# Object Representation

- Implicitly f(x,y,z)=0 – i.e. all points in 3D space where some function is zero.
- e.g. sphere $(x^2+y^2+z^2)=r^2$
- Any guesses for?:

$$(2x^2 + y^2 + z^2 - 1)^3 - (\tfrac{1}{10})x^2z^3 - y^2z^3 = 0$$

# Object Representation

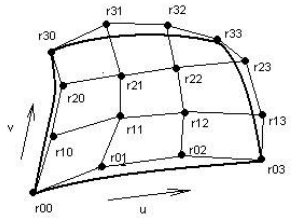
- **Implicitly** f(x,y,z)=0 – i.e. all points in 3D space where some function is zero.
- e.g. sphere $(x^2+y^2+z^2)=r^2$
- Any guesses for?:

$$(2x^2 + y^2 + z^2 - 1)^3 - (\tfrac{1}{10})x^2z^3 - y^2z^3 = 0$$

- It's a cartoid (heart!)
- **Implicits**:
  - Extremely compact storage

# Object Representation

- Parametrically
  e.g. Bezier surface patch




- Smooth joins, very good for curved surfaces

# Object Representation

- Polygon meshes (e.g. triangular mesh)
- Bruno Levy (Inria) of Michaelangelo's David

2 billion polygons, 32GB

# Triangular Meshes

- Advantages:
  - Ray/triangle intersection for ray-tracing is easy
  - Can project triangles in GPU hardware (at millions per second (Computer Games)
  - Can easily manipulate subdivide triangular meshes (e.g. for Level of detail models or radiosity)

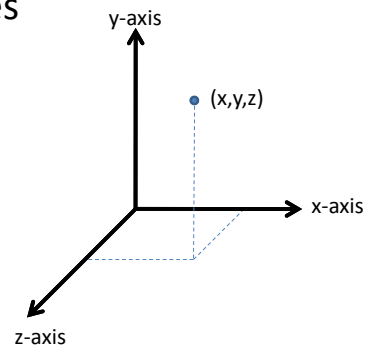
# Terminology

- GPU (Graphics Processing Unit) – a chip dedicated to the processing of vertices for the purposes of display
- OpenGL – Open Graphics Language – a language for programming graphics applications. The language maps to GPU hardware and is OS independent
- Direct3D – Microsoft's graphical programming language. The language maps to GPU hardware and is OS dependent (Windows)

# Modelling: 3D Primitives

- Point
- 3D location in space
- Represented by coordinates

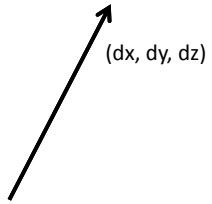
```
class Point {
private:
   float x, y, z;
...
};
```




# Modelling: 3D Primitives

- Vector
- 3D direction and magnitude (no position)

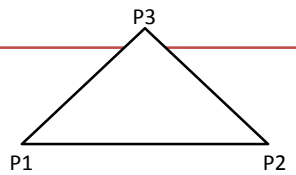
```
class Vector {
private:
   float dx, dy, dz;
public:
   float Magnitude() const {
      return
      sqrt(dx*dx+dy*dy+dz*dz));
   }
...
};
```



# Modelling: 3D Primitives

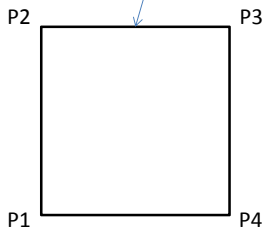
- Triangle
- These explicit representations lead to duplicated points
- This problem is examined in the next slides

```
class Triangle {
private:
   Point P1, P2, P3;
...
};
```



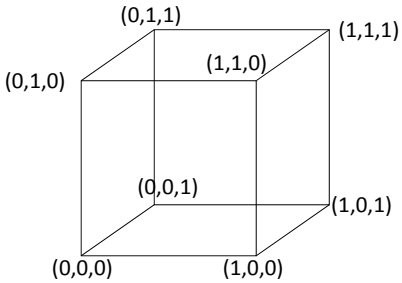

- Quad

```
class Quad{
private:
   Point P1, P2, P3, P4;
...
};
```




# Explicit Representation

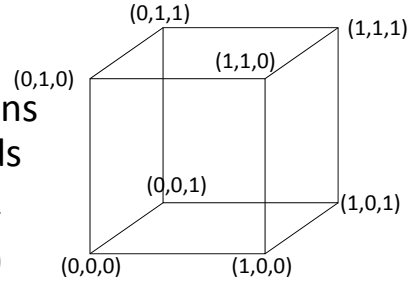
- Cube (6 faces / quads)
- (0,0,0) (0,0,1) (1,0,1) (1,0,0)
- (1,0,0) (1,1,0) (0,1,0) (0,0,0)
- (1,1,0) (1,1,1) (0,1,1) (0,1,0)
- (1,1,1) (1,0,1) (0,0,1) (0,1,1)
- (1,0,0) (1,0,1) (1,1,1) (1,1,0)
- (0,0,1) (0,0,0) (0,1,0) (0,1,1)
- Drawbacks:
- 3D transformations of 24 vertices (not 8)
- Draw 24 edges (rather than 12)
- Rounding errors – consider picking vertices




# Pointers to Vertex List

- Vertices / Points
- 0=(0,0,0)
- 1=(0,0,1)
- 2=(0,1,0)
- 3=(0,1,1)
- 4=(1,0,0)
- 5=(1,0,1)
- 6=(1,1,0)
- 7=(1,1,1)

- Polygons / Quads
- 0 1 5 4
- 4 6 2 0
- 6 7 3 2
- 7 5 1 3
- 4 5 7 6
- 1 0 2 3



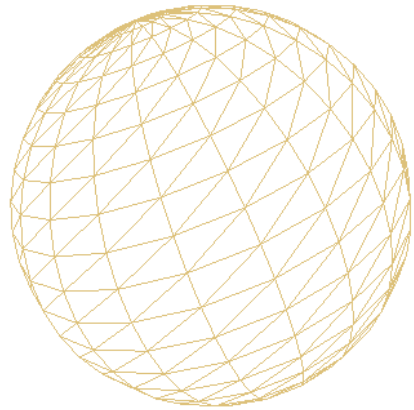

Advantages:
- 3D transformations of just 8 vertices.
- Rounding errors not a problem.

Drawbacks:
- Draw 24 edges (rather than 12)
- Extra memory
- Extra processing during modelling

# 3D Primitives

- Pointers to Vertex List widely used (although see triangle strips)
- Each triangle vertex is a pointer to a 3D point
- An object is a list of triangles (or quads)

```
class Triangle {
private:
   Point *P1, *P2, *P3;
...
};
```

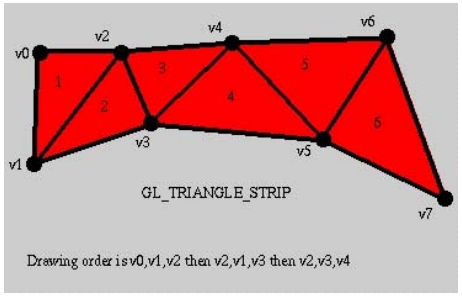


# Example

- For example, the previous sphere consists of 382 vertices and 760 triangles
- Each vertex is 3 floats (3x4 bytes=12)
- Each triangle is a list of 3 pointers (3x4 bytes=12)
- This model uses 13,704 bytes
- Using (next) triangular strip model uses 762x12=9,144 bytes

# Triangular Strips

- Compact (n triangles represented using n+2 vertices)
- Therefore transmission to GPU is lower
- Very efficient when drawing (particularly in hardware)
- Can be hard to create triangle strips from arbitrary geometry

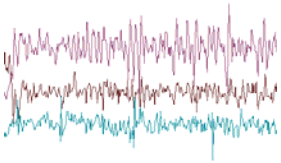


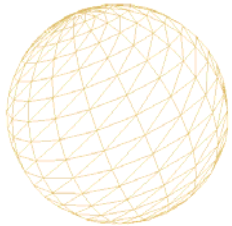

OpenGL

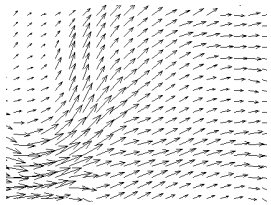
# Direct3D Drawing Primitives

- D3D_POINTLIST
  A list of isolated points (n vertices=n points)
- D3D_LINELIST
  A list of isolated lines (each pair of points are the ends of a line) (2n vertices=n lines)
- D3D_LINESTRIP
  The vertices make a continuous line (n+1 vertices=n lines)
- D3D_TRIANGLELIST
  Each group of 3 points define an isolated triangle (3n vertices=n triangles)
- D3D_TRIANGLESTRIP
  (Previous slide) (n+2 vertices=n triangles)
- (Direct3D allows pointers to a vertex list using VERTEX BUFFERS)

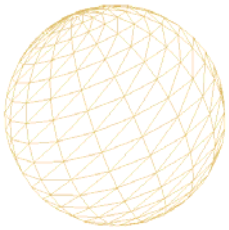
# What shall we use? (Answers in lecture)





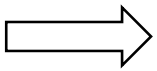


# Rendering

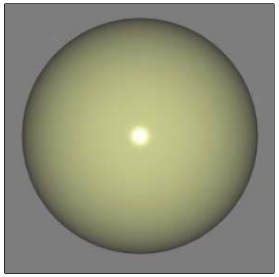


Model / scene comprised of geometric primitives in 3D coordinate space

Rendering

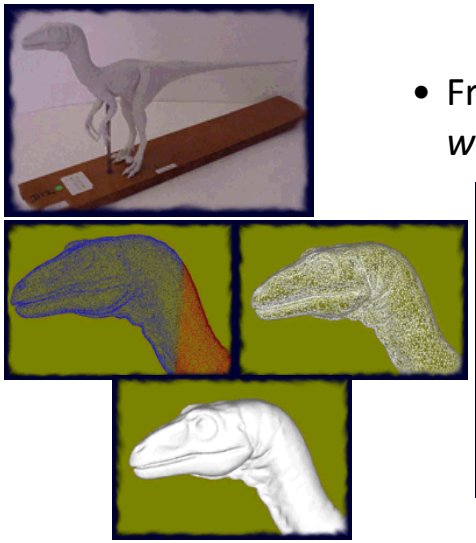


Transformation of 3D space



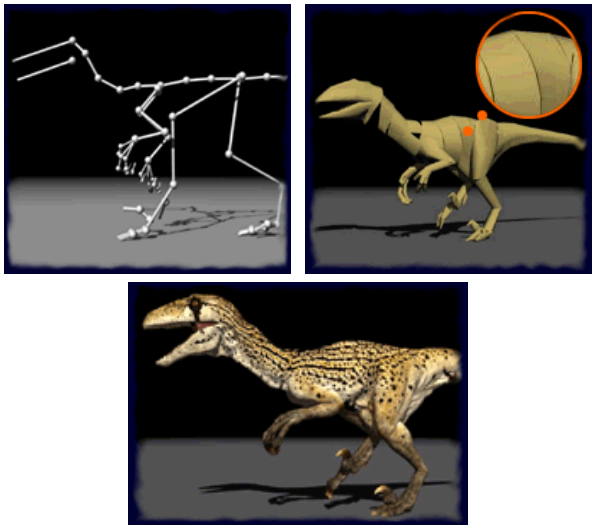
Raster image

# Modelling via capture

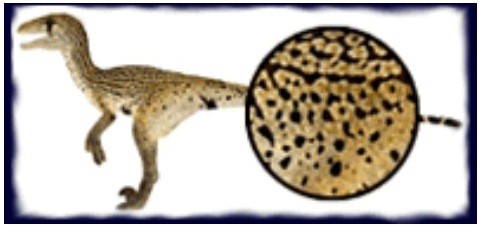


- Framestore's *Walking with Dinosaurs*

# Animation



# Textures



# Lighting

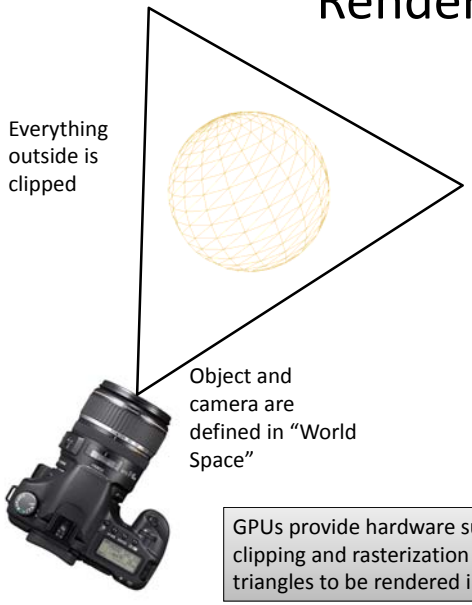


- Range
- Light source and intensity

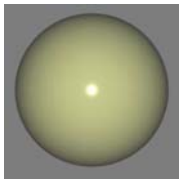
# Rendering

- We will study two forms of rendering – rasterization (briefly) and ray tracing (in detail)
- **Rasterization** consists of several steps:
  - **Transformation**, **clipping** and **scan conversion**
- **Matrices** for scaling, translation and rotation are applied to each vertex within the object during transformation
- After transformation each vertex (x,y) gives the screen coordinate, and z gives the depth
- **Clipping** removes any part of the scene not visible within the image
- Scan conversion colours pixels according to the object's colour and the **lighting model**

