2019

# Further Graphics

CAMBRIDGE COMPUTER SCIENCE TRIPOS PART IB, PAPER 7

ASHWIN AHUJA

## Table of Contents

# Ray-Based Rendering

## GPU Ray Tracing

Choose the colour of the pixel by firing a ray through and seeing what it hits

1. Use a minimal vertex shader (no transform) – 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. For each pixel, compute the ray from the eye through the pixel, using the interpolated coordinates to identify the pixel
    a. Run the ray tracing algorithm for every ray

```
// 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);
}
```

## Ray Marching

Alternative to classic ray-tracing – take a series of finite steps along the ray until we strike an object or exceed the number of permitted steps

- Scene objects answer: has this ray hit you?
- Good for sign distance fields
- But
  - Often involves too many steps
  - Too large a step size can lead to lost intersections
  - If() test in the heart of a for() loop is very hard for GPU to optimize

## 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 because the nearest intersection can be no closer than d

```
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;
}
```

## Combining SDFs

1. Union of two objects is taking min() of functions
2. Intersection of two SDFs by taking max() of their functions
3. A – B is max() of function A and negative of function B

4. **Interpolation**
    a. Simply taking the min, max, etc, this gives a sharp discontinuity. Interpolating the two SDFs with a smooth polynomial yields a smooth distance curve, blending the models
    b. **Example 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);
}
```

### Transforming SDF geometry

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

```
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);
}
```

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));
}
```

### SDF normal

To find the normal, you find the 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.

### SDF shadows

March a ray towards each light source – don't illuminate if the SDF ever drops too close to zero.

**Soft Shadows:** attenuate illumination by a linear function of the ray marching near to another object.

```
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;
}
```

### Repeating SDF Geometry

Take the modulus of a point's position along one or more axes before computing its signed distance, then segment the space into infinite parallel regions of repeated distance.

## Acceleration Structures

### Bounding Volumes

This is an optimization method for ray-based rendering. 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.

Good for:

1. Collision detection between scene elements
2. Culling before rendering
3. Accelerating ray-tracing and ray-marching

**Types of Bounding Volumes:** idea that speed trumps precision

1. Axis-aligned bounding boxes
2. Bounding spheres
3. Bounding cylinders

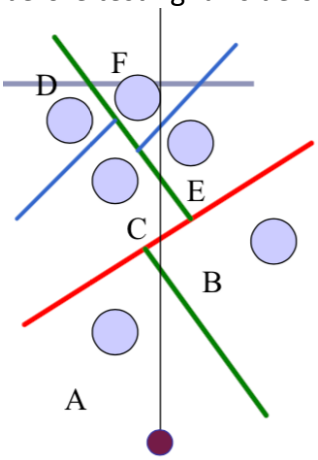
**Hierarchies of bounding volume:** allows discarding of rays that won't hit large parts of the scene. But, without spatial coherence ordering, the objects in a volume you hit, still have to hit-test every object.

### Subdivision of Space

Split space into cells and list in each cell, every object in the scene that overlaps that cell. This means the ray can skip empty cells but objects may overlap many filled cells or you may waste memory on many empty cells

### BSP Trees

BSP Tree pre-partitions the scene into objects in front of, on, and behind a tree of planes. This gives us an ordering in which to test scene objects against the ray – test all near-side objects before testing far-side objects.




However, (1) requires a slow pre-processing step, (2) strongly favours static scenes and (3) choice of planes is hard to optimise.

### kd-trees

This is a simplification of the BSD Tree data structure:

- Space recursively subdivided by axis-aligned planes and points on either side of each planes and points on either side of each plane are separated in the tree.
- Kd-tree has O(n log n) insertion time and $O(n^{2/3})$ search time
- Don't suffer from the mathematical slowdowns of BSPs because planes are always axis aligned

### Bounding Interval Hierarchies

Subdivides space around the volumes of objects and shrinks each volume to remove unused space – a best-fit kd-tree. It can be built dynamically as each ray is fired into the scene

## Implicit Surfaces

Method of producing very organic or bulbous surfaces very quickly without subdivision or NURBS

**Isoclines:** line on a diagram connecting points of equal gradient or inclination
**Isosurfaces:** three-dimensional analogue of an isocline – represents points of a constant value within a volume of space

### Modelling implicit surfaces

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. 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 solve:

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

### Force Functions

**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

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

### Rendering implicit surfaces

1. Render the surface directly to the GPU
    a. Realtime lighting, smooth surfaces, looks good
    b. Hard to integrate with other objects in scene
    c. 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
    a. Mesh can be manipulated
    b. Costly setup costs or runtime framerate hit
3. **With Signed Distance Fields – Blynn's Metaballs**
    a. 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;
    }
}
```

b. If we use Blynn's constants: a=1, b=3 – aim: answer question if F < 0.5, then we're outside surface. What is the minimum distance from our current position to F=0.5

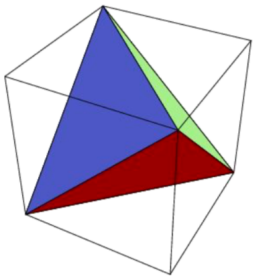
$$F = (3a/2)(1-r/b)^2$$
$$= (3/2)(1-r/3)^2$$
$$r^2 - 6r + (9-6F) = 0$$
$$r = 3 \pm \sqrt{(6F)}$$

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

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

