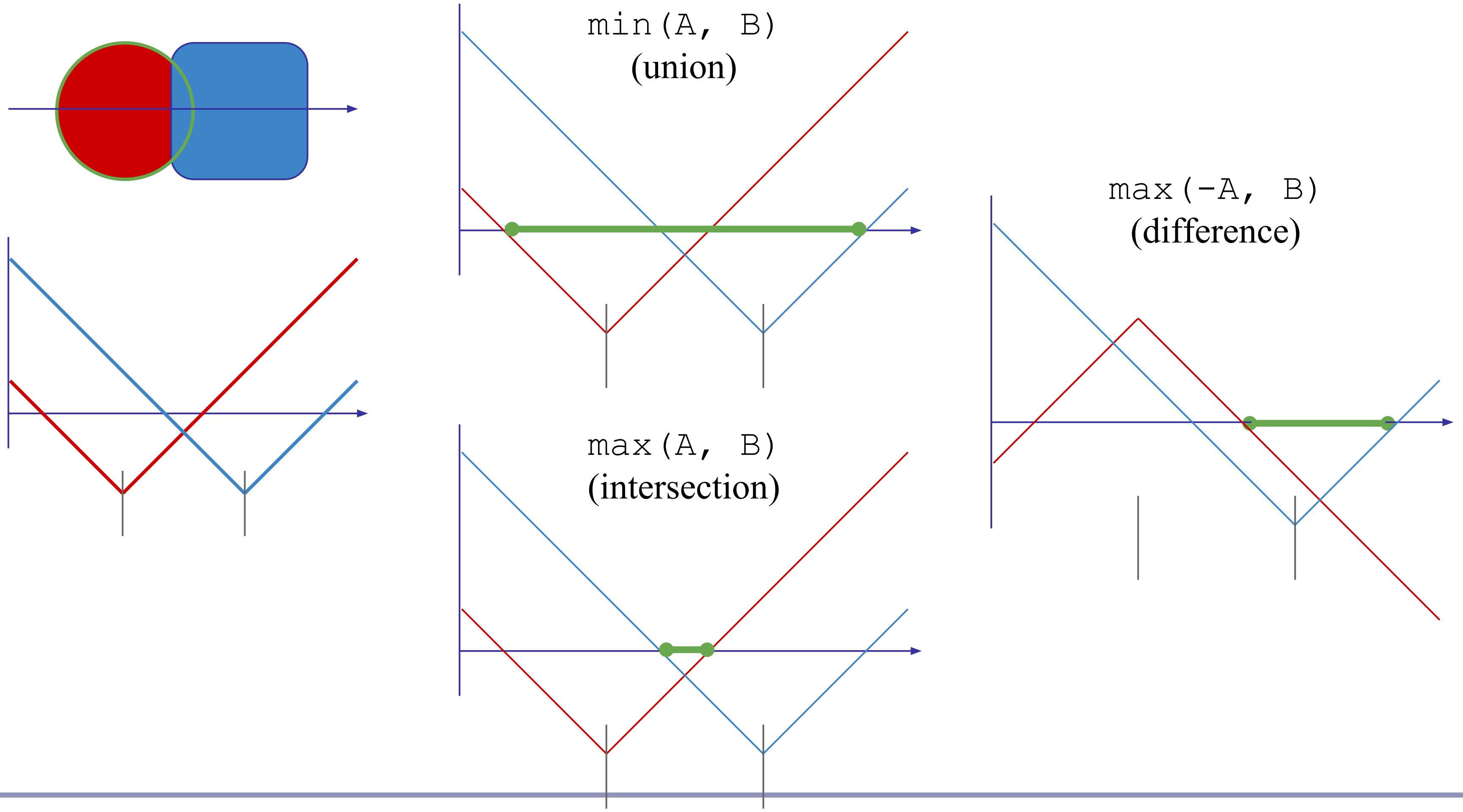
We combine SDF models by choosing which is closer to the sampled point.

- Take the **union** of two SDFs by taking the `min()` of their functions.
- Take the **intersection** of two SDFs by taking the `max()` of their functions.
- The `max()` of function A and the negative of function B will return the **difference** of A - B.

By combining these binary operations we can create functions which describe very complex primitives.

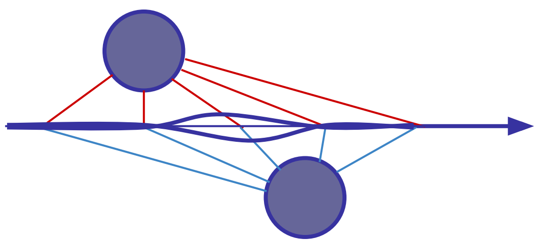


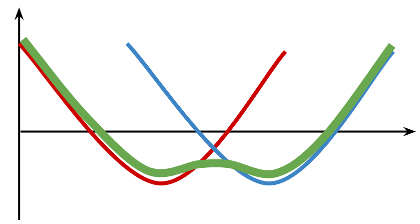
Combining SDFs

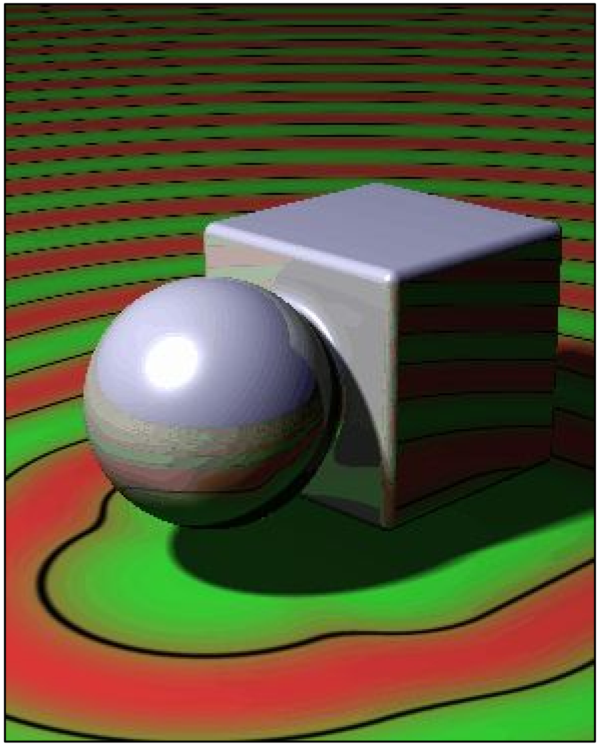


Blending SDFs

Taking the `min()`, `max()`, etc of two SDFs yields a sharp discontinuity. *Interpolating* the two SDFs with a smooth polynomial yields a smooth distance curve, blending the models:







Sample blending function (Quilez)

```
float blend(float a, float b, float k) {
  a = pow(a, k);
  b = pow(b, k);
  return pow((a * b) / (a + b), 1.0 / k);
}
```

Transforming SDF geometry

To rotate, translate or scale an SDF model, apply the inverse transform to the input point within your distance function.

Ex:

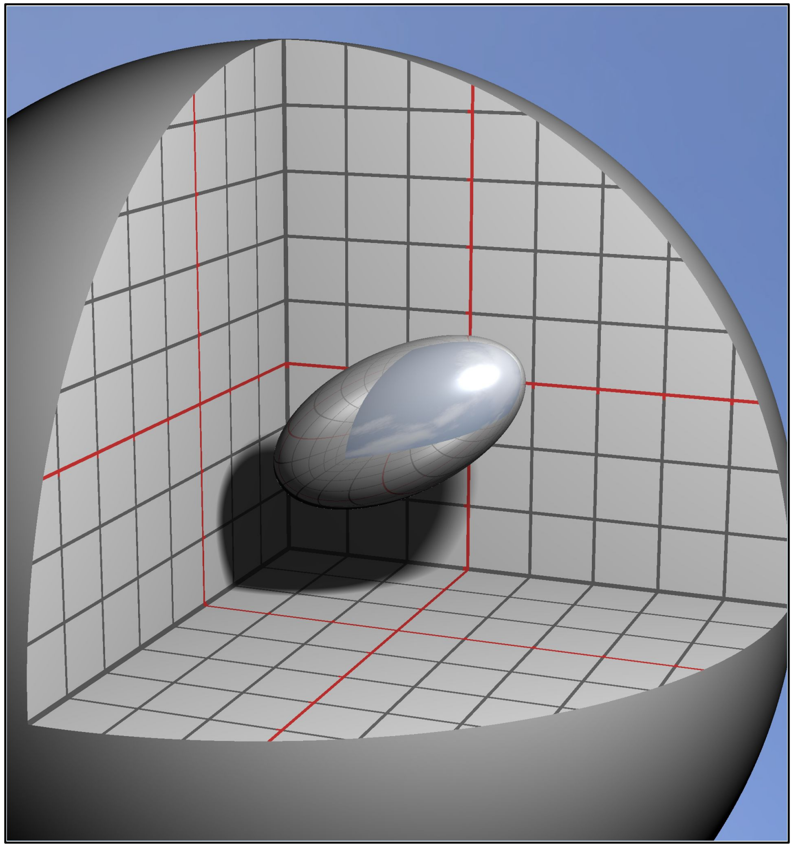
```
float sphere(vec3 pt, float radius) {
  return length(pt) - radius;
}

float f(vec3 pt) {
  return sphere(pt - vec3(0, 3, 0));
}
```

This renders a sphere centered at `(0, 3, 0)`.

More prosaically, assemble your local-to-world transform as usual, but apply its inverse to the pt within your distance function.

Transforming SDF geometry

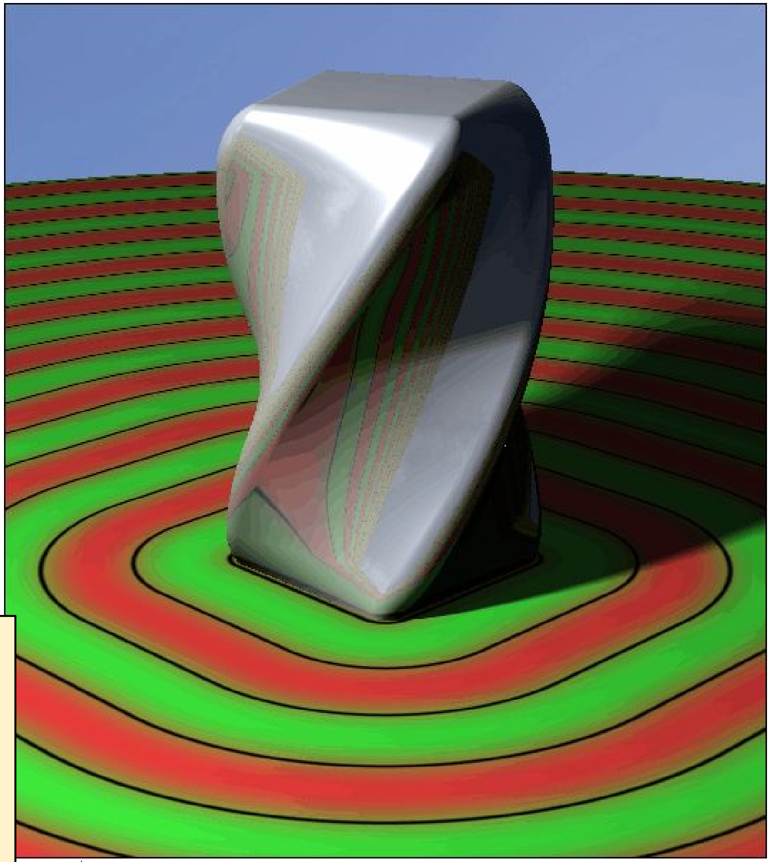
```
float fScene(vec3 pt) {

  // Scale 2x along X
  mat4 S = mat4(
      vec4(2, 0, 0, 0),
      vec4(0, 1, 0, 0),
      vec4(0, 0, 1, 0),
      vec4(0, 0, 0, 1));

  // Rotation in XY
  float t = sin(time) * PI / 4;
  mat4 R = mat4(
      vec4(cos(t),  sin(t), 0, 0),
      vec4(-sin(t), cos(t), 0, 0),
      vec4(0,       0,      1, 0),
      vec4(0,       0,      0, 1));

  // Translate to (3, 3, 3)
  mat4 T = mat4(
      vec4(1, 0, 0, 3),
      vec4(0, 1, 0, 3),
      vec4(0, 0, 1, 3),
      vec4(0, 0, 0, 1));

  pt = (vec4(pt, 1) * inverse(S * R * T)).xyz;

  return sdSphere(pt, 1);
}
```



Transforming SDF geometry

The previous example modified ‘all of space’ with the same transform, so its distance functions retain their local linearity.

We can also apply non-uniform spatial distortion, such as by choosing how much we’ll modify space as a function of where in space we are.



```
float fScene(vec3 pt) {
  pt.y -= 1;
  float t = (pt.y + 2.5) * sin(time);
  return sdCube(vec3(
    pt.x * cos(t) - pt.z * sin(t),
    pt.y / 2,
    pt.x * sin(t) + pt.z * cos(t)), vec3(1));
}
```

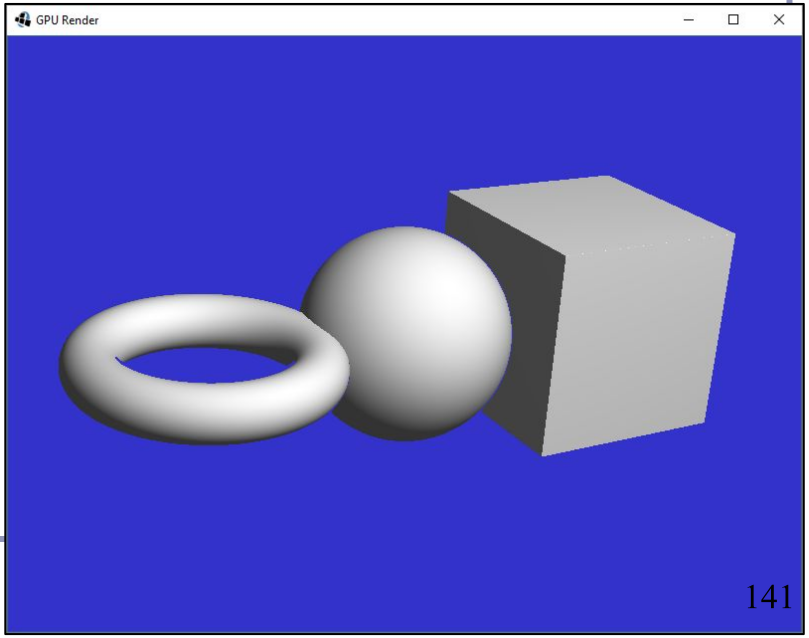
Find the normal to an SDF

Finding the normal: local gradient

```
float d = getSdf(pt);
vec3 normal = normalize(vec3(
    getSdf(vec3(pt.x + 0.0001, pt.y, pt.z)) - d,
    getSdf(vec3(pt.x, pt.y + 0.0001, pt.z)) - d,
    getSdf(vec3(pt.x, pt.y, pt.z + 0.0001)) - d));
```

The distance function is locally linear and changes most as the sample moves directly away from the surface. At the surface, the direction of greatest change is therefore equivalent to the normal to the surface.

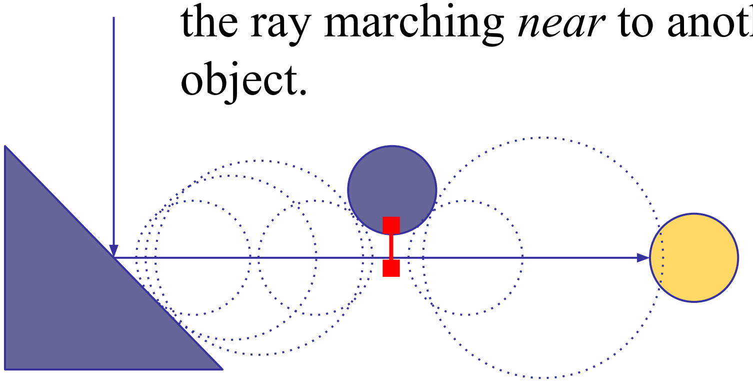
Thus the local gradient (the normal) can be approximated from the distance function.

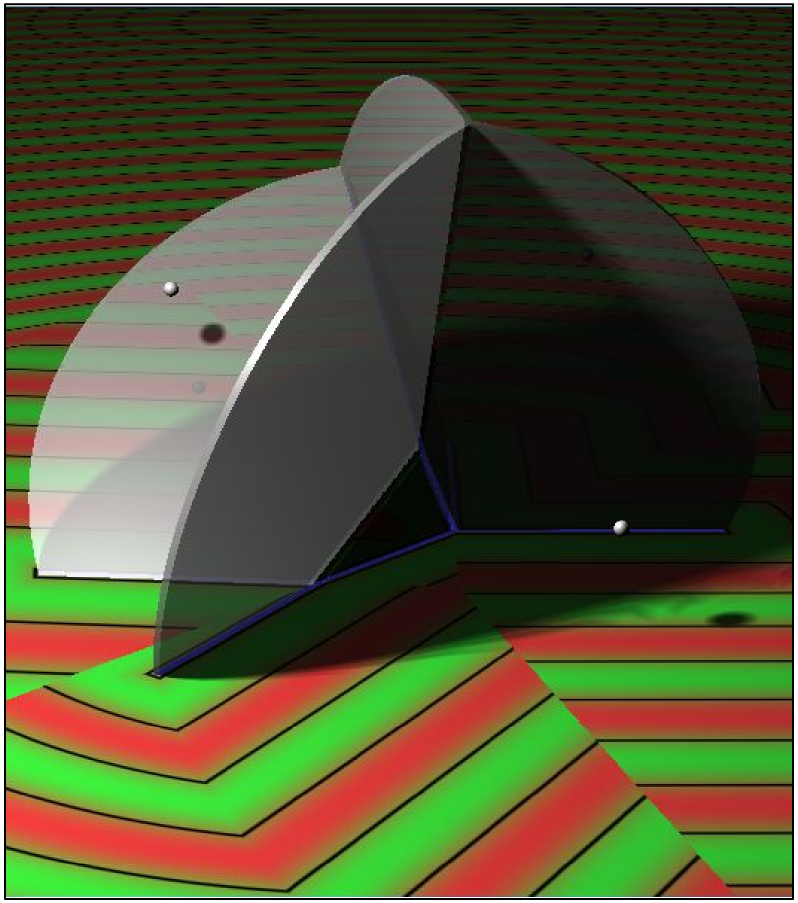


SDF shadows

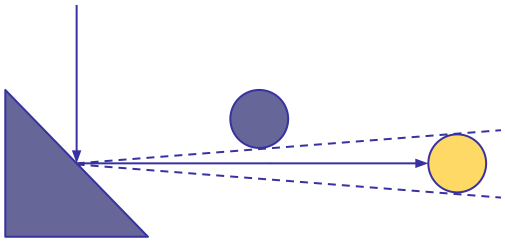
Ray-marched shadows are straightforward: march a ray towards each light source, don't illuminate if the SDF ever drops too close to zero.

Unlike ray-tracing, soft shadows are almost free with SDFs: attenuate illumination by a linear function of the ray marching *near* to another object.



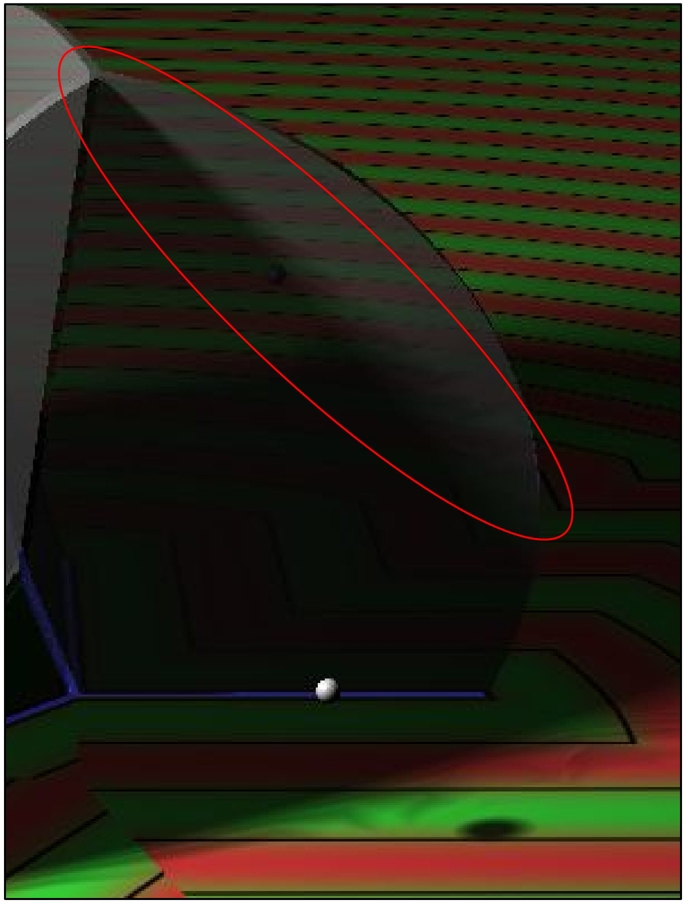


Soft SDF shadows



```
float shadow(vec3 pt) {
  vec3 lightDir = normalize(lightPos - pt);
  float kd = 1;
  int step = 0;

  for (float t = 0.1;
      t < length(lightPos - pt)
      && step < renderDepth && kd > 0.001; ) {
    float d = abs(getSDF(pt + t * lightDir));
    if (d < 0.001) {
      kd = 0;
    } else {
      kd = min(kd, 16 * d / t);
    }
    t += d;
    step++;
  }
  return kd;
}
```

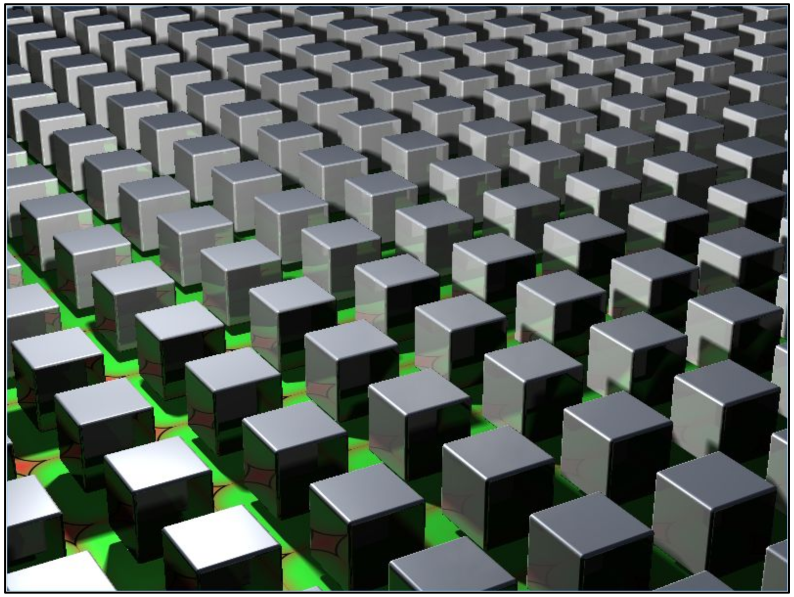
By dividing *d* by *t*, we attenuate the strength of the shadow as its source is further from the illuminated point.



