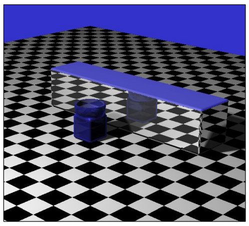
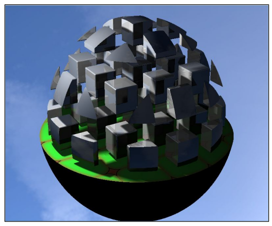
Further Graphics





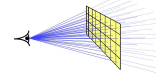
Ray Marching and Signed Distance Fields

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

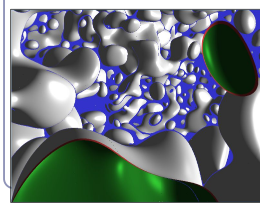
Supported in part by Google UK, Ltd

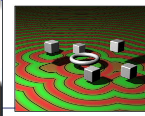
GPU Ray-tracing

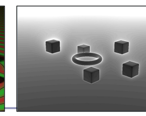
Ray tracing 101: "Choose the color of the pixel by firing a ray through and seeing what it hits."



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

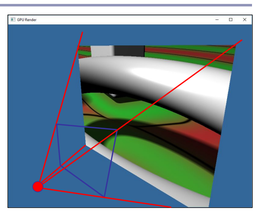






GPU Ray-tracing

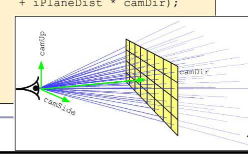
1. Use a minimal vertex shader (no transforms) - all work happens in the fragment shader
2. Set up OpenGL with minimal geometry, a single quad
3. Bind coordinates to each vertex, let the GPU interpolate coordinates to every pixel
4. Implement raytracing in GLSL:
   a. For each pixel, compute the ray from the eye through the pixel, using the interpolated coordinates 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);
}
```



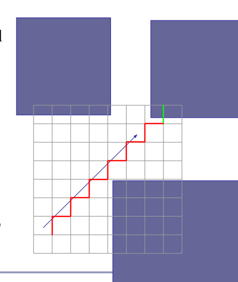
GPU Ray-tracing: Sphere

```
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-<u>marching</u>, in which we take a series of finite steps along the ray until we strike an object or exceed the number of permitted steps.

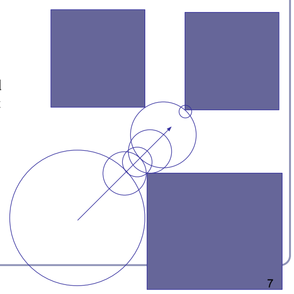
- Also sometimes called ray casting
- Scene objects only need to answer, *"has this ray hit you? y/n"*
- Great solution for data like height fields
- Unfortunately…
  - often involves many steps
  - too large a step size can lead to lost intersections (step over the object)
  - an `if()` test in the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



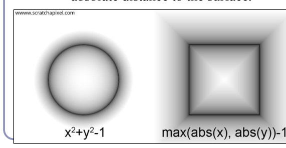
This is also sometimes called 'sphere tracing'. Early paper: http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf

Signed distance fields

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.



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

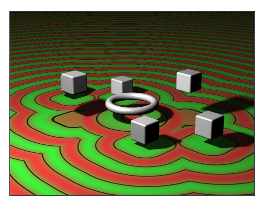
```
float sphere(vec3 p, float r) {
  return length(p) - r;
}

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;
}
```


Raymarching signed distance fields

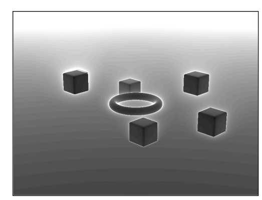
```
vec3 raymarch(vec3 pos, vec3 raydir) {
  int step = 0;
  float d = getSdf(pos);

  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

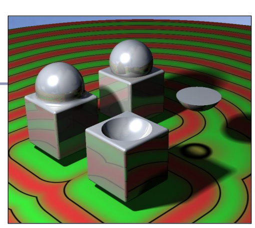


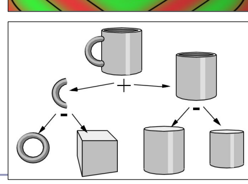
Combining SDFs

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

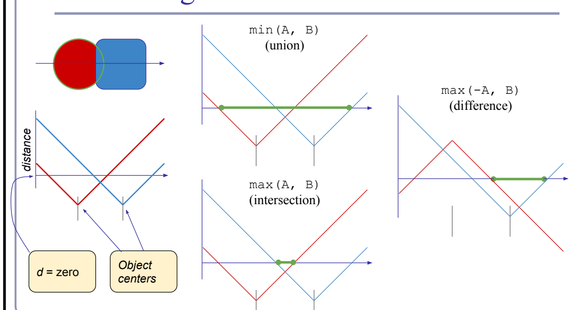
- Take the **union** of two SDFs by taking the `min()` of their functions.
- Take the **intersection** of two SDFs by taking the `max()` of their functions.
- The `max()` of function A and the negative of function B will return the **difference** of A - B.

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



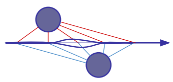


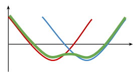
Combining SDFs

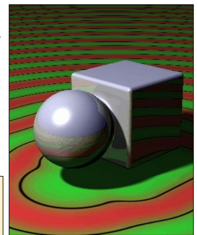


Blending SDFs

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 smin(float a, float b) {
  float k = 0.2;
  float h = clamp(0.5 + 0.5 * (b - a) / k, 0,
1);
  return mix(b, a, h) - k * h * (1 - h);
}
```

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

Transforming SDF geometry

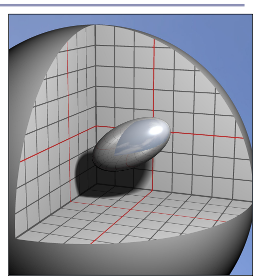
To rotate, translate or scale an SDF model, apply the inverse transform to the input point within your distance function.
Ex:

```
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.

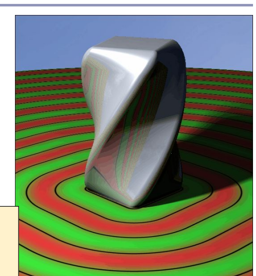
Transforming SDF geometry

```
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.



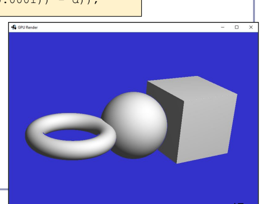
```
float fScene(vec3 pt) {
  pt.y -= 1;
  float t = (pt.y + 2.5) * sin(time);
  return sdCube(vec3(
    pt.x * cos(t) - pt.z * sin(t),
    pt.y / 2,
    pt.x * sin(t) + pt.z * cos(t)), vec3(1));
}
```


Find the normal to an SDF

Finding the normal: local gradient

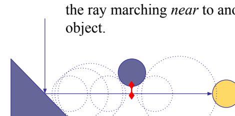
```
float d = getSdf(pt);
vec3 normal = normalize(vec3(
    getSdf(vec3(pt.x + 0.0001, pt.y, pt.z)) - d,
    getSdf(vec3(pt.x, pt.y + 0.0001, pt.z)) - d,
    getSdf(vec3(pt.x, pt.y, pt.z + 0.0001)) - d));
```

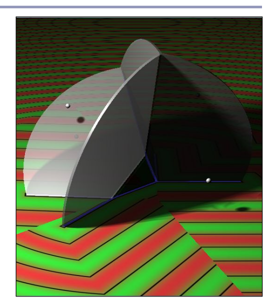
The distance function is locally linear and changes most as the sample moves directly away from the surface. At the surface, the direction of greatest change is therefore equivalent to the normal to the surface.
Thus the local gradient (the normal) can be approximated from the distance function.



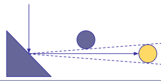
SDF shadows

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.



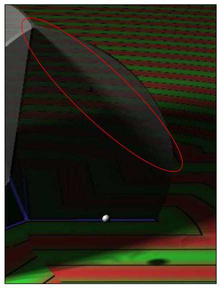


Soft SDF shadows



```
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;
}
```

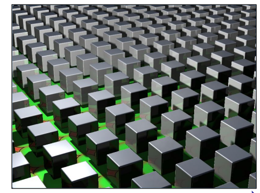
By dividing *d* by *t*, we attenuate the strength of the shadow as its source is further from the illuminated point.



Repeating SDF geometry

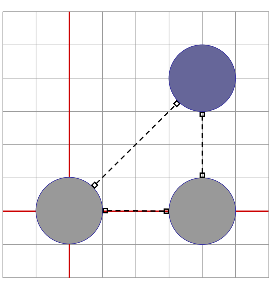
If we take the modulus of a point's position along one or more axes before computing its signed distance, then we segment space into infinite parallel regions of repeated distance. Space near the origin 'repeats'.

