Link to [Part One](https://docs.google.com/document/d/1RTvoXfLpXr29sLKX-n-hwOw8_AgDwyGww_wAzQNyslg)

# 06 Illumination & Shading

| **Realistic Image Synthesis** | |
| --- | --- |
| Photorealism vs. physically-based realism.  **Examples**:  Film, visual effects, architecture, computer games, lighting engineering, predictive simulations. |  |
| **Non-Photorealistic Rendering** | |
| More suited for an artistic, technical or educational approach.  **Examples**:  Pen and ink drawings, cartoon-style drawings, technical illustrations, watercolour painting. |  |

## Rendering

| Fundamentally, **rendering** is concerned with determining the **most appropriate colour** to assign to a pixel associated with an object in a scene.  The colour of an object at a point depends on:   1. The **geometry** of the object at that point (e.g. the normal direction). 2. The position, geometry and colour of the **light sources** / luminaires. 3. The position and visual response of the **viewer**. 4. The **surface reflectance properties** of the object at that point. 5. The **scattering** by any participating media (e.g. smoke, rising hot air). |  |
| --- | --- |

### Rendering Algorithms

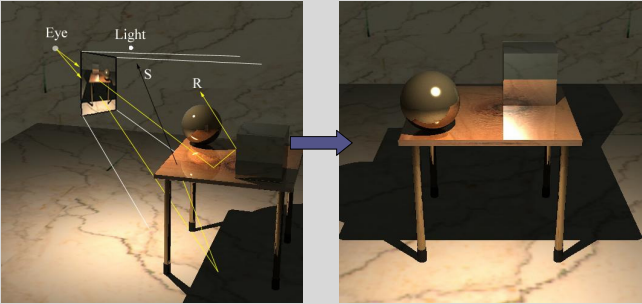
Rendering algorithms differ in the assumptions made regarding **lighting** and **reflectance** in the scene and solution space:

| **Local illumination**:  These consider lighting only from the **light sources**.  They ignore the effects of other objects in the scene (i.e. reflection off other objects / shadows). |
| --- |
| **Global illumination**:  These account for **all modes of light transport**. |
| **View-dependent**:  These determine an image by solving the illumination that arrives through the **viewport** **only**. |
| **View-independent**:  These determine the lighting distribution in an **entire scene**, regardless of viewing position.  Views are taken **after** light simulation by sampling the full solution to determine the view through the viewport. |

| **Local Illumination** | **Global Illumination** |
| --- | --- |
|  |  |

### View-Dependent Solutions

The following is an example of **ray tracing**:



The solution is determined only for directions through pixels in the viewport.

| **Advantages** | **Disadvantages** |
| --- | --- |
| * Only the **relevant parts** of the scene are taken into consideration. * It can work with **any kind** of geometry. * It can produce **high-quality** results. * In some methods, view-dependent portions of the solution can be **cached** (e.g. glossy reflections, refractions). * It requires **less memory** than a view-independent solution. | * It requires updating for **different camera positions**. |

| View-Independent Solutions The following is an example of **radiosity**.  A **single solution** for the light distribution in the entire scene is determined in advance.  We can take different snapshots of the solution from **different viewpoints** by sampling the complete solution for specific positions and directions. |  |
| --- | --- |

| **Advantages** | **Disadvantages** |
| --- | --- |
| * The solution needs to be computed only **once**. | * All scene geometry must be considered, even if it will **never** be visible. * The geometry is usually **restricted** to triangular / quadrangular meshes. (No procedural / infinite geometry). * Detailed solutions require **lots of memory**. * Only the **diffuse** portion of the solution can be cached. The view-dependent portions must still be computed (e.g. reflections). |

### Global Illumination Algorithms

The following algorithms are listed from **fast** → **slow**:

| **Z-buffer algorithms**:  These can compute **approximate** shadows and reflection from **planar surfaces**. |
| --- |
| **Ray tracing algorithms**:  These can determine **exact** shadows, reflection and refractive effects (e.g. transparency), assuming point light sources (cannot use volume). |
| **Radiosity algorithms**:  These compute **approximate** solutions assuming **no** shiny surfaces.  Light sources can be **arbitrarily large** and all surfaces **polygonal**. |
| **Path tracing algorithms**:  These employ an **expensive** Monte-Carlo solution to handle **arbitrary** geometries, reflectance and lighting. |

## Surface Reflectance

Much of current realistic image synthesis research is devoted to the modeling of surfaces, in particular the solving of **light reflection** off arbitrarily-complex surface geometries.

A surface **scatters** light that is **incident** to it. This scattering can theoretically distribute the scattered light in **any direction** from the scattering point.

Most algorithms make assumptions regarding the directions through which light is scattered.

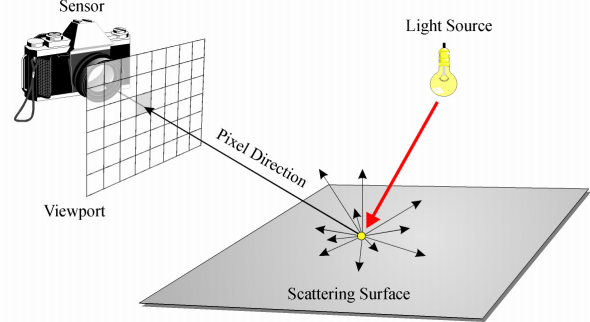
This is known as a **scattering distribution**.

**View-dependent** algorithms must determine the point in the scene which is visible through each pixel, then determine the light that is scattered towards the pixel.

|  | **BRDF** |
| --- | --- |

Energy is scattered from a surface in a **distribution** that depends on the surface’s microscopic geometry. This can be described as a function that records the **reflected energy per direction**.

**BRDF**: Bidirectional Reflectance Distribution Function.



An image is formed when light energy is scattered by surfaces in a scene / emitted towards the viewport.

### Solid Angles

A **solid angle** is a 3D equivalent of an angle. Its units are steradians .

| **Area** | **Solid Angle** |
| --- | --- |
|  |  |

The solid angle, subtended by a surface at a point , is related to the area of when projected towards onto a unit sphere centered at .

### Radiometric Units

| Power / **Flux** (Watts) is the total energy (Joules) leaving a surface per unit time.    The flux leaving a surface can change with the **position** on the surface.  Some points of a surface are brighter. |
| --- |

| **Radiosity**:  Flux per unit area. .   |  |  | | --- | --- | |
| --- | --- | --- |
| **Irradiance**:  Flux **arriving** per unit area. .    Irradiance at a point of the outer sphere is **less** than that at the inner one.  The energy received from a light source falls off with **squared distance** from it. |
| **Radiance**:  Radiosity per direction. .    Radiance is usually of **most interest** in computer graphics i.e. flux per area reflected towards the viewer. |

### Inverse Square Law

The **irradiance**  at point at distance from the light source can be calculated.

We know that the source radiates Watts in all directions.

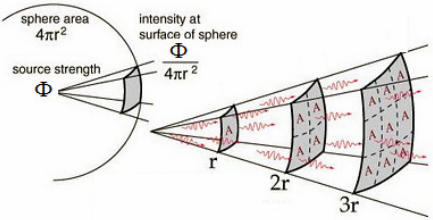
Therefore this power is irradiated through the sphere centered at the light source.

At distance from the source, the surface area of the sphere is .

Therefore the power per unit area at is .

**Note**:

This assumes the surface at is **perpendicular** to the direction towards the light source.



An object that is **twice the distance** from a point source of light will receive a **quarter of the illumination**.

#### Example

We have a 50W spherical bulb of diameter 20cm.

Calculate the irradiance on the bulb surface.

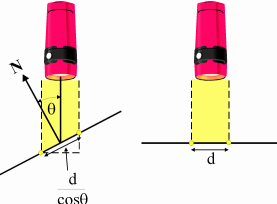
|  |
| --- |

Calculate the irradiance arriving on a surface 10cm away.

|  |
| --- |

### The Cosine Rule

A surface which is oriented **perpendicular** to a light source will receive more energy per unit area than one oriented at an angle.



The **irradiance** is proportional to .

As the **area increases**, the **irradiance decreases**.

(Angle ✕ Inverse square law)

#### Example

We have a 50W point light source.

The surface is located 40cm away.

Calculate the irradiance when the surface is tilted 30°.

|  |
| --- |

### BRDF (Bidirectional Reflectance Distribution Function)

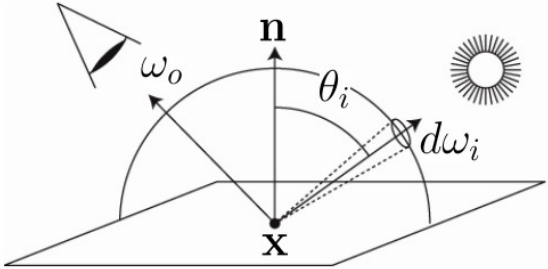
| BRDF is well-explained in [this video](https://youtu.be/bRwSZmJI87M).  A BRDF describes the **reflected radiance** given incident radiance where:   * is the **position**. * is the **incoming** direction. * is the **reflected** direction.   where:   * is the radiance. * is the differential solid angle. |  |
| --- | --- |

BRDF examples:

| **Ideal Specular** | **Rough Specular** |
| --- | --- |
|  |  |
| **Ideal Diffuse** | **Directional Diffuse** |
|  |  |

### Reflectance Equation

The reflectance equation relates **reflected radiance** to **incoming radiance** that is scattered according to the surface’s BRDF.



where:

* is the **reflected radiance**.
* is the domain of integration.
* is the BRDF.
* is the **incident radiance**.  
  This is a **hemisphere** if the surface is **opaque**.
* is the cosine of the incident angle.

The radiance equation often includes a **self-emitted term** to account for light sources.

where:

* is the **self-emitted radiance**.  
  This is non-zero for light sources.

We solve for at **each visible point** in the scene.