Repeating SDF geometry

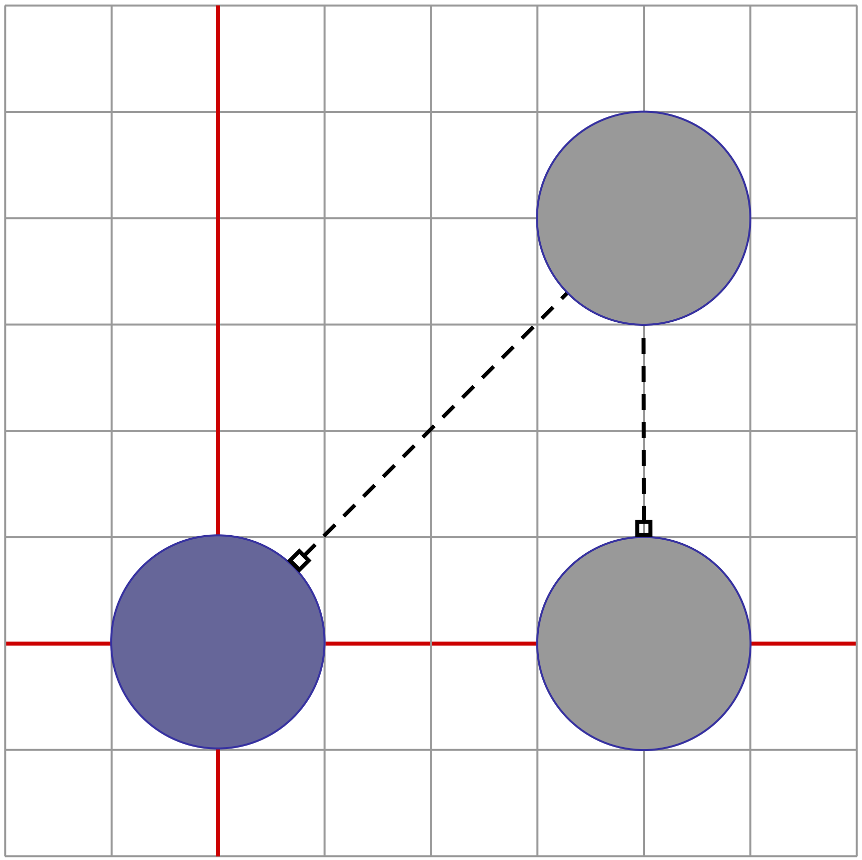
If we take the modulus of a point's position along one or more axes before computing its signed distance, then we segment space into infinite parallel regions of repeated distance. Space near the origin 'repeats'.

With SDFs we get infinite repetition of geometry for no extra cost.



```
float fScene(vec3 pt) {
  vec3 pos;
  pos = vec3(mod(pt.x + 2, 4) - 2, pt.y, mod(pt.z + 2, 4) - 2);
  return sdCube(pos, vec3(1));
}
```

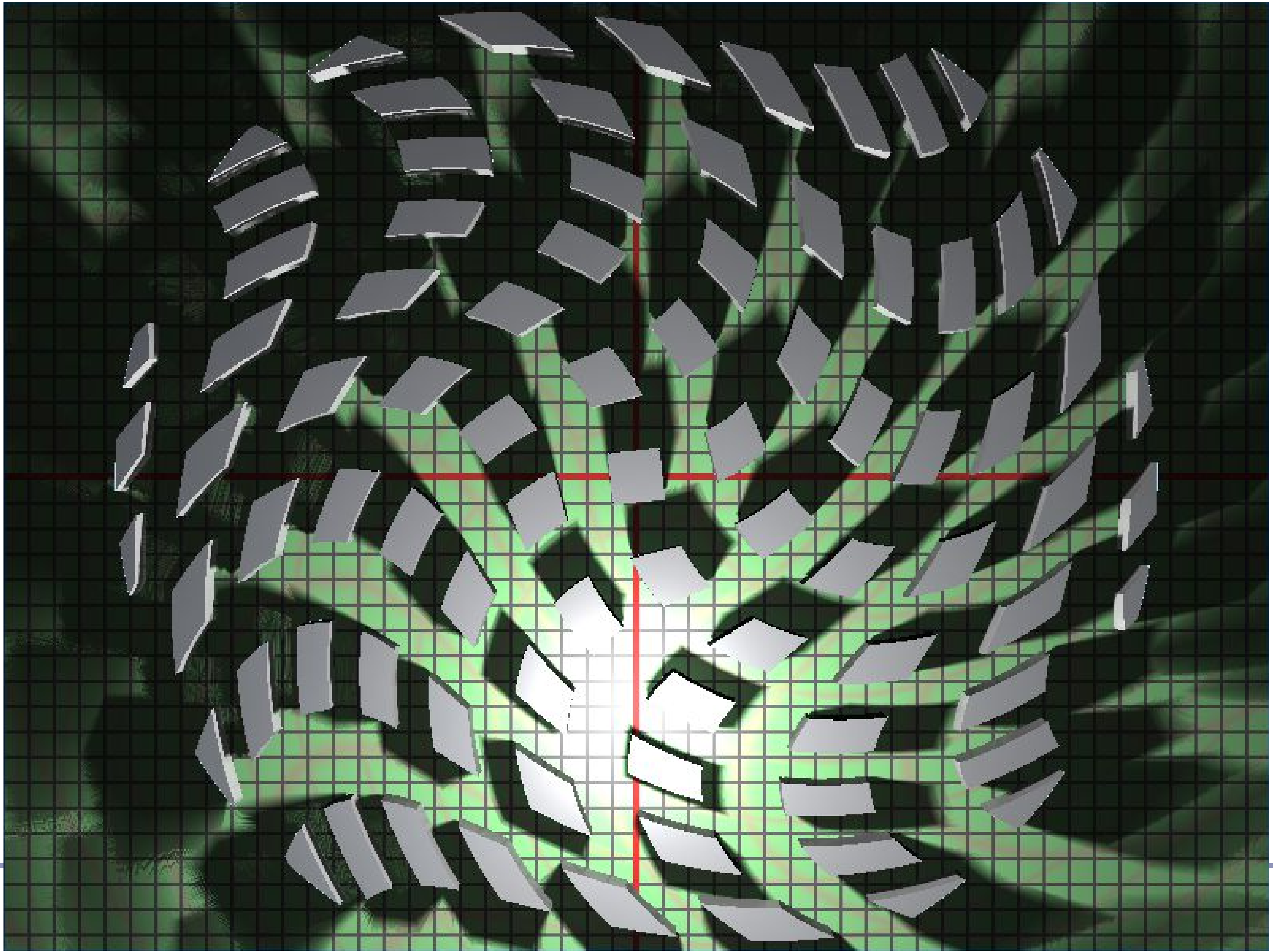
Repeating SDF geometry



```
float sphere(vec3 pt, float radius) {
  return length(pt) - radius;
}
```

- sdSphere(4, 4)
  = √(4*4+4*4) - 1
  = ~4.5
- sdSphere(
  ((4 + 2) % 4) - 2, 4)
  = √(0*0+4*4) - 1
  = 3
- sdSphere(
  ((4 + 2) % 4) - 2,
  ((4 + 2) % 4) - 2)
  = √(0*0+0*0) - 1
  = -1 // Inside surface

SDF - Live demo



Recommended reading

Seminal papers:

- John C. Hart et al., "Ray Tracing Deterministic 3-D Fractals", http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf
- John C. Hart, "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces", http://graphics.cs.illinois.edu/papers/zeno

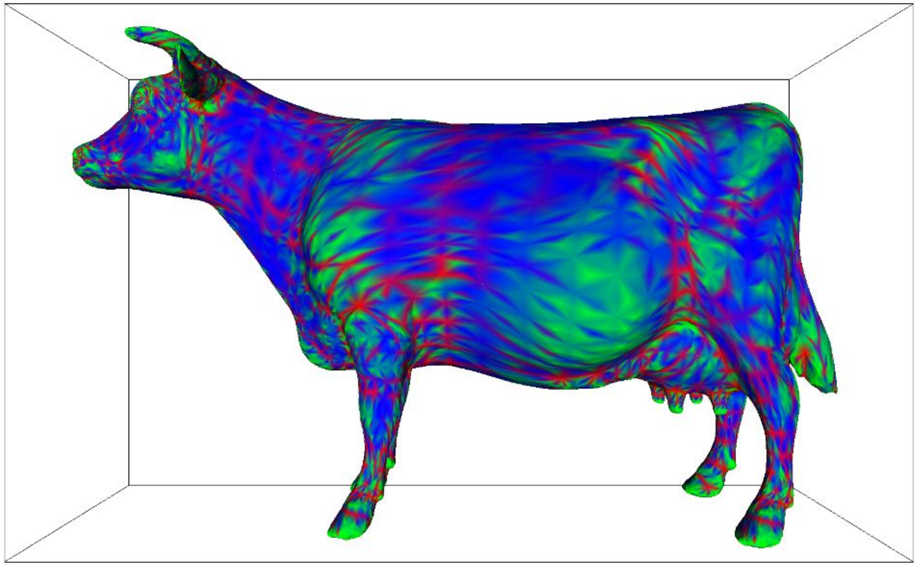
Special kudos to Inigo Quilez and his amazing blog:

- http://iquilezles.org/www/articles/smin/smin.htm
- http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

Other useful sources:

- Johann Korndorfer, "How to Create Content with Signed Distance Functions", https://www.youtube.com/watch?v=s8nFqwOho-s
- Daniel Wright, "Dynamic Occlusion with Signed Distance Fields", http://advances.realtimerendering.com/s2015/DynamicOcclusionWithSignedDistanceFields.pdf
- 9bit Science, "Raymarching Distance Fields", http://9bitscience.blogspot.co.uk/2013/07/raymarching-distance-fields_14.html

Advanced Graphics



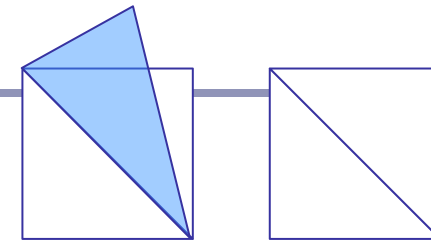
Computational Geometry and Implicit Surfaces

Alex Benton, University of Cambridge – alex@bentonian.com

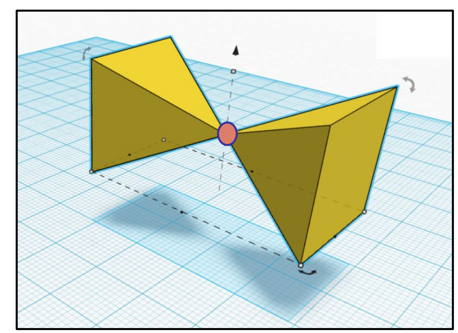
Supported in part by Google UK, Ltd

Terminology

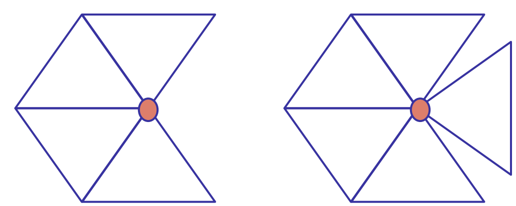
- We'll be focusing on *discrete* (as opposed to continuous) representation of geometry; i.e., polygon meshes
  - Many rendering systems limit themselves to triangle meshes
  - Many require that the mesh be *manifold*
- In a *closed manifold* polygon mesh:
  - Exactly two triangles meet at each edge
  - 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

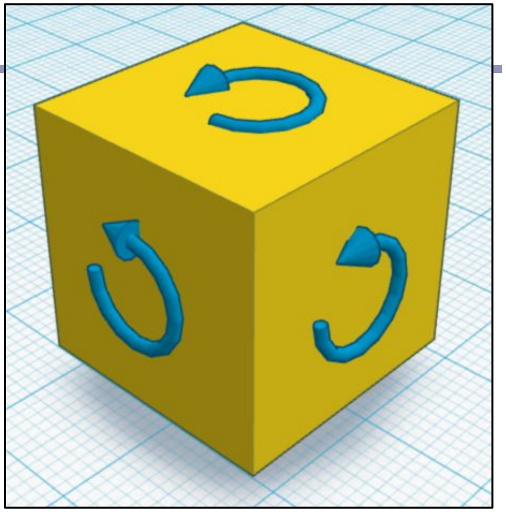


Vertex: Good boundary vs bad

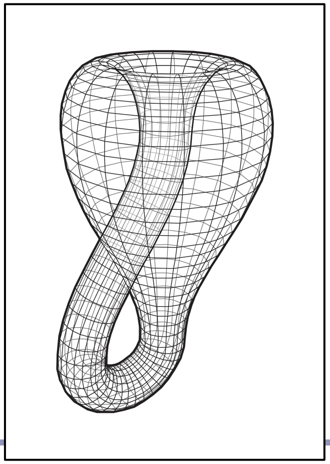
This slide draws much inspiration from Shirley and Marschner's *Fundamentals of Computer Graphics*, pp. 262-263

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.

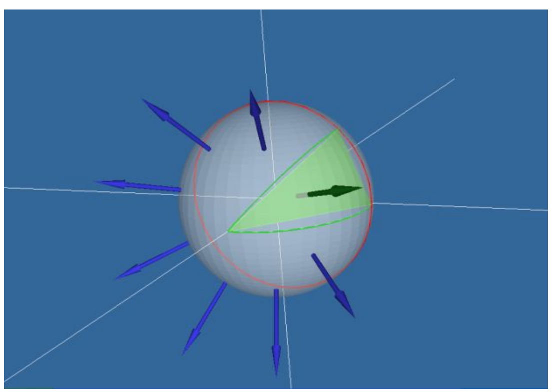
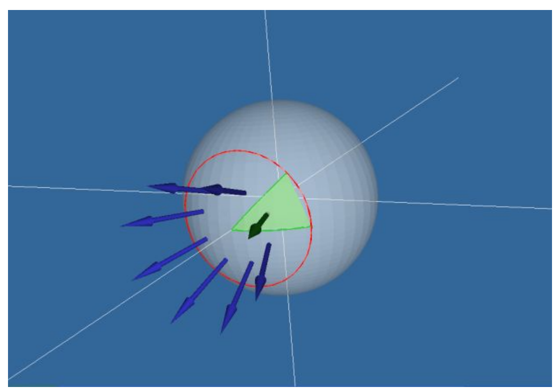
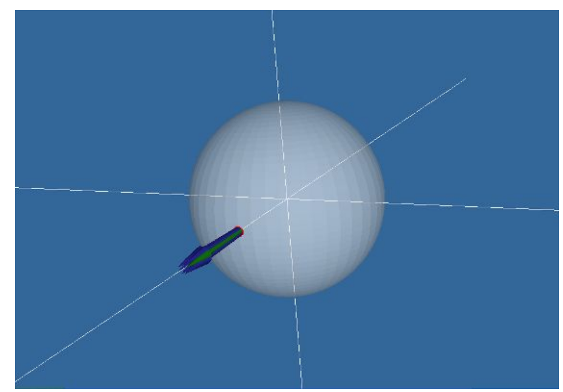
Also, terrible for holding drinks.

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

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*

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

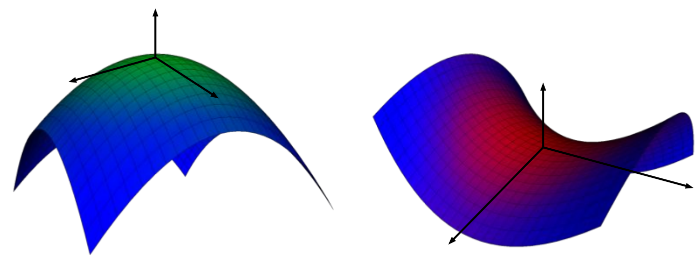
$$N(v) = \frac{\sum_F \alpha(F, v)\, N_F}{\left|\sum_F \alpha(F, v)\right|}$$

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*.



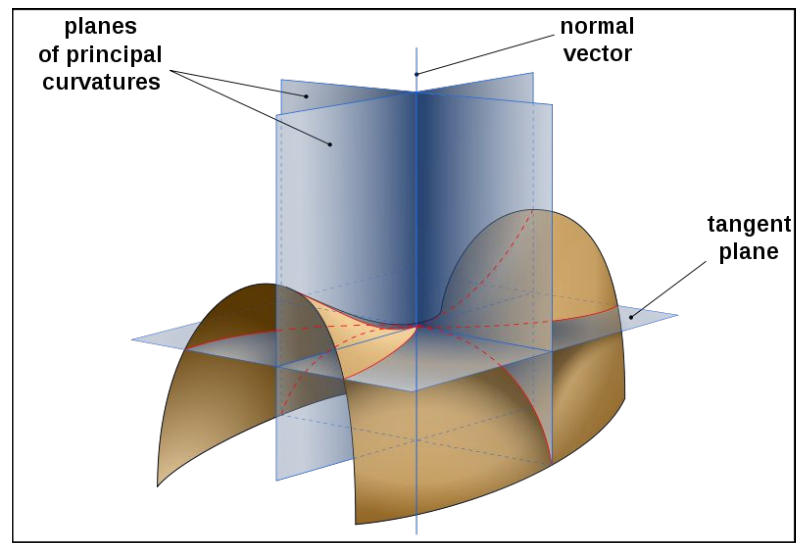
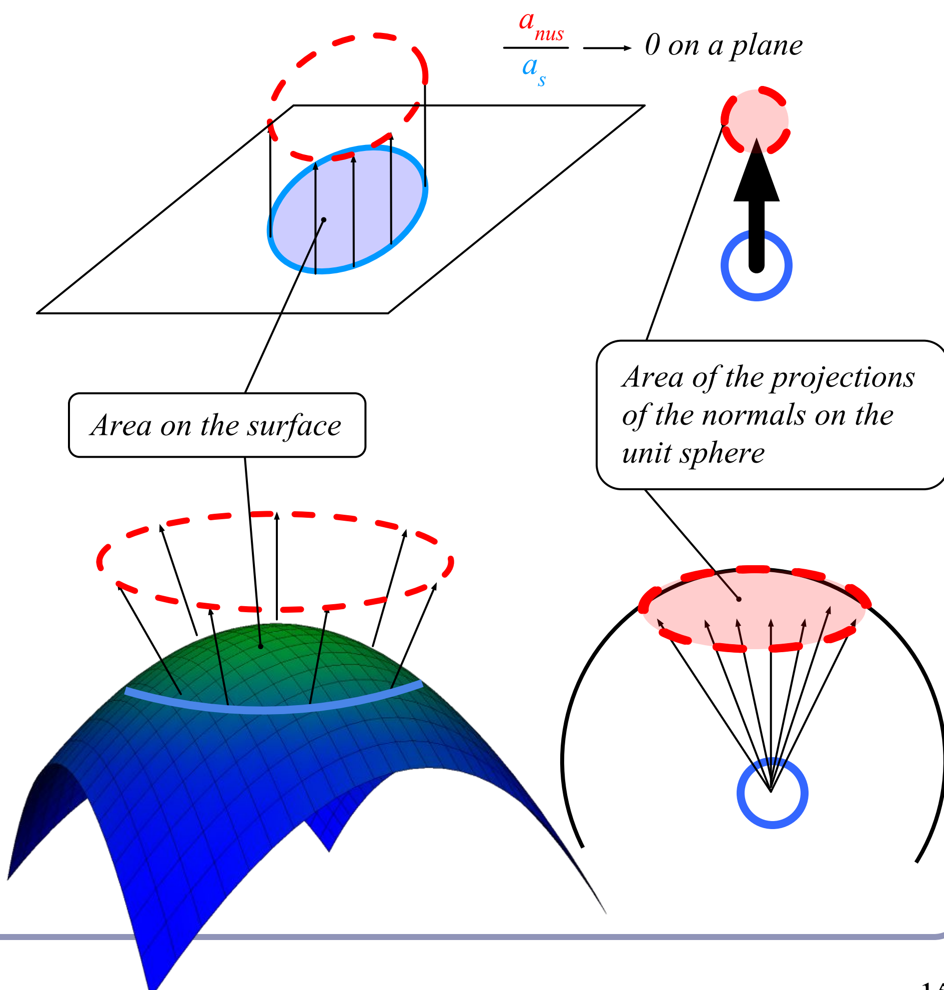


Image by Eric Gaba, from Wikipedia

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.