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));
}
```

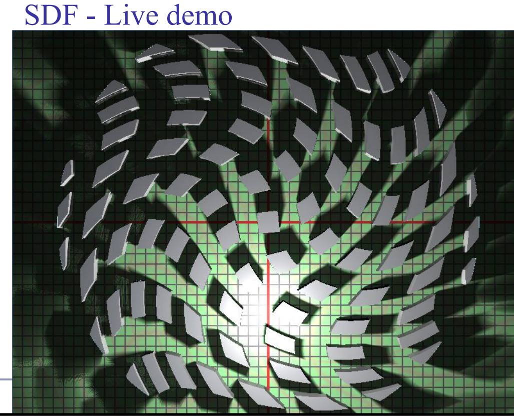

Repeating SDF geometry



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

Special kudos to Inigo Quilez and his amazing blog:

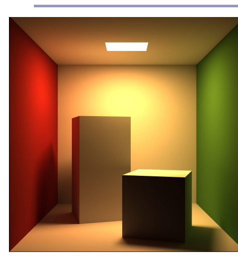
- http://iquilezles.org/www/articles/smin/smin.htm
- http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

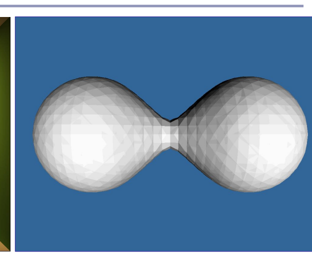
Other useful sources:

- 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

Further Graphics

Improved ray casting
Generalized Implicit Surfaces





Left: "*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.

Right: Animated polygonization of two metaballs dynamically generating an implicit surface at 5 levels of octree recursion (~3200 polygons per frame)

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

Speed things up!
Bounding volumes

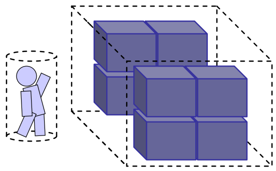
A common optimization method for ray-based rendering is the use of *bounding volumes*.

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, -marching



Types of bounding volumes

The goal is to accelerate volumetric tests, such as "does the ray hit the cow?" → *speed* trumps *precision*

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

Axis-aligned bounding boxes

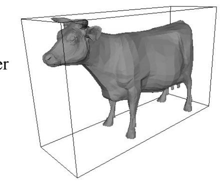
- max and min of x/y/z.

Bounding spheres

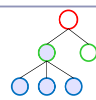
- max of radius from some rough center

Bounding cylinders

- common in early FPS games

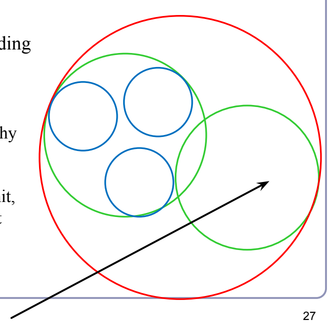


Bounding volumes in hierarchy



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

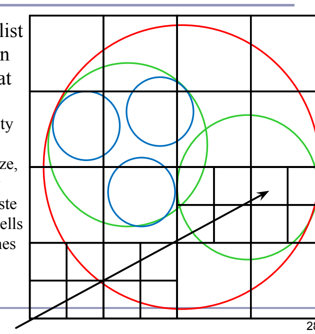
- Pro: Rays can skip subsections of the hierarchy
- Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



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 for voxelized games (ex: *Minecraft*)



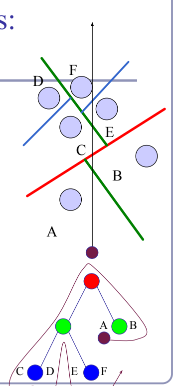
Popular acceleration structures:
BSP Trees

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

- This gives an ordering in which to test scene objects against your ray
- When you fire a ray into the scene, you test all near-side objects before testing far-side objects.

Challenges:

- requires slow pre-processing step
- strongly favors static scenes
- choice of planes is hard to optimize



Popular acceleration structures:
kd-trees

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

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

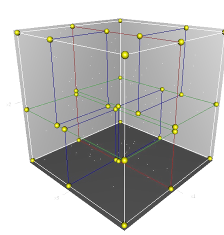


Image from Wikipedia, bless their hearts.

Popular acceleration structures: *Bounding Interval Hierarchies*

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

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

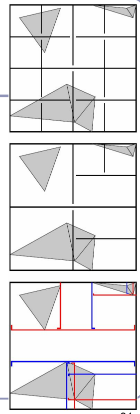


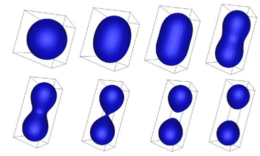
Image from Wächter and Keller's paper, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics (2006)

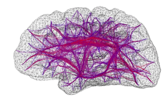
Implicit surfaces

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

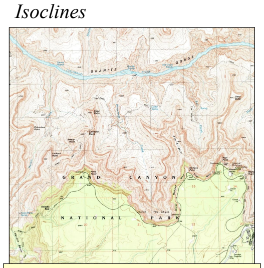




[1] AKA "*metaball modeling*", "*force functions*", "*blobby modeling*"…

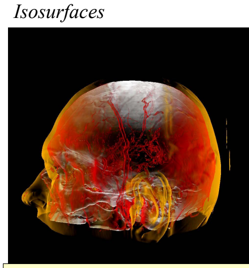
Terminology

Isoclines



Grand Cayon Quadrangle
Arizona-Coconino Co.
7.5 minute series (topographic)
Courtesy of National Park Maps (npmaps.com)

Isosurfaces



Balázs Csebfalvi, Balázs Tóth, Stefan Bruckner, Meister Eduard Gröller
Illumination-Driven Opacity Modulation for Expressive Volume Rendering, *Proceedings of Vision, Modeling & Visualization 2012*, pages 103-109. November 2012.

Implicit surface modeling

The user controls a set of *control points* or *primitives*. Each point generates a field of force, which drops off as a function of distance from the point (like gravity weakening with distance.)

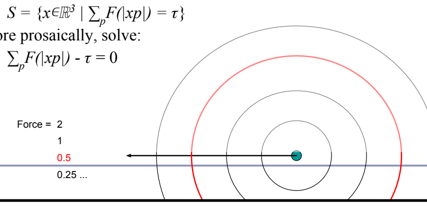
$F(r)$ = "The force at distance r"

For any real value τ, the set of all points in space where the sum of forces equals τ is an isosurface: an *implicit surface*.

$$S = \{x \in \mathbb{R}^3 \mid \sum_p F(|xp|) = \tau\}$$

...or, more prosaically, solve:

$$\sum_p F(|xp|) - \tau = 0$$



Force functions

A few popular force field functions:

- "Blobby Molecules" – Jim Blinn

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

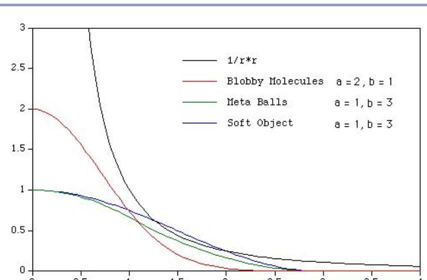
- "Metaballs" – Jim Blinn

  $$F(r) = \begin{cases} a(1 - 3r^2 / b^2) & 0 \le r < b/3 \\ (3a/2)(1 - r/b)^2 & b/3 \le r < b \\ 0 & b \le r \end{cases}$$

- "Soft Objects" – Wyvill & Wyvill

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

Comparison of force functions

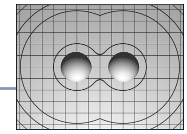


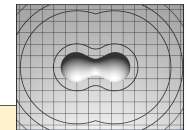
Rendering implicit surfaces

Several choices:

1. Render the surface directly to the GPU
   - **+:** Realtime lighting, smooth surfaces, looks great
   - **-:** Hard to integrate with other objects in scene
   - **-:** Solve the “intercept surface with ray” problem
2. Convert the surface into a mesh of connected polygons, approximating the surface to a fixed level of precision (“polygon soup”)
   - **+:** Mesh can be manipulated, interact with scene
   - **-:** Costly setup costs or runtime framerate hit

Rendering implicit surfaces with Signed Distance Fields





Blynn’s metaballs force function is a piecewise Polynomial:

$$F(r) = \begin{cases} a(1-3r^2/b^2) & 0 \le r < b/3 \\ (3a/2)(1-r/b)^2 & b/3 \le r < b \\ 0 & b \le r \end{cases}$$

GLSL:

```
float getMetaball(vec3 p, vec3 v) {
    float r = length(p - v);
    if (r < b / 3.0) {
        return a * (1.0 - 3.0 * r * r / b * b);
    } else if (r < b) {
        return (3.0 * a / 2.0) * (1.0 - r / b) * (1.0 - r / b);
    } else {
        return 0.0;
    }
}
```


Rendering implicit surfaces with Signed Distance Fields

Let’s use Blynn’s constants: a=1, b=3

We want to be able to answer the question, “if F < 0.5, then we’re outside the surface. What is the minimum distance from our current position to F=0.5?”

$F = (3a/2)(1-r/b)^2$

$= (3/2)(1-r/3)^2$

$r^2 - 6r + (9-6F) = 0$

$r = 3\pm\sqrt{(6F)}$

The square roots yield ± values, but we can discard the half of the polynomial whose *r* value is >b, leaving us with simply:

$r = 3-\sqrt{(6F)}$

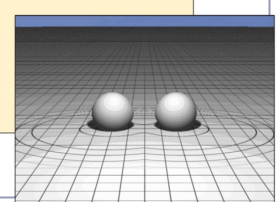
Solve for F = 0.5

→ $r = 3-\sqrt{3} = 1.2679529$

Insight: if we restrict ourselves to metaballs of weight 1, then only Blynn’s second polynomial applies outside the isosurface of F=0.5

Rendering implicit surfaces with Signed Distance Fields

```
float sdImplicitSurface(vec3 p) {
  float mb = getMetaball(p, BallA) + getMetaball(p, BallB);
  float minDist = min(length(p - BallA), length(p - BallB));

  // 1.2679529 is the x-intercept of the metaball expression
  // when force = 0.5
  float r = 1.2679529;

  float d;
  if (minDist > 3 /* b=3 */) {
    return max(minDist - 3, 3 - r);
  } else {
    return 3 - sqrt(6.0 * mb)- r;
  }
}
```



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

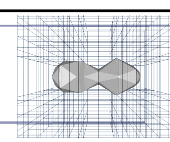
Rendering implicit surfaces with polygons

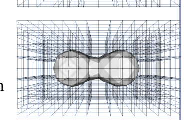
An *octree* is a recursive subdivision of space which “homes in” on the surface, from larger to finer detail.

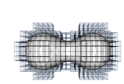
- An octree encloses a cubical volume in space. You evaluate the force function F(*v*) at each vertex *v* of the cube.
- As the octree subdivides and splits into smaller octrees, only the octrees which contain some of the surface are processed; empty octrees are discarded.

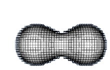
Image credit: J W Laprairie, Mark & Hamilton, Howard. (2018). *Isovox: A Brick-Octree Approach to Indirect Visualization*

Polygonizing the surface









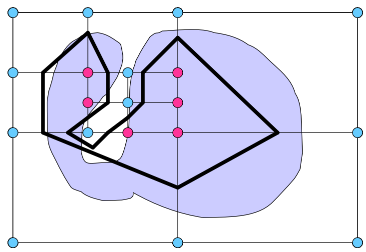
To display a set of octrees, convert the octrees into polygons.

- If some corners are “hot” (above the force limit) and others are “cold” (below the force limit) then the isosurface must cross 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):

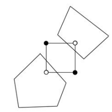


Polygonizing the surface

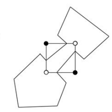
There are fifteen possible configurations (up to symmetry) of hot/cold vertices in the cube. →

- With rotations, that's 256 cases.

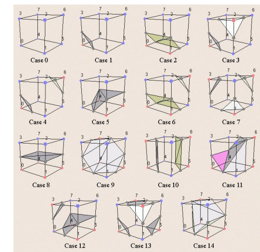
Beware: there are *ambiguous cases* in the polygonization which must be addressed separately. ↓



Break contour



Join contour



Images courtesy of Diane Lingrand

Polygonizing the surface

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

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

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

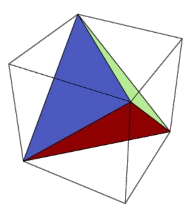
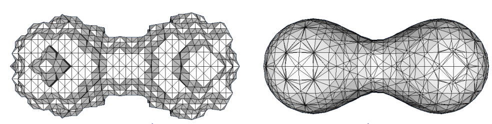


Image from the Open Problem Garden

Smoothing the polygonization

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.
  - $t = (0.5 - F(P1)) / (F(P2) - F(P1))$
  - $P = P1 + t\,(P2 - P1)$

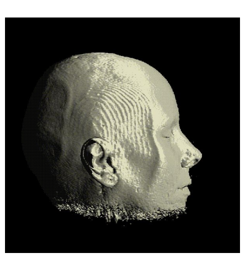


Same force points

Marching cubes

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

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



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

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

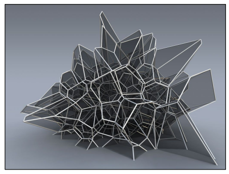
http://astronomy.swin.edu.au/~pbourke/modelling/implicitsurf/

http://www.cs.berkeley.edu/~job/Papers/turk-2002-MIS.pdf

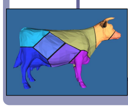
http://www.unchainedgeometry.com/jbloom/papers/interactive.pdf

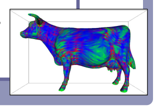
http://www-courses.cs.uiuc.edu/~cs319/polygonization.pdf

Further Graphics



A Brief Introduction to Computational Geometry



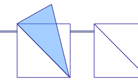


Alex Benton, University of Cambridge – alex@bentonian.com

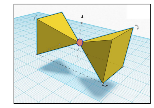
Supported in part by Google UK, Ltd

Terminology

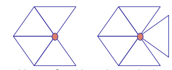
- We'll be focusing on *discrete* (as opposed to continuous) representation of geometry; i.e., polygon meshes
  - Many rendering systems limit themselves to triangle meshes
  - Many require that the mesh be *manifold*
- In a *closed manifold* polygon mesh:
  - Exactly two triangles meet at each edge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

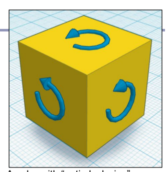


Vertex: Good boundary vs bad

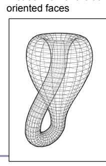
This slide draws much inspiration from Shirley and Marschner's *Fundamentals of Computer Graphics*, pp. 262-263

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.

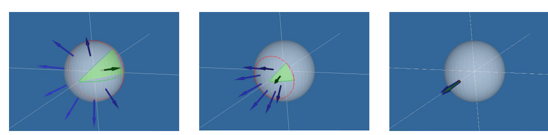
Also, terrible for holding drinks.

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*

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

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

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*.

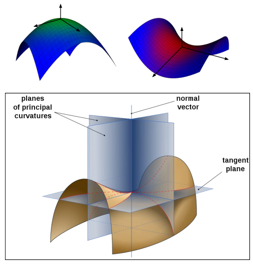
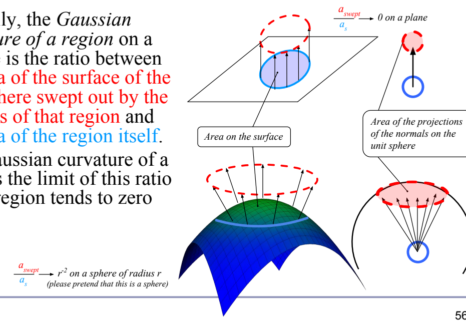


Image by Eric Gaba, from Wikipedia

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.



Gaussian curvature on discrete surfaces

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

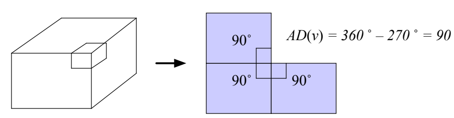
- Normals change instantaneously (as one's point of view travels across an edge from one face to another) or not at all (as one's point of view travels within a face.)

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

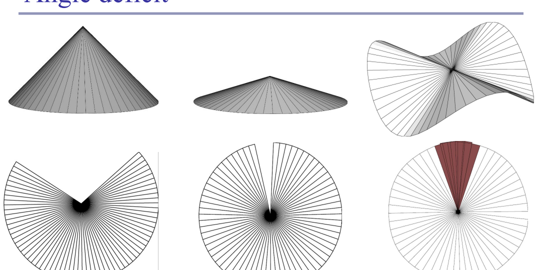
Angle deficit – a better solution for measuring discrete curvature

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

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

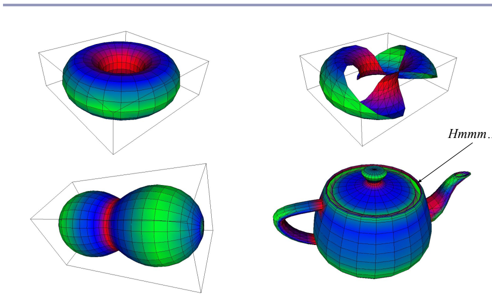


Angle deficit



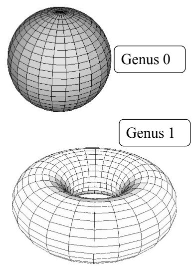
High angle deficit
Low angle deficit
Negative angle deficit

Angle deficit



Genus, Poincaré and the Euler Characteristic

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



Genus, Poincaré and the Euler Characteristic

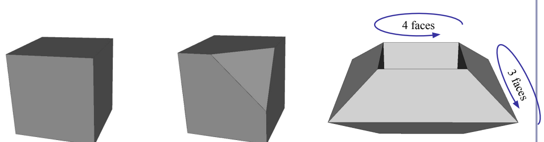
Given a polyhedral surface *S* without border where:

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

the Poincaré Formula states that:

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

Genus, Poincaré and the Euler Characteristic



$g = 0$	$g = 0$	$g = 1$
$E = 12$	$E = 15$	$E = 24$
$F = 6$	$F = 7$	$F = 12$
$V = 8$	$V = 10$	$V = 12$
$V\text{-}E\text{+}F = 2\text{-}2g = 2$	$V\text{-}E\text{+}F = 2\text{-}2g = 2$	$V\text{-}E\text{+}F = 2\text{-}2g = 0$

The Euler Characteristic and angle deficit

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

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

Cube:

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

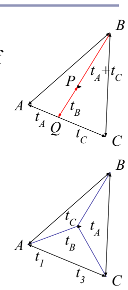
Tetrahedron:

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

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 proportional to the subtriangle areas of the three vertices.
  - The area of a triangle is ½ the length of the cross product of two of its sides.



Barycentric coordinates

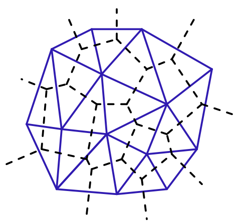
```
// Compute barycentric coordinates (u, v, w) for
// point p with respect to triangle (a, b, c)
vec3 barycentric(vec3 p, vec3 a, vec3 b, vec3 c) {
  vec3 v0 = b - a, v1 = c - a, v2 = p - a;
  float d00 = dot(v0, v0);
  float d01 = dot(v0, v1);
  float d11 = dot(v1, v1);
  float d20 = dot(v2, v0);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