## Shading

| **Flat** | |
| --- | --- |
| We compute the illumination **once per polygon**.  This is applied to the whole polygon. |  |
| **Smooth / Gouraud** | |
| We compute the illumination at **each vertex** and interpolate the colour. |  |
| **Phong** | |
| We interpolate the normals and compute illumination at **every fragment** of the polygon. | |

### Flat Shading

| Flat shading gives low-polygon models a faceted look.  It works poorly if the model represents a **curved** surface.  A smooth appearance implies a **large number of polygons**.  Flat shading is advantageous in modeling **boxy objects**. |  |
| --- | --- |

#### Mach Banding

Mach banding is an optical illusion in which the perceived intensity change at edges are exaggerated by receptors in our eyes.



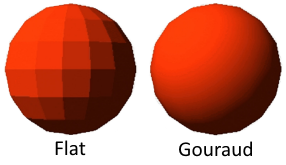
Abrupt changes in the shading of two adjacent polygons are perceived to be greater.

This makes the dark facet look darker and the light facet lighter.

### Smooth Shading

Smooth shading is used to approximate curved surface with a collection of polygons.

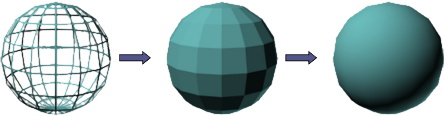
We calculate the illumination based on the approximation of a **curved surface**.



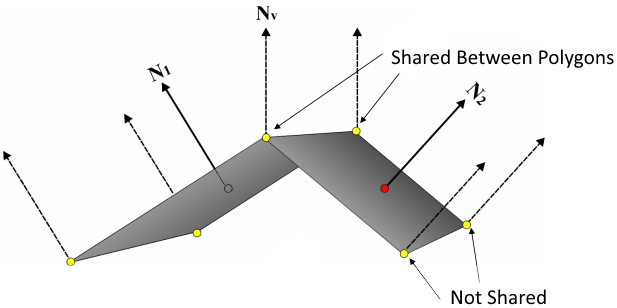
This does not change the geometry, so the **silhouette** is still polygonal.

### Surface Normal Interpolation

Often when we are approximating curved surfaced with polygons we get **edge artifacts** at polygon boundaries.



To combat this, we determine the **average normals** at the vertices of the polygons by averaging the normals of each polygon that shares a vertex.



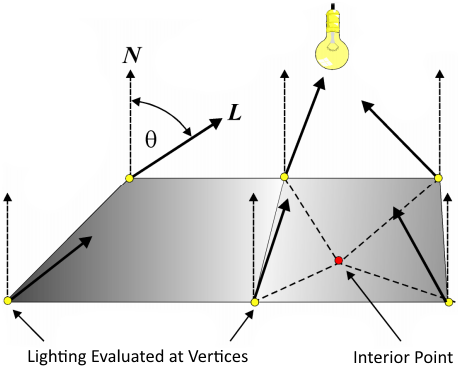
### Gouraud Shading

Gouraud shading is a method for **linearly interpolating** a colour across a polygon.



For each interior point in the polygon being shaded, we interpolate the intensity determined at the **vertices**. We need to do **lighting calculations** at the **vertices only**.

This is fine if the polygons are small. As the polygons increase in size so do the errors.



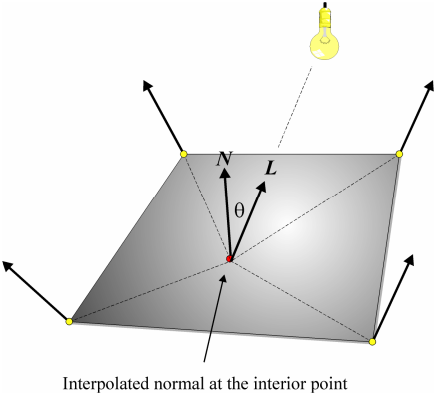
#### Interpolation Errors

| **Low Polygon Count** | **High Polygon Count** |
| --- | --- |
|  |  |

### Phong Shading

To improve upon Gouraud shading, we can **interpolate the normals** across the surface and apply the lighting model at each fragment.

All interpolated normals must be of **unit length** (normalised).



This method assumes that we are working with **polygonal models**.

Phong **shading** is **different** to the Phong **illumination model**.

| **Phong Shading** | **Gouraud Shading** |
| --- | --- |
|  |  |

* Phong shading vs. Gouraud shading:  
  Handles specular highlights much better.  
  Handles Mach bands better.  
  More expensive.

### Shading Models vs. Illumination Models

An **illumination model** captures **how light sources interact** with object surfaces.

A **shading model** determines **how to render faces** of each polygon in the scene.

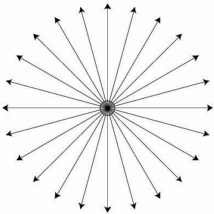
It describes how to interpolate over the faces of polygons given the illumination.

The shading model **depends** on the illumination model:

* Some shading models invoke an illumination model for **each pixel** (e.g. ray tracing).
* Some shading modes use the illumination model for **some pixels** and shade the remaining with **interpolation** (e.g. Gouraud shading).

### Light Sources

In order to simplify the solution to the radiance equation, we normally employ **isotropic point light sources** (radiate light equally in all directions) defined by a **position** and **colour**.



Normally we wish to associate a **radiance** with a light source.

However, the definition of radiance assumes an **area** over which energy is emitted.

Point light sources have **no area**, so we use **radiant intensity**  instead.

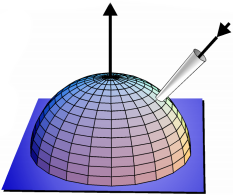
A point light source radiating energy **equally** in all directions has a radiant intensity of:

## Illumination Models

| **Ambient** | **Lambertian / Diffuse** | **Phong** |
| --- | --- | --- |
|  |  |  |

### Diffuse Light

The illumination that a surface receives from a light source is reflected equally in **all directions**.



The brightness of the surface is **independent of the observer**.

### Lambertian Illumination Model

We can use the **cosine rule** to implement shading of Lambertian / diffuse surfaces.

The **BRDF is constant** with respect to the reflected direction.

The surface may be characterised by a **reflectance**  rather than a BRDF.

The reflectance gives the ratio of total reflected power to total incident power.

#### Reflectivity

The reflectivity varies from zero (completely-absorbing black surface) to one (completely-reflective white surface).

There are **no natural** Lambertian surfaces, but matte paper is a good approximation.

To shade a diffuse surface we need to know:

1. The **normal** to the surface at the point to be shaded.
2. The **diffuse reflectance** of the surface.
3. The **positions** and **powers** of the light sources in the scene.

| We will assume **isotropic point** light sources.  The contribution from a single source is given by:  where:   * is the **reflected radiance**. * is the **BRDF**. * is the **incident radiance**. |  |
| --- | --- |

The brightness depends only on the angle between the direction to the **light source** and the **normal**.

The following Lambertian surfaces have varying lighting angles:



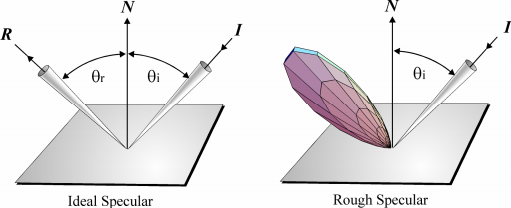
### Phong Illumination Model

**Specular** surfaces exhibit a high degree of coherence in their reflectance.

The **reflected radiance** depends heavily on the **outgoing direction**.

An ideal specular surface is **optically smooth**.

More specular surfaces reflect energy in a **tight distribution** (lobe) centered on the optical reflection direction.



To simulate reflection we should examine surfaces in the **reflected direction** to determine **incoming flux**. This would be a **global illumination**.

A **local illumination** approximation only considers the reflections of **light sources**.

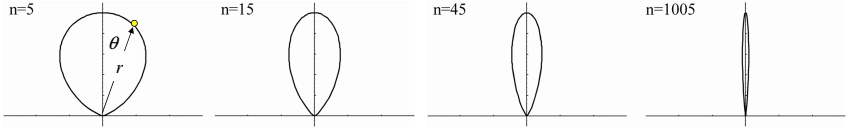
The Phong model is an empirical local model (based on observation) of shiny surfaces.

It is assumed that the BRDF of such surfaces may be approximated by a **spherical cosine function** raised to a **power**, known as the **Phong exponent**.

#### Function

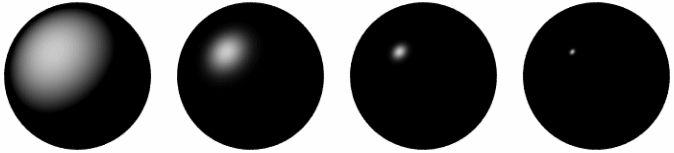
The cosine function defined on the sphere gives us a **lobe shape** which approximates the distribution of energy about a **reflected direction** controlled by the shininess parameter .

This is known as the **Phong exponent**.



In the limit the function becomes a single **spike** (i.e. ideal specular).

This is sometimes known as the **delta function**.

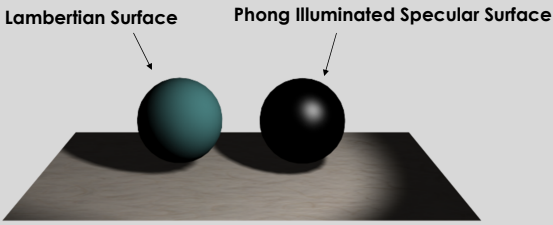


| where:   * is the **normalisation** term. * is the **specular reflectivity**. * is the **cosine lobe**. * is the light source **irradiance**. |  |
| --- | --- |

### Lambertian vs. Phong

Surface reflection:

| **Lambertian / Diffuse** | **Phong** |
| --- | --- |
| Dependent on:   1. Orientation to light source. 2. Distance to light source.   Independent of:   1. Orientation to viewer. 2. Distance to viewer.   This leads to a **matte** appearance | Dependent on:   1. Orientation to light source. 2. Distance to light source. 3. Orientation to viewer. 4. Distance to viewer.   This leads to a **glossy** appearance with **highlights**. |



### Ambient Illumination Model

Local illumination models only account for light scattered from **light sources**.

Light may be scattered from **all surfaces** in a scene. We typically miss 50% of light.

The **ambient term** is a coarse approximation of this missing flux.