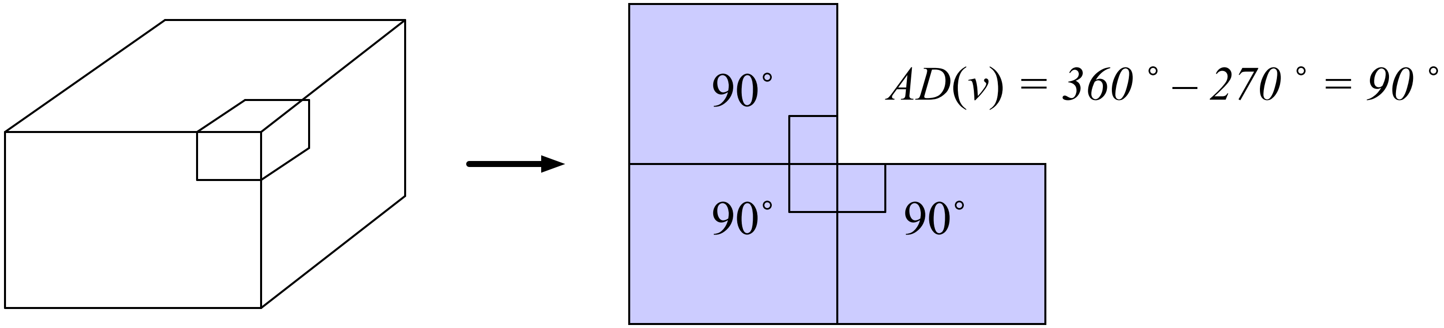
- Normals change instantaneously (as one's point of view travels across an edge from one face to another) or not at all (as one's point of view travels within a face.)

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

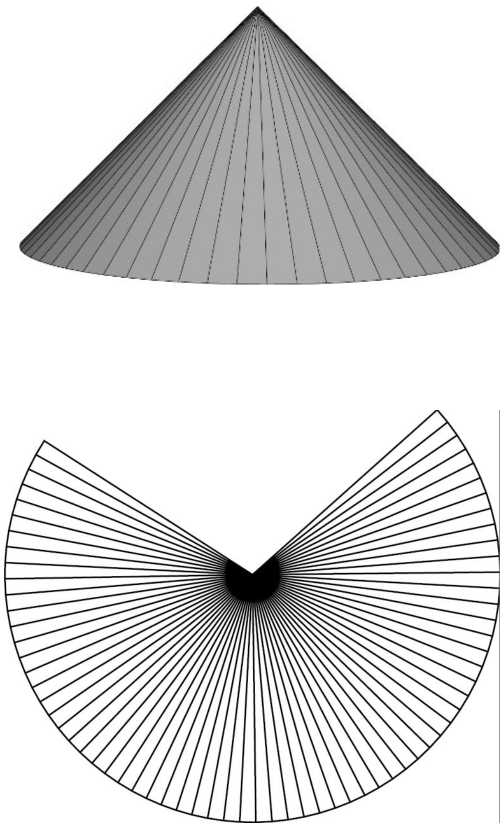
Angle deficit – a better solution for measuring discrete curvature

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.

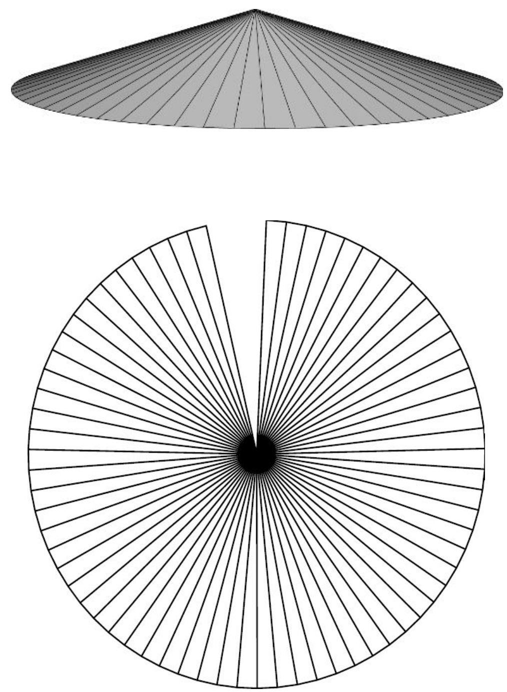
$$AD(v) = 2\pi - \sum_F \alpha(F, v)$$



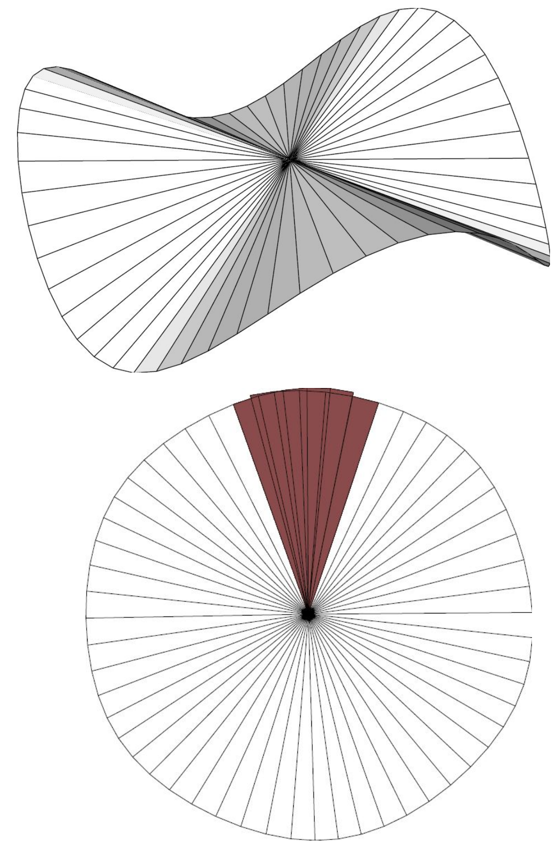
Angle deficit



High angle deficit

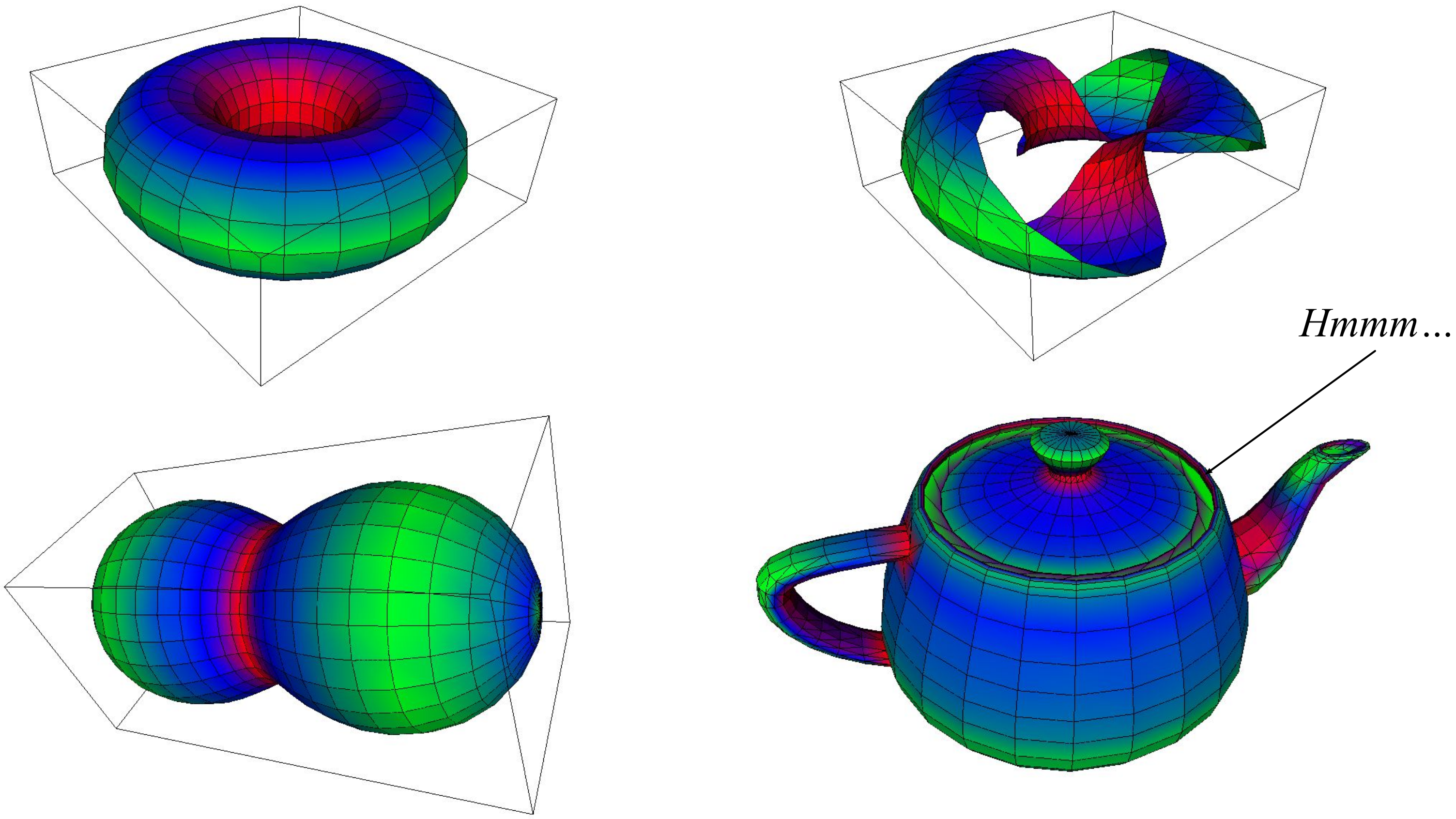


Low angle deficit



Negative angle deficit

Angle deficit



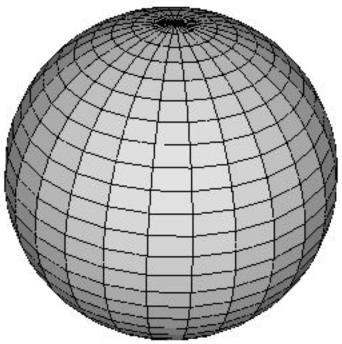
Genus, Poincaré and the Euler Characteristic

- 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."

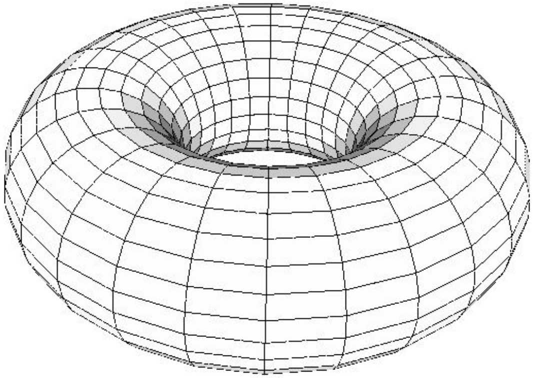
  --*mathworld.com*

- Informally, it's the number of coffee cup handles in the surface.



Genus 0

Genus 1



Genus, Poincaré and the Euler Characteristic

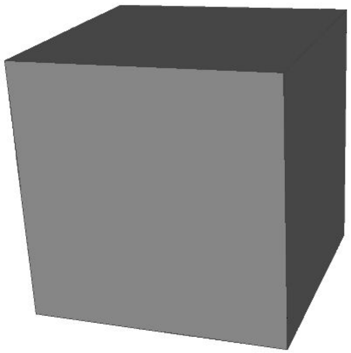
Given a polyhedral surface S without border where:

- 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,

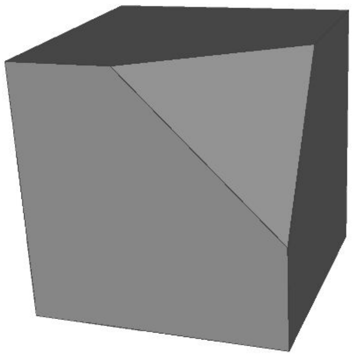
the Poincaré Formula states that:

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

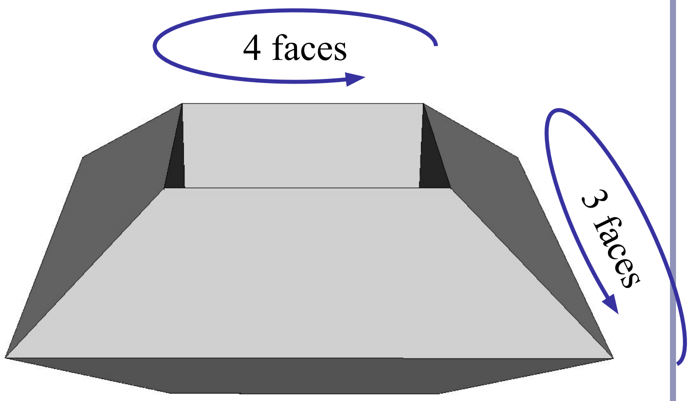
Genus, Poincaré and the Euler Characteristic



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



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



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

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:

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

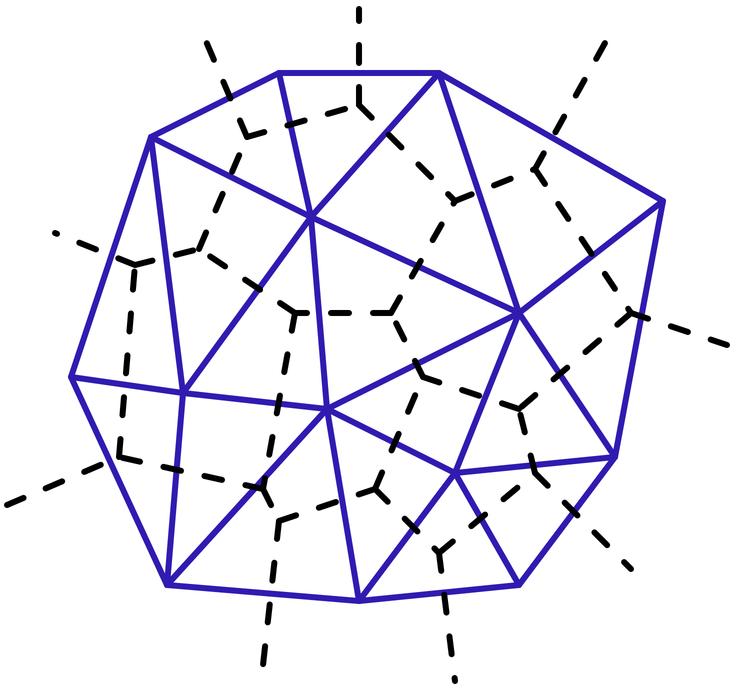
Tetrahedron:

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

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"…

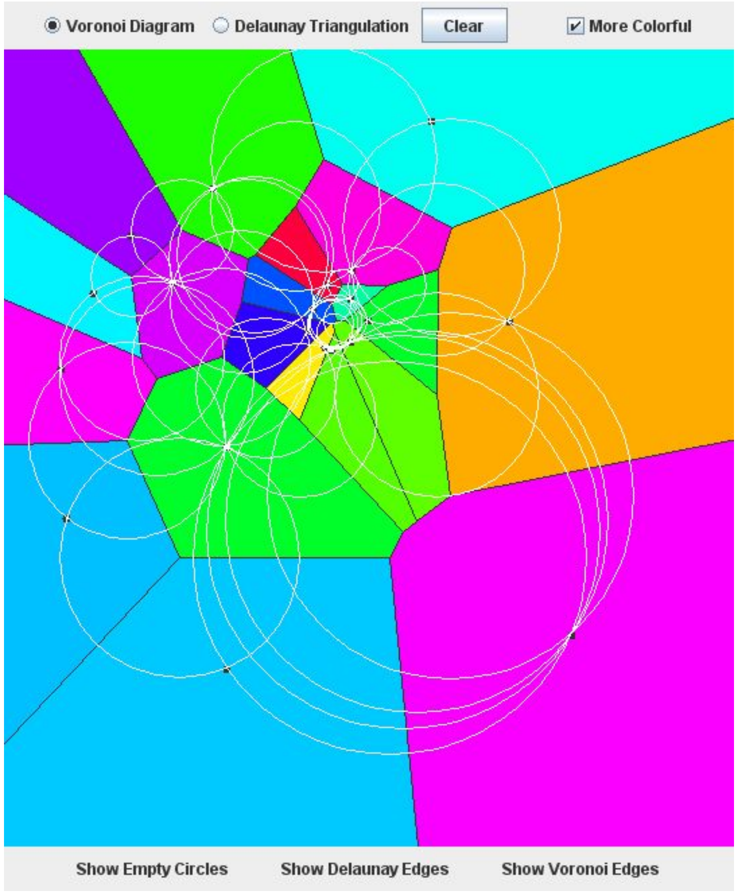
Voronoi diagrams

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 <u>equiangular</u>.

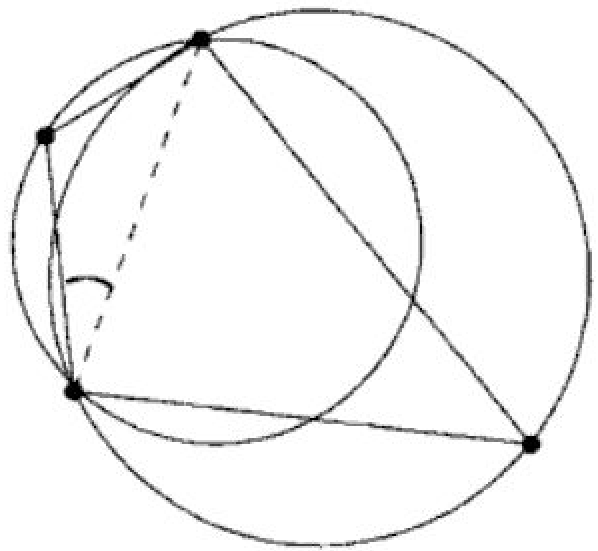


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

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.

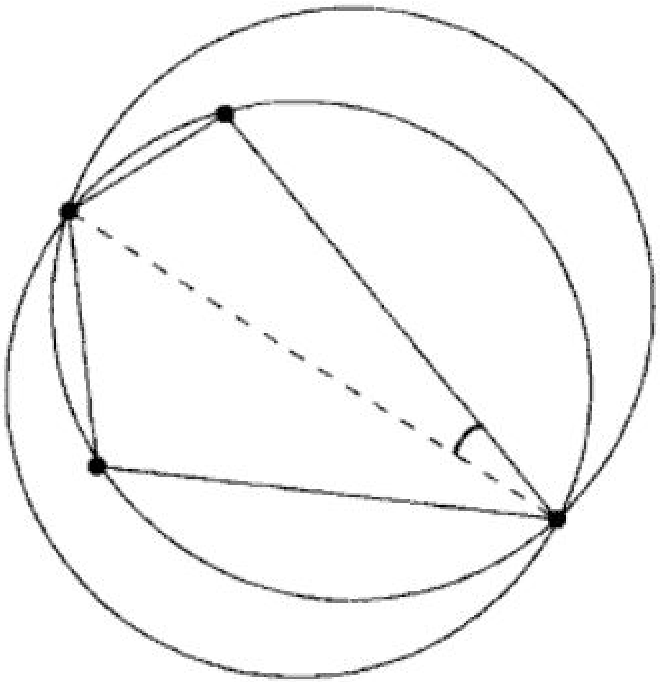


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

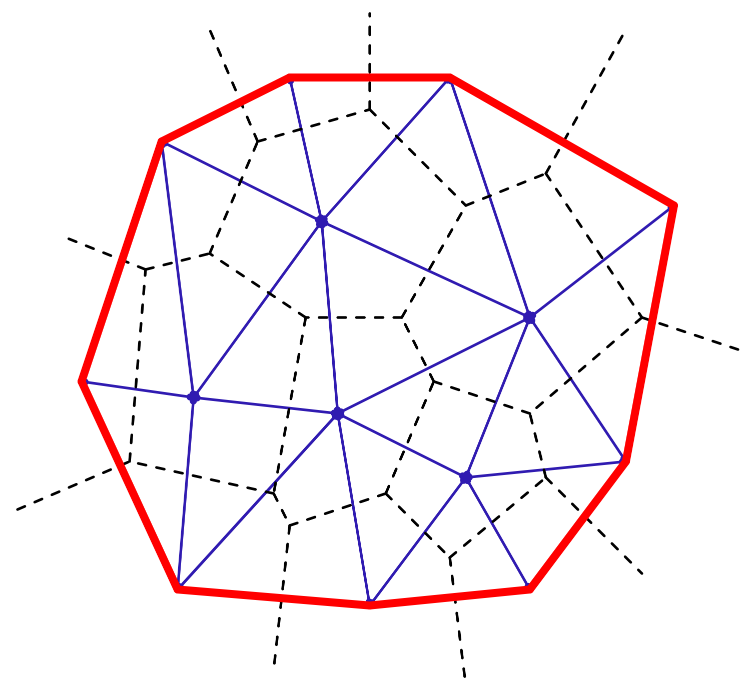
Delaunay triangulations and convex hulls

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}.

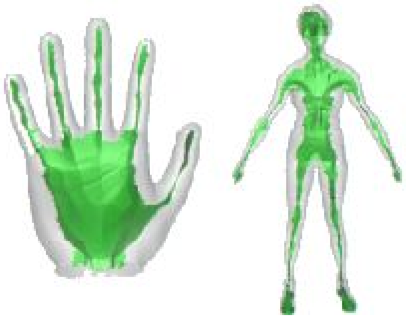
- Ex: from 2D ($P_i=\{x,y\}_i$), loft the points upwards, onto a parabola in 3D ($P'_i=\{x,y,x^2+y^2\}_i$). The resulting polyhedral mesh will still be convex in 3D.