Voronoi diagrams

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

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



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

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

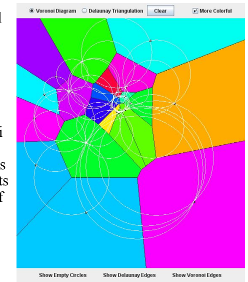
Voronoi diagrams

Given a set $S=\{p_1,p_2,\ldots,p_n\}$, the formal definition of a Voronoi cell $C(S,p_i)$ is

$C(S,p_i)=\{p \in R^d \mid |p-p_i|<|p-p_j|, i\neq j\}$

The p_i are called the *generating points* of the diagram.

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



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

Delaunay triangulations and *equi-angularity*

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

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

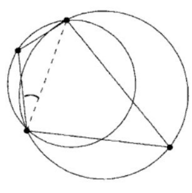


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

Delaunay triangulations and *empty circles*

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

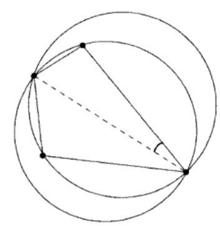


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

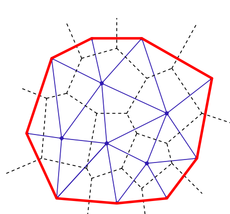
Delaunay triangulations and convex hulls

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

- This is true in 2D, 3D, 4D…

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

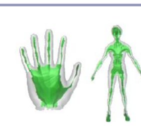
- Ex: from 2D ($P_i=\{x,y\}_i$), loft the points upwards, onto a parabola in 3D ($P'_i=\{x,y,x^2+y^2\}_i$). The resulting polyhedral mesh will still be convex in 3D.



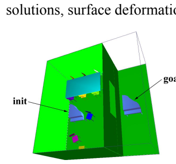
Voronoi diagrams and the *medial axis*

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

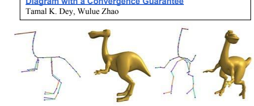
- This can be used to extract a skeleton of the surface, for (for example) path-planning solutions, surface deformation, and animation.



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



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

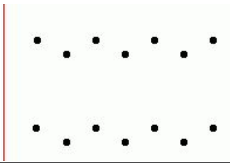


Shape Deformation using a Skeleton to Drive Simplex Transformations
IEEE Transaction on Visualization and Computer Graphics, Vol. 14, No. 3, May/June 2008, Page 693-706
Han-Bing Yan, Shi-Min Hu, Ralph R Martin, and Yong-Liang Yang

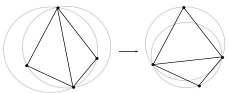
Finding the Voronoi diagram

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

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



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



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

Fortune's algorithm

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

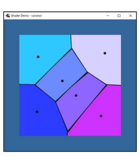
GPU-accelerated Voronoi Diagrams

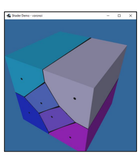
Brute force:

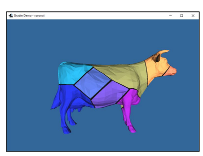
- For each pixel to be rendered on the GPU, search all points for the nearest point

Elegant (and 2D only):

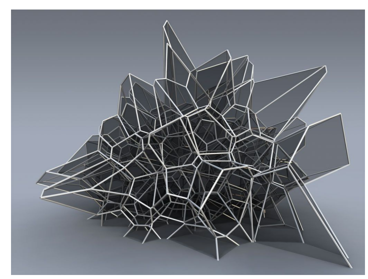
- Render each point as a discrete 3D cone in isometric projection, let z-buffering sort it out







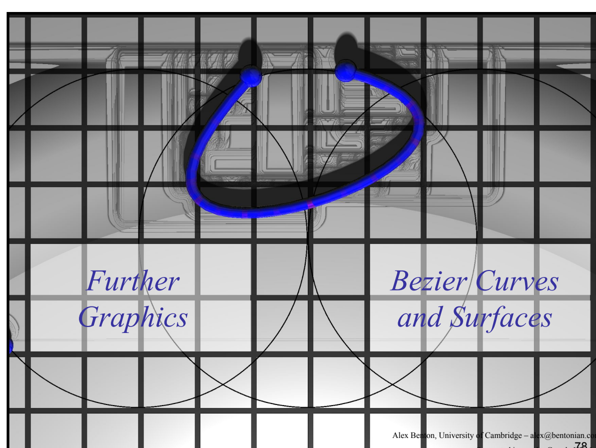
Voronoi cells in 3D



Silvan Oesterle, Michael Knauss

References

Gaussian Curvature:
http://en.wikipedia.org/wiki/Gaussian_curvature
http://mathworld.wolfram.com/GaussianCurvature.html
The Poincaré Formula:
http://mathworld.wolfram.com/PoincareFormula.html
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
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



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

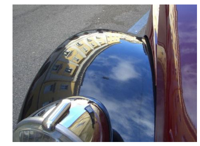
CAD, CAM, and a new motivation: *shiny things*

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

Shiny, but reflections are warped *Shiny, and reflections are perfect*

The drive for smooth CAD/CAM

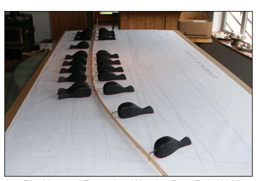
- *Continuity* (smooth curves) can be essential to the perception of *quality*.
- The automotive industry wanted to design cars which were aerodynamic, but also visibly of high quality.
- Bezier (Renault) and de Casteljau (Citroen) invented Bezier curves in the 1960s. de Boor (GM) generalized them to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



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.

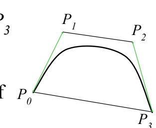


Top: Fig 3, P.7, Bray and Spectre, *Planking and Fastening*, Wooden Boat Pub (1996)
Bottom: http://www.pranos.com/boatsofwood/lofting%20ducks/lofting_ducks.htm

Bezier cubic

- A *Bezier cubic* is a function P(t) defined by four control points:

$$P(t) = (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3$$



- P_0 and P_3 are the endpoints of the curve
- P_1 and P_2 define the other two corners of the bounding polygon.
- The curve fits entirely within the convex hull of $P_0...P_3$.

Beziers

Cubics are just one example of Bezier splines:

- Linear: $P(t) = (1-t)P_0 + tP_1$
- Quadratic: $P(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2$
- Cubic: $P(t) = (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3$

...

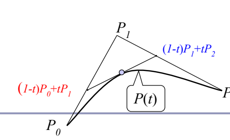
General:

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

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

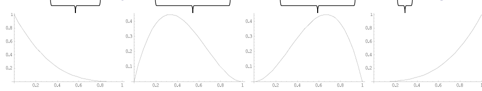
Beziers

- You can describe Beziers as *nested linear interpolations:*
  - The linear Bezier is a linear interpolation between two points:
    $P(t) = (1-t) (P_0) + (t) (P_1)$
  - The quadratic Bezier is a linear interpolation between two lines:
    $P(t) = (1-t) ((1-t)P_0+tP_1) + (t) ((1-t)P_1+tP_2)$
  - The cubic is a linear interpolation between linear interpolations between linear interpolations… etc.
- Another way to see Beziers is as a *weighted average* between the control points.



Bernstein polynomials

$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3$



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

$$\sum_{v=1}^{n} \binom{n}{v} t^v (1-t)^{n-v} = (t + (1-t))^n = 1$$

Drawing a Bezier cubic: Iterative method

Fixed-step iteration:

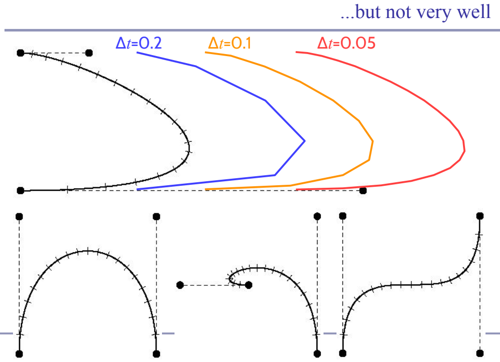
- Draw as a set of short line segments equispaced in parameter space, t:

```
(x0,y0) = Bezier(0)
FOR t = 0.05 TO 1 STEP 0.05 DO
    (x1,y1) = Bezier(t)
    DrawLine( (x0,y0), (x1,y1) )
    (x0,y0) = (x1,y1)
END FOR
```

- Problems:
  - Cannot fix a number of segments that is appropriate for all possible Beziers: too many or too few segments
  - distance in real space, (x,y), is not linearly related to distance in parameter space, t

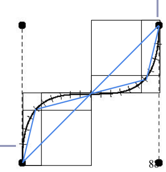
Drawing a Bezier cubic

...but not very well



Drawing a Bezier cubic: Adaptive method

- Subdivision:
  - check if a straight line between P_0 and P_3 is an adequate approximation to the Bezier
  - if so: draw the straight line
  - if not: divide the Bezier into two halves, each a Bezier, and repeat for the two new Beziers
- Need to specify some tolerance for when a straight line is an adequate approximation
  - when the Bezier lies within half a pixel width of the straight line along its entire length



Drawing a Bezier cubic: Adaptive method

```
Procedure DrawCurve( Bezier curve )
VAR Bezier left, right
BEGIN DrawCurve
  IF Flat(curve) THEN
     DrawLine(curve)
  ELSE
     SubdivideCurve(curve, left, right)
     DrawCurve(left)
     DrawCurve(right)
  END IF
END DrawCurve
```

e.g. if P_1 and P_2 both lie within half a pixel width of the line joining P_0 to P_3, then...

...draw a line from P_0 to P_3; otherwise,

...split the curve into two Beziers covering the first and second halves of the original and draw recursively

Checking for flatness

$P(t) = (1-t)\ A + t\ B$

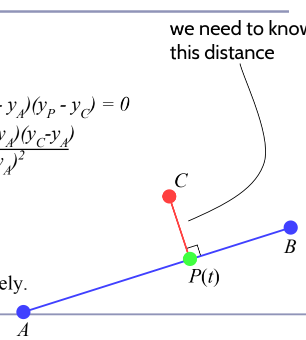
$AB \cdot CP(t) = 0$

$\rightarrow (x_B - x_A)(x_P - x_C) + (y_B - y_A)(y_P - y_C) = 0$

$\rightarrow t = \dfrac{(x_B-x_A)(x_C-x_A)+(y_B-y_A)(y_C-y_A)}{(x_B-x_A)^2+(y_B-y_A)^2}$

$\rightarrow t = \dfrac{AB \cdot AC}{|AB|^2}$

Careful! If $t < 0$ or $t > 1$, use $|AC|$ or $|BC|$ respectively.