It is defined by the ambient coefficient :

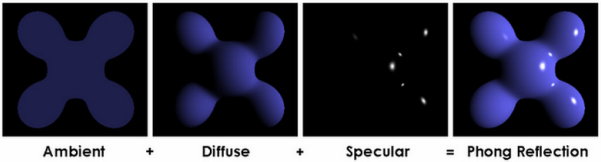
This term is **constant** everywhere in the scene.

The ambient term is sometimes **estimated** from the total powers and geometries of the light sources.

### Combining Terms

The complete Phong illumination model includes:

1. **Lambertian model** for diffuse reflection.
2. **Cosine lobe** for specular reflection.
3. **Ambient term** to approximate all other light.



A material must therefore have associated **material data** to define how diffuse, specular and ambient it is.

Surface data:

* is the **ambient** reflectance.
* is the **diffuse** reflectance.
* is the **specular** reflectance.
* is the **phong exponent**.

### Point Light Sources

The irradiance of a surface due to a point light source obeys the inverse square law:

However, this often makes it difficult to control the lighting in the scene.

We employ a **less accurate** but **more flexible** model of irradiance:

where:

* is a constant attenuation factor.
* is a linear attenuation factor.
* is a quadratic attenuation factor.

| **Constant** | **Linear** | **Quadratic** |
| --- | --- | --- |
|  |  |  |
| **No attenuation** at all. | Diminishes as it travels from source. | Diminishes faster the further it travels from source. |

### Phong Illumination Model

where:

Notes:

* The ambient term is **not** affected by light.
* The diffuse term is affected by **light**, but not by viewing angle.
* The specular term is affected by **viewing angle**.

For **multiple** light sources:

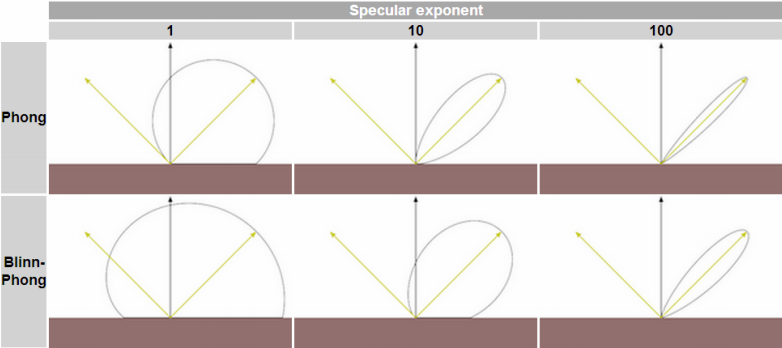
| Determining the Reflected Vector |  |
| --- | --- |

### Blinn-Phong

A problem with computing the **perfect reflection vector** is that is different for each point on a surface. We must recompute for every polygon, which is **slow**.

| Blinn proposed an alternative formulation which employs the **half vector** in place of .  The half vector is a vector with a direction half-way between the **eye vector** and **light vector**. |  |
| --- | --- |

The **specular term** is:



Both the Phong and Blinn-Phong reflectance functions cause a highlight to appear around the direction of reflection.

Blinn-Phong is **cheaper** to calculate, but appears **more spread out** at the same shininess.

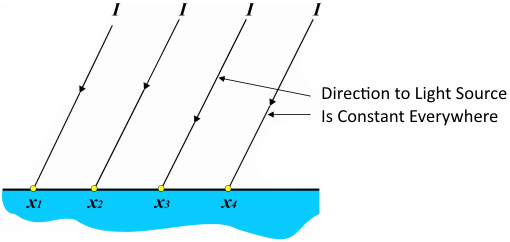
| **Phong** | **Blinn-Phong** | **Blinn-Phong**  (Higher Exponent) |
| --- | --- | --- |
|  |  |  |

### Other Light Source Types

We can extend the functionality of the illumination model by admitting a number of other light source types:

1. **Directional**:  
   The source is assumed to be at **infinity** and is represented by a **direction** rather than a position.
2. **Spot lights**:  
   We can admit illumination only within a **restricted angle**.

#### Directional Source



We don’t need to calculate for each .

The light radiant intensity is **constant** i.e. it does not vary with distance.

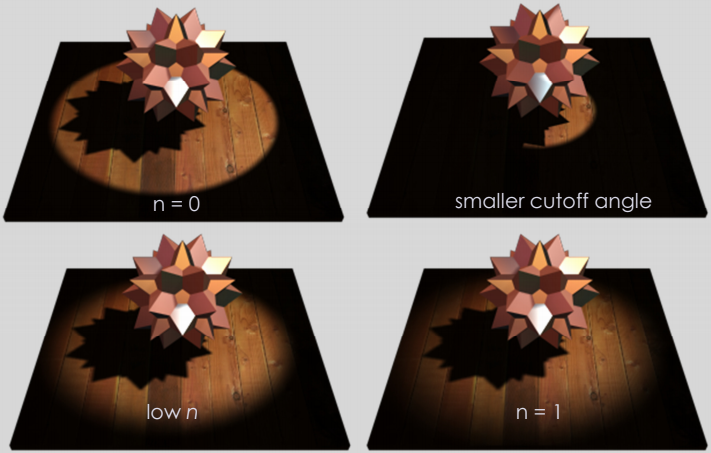
#### Spot Lights

| A light source is defined by a:   1. Position 2. Direction 3. Cutoff angle 4. Exponent   The radiant intensity of the spot light in a given direction is defined by a modal similar to the Phong illumination model. |  |
| --- | --- |

If then:

where:

* is the **spotlight attenuation**.



### Incorporating Colour

Many of the systems that implement the Phong model only differ in how they **handle colour**.

Typically, we handle diffuse and specular illumination **separately**:

1. **Diffuse**:  
   The reflected light is coloured by **selected absorption** by the surface.  
   i.e. A green surface absorbs all wavelengths except for green.
2. **Specular**:  
   Reflected light interacts once with the surface and is thus **not coloured** by it.  
   The reflection of a light source takes on the colour of the **source**.

The [Phong model](#_5hcm202i2ho5) becomes:

However, usually for flexibility the ambient, diffuse and specular reflections are scaled **independently** by **colour vectors**:

The model is applied using colour vectors, yielding the final colour vector to assign to a pixel.

Only the **specular term** is scaled by the **light’s colour**.

## Lighting in OpenGL

**Blinn-Phong** used to be specified as part of the OpenGL fixed-function pipeline.

### Light Source

We must specify:

1. **Colour**: Ambient, diffuse and specular.
2. **Location** & **direction**.

| color4 light\_diffuse, light\_specular, light\_ambient; point4 light\_position; float attenuation\_const, attenuation\_linear, attenuation\_quadratic; |
| --- |

### Materials

| color3 ambient = color3(0.2); color3 ambient = color3(0.2); color3 ambient = color3(0.2); float shininess; // Phong exponent. |
| --- |

### Implementing a Lighting Model

We’ll take the simple version of the **Blinn-Phong** model, using a **single** light source.

Multiple sources is **additive**, so we can repeat the calculations and add the contributions.

We have three choices as to where we do the calculation, with different levels of efficiency:

1. Application
2. Vertex shader
3. Fragment shader

#### Vertex Shader

| in vec3 vertex\_position; in vec3 vertex\_normal; uniform mat4 proj\_mat, view\_mat, model\_max; out vec3 position\_eye, normal\_eye;  void **main**() {  mat4 viewSpace\_mat = view\_mat \* model\_mat;  position\_eye = vec3(viewSpace\_mat \* vec4(vertex\_position, 1));  normal\_eye = vec3(viewSpace\_mat \* vec4(vertex\_normal, 1));  gl\_Position = proj\_mat \* vec4(position\_eye, 1); } |
| --- |

### Normals

If we are doing any scaling in the model matrix where the axes are different (e.g. horizontal stretch) then the model matrix will **incorrectly** scale the **normal**.

In this case, we would need to calculate a **new normal matrix** to correct the problem.

| mat4 normal\_mat = transpose(inverse(view\_mat \* model\_mat)); // or mat4 normal\_mat = inverse(transpose(view\_mat \* model\_mat)); |
| --- |

### Ambient Intensity Term

| in vec3 position\_eye, normal\_eye;  // Fixed point light properties. vec3 light\_position\_world = vec3(0, 0, 2); vec3 Ls = vec3(1); // White specular colour. vec3 Ld = vec3(0.7); // Dull white diffuse light colour. vec3 La = vec3(0.2); // Grey ambient colour.  // Surface reflectance. vec3 Ks = vec3(1); // Fully reflect specular light. vec3 Kd = vec3(1, 0.5, 0); // Orange diffuse surface reflectance. vec3 Ka = vec3(1); // Fully reflect ambient light. float specular\_exponent = 100; // Specular 'power'.  out vec4 fragment\_colour; // Final colour of surface.  void **main** () {  // Ambient intensity.  vec3 Ia = La \* Ka;   // Diffuse intensity.  vec3 Id = vec3(0); // TODO.   // Specular intensity.  vec3 Is = vec3(0); // TODO.   fragment\_colour = vec4 (Is + Id + Ia, 1); } |
| --- |

### Diffuse Intensity Term

Diffuse light should be:

* Brightest when the surface is **facing** the light (= 1).
* Not lit at all when the surface is **perpendicular** to the light (= 0).

We use the dot product.

| // Raise light position to eye space. vec3 light\_pos\_eye = vec3(view\_matrix \* vec4(light\_position\_world, 1)); vec3 distance\_to\_light\_eye = light\_pos\_eye - position\_eye; vec3 direction\_to\_light\_eye = normalize(distance\_to\_light\_eye);  float dot\_prod = dot(direction\_to\_light\_eye, normal\_eye); dot\_prod = max(dot\_prod, 0); // Prevent negative dot product.  vec3 Id = Ld \* Kd \* dot\_prod; // Final diffuse intensity. |
| --- |

### Specular Intensity Term

There is a GLSL function reflect to calculate the light reflected around the surface normal.

We can then compare the angle between the viewer and surface with this reflected vector:

* If these are the **same** then the specular colour should be **full** (= 1).
* If these are **perpendicular** then there should be **no specular light** (= 0).

