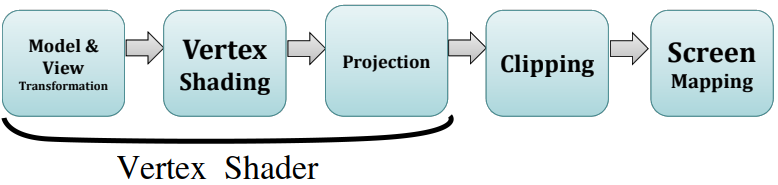
# Graphics Programming

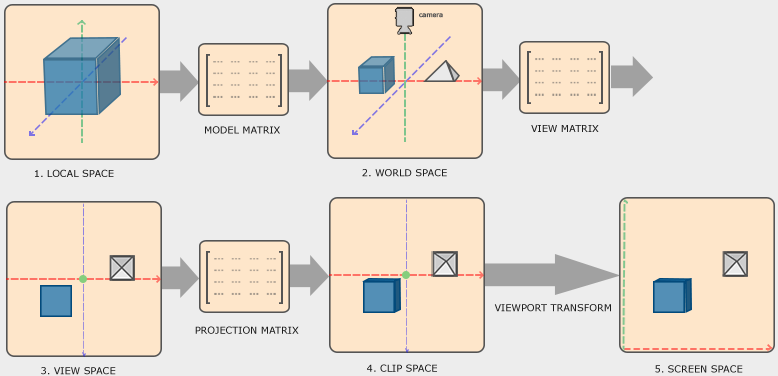
Rasterisation: Image → Pixels

Rendering: Models → Images

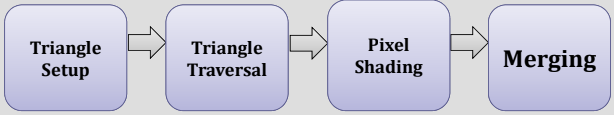
Vertex shader: Vertices → Homogeneous clip space

Geometry stage:





Rasteriser:



Double buffering:

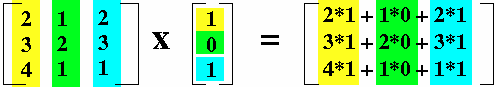
1. Back buffer
2. Front buffer

| OpenGL uses a 4-component vector to represent a vertex.  This is known as a homogeneous coordinate system. |  |
| --- | --- |

# Geometric Transformations

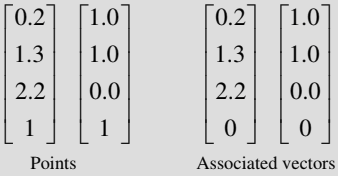
<http://www.opengl-tutorial.org/miscellaneous/math-cheatsheet/>

Matrix \* vector:

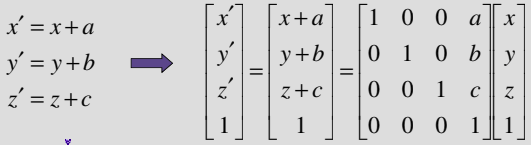


Points , vectors .

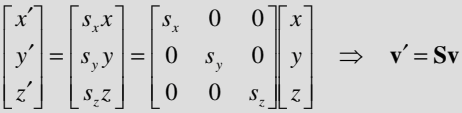
This prevents vectors from being translated.



Translation matrix \* vector:

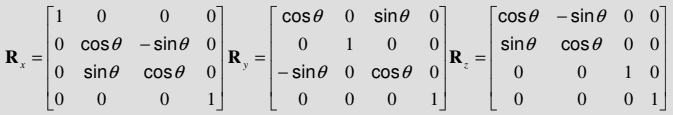


Scaling matrix \* vector:



Positive rotations are **anti-clockwise** about the origin.