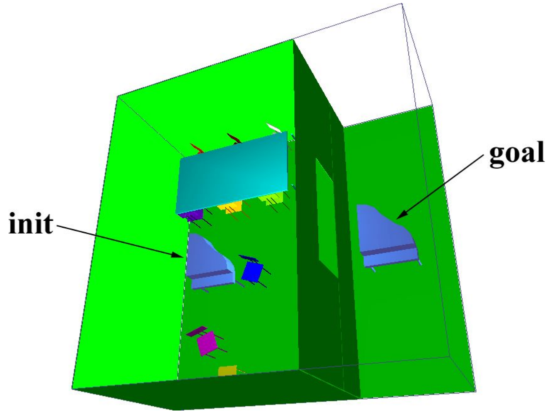
Voronoi diagrams and the *medial axis*

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.

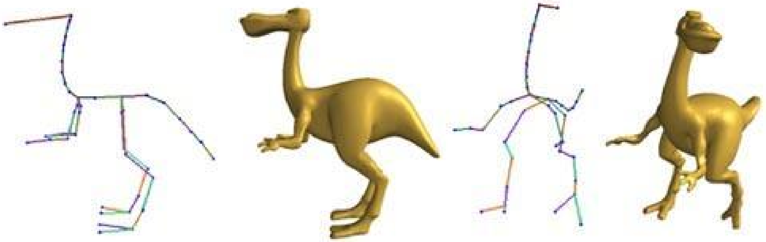
- This can be used to extract a skeleton of the surface, for (for example) path-planning solutions, surface deformation, and animation.



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

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)

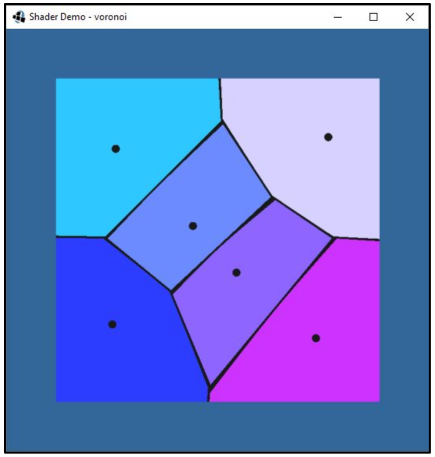
GPU-accelerated Voronoi Diagrams

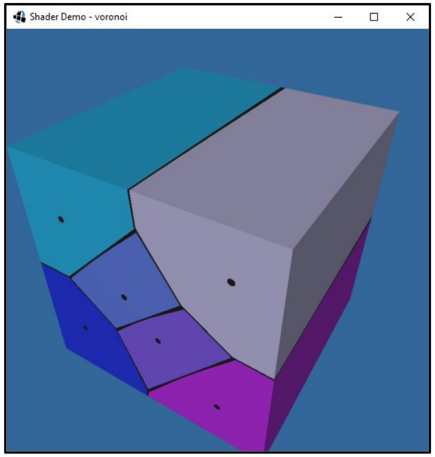
Brute force:

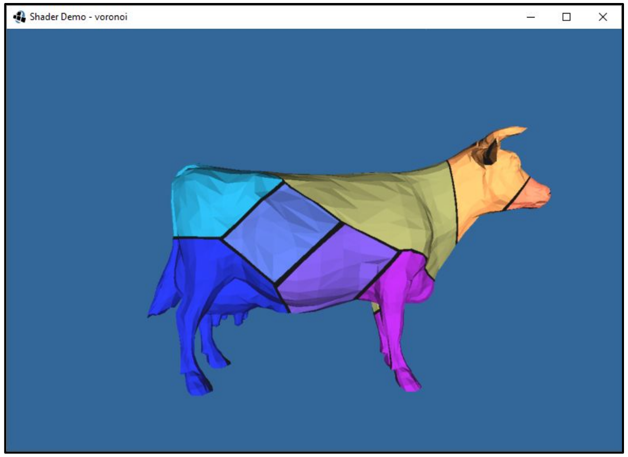
- For each pixel to be rendered on the GPU, search all points for the nearest point

Elegant (and 2D only):

- Render each point as a discrete 3D cone in isometric projection, let z-buffering sort it out





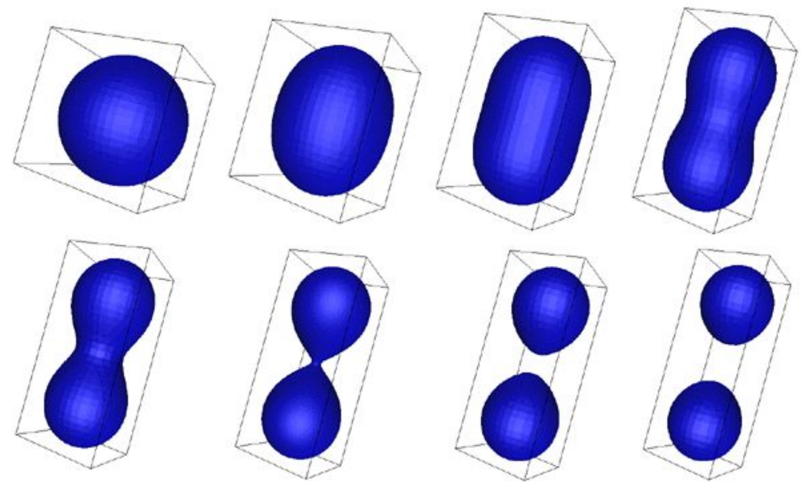


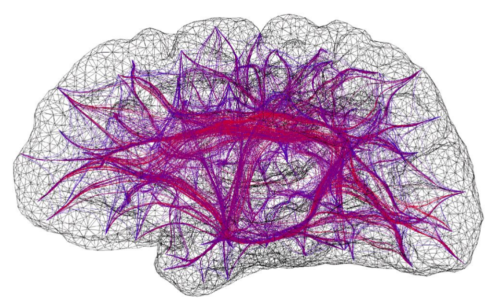
Implicit surfaces

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*; each point in space generates a field of force, which drops off as a function of distance from the point. This 3D field of forces defines an *implicit surface*: the set of all the points in space where the force field sums to a key value.

A few popular force field functions:

- "Blobby Molecules" – Jim Blinn

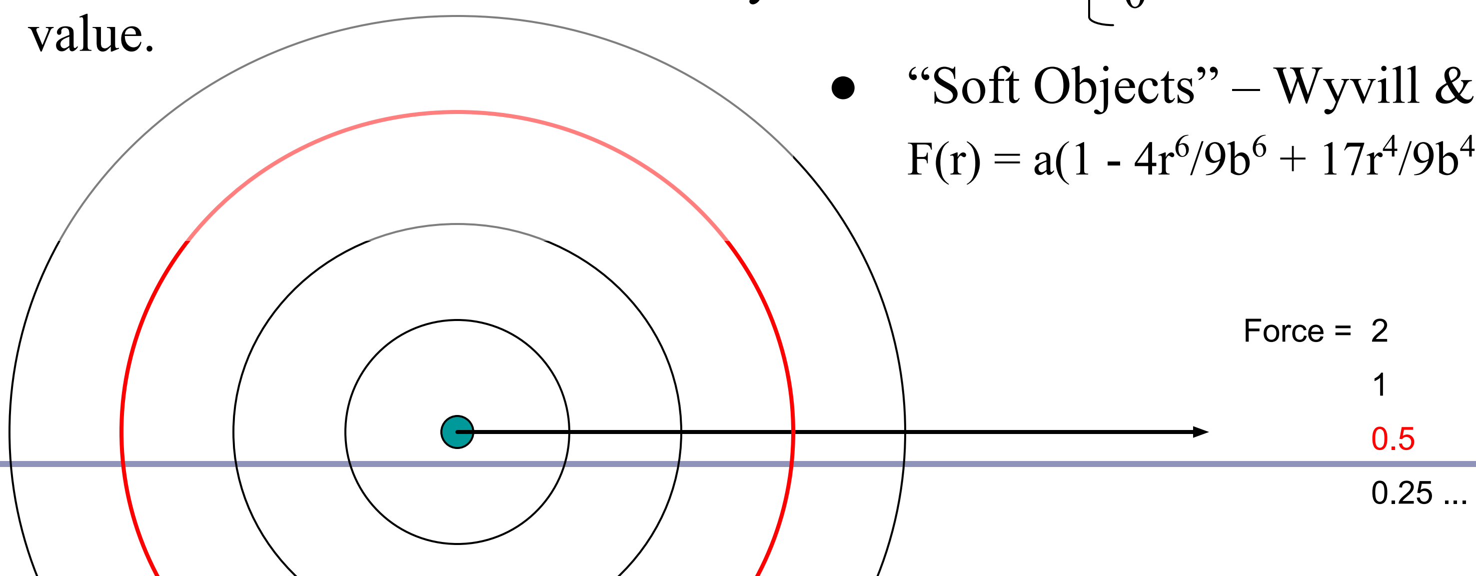
  $F(r) = a\, e^{-br^2}$

- "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

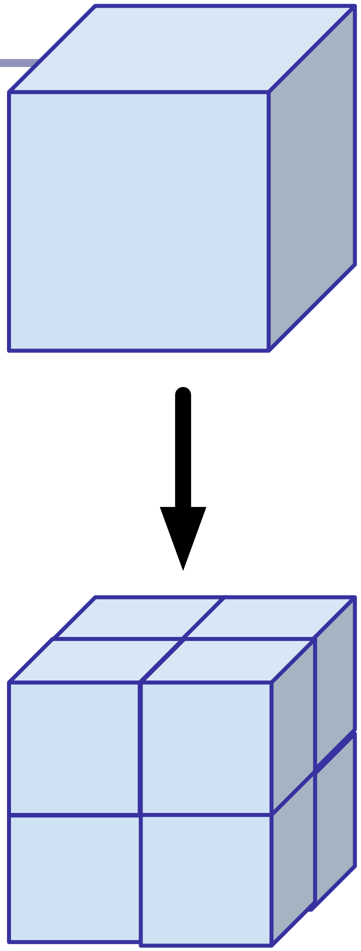
  $F(r) = a(1 - 4r^6/9b^6 + 17r^4/9b^4 - 22r^2 / 9b^2)$



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

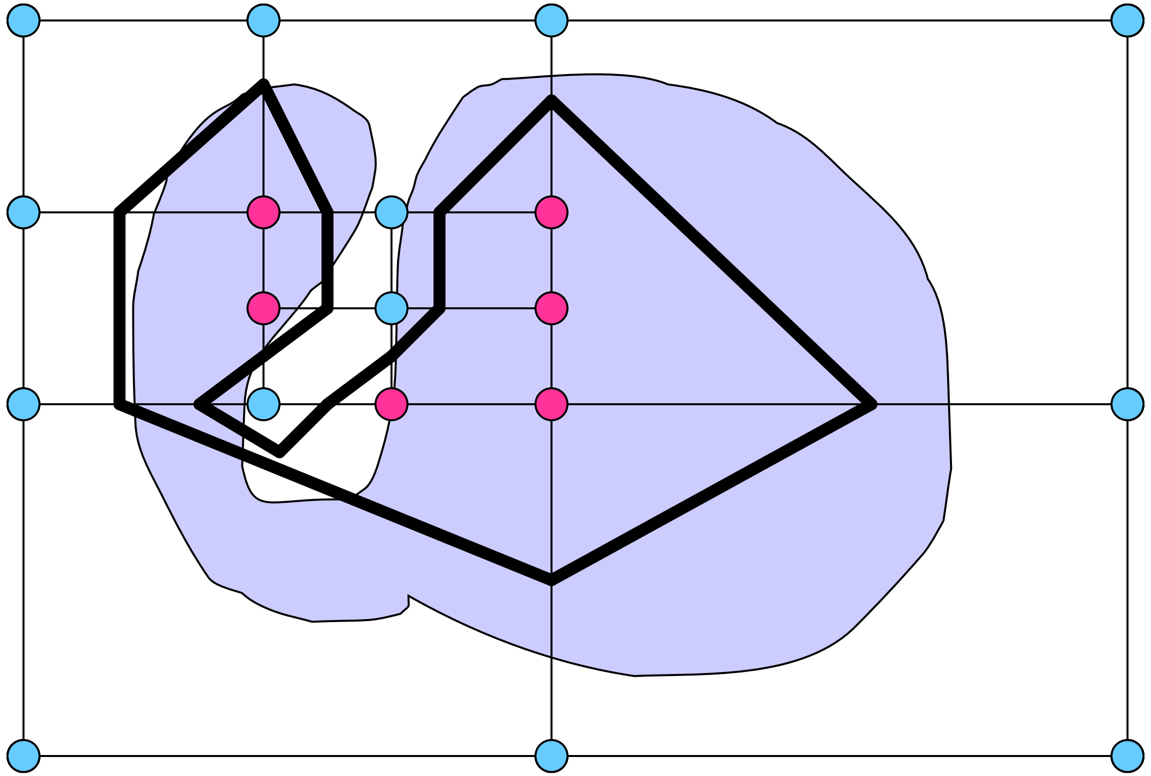
To display a set of octrees, convert the octrees into polygons.

- 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):

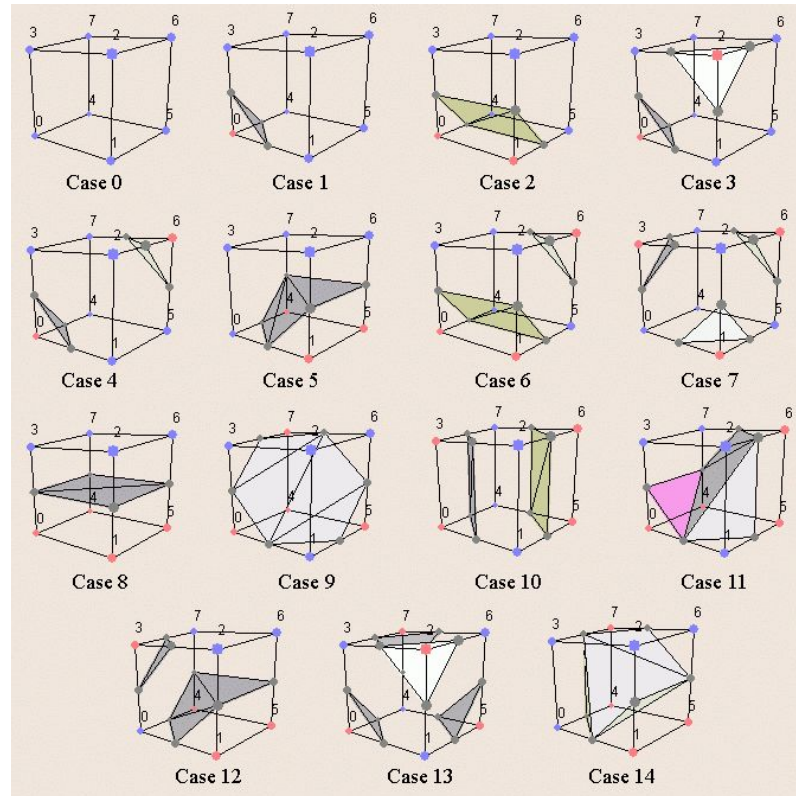


Polygonizing the surface

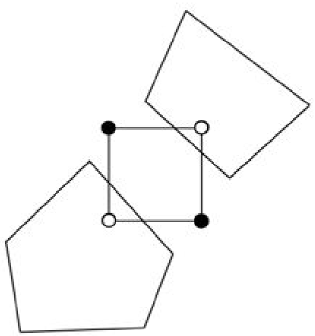
There are fifteen possible configurations (up to symmetry) of hot/cold vertices in the cube. →

- With rotations, that's 256 cases.

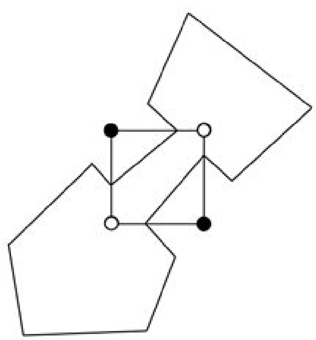
Beware: there are *ambiguous cases* in the polygonization which must be addressed separately. ↓



Images courtesy of [Diane Lingrand]



Break contour



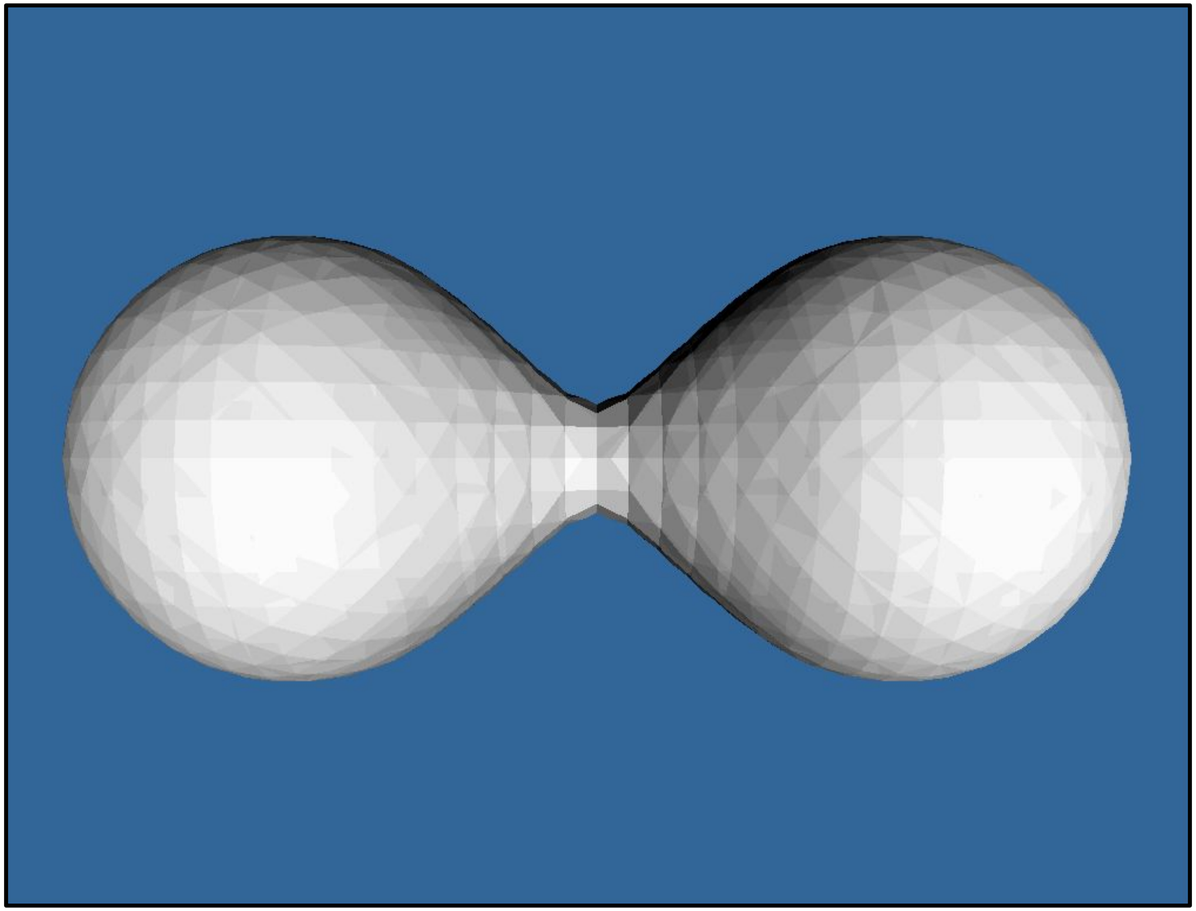
Join contour

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



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

Gaussian Curvature

http://en.wikipedia.org/wiki/Gaussian_curvature

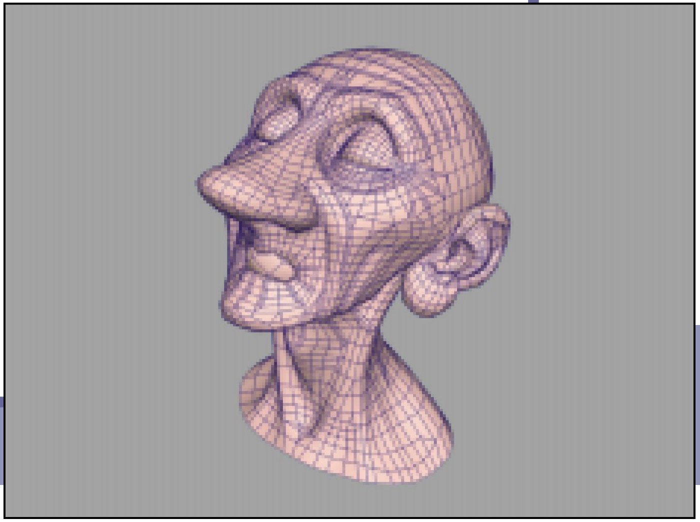
http://mathworld.wolfram.com/GaussianCurvature.html

The Poincaré Formula

http://mathworld.wolfram.com/PoincareFormula.html

Advanced Graphics

Subdivision Surfaces



Alex Benton, University of Cambridge – alex@bentonian.com

Supported in part by Google UK, Ltd

CAD, CAM, and a new motivation: *shiny things*

Expensive products are sleek and smooth.

→ Expensive products are C2 continuous.