Again, we use the dot product.

| vec3 reflection\_eye = reflect(-direction\_to\_light\_eye, normal\_eye); vec3 surface\_to\_viewer\_eye = normalize(-position\_eye);  float dot\_prod\_specular = dot(reflection\_eye, surface\_to\_viewer\_eye); dot\_prod\_specular = max(dot\_prod\_specular, 0.0); float specular\_factor = pow(dot\_prod\_specular, specular\_exponent);  vec3 Is = Ls \* Ks \* specular\_factor; // Final specular intensity. |
| --- |

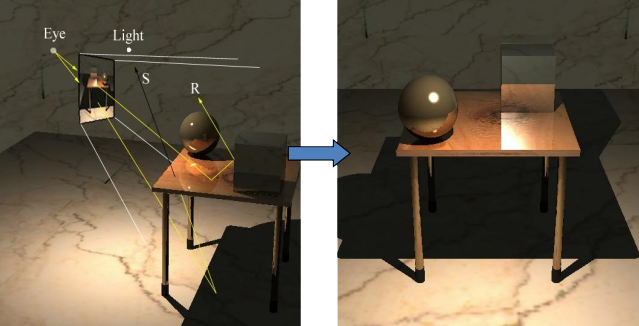
# 07 Ray Tracing

Features of ray tracing:

1. **Sharp** shadows.
2. **Perfectly-specular** reflections.
3. **Phong** illumination.



Ray tracing is a **view-dependent solution**.



Ray tracing **simplifies light transport**.

It simulates **specular to specular** only, without the spread.

**Local illumination rays** are spread **empirically**, but **eye rays** are **not**.

1. Reflections on an object (from other objects in the scene) appear as if the object were a **perfect mirror**.
2. Rays traced **through** objects appear as if the object were a **perfect transmitter**.

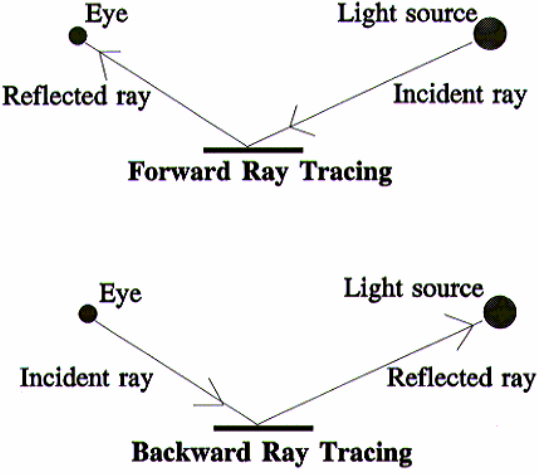
This is typical of “quick fixes” in computer graphics to achieve acceptable-looking results.

### Background

Ray tracing is an **expensive** algorithm.

Rays are usually considered to be **infinitely thin**.

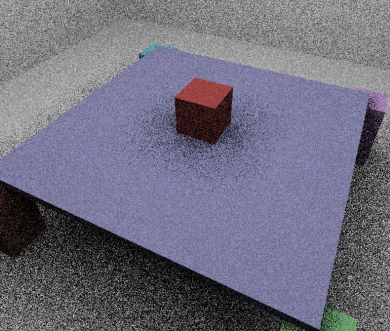
Reflection and refraction occur **without spreading** ⇒ **perfectly smooth** surfaces.



### Forward Ray Tracing

Only a **fraction of rays** reach the image

Many rays are required to get a value for each pixel.

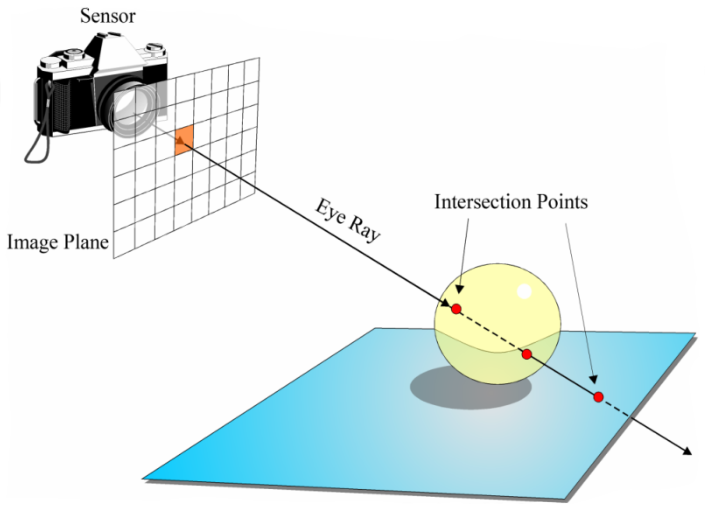


### Backward Ray Tracing

For **each pixel** in the viewport, we trace a ray from the eye (the **eye ray**) into the scene.

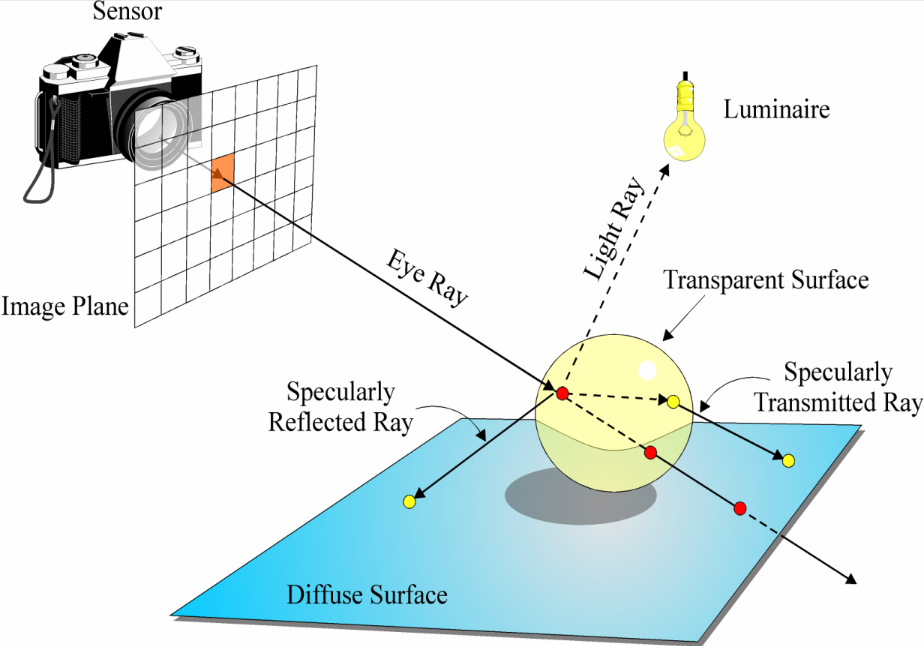
We trace it through the **pixel** and determine the **first object hit** by the ray.

This is known as **ray casting**.



We then shade this using an **extended** form of the **Phong model**.

### Ray Tracing



The eye ray will typically intersect a **number** of objects, some **more than once**.

We **sort the intersections** to find the **closest** one.

The **Phong illumination model** is then calculated, however:

1. We trace a **reflected ray**, if the surface is **specular**.
2. We trace a **refracted ray**, if the surface is **transparent**.
3. We trace **shadow rays** towards the **light sources** to determine which sources are visible to the point being shaded.

The reflected / refracted rays will hit surfaces, so we will **recursively** evaluate the illumination at these points.

⇒ A **large number of rays** must be traced to illuminate a **single pixel**.

### Whitted Illumination Model

where:

* is the **local** contribution.
* is the **global** contribution.

Ray tracing is a **hybrid** local / global illumination algorithm.

* We only consider **global** lighting effects from **ideal specular directions**.
* Before adding each light’s Phong contribution, we determine if it is **visible** to point , thus allowing **shadows** to be determined.

### Recursive Ray Tracing

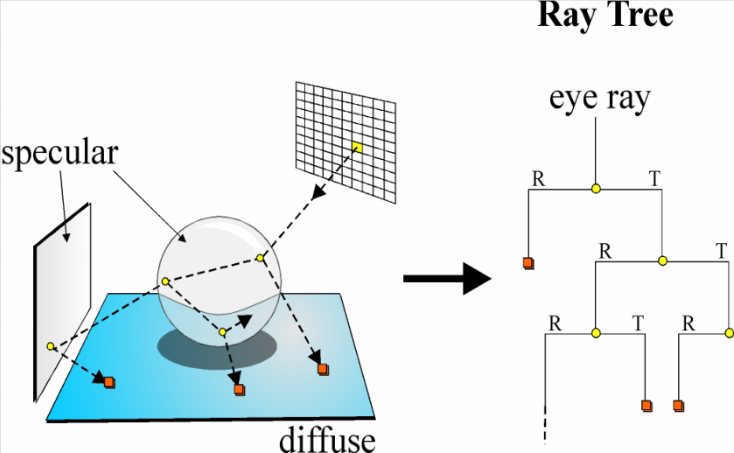
The ray tracing algorithm is **recursive**, just as the radiance equation.

At each intersection we trace a **transmitted ray**:

* If the surface is **specular** we trace a **specularly-reflected ray**.
* If the surface is **diffuse** we **terminate** the ray.

Thus we trace a ray back in time to determine its **history**, beginning with the **eye ray**.

This leads to a **binary tree**.



### Recursion Clipping

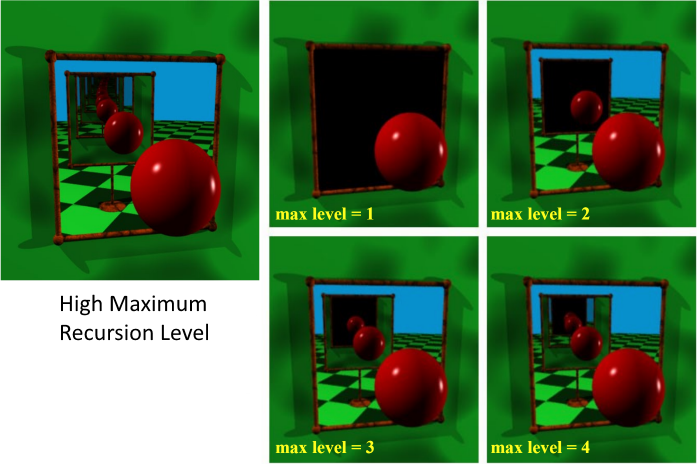
Theoretically, the recursive process *could* continue indefinitely.

