Dot Product: p.q = elementwise multiplic-sum
Cross Product: p x q = (for 3d vectors): make an ijk row in a 3x3 matrix, other two rows are p and q, and find the determinant.
For any two vectors: u x v = -v x u
Useful property: The cross between two vectors is perpendicular to both.
Outer/inner surface normal: The normal which points out/in of the surface. We compute it by finding the normal and then seeing if the angle between it. Acute – inner; obtuse outer.

1) Transformations and Projections

Low level primitives, setPixel and drawLine give device-specific pixel renderng. We need device independency. Graphics APIs provide this uniformity:
1) World Coordinate System: we render a window, and then everything within it, and don't render anything else. We do this like so:

1.1) Normalisation:

- Make a syscall to find view window corners.
- Translate world coordinates to viewspace coordinates:

$$\frac{(X_w - W_{xmin})}{(W_{xmax} - W_{xmin})} = \frac{(X_v - V_{xmin})}{(V_{xmax} - V_{xmin})}$$

- We can rearrange and simplify this as $X_v = AX_w + B$, $Y_v = CY_w + D$, where A, B, C, D are constants from discovered values of W_{xmin}, V_{xmin}.

1.2) Polygon Rendering:

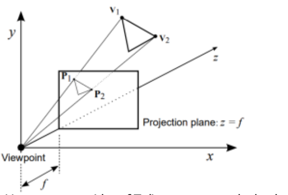
Graphics apps often use scenes built out of planar polyhedra – 3D objects whose faces are all planar polygons. To represent these we use:
- Numerical data – 3D coords of vertices
- Topological data – what's connected to what (index 0 – representing the 0^{th} face: (0, 1, 3) – suggests the face has vertices (0, 1, 3)).

1.3) Projections

1) The Orthogonal Projection:
- Viewpoint is at Z = -inf, plane of proj is z = 0

So, all projectors have same direction d = (0, 0, -1)
Each projection line has equation **P = v + μd**
Calculating the orthogonal projection:
1) Substitute d = $(0, 0, -1)^T$ into the projector vector equation: **P = v + μd**
2) This gives us cartesian equations for each component $P_x = V_x + 0$, $P_y = V_y + 0$, $P_z = V_z - \mu$
Since the plane of projection z = 0, $P_z = 0$.
So, the projected location on the screen is
P = $(V_x, V_y, 0)^T$ ie: just take the 3D x & y components.
*This projection is fine when we aren't concerned with depth, but for applications like Games, we are and thus we use the **Perspective Projection**.*

2) The Perspective Projection



Here we use an idea of Z distance to encode depth. $P_x = fV_x / V_z$, $P_y = fV_y / V_z$. For example:
V = (10 10 10), z = 5.
1) **Perspective**: f = z. So P_x = 5 10/10 = 5. P_y = 5 10/10 = 5. Thus, point is (5, 5, 5).
2) **Orthographic**: (10, 10, 0) (take x & y).

1.4) Transformations

To make Graphics Scenes easy to draw, set viewpoint at the origin, and the z-axis as viewdir.
In particular, ALWAYS translate things to the origin and do the transformation there, and then bring them back !!

1) Scaling: (e.g, scale by 2):

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

2) Translation: We use **homogenous coords** (add 4^{th} element **s** that acts as a scaler). Translation by a, b, c in dimensions x, y, z respectively:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

To translate **homogenous coords** to cartesian, take the x, y, z components but divide them by **s**
Note: **Affine Transformations**: transformations that preserve parallel lines. Transformations composed of scaling, rotation and translation are affine.

3) Composite Transformations
If we want to make an object centred at the origin wice as big and put its new centre at (5, 5, 20), first scale then translate. Note: MatMul not commutative!
4) Rotations: Defined by an angle, and the axis about which we rotate. **Left hand rule! Rotations anticlockwise!**

Every transformation shown is easily invertible.

$$\mathcal{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{R}_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{R}_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2) Animation Transformations: Flying Sequences

For Flying Animations, we must move the viewpoint around the screen, requiring change of origin:
- Take the required viewpoint as C = (C_x, C_y, C_z)
- Required view direction as d = $(d_x, d_y, d_z)^T$

The required transformation has 3 steps:
1) **Translate Point C towards the viewpoint** (translate by -(C_x, C_y, C_z)), make translation mat A.
2) **Rotate about y until d is in the y-z plane:**
$||v|| = v = (d_x^2 + d_z^2)^{1/2}$
$\cos\theta = d_z / v$, $\sin\theta = d_x / v$;

$$\mathcal{B} = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} d_z/v & 0 & -d_x/v & 0 \\ 0 & 1 & 0 & 0 \\ d_x/v & 0 & d_z/v & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We usually treat y dir as the vertical, and thus do this rotation first
3) **Rotate about x until d points along z axis:**
$||v|| = v = (d_x^2 + d_z^2)^{1/2}$
$\cos\theta = v / |d|$, $\sin\theta = d_y / |d|$

$$\mathcal{C} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & v/|d| & -d_y/|d| & 0 \\ 0 & d_y/|d| & v/|d| & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The overall transformation matrix: T = CBA.
EXAMPLE QN: Rotation about a general line:
Achieved by:
1. Making the line of rotation one of the Cartesian axes
2. Doing the rotation (about the chosen axis)
3. Restoring the line to its original place

This results in T = $A^{-1}B^{-1}C^{-1}R_z$ CBA
Object shrinking can be done similarly:
1. Move the object to the origin
2. Apply the scaling matrix
3. Move the object back

T = $A^{-1}B^{-1}C^{-1}$S CBA
AKA: replace the middle bit with the composite transformations we want to do; CBA is the matrix taking us to the origin, inverse takes us back.

2.1) Spaces:

Object Space: The local coordinate system of an object. Defines geometry of objects for transforming
World/Model Space: Global coordinate system encompassing all objects, providing a reference of space between objects allowing for the positioning and orienting of them, and for transforming them.
View/Camera Space: The coordinate system relative to the viewpoint / camera of the 3D scene. Defines the perspective from which we view the scene allowing for transformations mimicking movement and orientation
Projection Space: The final space on which the image is broadcast; the coordinate system mapping the 3D scene to a 2D flat image screen. Rendering and perspective effects and depth projection is done

2.2) Projection

Projection and drawing usually comes after the (world/object space) transformations.
1) Orthographic Projection Matrix:
For canonical orthographic projection, drop z coord (including for homogenous coordinates).
2) Perspective Projection of homogenous coordinates: We also do it with Mat Multiplication.

$$\mathcal{M}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix} \quad \mathcal{M}_p \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/f \end{pmatrix}$$

Normalise by just dividing the final vec by 4^{th} elem.
Projection Matrices are singular. Can't invert which makes sense – you can't recover a 3D scene from a 2D image of it!
To find if something is outer surface normal: find a normal, then find the angle between it and a point on the surface. Outer=deg>90

2.3) Structure and Characteristics of a Transformation Matrix

Types of Homogenous Coordinate:
1) Position Vectors [x y z s]T(4^{th} elem > 0, can be normalised to cartesian)
2) Direction vectors [x y z 0]T (Direction & Mag)
d+d=d d+p=p p+p=midpoint (as 4^{th} elem= s_1+s_2)
- Transformation Mats have bottom row [0 0 0 1]
- Columns of a transformation matrix are 3 direction vectors and 1 position vector
- Direction vectors in the trans mat are not affected by translation; position vectors are displaced equally.
- If we don't shear, the three dir vecs are orthogonal.
- The three direction vectors represent the transformed x, y and z axis respectively. The final column represents the transformed origin.