Subdividing a Bezier cubic in two

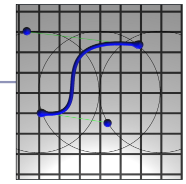
To split a Bezier cubic into two smaller Bezier cubics:

$Q_0 = P_0$
$Q_1 = \frac{1}{2} P_0 + \frac{1}{2} P_1$
$Q_2 = \frac{1}{4} P_0 + \frac{1}{2} P_1 + \frac{1}{4} P_2$
$Q_3 = \frac{1}{8} P_0 + \frac{3}{8} P_1 + \frac{3}{8} P_2 + \frac{1}{8} P_3$

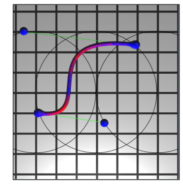
$R_3 = \frac{1}{8} P_0 + \frac{3}{8} P_1 + \frac{3}{8} P_2 + \frac{1}{8} P_3$
$R_2 = \frac{1}{4} P_1 + \frac{1}{2} P_2 + \frac{1}{4} P_3$
$R_1 = \frac{1}{2} P_2 + \frac{1}{2} P_3$
$R_0 = P_3$

These cubics will lie atop the halves of their parent exactly, so rendering them = rendering the parent.

Drawing a Bezier cubic: Signed Distance Fields



1. Iterative implementation
   SDF(P) = min(distance from *P* to each of *n* line segments)
   - In the demo, 50 steps suffices
2. Adaptive implementation
   SDF(P) = min(distance to each sub-curve whose bounding box contains *P*)
   - Can fast-discard sub-curves whose bbox doesn't contain *P*
   - In the demo, 25 subdivisions suffices

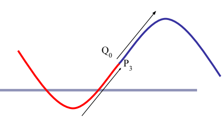


Overhauser's cubic

Overhauser's cubic: a Bezier cubic which passes through four target data points

- Calculate the appropriate Bezier control point locations from the given data points
  - e.g. given points A, B, C, D, the Bezier control points are:
  - P0 = B   P1 = B + (C-A)/6
  - P3 = C   P2 = C - (D-B)/6
- Overhauser's cubic *interpolates* its controlling points
  - good for animation, movies; less for CAD/CAM
  - moving a single point modifies four adjacent curve segments
  - compare with Bezier, where moving a single point modifies just the two segments connected to that point

Types of curve join

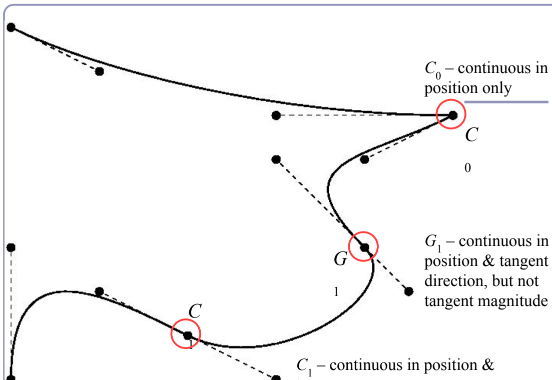


- each curve is smooth within itself
- joins at endpoints can be:
  - C_1 – continuous in both position and tangent vector
    - smooth join in a mathematical sense
  - G_1 – continuous in position, tangent vector in same direction
    - smooth join in a geometric sense
  - C_0 – continuous in position only
    - "corner"
  - discontinuous in position

C_n (mathematical continuity): continuous in all derivatives up to the n^{th} derivative

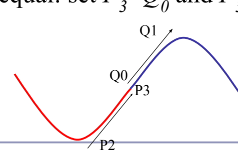
G_n (geometric continuity): each derivative up to the n^{th} has the same "direction" to its vector on either side of the join

$C_n \Rightarrow G_n$

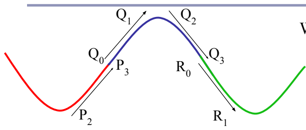


Joining Bezier splines

- To join two Bezier splines with C0 continuity, set $P_3=Q_0$.
- To join two Bezier splines with C1 continuity, require C0 and make the tangent vectors equal: set $P_3=Q_0$ and $P_3-P_2=Q_1-Q_0$.



What if we want to chain Beziers together?



Consider a chain of splines with many control points…

$P = \{P_0, P_1, P_2, P_3\}$
$Q = \{Q_0, Q_1, Q_2, Q_3\}$
$R = \{R_0, R_1, R_2, R_3\}$

…with C1 continuity…

$P3=Q_0$, $P_2-P_3=Q_0-Q_1$
$Q3=R_0$, $Q_2-Q_3=R_0-R_1$

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

The curve $C(t)$ would be:

$C(t) = P(t) \bullet ((0 \le t < 1)\ ?\ 1 : 0) +$
$Q(t-1) \bullet ((1 \le t < 2)\ ?\ 1 : 0) +$
$R(t-2) \bullet ((2 \le t < 3)\ ?\ 1 : 0)$

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

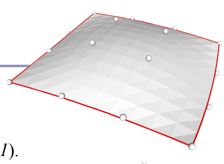
Tensor product

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

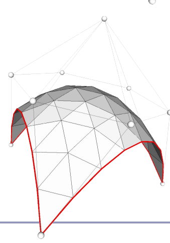
$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \otimes \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

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

Bezier patches



- If curve A has n control points and curve B has m control points then A⊗B is an (n)x(m) matrix of polynomials of degree $max(n-1, m-1)$.
  - *⊗ = tensor product*
- Multiply this matrix against an (n)x(m) matrix of control points and sum them all up and you've got a bivariate expression for a rectangular surface patch, in 3D
- This approach generalizes to triangles and arbitrary n-gons.



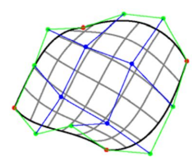
Bezier patch definition

The Bezier patch defined by sixteen control points,

$$\begin{matrix} P_{0,0} & \cdots & P_{0,3} \\ \vdots & & \vdots \\ P_{3,0} & \cdots & P_{3,3} \end{matrix}$$

is:

$$P(s,t) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(s) b_j(t) P_{i,j}$$



Compare this to the 2D version:

$$P(t) = \sum_{i=0}^{3} b_i(t) P_i$$

Continuity between Bezier patches

Ensuring continuity in 3D:

- C0 – continuous in position
  - the four edge control points must match
- C1 – continuous in position and tangent vector
  - the four edge control points must match
  - the two control points on either side of each of the four edge control points must be co-linear with both the edge point, and each other, *and* be equidistant from the edge point
- G1 – continuous in position and tangent direction the four edge control points must match the relevant control points must be co-linear

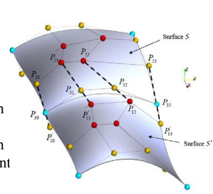
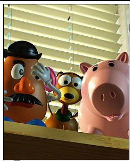
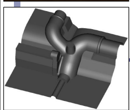


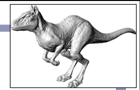
Image credit: Olivier Czarny, Guido Huysmans. *Bézier surfaces and finite elements for MHD simulations*. Journal of Computational Physics Volume 227, Issue 16, 10 August 2008

References

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


Further Graphics



NURBS

Non-Uniform Rational B-Splines

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

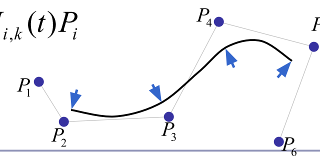
NURBS curves

Like Bezier cubics, NURBS curves are parametric

Their shape is determined by:

- control points, P_i
- the *NURBS basis functions*, $N_{i,k}$

$$P(t)=\sum_{i=1}^{n}N_{i,k}(t)P_i$$

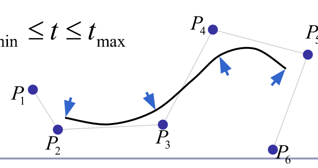


Properties of NURBS curves

1. The basis functions must sum to 1.0

$$P(t)=\sum_{i=1}^{n}N_{i,k}(t)P_i \longrightarrow$$

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

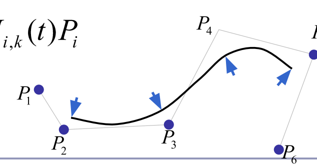


Properties of NURBS curves

2. The basis functions are calculated from a *knot vector*

- This is a non-decreasing sequence of real numbers
  - e.g. [0,0,0,1,1,1]
  - or [1,2,3,4,5,6]
  - or [1.2, 3.4, 5.6, 5.6, 7.2, 15.6]

$$P(t)=\sum_{i=1}^{n}N_{i,k}(t)P_i$$

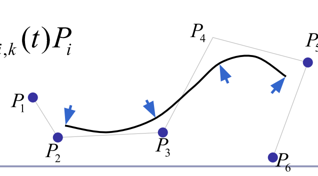


Properties of NURBS curves

3. If the basis functions are C*m*-continuous at *t*, then *P(t)* is guaranteed to be C*m*-continuous at *t*

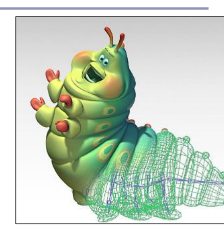
- So continuity depends only on the basis functions, $N_{i,k}$
  - Continuity **does not depend** on the locations of the control points

$$P(t)=\sum_{i=1}^{n}N_{i,k}(t)P_i$$



Properties of NURBS surfaces

NURBS surfaces are a bivariate generalisation of the univariate NURBS curve

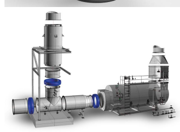


$$P(t)=\sum_{i=1}^{n}N_{i,k}(t)P_i$$

$$P(s,t)=\sum_{i=1}^{m}\sum_{j=1}^{n}N_{i,k}(s)N_{j,k}(t)P_{i,j}$$

NURBS

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

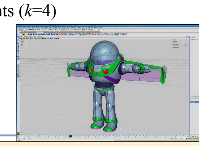


Images from www.rhino3d.com/gallery

B-Splines

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

- d, the *degree* of the curve
- $k = d+1$, called the *parameter* of the curve
- $\{P_1 ... P_n\}$, a list of n *control points*
- $[t_1, ..., t_{k+n}]$, a *knot vector* of (k+n) parameter values ("knots")
- $d = k-1$ is the degree of the curve, so k is the number of control points which influence a single interval
  - Ex: a cubic (d=3) has four control points (k=4)
- There are $k+n$ knots t_i, and $t_i \leq t_{i+1}$ for all t_i
- Each B-spline is $C^{(k-2)}$ continuous: *continuity* is degree minus one, so a k=3 curve has d=2 and is C1



http://www.mikekrummhoefener.com/toy-story-char-g...