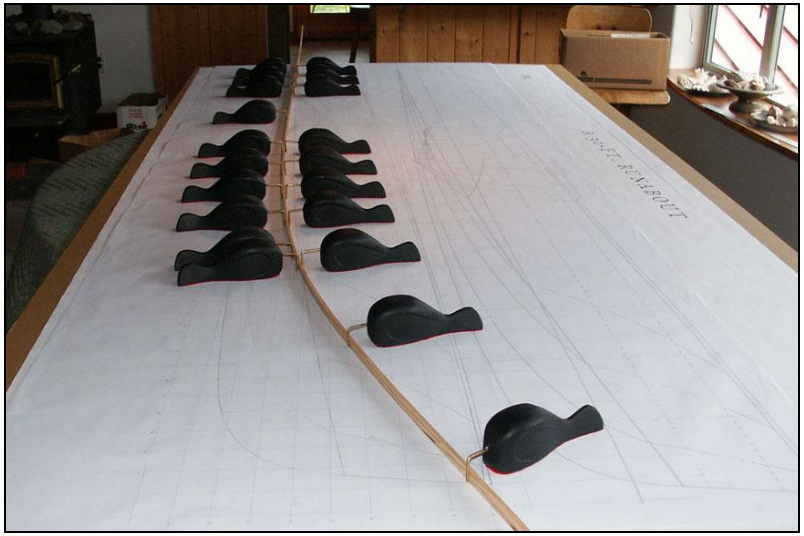
Shiny, but reflections are warped

Shiny, and reflections are perfect

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)

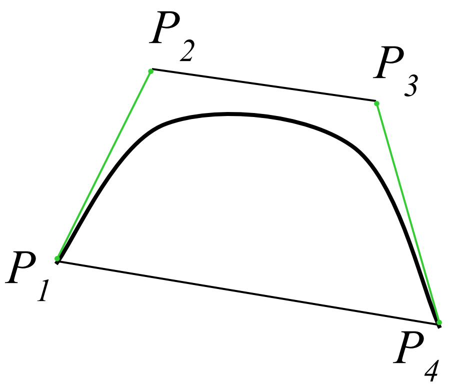
Bottom: http://www.pranos.com/boatsofwood/lofting%20ducks/lofting_ducks.htm

The drive for smooth CAD/CAM

- *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.

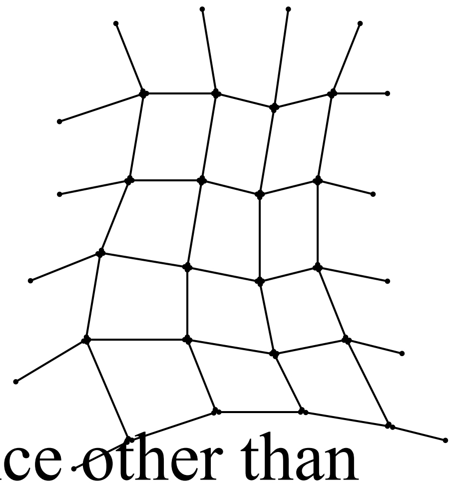
Beziers—a quick review

- 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 $P_1 \ldots P_4$.
- Beziers are a subset of a broader class of splines and surfaces called *NURBS*: *Non Uniform Rational B-Splines*.
- For decades, NURBS patches have been the bedrock of CAD/CAM.



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

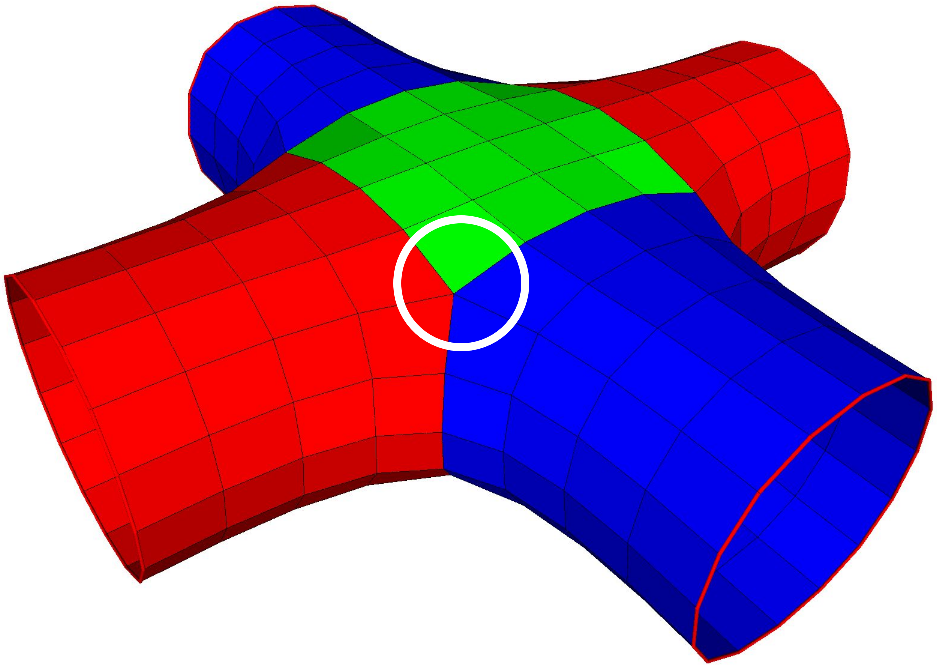
Bezier (NURBS) patches aren't the greatest



- NURBS patches are n×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.

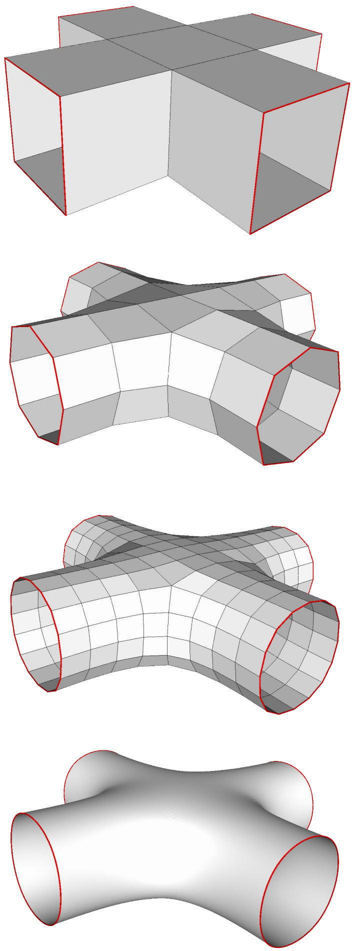
Subdivision surfaces

- 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*.



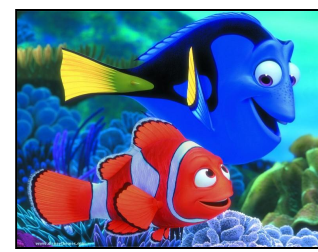
(Catmull-Clark in action)

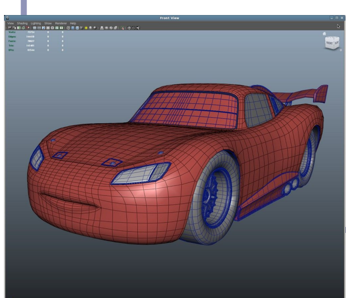
Subdivision surfaces – History

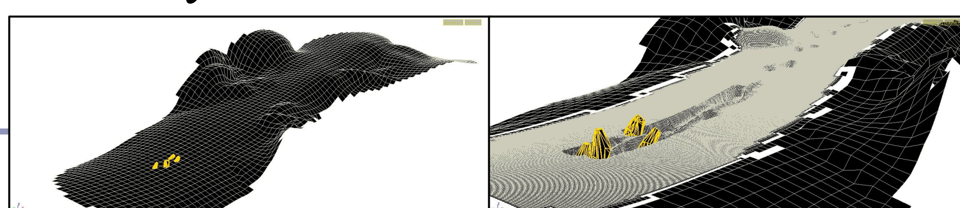
- de Rahm described a 2D (curve) subdivision scheme in 1947; rediscovered in 1974 by Chaikin
- Concept extended to 3D (surface) schemes by two separate groups during 1978:
  - Doo and Sabin found a biquadratic surface
  - Catmull and Clark found a bicubic surface
- Subsequent work in the 1980s (Loop,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

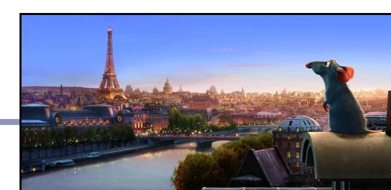
Subdivision surfaces and the movies

- 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 - creases and edges can be detailed by artists and regions of subdivision can themselves be dynamically subdivided



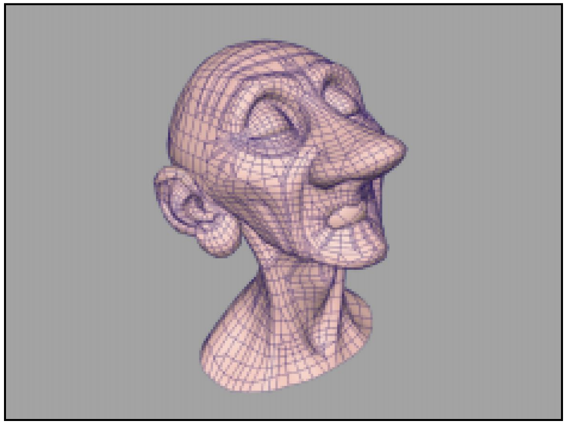






Useful terms

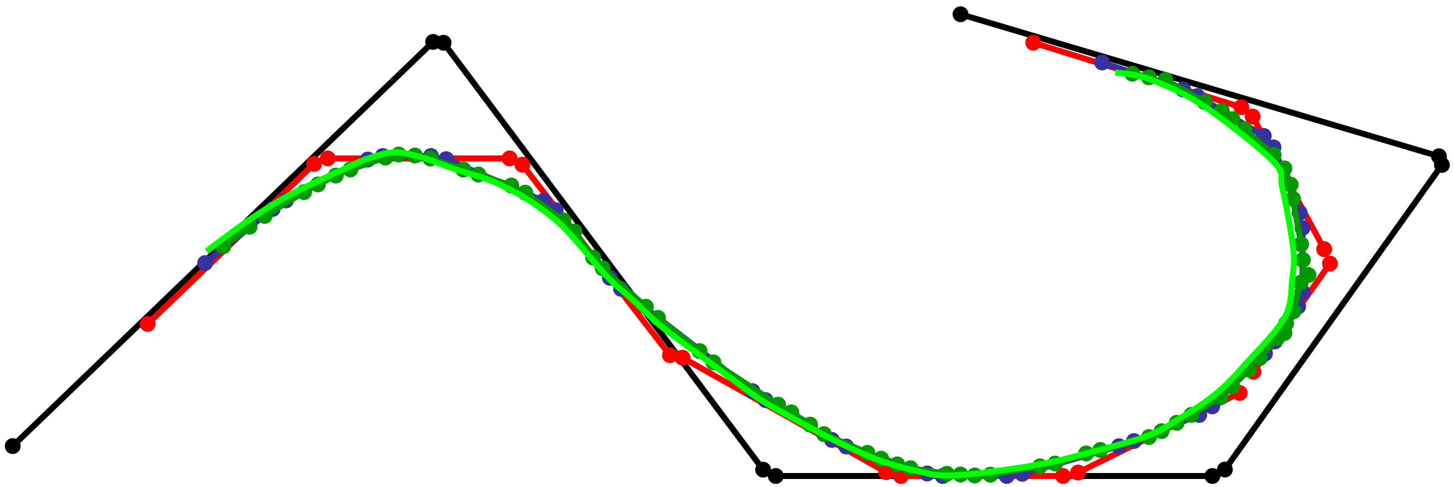
- A scheme which describes a 1D curve (even if that curve is travelling in 3D space, or higher) is called *univariate*, referring to the fact that the limit curve can be approximated by a polynomial in one variable (t).
- A scheme which describes a 2D surface is called *bivariate*, the limit surface 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

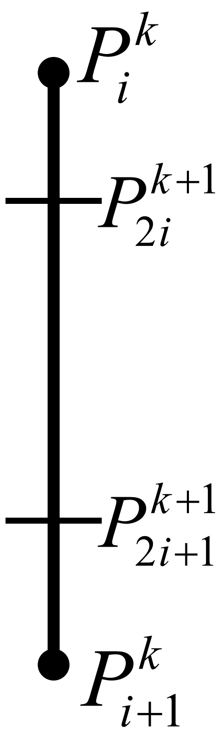
How it works

- 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:



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

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

…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:

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

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

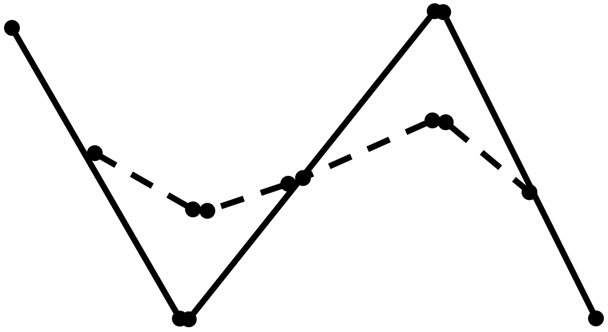
Consider the kernel

$$h=(1/8)[\ldots,0,0,1,4,6,4,1,0,0,\ldots]$$

You would read this as

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

$$P_{2i+1}^{k+1} = (\tfrac{1}{8})(4P_{i}^{k} + 4P_{i+1}^{k})$$



The limit curve is provably C2-continuous.

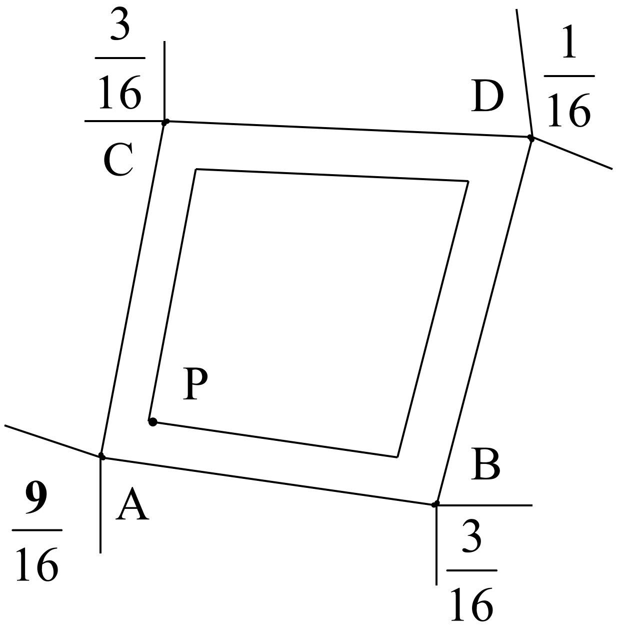
Making the jump to 3D: Doo-Sabin

Doo-Sabin takes Chaikin to 3D:

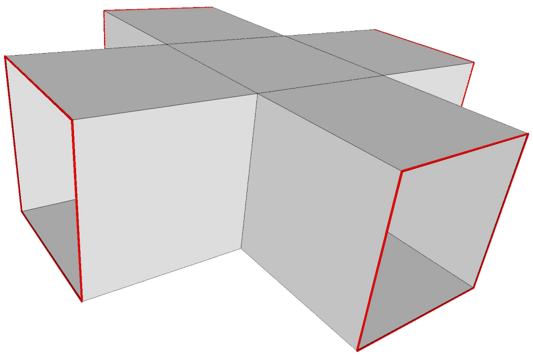
$$P = (9/16)\,A + (3/16)\,B + (3/16)\,C + (1/16)\,D$$

This replaces every old vertex with four new vertices.

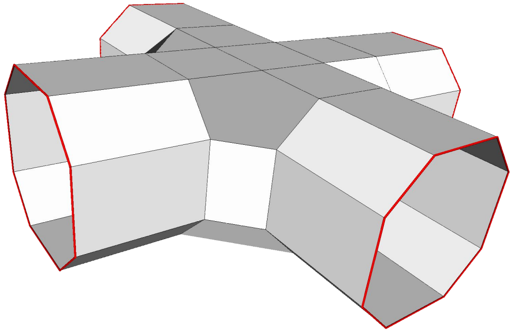
The limit surface is biquadratic, C1 continuous everywhere.



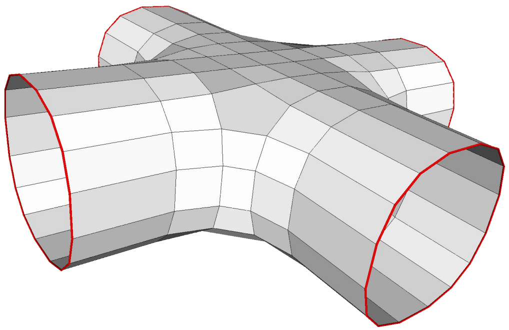
Doo-Sabin in action



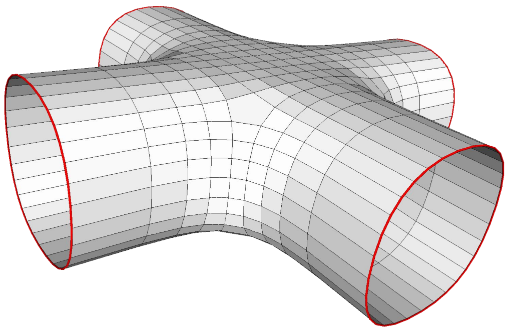
(0) 18 faces



(1) 54 faces



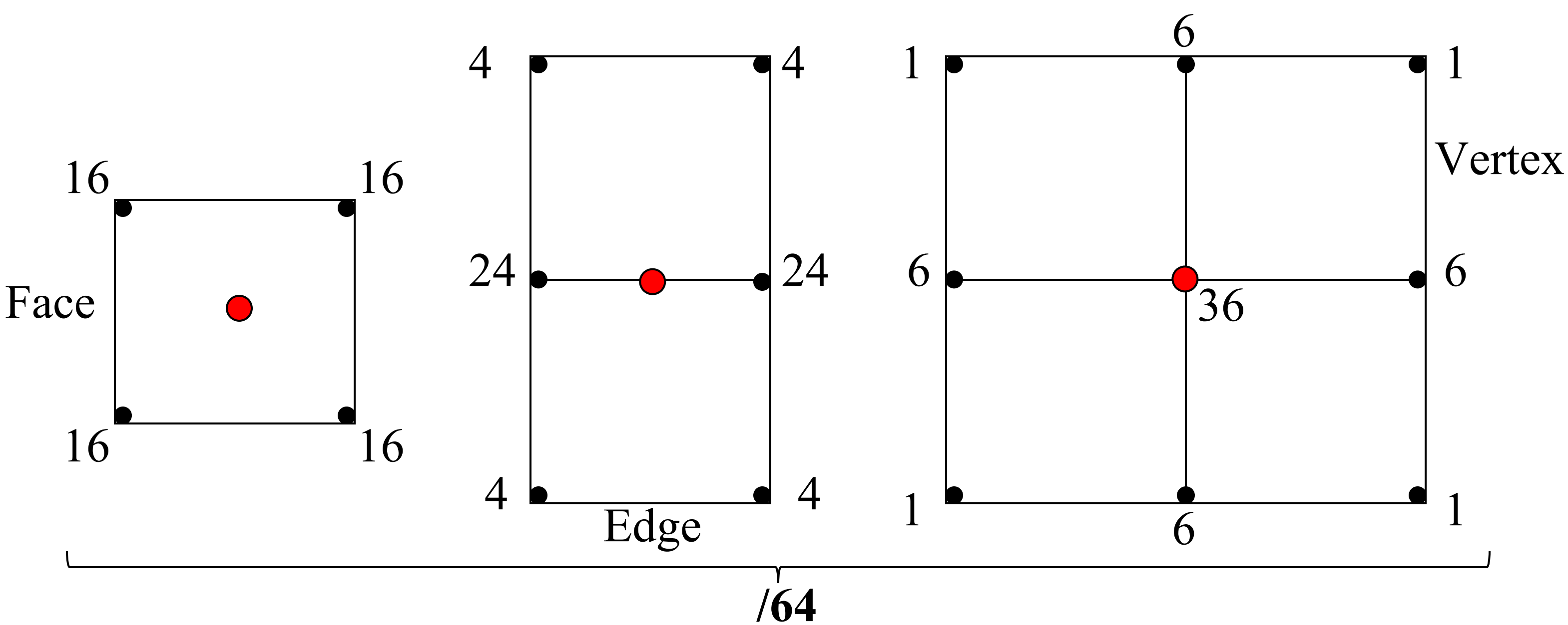
(2) 190 faces



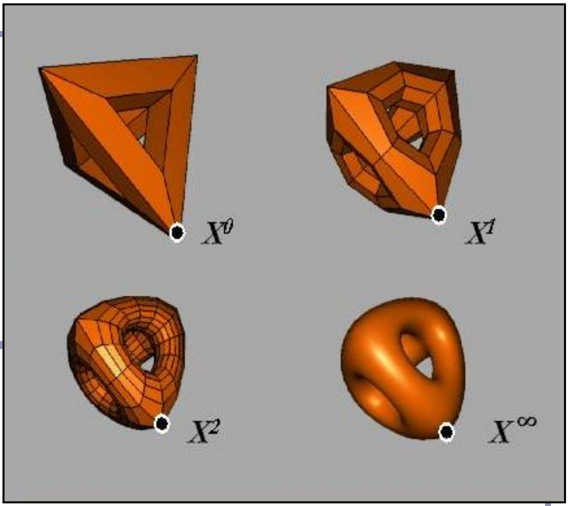
(3) 702 faces

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

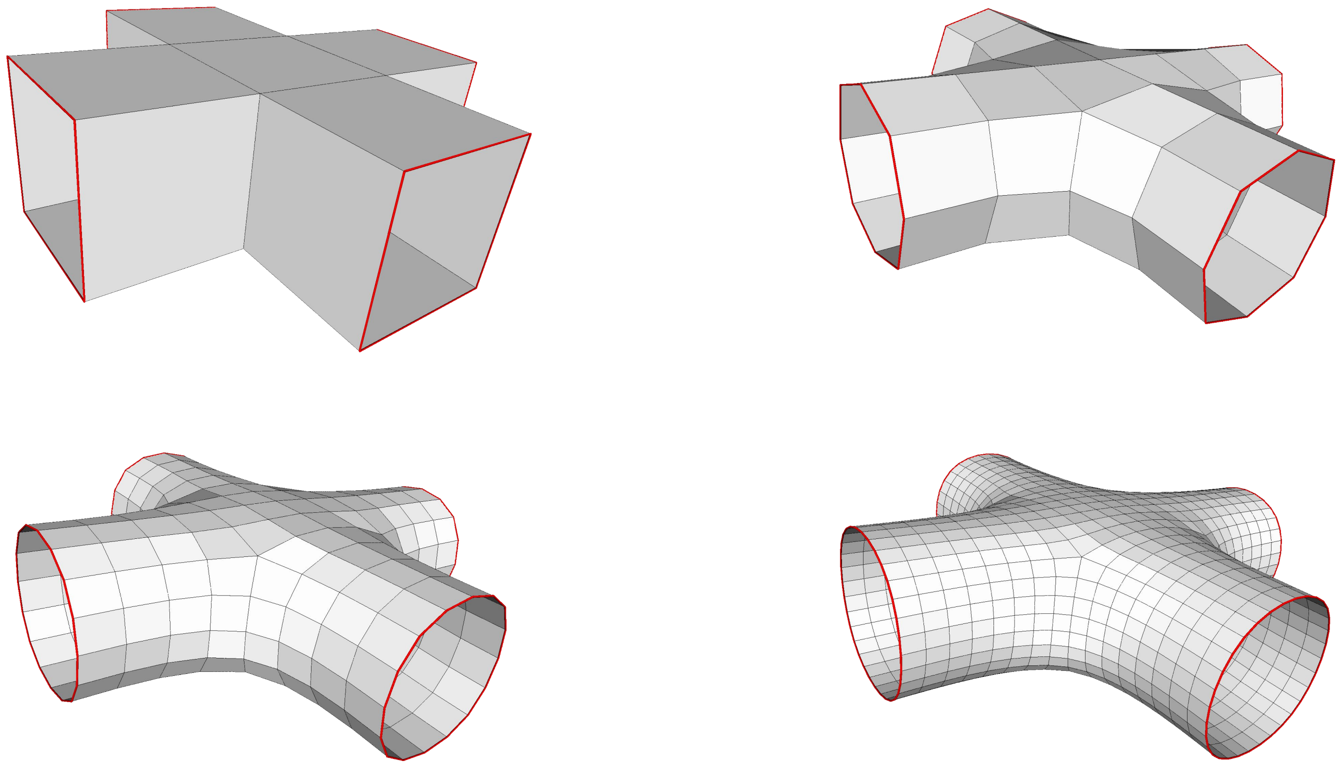


Getting tensor again:

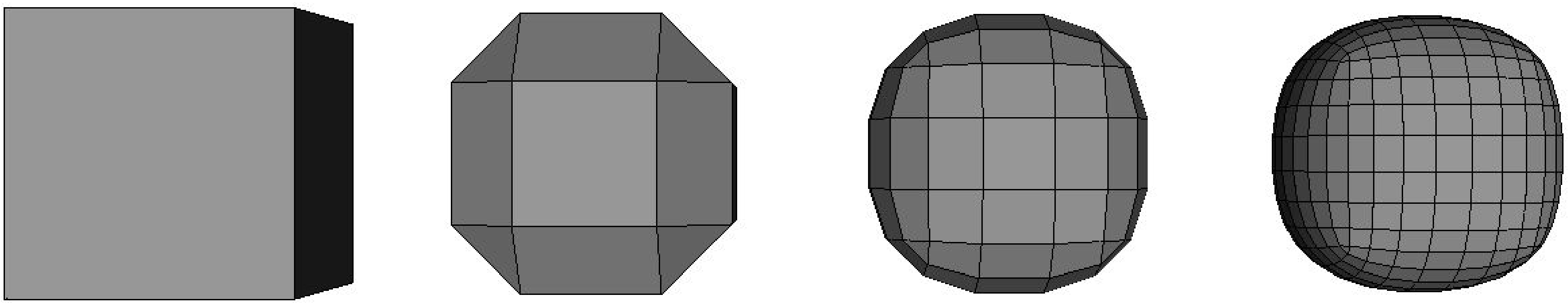
$$\frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix}\otimes\frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix}=\frac{1}{64}\begin{bmatrix}\boxed{1} & 4 & \boxed{6} & 4 & \boxed{1}\\ \boxed{4} & \boxed{16} & \boxed{24} & \boxed{16} & \boxed{4}\\ \boxed{6} & 24 & \boxed{36} & 24 & \boxed{6}\\ \boxed{4}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

Vertex rule　　Face rule　　Edge rule

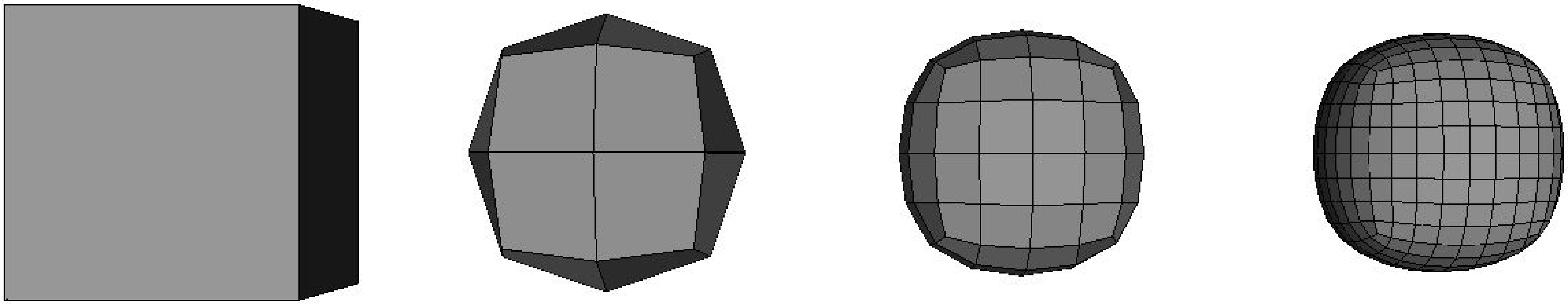
Catmull-Clark in action



Catmull-Clark vs Doo-Sabin



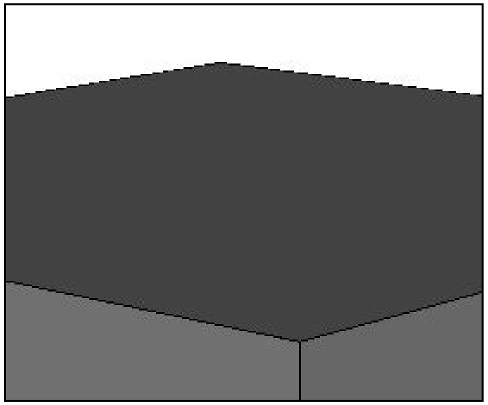
Doo-Sabin

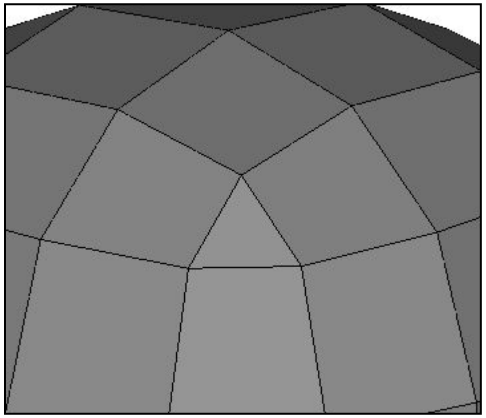


Catmull-Clark

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:
  $(4n\text{-}7) / 4n$
- Immediate neighbors in the one-ring:
  $3/2n^2$
- Interleaved neighbors in the one-ring:
  $1/4n^2$

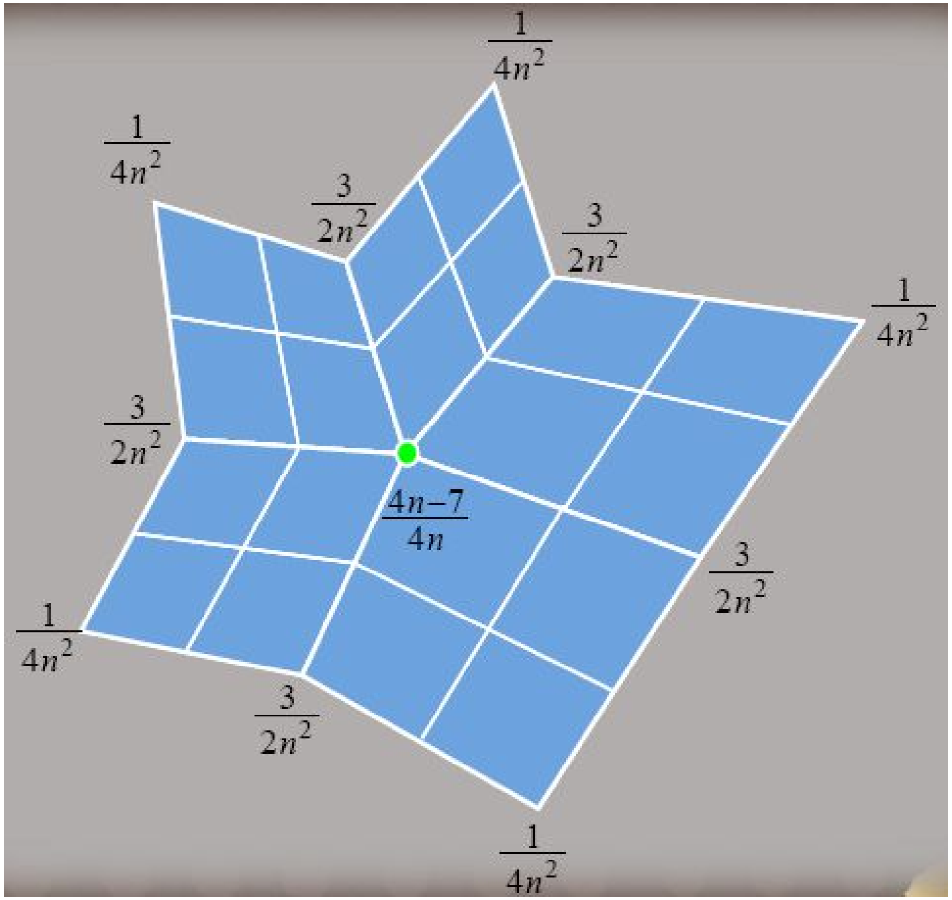
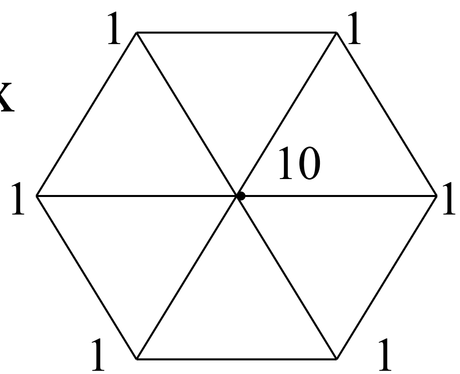


Image source: "*Next-Generation Rendering of Subdivision Surfaces*", Ignacio Castaño, SIGGRAPH 2008

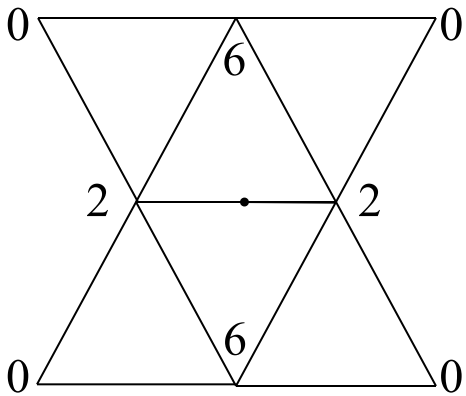
Schemes for simplicial (triangular) meshes

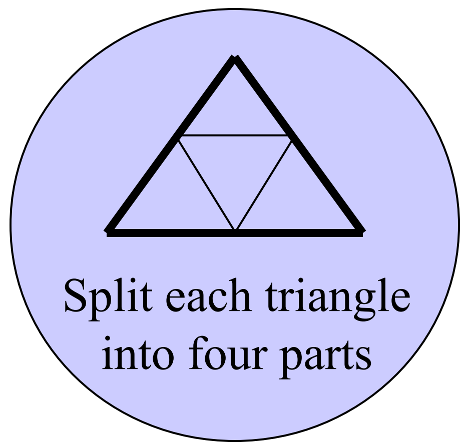
- *Loop* scheme:

Vertex



Edge

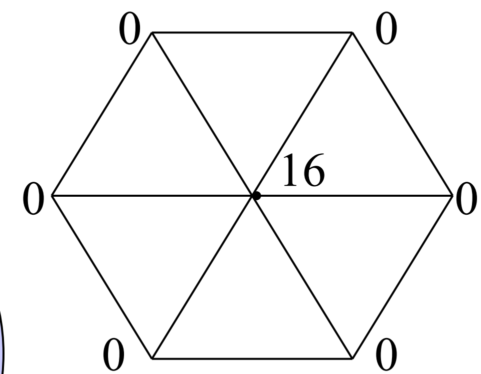




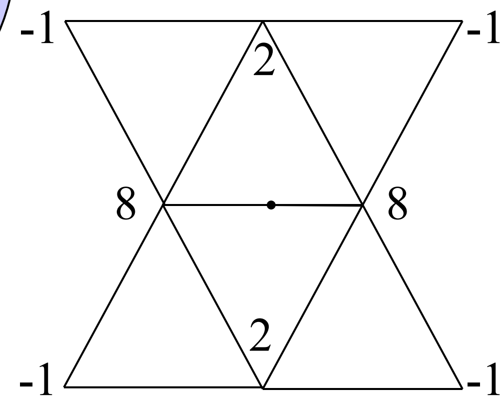
(All weights are /16)

- *Butterfly* scheme:

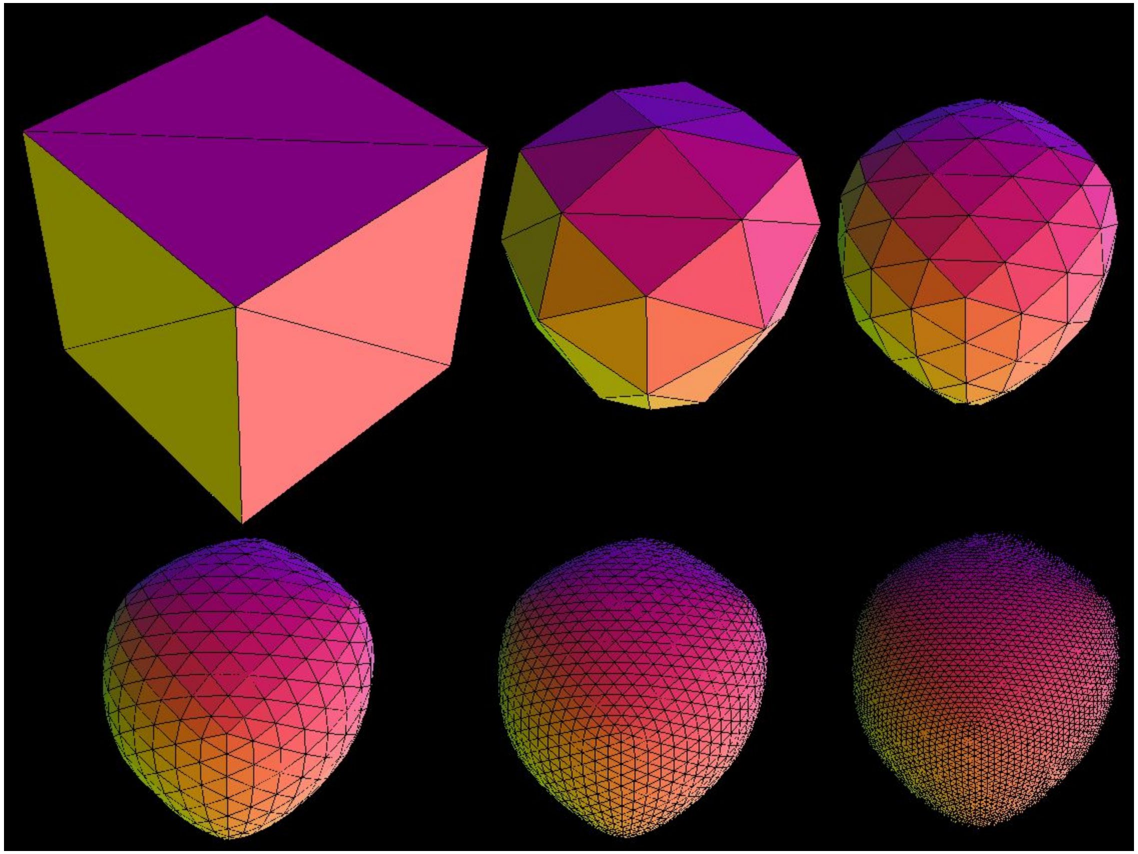
Vertex



Edge



Loop subdivision

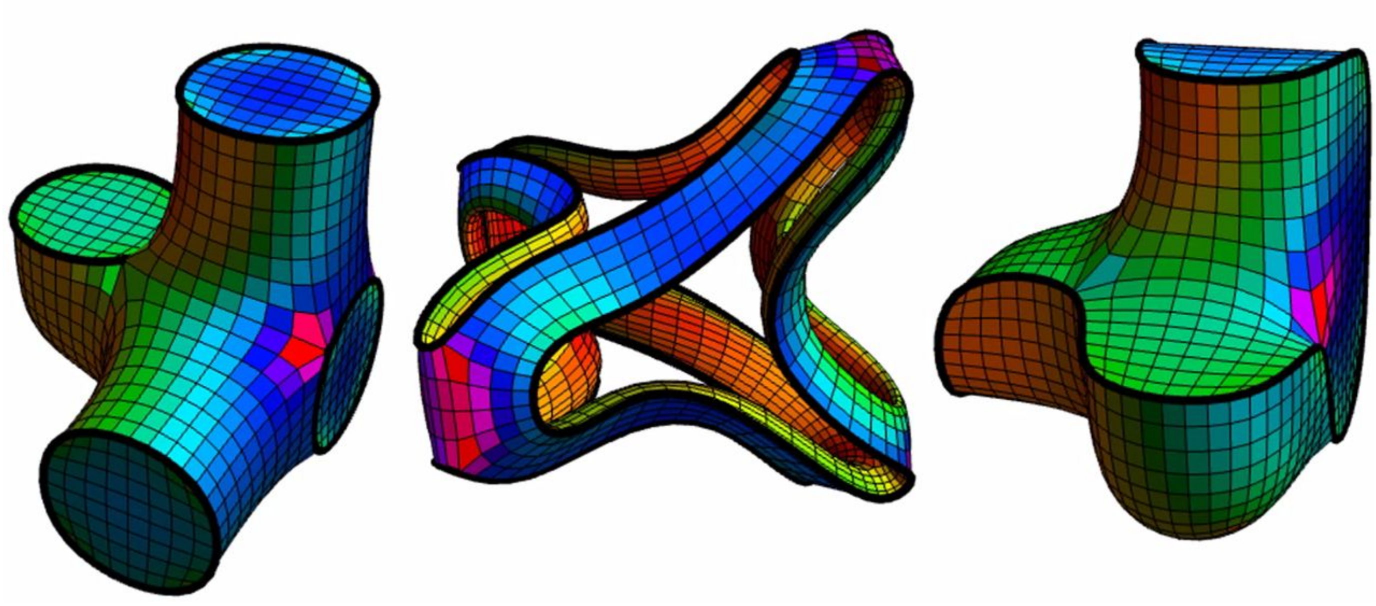


Loop subdivision in action. The asymmetry is due to the choice of face diagonals.

Image by Matt Fisher, http://www.its.caltech.edu/~matthewf/Chatter/Subdivision.html

Creases

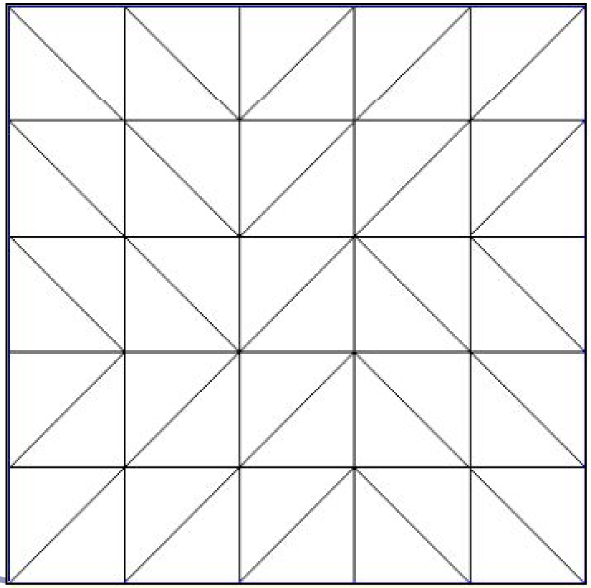
Extensions exist for most schemes to support *creases*, vertices and edges flagged for partial or hybrid subdivision.