Given a view direction d and location C we can compute the matrix that moves the scene to that coordinate system by using dot products as projs:
Dot Product: p · u = |p||u|cosθ
- If u is a unit vector then p·u = |P| cos θ
- If u is along a coordinate axis then p·u is the ordinate of p in the direction of u (ie: how far along p goes across axis u).
- A change of axis can thus be represented as projections using dot products:

Given a new origin with pos vec C, and axis u,v,w, and a point P in the x,y,z axis system:
P_x^t = (P-C)·u = P·u - C·u P_y^t = (P-C)·v = P·v - C·v
P_z^t = (P-C)·z = P·z - C·z
In matrix form to transform to a **new axis/space:**

$$\begin{pmatrix} P_x^t \\ P_y^t \\ P_z^t \\ 1 \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -\mathbf{C}\cdot\boldsymbol{u} \\ v_x & v_y & v_z & -\mathbf{C}\cdot\boldsymbol{v} \\ w_x & w_y & w_z & -\mathbf{C}\cdot\boldsymbol{w} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

(Remember Camera Pos was neg last col!)
We now get to control vertical too, assume the v direction is vertical and constrain it in the software to point upwards.
EXAMPLE QN: Drawing Flying Scenes
Given a viewpoint C, viewdir d, need to find the trans mat for the canonical view: e.g: find vectors u,v,w.
1) d is direction of new z-axis; **w = d / |d|**
2) We can write u in terms of a horiz vec p: u=p/|p| p_y = 0, it has no vertical.
3) We can write q in terms of a vertical v: v=q/|q| q_y = 1.
SOLVING THIS: we have 4 unknowns:
p = [p_x, 0, p_z] and q = [q_x, 1, q_z]
d = p x q
Doing this yields: p = (d_z0 –d_x), since p.q = 0: p_xq_x – p_zq_z = 0, d_y = p_zq_x - p_xq_z
Since we know what p_x, p_z are we simply just solve those two linear equations for q. Then we form the final transformation mat as above in 2.3

3) Clipping: eliminate things outside of the view frustrum

We could clip at various points:
- *Before the perspective transform in 3D Space:* We use the equation of 6 planes of the bounding box. Natural, avoids degenerate cases.
- *In homogeneous coordinates after projection*
- Before perspective divide (cartesianization – so we have 4D space, with w vals). Camera indep, simple.
- *In the transformed 3D screen space after perspective division* problem - objects in the plane of the camera

2D plane equation of a halfspace: Ax+By+C=0
3D plane equation: Ax+By+Cz+D=0
This plane eq links to homogenous coords, consider the vector H=(A,B,C,D). Each point (x,y,z,w) has an infinite num of equivalent homogenous points (sx,sy,sz,sw), similarly we have infinite plane eqs s(Ax+By...) -> H = (sA, SB...)
Calculating Point to Plane Distance:
1) Scale H to normalise s.t: $A^2+B^2+C^2$=1
Then, the distance d = H·p = H^TP
This projects d onto the plane normal H.
d is signed, positive -> inside clipping plane, neg not.
Normalising isn't needed but it is good practice.

3.1) Clipping w.r.t the View Frustum

We need to test point p against each of the 6 planes.
For each H, if H·p < 0, cull.
By defining the closest face of the near plane's bottom left point as (left, bottom, near), and the top right as (right, top, near), we can define the entire set of plane normal in terms of this:
For example to get H_{bottom} (normal of bottom plane): cross two points on it: (l b n) x (r b n) => since r-l = 1; we get (0, n, -b, 1)T in homogenous terms.
In the end: H_{near} = (0 0 1 –near)T, H_{far} = (0 0 -1 far)T, H_{bottom} = (0 near –bottom 0)T, H_{top} =(0 –near top 0)T, H_{left} = (-near 0 left 0)T, H_{right} = (near 0 –right 0)T.

3.2) Clipping Line Segments

Line eq: L(u) = p_0 + u(p_1 – p_0) OR up_1 + (1-u)p_0
Set this equal to the plane eq, and find intersection:
1. For any vec on the plane, n.v = 0
2. Select intersection point up_1 + (1-u)p_0
3. Choose any v on the plane, thus:
up_1 + (1-u)p_0 – v = 0, so n(up_1 + (1-u)p_0 – v) = 0
4. Solve this for u
We clip the line below or above the intersec point.
Clipping cases: for p – the start of the line, q - end
1. H.p > 0, H.q < 0: clip q to the plane
2. H.p < 0, H.q > 0: clip p to the plane
3. H.p > 0, H.q > 0: don't clip, it's fully in the plane
4. H.p < 0 H.q < 0: clip out, it's not in the plane at all
This type of clipping can be done in any 3D obj, not just a frustum. However, it's harder for concave objs.

3.3) Clipping with Convex Objects

Convex Object: A shape for which any line joining two boundary points is fully within the obj. AKA: the intersection of two planar halfspaces.
1) Testing if an object is convex:

```
convex = true; for each face {
  find plane equation of face: F(x,y,z) = 0
  choose a point on the object (xi, yi, zi) not on face
  for all other object points {
    if (sign(F(xj, yj, zj) != sign(F(xi, yi, zi)) convex=false;
} #F(point) just works out the planar val at that pt.
```

Works as all obj points lie on one side of the face.
2) Testing containment within an object:

```
test point = (xt, yt, zt); in = true
for each face of the convex object {
  find plane equation of face: F(x, y, z) = 0
  choose an object point (xi, yi, zi) not on the face
  if (sign F(xt, yt, zt) != sign F(xi, yi, zi)) in = false;
}
```

For efficiency should store the plane eqs of faces for efficiency, and use a vector dot product proj test instead: **P is inside a face if:** *θ is acute, cosθ > 0, and n . (P-A) > 0, A is a point on the obj face.*
Remember to take cross products tip to tail!

3.4) Clipping with Concave Objects

We use the ray containment test: fire a ray, num intersections odd: we are inside the object.
Ray = R = T = ud; u > 0.
In the 2D case: Choose easy **d** e.g: (1 0)T
Take line segments and solve
T + ud = p_0 + u(p_1 – p_0), for v and u.
u > 0, 0 <= v <=1 means it's a valid intersection.
3D: Compute the intersection of the ray with the plane of each face. If intersec is in positive part (u>0) check whether that intersection point is within face.
3.4.1) Clipping (of lines) to concave volumes:
1) Find every intersec of the line to be clipped with the volume.
2) This divides the line into one or more segments.
3) Test a point on the first segment for containment.
4) Adjacent segments will alternate inside and out.
We can also split a concave volume into convex parts, we then apply tests for each convex chunk.

4) Graphics Pipelines, APIs and Programming

OpenGL is imperative – we emit drawing commands. Application sends commands to -> Geometry which sends primitives to -> Rasterization which sends fragments to -> Frame Buffer -> Display
Real time graphics is expensive and based on:
- Rasterization of primitives: points, lines, **triangles**
- Implemented in hardware with APIs like OpenGL.

We can program parts of that pipeline.
Vertex: a point in space defining geometry
Fragment: Sample made during rasterization, we merge multiple of them to form Display Pixels.

4.1) Application Stage

Inits everything: generates render area in OS, data structures (to represent the scene/view frust and acceleration (k-d tree)), input event handlers, utils.
E.g: define a teapot with faces with vertices.

4.2) Geometry Stage (GS)

Vertex Processing (programmable) -> Clipping -> Projection (map clipspace coords to the view plane) -> Viewport Transform (maps normalised device coords to a rectangular window in the framebuffer – the **viewport**) Stages of the GS:
1) Input vertex stream composed of attrs (pos, color) is transformed into stream of vertices mapped onto screen using their clipspace homogenous coords, and extra user defined attrs (color, tex coords) by the **vertex shader**. 2) Vertices are post processed: object space input vertex coords are put into world space using a model matrix, wspace -> viewspace using view mat, vspace -> clipspace using proj mat and these are the outputted vec coords from the GS. ModelView mat takes from objspace to vspace, MVP mat takes from input objspace to output clipspace. Geometry shaders are an optional intermediate between Frag and Vert shaders. Has knowledge of primitives it works on – ie: vertex/adj info. Can emit its own primitives: ie: for drawing growing plants.

4.3) Rasterization Stage

1) Primitive Assembly (Fixed):
- Backspace Culling, setup primitives for **traversal**

2) Primitive Traversal (Fixed):
- Sampling (reading triangles as frags) to produce samples. Multiple Frags merge to form pixels.
- Interpolation of vertex attrs (depth, colour). Interpolation depends on obj rules – we can just colour the wireframe or also interpolate the inside.

3) Fragment Shading (Programmable):
- Compute fragment colours applying tex, lighting

4) Fragment Merging (Fixed):
- From frags, compute pixel cols. Since multip. frags cover a pixel, blend for final col. Resolve visibility.

4.4) Display Stage

Digital Scan out, HDMI encryption.
Format: RGBA, with 16/32/64 bit FPs. Double or Single Buffered (use one buffer to read from and display – buffers are good as we want to produce e.g: 60 completed fps, not display the images as they form), quad buffer stereo if 2 views.
Summary: Geometry processing is per vertex, transformation and lighting. Complex FP OPs. Millions of vertices per second. Fragment processing is per frag – colour blending, texture combining. Simpler FP OPs, billions of frags/second.

4.5) Architecture

Graphics hardware are shared resources. From the program, in layers (but communic. goes both ways):
User Mode Driver – preps CMD buffers for hardware
Graphics Kernel Subsys – sched access to hardware
Kernel Mode Driver – submit CMD buffs to hardware
We now have a unified shader model – a unified ALU with the same ISA and capabilities for all shader type IEEE-754; Geometry Shader can write to memory, stream output. Allows multiple geometry passes.