B-Splines

- A B-spline curve is defined between t_{min} and t_{max}:

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

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

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

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

B-Splines

t_1 | t_2 | t_3 | t_4 | t_5 | …

k=1: $N_{1,1}(t)$ $N_{2,1}(t)$ $N_{3,1}(t)$ $N_{4,1}(t)$ …

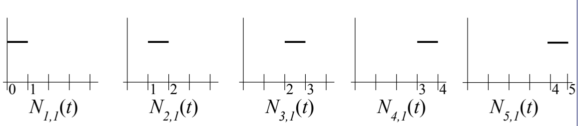
k=2: $N_{1,2}(t)$ $N_{2,2}(t)$ $N_{3,2}(t)$ …

k=3: $N_{1,3}(t)$ $N_{2,3}(t)$ …

k=4: $N_{1,4}(t)$ …

B-Splines

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



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

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

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

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

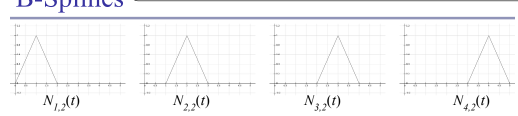
$N_{4,1}(t)=1, 3 \leq t < 4$

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

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

B-Splines

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



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

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

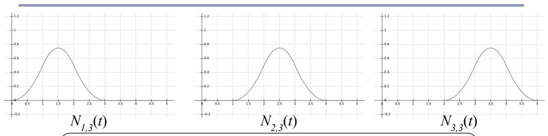
$$N_{3,2}(t) = \frac{t-2}{3-2} N_{3,1}(t) + \frac{4-t}{4-3} N_{4,1}(t) = \begin{cases} t-2 & 2 \leq t < 3 \\ 4-t & 3 \leq t < 4 \end{cases}$$

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

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

B-Splines

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



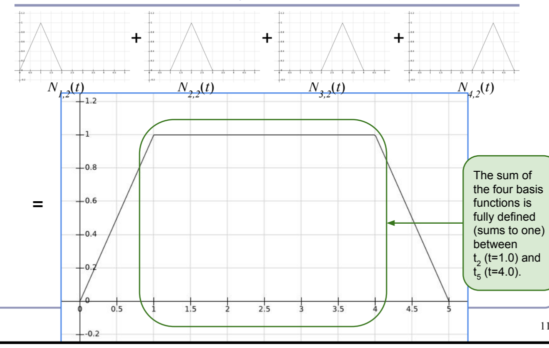
$$N_{1,3}(t) = \frac{t-0}{2-0}N_{1,2}(t) + \frac{3-t}{3-1}N_{2,2}(t) = \begin{cases} t^2/2 & 0 \le t < 1 \\ -t^2 + 3t - 3/2 & 1 \le t < 2 \\ (3-t)^2/2 & 2 \le t < 3 \end{cases}$$

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

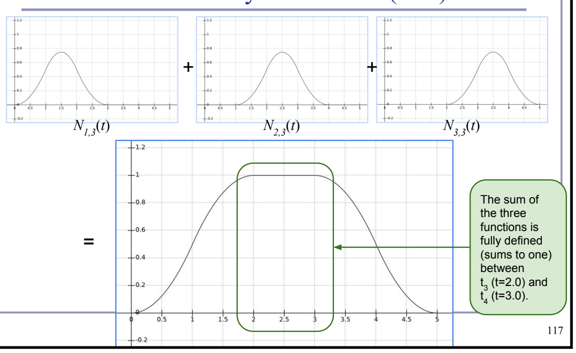
$$N_{3,3}(t) = \frac{t-2}{4-2}N_{3,2}(t) + \frac{5-t}{5-3}N_{4,2}(t) = \begin{cases} (t-2)^2/2 & 2 \le t < 3 \\ -t^2 + 7t - 23/2 & 3 \le t < 4 \\ (5-t)^2/2 & 4 \le t < 5 \end{cases}$$

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

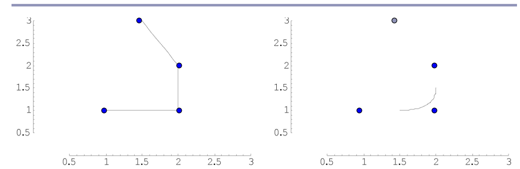
Basis functions really sum to one (k=2)



Basis functions really sum to one (k=3)



B-Splines



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

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

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

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

Non-Uniform B-Splines

- The knot vector {0,1,2,3,4,5} is *uniform*:

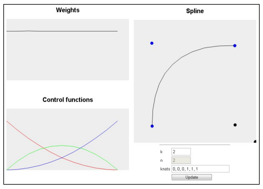
  $t_{i+1} - t_i = t_{i+2} - t_{i+1} \ \forall t_i$.
- Varying the size of an interval changes the parametric-space distribution of the weights assigned to the control functions.
- Repeating a knot value reduces the continuity of the curve in the affected span by one degree.
- Repeating a knot k times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

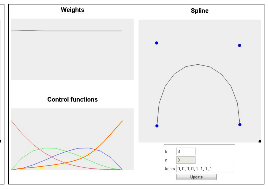
Open vs *Closed*

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

Open vs Closed

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





Non-Uniform *Rational* B-Splines

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

Non-Uniform Rational B-Splines

- Recall: $[x, y, z, \omega]_H \rightarrow [x / \omega, y / \omega, z / \omega]$
  - Or: $[x, y, z, 1] \rightarrow [x\omega, y\omega, z\omega, \omega]_H$
- The control point
  $P_i=(x_i, y_i, z_i)$
  becomes the homogeneous control point
  $P_{iH}=(x_i\omega_i, y_i\omega_i, z_i\omega_i)$
- A NURBS in homogeneous coordinates is:

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

Non-Uniform Rational B-Splines

- To convert from homogeneous coords to normal coordinates:

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

$$x(t) = x_H(t) / \omega(t)$$
$$y(t) = y_H(t) / \omega(t)$$
$$z(t) = z_H(t) / \omega(t)$$

Non-Uniform Rational B-Splines

- A piecewise rational curve is thus defined by:

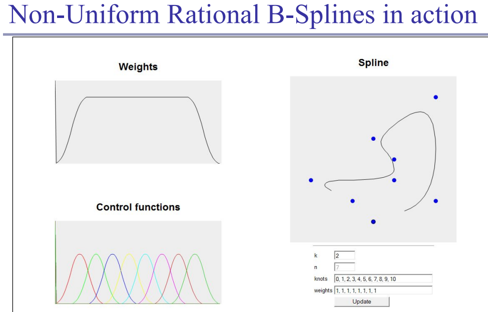
$$P(t) = \sum_{i=1}^{n} R_{i,k}(t)P_i,\ t_{min} t < t_{max}$$

with supporting *rational basis functions*:

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

This is essentially an average re-weighted by the ω's.
- Such a curve can be made to pass arbitrarily far or near to a control point by changing the corresponding weight.

Non-Uniform Rational B-Splines in action



Demo

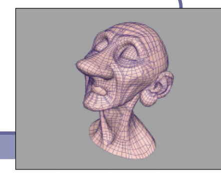
References

Demo: http://geometrie.foretnik.net/files/NURBS-en.swf

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

Further Graphics

Subdivision Surfaces

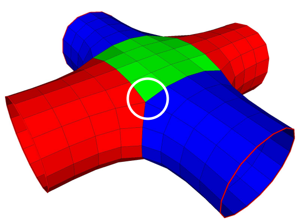


Alex Benton, University of Cambridge – alex@bentonian.com

Supported in part by Google UK, Ltd

Problems with Bezier (NURBS) patches

- Joining spline 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.

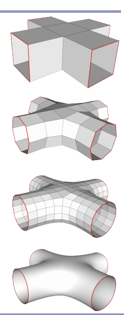
Subdivision surfaces

- 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)

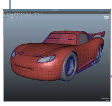
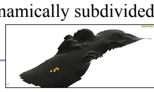
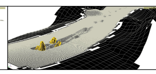
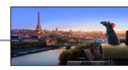
Subdivision surfaces – History

- 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

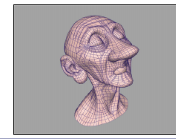
- Pixar first demonstrated subdivision surfaces in 1997 with Geri's Game.
  - Up until then they'd done everything in NURBS (Toy Story, A Bug's Life.)
  - From 1999 onwards everything they did was with subdivision surfaces (Toy Story 2, Monsters Inc, Finding Nemo...)
  - Two decades on, it's all heavily customized - creases and edges can be detailed by artists and regions of subdivision can themselves be dynamically subdivided



Useful terms

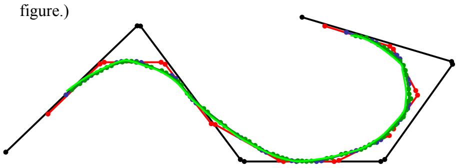
- A scheme which describes a 1D curve (even if that curve is travelling in 3D space, or higher) is called *univariate*, referring to the fact that the limit curve can be approximated by a polynomial in one variable (t).
- A scheme which describes a 2D surface is called *bivariate*, the limit surface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

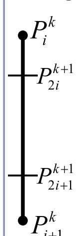
How it works

- 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:



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

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

…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:

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

Notation

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

Reading the kernel

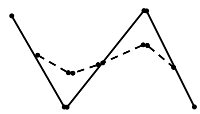
Consider the kernel

h=(1/8)[…,0,0,1,4,6,4,1,0,0,…]

You would read this as

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

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

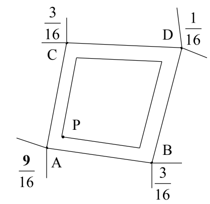


The limit curve is provably C2-continuous.

Making the jump to 3D: Doo-Sabin

Doo-Sabin takes Chaikin to 3D:

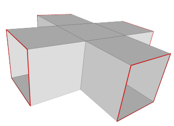
P =(9/16) A +
(3/16) B +
(3/16) C +
(1/16) D



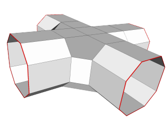
This replaces every old vertex with four new vertices.

The limit surface is biquadratic, C1 continuous everywhere.

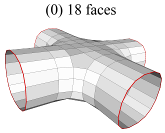
Doo-Sabin in action



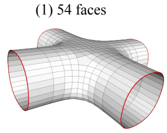
(0) 18 faces



(1) 54 faces



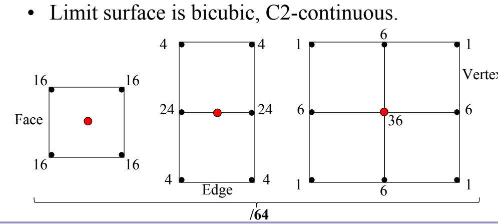
(2) 190 faces



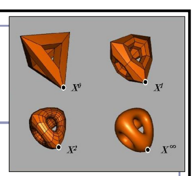
(3) 702 faces

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

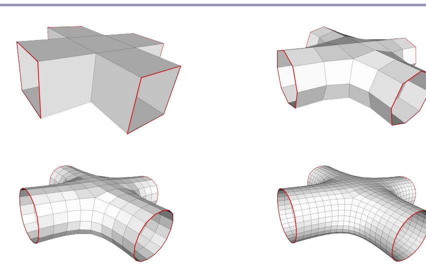


Getting tensor again:

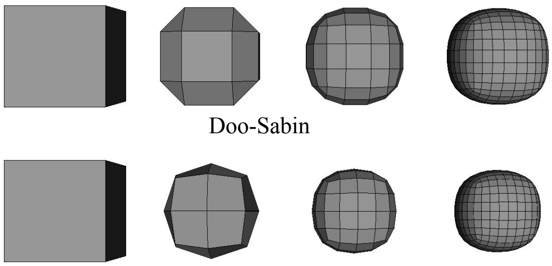
$$\frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix}\otimes\frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix}=\frac{1}{64}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&36&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$$

Vertex rule Face rule Edge rule

Catmull-Clark in action

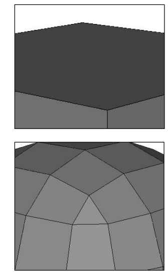


Catmull-Clark vs Doo-Sabin



Extraordinary vertices

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



Detail of Doo-Sabin at cube corner

Extraordinary vertices: Catmull-Clark

Catmull-Clark vertex rules generalized for extraordinary vertices:

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

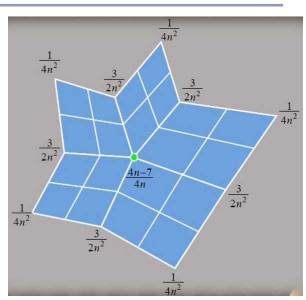
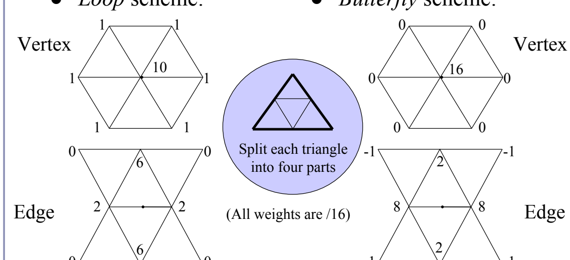


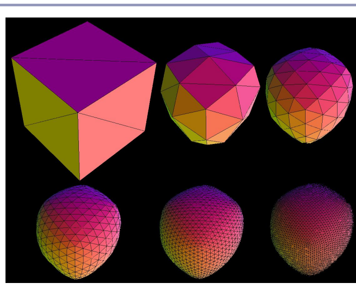
Image source: "*Next-Generation Rendering of Subdivision Surfaces*", Ignacio Castaño, SIGGRAPH 2008

Schemes for simplicial (triangular) meshes

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



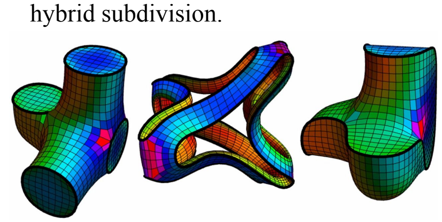
Loop subdivision



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

Creases

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



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

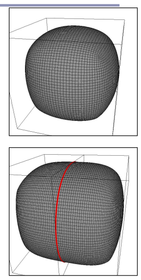
Splitting a subdivision surface

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)

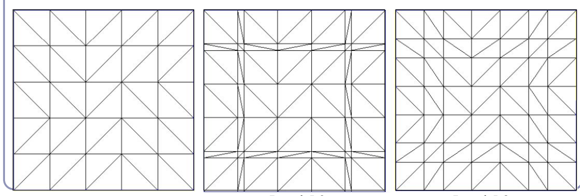
- Need to include all control points from the previous generation, which influence the limit surface in this smaller part.