# Rendering




World coordinates are transformed into view coordinates (x,y,z)

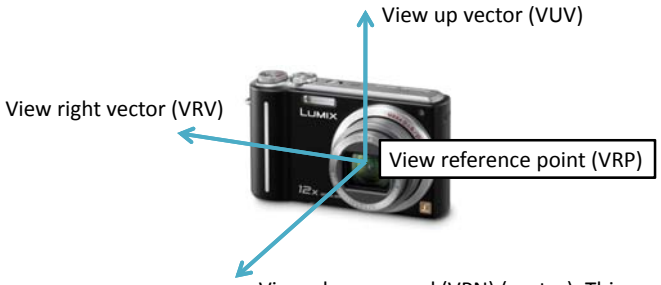


Such that (x,y) give the position within the view (image) and z gives the depth to that position. The depth can be used to make sure that occluded surfaces are hidden by closer surfaces

GPUs provide hardware support for transformation, clipping and rasterization allowing scenes of millions of triangles to be rendered in real-time

# Camera Model

For both Rasterization and Ray Tracing, we need to define a camera model. We explore which parameters are needed

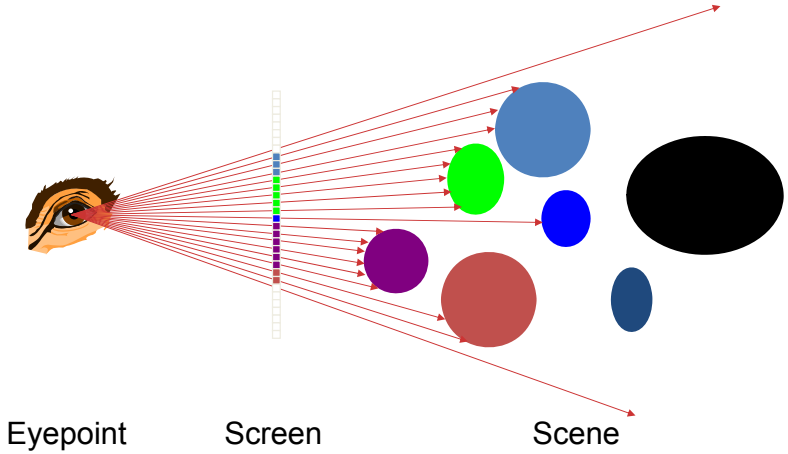



View plane normal (VPN) (vector). This can be calculated from the position (point) the camera looks at minus the view reference point of the camera

# Transformation

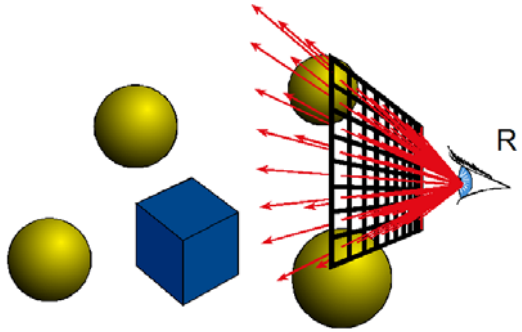
- The transformation matrix can be calculated from the view plane normal (=look at – vrp), the view up vector and the view right vector
- For rasterization, each vertex is multiplied by the matrix (in GPU hardware)
- The resulting (x,y,z) points can be clipped and scan converted
- In ray tracing, rays are sent out from the view plane, into the scene to detect which objects are hit
- **Ray tracing** is studied in more detail in the next lectures

# Ray Tracing



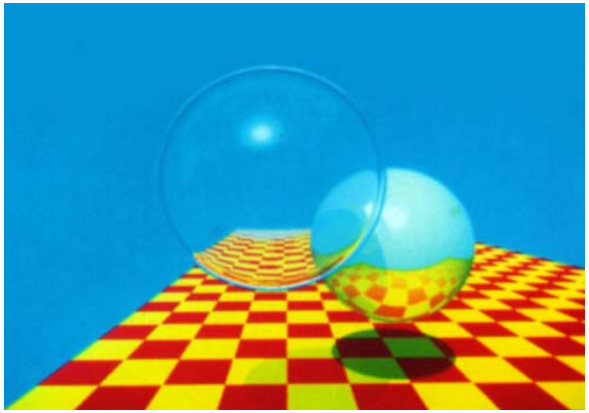

# Ray Tracing

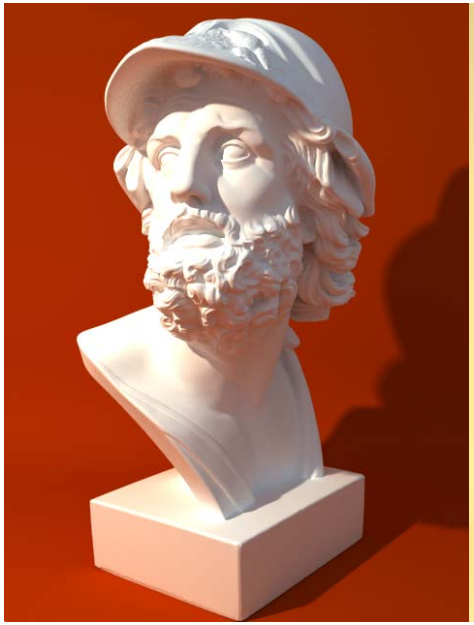
- Similar diagram (but in 3D)

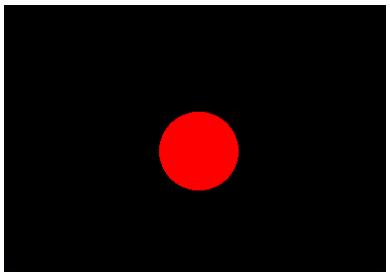



# Ray Tracing - 1979

- Shadows, refraction, reflection and texture mapping







- What about a 3rd year project in ray tracing?
- The first image most people see!
- Intersection between 3D line p=o+dt and sphere (p-c)^2=r^2

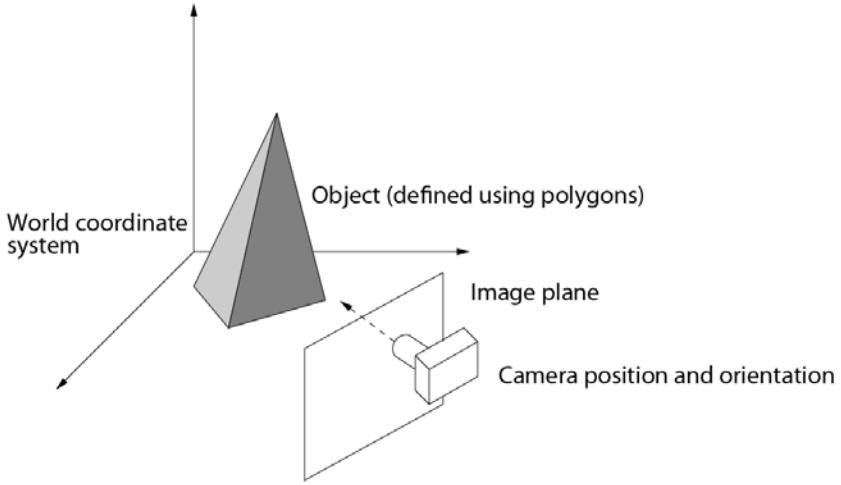
## Ray/Sphere intersection

- Ray $p=o+dt$, sphere $(p-c)^2=r^2$
- Substitute ray p into sphere
- $((o+dt)-c)^2=r^2$
- Expand
- $(o-c)^2+(dt)^2+2dt(o-c)=r^2$
- Rearrange
- $d^2t^2+2d(o-c)t+(o-c)^2-r^2=0$

## Ray/Sphere intersection

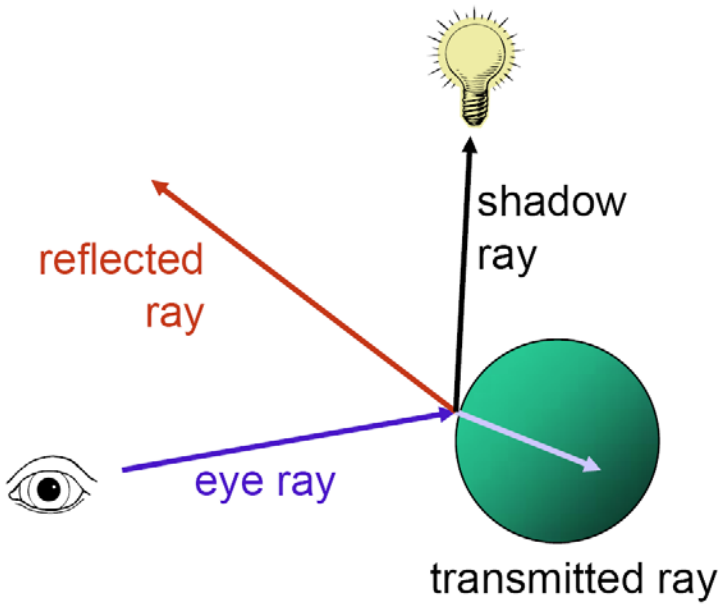
- $d^2t^2+2d(o-c)t+(o-c)^2-r^2=0$
- but to solve $at^2+bt+c=0$, we use
- $t=(-b+-sqrt(b^2-4ac))/2a$
- where $a=d^2$, $b=2d(o-c)$, $c=(o-c)^2-r^2$
- The line starts at the eyepoint (o), and goes in a direction (d) through each pixel

# World Coordinates




# Code (for reference)

```
    procs=omp_get_num_procs();
    omp_set_num_threads(procs);
#pragma omp parallel private(tid, i, j, C, ray_orig, my_RayTri)
{
    tid=omp_get_thread_num();
    for (j = tid; j < tex_h; j+=procs) {
        for (i = 0; i < tex_w; i++) {
            /*** Calculate ray origin ***/
            ray_orig=my_camera.Ray(((double)(i-
centreX))/((double) tex_w), ((double)(j-centreY))/((double)
tex_h));
            my_RayTri.SetOrigin(ray_orig);
            my_RayTri.SetDir(my_camera.VPN);
            C=my_RayTri.TraceRay();
            *(*Image+i*4+j*tex_w*4)=(GLubyte) C.x;
            *(*Image+i*4+j*tex_w*4+1)=(GLubyte) C.y;
            *(*Image+i*4+j*tex_w*4+2)=(GLubyte) C.z;
            *(*Image+i*4+j*tex_w*4+3)= (GLubyte) 255;
        }
    }
}
```



# 1 Ray Tracing

## 1.1 Scene Definition

Objects are defined within world coordinates. For example, a sphere at $(2, 4, 6)$ with radius 10 could be defined using the equation $(x-2)^2 + (y-4)^2 + (z-6)^2 = 10^2$.

The 'camera' is also defined in world coordinates. Its position, orientation (known as *view up vector*) and its direction (known as *view plane normal*) must all be given (figure 1). Once these factors are known (along with the size of the image) rays can be fired through each pixel into the scene (the origin of the pixel is known, $o$, and the direction, $\vec{d}$, and therefore the line representing the ray can be defined:

$$L = o + t\vec{d} \tag{1}$$

where $0 \leq t \leq \infty$.

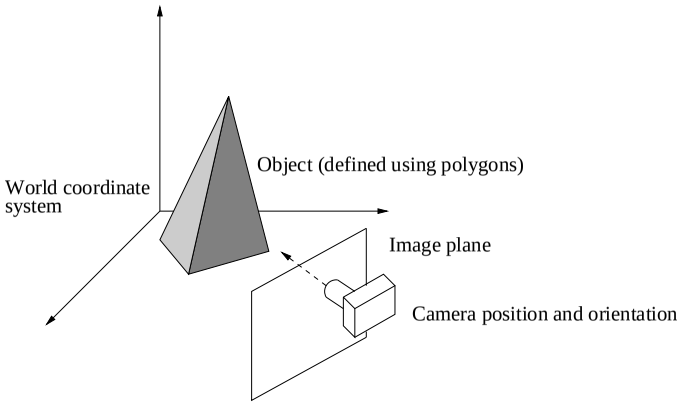



Figure 1: Object defined within world coordinates, with camera position, orientation and image plane.