4.6) Graphics APIs

OpenGL is a low-level API that is cross-language and cross platform for rendering 2D and 3D vec graphics. Lots and lots of others – declarative models.
OpenGL in more detail: it is not a library, it is a low level API Spec. Defines an abstract renderer, and the functions to use it. Draws commands immediately – no concept of permanence, it's a state machine.
OpenGL programs:
1) Render Window (context providing libraries - glut, Qt, browser SDKs)
2) Setup&init functions (vport/model trform, file I/O)
3) Frame generation (update/rendering functions) - defines what happens every frame.
#include <GL/gl.h>
OpenGL types: GLFloat, Glint, GLenum, …
Object oriented. Obj instances have names (uints), commands work on targets (that have an obj bound to it). Targets ⇔ types, cmds ⇔ methods.
Basic Concepts:
1) **Context**: An OpenGL Instance. Processes can have many contexts. Threads only have one **current** context, each cur.context is cur in 1 thread. Operations occur in the current context.
2) **Resources**: sources of input, sinks for output. Buffers (linear mem chunks), images (1/2/3D texel arrs -> input for texture sampling), state objects.
Buffers: store arr of unformatted mem (vertex data, pix. data from images) allocated by the context.
We bind them to the context:
void glBindBuffer(enum target, uint bufferName)
void glBufferStorage() (immut. - stream in the data)
glGenBuffers(n, &my_buff) -> req. n unused buf obj names
glBindBuffer(NAME, my_buff) -> creates buff with NAME
glBufferStorage(NAME, …) -> stores data in the buffer
glBindBuffer(NAME, 0) -> unbind the buffer
glDrawArrays(type, 0, 33) -> draw content of type from start idx 0, with 33 elems. Types: GL_POINTS/LINES/etc…
glDeleteBuffers(1, &my_buff) -> del buff, free resourc/name
Triangles verts are stored in counterclockwise order.
Triangle strips – formed of many triangles, store in alternating fashion: t1: counterclock, t2: clock, …
After setting up the pipeline we make a draw call:
glBegin(GL_TRIANGLE_STRIP);
glColor3f(0.0, 1.0, 0.0); ← colour state
glVertex3f(1.0, 0.0, 0.0); ← vertex index
……… glEnd();
glVertex3fv -> 3 parts, float, omit v for scalar form.
We also have commands to load textures into mem.
OpenGL 4: faster, more efficient but not fixed func graphic ops. All apps must use shaders and buffers.
4 steps: create shader prog, create bufs & load data into it, link data locations with shader vars, render.
Shader Objects are parts of our pipeline, vertex, frag shader. Basically what we wrote for our CWs.
Program Object: whole pipeline. Shader objects linked at runtime.
Extra types: float, int, bool, vec4, mat4, sampler2D Can access vecs like v[1], v.xyzw, v.rgba, etc..
Qualifiers: in, out, uniform.
Builtin Vars: gl_vertexID/gl_Position (in/out v.shader), gl_fragCoord/gl_fragColour (in/out from frag shader).

5) Illumination and Shading (Local Lighting)

After modelling transform, before viewing transform. When looking at an object we see the light it reflects, determined by: obj.pos relative to light, surface normal, reflectivity of the surface. Light ⇔ Photons.
Basic Formulae: 1) Photon Energy: e_λ=hc/λ.
h = 6.63*10^{-34}Js, c = 3 * 10^8m/s, λ = wavelength
2) Energy of n Photons: **Q** = $\Sigma^n_{i=1}$= hc/λ_i
3) Radiation Flux (energy/unit time): **Φ** = dQ / dt
4) Radiance – *radiant flux per unit solid angle per unit projected area*: (number of photons per time at a small area in a particular direction):
L(w) = d²Φ / cosθ dAdw (watt / meter² steradian)
5) Irradiance - differential flux on a differential area:
E = dΦ/dA; units = Watt/meter²
BRDF: Bidirectional Reflectance Distribution Function Reflectance - *fraction of incident flux that is reflected:*
BRDF: **$f_r(\theta_i,\varphi_i,\theta_r,\varphi_r)$**=$dL_r(\theta_r\varphi_r)/dE_i(\theta_i,\varphi_i)$ (1/steradian)
Isotropic BRDF: rotating along the surface norm doesn't change reflectance: **$f_r(\theta_i,\varphi_i,\theta_r,\varphi_r-\varphi_i)$** = $dL_r(\theta_r,\varphi_r)/dE_i(\theta_i,\varphi_i)$
Anisotropic: Rotating along DOES change it – for objects with strongly oriented microgeom elems – fur, cloth, hair, brushed metals.
BRDF Properties:
Non-negativity

$$f_r(\theta_i,\phi_i,\theta_r,\phi_r) \geq 0$$

Energy Conservation

$$\int_\Omega f_r(\theta_i,\phi_i,\theta_r,\phi_r)d\mu(\theta_r,\phi_r) \leq 1 \quad \text{for all } (\theta_i,\phi_i)$$

Reciprocity

$$f_r(\theta_i,\phi_i,\theta_r,\phi_r) = f_r(\theta_r,\phi_r,\theta_i,\phi_i)$$

BSSRDFs: Bidirection Scattering-Surface DFs: Produces a better visual scene. Does this by allowing the light ray to pierce the surface, and reflect and refract off in there. More complex.
Computing Reflected Radiance:
Continuous version:

$$L_r(\omega_r) = \int_\Omega f_r(\omega_i,\omega_r)dE_i(\omega_i) = \int_\Omega f_r(\omega_i,\omega_r)\,dL_i(\omega_i)\cos(\omega_i\cdot n)\,d\omega_i \quad \omega = (\theta,\phi)$$

Discrete with n point light sources:

$$L_r(\omega_r) = \sum_{j=1}^{n} f_r(\omega_{ij},\omega_r)E_j = \sum_{j=1}^{n} f_r(\omega_{ij},\omega_r)\cos\theta_j \frac{\Phi_{sj}}{4\pi\, d_j^2}$$

5.1) Ideal Surface Reflectance:

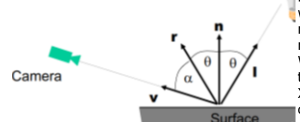
Assume the surface reflects equally in all directions.
View agnostic. Only for very matte materials
So we can pull the BRDF out if the integral, which gives us the integral of the derivative of E_i.
So, given the continuous formula above we get:
f_rE_i = L(w_r) = k_d(n·l) Φ_s / (4πd²)
where: k_d is the diffuse reflection coefficient, n is the surface normal, l is the light direction.
If n and l face away from each other the angle exceeds 90 and it is negative. Thus, we need to use max((n*l), 0), to clamp the result.
Remember, all vectors point to outside of the surface!

5.2) Ideal Specular Reflectance:

Assume the reflection is only at the mirror angle.
View dependent. Example: mirrors, polished metals.
The incoming ray, the surface normal, and the reflected ray all lie in a common plane, ie:
$n_I\sin\theta_I = n_r\sin\theta_r$; $n_I = n_r$; $\theta_i = \theta_r$, for surface normal n, incident ray I, and reflected ray r.

$$L(\omega_r) = k_s(\cos\alpha)^q \frac{\Phi_s}{4\pi\, d^2} = k_s(\mathbf{v}\cdot\mathbf{r})^q \frac{\Phi_s}{4\pi\, d^2}$$

The light reflectance depends on the angle between the ideal reflection direction and viewer direction a:



k_s: specular reflection coefficient
q: specular reflection exponent (shininess)
q:1 - rough, 10 - glossy, 100 – shiny
Since r+l = 2 cosθn, r = 2(n . l)n – l, so our formula:

$$k_s(\mathbf{v}\cdot(2(\mathbf{n}\cdot\mathbf{l})\mathbf{n}-\mathbf{l}))^q \frac{\Phi_s}{4\pi\, d^2}$$

In truth, in the real world, objects are part specular, part diffuse.
Phong's Model composes diffuse and specular lighting, along with an ambient term as a hack to approximate global illumination to produce a local lighting model!

5.3) Phong Lighting Model:

$$L(\omega_r) = k_a + \left(k_d(\mathbf{n}\cdot\mathbf{l}) + k_s(\mathbf{v}\cdot\mathbf{r})^q\right)\frac{\Phi_s}{4\pi\, d^2}$$

Combines all three terms!
1/d² represents the inverse square law, but doesn't produce the best results.
Using denominator 4π(d+s) looks better.
Blinn-Phong: We instead use the vector h that's halfway between v and l:
h = (l + v) / ||l + v||
Beta is the angle between h and n:
Cheaper, and can look better than Phong.