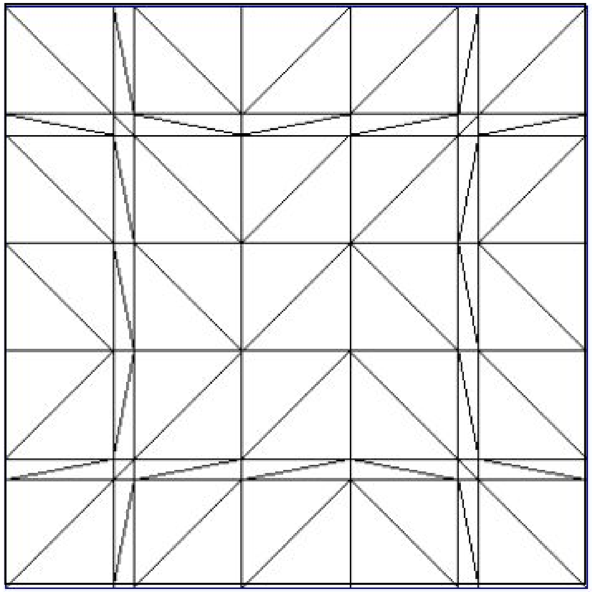
Still from "Volume Enclosed by Subdivision Surfaces with Sharp Creases" by Jan Hakenberg, Ulrich Reif, Scott Schaefer, Joe Warren http://vixra.org/pdf/1406.0060v1.pdf

Continuous level of detail

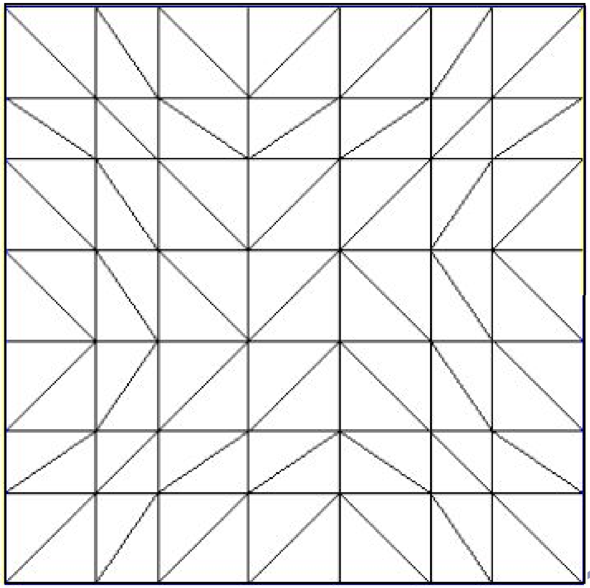
For live applications (e.g. games) can compute *continuous* level of detail, e.g. as a function of distance:



Level 5



Level 5.2



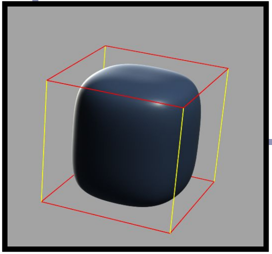
Level 5.8

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).



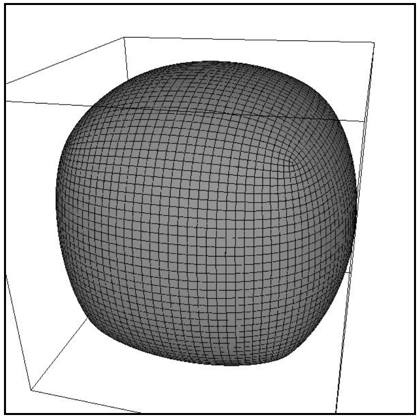
Splitting a subdivision surface

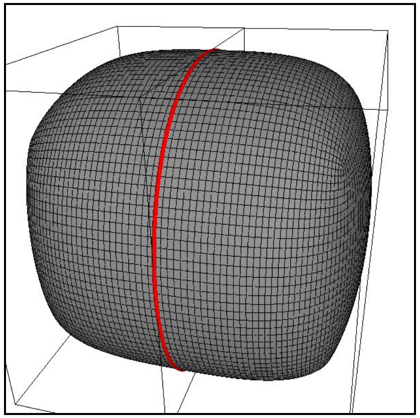
Many algorithms rely on subdividing a surface and examining the bounding boxes of smaller facets.

- 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.

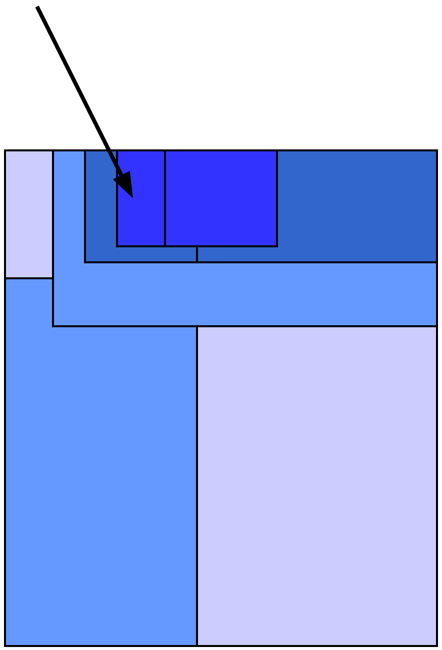




(Top) 5x Catmull-Clark subdivision of a cube
(Bottom) 5x Catmull-Clark subdivision of two halves of a cube; the limit surfaces are clearly different.

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

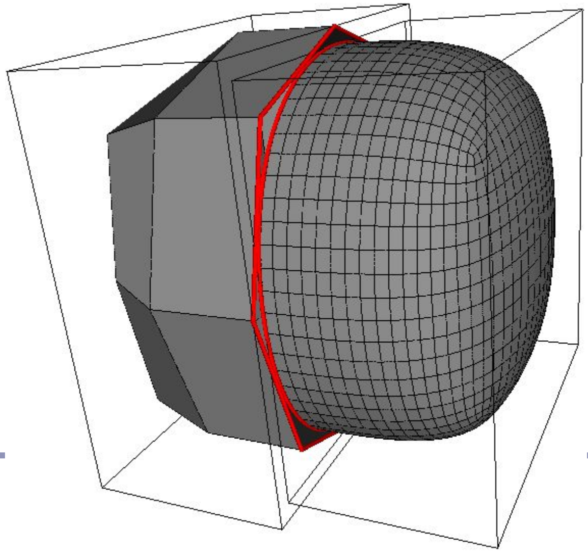
- The algorithm to render any subdivision surface is exactly the same as for Bezier curves:

  "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 using geometry (tesselation) 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

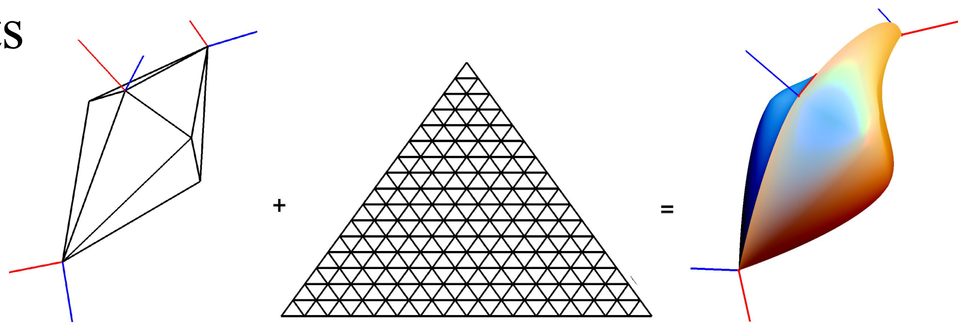


Figure from *Generic Mesh Renement on GPU,*
Tamy Boubekeur & Christophe Schlick (2005)
LaBRI INRIA CNRS University of Bordeaux, France

Subdivision Schemes—A partial list

- <u>Approximating</u>
  - 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

- <u>Interpolating</u>
  - 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

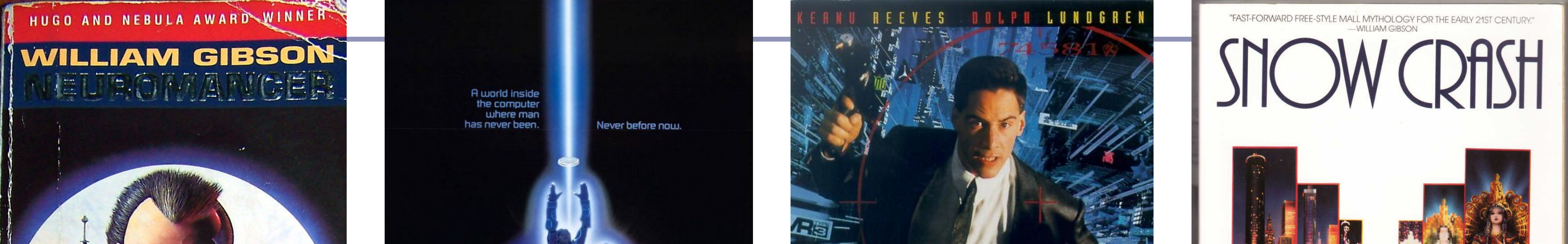
Ignacio Castano, "Next-Generation Rendering of Subdivision Surfaces." Siggraph '08, http://developer.nvidia.com/object/siggraph-2008-Subdiv.html

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

Virtual Reality

Immersion and Presence in digital realities

Alex Benton, University of Cambridge – alex@bentonian.com
Supported in part by Google UK, Ltd



"Cyberspace. A consensual hallucination experienced daily by billions of legitimate operators, in every nation, by children being taught mathematical concepts... A graphic representation of data abstracted from banks of every computer in the human system. Unthinkable complexity. Lines of light ranged in the nonspace of the mind, clusters and constellations of data. Like city lights, receding..."

— William Gibson, Neuromancer (1984)

~~*What is… the Matrix?*~~

What is Virtual Reality?

Immersion is the art and technology of surrounding the user with a virtual context, such that there's world above, below, and all around them.

Presence is the visceral reaction to a convincing immersion experience. It's when immersion is so good that the body reacts instinctively to the virtual world as though it's the real one.

When you turn your head to look up at the attacking enemy bombers, that's immersion; when you can't stop yourself from ducking as they roar by overhead, that's presence.

Top: HTC Vive (Image creduit: *Business Insider*)
Middle: *The Matrix* (1999)
Bottom: Google Daydream View (2016)

The "Sword of Damocles" (1968)

In 1968, Harvard Professor Ivan Sutherland, working with his student Bob Sproull, invented the world's first *head-mounted display*, or *HMD*.

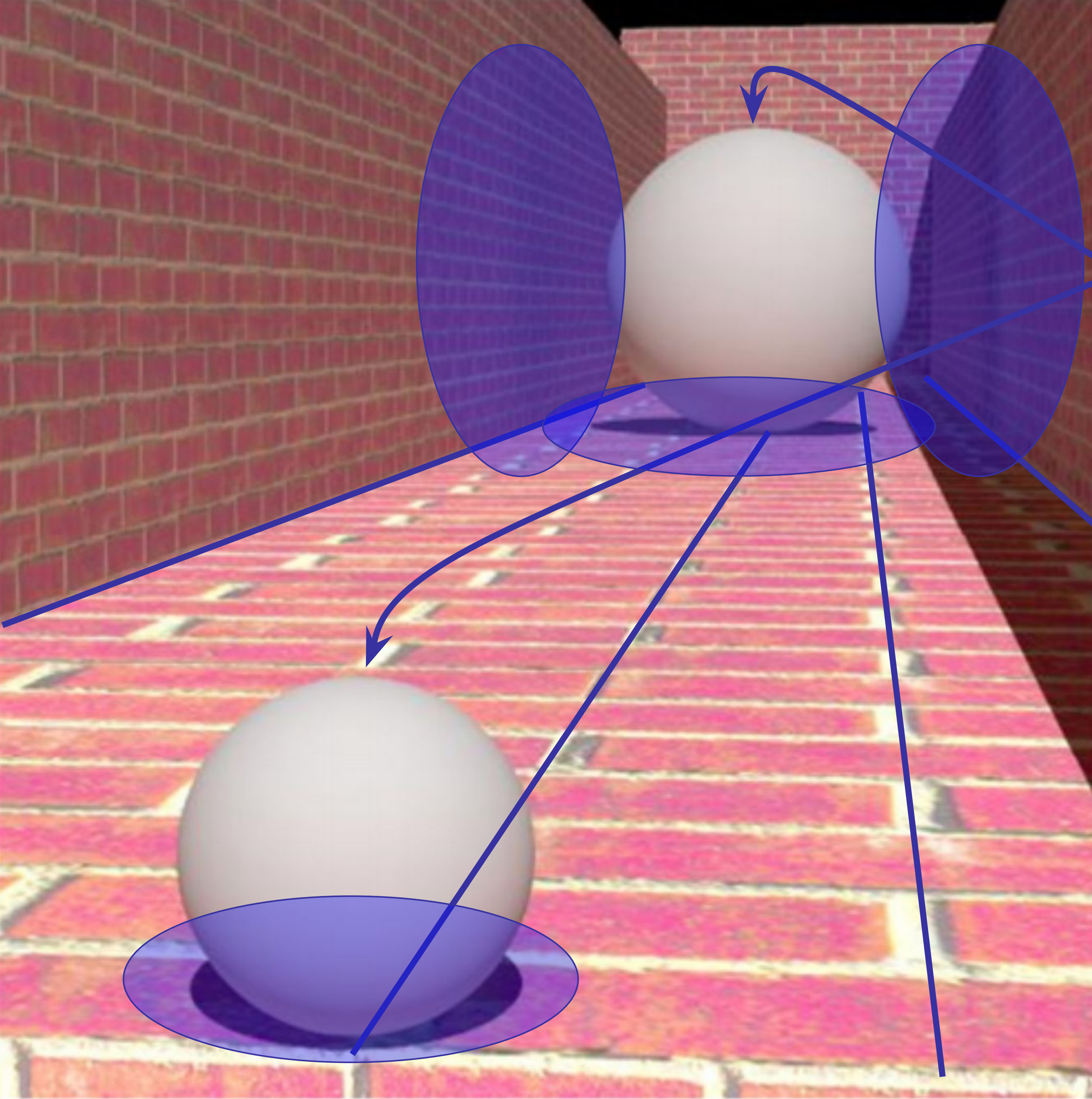
> *"The right way to think about computer graphics is that the screen is a window through which one looks into a virtual world. And the challenge is to makes the world look real, sound real, feel real and interact realistically."*
> *-Ivan Sutherland (1965)*

Distance and Vision

Our eyes and brain compute *depth cues* from many different signals:

- **Binocular vision ("stereopsis")**
  The brain merges two images into one with depth
  - Ocular convergence
  - Shadow stereopsis
- **Perspective**
  Distant things are smaller
- **Parallax motion** and **occlusion**
  Things moving relative to each other, or in front of each other, convey depth
- **Texture**, **lighting** and **shading**
  We see less detail far away; shade shows shape; distant objects are fainter
- **Relative size and position** and **connection to the ground**
  If we know an object's size we can derive distance, or the reverse; if an object is grounded, perspective on the ground anchors the object's distance

Image: Pere Borrell del Caso's *Escapando la Critica* ("*Escaping Criticism*") (1874)



Perspective

Occlusion

Shadows

Ambient shadows

Image credit: Scott Murray

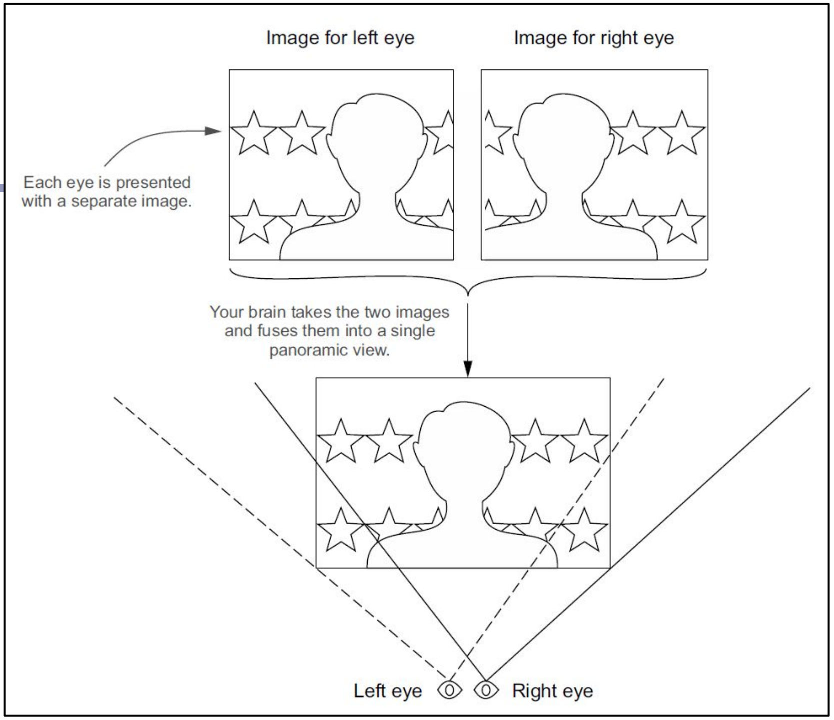
Murray, Boyaci, Kersten, *The representation of perceived angular size in human primary visual cortex*, Nature Neuroscience (2006)

Binocular display

Today's VR headsets work by presenting similar, but different, views to each eye

Each eye sees an image of the virtual scene from that eye's point of view in VR

This can be accomplished by rendering two views to one screen (Playstation VR, Google Daydream) or two dedicated displays (Oculus Rift, HTC Vive)



Top: Davis, Bryla, Benton, *Oculus Rift in Action* (2014)
Bottom: Oculus DK1 demo scene "Tuscanny"

Teardown of an Oculus Rift CV1

Teardown of an Oculus Rift CV1 showing details of lenses and displays
https://www.ifixit.com/Teardown/Oculus+Rift+CV1+Teardown/60612

Accounting for lens effects

Lenses bend light: the lenses in the VR headset warp the image on the screen, creating a *pincushion distortion*.

This is countered by first introducing a *barrel distortion* in the GPU shader used to render the image.

The barrel-distorted image stretches back to full size when it's seen through the headset lenses.

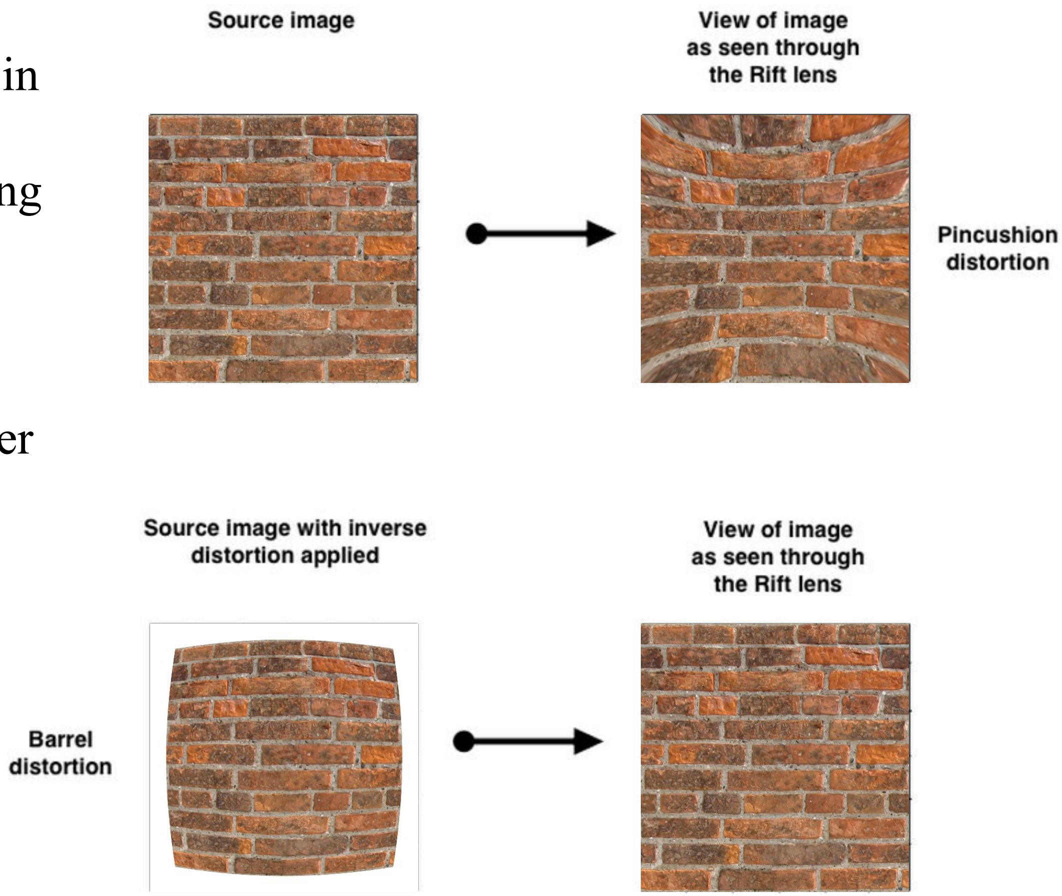
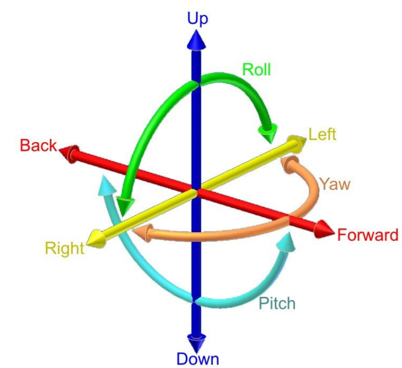


Image credit: Davis, Bryla, Benton, *Oculus Rift in Action* (2014)

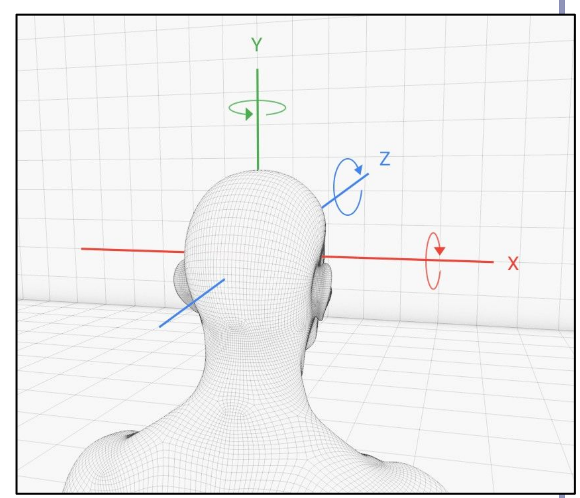
Sensors

Accelerometer and electromagnetic sensors in the headset track the user's *orientation* and *acceleration*. VR software converts these values to a basis which transforms the scene.