```
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;
    }
}
```

4. **With Polygons**
   a. **Octree:** recursive subdivision of space which homes in on surface, from larger to finer detail
      i. Encloses a cubical volume in space **–** evaluate the force function at each vertex of the cube
      ii. As the octree subdivides and splits into smaller octrees and splits into smaller octrees, only octrees which contain some of surface are processed
   b. **Polygonising the surface**
      i. To display set of octrees – convert them into polygons
      ii. If corners above the force limit (hot) and others are below the force limit (cold) then the isosurface must cross the cube edges in between
      iii. Set of midpoints of adjacent crossed edges forms rings which can be triangulated. Normal is known from the direction of edges.
      iv. Can also discard any child whose vertices are all above or all below the force limit.
      v. **Ambiguities in cubes – fifteen possible configurations of hot / cold vertices in cube**
         1. Can overcome this by decomposing cube into tetrahedra – into 5 is the most common



2. Can also just do subdivision on tetrahedra

vi. **Smoothing polygonization**

1. Naïve implementation builds polygons whose vertices are the midpoints of the edges between hot and cold vertices
2. Can be better done using linear interpolation of relative values of the force functions

c. **Marching Cubes Algorithm**

i. Alternative if you only want to compute the final stage

1. Fire ray from any point known to be inside the surface
2. Find where ray crosses surface – using Newton's method or binary search
    a. Newton: derivative estimated from discrete local sampling
3. Drop cube around intersection point, it will have some vertices hot and some cold
4. While there exists a cube, which has at least one hot, and one cold vertex, create neighbouring cube

## Computational Geometry

**Closed Manifold Polygon Mesh:** Exactly two triangles meet at each edge – the faces meeting at each vertex belongs to a single, connected loop of faces

**Manifold with boundary:** At most two triangles meet at each edge – faces meeting at each vertex belong to a single, connected strip of faces

**Oriented Surface: iff**

1. Vertices of each face are stored in a fixed order
2. 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

**Embedded Surface:** nothing pokes through that is: 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
Closed, embedded surface must separate 3-space into two parts: a bounded interior and an unbounded exterior

### Normal at a vertex

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 -> 0

- Normal at any point on a face is a constant vector

- Normal to surface at any edge is an arc swept out on a unit sphere between the two normal of the two faces
- Normal to the surface at a vertex is a space swept out on the unit sphere between the normal of all of the adjacent faces

**Finding Normal:** take weighted average of the normal of surrounding polygons – weighted by each polygon's **face angle** (the angle α formed at the vertex v by the vectors to the next and previous vertices in the face F) at the vertex

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

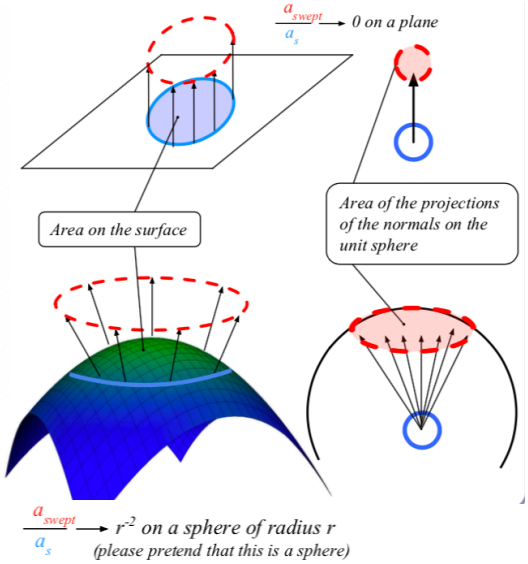
$$N(v) = \frac{\sum_F \alpha(F,v)\, N_F}{|\sum_F \alpha(F,v)|}$$

## Curvature

### Gaussian Curvature on Smooth Surfaces

Expresses how flat the surface isn't. We measure the directions in which the surface is curving most – these are the directions of principle curvature $k_1$ and $k_2$. The product of $k_1$ and $k_2$ is the scalar Gaussian curvature.

**Gaussian Curvature of a region on a surface:** Ratio between the area of the surface of the unit sphere swept out by the normal of that region and the area of the region itself. Gaussian curvature of a point if the limit of this ration as the region tends to zero area.



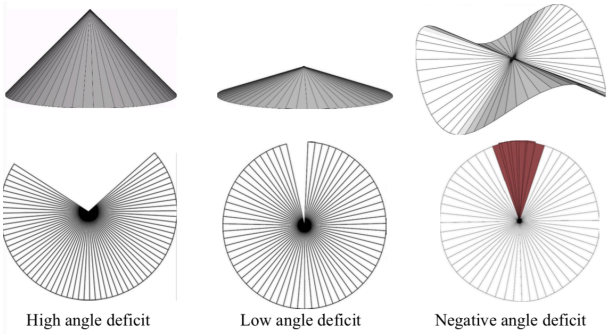

### Gaussian Curvature on Discrete Surfaces

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

### Angle Deficit

Angle Deficit of a vertex is defined to be two $\pi$ minus the sum of the face angles of the adjacent faces

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




## Genus

A topologically invariant property of a surface defined as the larger number of non-intersecting simple closed curves that can be drawn on the surface without separating it. Informally, it's the number of coffee cup handles in the surface.

## Poincare and Euler Characteristics

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

- V – the number of vertices of S
- E – the number of edges between the vertices
- F – the number of faces between the edges
- χ – Euler Characteristic of the surface

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

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

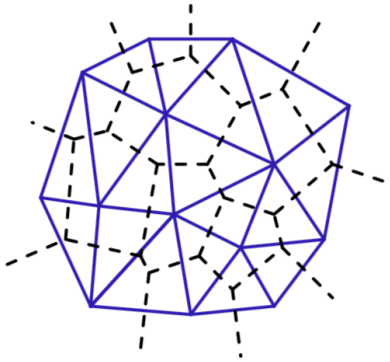
## Barycentric Coordinates

Coordinate system for describing the location of a point inside a triangle. The points are considered masses placed at each of the vertices of the triangle  st the centre of gravity of the triangle lies at P. ($t_A$, $t_B$, $t_C$) is proportional to the subtriangle areas of the three vertices – area of a triangle is ½ the length of the cross product of two of its sides.