In practice, at each **intersection** the ray loses some of its contribution to the pixel - its **importance** decreases:

1. If the eye ray hits a **specularly-reflecting surface** with a reflectivity of 50%, then only 50% of the energy is reflected towards the pixel.
2. If the next surface hit is of the **same material**, the reflected ray will have its contribution reduced to 25%.

We **terminate** the recursion if:

1. The current **recursive depth** is greater than a predetermined maximum depth.
2. The ray’s **contribution** to the pixel is less than a predetermined threshold .



### Rays

| A ray is mathematically the affine half-space defined by:    All points on the ray correspond to some positive value of , the parametric distance along the ray. |  |
| --- | --- |

If is normalised, then is the length along the ray of the point.

### Ray Tracing Algorithm

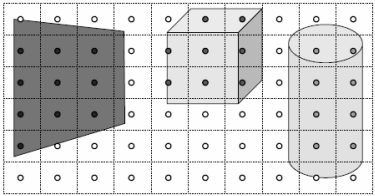
| for each pixel in viewport {  ray = determineEyeRay(pixel)  intersection = trace(ray, objects)  colour = shade(ray, intersection) }  trace(ray, objects) {  for each object in scene  intersect(ray, object) // Ray casting.  sort intersections  return closest intersection } |  |
| --- | --- |

| colour **shade**(ray, intersection) {  if no intersection  return background colour  for each light source  if (visible)  colour += Phong contribution   if (recursion level < maxlevel and surface not diffuse) {  ray = reflected ray  intersection = trace(ray, objects)  colour += \* shade(ray, intersection)  }   return colour } |
| --- |

### Ray Casting

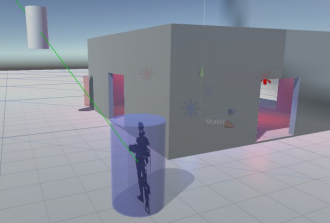
For each sample:

1. Construct a ray from the **eye position** through the **view plane**.
2. Find the **first surface** intersected by the ray through the pixel.
3. Compute the colour sample based on the **surface radiance**.



Other uses:

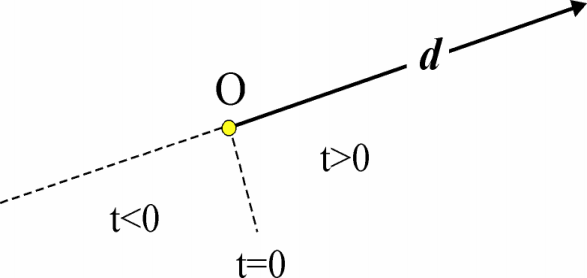
* Collision detection.
* Shooting guns.
* Line of sight.
* Occlusion culling.



### Ray Object Intersection Testing

Once we’ve constructed the eye rays, we need to determine the **intersections** of these rays and objects in the scene.

Upon intersection, we need the **normal** to the object at the point of intersection in order to perform **shading calculations**.



Objects are defined either **implicitly** or **explicitly** (parametrically).

#### Implicit Definition

if is **inside** the surface.

if is **on** the surface.

if is **outside** the surface.

The set of points on the surface are the **zeroes** of the function .

#### Explicit Definition

are the **generating parameters** and usually have **infinite ranges**.

For surfaces in 3D-space, , there will be **two** generating parameters.

We **iterate over all parameters** to generate the set of all points on the surface.

### Spheres

A sphere of center with radius is given by:

#### Implicit Form

#### Explicit Form

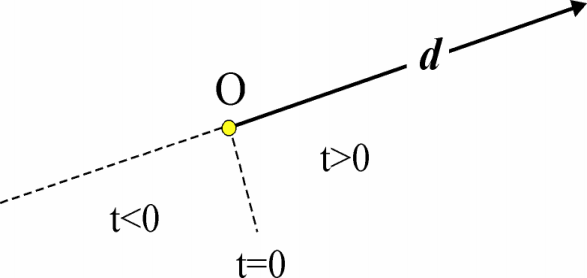
We can use either form to determine the intersection.

We will choose the **implicit** form.

## Intersection

### Ray Sphere Intersection

All points on the ray are of the form:



Any intersection points (points shared by both the ray and sphere) must satisfy **both** equations.

We **substitute** the ray equation into the sphere equation and solve for :

#### Solving the Equation

We can expand the terms as they are of the form :

We rearrange this into the form :

|  |
| --- |

We employ the **quadratic formula** to determine the two possible values of :

### Intersection Classification

Depending on the number of **real roots**, we have a number of outcomes which have nice geometric interpretations.

| We use the **discriminant**.  ⇒ **No** real roots - Ray **misses**.  ⇒ **One** root - Ray is **tangent**.  ⇒ **Two** real roots:   * Both positive ⇒ Ray **intersects** sphere. * One negative ⇒ Ray **originates** in sphere. |  |
| --- | --- |

### Ray Sphere Intersection Test

If we have two positive values of , we use the **smallest** (i.e. nearest to the origin of the ray).

is substituted back into the ray equation, yielding the **point of intersection**:

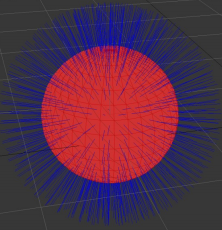
We then evaluate the **Phong model** at this point.

To do so, we need the **normal** to the surface of the sphere at the point of intersection.

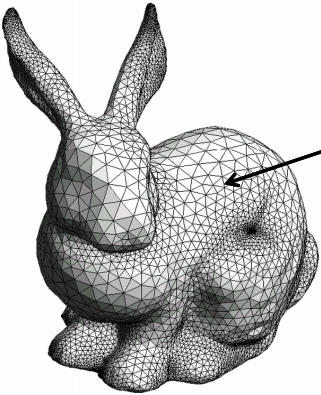
The normal and original ray direction are then used to determine the directions of the **reflected** and **refracted** rays.

#### Normal to Sphere

We can compute the normal to a sphere at a point



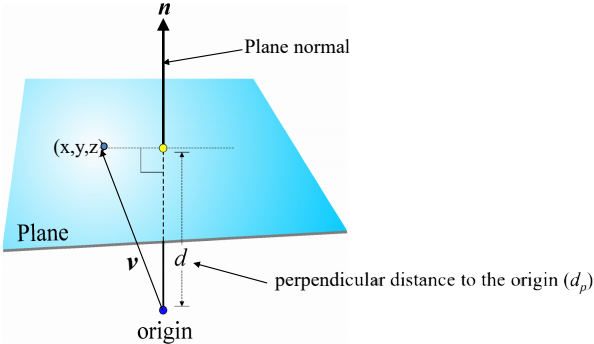
### Ray Polygon Intersection



### Ray Plane Intersection

A **plane** may be defined by:

1. Its **normal**.
2. The perpendicular **distance** of the plane to the origin.



or

The dot product between the normalised ray and the triangles normal:

If , the ray and normal are **perpendicular** ⇒ **No** intersection.

If , the ray and normal are in the **same direction** ⇒ **Back face** intersection.

If , the plane is facing direction of the ray ⇒ intersection.

There can only be **one intersection**.

The normal to the plane at each point is the same i.e. the plane normal .

To intersect a ray with a plane, we use the same approach as with the sphere.

where:

* is the **origin** of the ray.
* is the **direction** of the ray.

We solve for which gives us the point on the plane:

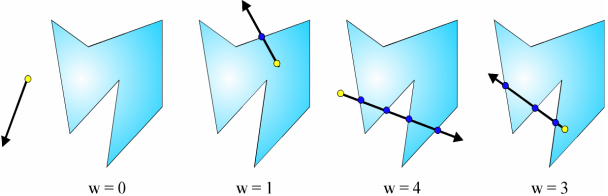
is substituted back into the ray equation, yielding the **point of intersection**:

### Ray Polygon Intersections

We first intersect the ray with the polygon plane.

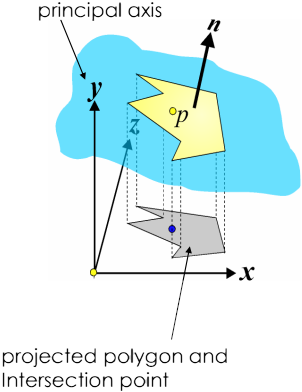
Given this intersection point (assuming the ray is not parallel to the plane), we determine if it is within the polygon interior using the **Jordan Curve Theorem**:

1. Construct any ray with the **intersection point** as an origin.
2. Count the number of polygon **edges** the ray crosses (the **winding number** ).
3. If is **odd**, then the point is in the **interior**.



Note that this is essentially a two-dimensional problem:

1. After intersecting the ray and the plane, reduce the problem to **2D** by projecting parallel to the polygon’s **principal axes** onto the plane formed by the other two axes.
2. The principal axis is given by the **largest ordinate** of the polygon normal.