## 1.2 Orthographic Projection

A ray is traced that originates at the pixel and travels parallel to all other pixels and orthogonal to the plane (Figures 2).

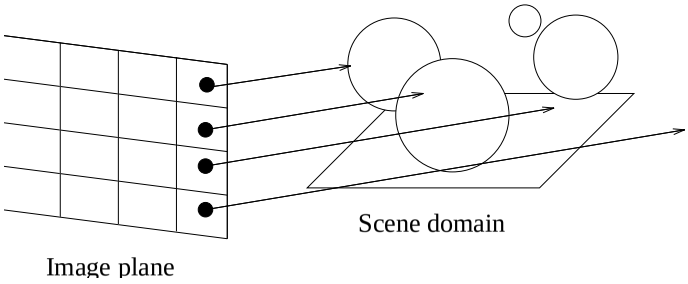



Figure 2: Orthographic projection for ray tracing.

## 1.3 Perspective Projection

A ray is traced that originates at the eye and travels though a pixel in the image (Figures 3).

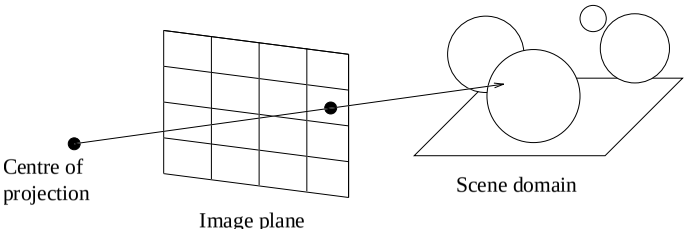



Figure 3: Perspective ray tracing, rays emit from the centre of projection, through each pixel, and on to intersect (or not) objects within the scene.

## 1.4 Ray / Object Intersection

Ray tracing is the process of sending imaginary rays out from the viewer's eye into the scene of objects which we wish to visualise. The object each ray hits is determined, and the colour for the pixel that ray corresponds to is accordingly found. Naively we have to determine if each ray intersects any of the objects in the scene. We then have to find which object was the closest, and obtain the colour of the object.

Typical objects used in ray tracing are those that can be intersected with a 3D line (ray) easily. These include spheres, boxes, cones, cylinders, swept surfaces and surfaces of revolution.

The equation of a 3D line is solved with this equation of the object in question. One of the simplest equations to solve is that of the line and a sphere which accounts for why we see so many spheres in ray tracing.

For example, all points, $p$, on a sphere can be defined using its centre, $c$, and radius, $r$ as $(p-c)^2 = r^2$. We can substitute the equation for the line (equation 1), into the equation for the sphere as $p$:

$$(o + t\vec{d} - c)^2 = r^2 \tag{2}$$

$$(o-c)^2 + \vec{d}^2t^2 + 2\vec{d}t(o-c) = r^2 \tag{3}$$

$$\vec{d}^2t^2 + 2\vec{d}(o-c)t + (o-c)^2 - r^2 = 0 \tag{4}$$

Solving for $t$ in $at^2 + bt + c = 0$ gives $t = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$, and we can apply it in this case with $a = \vec{d}^2$, $b = 2\vec{d}(o-c)$, $c = (o-c)^2 - r^2$

Once the intersection has been found; the exact point of contact is known; the surface normal is known; and the object hit is known. The pixel can be given the appropriate colour by calculating the lighting equations at that point.

Disadvantage: sheer computational expense! For example; A 1000x1000 image requires 1 million rays to be intersected with the scene domain. If the scene involves 1000 primitives (which is quite low) then 1 billion calculations must be done for 1 frame.

# 2 Recursive Ray Tracing

Why do we use ray tracing? Why not use a faster method? Ray tracing is the most accurate rendering model so far (when advanced global illumination techniques are included). It closely models the real physics of light, and therefore known properties of light such as frequency, different speeds of travel in a different medium (refraction), intersurface absorption, colour, reflection, etc. can all be simulated. Ray tracing research has progressed on two fronts - better realism and quicker ray tracing.

Recursive ray tracing is the process of continuing rays (or initiating new ones) at object intersection points so that we can determine reflections, shadows and refractions.

## 2.1 Reflection

If an object is partially or totally reflective we can use the intersection point as the origin of a ray to find out what is reflected (Figure 4).

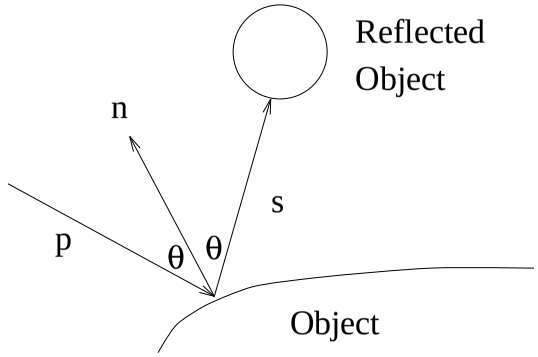



Figure 4: A ray being reflected.

n = surface normal.
$\theta$ = angle of incidence (physics of reflection).
p = primary ray (from pixel).
s = secondary ray (from object).

If the object is a total reflector (100%), the colour of the pixel will come from the secondary ray. If the object is a partial reflector of x%, the colour of the pixel is (100- x)% of the object, and x% of the secondary ray.

## 2.2 Shadows

If a ray hits an object, the object may or may not be lit. To determine this we shoot a secondary ray towards each light source and determine if it hits anything in-between. If it hits something that point on the object is not lit, otherwise it is (Figure 5).

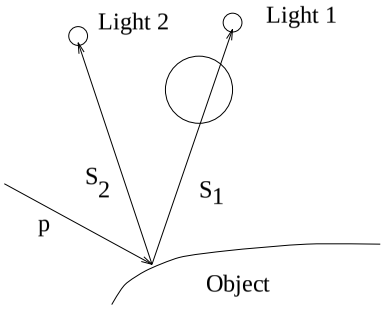



Figure 5: Checking if a point can see the light sources.

In this case the object is only lit by one light (2).

## 2.3 Transparency

If an object is partially or totally transparent then the pixel will take on some of the colour of the object and some of the colour of objects behind the partially transparent object (Figure 6).

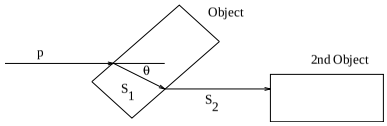



Figure 6: A ray refracted through a transparent object.

$\theta$ = angle of refraction (physics of light moving from one medium to another.)
p = primary ray.
$S_1$ = first secondary ray.
$S_2$ = second secondary ray.

If the object is n% transparent the pixel will take on (100-n)% of the objects colour and n% of the second object.

## 2.4 Putting it all together : Recursive ray tracing

Each ray can spawn several rays (Figure 7) - each of which can spawn other rays depending upon what they hit. If we are not careful there could be a recursive explosion of rays which may not terminate - e.g. a ray bouncing between two mirrors. We can end this by counting the levels of recursion and stopping at some cut-off point.

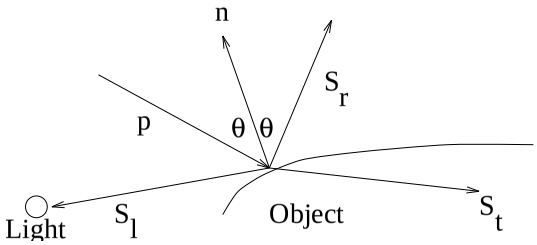



Figure 7: Recursive ray tracing.

p = primary ray.
n = surface normal.
$S_l$ = secondary ray to light.
$S_t$ = secondary ray due to transparency (refracted).
$S_r$ = secondary ray due to reflection.

Through the use of recursive ray tracing we can simulate reality - partial mirrors, partially transparent objects, light sources. When this is combined with conventional shading techniques we have a very powerful way of producing accurate images. More advanced techniques simulate how a camera works. By doing this we can access features such as depth of field, focus and shutter speed (hence motion blurring). The only drawback is the computational expense involved. We can go some way to addressing this problem by using the bounding boxes and spatial subdivision.

# 3 Bounding Volumes

We shall look at the best algorithms for speeding up ray-tracing - bounding volumes and spatial subdivision.

Example: Consider a complex scene such as a chess board. Each piece could consist of say 100 primitives. We also have the board itself. Altogether there would be around 3000 primitives

(by primitive we mean triangle, sphere, plane etc.) Using our 1000x1000 image that would be 3 billion intersection calculations. The easiest acceleration technique to use is the bounding box. Each object is defined - and then has a box or sphere put around it which contains the object totally. Our scene can now be viewed as a board and 32 bounding boxes for the chess pieces. Each ray is intersected with this scene and for each pixel that intersects a bounding box, it is then intersected with the primitives in that box. We now have the case that many pixels will just be intersected with 33 objects, and those that intersect a bounding box will be intersected with the 100 primitives that comprise it. Our 1000x1000 image now requires around 100 million calculations - 30 times faster.

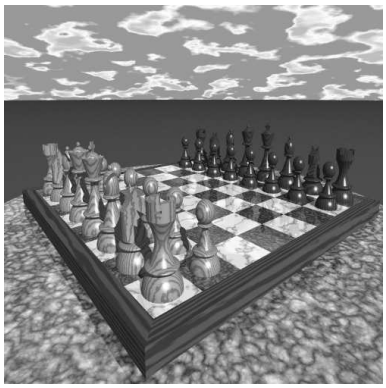


Figure 8: A ray traced chessboard (from POV-Ray).

If a ray hits a bounding volume, it will have to be tested with every primitive inside that bounding volume. Obviously we require bounding volumes to fit the object as tightly as possible. In Figure 9 we use both a cylinder and a sphere to bound the pawn. The cylinder is the tighter fit, and correspondingly less rays will hit the bounding volume than in the case of using a sphere.

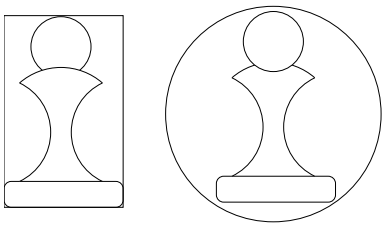


Figure 9: Using a cylinder and a sphere as bounding volumes.

# 4 Spatial Subdivision

It should be obvious that using bounding boxes is a good idea. We can take this one step further and say that; of those 100 primitives can we fit several boxes around groups of those primitives? When testing against an actual chess piece we now check against say 4 bounding boxes, and then whichever the ray intersects we check against those 25 primitives.

This should remind you of the binary tree - we can eliminate up to half the search space at each stage by just one decision, thus enabling us to get down to the item we require quickly.

We can extend the binary tree to 3D. Given a box surrounding the whole scene we can split it into 2 in each of the three axis (x, y and z). Each box can further be divided into 2 in each axis and so on. We regard this as an **octree** (8 splits).

We cover the scene with a box - this is our top node.

**Recursive step** *If* the box contains some primitive in the scene, we divide it into eight. We then test each primitive in the scene to see which box it is in. For each box it is in (it may be in more than one), we place a pointer from that box's object list to this object. We continue until we have located all parts of the scene.

We then take each of the eight boxes in turn and repeat the recursive step until we reach a certain predetermined tree-depth. Usually if a box only contains a few primitives in its list (e.g. 10), we would not split it any further, as we would be replacing a test against 10 primitives with a test against 8 boxes plus whatever primitives are in each box intersected (i.e. no benefit).

This is done once for our static scene. We can then use this octree data structure to ray trace many views of the scene.

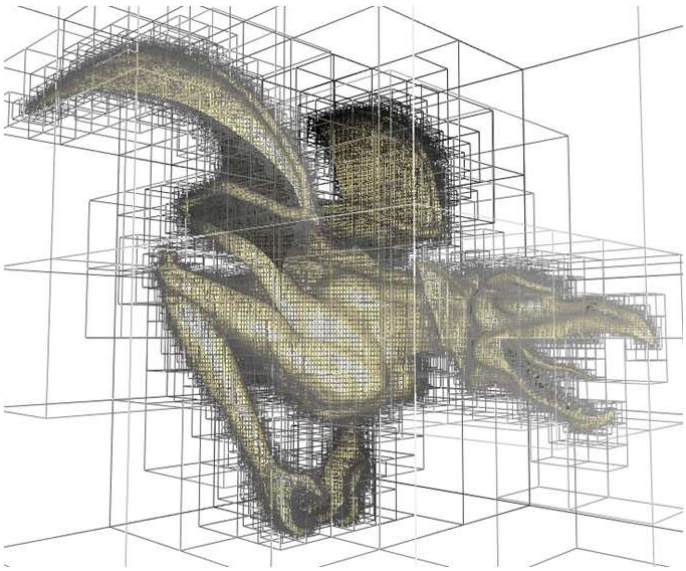


Figure 10: An octree around a complex model. Note how sparse regions are encoded using a large box, and detailed regions have lots of small bounding boxes (recursion is high in these regions). This image was produced by Sylvian Lefebvre

Referring to figure 10, the ray is tested with the top bounding box, and if it is found to intersect, it will be tested with each of the 8 children. If not, we have saved testing with the complex model. If it intersects any children, the closest one is checked by checking the ray against each of its 8 children and so on recursively. This recursive search ends with leaf nodes. Each leaf node is either empty (in which case we return, and the recursion ensures the next box is checked), or has a pointer to a list of objects with which the ray is checked. If the ray hits any of the objects, we know it is the closest intersection because we have traversed the tree in order of proximity. We have therefore found the closest intersection by only testing boxes along the recursive descent and the few objects in the node. In the case of complex models we may have found our closest object with just a few intersection test rather than with millions of intersection tests.