(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.

Continuous level of detail

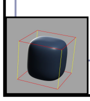
For live applications (e.g. games) can compute *continuous* level of detail, typically as a function of distance:



Level 5 Level 5.2 Level 5.8

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).



Subdivision Schemes—A partial list

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

Many more exist, some much more complex

This is a major topic of ongoing research

References

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

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

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

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

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

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

Improving on the classic lighting model

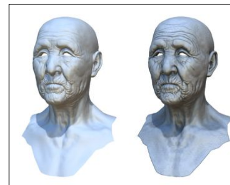
- Soft shadows are expensive
- Shadows of transparent objects require further coding or hacks
- Lighting off reflective objects follows different shadow rules from normal lighting
- Hard to implement diffuse reflection (color bleeding, such as in the Cornell Box—notice how the sides of the inner cubes are shaded red and green.)

- Fundamentally, the ambient term is a hack and the diffuse term is only one step in what should be a recursive, self-reinforcing series.

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

Ambient occlusion

- *Ambient illumination* is a blanket constant that we often add to every illuminated element in a scene, to (inaccurately) model the way that light scatters off all surfaces, illuminating areas not in direct lighting.
- *Ambient occlusion* is the technique of adding/removing ambient light when other objects are nearby and scattered light wouldn't reach the surface.
- Computing ambient occlusion is a form of *global illumination*, in which we compute the lighting of scene elements in the context of the scene as a whole.



Original model With ambient occlusion

Image from "ZBrush® Character Creation: Advanced Digital Sculpting, Second Edition", by Scott Spencer, 2011

Ambient occlusion in action

Car photos from John Hable's presentation at GDC 2010, "Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

Ambient occlusion in action

Car photos from John Hable's presentation at GDC 2010, "Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

Ambient occlusion in action

Car photos from John Hable's presentation at GDC 2010, "Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

Ambient occlusion in action

Car photos from John Hable's presentation at GDC 2010, "Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

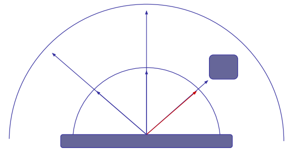
Ambient occlusion - Theory

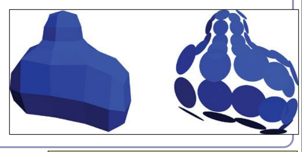
We can treat the background (the sky) as a vast ambient illumination source.

- For each vertex of a surface, compute how much background illumination reaches the vertex by computing how much sky it can 'see'
- Integrate occlusion A_p over the hemisphere around the normal at the vertex:

$$A_{\bar{p}} = \frac{1}{\pi}\int_{\Omega} V_{\bar{p},\hat{\omega}}(\hat{n} \cdot \hat{\omega})d\omega$$

- A_p occlusion at point p
- n normal at point p
- $V_{p,\omega}$ visibility from p in direction ω
- Ω integrate over area (hemisphere)

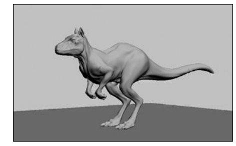




Bottom image credit: "GPU Gems 2", nVidia, 2005. Vertices mapped to illumination disks for hemispheric illumination mapping.

Ambient occlusion - Theory

- This approach is very flexible
- Also very expensive!
- To speed up computation, randomly sample rays cast out from each polygon or vertex (this is a *Monte-Carlo* method)
- Alternatively, render the scene from the point of view of each vertex and count the background pixels in the render
- Best used to pre-compute per-object "*occlusion maps*", texture maps of shadow to overlay onto each object
- But pre-computed maps fare poorly on animated models...



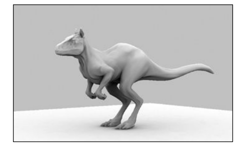


Image credit: "GPU Gems 1", nVidia, 2004. Top: without AO. Bottom: with AO.

Screen Space Ambient Occlusion ("SSAO")

"True ambient occlusion is hard, let's go hacking."

- Approximate ambient occlusion by comparing z-buffer values in screen space!
- Open plane = unoccluded
- Closed 'valley' in depth buffer = shadowed by nearby geometry
- Multi-pass algorithm
- Runs entirely on the GPU

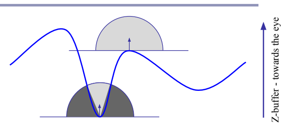
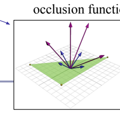


Image: CryEngine 2. M. Mittring, "Finding Next Gen – CryEngine 2.0, Chapter 8", SIGGRAPH 2007 Course 28

Screen Space Ambient Occlusion

1. For each visible point on a surface in the scene (ie., each pixel), take multiple samples (typically between 8 and 32) from nearby and map these samples back to screen space
2. Check if the depth sampled at each neighbor is nearer to, or further from, the scene sample point
3. If the neighbor is nearer than the scene sample point then there is some degree of occlusion
   a. Care must be taken not to occlude if the nearer neighbor is too much nearer than the scene sample point; this implies a separate object, much closer to the camera
4. Sum retained occlusions, weighting with an occlusion function



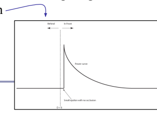


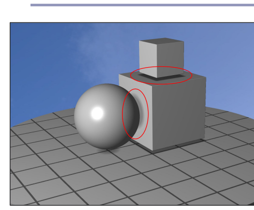
Image: StarCraft II. *Advances in Real-Time Rendering in 3D Graphics and Games* - Course notes, SIGGRAPH 2008

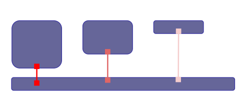
SSAO example- Uncharted 2

4) Low Pass Filter (significant blurring)

John Hable, GDC 2010, "Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

Ambient occlusion and Signed Distance Fields





In a nutshell, SSAO tries to estimate occlusion by asking, "how far is it to the nearest neighboring geometry?"

With signed distance fields, this question is almost trivial to answer.

```
float ambient(vec3 pt, vec3 normal) {
  return abs(getSdf(pt + 0.1 * normal)) / 0.1;
}
```

```
float ambient(vec3 pt, vec3 normal) {
  float a = 1;
  int step = 0;

  for (float t = 0.01; t <= 0.1; ) {
    float d = abs(getSdf(pt + t * normal));
    a = min(a, d / t);
    t += max(d, 0.01);
  }
  return a;
}
```

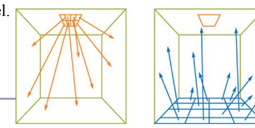
Radiosity

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

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

Radiosity—algorithm

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



Radiosity—mathematical support

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

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

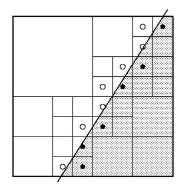
$$B_i = E_i + R_i \sum_{j=1, j \neq i}^{n} B_j F_{ij}$$

This forms a system of linear equations, where…

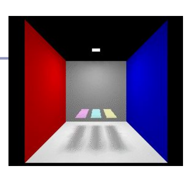
B_i is the radiosity of patch i;
B_j is the radiosity of each of the other patches ($j \neq i$)
E_i is the emitted energy of the patch
R_i is the reflectivity of the patch
F_{ij} is the view factor of energy from patch i to patch j.

Radiosity—form factors

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



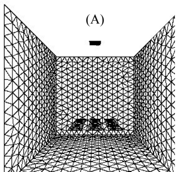
Radiosity—implementation



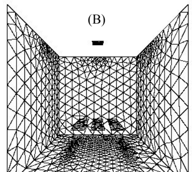
(A) Simple patch triangulation
(B) Adaptive patch generation: the floor and walls of the room are dynamically subdivided to produce more patches where shadow detail is higher.

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

(A)



(B)

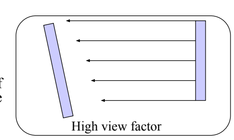


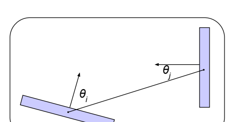
Radiosity—view factors

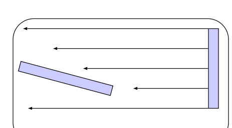
One equation for the view factor between patches *i*, *j* is:

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

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

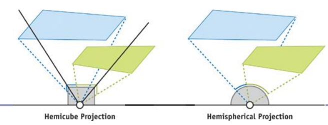






Radiosity—calculating visibility

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



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

Radiosity gallery

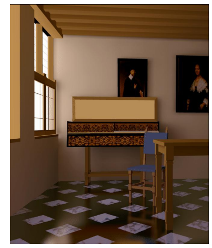


Image from *A Two Pass Solution to the Rendering Equation: a Synthesis of Ray Tracing and Radiosity Methods*, John R. Wallace, Michael F. Cohen and Donald P. Greenberg (Cornell University, 1987)

Image from *GPU Gems II*, nVidia

Teapot (wikipedia)

Shadows, refraction and caustics

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

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

Shadows, refraction and caustics

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

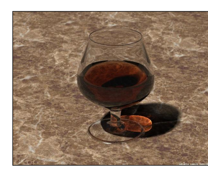


Image from http://graphics.ucsd.edu/~henrik/
Generated with photon mapping

Photon mapping

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

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

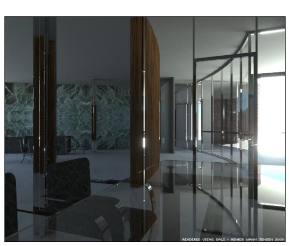


Image by Henrik Jensen (2000)

Photon mapping—algorithm (1/2)

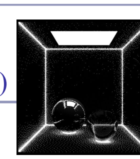


Image by Zack Waters

Photon mapping is a two-pass algorithm:

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

Photon mapping—algorithm (2/2)

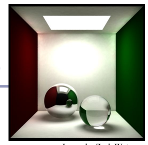


Image by Zack Waters

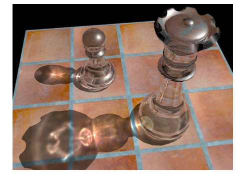
Photon mapping is a two-pass algorithm:

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

Photon mapping is probabilistic

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

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


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

Photon mapping is probabilistic

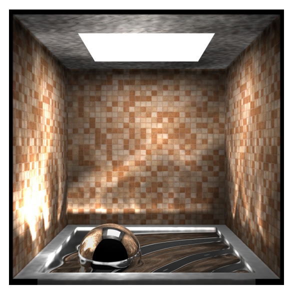
- Initial photon direction is random. Constrained by light shape, but random.
- What exactly happens each time a photon hits a solid also has a random component:
  - Based on the diffuse reflectance, specular reflectance and transparency of the surface, compute probabilities p_d, p_s and p_t where $(p_d+p_s+p_t)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

0 p_d p_s p_t 1

This surface would have minimal specular highlight.

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

Photon mapping gallery


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


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


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

References

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

Anisotropic surface:

- A. Watt, 3D Computer Graphics - Chapter 7: Simulating light-object interaction: local reflection models
- Eurographics 2016 tutorial - D. Guarnera, G. C. Guarnera, A. Ghosh, C. Denk, and M. Glencross - BRDF Representation and Acquisition

Ambient occlusion and SSAO:

- "GPU Gems 2", nVidia, 2005. Vertices mapped to illumination. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter14.html
- Mittring, M. 2007. *Finding Next Gen – CryEngine 2.0,* Chapter 8, SIGGRAPH 2007 Course 28 – Advanced Real-Time Rendering in 3D Graphics and Games http://developer.amd.com/wordpress/media/2012/10/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf
- John Hable's presentation at GDC 2010, "Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

Radiosity:

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

Photon mapping

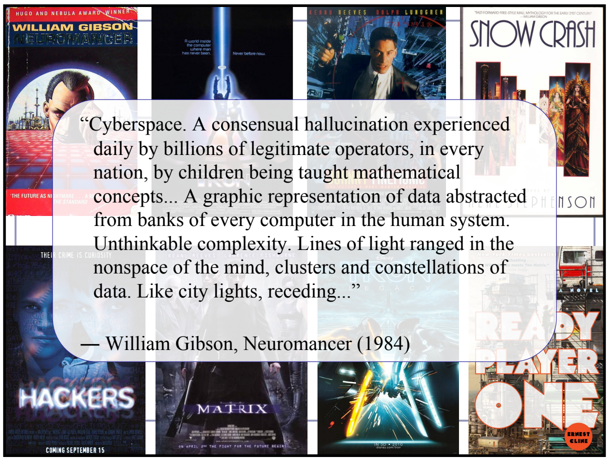
- Henrik Jenson, "Global Illumination using Photon Maps": http://graphics.ucsd.edu/~henrik/
- Zack Waters, "Photon Mapping": http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html

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*.