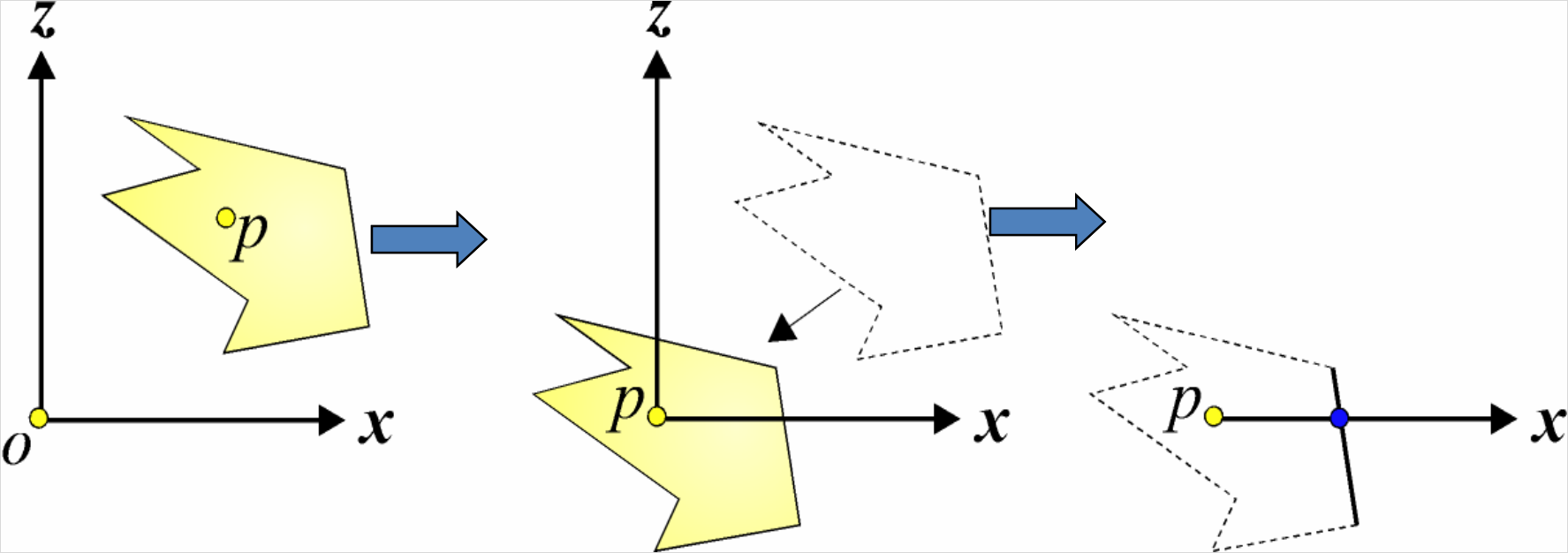
This projection **preserves the topology**, assuming the projection is **not parallel** to the polygon.

Note that this projection is **constant** for each polygon.

Therefore, we can **precompute** the polygon projection and principal axis.

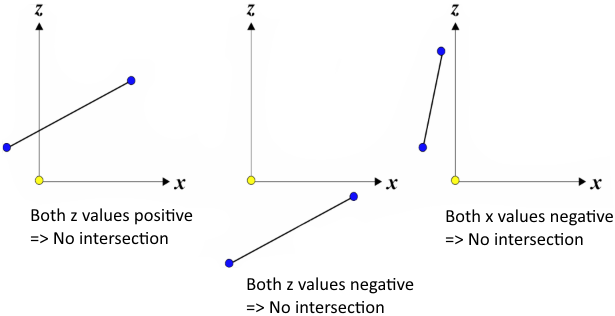
Once projected, the **winding number** is determined.

1. Translate the **intersection point** to the **origin**.  
   Apply the same translation to all projected polygon vertices.
2. Use an axis (not the principal axis) as the direction to try.
3. Determine the number of **edge crossings** along the **positive axis**.



We can **speed up** the process of determining edge intersection with an axis.

There are **three redundant cases** of edge axis topology:



If none of these cases are satisfied, then we must compute the intersection with the axis.

We determine if it lies on the **positive** half of the axis.

| Using the general equation of a line , where is the intersection with the -axis:  If , then the edge crosses the -axis and the winding number is **incremented**. |  |
| --- | --- |

#### Summary

Steps:

1. Assume **planar** polygons.
2. Check if the ray is **parallel** or **behind** the plane.
3. If not, find the intersection with the plane.  
   Substitute the ray equation into the plane equation.
4. Now that you have the **intersection point**, we find out if it is **contained in the polygon**.

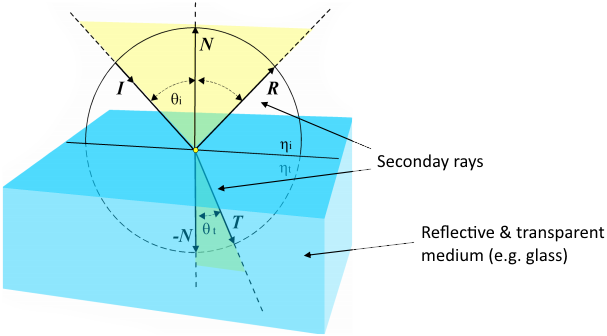
## Secondary Rays

### Whitted Illumination Model

Having determined the **closest intersection** along a ray path, we then evaluate the **Whitted illumination model** at this point.

This requires the construction of:

1. A **reflected** ray .
2. A **refracted** ray .



, , and all lie in the same plane.

| Secondary Rays The rays will have an intersection point as their origin.  The **reflected** ray direction is given by: |  |
| --- | --- |

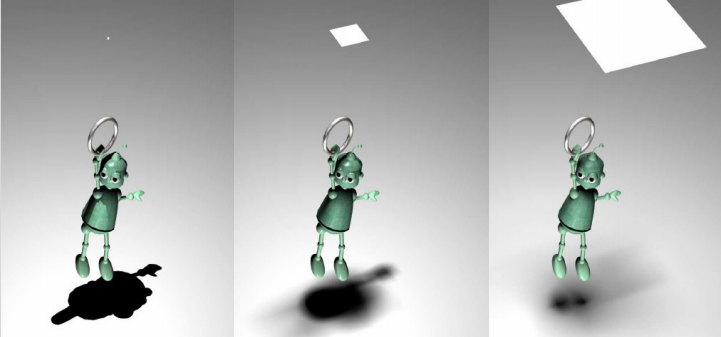
## Shadows

Whether a point is in shadow or not is determined by casting a ray from **each intersection point** to the **light source**. This is a **shadow feeler**.

If the ray **intersects any object**, then the point of interest is deemed to be **in shadow**.

This is **easier** than ray / object intersections, as we only need to know if the intersection has occurred. We don’t need to find the nearest object.

Shadow calculations impose a computational overhead in ray tracing that increases rapidly with the **number of light sources**.



### Point Light Sources

A point light source is usually not truly a point light source.

Considering a light source as a point is just a convenient model.

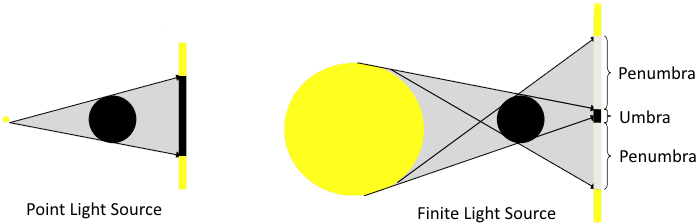
For example, a light source is **not** an infinitesimally small point. It has **volume**.

### Soft Shadows

Shadows are **not uniformly** dark.

A shadow is divided into two parts:

1. The **umbra**.
2. The **penumbra**.



**No light** reaches the umbra ⇒ It is **completely dark**.

**Some light** reaches the penumbra ⇒ It is **partially dark**.

If we model the light source as a **point** light source, there will be **no penumbra**.

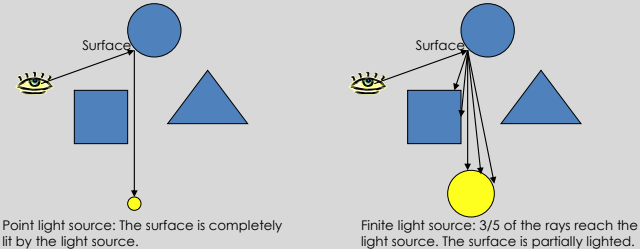
### Soft Shadows in Ray Tracing

For more **realistic** shadows, we model light sources as **volumes** rather than points.

We use a **sphere** to model a light source as many light sources are spherical (e.g. light bulbs).

When calculating lighting, we shoot **several rays** to the light source instead of just one.

We calculate **each ray**, and then take the **average**.



## Aliasing

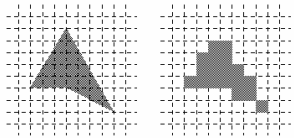
Ray tracing gives a colour for every possible point in the image.

However, a square pixel contains an **infinite number of points**.

These points may not all have the same colour.

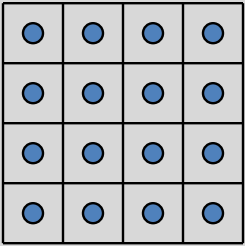
**Sampling**: Choose the colour of **one point** (i.e. center of the pixel).

Regular sampling leads to **aliasing**.



### Anti-Aliasing: Supersampling

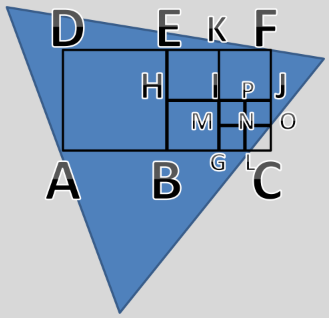
We can use **more than one ray** for each pixel and **average** the results, perhaps using a filter.



#### Adaptive Supersampling

This can be done adaptively:

1. Divide the pixel into a grid.
2. Trace 5 rays: 4 at corners + 1 at center.
3. If the colours are **similar** than use their **average**.  
   Otherwise **recursively subdivide** each cell of the grid.
4. Keep going until each grid is **close to uniform** or a **limit** is reached.
5. Filter the result.



What colour is the second pixel ?

|  |
| --- |

| **Advantages** | **Disadvantages** |
| --- | --- |
| * Areas with fairly-**consistent** appearance are **sparsely** sampled. * Areas with lots of **variability** are **heavily** sampled. | * Even with massive supersampling, visible aliasing is possible when the sampling grid interacts with **regular structures**. |

Objects tend to be almost **aligned** with the sampling grid.

Noticeable beating, more patterns are possible.

This can be remedied with **stochastic sampling**:

1. Instead of using a **regular** grid, subsample **randomly**.
2. Then adaptively subsample.

### Random Sampling

Splitting a pixel into a regular sub-pixel grid may still have aliasing if the **underlying pattern** matches the sub-pixel division.

| **Alternative 1**:  We can **randomly sample** from the pixel.  However, randomly picked positions may be very **uneven**. |  |
| --- | --- |

| **Alternative 2**:  We can pick **random positions** within a regular sub-pixel grid. |  |
| --- | --- |

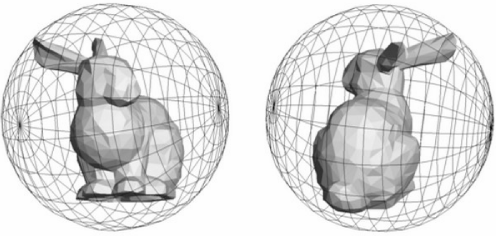
## Speeding Up Ray Tracing

Testing **each object** in the scene for intersection with **each ray** is too time-consuming.