```
// 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);
}
```

## Voronoi Diagram

Voronoi diagram of a set of points divides space into cells, where each cell contains the points in space closer to that point than any other point. The Delaunay triangulation is the dual of the Voronoi diagram: a graph in which an edge connects every point which share an edge in the Voronoi diagram.



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

**Formal Definition:** Given a set $S = \{p_1, p_2, \ldots, p_n\}$, a Voronoi cell $C(S, p_i) = \{p \in R^d \mid |p-p_i| < |p-p_j|, i \neq j\}$

**Voronoi point:** Where three or more boundary points – each point is at the centre 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.

### Equiangularity of a triangulation

Sorted list of the angles of the triangles – range of angles of the triangle

- Triangulation is said to be equiangular if it possesses larges equiangularity amongst all possible triangulations
- Delaunay triangulation is equiangular

**Empty Circle Property:** all Voronoi triangulation have this property that within any Voronoi triangulation of S, no point of S will lie inside the circle circumscribing any three points sharing a triangle in a Voronoi diagram.

### Delaunay Triangulation

- Border is always convex – whether in 2D, 3D, 4D, etc
- Delaunay triangulation of a set of points in $R^n$ is the planar projection of a convex hull in $R^{n+1}$

**Medical Axis of a surface:** set of all points within the surface equidistant to the two or more nearest points on the surface. Can be used to extract a skeleton of the surface.

### Finding Voronoi diagram

Four classes of algorithm for computing the Delaunay triangulation:

1. Divide and Conquer
2. Sweep plane – **Fortune's Algorithm**

   a. Algorithm maintains a sweep line and a beach line – set of parabolas advancing left-to-right from each. Beach line is the union of parabolas.
      i. Intersection of each pair of parabolas is an edge of a Voronoi diagram
      ii. All data to the left of the beach line is known – nothing to the right can change it
      iii. Beach line stored in binary tree
   b. Maintain a queue of two classes of event: the addition of, or removal of, a parabola
   c. O(n log n)
3. Incremental Insertion
4. Flipping – repairing an existing triangulation until it becomes Delaunay

**GPU Acceleration:** (1) For each pixel to be rendered on the GPU, search all the points for the nearest point – can be done in fragment shader **OR** (2) Render each point as a discrete 3D cone in isometric projection and let the z-buffering sort it out

## Bezier Curves and Surfaces

**Tensor Product:** Tensor product of two vectors is a matrix

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

**Continuity:** can be essential to the perception of quality

- C1 – continuous in both position and tangent vector
- G1 – continuous in position, tangent vector in same direction
- C0 – continuous in position only
- Cn (Mathematical Continuity) – continuous in all derivatives up to the $n^{th}$ derivative
- Gn (Geometric Continuity) – each derivate up to the $n^{th}$ has the same direction to its vector on either side of the join
- **Cn => Gn**

**Splin - Shipbuilding:** long, thin strips of wood or metal would be bent and held in place by heavy lead weights which acted as control points of the curve. Splines can be described by Cn-continuous Hermite polynomials which interpolate n+1 control points

### Beziers

Nested linear interpolations / weighted average between the control points

**General Formula:**

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

**Linear:** Linear interpolation between two points

$P(t) = (1-t)P_0 + tP_1$

**Quadratic:** Linear interpolation between two lines
$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$

### Bezier Cubic

$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$
$P_0$ and $P_3$ are the endpoints of the curve and $P_1$ and $P_2$ define the other two corners of the bounding polygon.
The curve fits entirely within the convex hull of $P_0$…$P_3$

**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
```

  - Issues
    - Cannot fix number of segments appropriate for all possible Beziers
    - Distance in real space, (x, y) is not linearly related to distance in parameter space, t
- **Adaptive Method** – subdivision
  - Check if a straight line between $P_0$ and $P_3$ is an adequate approximation to the Bezier
    - Need to specify some tolerance for when a straight line is an adequate approximation
  - If so, draw the straight line
  - If not, divide the Bezier into two halves, each a Bezier, and repeat for the two new Beziers

```
Procedure DrawCurve( Bezier curve )
VAR Bezier left, right
BEGIN DrawCurve
    IF Flat(curve) THEN // if P1 and P2 both lie within half a pixel
width of the line joining P0 to P3
        DrawLine(curve)
    ELSE
        SubdivideCurve(curve, left, right)
        DrawCurve(left)
        DrawCurve(right)
    END IF
END DrawCurve
```

  - Checking for flatness

$P(t) = (1\text{-}t)\,A + t\,B$

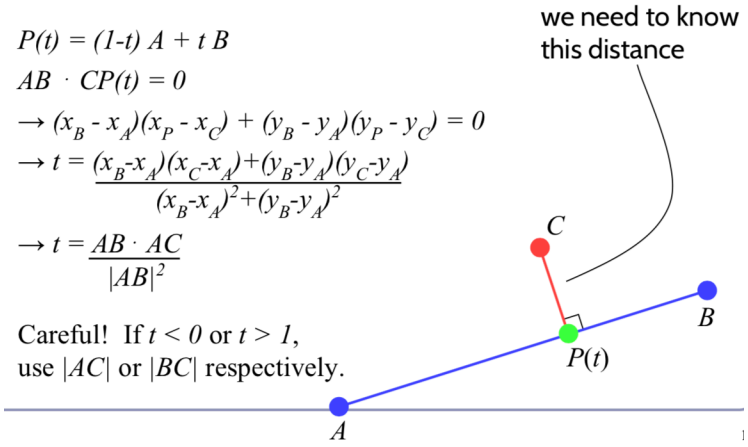
$AB \cdot CP(t) = 0$

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

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

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

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




- **Signed Distance Fields**
  - (1) Iterative implementation – SDF(P) = min(distance from P to each of the n line segments)
  - (2) Adaptive implementation – SDF(P) = min(distance to each sub-curve whose bounding box contains P) – can fast-discard sub-curves whose bounding box doesn't contain P

**Subdividing a Bezier cubic in two smaller Bezier cubics:**

$Q_0 = P_0$ $\qquad R_3 = \frac{1}{8} P_0 + \frac{3}{8} P_1 + \frac{3}{8} P_2 + \frac{1}{8} P_3$