5.4) Shading

Our techniques before simply calculated the colours at vertices. We need to colour the rest of the surface, too!
1) Flat Shading: Polygon are shaded uniformly. Shade is computed by taking a centre point and the surface normal vector. (Equivalent to a light source at infinity). Usually only diffuse and ambient components are used.
2) Interpolation (Gouraud Shading): Compute an independent shade value at each point via interpolation.
1. Compute a shade value at each vertex: ie: Phong Illumination.
2. Interpolate to find the shade value at the boundary line edges:
For example: to find the colour at L, between vertices L_1 and L_2 with distances d_1 from L_1 to L, d_2 from L_2 to L:
L = $(d_1L_2+d_2L_1)/(d_1+d_2)$
3. Interpolate to find the shade values in the middle:
For a point L_A on an edge, and L_B on another edge, the point L between them at distance d_1 from L_A and d_2 from L_B is coloured via this interpolation:
L = $(d_1L_B+d_2L_A) / (d_1+d_2)$

After this we should also interpolate over groups of polygons to have smooth lighting over faces. Each vertex has average intensity of all the polygons that meet at that vertex.
3) Phong Shading:
Interpolate normal over triangles, onto each pixel we want to shade. Redo lighting calculation using the individual normal we calculated. Done in frag shader. Looks better, slower.
e.g: we have 4 triangles meet, avg norm: $n_{ave} = (n_1+n_2+n_3+n_4)/4$
Take a point P = $V_1 + u_1(V_2-V_1) + u_2(V_3-V_1)$
The avg norm vector a is calculated as so:
a = $n_1 + u_1(n_2-n_1) + u_2(n_3-n_1)$
and n_{ave} = a / |a|
The interpolation calculations may be done in either 2D or 3D For specular reflections the calculation of the reflected vector and viewpoint vector must be done in 3D.

6) Colour

Colour is a wave, and thus has wavelength (λ) and amplitude (intensity/energy) (I)

Color	Frequency	Wavelength
violet	668–789 THz	380–450 nm
blue	631–668 THz	450–475 nm
cyan	606–630 THz	476–495 nm
green	526–606 THz	495–570 nm
yellow	508–526 THz	570–590 nm
orange	484–508 THz	590–620 nm
red	400–484 THz	620–750 nm

Light is a mixture of lots of colours (wavelengths) – if you see red light it just means we mainly have red waves.

We can represent all waves in terms of R, G & B; ie: X = rR + gG + bB, where R, G, B are pure light sources and r, g, b are their intensities. So, we can represent a colour as 3 coeffs.
We can match all colours by adding light to the colour we try to match, e.g:
X + (-r) = g+b. This way we can match all colours. We can use the CIE Diagram to represent colours:

6.1) CIE Diagrams:

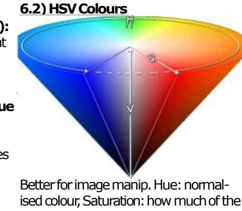
Normalise the coordinates x, y, z so that the components sum to 1: x = r/(r+g+b) y = g/(r+g+b), z = b/(r+g+b) = 1 - x - y
Since z is in terms of x and y, diag is 2D
X = 0.33, y = 0.33 is the white point.
Saturation of an arbitrary point:
ratio of its distance to the white point over the distance of the white point to the edge
Pure = fully saturated.
Complement colour - the colour diametrically opposite through the white point. colour + complement = white.

RGB format is an **additive primary representation** - sums to make white.
When printing colour we use a subtractive representation instead - subtract components to create colour. Subtractive primaries: Magenta, Cyan, Yellow. They sum to make black.
We use 24 bits in total - 8 for each of R, G, B, since human eye only can see 350k colours. Screens use 3 diodes to represent colour, where the diodes each have these colours on the CIE diagram:

	x	y	z
R	0.628	0.346	0.026
G	0.268	0.588	0.144
B	0.150	0.07	0.780

Thus we can't actually display the full CIE spectrum. We convert CIE to monitor RGB by taking that table as a 3x3 matrix and setting RGB to the right and mult. Better monitors have better diodes.

6.2) HSV Colours



Better for image manip. Hue: normalised colour, Saturation: how much of the colour, Value: colour brightness
Converting between HSV and RGB:
V = max(r, g, b)
S=(max(r,g,b)-min(r,g,b))/ max(r,g,b)
H = an angle between 0 and 360; a rotation around the conical base.
We have cases:
- If (r=g=b) Hue is undefined, the colour is black, white or grey.
- If (r>b) and (g>b)

Hue = 120*(g-b)/((r-b)+(g-b))
- If (g>r) and (b>r)

Hue = 120 + 120*(b-r)/((g-r)+(b-r))
- If (r>g) and (b>g)

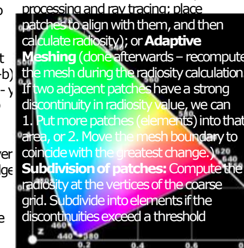
Hue = 240 +120*(r-g)/((r-g)+(b-g))
We perceive Green the strongest and Blue the weakest, so to have colours visible to us in equal levels they need scaled intensities.

6.3) Alpha Channels

When using RGB in a rendering system we also need alpha as a 4th parameter. α is an attenuation of intensity; allows greater flexibility in colour representation avoids truncation error, and allows masking of specific parts of the image. α = 0 transparent; 0 < α< 1 semi-transparent; α = 1 opaque Essentially, RGBα represents a pixel with colour C = (R,G,B) as C = (αr, αg, αb,α)

11.3.1) Gauss Seidel Discussion:

1) Start by with all unshot radiosity to 0, except emitting patches where $\Delta B_i = E_i$
2) Choose patch w/largest unshot radiosity ΔB_i
3) Shoot the radiosity for all other patches: $\Delta B_j = R_jF_{ji}\Delta B_i$; add to radiosity.
4) Now set ΔB_i = 0; iterate.
Because we do iters, objects next to each other may have different iter counts and thus look discontinuous. lerp.
Meshing: the process of dividing scene into patches. Scene dependent. D^0 artifacts are very visible on shadow boundaries, caused by discontinuities in the radiosity function.To fix:
lerp/**Discontinuity Meshing** (compute discontinuities in advance, with pre-processing and ray tracing: place



patches to align with them, and then calculate radiosity); or **Adaptive Meshing** (done afterwards – recompute the mesh during the radiosity calculation. If two adjacent patches have a strong discontinuity in radiosity value, we can 1. Put more patches (elements) into that area, or 2. Move the mesh boundary to coincide with the greatest change.)
Subdivision of patches: Compute the radiosity at the vertices of the coarse grid. Subdivide into elements if the discontinuities exceed a threshold

7) Texture Mapping

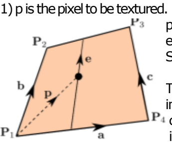
We can use images as textures to map onto surfaces. Simplifies graphics processing.
Textures: 1D funcs – parameter along which tex is defined has arb domain e.g: incident angle
2D funcs – info is provided for all tex coords (u,v)
3D Funcs – Calculated along volume T(u,v,w)
Most commonly: **Raster images ("texels")**: images from scanner, photos or calculation. Used as writing realistic texture functions is hard.
Texture coords (t, s) are applied onto 2D object surfaces (v, u) displayed on screen coords (x,y). Textures can get distorted after transformations.

7.1) Texture Parameterization Techniques

1) **Planar Mapping:** dump a set of coordinates from an axis onto the surface. ONLY looks good from that angle.
2) **Cylindrical / Spherical Mapping:** Comput angles between vertex and object centre, using polar coord system determine (v,u) texture.
3) **Box Mapping:** Make the texture a cubic net enclosing the object, apply it this way
Parameterization produces mappings that always have distortions. Artists can use CAD to manually get mappings that look good.

7.2) Applying Textures in Practice

Canonical texture coords are between (0,0),(1,1) Usually texture size is a pow of 2. We need to tile.
Texture Addressing Modes (dealing with points outside of the canonical range):
1) Apply Static Colour outside of texture range
2) Clamp (repeat the colours at texture border)
3) Repeat (just repeat the texture pattern)
4) Mirror (repeat texture by reflecting w.r.t dirs)
Modes rarely good enuf. **Texture Synthesis** Algorithms – produce larger examples of the texture we had w/o repeating or tiling - by replicating pixels cleanly.
Idea: Map textures to vertices, then Gouraud Shade. **Problem:** Distortion along the diagonal edge of cube faces after perspective projection on the triangles of texture: perspective projection does not preserve linear combinations of points.e.g: take points P & R and their midpoint Q in 3D space. When perspective projecting it onto our viewspace, Q could become closer to R! Equal distances in 3D space do not map to equal distances in screen space. **Solution:**
1) Assign param texture coords t_0=0 and t_1=1 to 3D vertices p & r. t controls a linear blend of tex coords p&r.
2) Assume the image plane is at z = 1 (so f=1 fo projection). Using our knowledge of perspective projection, the mapping on the 2D screen is:
$p' = p/z_p$, $q' = q/z_q$, $r' = r/z_r$
3) We need to carry out perspective correct interpolation – we need to compute $t_{p'}$, $t_{q'}$, $t_{r'}$.
$t_{p'} = t_p/z_p$, $t_{r'} = t_r/z_r$. We can lerp $t_{p'}$ and $t_{r'}$ for $t_{q'}$
4) Use $t_{q'}$ to compute t_q:
$t_q = t_{q'} \times z_q = \text{lerp}(t_{p'}, t_{r'}) / \text{lerp}(1/z_p, 1/z_r)$
Summary: Given texture param t at vertices:
1) Compute 1 / z for each vertex for persp proj
2) Lerp 1 / z onto perspective proj for the t_p, t_r
3) Lerp t_p, t_r for $t_{q'}$
4) Use the formula for t_q.
We could also use Bilinear Texture Mapping:
1) p is the pixel to be textured.