Approximately 75-95% of runtime is spent in intersection routes.

We can try to keep the objects in some **data structure** so that we can quickly determine if a **set of objects** will definitely not intersect a ray.

### Bounding Volumes



An object that is relatively expensive to test for intersections may be **enclosed in a bounding volume** whose intersection test is less expensive (e.g. sphere, ellipsoid).

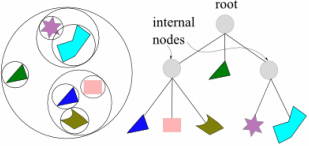
The object can be **excluded** from testing if the ray fails to hit the bounding volume.

#### Bounding Hierarchies

Bounding volumes by themselves do not determine the order or frequency of intersection tests.

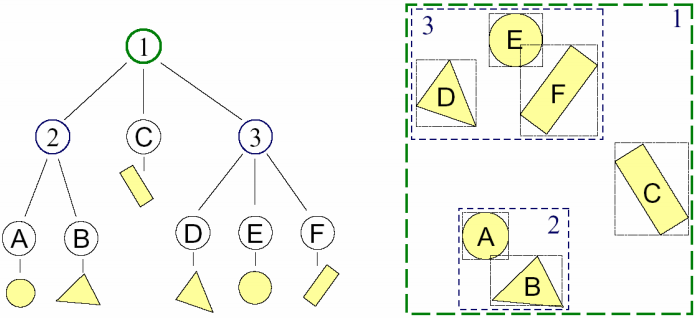
They may be ordered in **nested hierarchies** with:

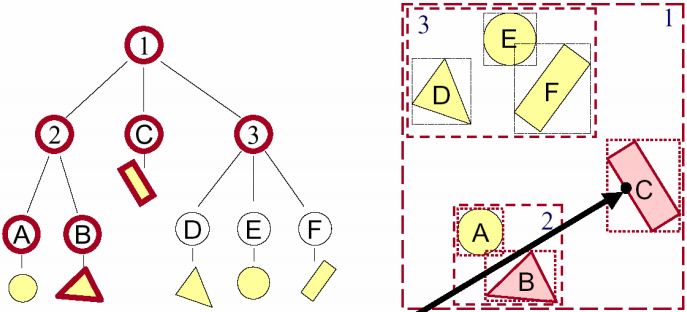
1. Objects at the leaves.
2. Internal nodes that bound their children.



A child volume is **guaranteed** not to intersect with a ray if its parent does not.

Thus if intersection tests begin with the **root**, many branches of the hierarchy can be trivially rejected.





### Octrees

| We can represent the 3D volume with a **cube**.  We subdivide the cube by half in each direction to get **8 sub-cubes**.  We can **repeatedly subdivide** each sub-cube the same way.    An **octree** is a data structure that describes how the objects a scene are **distributed** throughout the 3D space occupied by the scene.  Objects that are **close to each other** in space are represented by nodes that are close to each other in an octree. |  |
| --- | --- |

When **tracing a ray**, instead of doing intersection calculations between the ray and every object in the scene, we can now trace the ray from sub-region to sub-region in the subdivision of occupied space.

For each sub-region that the ray passes through, there will only be a **small number of objects** with which it could intersect. This lowers the number of intersection tests.

#### Constructing an Octree

| **Step 1**:  Place a **bounding cube** around **all objects**.  This is the **root** of the tree. |  |
| --- | --- |
| **Step 2**:  **Subdivide** each node into 8 equal cubes.  These will be the **children** of the node.  Place each object in their respective node. |  |
| **Step 3**:  For each node, **subdivide recursively** until:   1. The **maximum depth** has been reached. 2. There is **no object** left in the node. |  |

#### Traversing an Octree

Suppose we have a ray and want to know which object it intersects.

We **don’t** have to test all the objects in the scene.

We only need to test the objects in the nodes that are **intersected** by the ray.

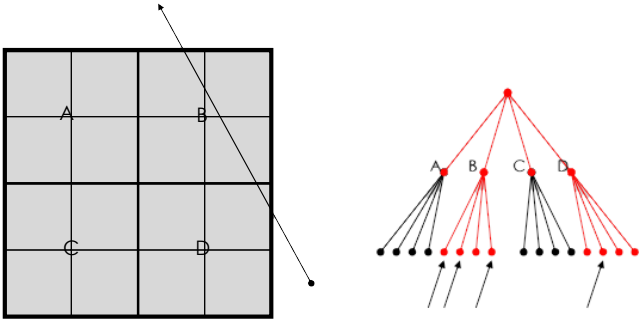
| **Example**:   1. We test intersection with the **root** node. 2. We intersect with the root. Therefore we check **all children** of the root. 3. We find that the ray only intersects node . Therefore we only traverse the octree down node . 4. We find the ray only intersects nodes . Therefore we only traverse these nodes. |  |
| --- | --- |

1. Nodes are leaf nodes.  
   Therefore we just test ray-object intersections for **all objects** contained in these leaf nodes.

#### Speeding Up Ray Tracing

The following example shows a **quad-tree**, a 2D version of an octree.

The **camera** and **ray** are shown.



Only the **red nodes** need to be tested for intersections.

All the objects in the **marked leaf nodes** need to be tested for intersections.

There is a tradeoff to be made between:

1. We **limit the depth** of any branch.  
   This **increases** the number of intersection candidates for the small volume.
2. Subdivide until a voxel contains a **single object**:  
   The **maximum length** of any branch may become long.

It is possible to have large volumes that contain only a **single** small object.

Many rays may enter this region that **do not intersect** the object.

This is similar to having a bounding volume that contains a **large void area**, because the bounding shape is less than optimal for the object it encloses.

# 08 Animation

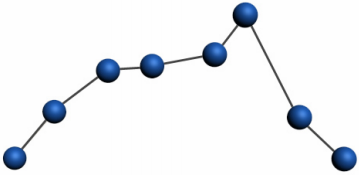
Character animations can be created with:

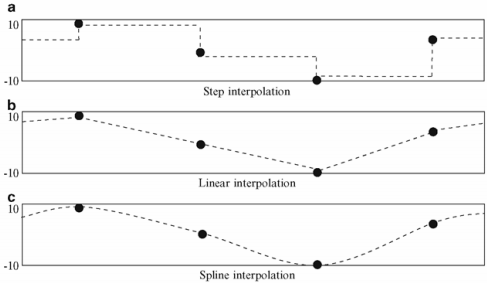
1. Motion capture (Mocap)
2. Kinematics

## Interpolation

The animator has a **list of values** associated with a given parameter of **key frames**.

We use **interpolation** to generate the values of the parameters for frames **between** keyframes.





### Linear Interpolation

| We connect **straight lines** between data points.  Given , we define a segment as: |  |
| --- | --- |

| Spline Curves We make a **smooth curve** from data points.  We build an order- polynomial from points. |  |
| --- | --- |

| Interpenetration Interpolation is not foolproof.  It may not follow the laws of physics.  Splines may **undershoot** and cause **interpenetration**.  The animator must watch for these side effects. |  |
| --- | --- |

### Representing Curves

There are different methods of representing general curves:

1. Cubic splines
2. Bezier curves
3. B-splines

## Character Animation

| Animating a character model described as a polygon mesh by moving **each vertex** is **impractical**.  Instead, we specify the motion of characters through the movement of an internal articulated **skeleton**.  Movement of the surrounding polygon mesh may be then deducted.  The mesh must deform in a manner than the viewer would expect, consistent with underlying **muscle** and **tissue**. |  |
| --- | --- |

| Human Skeleton Spine:   * It is impractical to model each vertebra. * We typically use 3-4 **spine links**.   Shoulders:   * Can **translate** as well as **rotate**. * Offer a **wide** range of motion. * Are prone to **dislocation**. |  |  |
| --- | --- | --- |

| Rigid Body Limitations  When human joints bend, the body shape bends as well. There are **no distinct parts**.  We cannot represent human joints with **rigid bodies**, else the pieces would **separate**. |  |
| --- | --- |

### Skinning

**Skinning** is the process of attaching a **renderable skin** to an underlying **articulated skeleton**.

**Binding** refers to:

1. The initial attachment of **skin** to the underlying skeleton.
2. Assigning any necessary information to the **vertices**.

Each vertex in the mesh can be attached to **more than one** joint.

Each attachment affects the vertex with a different **strength** or weight.

| Linear Blend Skinning Features:   * Skeletal subspace deformation * Enveloping * Vertex blending |  |
| --- | --- |

Linear blend skinning determines the **new position** of a vertex by **linearly combining** the results of the vertex transformed rigidly with **each bone**.

A **scalar weight** is given to each influencing bone.

The **weighted sum** gives the vertex position in the **new** pose.

Weights are set such that the **sum of all weights** for a vertex is 1.

### Motion Capture

Motion capture is the process of translating a **live performance** to a **digital performance**.

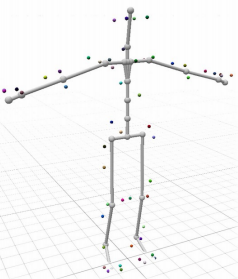
Advantages:

* Realistic **human motion**.
* More **rapid results** can be obtained.
* The amount of work required does not vary with the **complexity** or **length** of the performance.
* **Complex movement** and **realistic physical interactions** can be easily recreated.
* One actor can play **multiple roles**.

| Optical Motion Capture **Reflective markers** are placed on the actor, who is shot by **multiple cameras**.  Each camera has a **light source**.  Light is reflected by markers back towards the camera.  The 3D location of the markers is computed by **stereo vision**. |  |
| --- | --- |

#### Computing Joint Angles

The joint angles are computed based on the marker positions.