$Q_1 = \frac{1}{2} P_0 + \frac{1}{2} P_1$ $\qquad R_2 = \frac{1}{4} P_1 + \frac{1}{2} P_2 + \frac{1}{4} P_3$

$Q_2 = \frac{1}{4} P_0 + \frac{1}{2} P_1 + \frac{1}{4} P_2$ $\qquad R_1 = \frac{1}{2} P_2 + \frac{1}{2} P_3$

$Q_3 = \frac{1}{8} P_0 + \frac{3}{8} P_1 + \frac{3}{8} P_2 + \frac{1}{8} P_3$ $\qquad R_0 = P_3$

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

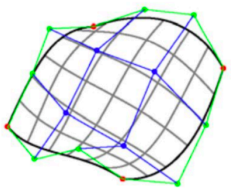
**Bezier Patches:**

- If a curve A has n control points and B has m control points, then A ⊗ B is an n x m matrix of polynomials of degree max(n-1, m-1)
- If you multiply this matrix against an n x m matrix of control points and sum them all up and have a bivariate expression for a rectangular surface patch in 3D – approach generalises to triangles and arbitrary n-gons
- It is defined by sixteen control points

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

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**
  - C0 – four edge control points must match
  - C1
    - Four edge control points must match

- Two control points on either side of the four edge control points must be colinear with both the edge point and each other, and be equidistant from the edge point

- G1
  - Four edge control points must match the relevant control points and be collinear

## 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. Has general form:

$$b_{v,n}(t) = \binom{n}{v} t^v (1-t)^{n-v}$$

Bernstein polynomials in $0 \le t \le 1$ always sums to 1:

$$\sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i}\, t^i = \big(t + (1-t)\big)^n = 1$$

## Overhauser's Cubic

A Bezier cubic which passes through four target data points – can calculate the appropriate Bezier control point locations from the given data points. This cubic interpolates it's controlling points which is good for animation

For example, given points A, B, C and D, the Bezier control points are:

P0 = B

P1 = B + (C – A)/6

P2 = C – (D – B)/6

P3 = C

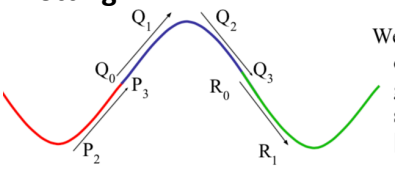
## Joining Bezier splines

**Simple Joining**

To set **C0** continuity, set $P_3 = Q_0$

To set **C1** continuity, require C0 and make the tangent vectors equal: set $P_3 = Q_0$ and $P_3 - P_2 = Q_1 - Q_0$

**Knotting**




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:

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

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

# Non-Uniform Ration B-Splines (NURBS)

NU: Non-Uniform: knots in the knot vector do not need to uniformly spaced
R: spline defined by rational polynomials
BS: B-Spline – generalisation of Bezier splines with controllable degree

NURBS are parametric – defined by:

1. Control points, $P_i$
2. NURBS basis functions, $N_{i,k}$

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

## Properties of NURBS curves

1. The basis functions must sum to 1

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

2. The basis functions are calculated from a **knot vector – a non-decreasing sequence of real numbers**
3. If the basis functions are Cm-continuous at t, then P(t) is guaranteed to be Cm-continuous at t – therefore continuity does not depend on the locations of the control points

## NURBS surfaces

Bivariate generalisation of the univariate NURBS curve

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

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

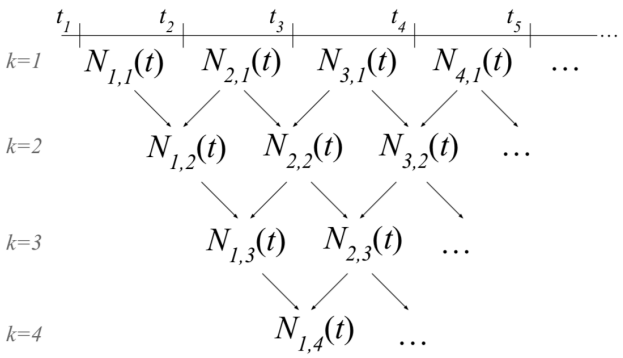
## B-Splines

d: degree of the curve
k = d + 1: parameter of the curve – number of control points which influence a single interval (eg cubic has four control points)
$\{P_1, \ldots, P_n\}$: list of n control points
$\{t_1, \ldots, t_{k+n}\}$: a knot vector of (k+n) parameter values
B-spline is $C^{k-2}$ continuous

**Basis Function:** Basis Function of control point $P_i$, $N_{i,k}(t)$ is defined recursively

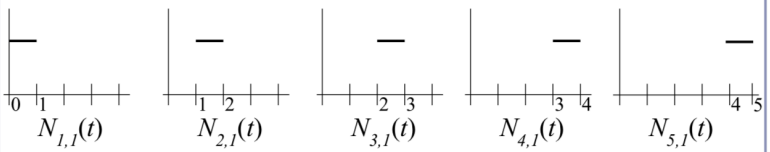
$$N_{i,1}(t) = \begin{cases} 1, t_i \le t < t_{i+1} \\ 0, \text{otherwise} \end{cases}$$

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




## B-Splines

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




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

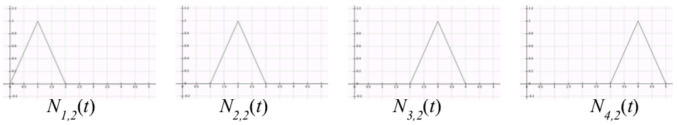
$N_{1,1}(t)=1,\ 0 \le t < 1$
$N_{3,1}(t)=1,\ 2 \le t < 3$
$N_{5,1}(t)=1,\ 4 \le t < 5$