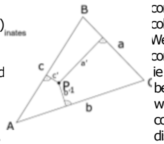


p = αa + βe
e = b+a(c-b)
So: p = αa + βb + αβ(c-b)
This is a quadratic in αβ. This might cause straight lines in the texture to curve when we do the mapping. But if the mapping is to a parallelogram, b = c; it simplifies.
We can use these textures to alter illumination: Note I_a, I_d, I_s. Or it can just replace ambient col.
$L(w_r) = k_a I_a + k_d I_d(n.s) + k_s I_s(v.r)^q$

2D Texture Mapping still has problems: If we move around we find out our texture is just a flat image due to lighting.
1)We fix this by instead of using triangle normal, interpolate an averaged normal at each vertex across the face. Smooths it out.
2) We can also use bump maps – textures alter the surface normal of the object – we shade it as if we had the normal given on the map!
Bump maps – cheap, but shadows incorrect.
Displacement Mapping – use a texture map to move surface point. Done before visibility – helps determine occlusions.
Environment maps – have an image of environment around us, and simulate reflections that way using the direction of the ray to index a spherical texmap. Assume rays come frm 1 point.

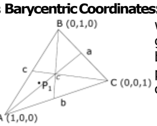
8) Rasterization, Visibility and Anti-Aliasing

Rasterization: Turn Objects into pixels.
Interpolate values inside objects (colour, depth):
Trilinear Coordinates: Each coordinate vertex contributes to the final colour of the object.



We calculate weights of contrib of opposite verts ie: if P lied on A the line between A and B, c would have no contribution, as the distance is 0.

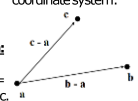
Barycentric Coordinates:



weight contribution given by the alpha, beta and gamma parameters of the coordinate system:

8.1) Vector Formation:

For P inside the triangle:
P = a + β(b-a) + γ(c-a) =
p(α, β, γ) = αa + βb + γc.



Homogenous Barycentric Coords: α + β + γ = area of triangle.
Areal Coordinates / Abs Barycentric coords: α + β + γ = 1
Each coeff is between 0 and 1; edge: 1 coeff = 0, vertex: 2 are 0.
A barycentric coordinate (e.g. β) is a signed distance from a line (e.g. line going through ac)
Implicit Line Equation in 2D:
f(x, y) = 0, if on the line. Ax + By + C = 0.
To find implicit eq, take two points (x_a, y_a), (x_b, y_b)
The implic eq is: $(y_a-y_b)x+(x_b-x_a)y+x_ay_b-x_by_a = 0$
Just sub points in; we get 2 possibilities depending on which way we sub the points in
Line eqs for triangle w verts $(x_a,y_a),(x_b,y_b),(x_c,y_c)$:
$f_{ab}(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_ay_b - x_by_a$
$f_{bc}(x, y) = (y_b - y_c)x + (x_c - x_b)y + x_by_c - x_cy_b$
$f_{ca}(x, y) = (y_c - y_a)x + (x_a - x_c)y + x_cy_a - x_ay_c$
Signed distance from line ac: compute $f_{ac}(x,y)$
Computing Barycentric Coordinate of Point:

$$\alpha = \frac{f_{bc}(x,y)}{f_{bc}(x_a,y_a)} \quad \beta = \frac{f_{ac}(x,y)}{f_{ac}(x_b,y_b)} \quad \gamma = 1-\alpha-\beta$$

In other terms: barycentric coordinates for p are the area coords are the solution to:

$$\begin{pmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Converting Barycentric Coords to Trilinear:
$P_t(t_1, t_2, t_3)$ in trilinear coordinates has barycentric coordinates of (t_1a, t_2b, t_3c) where a, b and c are the side lengths of the triangle. $P_b(\alpha, \beta, \gamma)$ is in barycentric has trilinear coordinates (α/a, β/b, γ/c).
With this understanding, we write our rasterizer:

```
for all x do
  for all y do
    compute (alpha, beta, gamma) for (x, y)
    if (0 < alpha < 1 and 0 < beta < 1 and
        0 < gamma < 1 ) then
      c = alpha c0 + beta c1 + gamma c2
      drawpixel with colour c
```

8.2) Visibility

Visibility: Handles occlusions / determines the closest objects to decide what's visible:
Fragments might overlap, and triangles may too. Pixels have to determine what to actually draw.
1) Painter's Algorithm for visibility:
1) Sort triangles (using z values in view space)
2) Draw triangles from back to front (closest triangle last drawn) **Issues**: Intersections and cycles not rendered properly (part of the triangle behind another). We can sort this by splitting triangles but this is ugly and expensive. It's also inefficient due to requiring sorting.
2) The Z-Buffer Algorithm:
1) Each fragment gets a z value in screen space
2) Keep only the fragment with the smallest z value (store the closest fragment so far)
3) Anytime we want to write anything to the colour buffer you check the corresponding z buffer to see if the new fragment is closer (smaller z value) than the previous.
4) If not then don't write, if so then write and update the z buffer. The Z buffer is a 2D buffer the same size as the image, as we store Z for each pixel location (as many fragments may map to the same pixel).

8.3) Anti-Aliasing – blur reduce jaggedness

Rasterization can make lines/boundaries jagged due to undersampling (turning a continuous line into discrete pixels). Methods:
1) Supersampling:
1) Compute image at higher res than the display.
2) Average supersamples to find the pixel value This blurs boundaries and leaves coherent areas of colour unchanged. Good for scenes made up of filled polygons. But expensive & doesn't work for line drawings. As we can see, we compute the overall pixel by using the colour average of the "higher resolution" picture version at a lower level. More accurate edge colours.
2) Convolution Filtering: cheaper. Use filter to blur image by taking average over region around pixel. Fast, in hardware. We use weighted avgs – weigh closest pixels. Degrades the image.
Doing it for textures: when we identify a pt in tex map, return the average around it. Scaling applied; less samples then the bigger the local area of sampling.

9) Ray Tracing for Global Illumination

Global Illumination: Receive light not just from the source, but of the environment around it.

9.1) Ray Tracing Algorithms

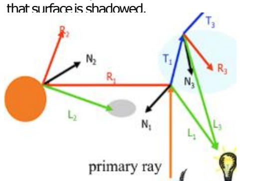
1) Ray Casting: per pixel, cast a ray. For each object intersection, trace a ray BACK to the light to check for shadows. (works backwards to how light works – we trace rays from eye to obj)

```
cast ray
   Intersect all objects
   color = ambient term
   For every light
       cast shadow ray
       col += local shading term
```

Recursive Ray Casting: allows object colours to influencing by both lights and other objects. Simulates specular reflect/refractive transmission. As above, but func is called trace and with:

```
If mirror
    col+= k_ref * trace reflected ray
  If transparent
    col+= k_tra * trace transmitted ray
```

at the end. Stop on rec depth/ray contrib<thresh.
Primary Rays: Rays fired out of viewer perspective into the scene. Represents the path along which light travels from the camera. For each ray test what objects intersect it:
• If intersection, calculate distance between viewpoint and intersection
• If more than one, then we select the smallest distance as the visible surface
Secondary Rays: Rays originating at intersection points. Caused by: • Shadow Rays
• Rays reflected off intersection point in reflec dir
• Rays trnsmted thru t.sparent mats in refrac dir
If we trace a ray to a source and its blocked, then that surface is shadowed.