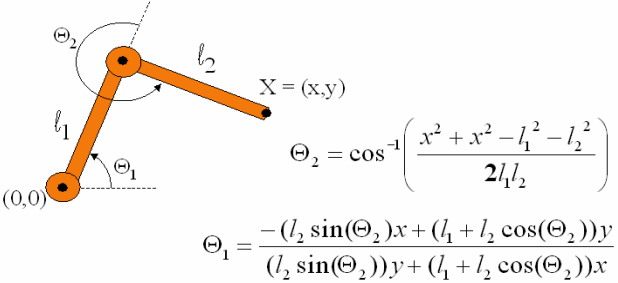
### Kinematics

| Kinematics describes the **positions of body parts** as a function of the **joint angles**.   | **Forward kinematics** | | --- | | The is a **low level** approach where the animator has to explicitly specify all motions of every part of the animated structure.  Each node in the hierarchy inherits the movement of all ancestor nodes. | | **Inverse kinematics** | | This system requires only the position of the **ends** of the structure.  It functions as a **black box**.  Controls are detailed by the movement of the entire structure. | |  |
| --- | --- | --- | --- | --- | --- |

#### Inverse Kinematics

The animator specifies the **end-effector positions**.

The computer computes the **joint angles**.



The system keeps the end of limbs **fixed** while the body moves.

We position the end of limbs by **direct manipulation**.



Inverse kinematics is difficult:

* There is not always a **unique** solution.
* The system is not always **well behaved**.
* It is a **nonlinear** problem.
* There are **joint limits**.

### Physically-Based Animation

**Forces** are used to maintain relationships between **geometric elements**.

Modelling physics usually incurs a **high computation** expense, but it is **flexible**.

For example, when simulating cloth, an animator can set the parameters that indicate the type and thickness of the cloth material. Wrinkles occur naturally, rather than specifying exact positions of wrinkles.

|  |  |
| --- | --- |

## Behavioural Animation

Behavioural animation is a type of **procedural animation**.

An **autonomous** character determines their own actions to some extent.

This gives the character some ability to **improvise** (e.g. obstacle avoidance, goal seeking).

### Boids

| The image shows a simulated boid flock avoiding cylindrical obstacles.  Each boid has direct access to the whole scene’s geometric description.  Flocking requires that it reacts **only to flockmates** within a certain small neighbourhood around itself. |  |
| --- | --- |

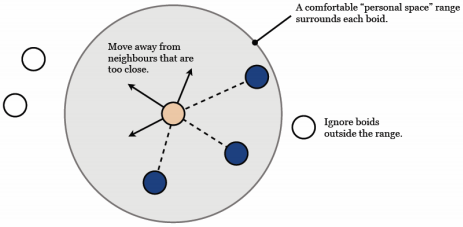
The neighbourhood is characterised by:

1. A **distance** measured from the center of the boid.
2. An **angle** measured from the boid’s direction of flight.

Flockmates outside this local neighbourhood are ignored.

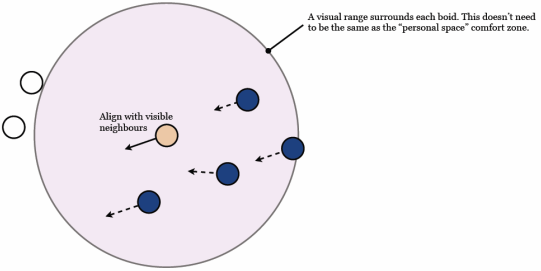
#### Rule 1: Separation

An easy rule to avoid collisions with a flock of other boids is by **keeping distance**.



#### Rule 2: Alignment

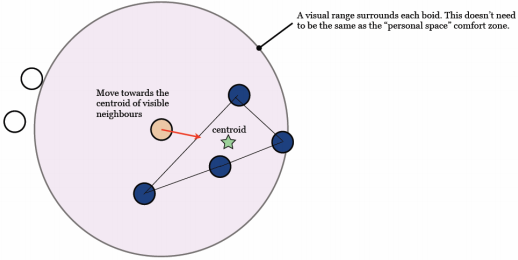
Another way to avoid collisions with a flock of other boids is by flying in the **same direction** as visible neighbours.



#### Rule 3: Cohesion

A flock is a group flying together.

We don’t want birds to fly too far apart, so we add a localised **flock centering** tendency.



# 09 Texture Mapping

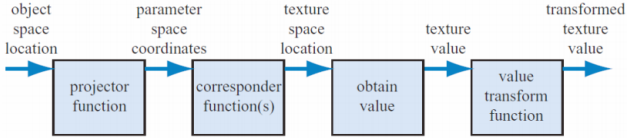
Texture mapping allows us to give a **polygon** the appearance of something more complex.

Instead of calculating colour, shade, light for each pixel, we paste **images** onto our objects in order to create the illusion of realism.



Texturing works by modifying the values used in the **shading equation**.

The pixels in the image textureare often called **texels**.



### Artist Intervention

Artists often manually decompose the model into near planar pieces.

Tools help minimise distortion by **unwrapping the mesh**.



We want each polygon to be given a fair share of the texture’s area, while also maintaining as much **mesh connectivity** as possible.

### Corresponder Functions

Corresponder functions convert **parameter-space values** to **texture-space locations**.

1. Select a subset of the image for texturing.
2. Select what happens at **boundaries**.

In OpenGL we set a **wrapping mode**:



### Magnification & Minification

| We use magnification or minification when the projected texture **does not match** the screen size.  If the viewer is **close**, the object gets larger.  ⇒ **Magnify** the texture.  If the viewer is **far**, the object gets smaller.  ⇒ **Minify** the texture.  **Efficiency** is an issue with minification. |  |
| --- | --- |

#### Magnification

| **Nearest-point sampling** | **Bilinear interpolation** |
| --- | --- |
|  |  |
|  |  |

#### Minification

The following screenshot shows **minification aliasing**.

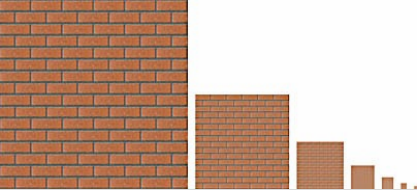
Trees are a higher resolution than the onscreen pixels.



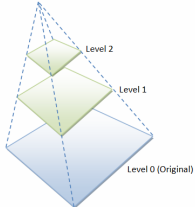
There is a visible flicker when the camera moves.

### MIP Maps

MIP maps are precalculated **optimised collections** of images based on the original texture.



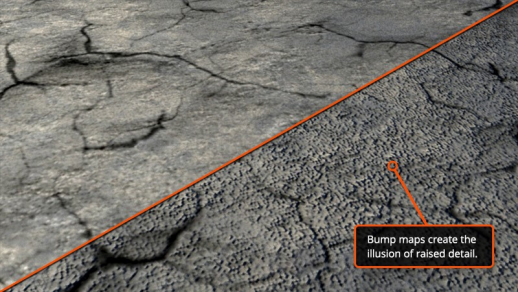
There are **dynamically chosen** based on the **depth** of the object.



MIP maps are supported by today’s hardware and APIs.

### Bump Maps

Real surfaces are not flat, but often **rough** and **bumpy**.



**Bump maps** give the illusion of depth on the surface of a model using **lighting**.

No additional resolution is added to the model.

Typically, bump maps are 8-bit greyscale images.

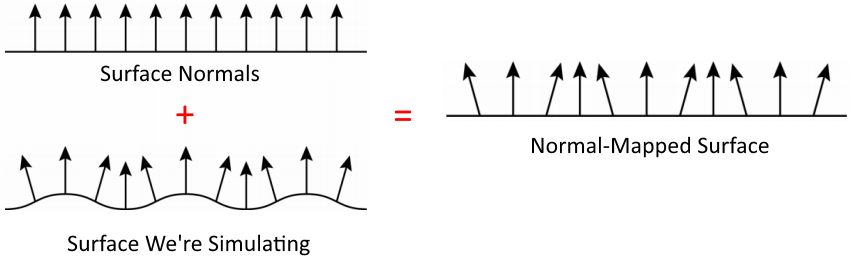
These values are used to tell perturb the **surface normals**.

### Normal Maps

Normal maps are newer and better versions of bump maps.

They are also used to fake detail in a texture.

The bumps cause slightly **different reflections** of the light.



Instead of mapping an image or noise onto an object, we can also apply a **normal map**.

This is a 2D / 3D array of vectors.

These vectors are **added to the normals** at the points for which we do shading calculations.

The effect of normal mappings is an **apparent change** of the geometry of the object.

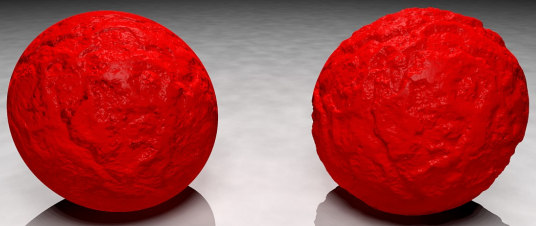
Instead of using a texture to change the colour component in the illumination equation, we access a **texture** to modify the surface normal.

It is implemented by modifying the fragment shader.

A **normal map** uses RGB information that corresponds directly with the axes.



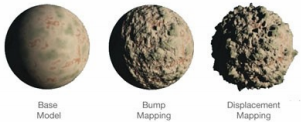
An issue with normal / bump mapping is that it does not change the **silhouette** of the object.

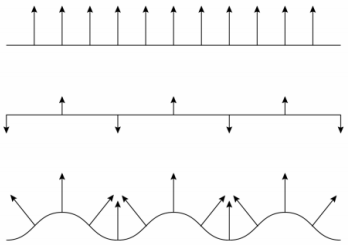


### Displacement Mapping

To overcome this shortcoming, we can use a **displacement map**.

This is also a 2D / 3D array of vectors, but here the points are **actually displaced**.





Challenges:

* Collision detection.
* Object intersection.
* Foot placement.

### Environment Mapping

When looking at a **highly-reflective** object such as a chrome sphere, you see how the object **reflects its environment**.



Ideally, each environment-mapped object in a scene should have its own environment map.

However in practice, object often **share** environment maps.

We place a **cubemap** around the object and project this onto the object.

In theory, you should **regenerate** the map when:

1. Objects in the scene move.
2. When the reflective object moves relative to the environment.

In practice, convincing reflections are possible with **static** environment maps.

Because environment mapping depends solely on **direction** and not on position, it works **poorly on flat surfaces** e.g. mirrors.

Environment mapping works best on **curved surfaces**.



Environment mapping can be combined with **subsurface scanning**.