$N_{2,1}(t)=1,\ 1 \le t < 2$
$N_{4,1}(t)=1,\ 3 \le t < 4$

Knot vector = {0,1,2,3,4,5}, $k = 1 \rightarrow d = 0$ (degree = zero) [11]

## B-Splines

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



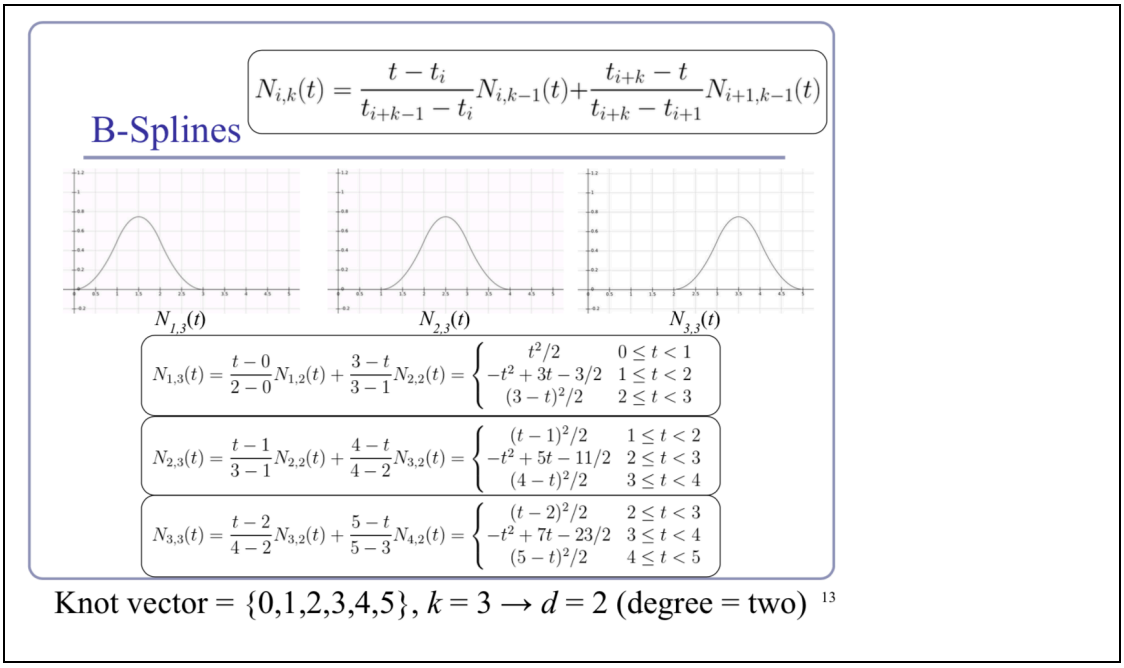

$$N_{1,2}(t) = \frac{t-0}{1-0} N_{1,1}(t) + \frac{2-t}{2-1} N_{2,1}(t) = \begin{cases} t & 0 \le t < 1 \\ 2-t & 1 \le t < 2 \end{cases}$$

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

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

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

Knot vector = {0,1,2,3,4,5}, $k = 2 \rightarrow d = 1$ (degree = one) [12]




**Uniformity:**

- Varying 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 only by that knot value – the spline will pass through the corresponding control point with C0 continuity

**Open vs Closed:**

- Knot vector which repeats its first and last knot values k times is called open, otherwise closed
- This is the only way to force the curve to pass through the first or last control point

## Homogenous Coordinates

$[x, y, z, w]_H$ -> $[x / w, y / w, z / w]$
OR
$[x, y, z, 1]$ -> $[xw, yw, zw, w]_H$

For control points: $P_i = (x_i, y_i, z_i)$ -> $P_{iH} = (x_i w_i, y_i w_i, z_i w_i)$
NURBS becomes:

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

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

**Piecewise Rational Curve** defined by:

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

with supporting *rational basis functions*:

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

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

## Subdivision Surfaces

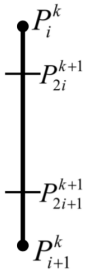
Problem is getting guaranteed continuity without having to build everything out of rectangular patches.

Instead of ticking parameter t along a parametric curve, 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.

**Schemes**

- **Univariate:** Scheme which describes a 1D curve
- **Bivariate:** Scheme which describes a 2D surface
- **Interpolating:** Scheme which retains and passes through its original control points
- **Approximating:** Scheme which moves away from its original control points, converging to a limit curve or surface

### Chaikin Curve Subdivision




On each edge, insert new control points at ¼ and ¾ between old vertices and delete the old points. The limit curve is C1 – quadratic B-spline

**Points between $P_i^k$ and $P_{i+1}^k$**

$$P_{2i}^{k+1} = \frac{3}{4} P_i^k + \frac{1}{4} P_{i+1}^k$$

$$P_{2i+1}^{k+1} = \frac{1}{4} P_i^k + \frac{3}{4} P_{i+1}^k$$

Where k is the generation – each generation will have twice as many control points as before

Terminal points are a special case

**Vector Notation:** Chaikin can be written in vector notation as:

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

**Standard notation**: compresses the scheme to a kernel – interlaces the odd and even rules.

- $h = (1/4)[\ldots,0,0,1,3,3,1,0,0,\ldots]$
- It also makes matrix analysis possible – eigenanalysis of the matrix form can be used to prove the continuity of the subdivision limit surface

Consider the kernel

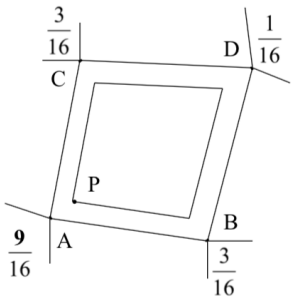
$h=(1/8)[\ldots,0,0,1,4,6,4,1,0,0,\ldots]$

You would read this as

$$P_{2i}^{k+1} = (1/8)(P_{i-1}^{k} + 6P_{i}^{k} + P_{i+1}^{k})$$
$$P_{2i+1}^{k+1} = (1/8)(4P_{i}^{k} + 4P_{i+1}^{k})$$