## 10-2 SURFACE LIGHTING EFFECTS

An illumination model computes the lighting effects for a surface using the various optical properties that have been assigned to that surface. These properties include degree of transparency, color reflectance coefficients, and various surface-texture parameters.

When light is incident on an opaque surface, part of it is reflected and part is absorbed. The amount of incident light reflected by the surface depends on the type of material. Shiny materials reflect more of the incident light, and dull surfaces absorb more of the incident light. For a transparent surface, some of the incident light is also transmitted through the material.

Surfaces that are rough, or grainy, tend to scatter the reflected light in all directions. This scattered light is called **diffuse reflection.** A very rough, matte surface produces primarily diffuse reflections, so that the surface appears equally bright from any viewing angle. Figure 10-7 illustrates diffuse light scattering from a surface. What we call the color of an object is the color of the diffuse reflection when the object is illuminated with white light, which is composed of a combination of all colors. A blue object, for example, reflects the blue component of the white light and absorbs all the other color components. If the blue object is viewed under a red light, it appears black since all of the incident light is absorbed.

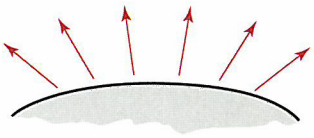


FIGURE 10-7 Diffuse reflections from a surface.

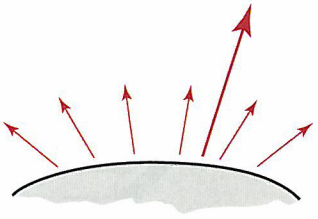


FIGURE 10-8 Specular reflection superimposed on diffuse reflection vectors.

In addition to diffuse light scattering, some of the reflected light is concentrated into a highlight, or bright spot, called **specular reflection.** This highlighting effect is more pronounced on shiny surfaces than on dull surfaces. And we can see the specular reflection when we look at an illuminated shiny surface, such as polished metal, an apple, or a person's forehead, only when we view the surface from a particular direction. A representation of specular reflection is shown in Fig. 10-8.

Another factor that must be considered in an illumination model is the **background light** or **ambient light** in a scene. A surface that is not directly exposed to a light source may still be visible due to the reflected light from nearby objects that are illuminated. Thus, the ambient light for a scene is the illumination effect produced by the reflected light from the various surfaces in the scene. Figure 10-9 illustrates this background lighting effect. The total reflected light from a surface is the sum of the contributions from light sources and from the light reflected by other illuminated objects.

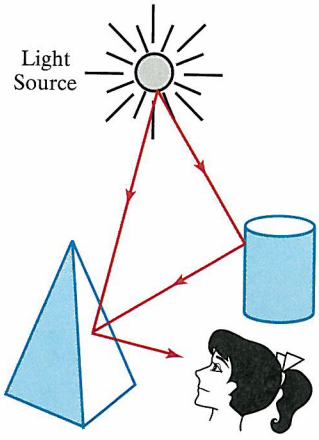



FIGURE 10-9 Surface lighting effects are produced by a combination of illumination from light sources and reflections from other surfaces.

## 10-3 BASIC ILLUMINATION MODELS

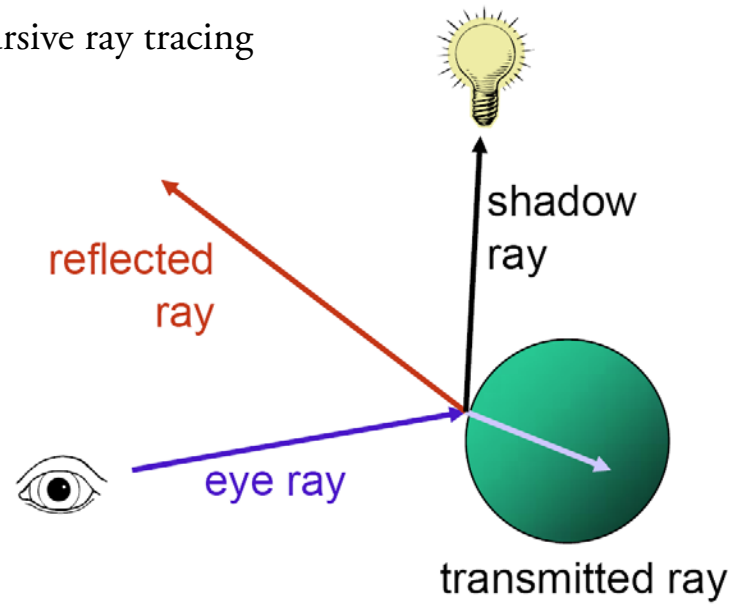
Accurate surface lighting models compute the results of interactions between incident radiant energy and the material composition of an object. To simplify the surface-illumination calculations, we can use approximate representations for the physical processes that produce the lighting effects discussed in the previous section. The empirical model described in this section produces reasonably good results, and it is implemented in most graphics systems.

Light-emitting objects in a basic illumination model are generally limited to point sources. However, many graphics packages provide additional functions for dealing with directional lighting (spotlights) and extended light sources.

### Ambient Light

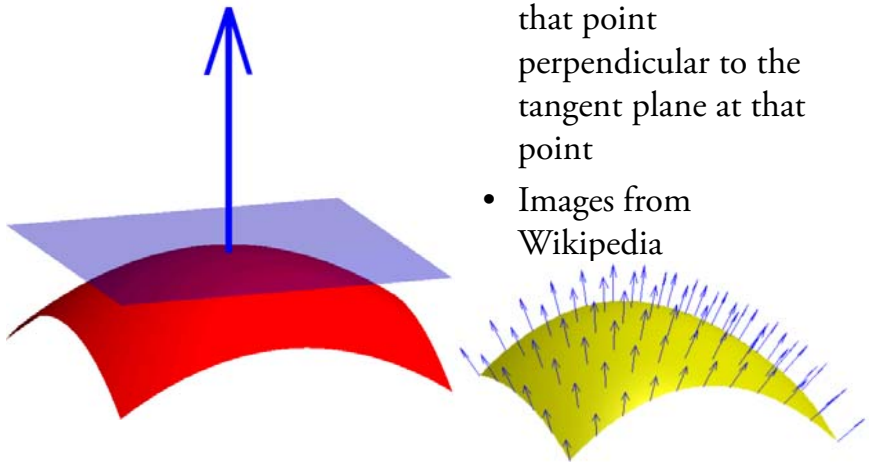
In our basic illumination model, we can incorporate background lighting by setting a general brightness level for a scene. This produces a uniform ambient

Recursive ray tracing



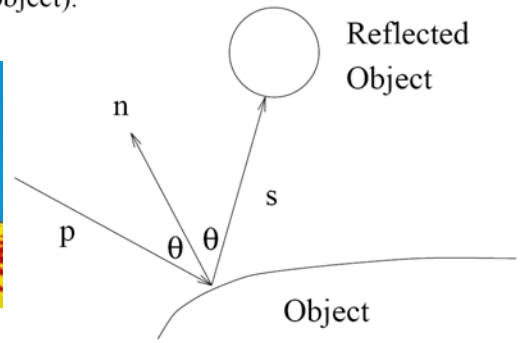

# Surface Normals

- A surface normal at a point is a vector from that point perpendicular to the tangent plane at that point
- Images from Wikipedia

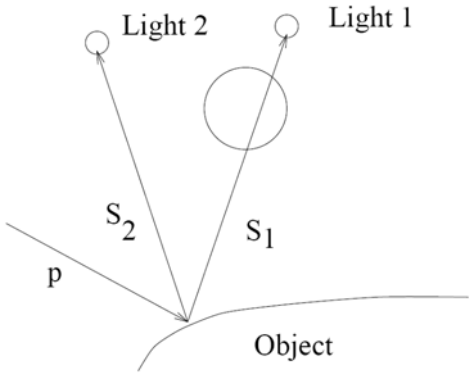


# Reflections

n = surface normal.
$\theta$ = angle of incidence (physics of reflection)
p = primary ray (from pixel).
s = secondary ray (from object).




# Shadows



# Transparency

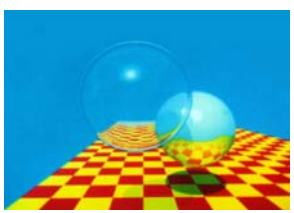
$\theta$ = angle of refraction (physics of light moving from one medium to another.)

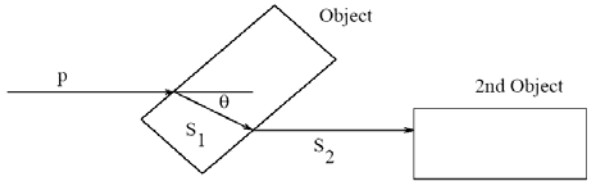
Snell's Law

p = primary ray.

$S_1$ = first secondary ray.

$S_2$ = second secondary ray.



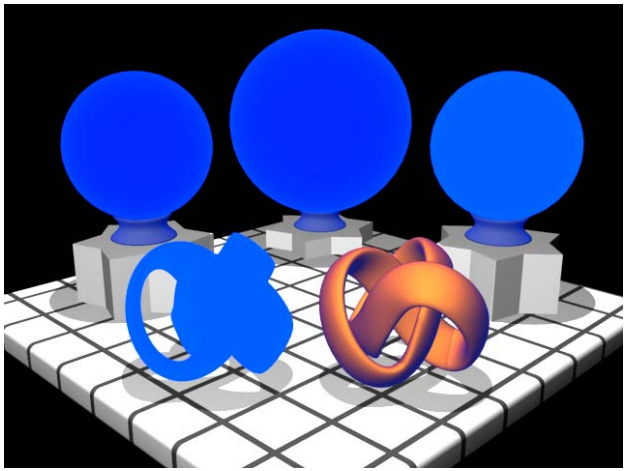



# Recursive Ray Tracing

- Each reflected or refracted ray is treated like a primary ray – i.e. can spawn shadow rays, reflected rays and refracted rays
- Recursive cut-off (e.g. what if a ray bounces between two mirrors – code does not terminate)
- How does that affect things?