Shadow Rays: Rays we trace back to the light source from intersection points to check if the point should be illuminated, or if occluded.
Due to FLOP error, we might generate secondary rays below obj surface. Thus, to stop occlusion:
• if abs(μ) < ε: the point is on the surface
• else: inside/outside: move the point out by ε across the surface normal. Then check if in/out via the implicit sphere equation. Known as ε tolerance

9.2) Generating Secondary Rays

1) For Mirror Reflections:
• Compute Mirror Contribution
• Cast ray in a symmetric direction w.r.t normal.
• Multiply by reflec coeff (as it won't perfectly refl)
• Add epsilon to ray
2) Calculating Illumination for Refractions:
• Same as before, but different dir:
  • n is the unit surface normal
  • v is the dir of primary ray
  • v' is the dir of 2^{nd}ary ray due to refraction;
  $v' = v - (2v.n)n$
3) Objects that reflect and refract:
• Angle of reflec (Snell): $\eta_1\sin(\varphi_1) = \eta_2\sin(\varphi_2)$
• Refrac: η_1 and η_2 are refractive indexes.
As a vector: $k_1(v \cdot n) = k_2(v' \cdot n)$. v' = direc =

$$v' = \frac{\eta_1}{\eta_2}\left(\left[\sqrt{(n \cdot v)^2 + \left(\frac{\eta_2}{\eta_1}\right)^2 - 1} - n \cdot v\right] n + v\right)$$

Only has dir if root +ve: **limiting angle property:**
• If light goes from a medium to one which has a lower refractive index, the refracted ray's angle is greater than the incident ray's angle.
• If the angle of the incident ray is large, angle of refracted ray then exceeds 90.
• In this case, no solutions and ray is reflected rather than refracted. Improvement: mix reflected and refracted using **Fresnel Property:**
$L = k_{fresnel}L_{reflection} + (1-k_{fresnel})L_{refraction}$
Considers more reflection at grazing angles.
Schlick's Approximation:
$k_{fresnel}(\theta) = k_{fresnel}(0) + (1 - k_{fresnel}(0))(1-(n \cdot l))^5$
Where we select $k_{fresnel}(0)$ – the factor at 0 deg. 0.8 looks like stainless steel.

$$L = k_a + S(k_d(\mathbf{n}\cdot\mathbf{l}) + k_s(\mathbf{v}\cdot\mathbf{r})^q)I_s + k_{reflected}L_{reflected} + k_{refracted}L_{refracted}$$

S = 0 or 1 depending on if light source is obscured
1 Shadow Ray per intersec: solid black shadow
Multiple (for area light source): softer, realistic.
Monte Carlo Ray Tracing: Cast a ray from the eye. Cast many rand primary rays per point. Recurse **one** secondary ray. Cast prim&sec back to light. Cheaper, better. **Path Tracing:** send many primary rays per pixel.

9.3) Intersect all objects step

Find nearest intersec point.
Rays: parametric lines: $p(\mu) = p_0 + \mu d$
p_0: curr pixel on viewing plane p_v: viewpoint
d: direction vector: $d = p_0 - (p_v / |p_0 - p_v|)$
μ < 0: behind viewing plane. visible iff μ > 0
Our scene is defined in solid models (spheres, cylinders) or surface models (plane, triangle, polygon)
1) Intersection Calculations for Spheres:
$$\mu = -d \cdot \Delta p \pm \sqrt{(d.\Delta p)^2 - |\Delta p|^2 + r^2}$$
Where: μ and d are from the ray equations, $\Delta p = p_0 - p_s$ (where p_0 is the ray origin and p_s is the sphere center) r = sphere radius
If the quadratic has 0 solns: intersect. 1 soln: 1: touch. 2 solutions: smaller solution = entry, bigger = exit
Remember to check for self-shadowing! (ε)
2) Intersection Calculations for Cylinders:
r = cylinder radius, p_1 is the position vector of the center of the top circle on the cylinder, p_2 is the position vector of the bottom, $\Delta p = p_1 - p_2$; μ,d are both from the ray equation. We can parameterize Δp as a ratio 0 < α < 1. We get:
$$r^2 = (p_0 + \mu d - p_1 - (\frac{p_0\Delta p + \mu d\Delta p - p_1\Delta p}{\Delta p\Delta p})\Delta p)^2$$
The two solutions, α_1 and α_2 have μ_1 and μ_2 terms. (α_1 or α_2 = inner-inner bracket part with μ_1 or μ_2)
Assuming $\mu_1 < \mu_2$: α_1 between 0 & 1: intersec is on the outside surface of the cylinder.
α_2 between 0 & 1: on inside surface of the cylinder
3) Intersection Calculation for Planes:
Motivation: Objects are defined usually in terms of triangles, planar quads or planar polygons.
Intersection: $\mu = -(p_0 - p_1) \cdot n / d \cdot n$
Where: p_1 is a point in the plane, n is the planar normal, and everything else comes from the ray.
3.1) Intersection Calculation for Triangles:
1) Test whether the triangle is front facing: d·n<0 n is planar surface normal, d is ray direc vector.
2) Test if plane of the triangle intersects ray:
Do the exact same thing as in 3).
3) Test whether the planar intersection point q is actually within the triangle: q = αa+βb; 0 ≤ α or β ≤ 1. and they sum to ≤ 1. Calculate them via:
$$\alpha = \frac{(\mathbf{b}\cdot\mathbf{b})(\mathbf{q}\cdot\mathbf{a}) - (\mathbf{a}\cdot\mathbf{b})(\mathbf{q}\cdot\mathbf{b})}{(\mathbf{a}\cdot\mathbf{a})(\mathbf{b}\cdot\mathbf{b}) - (\mathbf{a}\cdot\mathbf{b})^2}$$
$$\beta = \frac{\mathbf{q}\cdot\mathbf{b} - \alpha(\mathbf{a}\cdot\mathbf{b})}{\mathbf{b}\cdot\mathbf{b}}$$
What this is doing:
1) Translates the origin of ray and then changes base of that vector to get param vec (t, u, v)
2) t is the distance to the plane with the triangle
3) u, v are barycentric coordinates in the triangle.
4) We don't need to store the plane equation as we can calculate the norm easily

9.4) Accelerating Ray Tracing

9.41) Bounding Box Approach: make ideally tight bounding box around an object. If we don't intersect the box, we don't intersect the object.
Step 1) Create Grid by constructing a bounding box of scene. Grid Resolution (n_x, n_y, n_z) - controls how many of each cell we have in each direction.
2) Put the primitive into the grid – they can overlap multiple cells. Use pointers.
3) When tracing a ray, if we have an intersection we return the closest object.
4) Once we hit an object we mark it so we don't reintersect it later for efficiency.
5) If we have an object that is in the cell but the intersection is not within the cell's range, don't return this intersection as we may have something closer (efficiency). Ie: the case where we intersected a cell containing a primitive, but not the primitive itself.

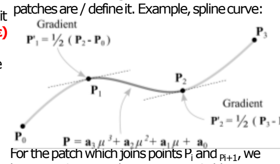
Adaptive grids: Subdivide until each cell contains no more than n elements, or max depth d reached
- guarantees efficiency by not putting too many elements in 1 cell. Can have them at intermediate levels or all primitive at leaves (span just one cell)
9.4.2) Binary Space Partition Trees:
Recursively partitions the space with planes - each cell is a convex polyhedron. Tries to reduce number of object in each division to as little as possible. This lets us locate our intersections easily later.
Regular Grids: +easy to construct & traverse -may be sparsely filled -geometry may be clumped
Adaptive: +grid complexity matches geometric density -more expensive to traverse (esp BSP tree)

10) Spline Curves

How do we represent non planar shapes?
Splines: smooth curves formed from a small set of control points. Types of spline:
1) **Interpolating splines:** curve passes through each point of the set.
2) **Approximating Splines:** curves don't pass through all points. These act as control points; user can adjust the shape of the curve interactively.
Polynomial Spline: $y = a_2x^2 + a_1x + a_0$
Parametric Splines: $P = a_2\mu^2 + a_1\mu + a_0$
μ ranges from 0 to 1 to trace along the curve.
We solve for vector constants a_0, a_1, a_2.
a_0: Set it to point we want curve to start at ("P_0")
If spline ends at P_2, μ = 1 at P_2, so $P_2 = a_2+a_1+P_0$
If we want middle to be P_1: $P_1 = ¼a_2+½a_1+P_0$

10.1) Patching – connecting many splines

Higher order splines are hard to work with.
So, we can piece smaller simpler splines to form each curve, the pieces are called **spline patches.**
To calculate the parametric spline patches we use a cubic polynomial $P = a_3\mu^3 + a_2\mu^2 + a_1\mu + a_0$
Depending on the desired spline curve behaviour, we can figure out what the patches are / define it. Example, spline curve:



For the patch which joins points P_i and P_{i+1}, we have μ = 0 at P_i and μ = 1 at P_{i+1}. Subbing in we get: $P_i = a_0$, $P_{i+1} = a_3 + a_2 + a_1 + a_0$.
Differentiate the spline w.r.t μ:
$P' = 3a_3\mu^2 + 2a_2\mu + a_1$
Substitute for μ = 0 at P'_i and μ = 1 at P'_{i+1}:
$P'_i = a_1$, $P'_{i+1} = 3a_3 + 2a_2 + a_1$
In matrix form, we get the equations:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} P_i \\ P'_i \\ P_{i+1} \\ P'_{i+1} \end{pmatrix}$$

Invert the matrix on both sides, and we get the a values we need to define our overall cubic spline, given the chosen spline patches we have!
We want to create smooth realistic shapes.
Smooth Function: Usually, its smooth if its derivatives are well defined up to some order. Smoothness is defined differently for curves and surfaces depending if we're looking at them as a function or as a shape.
1) Parametric Continuity: Defined in terms of the function, not the shape:
• A junction between two curves is C^0 continuous if the (x, y) values of the two curves agree.
• C^1 continuous if the (x,y) and first derivatives (dx/ds, dy/ds) agree. C^2 for 2nd order as well.
2) Geometric Continuity: Defined only in terms of the shape (Parameterization doesn't influence):
• G^0 and C^0 continuity are equivalent.
• A junction between two curves is said to be G^1 continuous if the (x, y) values of the two curves agree, and all first derivs (dx/ds, dy/ds) are proportional (tangent vecs are parallel) at the junction.
• Higher order geom. continuity is hard to define

10.2) Bezier Curves (BC)

Give predictable results for small knot sets. Good to use to construct spline patches. Characteristics:
• They interpolate the end points
• The slope at an end is the same as the line joining the end point to its neighbour
1) Casteljau's Algorithm for BC Construction
We have parameter μ which is 0 at the start of the curve and 1 at the end. **Process:** start with our BC points:
1) Shift each point by μ along the line to the next point. Do this shift n-1 times; n = num points.
2) Do it for a value of μ to discover the curve, as each time we end up getting a point!
3) Use more values of μ, each time you get a point! Will give us more and more Bezier knots.

Bernstein Blending Function (BBF): Forms Splines and Bezier Curves as a blend of the knots.
Any point on spline is blend of all other points; e.g:

$$\mathbf{P}(\mu) = \sum_{i=0}^{N} W(N,i,\mu)\,\mathbf{P}_i$$

W is the BBF: $W(N,i,\mu) = \binom{N}{i}\mu^i(1-\mu)^{N-i}$

$$\binom{N}{i} = \frac{N!}{(N-i)!\,i!}$$

Bezier Curves are blended so can't be used for fine detail curves, but they are good spline patches.

We can show equivalence of N point Bezier Curves and the patch going through the first and last knots P_0 and P_{N-1}.
Do this by expanding the iterative blending eq and reversing Casteljau's algorithm.
1) Note that N-1 in BBF. Then simply just plug N-1 and expand it mathematically.
2) We'll end up with equations for a. Now calculate P'(0) and P'(N-1) by differentiating and plugging in 0 and 1 respectively.
3) Form a matrix that looks like this (this is the case N = 4), and then use our derivatives from part 2 to write P'(0) and P'(N+1) things in terms of $P_1, P_2 \ldots P_{N+1}$

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}'_0 \\ \mathbf{P}_3 \\ \mathbf{P}'_3 \end{pmatrix}$$

Now we've shown blending == cubic patch.
4) For the other side of the proof, reverse Casteljau example on N=4

$$\begin{aligned}
\mathbf{P}_{3,0} &= (1-\mu)\mathbf{P}_{2,0} + \mu\mathbf{P}_{2,1} \\
&= (1-\mu)\{(1-\mu)\mathbf{P}_{1,0} + \mu\mathbf{P}_{1,1}\} + \mu\{(1-\mu)\mathbf{P}_{1,1} + \mu\mathbf{P}_{1,2}\} \\
&= (1-\mu)^2\mathbf{P}_{1,0} + 2\mu(1-\mu)\mathbf{P}_{1,1} + \mu^2\mathbf{P}_{1,2} \\
&= (1-\mu)^2\{(1-\mu)\mathbf{P}_{0,0} + \mu\mathbf{P}_{0,1}\} \\
&\quad + 2\mu(1-\mu)\{(1-\mu)\mathbf{P}_{0,1} + \mu\mathbf{P}_{0,2}\} \\
&\quad + \mu^2\{(1-\mu)\mathbf{P}_{0,2} + \mu\mathbf{P}_{0,3}\}
\end{aligned}$$

Collapsing the recursion:

$$\begin{aligned}
\mathbf{P}(\mu) &= (1-\mu)^2\{(1-\mu)\mathbf{P}_0 + \mu\mathbf{P}_1\} \\
&\quad + 2\mu(1-\mu)\{(1-\mu)\mathbf{P}_1 + \mu\mathbf{P}_2\} \\
&\quad + \mu^2\{(1-\mu)\mathbf{P}_2 + \mu\mathbf{P}_3\} \\
&= (1-\mu)^3\mathbf{P}_0 + 3\mu(1-\mu)^2\mathbf{P}_1 + 3\mu^2(1-\mu)\mathbf{P}_2 + \mu^3\mathbf{P}_3
\end{aligned}$$

We can summarise Bezier Curves in terms of by saying that it has: two points that are interpolated (P_0, P_3) and two control points (P_1, P_2).
We can interactively shape the curve by moving P_1 and P_2 with a mouse.

10.3) Spline Surfaces – 3D Spline Curves

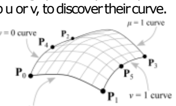
Can define parametric polynomial surfaces:

$$\mathbf{P}(\mu,\nu) = (\mu,\nu,1)\begin{pmatrix} a & b & d \\ b & d & e \\ c & e & f \end{pmatrix}\begin{pmatrix} \mu \\ \nu \\ 1 \end{pmatrix}$$

$$\mathbf{P}(\mu,\nu) = \mathbf{a}\mu^2 + \mathbf{d}\nu^2 + 2\mathbf{b}\mu\nu + 2\mathbf{c}\mu + 2\mathbf{e}\nu + \mathbf{f}$$

We can solve this easily by subbing in our choice of u and v and dealing with the simultaneous equations to discover the a..f coefficients.
Boundary points are spline patches, so we can sub in 0 or 1 for the edge point they correspond to, and then keep u or v, to discover their curve.

	μ	ν
P_0	0	0
P_1	0	1
P_2	1	0
P_3	1	1
P_4	1/2	0
P_5	1/2	1



$$\begin{aligned}
\mathbf{P}(0,\nu) &= \mathbf{d}\nu^2 + 2\mathbf{e}\nu + \mathbf{f} \\
\mathbf{P}(1,\nu) &= \mathbf{a} + 2(\mathbf{b}+\mathbf{e})\nu + 2\mathbf{c} + \mathbf{d}\nu^2 + \mathbf{f} \\
\mathbf{P}(\mu,0) &= \mathbf{a}\mu^2 + 2\mathbf{c}\mu + \mathbf{f} \\
\mathbf{P}(\mu,1) &= \mathbf{a}\mu^2 + 2(\mathbf{b}+\mathbf{c})\mu + \mathbf{d} + 2\mathbf{e} + \mathbf{f}
\end{aligned}$$

Now any middle point can be expressed as a linear combo of these.
This formulation is flexible, but only good for simple surfaces. Higher orders are difficult to use. Instead, we use a **cubic spline patch** technique where we connect many spline surfaces depending on the desired behaviour.
Forming Spline Surfaces with Patches:
Instead, we can make a terrain map where describe every single bounding curve as a cubic splines.
At each corner of the patch we need to interpolate the points and set the gradients to match the adjacent patch. We have two gradients: dy/dz and dy/dx.
We use the μ, ν formulation to model the terrain maps (what we look at the derivative on).
We need to match three values at each corner: P(μ, ν), dP(μ, ν) / dμ, dP(μ, ν)/dν

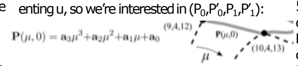
10.3.1) Constructing Spline Surfaces

Adopt corners as param vals (0,0),(0,1),(1,0),(1,1)
1) Edges: since patch neighbours need to join exactly, we need to ensure edge contours are equivalent on adjacent patches.
We do the same method as for cubic spline patches: make gradients equal
Easier method, instead of computing all these gradients, use a Coons Patch:
Define internal points by **lerp of edge curves:**
P(u, v) = P(u,0)(1-v) + P(u,1)v + P(0,v)(1-u) + P(1,v)u - P(0,1)(1-u)v - P(1,0)u(1-v) - P(0,0)(1-u)(1-v) - P(1,1)uv
2) Rendering a Patch: (Polygonalisation)
transform it into polygons. Select a grid of points, e.g. u = {0.0, 0.1, ... 1.0}, v = {0.0,0.1,0.2...1.0} and triangulate to the grid. Approximation.
For speed: large polygons, with Gouraud/phong.
For accuracy: small, pixel sized polygons
Lofting can be used too – drawing contours of constant u and/or v. Obsoleted algorithm.
3) Ray tracing a Coons Patch: No closed form solution, as patch eq is 4^{th} order. Must use numerical methods:
1. Polygonise the patch at a low resolution (4x4)
2. Calculate the ray intersection with the 32 triangles and find the nearest intersection.
3. Polygonise the immediate area of the intersection and calculate a better estimate of the intersection
4. Continue until the best estimate is found
We might have multiple intersections - we don't necessarily find the nearest. Works well for smooth objects.
We must do ray intersection per object patch.