## Doo-Sabin

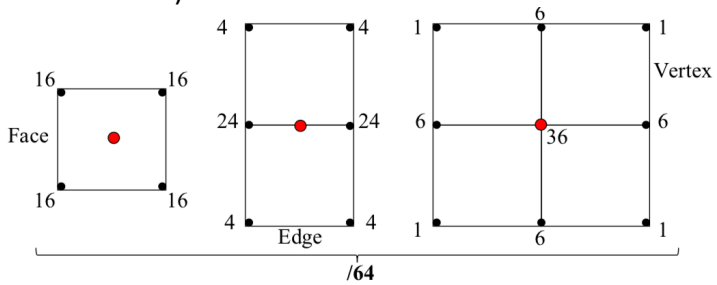
Takes Chaikin to 3D – replaces every old vertex with four new vertices. The limit surface is biquadratic (C1 continuous everywhere)




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

## Catmull-Clark

Bivariate approximating scheme with kernel h = (1/8)[1, 4, 6, 4, 1] – limit surface is bicubic (C2 continuous)

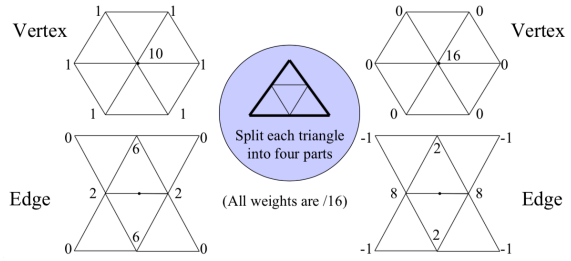



$$\frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix} \otimes \frac{1}{8}\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix} = \frac{1}{64}\begin{bmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{bmatrix}$$

Vertex rule · Face rule · Edge rule

## Schemes for Simplicial (Triangular) Meshes

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

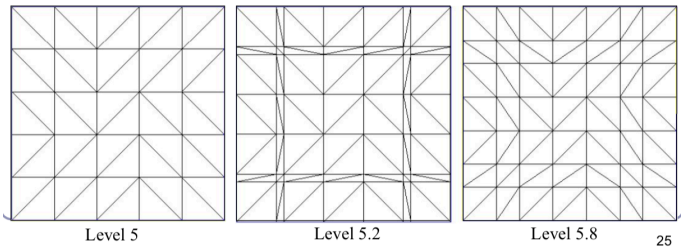



**Creases:** Extensions exist for most schemes to support creases, vertices and edges flagged for partial or hybrid subdivision

**Splitting subdivision surface**

- Algorithms rely on subdividing surface and examining the bounding boxes of smaller facets – rendering, ray / surface intersections
- Not enough to delete half control points – limit surface will change, therefore need to include all control points from the previous generation, which influence the limit surface in this smaller part

**Continuous Level of Detail – required for live applications – found as a function of distance**




25

## Extraordinary Vertices

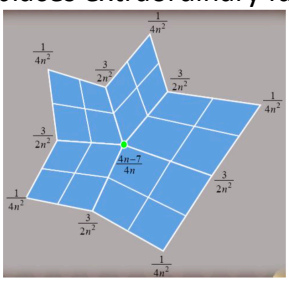
Catmull-Clark and Doo Sabin both operate on quadrilateral meshes – all faces have four boundary edges and all vertices have four incident edges – items which don't agree with this are **extraordinary**

**Extraordinary Vertices and Faces:** For many schemes, adaptive weights exist which can continue to guarantee at least some degree of continuity, but not always possible

**Catmull-Clark** replaces extraordinary faces with extraordinary vertices.

Catmull-Clark vertex rules generalized for extraordinary vertices:

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



**Doo-Sabin** replaces extraordinary vertices with extraordinary faces

### Bounding boxes and convex hulls (for subdivision surfaces)

Limit surface is the weighted average of the weighted averages of the weighted … of the original control points => for a scheme where all weights are positive and sum to one, the limit surface lies entirely within convex hull of the original control points.

For schemes with negative weights:

- $L = \max_t \sum_i |N_i(t)|$ be the greatest sum throughout parameter space of the absolute values of the weights
- For a scheme with negative weights, L exceeds 1
- Then limit surface must lie within convex hull of the original control points, expanding unilaterally by a ratio of (L-1)

### Summary of Subdivision Schemes

<u>Approximating</u>

- Quadrilateral
  - (1/2)[1,2,1]
  - (1/4)[1,3,3,1] (Doo-Sabin)
  - (1/8)[1,4,6,4,1] (Catmull-Clark)
  - *Mid-Edge*
- Triangles
  - Loop

<u>Interpolating</u>

- Quadrilateral
  - *Kobbelt*
- Triangle
  - Butterfly
  - *"√3" Subdivision*

## Global Illumination

### Classic Lighting Model

- Soft shadows are expensive
- Shadows of transparent objects require further hacks
- Lighting off reflective objects follows different shadow rules from normal lighting
- Hard to implement diffuse reflection (colour bleeding – Cornell Box)
- Ambient term is a hack **AND** diffuse term is only one step in what should be a recursive, self-reinforcing series

### Ambient Occlusion

**Ambient Illumination:** blanket constant that we often add to every illuminated element in a scene, to (inaccurately) model the way in which light scatters off all surfaces, illuminating area not in direct lighting

**Ambient Occlusion:** 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

#### Theory

Treat the background (sky) as a vast ambient illumination source:

1. For each vertex of a surface, compute how much background illumination reaches the vertex by computing how much sky it can see

2. 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 $\omega$ |
| $\Omega$ | integrate over area (hemisphere) |

This approach is very flexible – but very expensive. We speed it up by randomly sampling rays cast from evert polygon or vertex – Monte-Carlo Method. Otherwise, we render the scene from the point of view of each vertex and count the background pixels in the render.