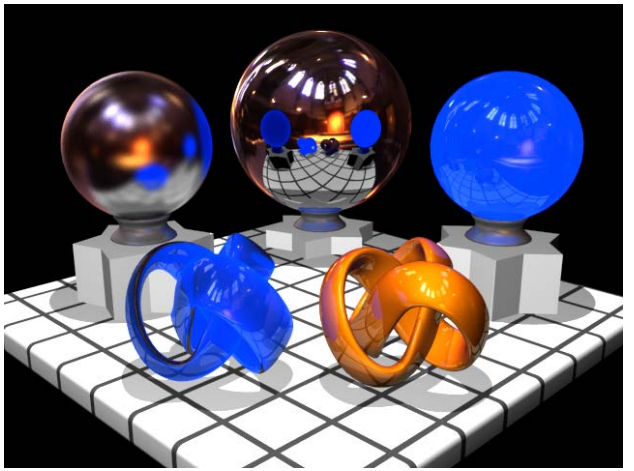
# Recursive Depth

- Primary rays (1)



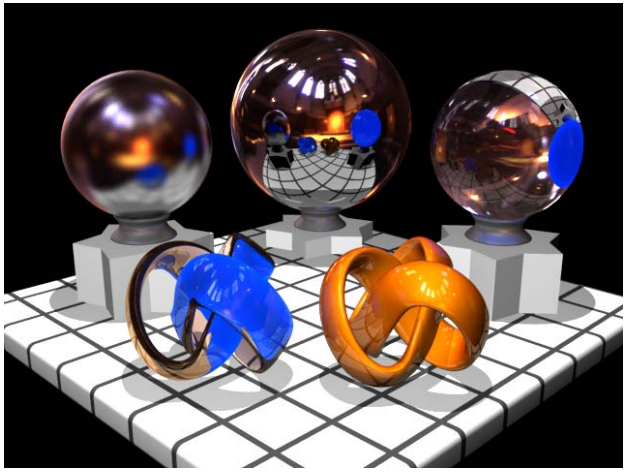
# Recursive Depth

- Depth= 2



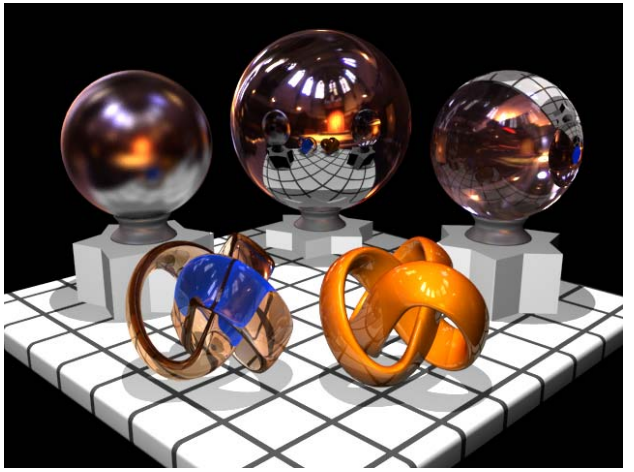
## Recursive Depth

- Depth= 3



## Recursive Depth

- Depth= 5



## Recursive Depth

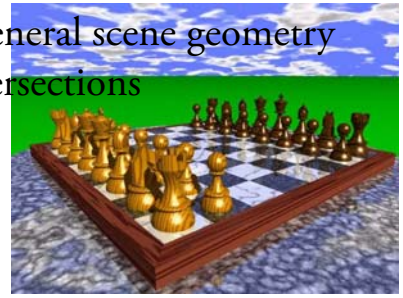
- Depth= 10

## Computational Expense

- No reflections, shadows or transparency
- Chess board – 32 pieces with 100 triangles each, extra 20 triangles for board, plane, sky etc. = 3220 triangles
- Image, 1000x1000=1 million pixels
- Algorithm – for each pixel, solve ray with each triangle. Select the closest hit
- 1000x1000x3220=3.22 billion intersection calculations!!

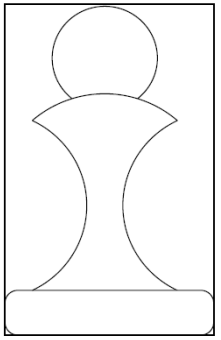
# Bounding Volumes

- Artificially surround each chess piece by a cylinder (easy to calculate ray-cylinder intersection)
- What does a ray hit?
- First trace it against the general scene geometry and 32 cylinders = 52 intersections



# Example

- Find closest hit point
- If the closest hit point is a cylinder, then intersect ray with 100 triangles within.
- Many rays will require just these 152 intersections
- Some will "miss" the chess piece although they hit the cylinder



# Example

- Assume 60% of rays miss cylinders, 30% hit a triangle in the first cylinder and 10% hit a triangle in the second cylinder.
- Therefore intersections ≈
  0.6x1000x1000x52
  0.3x1000x1000x152+
  0.1x1000x1000x252=102 million (about 30 times faster in this case)
- Why stop there? – hierarchical bounding volumes

# Hierarchical bounding volumes

- 1 box around the whole of the chess board and pieces (= 6 intersection tests)
- 3 other planes for the sky and two ground textures (=3 intersection tests)
- Inside the chess box, place 2 boxes – one over the white pieces, and one over the black (each 6 tests)
- Inside those have the 32 cylinders (1 test)
- Inside each cylinder have 3 cylinders (3 tests)
- Inside each of those have 30-40 triangles (30-40 tests)

# Hierarchical bounding volumes

- At the first level of our hierarchy, 9 tests are used to see if we have hit sky, ground or are in the region of the board.
- If we hit board region, another 6 tests see if we have hit white or black chess pieces
- If we have hit white, another 16 tests see if we have hit a chess piece
- Another 3 determine which part
- And a maximum of 40 to determine the triangle
- Total for a hit chess piece (previously 152) = 74

# Example

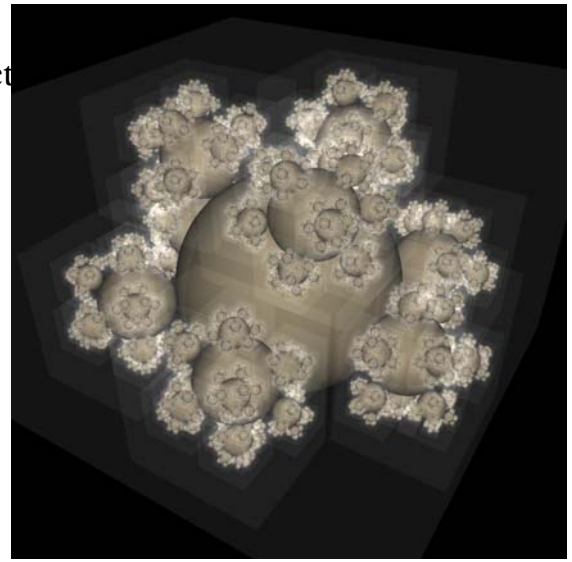
- Using same assumptions as before: 60% of rays miss board, 30% hit a triangle in the first cylinder and 10% hit a triangle in the second cylinder.
- Therefore intersections ≈
  0.6x1000x1000x9
  0.3x1000x1000x74+
  0.1x1000x1000x117=39.3 million (about 100 times faster in this case)

# Hierarchical Bounding Volumes

- Advantages
  - Fast ray tracing times
  - User can decide on tight fitting objects
  - User can decide logical splitting of scene
- Disadvantages
  - User interaction required
  - Could take a lot more time than is saved
  - Not all scene modellers will understand BVs

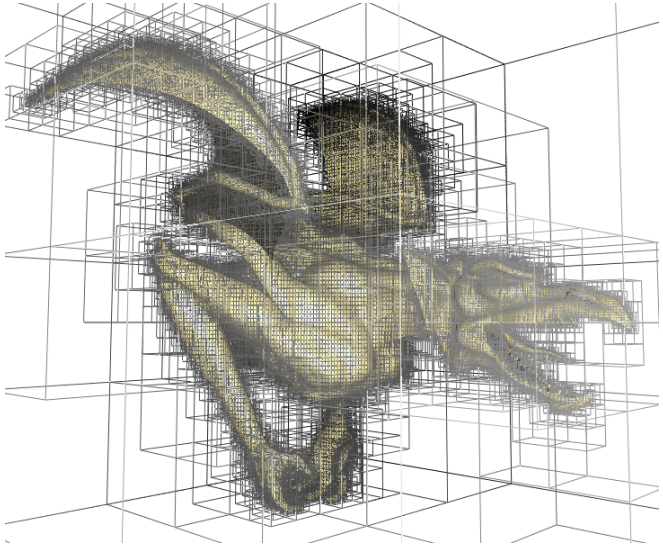
# Hierarchical Bounding Volumes

- Andreas Diet

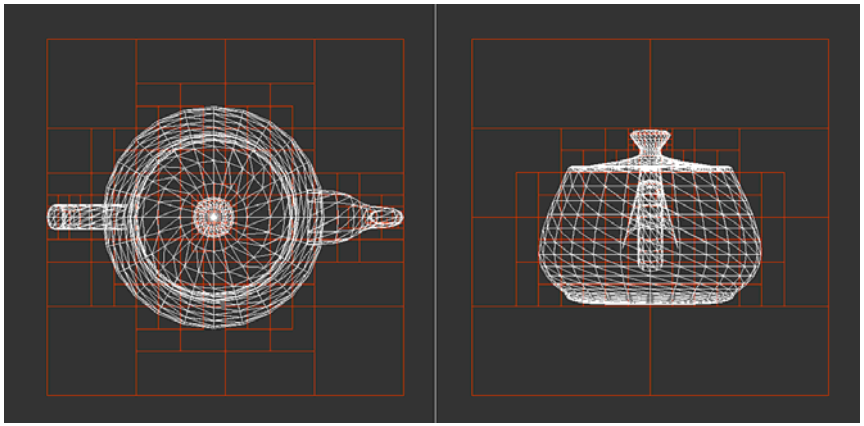


# Octrees

- Image from Sylvian Lefebvre
- (max depth= 12)

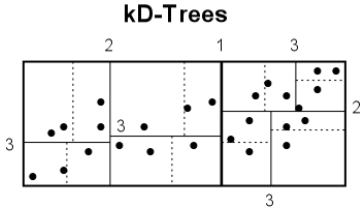


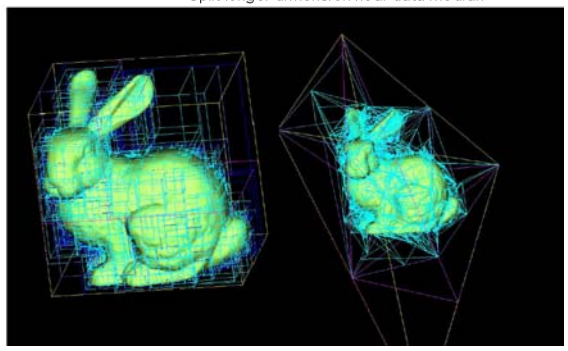
# Octrees



# Others

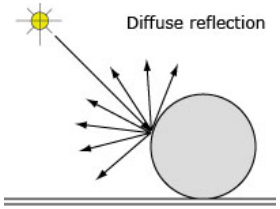
- kd-trees (cut is not necessarily in the centre)
- bsp-trees (cut is a plane – not a cube)



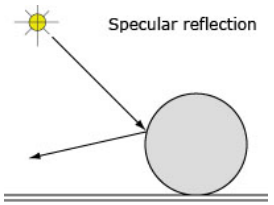



# Surface Lighting

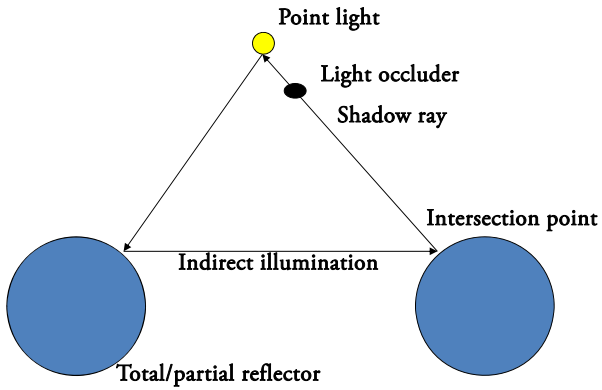
- Ambient
- Diffuse



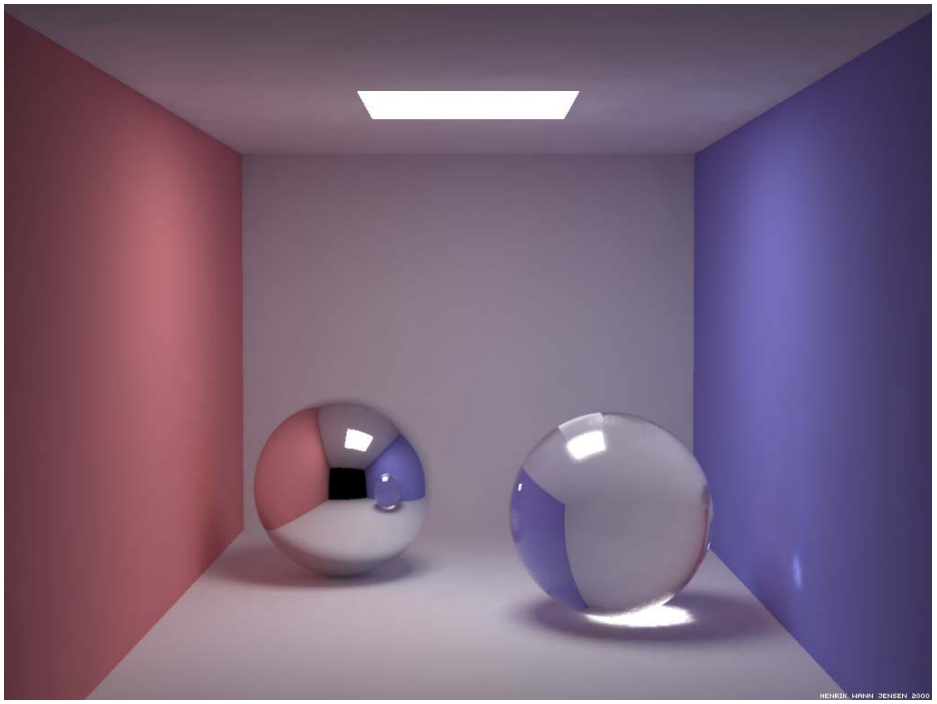

- Specular



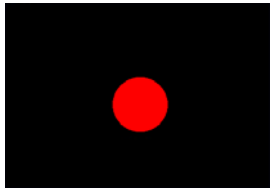
# Indirect Illumination




We've seen ray tracing using DIRECT ILLUMINATION
INDIRECT illumination is VERY expensive to calculate