Ex: [WebVR API](#):

```
interface VRPose {
  readonly attribute Float32Array? position;
  readonly attribute Float32Array? linearVelocity;
  readonly attribute Float32Array? linearAcceleration;

  readonly attribute Float32Array? orientation;
  readonly attribute Float32Array? angularVelocity;
  readonly attribute Float32Array? angularAcceleration;
};
```



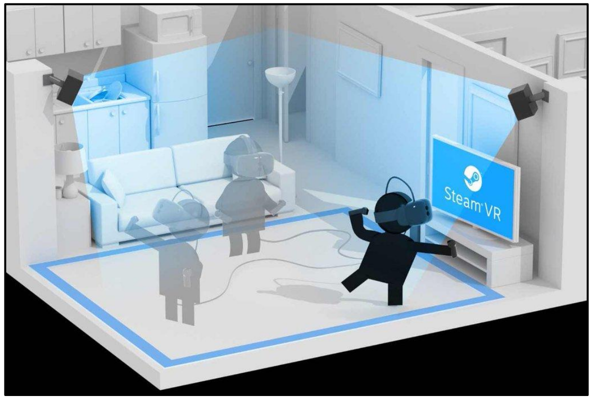
Top: 6DoF (6 degrees of freedom) - Wikipedia
Bottom: Roll (Z), Pitch (X) and Yaw (Y) - Google Design

Sensor fusion

Problem: Even the best accelerometer can't detect all motion. Over a few seconds, position will drift.

Solution: Advanced headsets also track position with separate hardware on the user's desk or walls.

- Oculus Rift: "Constellation", a desk-based IR camera, tracks a pattern of IR LEDs on the headset
- HTC Vive: "base station" units track user in room
- Playstation VR: LEDs captured by PS camera



The goal is to respond in a handful of milliseconds to any change in the user's position or orientation, to preserve presence.

Top: Constellation through an IR-enabled camera (image credit: ifixit.com)
Bottom: HTC Vive room setup showing two base stations (image credit: HTC)

Sensors - how fast is fast?

- To preserve presence, the rendered image must respond to changes in head pose faster than the user can perceive
- That's believed to be about 20ms, so no HMD can have a framerate below 50hz
- Most headset display hardware has a higher framerate
  - The Rift CV1 is locked at 90hz
  - Rift software **must** exceed that framerate
  - Failure to do so causes 'judder' as frames are lost
  - Judder leads to nausea, nausea leads to hate, hate leads to the dark side

Dealing with latency: sensor prediction

A key immersion improvement is to *predict the future basis*. This allows software to optimize rendering.

- At time t, head pos = X, head velocity = V, head acceleration = A
- Human heads do not accelerate very fast
- Rendering a single frame takes dt milliseconds
- At $t + dt$, we can predict pos = $X + V dt + \frac{1}{2} A dt^2$
- By starting to render the world from the user's predicted head position, when rendering is complete, it aligns with where there head is by then (hopefully).

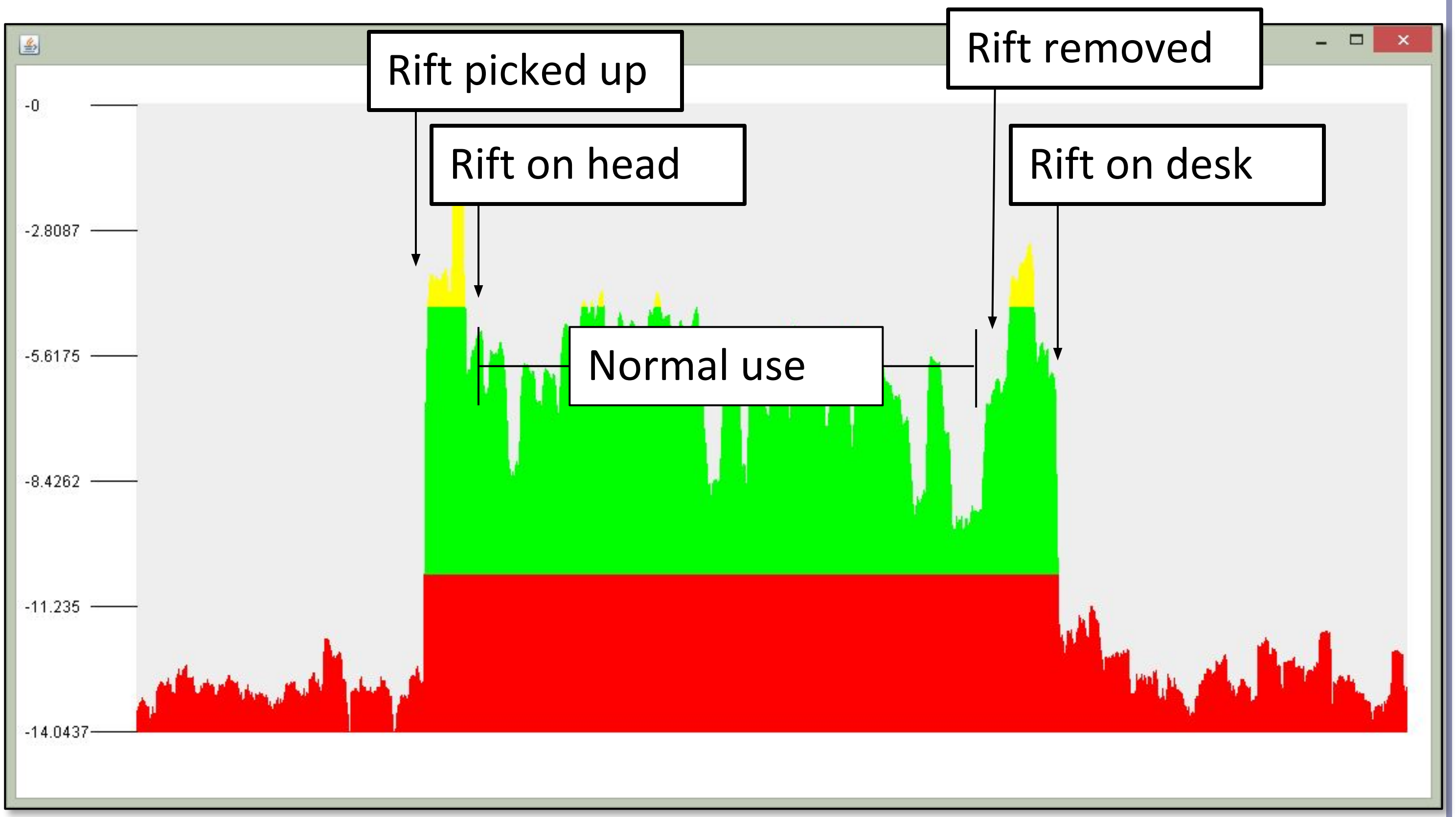
Ex: The WebVR API returns predicted pose by default

Advanced sensor tricks: detecting when the headset is in use

This graph shows the `noise' of an Oculus Rift DK2's position when it's on a desk or on a user's head.

Desk: *(approx.)* log(σ) < -10.5

Head: *(approx.)* log(σ) < -4.5

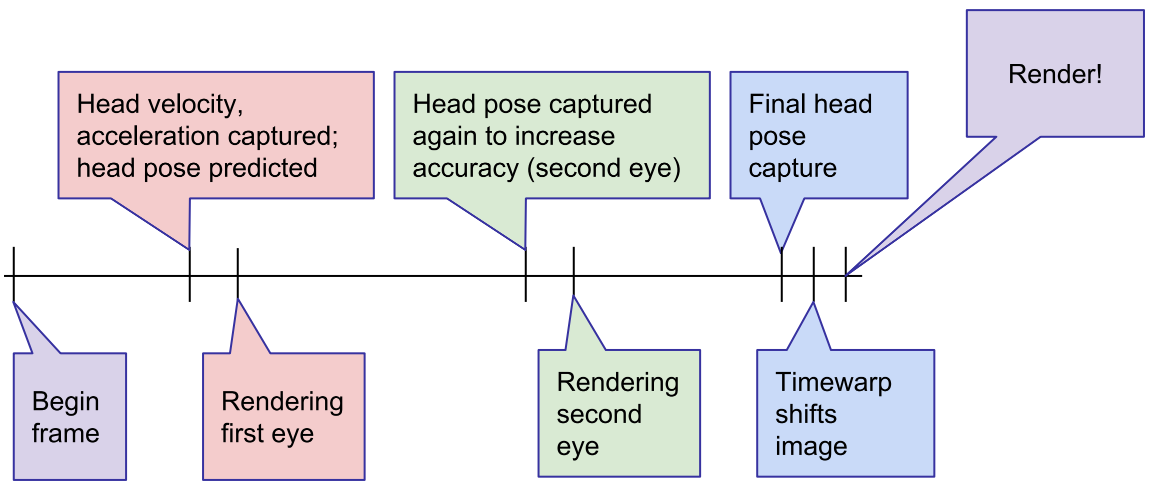


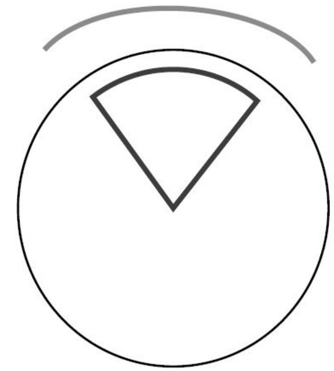
Davis, Bryla, Benton, *Pitfalls and Perils of VR: How to Avoid Them* (2014) http://rifty-business.blogspot.co.uk/2014/09/slides-from-our-pax-dev-2014-talk.html

Dealing with latency: ‘timewarp’

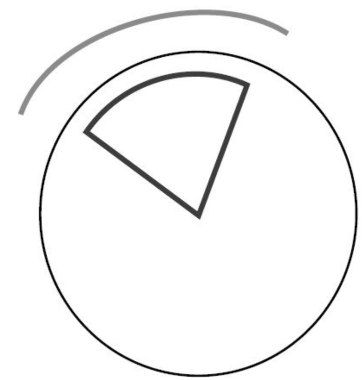
Another technique to deal with lost frames is *asynchronous timewarp*.

- Headset pose is fetched immediately before frame display and is used to shift the frame on the display to compensate for ill-predicted head motion





Original head pose



With timewarp applied

Image credit: Davis, Bryla, Benton, *Oculus Rift in Action* (2014)

Developing for VR

Dedicated SDKs

- HTC Vive
- Oculus Rift SDK
  - C++
  - Bindingsfor Python, Java
- Google Daydream SDK
  - Android, iOS and Unity
- Playstation VR
  - Playstation dev kit

General-purpose SDKs

- WebGL - three.js
- WebVR API

Higher-level game development

- Unity VR

“Sim sickness”

The Problem:

1. Your body says, “Ah, we’re sitting still.”
2. Your eyes say, “No, we’re moving! It’s exciting!”
3. Your body says, “Woah, my inputs disagree! I must have eaten some bad mushrooms. Better get rid of them!”
4. Antisocial behavior ensues

The causes of *simulation sickness* (like motion sickness, but in reverse) are many. Severity varies between individuals; underlying causes are poorly understood.

Reducing sim sickness

The cardinal rule of VR:

The user is in control of the camera.

1. **Never** take head-tracking control away from the user
2. Head-tracking **must match the user's motion**
3. **Avoid moving the user** without direct interaction
4. If you must move the user, do so in a way that **doesn't break presence**

How can you mitigate sim sickness?

Design your UI to reduce illness

- Never mess with the field of view
- Don’t use head bob
- Don’t knock the user around
- Offer multiple forms of camera control
    - Look direction
    - Mouse + keyboard
    - Gamepad
- Try to match in-world character height and *IPD* (*inter-pupilary distance*) to that of the user
- Where possible, give the user a stable in-world reference frame that moves with them, like a vehicle or cockpit

Hawken, by Meteor Entertainment (2014)

Further ways to reduce sim sickness

Design your VR world to reduce illness

- Limit sidestepping, backstepping, turning; never force the user to spin
- If on foot, move at real-world speeds (1.4m/s walk, 3m/s run)
- Don’t use stairs, use ramps
- Design to scale--IPD and character height should match world scale
- Keep the horizon line consistent, static and constant
- Avoid very large moving objects which take up most of the field of view
- Use darker textures
- Avoid flickering, flashing, or high color contrasts
- Don’t put content where they have to roll their eyes to see it
- If possible, build breaks into your VR experience
- If possible, give the user an avatar; if possible, the avatar body should react to user motion, to give an illusion of proprioception

Classic user interfaces in 3D

Many classic UI paradigms *will not work* if you recreate them in VR

- UI locked to sides or corners of the screen will be distorted by lenses and harder to see
- Side and corner positions force the user to roll their eyes
- Floating 3D dialogs create a virtual plane within a virtual world, breaking presence
- Modal dialogs 'pause' the world
- Small text is much harder to read in VR

Top: *EVE Online* (2003)
Bottom: *Team Fortress* (2007)

In-world UIs are evolving

Deus Ex Human Revolution (2011)

Deus Ex Mankind Divided (2016)

Increasingly, UI elements are being integrated into the virtual world

The best virtual UI is in-world UI

Top left: Call of Duty: Black Ops (2010)
Bottom left: Crysis 3 (2013)

Top right: Halo 4 (2012)
Bottom right: Batman: Arkham Knight (2015)

Strike Suit Zero (2013)

Elite: Dangerous (2014)

Storytelling in games

The visual language of games is often the language of movies

- Cutscenes
- Angle / reverse-angle conversations
- Voiceover narration
- Pans
- Dissolves
- Zooms...

Call of Duty: Modern Warfare 3 (2012)
The player's helicopter has been shot down; they emerge into gameplay, transitioning smoothly from passive to active.

In VR, storytelling by moving the camera will not work well because the user *is* the camera.

"It's a new communications medium. What is necessary is to develop a grammar and syntax. It's like film. When film was invented, no one knew how to use it. But gradually, a visual grammar was developed. Filmgoers began to understand how the grammar was used to communicate certain things. We have to do the same thing with this.“

Neal Stephenson, *Interface,* 1994

In-game video content

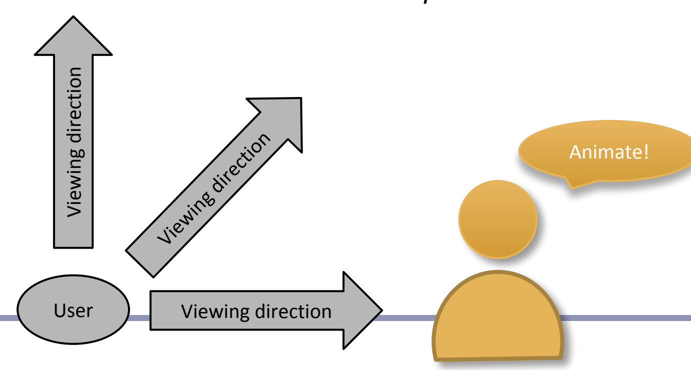
Your virtual world may have screens of its own. If it does, use them: they're perfect for prerecorded 2D content.

Drawing the user's attention

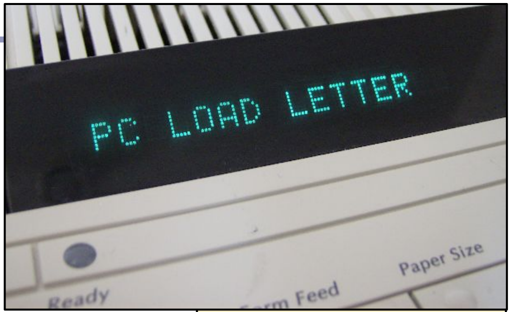
When presenting dramatic content in VR, you risk the user looking away at a key moment.

- Use audio cues, movement or changing lighting or color to draw focus
- Use other characters in the scene; when they all turn to look at something, the player will too
- Design the scene to direct the eye
- Remember that in VR, you know when key content is in the viewing frustum

The Emperor's New Groove (2000)



Advice for a good UI



An unhelpful error message

Always display relevant state—Primary application state should be visible to the user. For an FPS shoot-em-up, this means showing variables like ammo count and health. Combine audio and video for key cues such as player injury.

Use familiar context and imagery—Don't make your users learn specialized terms so they can use your app. If you're writing a surgery interface for medical training, don't force medical students to learn about virtual cameras and FOVs.

Support undo/redo—Don't penalize your users for clicking the wrong thing. Make undoing recent actions a primary user interface mode whenever feasible.

Design to prevent error—If you want users to enter a value between 1 and 10 in a box, don't ask them to type; they could type 42. Give them a slider instead.

Build shortcuts for expert users—The feeling that you're becoming an expert in a system often comes from learning its shortcuts. Make sure that you offer combos and shortcuts that your users can learn—but don't require them.

Don't require expert understanding—Visually indicate when an action can be performed, and provide useful data if the action will need context. If a jet fighter pilot can drop a bomb, then somewhere on the UI should be a little indicator of the number of bombs remaining. That tells players that bombs are an option and how many they've got. If it takes a key press to drop the bomb, show that key on the UI.

Keep it simple—Don't overwhelm your users with useless information; don't compete with yourself for space on the screen. Always keep your UI simple. "If you can't explain it to a six-year-old, you don't understand it yourself" (attributed to Albert Einstein).

Make error messages meaningful—Don't force users to look up arcane error codes. If something goes wrong, take the time to clearly say what, and more important, what the user should do about it.

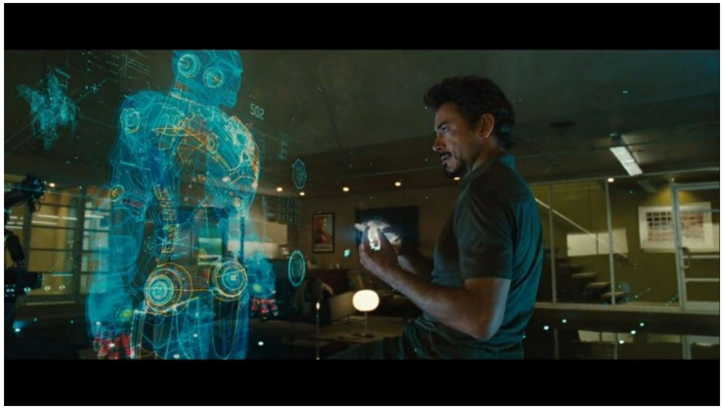
Abridged from *Usability Engineering* by Jakob Nielsen (Morgan Kaufmann, 1993)

Gesticular interfaces

Hollywood has been training us for a while now to use *gesticular user interfaces*.

A gesticular interface uses pre-set, intuitive hand and body gestures to control virtual representations of material data.

Many hand position capture devices are in development.



Johnny Mnemonic (1995)

Marvel's Agents of S.H.I.E.L.D. (2013) S01 E13

References

Developing in VR

- *Fundamentals of Computer Graphics*, by P. Shirley, M. Ashikhmin, and S. Marschner (A. K. Peters/CRC Press, 2009)
- *Computer Graphics: Principles and Practice*, by J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes (Addison-Wesley Professional, 2013)
- *Oculus Rift in Action*, by Davis, Bryla and Benton (2014)
- *Oculus Best Practices Guide* - developer.oculus.com/documentation

Motion sickness/simulator sickness

- *Textbook of Maritime Medicine*, by the Norwegian Centre for Maritime Medicine (2013). See chapter 20, "Motion Sickness" (textbook.ncmm.no)
- *Validating an Efficient Method to Quantify Motion Sickness*, by B. Keshavarz and H. Hecht (2011). *Human Factors: The Journal of the Human Factors and Ergonomics Society* 53.4: 415–26.
- *Simulator Sickness Questionnaire*, by R. S. Kennedy, N. E. Lane, K. S. Berbaum, and M. G. Lilienthal (1993). *The International Journal of Aviation Psychology* 3(3): 203–20.
- Motion Sickness Susceptibility Questionnaire Revised and Its Relationship to Other Forms of Sickness, by J. F. Golding (1998). *Brain Research Bulletin*, 47(5): 507–16.

UI design for VR

- 3D User Interfaces: New Directions and Perspectives, by D. A. Bowman, S. Coquillart, B. Froehlich, M. Hirose…and W. Stuerzlinger. (2008). *IEEE Computer Graphics and Applications* 28(6): 20–36.
- Design and Evaluation of Mouse Cursors in a Stereoscopic Desktop Environment, by L. Schemali and E. Eisemann (2014). *3D User Interfaces (3DUI), 2014 IEEE Symposium* (pp. 67-70). IEEE. Recorded talk is available at vimeo.com/91489021
- Developing Virtual Reality Games and Experiences— www.gdcvault.com/play/1020714. Presented at GDC 2014.
- Egocentric Object Manipulation in Virtual Environments: Empirical Evaluation of Interaction Techniques, by I. Poupyrev, S. Weghorst, M. Billinghurst, and T. Ichikawa (1998). *Computer Graphics Forum*, 17(3): 41–52.
- Kinect Hand Detection, by G. Gallagher—video.mit.edu/watch/kinect-hand-detection-12073
- *Make It So: Interaction Design Lessons from Science Fiction*, by N. Shedroff and C. Noessel (Rosenfeld Media, 2012)
- Lessons learned porting *Team Fortress 2* to virtual reality—media.steampowered.com/apps/valve/2013/Team_Fortress_in_VR_GDC.pdf
- Pointing at 3D Target Projections with One-Eyed and Stereo Cursors, by R. J. Teather and W. Stuerzlinger. (2013). *ACM Conference on Human Factors in Computing Systems*: 159–68.
- Pointing to the future of UI, by J. Underkoffler (2010). Talk given at TED. www.ted.com/talks/john_underkoffler_drive_3d_data_with_a_gesture
- Selection Using a One-Eyed Cursor in a Fish Tank VR Environment, by C. Ware and K. Lowther. (1997). *ACM Transactions on Computer-Human Interaction Journal,* 4(4): 309–22.
- *Usability Engineering*, by J. Nielsen (Morgan Kaufmann, 1993)