**Pre-Compute Per-Object Occlusion Maps:** Texture maps of shadow to overlay onto each object – but pre-computed maps fare poorly on animated models

### Screen Space Ambient Occlusion (SSAO)

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 GPU

**Process**

1. For each visible point (pixel) on a surface in the scene, take multiple samples (8-32) from nearby and map these samples back to screen space
2. Check if the depth sampled at each neighbour is nearer to, or further from, the scene sample point
3. If the neighbour is nearer than the scene sample point, then there is some degree of occlusion – make sure not to occlude if nearer neighbour is too much nearer than the scene sample point.
4. Sum retained occlusions, weighting with an occlusion function

### Implementation using Signed Distance Fields

SSAO is trivial with signed distance fields:

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

### Radiosity

Illumination method which simulates the global dispersion and simulates the global dispersion and reflection of diffuse light – breaks the scene into many small elements and calculates the energy transfer between them.

### Algorithm

1. Divide surfaces divided into patches, small subsections of each polygon or object

2. For each pair of patches, compute **view factor –** describe how much energy from one reaches the other
    a. Further they are in space or orientation, less light they shed on each other => lower view factors
3. Calculate lighting of all directly-lit patches
4. Bound light from all lit patches to all those they light, carrying more light to patches with higher relative view factors. Repeating this step distributes the total light across the scene, producing a global diffuse illumination model.

### Radiosity of single patch

Amount of energy leaving the patch per discrete time interval – total light being emitted directly from the patch combined with the total light being reflected by the patch

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

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 $j$.

### Form factors

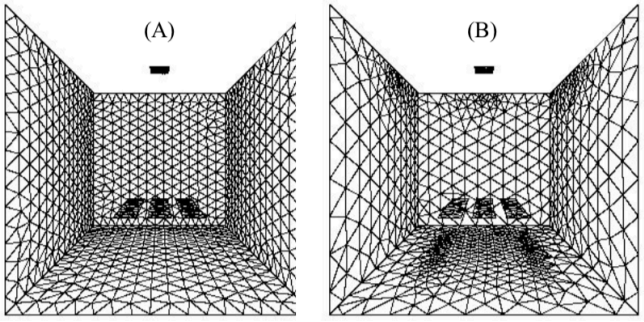
Can be found procedurally or dynamically

1. Subdivide every surface into small patches of similar size
2. Can dynamically subdivide wherever first derivative of calculated intensity rises above some threshold

Computing cost for a general radiosity solution $\propto$ number of patches squared => try to keep number of patches down. Align with lines of shadow for reduced computation cost.

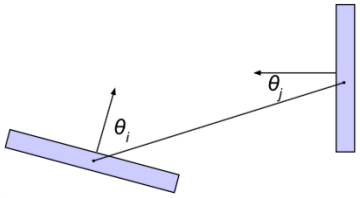
### Implementation

**1. Patch Creation**
    a. (1) Simple patch triangulation
    b. (2) Adaptive patch generation – the floor and walls of the room are dynamically subdivided to produce more patches where shadow detail is higher

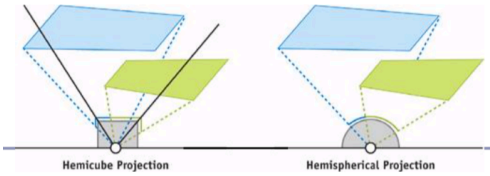



2. **View Factors**
    a. View factor between patches i and j if Fi -> j
    b. $Fi \rightarrow j = \frac{cos\ \theta_i\ cos\ \theta_j}{\pi r^2} V(i,j)$

c. Where $\Theta_i$ is the angle between the normal of the patch I and the line of the 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)




3. **Visibility**
    a. Calculating V(I, j) is slow
    b. Speed it up with a hemicube (in which each form factor is encased in a hemicube).
    c. Scene rendered from point of view of the patch through the walls of the hemicube
    d. V(I, j) is computed for each patch based on which patches it can see in its hemicube




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.

## Shadows, Refraction and Caustics

**Problems:**

1. Shadow ray strikes transparent, refractive object
    a. Refracted shadow ray now misses the light
    b. This destroys the validity of the Boolean shadow test
    **c. Solutions**
        **i.** Backwards ray tracing – computationally heavy but improved by stencil mapping
        **ii.** Shadow attenuation – low refraction, no caustics
        **iii. Photon Mapping**
2. Light passing through a refractive object sometimes forms caustics – artefacts where the envelope of a collection of rays falling on the surface is bright enough to be visible

## Photon Mapping

Process of:

1. Emitting photons into a scene and tracing their paths probabilistically to build a **photon map –** this data structure describes the illumination of the scene independently of its geometry.
2. Data then combined with ray tracing to compute the global illumination of the scene

**Photon Map:** must support fast insertion and fast nearest-neighbour lookup – kd-tree is often used

**Probabilism:** Monte-Carlo integration – simulated by randomly sampling values from within the integral's domain until enough samples average out to about the right answer

- **(1)** Initial photon direction random – constrained by light shape, but random

- **(2)** When photon hits a solid, also has random component based on (1) diffuse reflectance, (2) specular reflectance and (3) transparency of the surface
- **(3)** Compute probabilities: $p_d$, $p_s$ and $p_t$ – gives a probability map and choose a random value in those probabilities
- **(4)** This determines whether photon is reflected, refracted or absorbed

### Algorithm

1. **Photon Scattering**
    a. Photons fired from each light source, scattered in randomly chosen directions
        i. Number of photons per light is a function of its surface area and brightness
    b. Photons fire through scene – where they strike a surface, they are either absorbed, reflected or refracted
    c. When energy is absorbed, cache location, direction and energy of the photon in the photon map
2. **Rendering**
    a. Ray trace the scene from the point of view of camera
    b. For each first contact point P (1) use ray tracer for specular but (2) compute diffuse from the photon map and (3) do away with ambient
    c. Radiant illumination = sum of contribution along the eye ray of all photons within a sphere of radius r of P
    d. Caustics calculated directly from the photon map – caustic map usually distinct from radiance map