Computing a Coons Patch:

		y, ν →					
		2	3	4	5	6	7
	7			.	.	.	.
	8	.	.	10	9	.	.
x, μ	9	.	14	12	11	10	.
↓	10	.	15	13	14	10	.
	11		.	10	11	.	.

Returned: (x,y,output)
Find the coons patch on the centre 4 points:
Well, we can see that:
P(0,0) = (9,4,12), P(1,0) = (10,4,13),
P(0,1) = (9,5,11), P(1,1) = (10,5,14).
We can find the gradients in each dir by doing a computer vision type convolution. for example in the x,u direction (for all 4 corners):
$dP/du_{(0,0)} = ((10\ 4\ 13)^T - (8\ 4\ 10)^T)/2 = (1\ 0\ 1.5)$
$dP/du_{(0,1)} = (1\ 0\ 2.5)$
$dP/du_{(1,0)} = (1\ 0\ -1)$
$dP/du_{(1,1)} = (1\ 0\ 0)$
We do this for y/v too.
With this information we can find the boundary coons patches: e.g: for P(u, 0): we are incrementing u, so we're interested in (P_0,P'_0,P_1,P'_1):



$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} 9 & 4 & 12 \\ 1 & 0 & 1.5 \\ 10 & 4 & 13 \\ 1 & 0 & -1 \end{pmatrix}$$

After substituting in $\mathbf{P}(0,0)$, $\frac{\partial\mathbf{P}}{\partial\mu}\big|_{(0,0)}$, $\mathbf{P}(1,0)$, $\frac{\partial\mathbf{P}}{\partial\mu}\big|_{(1,0)}$

We get out constant vectors and now we've discovered one of the boundary curves. Note we do have to do this for all 4 corners. This tells us the 4 boundary curves, and these together define a coons patch. (as seen in blue formula above).

11) Radiosity for Global Illumination

Rather than using an ambient light term in the reflectance equation, or assuming n point light sources, since every surface is a light reflector, shouldn't they all be considered a light source?
A better approximation: Make ambient light a function of incident light:
$I_{reflected} = k_a I_{incident} + k_d(n \cdot l)I_{incident} + k_s(r \cdot v)^q I_{incident}$
AKA: $I_{reflected} = RI_{incident}$; R is old reflectance func

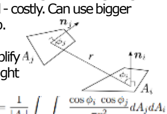
11.1) Patching for Radiosity

Radiosity: energy per unit area leaving a surface = emitted energy per unit area and reflected energy
AKA: for a small area of the surface (patch) dA (where emitted energy is regarded as constant):
BdA = EdA + RI
For the scene, per patch i...n, this is written to be:
$B_i = E_i + R_iI_i$ where:
B_i = total energy leaving the patch
E_i = the energy emitted by the patch itself
R_i = reflectance value (as according to the old eq)
I_i = Incident light energy arriving at the patch
AKA: we regard each patch as both a light reflector and light emitter.

Incident energy for patch I can be estimated as:

$$I_i = \sum_{j=1}^{n} B_j F_{ij}$$

Where: Sum is taken over all scene surface patches
B_js: represents energy from other scene patches
$F_{ii} = 0$.
F_{ij} is a constant (the form factor) - links surface patch i with patch j by how much of that energy it gets from j. Accounts for patch areas & angles.
Substituting the incident light expression:
Energy leaving i-th patch: $B_i = E_i + R_i\sum_j B_jF_{ij}$
Difficulties:
We must solve this for each B_i to render our light model. However, we usually have many, many patches, and need to find form factors beforehand.
In matrix form, it's enormous. **(check 11.2)**
Radiosity has an R,G,B value for each patch; 3 calcs. Also, the specular term in the reflectance eq accounts for angle. But must do it for **every patch!** Thus, we usually only calculate **diffuse radiosity.** Need to decide patch size; big – visible lighting diff across patches, small - costly. Can use bigger patches and then lerp.
Computing Form Factors: We can simplify the equation on the right



By assuming that A_i and A_j are small compared to r, we get the simplified equation

$$F_{ij} = \frac{1}{|A_i|}\int_{A_i}\int_{A_j}\frac{\cos\phi_i\cos\phi_j}{\pi r^2}\,dA_j\,dA_i$$

$$F_{ij} = \frac{\cos\phi_i\cos\phi_j|A_j|}{\pi r^2}$$

IE: for objects close by & facing each other: high F_{ij}

11.1) The Hemicube Method:

Cheaper method; computes form factors in groups:
• Using a bounding hemisphere, patches that project onto the same hemisphere area have equal FFs.
• Use Hemicubes -> Hemispheres as cheaper.
• Hemicubes are divided into small pixel areas and FFs are precomputed for each. The resulting form factors can be used for every patch in the scene. We just 'place' the same hemicube over each.
• If the area of a hemicube pixel is |A|, FF is:

$$\frac{\cos\phi_i\cos\phi_j|A|}{\pi r^2}$$

(remember, φ_i is angle bw the straight line dist vec r and patch i's normal)
Because it's a hemicube we have the side face and top face cases.
For a top face, $\cos\varphi_i = \cos\varphi_j$ so FF = $|A| / \pi r^4$
For a side, $\cos\varphi_i = 1/r$, $\cos\varphi_j = z_p/r$, FF = $z_p|A|/\pi r^4$
We also need to know which patch is visible from each hemicube pixel: done by ray tracing or projection which needs z-buffering (check 8.2).
Summary:
1. Divide the graphics world into discrete patches
Meshing strategies -> meshing errors
2. Compute form factors by the hemicube method
Alias errors
3. Solve the matrix equation for the radiosity of each patch. Use computational strategies
4. Average the radiosity values at the corners of each patch. Use Interpolation approximations
5. Compute a texture map of each point or render directly
Form Factor Reciprocity: We only need to compute half of them because: $F_{ij} = F_{ji}|A_i| / |A_j|$
But this means we'd have to store them and there's potentially **billions** -> Gigabytes!
We can save some space since many FFs are 0 -> use an 1-bit indexing scheme for FFs: 0 -> FF = 0; not stored in array.

11.2) Gauss-Seidel Numerical Method

$$\begin{pmatrix} 1 & -R_1F_{12} & -R_1F_{13} & \cdot & \cdot & -R_1F_{1n} \\ -R_2F_{21} & 1 & -R_2F_{23} & \cdot & \cdot & -R_2F_{2n} \\ -R_3F_{31} & -R_3F_{32} & 1 & \cdot & \cdot & -R_3F_{3n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -R_nF_{n1} & -R_nF_{n2} & -R_nF_{n3} & \cdot & \cdot & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \cdot \\ \cdot \\ \cdot \\ E_n \end{pmatrix}$$

Putting this matrix into RAM is hard – it's enormous!
But Gauss Seidel solves systems **row by row,** iteratively. Since a row's equation looks like this:
Set for all $iB^0 = 0$. Make $B_i = E_i + R_i\sum_j B_jF_{ij}$
successive iterations B^1,B^2,...
For example, with three patches, we can write the iterations like so, as update equations:
and have the vals of the Rs, Es & F_{ij}s

$$\begin{aligned}
B_0 &\leftarrow E_0 + R_0(F_{01}B_1 + F_{02}B_2) \\
B_1 &\leftarrow E_1 + R_1(F_{10}B_0 + F_{12}B_2) \\
B_2 &\leftarrow E_2 + R_2(F_{20}B_0 + F_{21}B_1)
\end{aligned}$$

Cyclical, run iterations of this and we eventually get **convergence**. Radiosity matrices are diagonally dominant, which is why we converge.
Cutoff = num iters, or delta thresh. Extra Iters model the bouncing of light around the scene.
Speeding up Gauss Seidel: evaluating a row is known as gathering (rays). We can find the change to every other patch after a row changes using:
$B_j^k = B_j^{k-1} + R_jF_{ji}\Delta B_i^k$ Known as shooting
This lets us speed things up: evaluate patches with largest ΔB (unshot radiosity) first:
(Continued on the other page, above the CIE diagram lol)