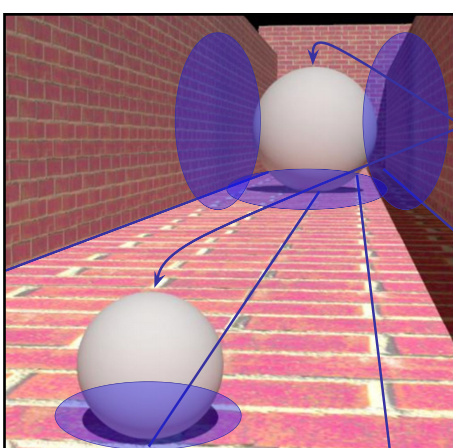
"The right way to think about computer graphics is that the screen is a window through which one looks into a virtual world. And the challenge is to makes the world look real, sound real, feel real and interact realistically."
-Ivan Sutherland (1965)

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

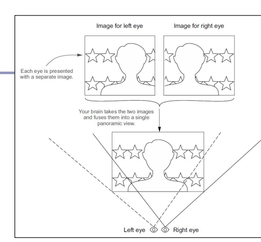
Murray, Boyaci, Kersten, *The representation of perceived angular size in human primary visual cortex*, Nature Neuroscience (2006)

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.

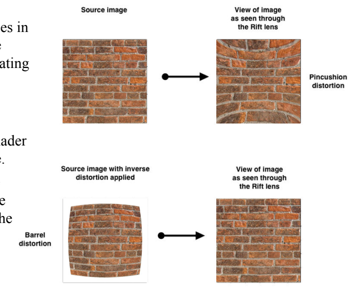
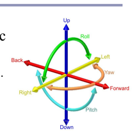


Image credit: Davis, Bryla, Benton, *Oculus Rift in Action* (2014)

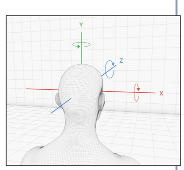
Sensors

Accelerometer and electromagnetic sensors in the headset trac 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;
};
```



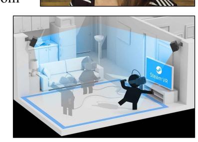
Top: 6DoF (6 degrees of freedom) - Wikipedia
Bottom: Roll (Z), Pitch (X) and Yaw (Y) - Google Design

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.

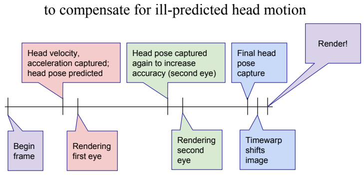
- At time t, head pos = X, head velocity = V, head acceleration = A
- Human heads do not accelerate very fast
- Rendering a single frame takes dt milliseconds
- At $t + dt$, we can predict pos = $X + Vdt + \frac{1}{2} Adt^2$
- By starting to render the world from the user's predicted head position, when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

Ex: The WebVR API returns predicted pose by default

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



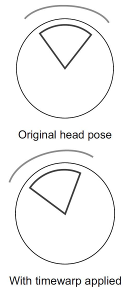


Image credit: Davis, Bryla, Benton, *Oculus Rift in Action* (2014)

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:

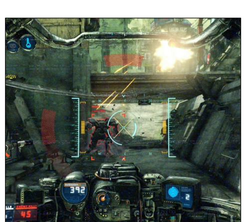
> The user is in control of the camera.

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)

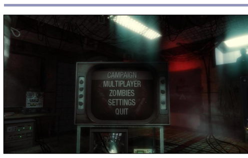
In-world UIs are evolving

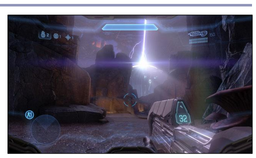
Deus Ex Human Revolution (2011)

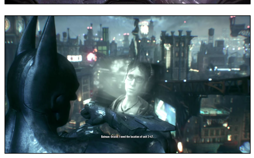
Deus Ex Mankind Divided (2016)

Increasingly, UI elements are being integrated into the virtual world

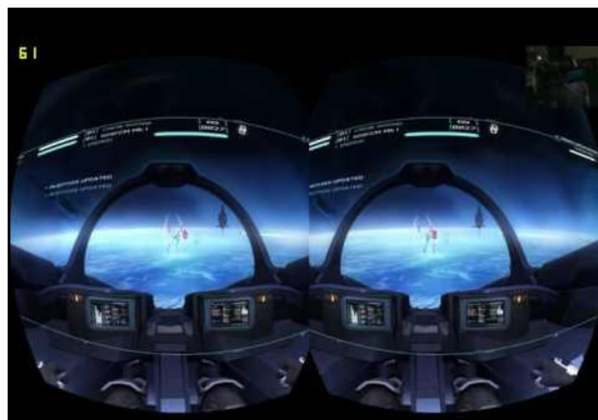
The best virtual UI is in-world UI







Top left: Call of Duty: Black Ops (2010)
Bottom left: Crysis 3 (2013)
Top right: Halo 4 (2012)
Bottom right: Batman: Arkham Knight (2015)



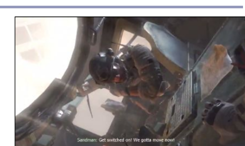
Strike Suit Zero (2013)

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.

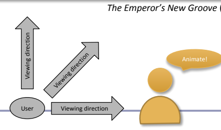
"It's a new communications medium. What is necessary is to develop a grammar and syntax. It's like film. When film was invented, no one knew how to use it. But gradually, a visual grammar was developed. Filmgoers began to understand how the grammar was used to communicate certain things. We have to do the same thing with this."
Neal Stephenson, *Interface*, 1994

Drawing the user’s attention

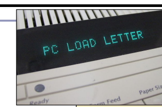
When presenting dramatic content in VR, you risk the user looking away at a key moment.

- 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.

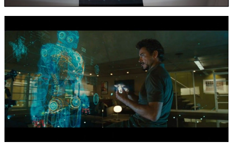
Abridged from *Usability Engineering* by Jakob Nielsen (Morgan Kaufmann, 1993)

Gestural interfaces

Hollywood has been training us for a while now to expect *gestural user interfaces*.

A gestural interface uses predetermined intuitive hand and body gestures to control virtual representations of material data.

Many hand position capture devices are in development (ex: Leap Motion)



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)

Appendices

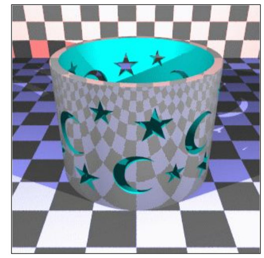
Additional topics of interest in computer graphics.
These slides are not examinable.

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

Appendix A: Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a ray-tracing technique which builds complicated forms out of simple primitives, comparable to (and more complicated than, but also more precise than) Signed Distance Fields.

These primitives are combined with the standard boolean operations: *union, intersection, difference*.

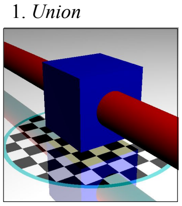


CSG figure by Neil Dodgson

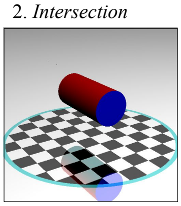
Constructive Solid Geometry

Three operations:

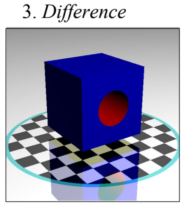
1. *Union*



2. *Intersection*



3. *Difference*



Constructive Solid Geometry

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

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

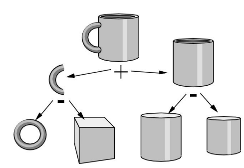
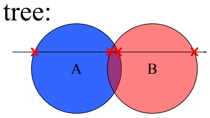


Figure from Wyvill (1995) part two, p. 4

Ray-tracing CSG models

For each node of the binary tree:

- Fire ray *r* at *A* and *B*.
- List in *t*-order all points where *r* enters of leaves *A* or *B*.
  - You can think of each intersection as a quad of booleans-- (*wasInA, isInA, wasInB, isInB*)



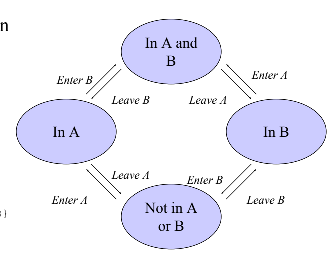
- Discard from the list all intersections which don't matter to the current boolean operation.
- Pass the list up to the parent node and recurse.

Ray-tracing CSG models

Each boolean operation can be modeled as a state machine.

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

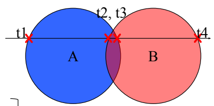
- Union: `{In A | In B | In A and B}`
- Intersection: `{In A and B}`
- Difference: `{In A}`



Ray-tracing CSG models

Example: Difference (A-B)

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

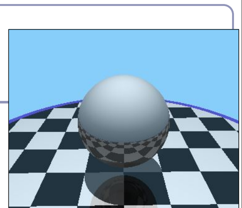


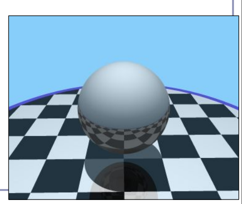

```
difference =
((wasInA != isInA) &&
 (!isInB)&&(!wasInB))
||
((wasInB != isInB) &&
 (wasInA || isInA))
```


Constructive Solid Geometry - References

- Jules Bloomenthal, *Introduction to Implicit Surfaces* (1997)
- Alan Watt, *3D Computer Graphics*, Addison Wesley (2000)
- MIT lecture notes: http://groups.csail.mit.edu/graphics/classes/6.837/F98/talecture/

Aliasing





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

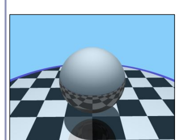
the misidentification of a signal frequency, introducing distortion or error.

"high-frequency sounds are prone to aliasing"

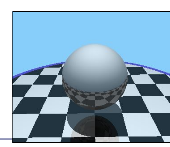
2. COMPUTING

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.

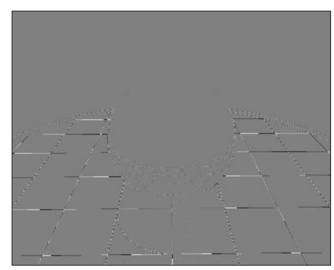
Aliasing



-



=



Antialiasing

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

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

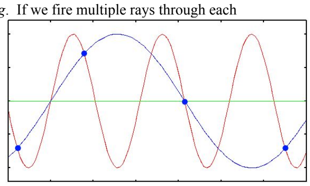


Image source: www.svi.nl

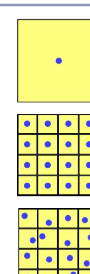
To avoid heavy computational load And also avoid sub-super-sampling artifacts, consider using *jittered super-sampling*.

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

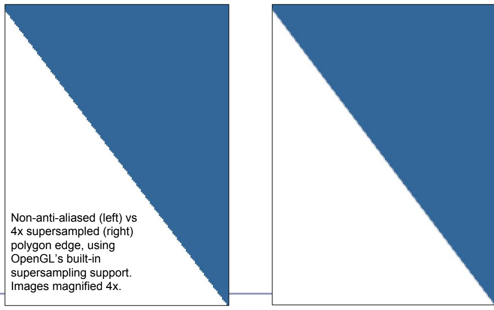
Antialiasing with OpenGL

Antialiasing remains a challenge with hardware-rendered graphics, but image quality can be significantly improved through GPU hardware.

- The simplest form of hardware anti-aliasing is Multi-Sample Anti-Aliasing (*MSAA*).
- "Render everything at higher resolution, then down-sample the image to blur jaggies"
- Enable MSAA in OpenGL with `glfwWindowHint(GLFW_SAMPLES, 4);`



Antialiasing with OpenGL: MSAA



Non-anti-aliased (left) vs 4x supersampled (right) polygon edge, using OpenGL's built-in supersampling support. Images magnified 4x.

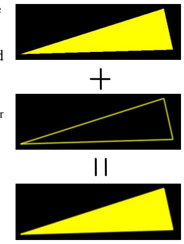
Antialiasing on the GPU

MSAA suffers from high memory constraints, and can be very limiting in high-resolution scenarios (high demand for time and texture access bandwidth.)

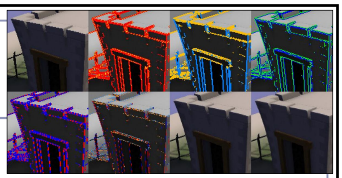
Eric Chan at MIT described an optimized hardware-based anti-aliasing method in 2004:

1. Draw the scene normally
2. Draw wide lines at the objects' silhouettes
   a. Use blurring filters and precomputed luminance tables to blur the lines' width
3. Composite the filtered lines into the framebuffer using alpha blending

This approach is great for polygonal models, tougher for effects-heavy visual scenes like video games



Antialiasing on the GPU



More recently, NVIDIA's *Fast Approximate Anti-Aliasing* ("FXAA") has become popular because it optimizes MSAA's limitations.

Abstract:

1. Use local contrast (pixel-vs-pixel) to find edges (red), especially those subject to aliasing.
2. Map these to horizontal (gold) or vertical (blue) edges.
3. Given edge orientation, the highest contrast pixel pair 90 degrees to the edge is selected (blue/green)
4. Identify edge ends (red/blue)
5. Re-sample at higher resolution along identified edges, using sub-pixel offsets of edge orientations
6. Apply a slight blurring filter based on amount of detected sub-pixel aliasing

Image from https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

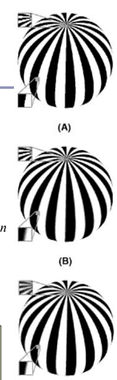
Preventing aliasing in texture reads

Antialiasing technique: *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 GLSL by the methods `dFdx(F)` and `dFdy(F)`.

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

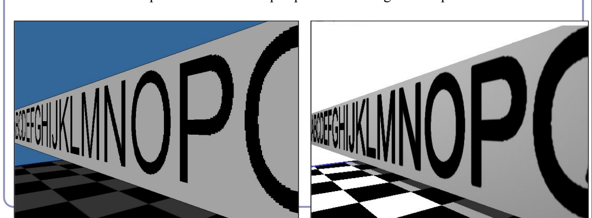


(A) Jagged lines visible in the box function of the procedural stripe texture
(B) Fixed-width averaging blends adjacent samples in texture space; aliasing still occurs at the top, where adjacency in texture space does not align with adjacency in pixel space.
(C) Adaptive analytic prefiltering smoothly 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.

Antialiasing texture reads with Signed Distance Fields

Conventional anti-aliasing in texture reads can only smooth pixels immediately adjacent to the source values.

Signed distance fields represent monochrome texture data as a distance map instead of as pixels. This allows per-pixel smoothing at multiple distances.

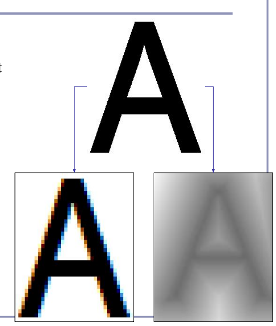


Antialiasing texture reads with Signed Distance Fields

The bitmap becomes a height map.

Each pixel stores the distance to the closest black pixel (if white) or white pixel (if black). Distance from white is negative.

3.6	2.8	2	1	-1
3.1	2.2	1.4	1	-1
2.8	2	1	-1	-1.4
2.2	1.4	1	-1	-2
2	1	-1	-1.4	-2.2
2	1	-1	-2	-2.8



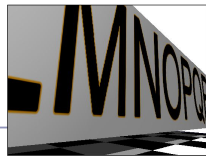
Antialiasing texture reads with Signed Distance Fields

Conventional bilinear filtering computes a weighted average of color, but an SDF computes a weighted average of distances.

This means that a small step away from the original values we find smoother, straighter lines where the slope of the isocline is perpendicular to the slope of the source data.

By smoothing the isocline of the distance threshold, we achieve smoother edges and nifty edge effects.