## Virtual Reality

**Immersion:** art and technology of surrounding the user with a visual context, such that there's world above, below, and all around them
**Presence:** 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.
**Sword of Damocles (1968):** first head-mounted display

**Developing for VR**

- Dedicated SDKs
    - HTC Vive
    - Oculus Rift SDK: C++ with bindings for Python and Java
    - Google Daydream SDK: Android iOS and Unity
    - Playstation VR
- General-purpose SDKs
    - WebGL
    - WebVR API
- Higher-level game development
    - Unity VR

### Distance and Vision

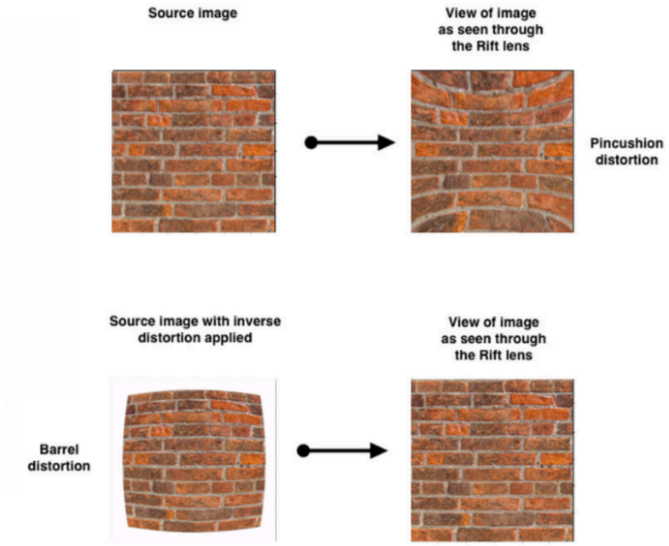
How to detect depth?

1. Binocular vision: ocular convergence and shadow stereopsis
    a. VR headsets work by presenting similar, but different views to each eye

2. Perspective
3. Parallax motion and occlusion
4. Texture, lighting and shading
5. Relative size and position and connection to the ground

## Lens Effects

Lenses in VR headset warp the image on the screen, creating a **pincushion distortion** – countered by introducing a **barrel distortion** in the GPU shader used to render the image => barrel-distorted image stretches back to full-size when seen through headset lenses




## Sensors

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

## Sensor Fusion

- **Issue:** Position drift – due to inaccurate sensor readings.
- **Solution:** Advanced headsets also track position with separate hardware on the user's desk or walls
    - **Oculus Rift:** Constellation – desk-based IR
    - **HTC Vive:** Base station units
    - **Playstation VR:** LEDs captured by camera

## Latency

Rendered image must respond to changes in head pose faster than the user can perceive – approximately 20ms, so no HMD can have framerate below 50Hz. Any loss of frames can lead to judder – nausea and hate.

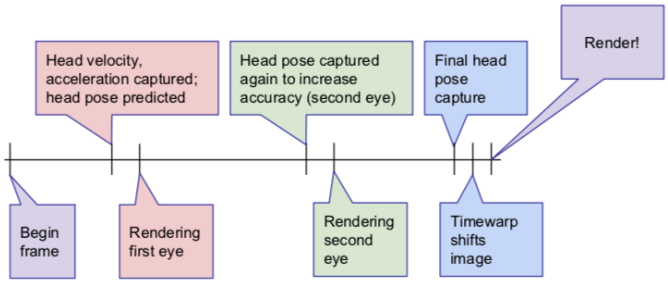
**Dealing with Latency:**

1. **Sensor Prediction:** predict the future basis – allow software to optimize rendering
    a. At time t, head pos = X, head velocity = V, head acceleration = A
    b. Human heads do not accelerate very well

c. Rendering a single frame takes dt ms
d. At t + dt, predict: pos = X + Vdt + ½ $Adt^2$
e. By rendering the world from the user's predicted head position, when rendering is complete, it aligns with where the head is by then

**2. Asynchronous Timewarp**

a. 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




## Sim Sickness

The feeling when body things it is sitting still, but eyes think the body is moving.

## Reducing Sim Sickness

**User is in control of the camera**

1. Head-tracking control is always with user
2. Head-tracking must match the user's motion
3. Avoid moving the user without direct interaction
4. If moving the user, make sure it doesn't break presence.

## User Interface

**1. UI rules**

a. Never mess with the field of view
b. Don't use head bob
c. Don't knock the user around
d. Offer multiple forms of camera control: look direction, mouse + keyboard, gamepad
e. Try to match in-world character height and inter-pupillary distance (IPD) to that of the user
f. Where possible, give user a stable in-world reference frame that moves with them
g. **Broken classic UI paradigms**
   i. UI locked to sides or corner distorted by lenses and harder to see
   ii. Floating 3D dialogs create a virtual plane within a virtual world – breaking presence
   iii. Small text is impossible to read in VR

2. **VR world**

a. Limit sidestepping, backstepping, turning – never force user to spin
b. Move at real-world speeds (1.4 m/s walk, 3 m/s run)
c. Ramps not stairs
d. Keep horizon line consistent, static and constant
e. Avoid very large moving objects which take up most of the field of view
f. Darker textures

g. Give user an avatar – react to user motion to give illusion of proprioception
h. Build in breaks

## Storytelling in Games

Can't use traditional methods, such as moving the camera, because the user is the camera and you don't want to move them. Therefore, needs to develop a new system of storytelling.

**Presenting VR dramatic content:** big issue is user looking away at a key moment

- Use audio cues, movement or changing lighting or colour to draw focus
- Use other characters in the scene – all turn to look at something
- Design scene to direct the eye – this can be dynamic since we know when the key content is in the viewing frustrum

## UI Advice

**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 User Interfaces:** Uses predetermined intuitive hand and body gestures to control virtual representations of material data.