# Ambient Reflection

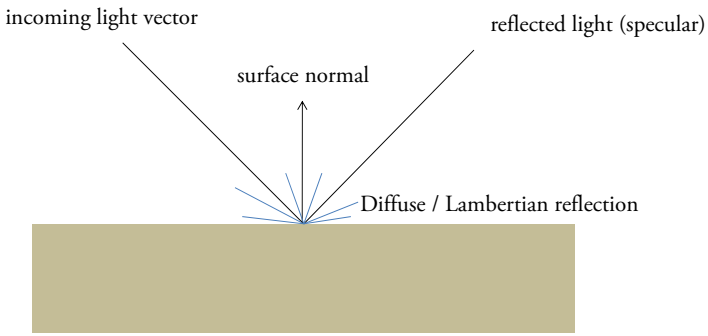


- Ambient reflection
- Pixel intensity, $I=I_aK_a$
- $I_a$=ambient light intensity
- $K_a$=ambient reflection coefficient of the surface – how much light it reflects (0-1)
- Usually r,g,b triples
- e.g. White light (1,1,1) x red object (1,0,0)=(1,0,0) (red)

# Light Sources

- To create a realistic rendering (during rasterization or ray tracing), we need to calculate the light incident upon the object
- A point (x,y,z) can be the spatial location of a **point light source**
- A vector (dx, dy, dz) can be the direction of a **directional light source**
- As we shall see, many of the following calculations require a vector
- Either a directional light source is used, or the direction can be calculated by subtracting the intersection point from the point light

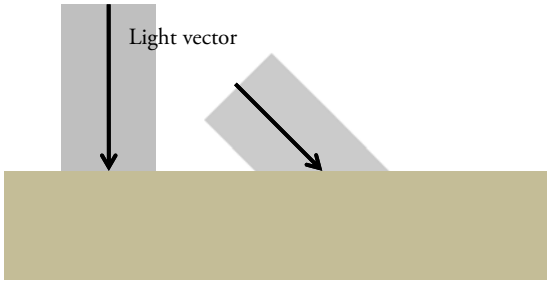
# Reflection of light




Light can be perfectly reflected. This is known as specular reflection and is often rendered using Phong's method.
Reflected light can be evenly spread. This is known as diffuse or Lambertian reflection.

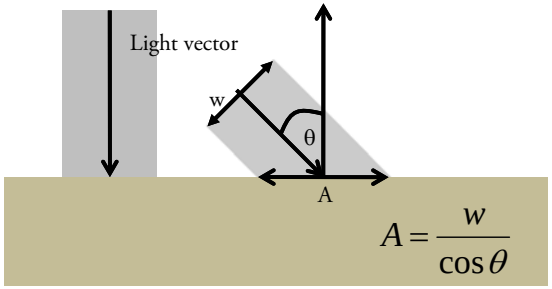
# Incident Light

Same shape at 90 and 45 degrees to the surface




Light energy per unit area is higher when at 90°, compared to 45°.
Example: The sun striking at the equator compared to higher latitudes.

# Incident Light




Light energy per unit area is higher when at 90°, compared to 45°.
Light Energy in box is I/w. Energy hitting the ground is $\frac{I}{A} = \frac{I \cos\theta}{w}$

# Lambertian / Diffuse reflection

The incoming light at an intersection point (ray tracing) is $I_{Light} \cos\theta$

The outgoing light at an intersection point, that is, what we see at the pixel is $I_{out} = I_{Light} \cos\theta$ , although, if cosθ<0, use 0* see lecture

cos θ is the angle between the surface normal **n** and the direction to the light **l**.
Both **n** and **l** are vectors.
If we assume the vectors are normalised, cos θ is the dot product of the two vectors

$$I_{out} = I_{Light} \mathbf{n}.\mathbf{l}$$

Remember, a vector (dx, dy, dz) can be normalised by finding its length, l:

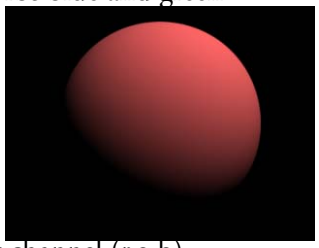
$$l = \sqrt{dx^2 + dy^2 + dz^2}$$

and dividing each component by the length to get a new vector (dx/l, dy/l, dz/l)

# Lambertian relection in colour

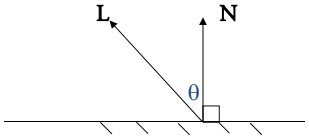
- Assign each object a diffuse reflection coefficient $k_d$ in each colour channel ($k_{d,r}$, $k_{d,g}$, $k_{d,b}$) where each is a value between 0 (absorb all light) and 1 (reflect all light)
- Example, a red object will be (1,0,0) – absorbs blue and green light, but reflects all red light
- The colour of a pixel $I_{out}$ is

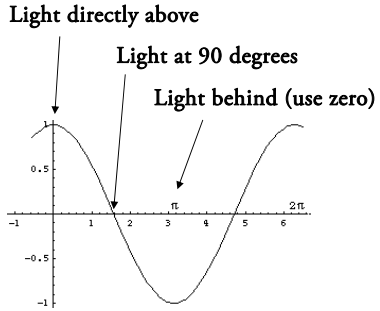
$$I_{out,r} = K_{d,r} I_{Light,r} \mathbf{n}.\mathbf{l}$$
$$I_{out,g} = K_{d,g} I_{Light,g} \mathbf{n}.\mathbf{l}$$
$$I_{out,b} = K_{d,b} I_{Light,b} \mathbf{n}.\mathbf{l}$$



- Note, we apply the equation in each colour channel (r,g,b)

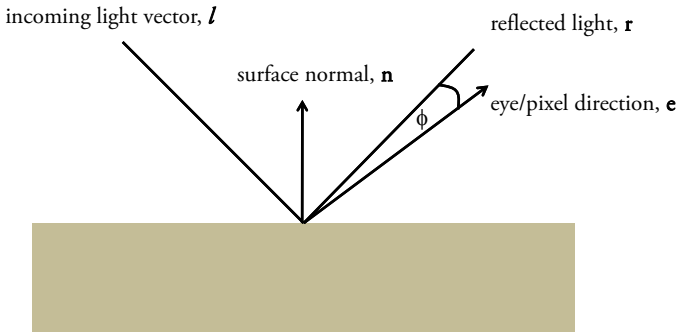
# cosθ



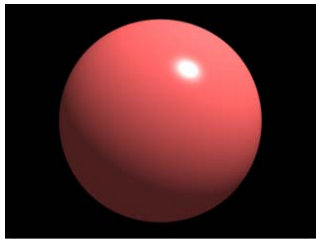

- N = surface normal
- L = direction to light source
- cosθ= N.L/|N||L|
- N and L are usually normalised during calculation, so
- cosθ= N.L
- if cosθ<0, use 0

# Specular Reflection




If the object is a perfect mirror, then the light will only be seen if ϕ=0
Less perfect mirrors will reflect light in a cone
Phong proposed that the reflected light is approximately cos $\phi^n$, where n is chosen to be a shininess coefficient for the object

# Phong Reflection



Phong's equation for specular reflection

$$I_{out,r} = K_{s,r} I_{Light,r} (\mathbf{r}.\mathbf{e})^n$$
$$I_{out,g} = K_{s,g} I_{Light,g} (\mathbf{r}.\mathbf{e})^n$$
$$I_{out,b} = K_{s,b} I_{Light,b} (\mathbf{r}.\mathbf{e})^n$$

cosϕ is calculated using the dot product of **r** and **e**
$K_s$ is the specular reflection coeffiecient for the surface.

# Simple Reflection Model

$$I_{out,c} = K_{a,c} I_{Light,c} + K_{d,c} I_{Light,c} \mathbf{n}.\mathbf{l} + K_{s,c} I_{Light,c} (\mathbf{r}.\mathbf{e})^n$$

where c takes on the values of the colour channels: r, g, and b
Note the equation adds together the **Ambient**, **Diffuse** and **Specular** terms

# Aliasing and Anti-Aliasing

- **Aliasing** is caused by taking samples at too low a frequency
- Red sinusoid is sampled regularly (blue squares)
- Reconstructing a curve through the samples, gives the wrong frequency (blue)

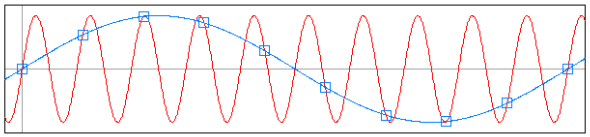


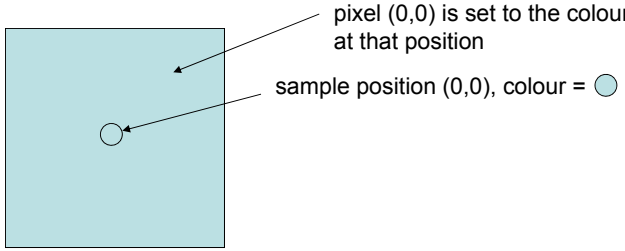
image from wikipedia

# Nyquist rate

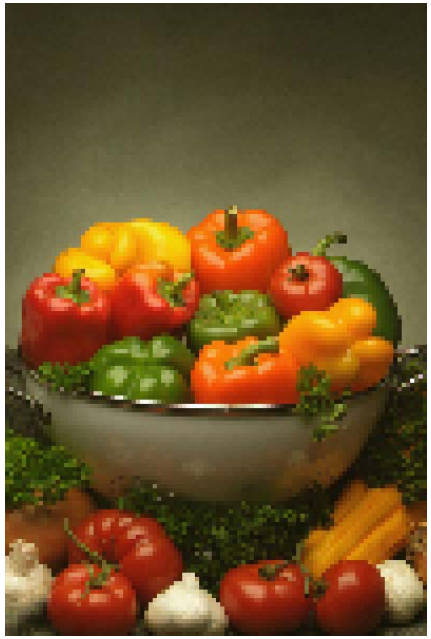
- To reconstruct the signal correctly, sample above the **Nyquist rate**
- If the frequency of the signal is f, samples must be taken at 2f or higher
- Higher rates, ensure more accurate reconstruction

# Graphics and aliasing

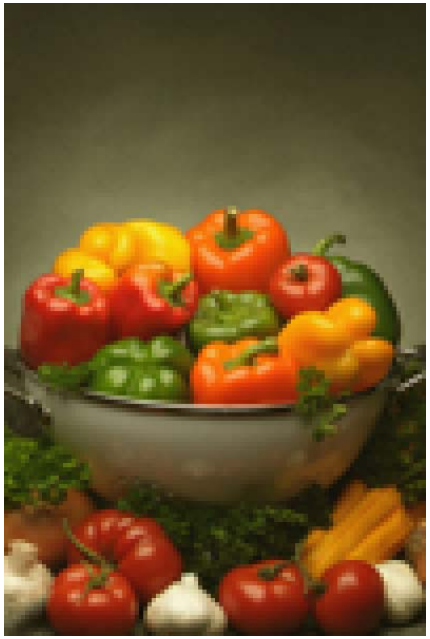
- Images are digital representations of continuous space (see first lecture)
- Simplest approach regards pixels as “samples” taken from centre




Reality (almost)



15,000 samples taken at pixel centres



240,000 samples with 4x4 averaging (gives 15,000)

Less noisy image (see bowl rim, broccoli and general improvements)

# Example

- Left – aliased image, right – anti-aliased using GPU (graphics hardware)