```
low = 0.02;    high = 0.035;
double dist =
  bilinearSample(tex coords);
double t =
   (dist - low) / (high - low);
return (dist < low) ? BLACK
   : (dist > high) ? WHITE
   : BLACK*(1 - t) + WHITE*(t);
```



Adding a second isocline enables colored borders.

Antialiasing - Interesting further reading

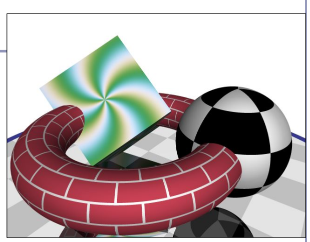
- https://people.csail.mit.edu/ericchan/articles/prefilter/
- https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf
- http://iryoku.com/aacourse/downloads/09-FXAA-3.11-in-15-Slides.pdf

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):

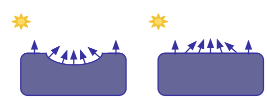


I've cheated slightly and multiplied the u coordinate by 4 to repeat the brick texture four times around the torus.

```
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
```


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.

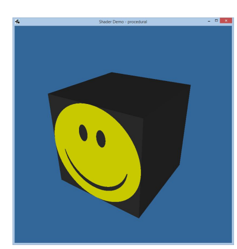
If we duplicate the normals, we don't have to duplicate the dent.

In a sense, the renderer 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

Procedural texturing in the fragment shader

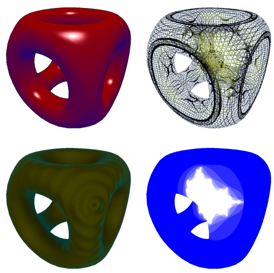
```
// ...
const vec3 CENTER = vec3(0, 0, 1);
const vec3 LEFT_EYE = vec3(-0.2, 0.25, 0);
const vec3 RIGHT_EYE = vec3(0.2, 0.25, 0);
// ...

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--check out the source on github for how I did the curved mouth and oval eyes.)

Advanced surface effects

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



Perlin Noise

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
- $f(P) = (X_p^2 + Z_p^2) \bmod 1$
- $color(P) = earlyWood + f(P) * (lateWood - earlyWood)$

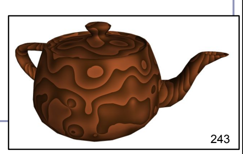
Adding realism

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

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

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

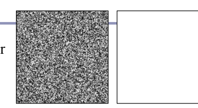
For natural-looking results, use *Perlin noise*, which interpolates smoothly between noise values.



Perlin noise

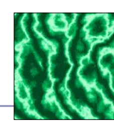
Perlin noise (invented by Ken Perlin) is a method for generating noise which has some useful traits:

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

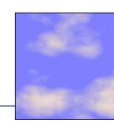


Non-coherent noise (left) and Perlin noise (right)
Image credit: Matt Zucker

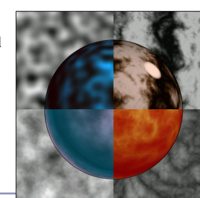
Matt Zucker



Matt Zucker



Matt Zucker



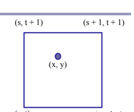
Ken Perlin

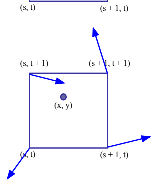
Perlin noise 1

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

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

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



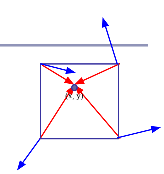


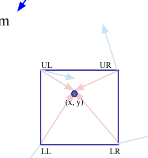
These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Perlin noise 2

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

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





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

Perlin noise 3

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

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

We interpolate along one axis first:

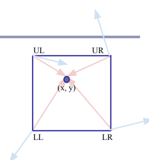
$L(x, y) = LL + S(x - floor(x))(LR - LL)$

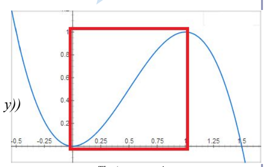
$U(x, y) = UL + S(x - floor(x))(UR - UL)$

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

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

Voila!





The 'ease curve'

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

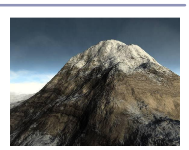
Perlin Noise - References

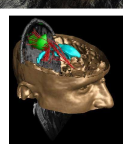
- https://web.archive.org/web/20160303232627/http://www.noisemachine.com/talk1/
- http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Voxels and volume rendering

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

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

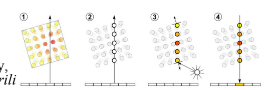




Volume ray casting

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

- Fire a ray through each pixel;
- Sample the voxel data along the ray, computing the weighted average (*trili* filter) of the contributions to the ray o1 each voxel it passes through or near;
- Compute surface gradient from of each voxel from local sampling; generate surface normals; compute lighting with the standard lighting equation;
- 'Paint' the ray from back to front, occluding more distant voxels with nearer voxels; this gives hidden-surface removal and easy support for transparency.

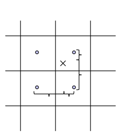


The steps of volume rendering; a volume ray-cast skull. Images from wikipedia.

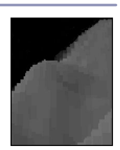
Sampling in voxel rendering

Why trilinear filtering?

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



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



Reasonably fast voxels

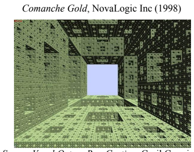
If speed is of the essence, cast your rays but stop at the first opaque voxel.

- Store precomputed lighting directly in the voxel
- Works for diffuse and ambient but not specular
- Popular technique for video games (e.g. Comanche →)


Comanche Gold, NovaLogic Inc (1998)

Another clever trick: store voxels in a *sparse voxel octree*.

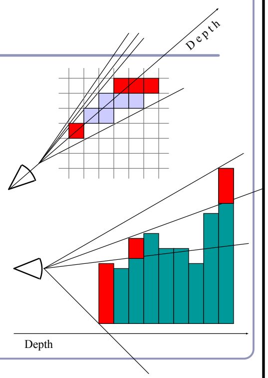
- Watch for it in id's next-generation engine…


Sparse Voxel Octree Ray-Casting, Cyril Crassin

Ludicrously fast voxels

If speed is essential (like if, say, you're writing a video game in 1992) and you know that your terrain can be represented as a height-map (ie., without overhangs), replace ray-casting with 'column'-casting and use a "*Y-buffer*":

- Draw from *front to back*, drawing columns of pixels from the bottom of the screen up. For each pixel in receding order, track the current max *y* height painted and only draw new pixels above that *y*. Anything shorter must be behind something that's nearer, and it's shorter; so don't draw it.



References

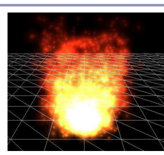
Voxels:
J. Wilhelms and A. Van Gelder, *A Coherent Projection Approach for Direct Volume Rendering*, Computer Graphics, 35(4):275-284, July 1991.
http://en.wikipedia.org/wiki/Volume_ray_casting

Particle systems

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

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

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



A particle system created with 3dengfx, from wikipedia.

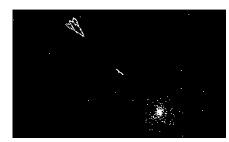
Screenshot from the game *Command and Conquer 3* (2007) by Electronic Arts; the "lasers" are particle effects.

History of particle systems

1962: Ships explode into pixel clouds in "Spacewar!", the 2^{nd} video game *ever*.

1978: Ships explode into broken lines in "Asteroid".

1982: The Genesis Effect in "*Star Trek II: The Wrath of Khan*".



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

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

Particle systems

How it works:

- Particles are generated from an *emitter.*
  - Emitter position and orientation are specified discretely;
  - Emitter rate, direction, flow, etc are often specified as a bounded random range (monte carlo)
- Time ticks; at each tick, particles move.
  - New particles are generated; expired particles are deleted
  - Forces (gravity, wind, etc) accelerate each particle
  - Acceleration changes velocity
  - Velocity changes position
- Particles are rendered.

Particle systems — emission

Each frame, your emitter will generate new particles.

Here you have two choices:

- Constrain the average number of particles generated per frame:
  - # new particles = average # particles per frame + rand() * variance
- Constrain the average number of particles per screen area:
  - # new particles = average # particles per area + rand() * variance * screen area

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

Particle systems — integration

Each new particle will have at least the following attributes:

- initial position
- initial velocity (speed and direction)

You now have a choice of integration technique:

- Evaluate the particles at arbitrary time *t* as a closed-form equation for a stateless system.
- Or, use iterative (numerical) integration:
  - Euler integration
  - Verlet integration
  - Runge-Kutta integration

Particle systems — two integration shortcuts:

Closed-form function:

- Represent every particle as a parametric equation; store only the initial position p_0, initial velocity v_0, and some fixed acceleration (such as gravity *g*.)
- $p(t)=p_0+v_0t+½gt^2$

No storage of state

- *Very* limited possibility of interaction
- Best for water, projectiles, etc—non-responsive particles.

Discrete integration:

- Remember your physics—integrate acceleration to get velocity:
  - $v' = v + a \cdot \Delta t$
- Integrate velocity to get position:
  - $p' = p + v \cdot \Delta t$
- Collapse the two, integrate acceleration to position:
  - $p'' = 2p' - p + a \cdot \Delta t^2$

Timestep must be nigh-constant; collisions are hard.

Particle systems—rendering

Can render particles as points, textured polys, or primitive geometry

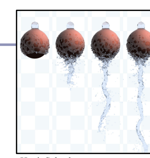
- Minimize the data sent down the pipe!
- Polygons with alpha-blended images make pretty good fire, smoke, etc

Transitioning one particle type to another creates realistic interactive effects

- Ex: a 'rain' particle becomes an emitter for 'splash' particles on impact

Particles can be the force sources for a blobby model implicit surface

- This is sometimes an effective way to simulate liquids



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

nvidia

References

Particle Systems:

William T. Reeves, "*Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*", Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83).

Lutz Latta, *Building a Million Particle System*, http://www.2ld.de/gdc2004/MegaParticlesPaper.pdf , 2004

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