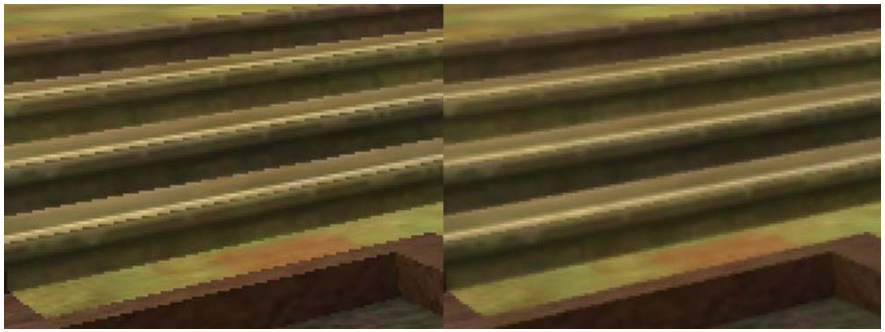
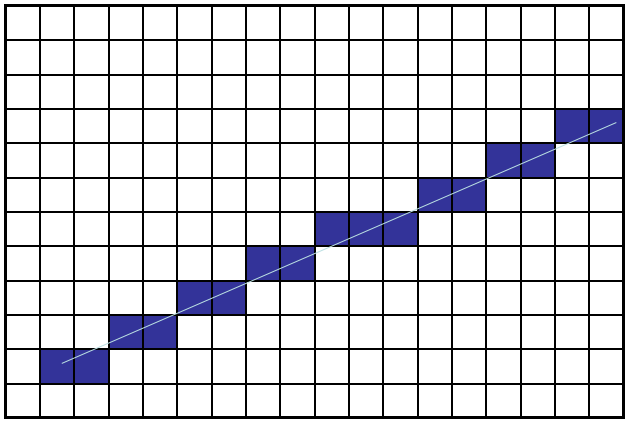


image from Toms Hardware

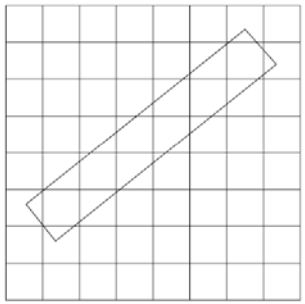
# Aliased Line

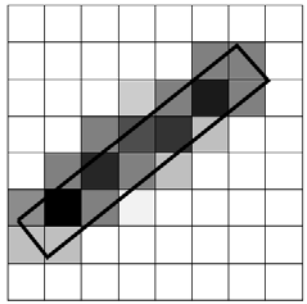
- high frequency+low sampling = jagged line



# Solution

- Unweighted area sampling
- Treat line as a 1 pixel thick box
- Measure the area = x%, set colour appropriately



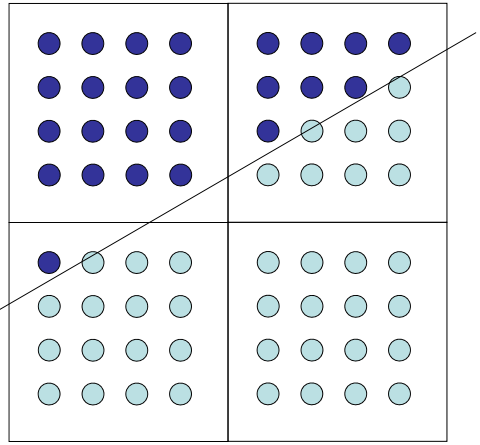


# Details

- Now a smooth transition from on to off
- Calculated using area of pixel and line (accurate)
- Or subdivide the pixel into a number of subpixels – e.g. 16, and check if each one is within line or not
- Gives 16 levels from on to off (see next slide)

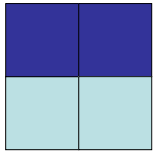
# Super-sampling

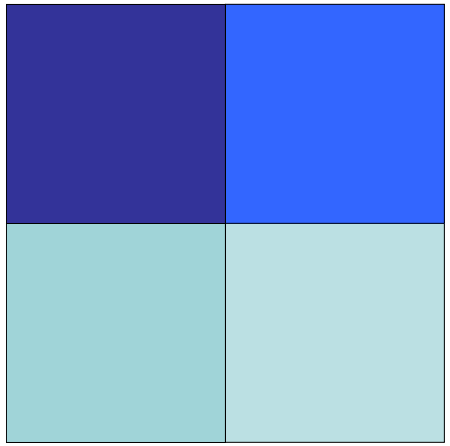
- 4 pixels
- Each sampled by 4x4=16 sub-pixels
- In next slide each pixel becomes average of 16 samples' colours



# Super-sampling

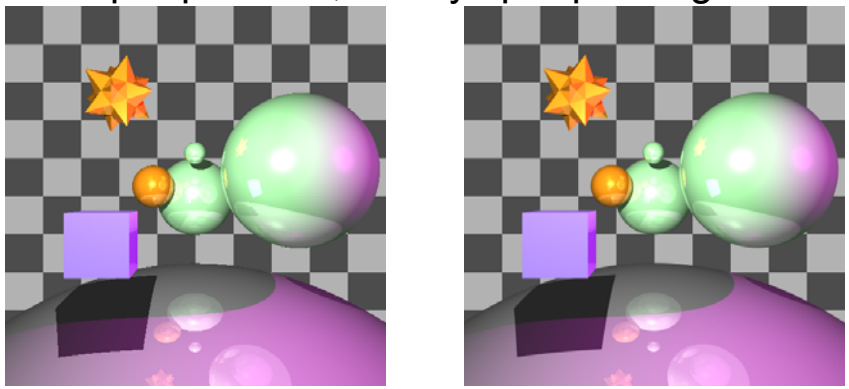
Without super-sampling pixels would have been set to these colours - check





# Super-sampling

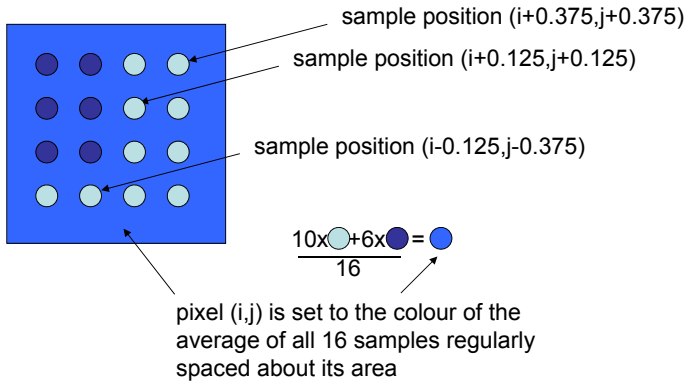
- e.g. Ray-Tracing – trace more rays per pixel – 1 per pixel left, 36 rays per pixel right



images from S. Depooter

# Anti-Aliasing – **Super-sampling** =More samples per pixel

- Other sampling positions can be used

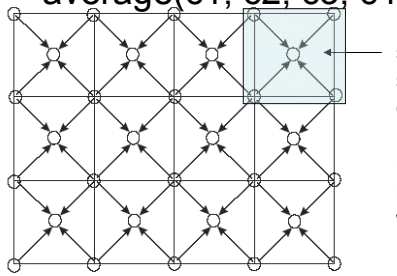



# Super-Sampling

- More samples
  - give better images
  - take much longer to compute
  - e.g. 4x4 samples will take 16 times longer
- "Trade-off"
  - Computer Graphics is all about trading off visual accuracy for better speed

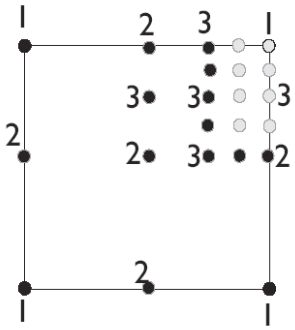
# Adaptive Super-sampling

- Decide upon a threshold, t, e.g. t=20
- Sample the scene at each pixel corner
- Obtain 4 colours – c1, c2, c3, c4
- If (max(c1, c2, c3, c4)-min(c1, c2, c3, c4))≤t then pixel colour = average(c1, c2, c3, c4)

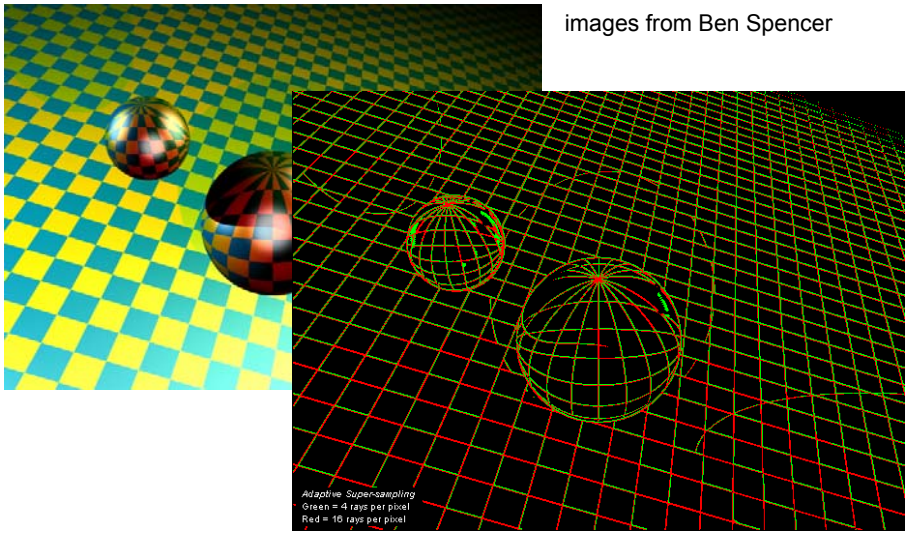


# Adaptive Super-sampling

- Pixels labelled 1 were first 4 samples
- If max-min>t, then take additional samples
- These are labelled 2 in the picture
- We now recurse on the 4 new sub-pixels
- Recurse each one until max-min ≤ t, or until a limit is reached
- Final pixel is average of all samples



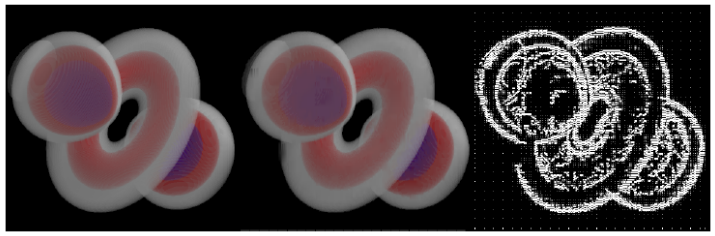

# Example




images from Ben Spencer

# Example

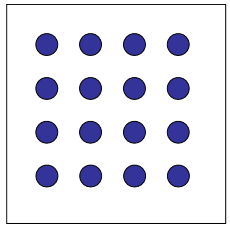
- 100x100 image = 10,000 samples
- 4x4 super-sampling = 160,000 samples
- 100x100 image, adaptively sampled = 34,335 samples in this case (samples shown on right)
- i.e. 3.4 times slower than 1 ray per pixel, but same visual quality as 4x4 supersampling (would take 16 times longer)



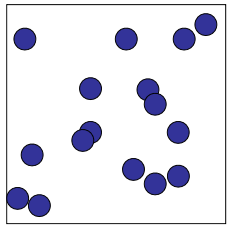
# Summary

- Aliasing problems caused by under-sampling
- In images – high frequency/contrast regions
- Solved using
  - unweighted area sampling (good for drawing primitives (like lines, circles, etc.)
  - weighted area sampling (ditto), not studied
  - super-sampling (good for ray tracing)
  - adaptive super-sampling (ditto)
- Each has advantages and disadvantages - learn

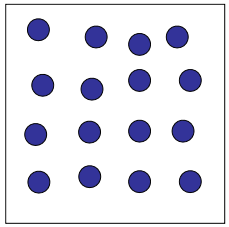
# Sample Positions



**Grid**, easy to compute, can still lead to aliasing



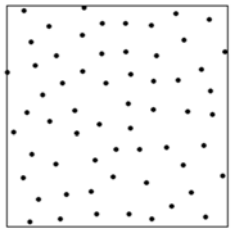
**Random** (or **stochastic** sampling), can lead to some areas being undersampled or oversampled



**Jitter**, using the grid, each sample is moved a random, but small amount, so that only one sample still occurs in each grid cell. The sample pattern is fast to compute, and does not suffer from too many problems

See next slide for 1 more

# Sample Positions



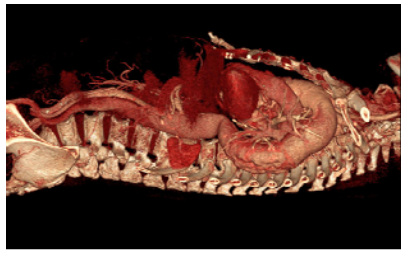
Poisson-disk

**Poisson disk**. The initial sample is randomly placed. During an iterative process, further samples are randomly placed, but are immediately tested against all existing samples, and removed if they are closer than a certain distance to an existing sample.
This produces an excellent sampling pattern, but the placement algorithm is computationally intensive (higher complexity than the previous ones)

Lean the name and advantages and disadvantages (and diagram) for each of the methods

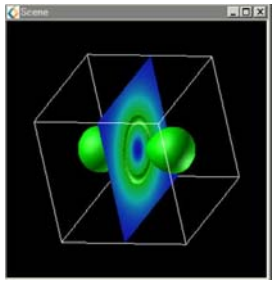
# Volume Data

- 3D array of data. Each volume element (voxel) is a value representing some measurement or calculation
- e.g. Magnetic resonance imaging (MRI) data: Strong magnetic field aligns magnetization of hydrogen atoms, and then measures radio waves they emit (body tissue contains a lot of hydrogen (water)). Computed Tomography (CT) data: Interior images of the body are calculated using many X-rays of the body.



# Volume Data

- e.g. simulation: probability of electron position around a $H_2$ molecule
- Machine representation simple: 3D array of char/integer/real, etc.
- Conversion to volume data: scanned (MRI/CT), numerical simulation (hydrogen data), or distance fields
- Rendering to screen (e.g. MIP – later slides)

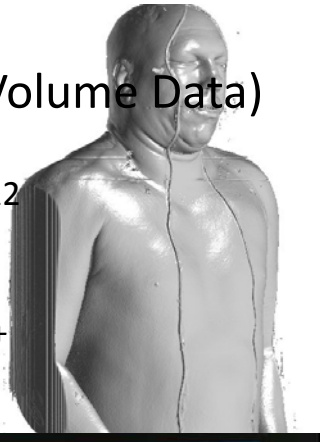


Hydrogen data

Distance around object

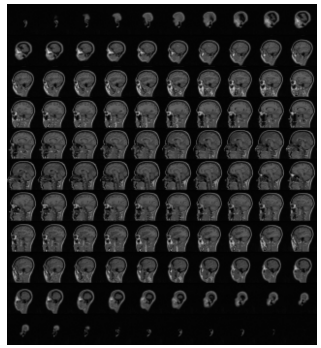
# Visible Human Project (Volume Data)

- CT scan 1800+ slices, each 512x512 voxels. Each voxel is 2 bytes (integer) ~ 1GB. MRI scan similar
- Body then photographed at 1800+ 1mm intervals (each photograph represents a slice through the body)



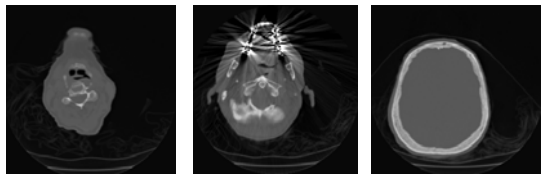
# Volume Rendering Input

- The data consists of a number of images representing slices through the body
- Each pixel in each image has the measured quantity (X-ray absorption for CT scans)
- Stacking the images creates a 3D array of “voxels” = volume elements (pixels=picture elements)



# Maximum Intensity Projection

- Type of rendering algorithm for volume data
- See lecture for example application for reading data
- Typical slices from CThead – bit.ly/cthead



# Maximum Intensity Projection

- Example for side view from CThead
- Rays are traced through the data set
- For each ray find the maximum value
- Then map it from [min,max] to [0,255]

# Maximum Intensity Projection

- Algorithm (for 256x256x113 data set – CThead):

  e.g. for top view, first loop over all the pixels:

  for j=row 0 to row 255

  for i=column 0 to column 255

  now set the maximum to a default value

  maximum=a small value

  now for the ray (i,j) going through the data from slice 0 to slice 112, find the maximum

  for k=slice 0 to slice 112

  maximum=max(cthead[k][j][i],maximum)

  still, for each ray (i,j), now the maximum has been found, set the colour of the pixel as usual (as in the code)

# Maximum Intensity Projection

- Further definitions
- each position in the 3D data set is known as a voxel
- a ray is a straight line “fired” through the data set
- Maximum intensity projection (MIP) has the advantage that it is close to a doctors understanding of an X-Ray
- Its disadvantage is that the depth of the structure is hard to distinguish (sometimes depth weighting is used)

# Colour Table

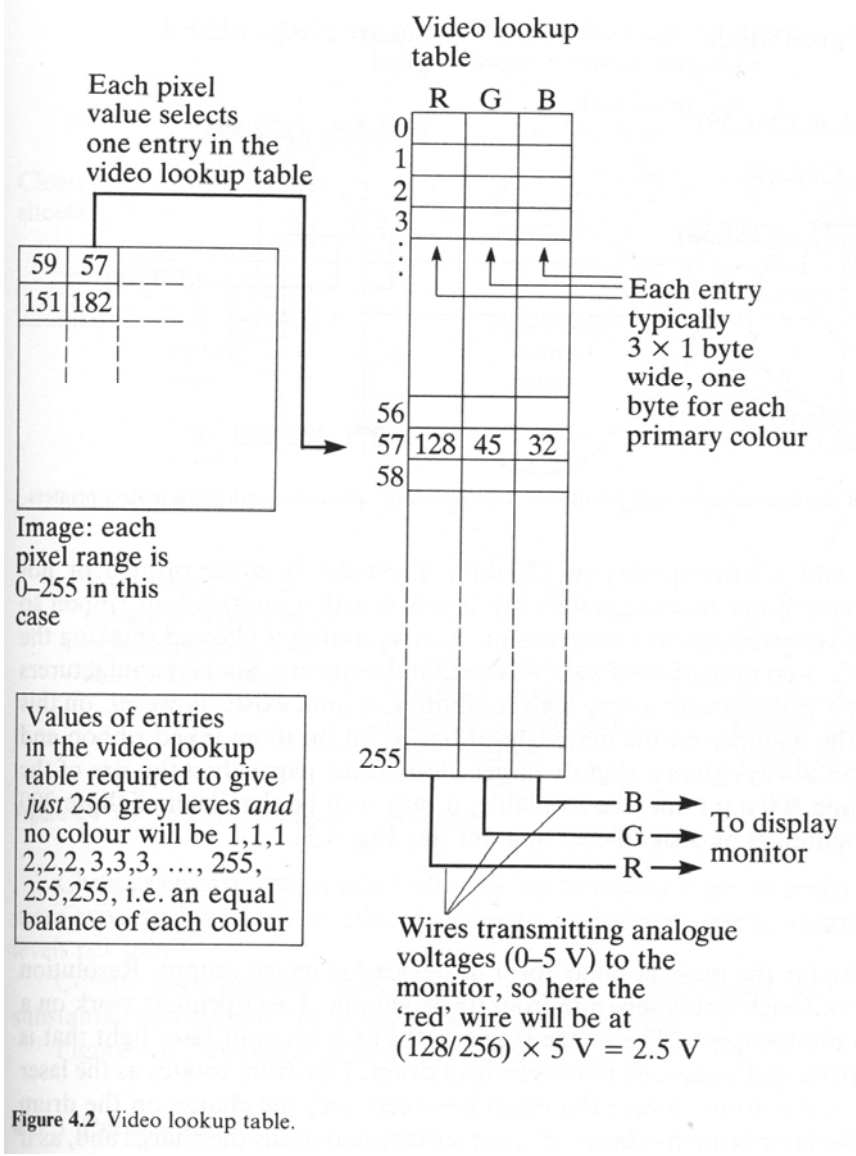
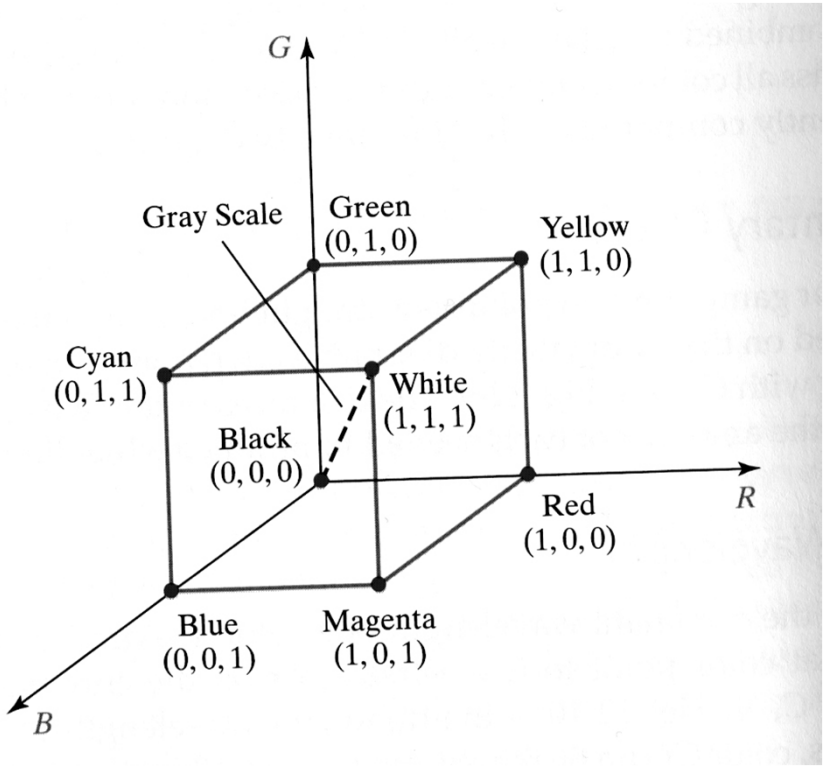



**Figure 4.2** Video lookup table.

- From p35, Introductory Computer Vision and Image Processing, Adrian Low, McGraw-Hill, 1991
- GIF
- Quantization: Reduces Colour
- Mach banding
- Problem reduced by dithering

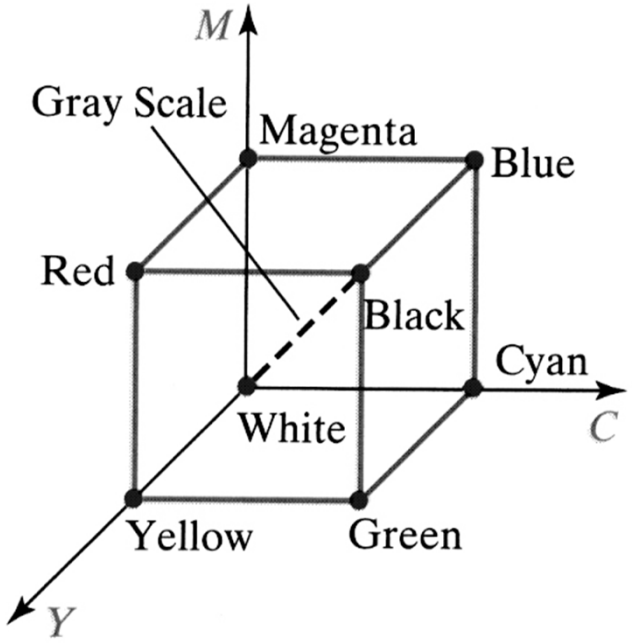
# RGB Colour Model




- From Hearn and Baker

- RGB Colour Cube
- (r,g,b) triples plotted on R,G,B axes in 3D
- Point in unit cube represents a colour
- Additive colour model

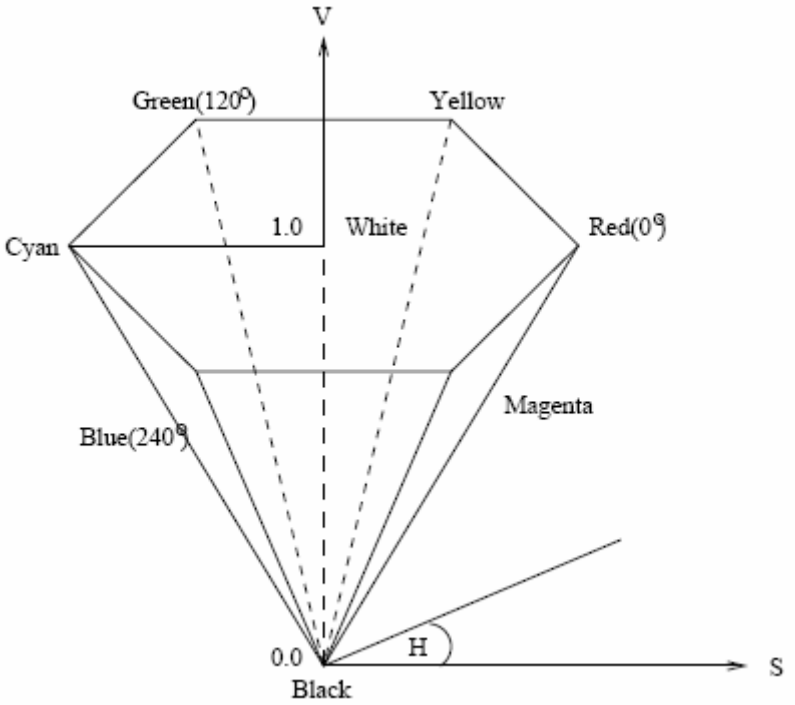
# CMY Colour Model




- From Hearn and Baker

- CMY Cube
- (c,m,y) triples plotted on C,M,Y axes in 3D
- Point in unit cube represents a colour
- Subtractive colour model (print)

# HSV Colour Model




- Hue, Saturation, Value
- (Sometimes HSB=brightness or HSI=intensity)
- H=Hue, angle, 0-360
- S=Saturation, 0-no colour, 1-pure colour
- V=Value, 0=black, 1=brightest
- Best for user interaction (for selecting colours – see Photoshop)