3D homogeneous rotation matrices:



We multiply matrix operations in reverse order (e.g. scale, then rotate, then translate = TRS):

Euler angles:

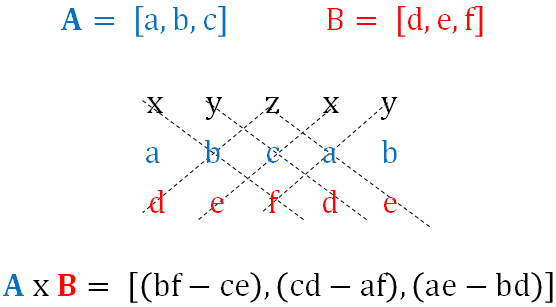
Rotation around an arbitrary axis:

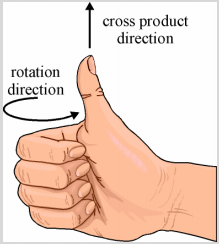
| We assume that the axis is defined by points and .  The **pivot point** is .  The **rotation vector** is .    Steps:   1. Translate the **pivot point** of the axis to the **origin**. 2. Rotate the axis and object so that the axis **lines up** with . e.g. 3. Rotate about by the required angle : 4. **Undo** the first two rotations to bring us back to the **original rotation**. 5. Translate back to the **original position**:   The final **rotation matrix** is: |
| --- |

# Linear Algebra

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

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

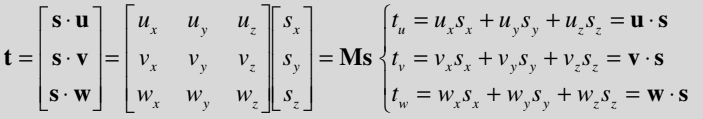




Magnitude / length / norm of vector:

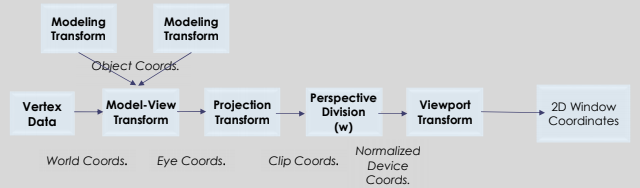
Normalised vector:

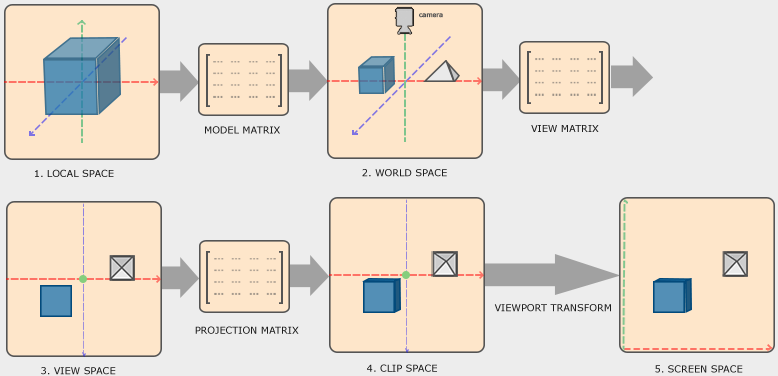
Change of basis from to :



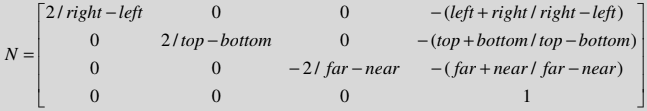
# Viewing

Transformation pipeline:

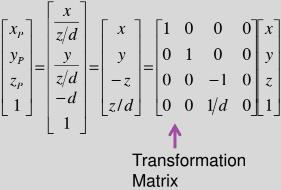




Orthographic projection:



Perspective projection:



| Normalised Device Coordinates | Window Coordinates |
| --- | --- |
|  |  |

Unmatched window and viewport aspect ratios will cause affine distortion.

Object hierarchicies:

| modelMatrix = parentMatrix \* localMatrix |
| --- |

# Lighting

Rendering algorithms:

1. Local
2. Global
3. View-dependent
4. View-independent

Global illumination algorithms (fast → slow):

1. Z-buffer
2. Ray tracing
3. Radiosity
4. Path tracing

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.

A BRDF (Bidirectional Reflectance Distribution Function) describes how light is scattered / reflected from a point on a surface.

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

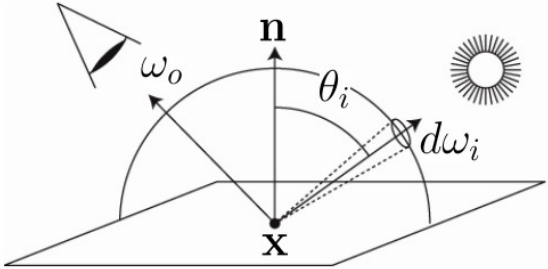
| Power / Flux(Watts) is the total energy (Joules) leaving a surface per unit time. |
| --- |
| **Radiosity**: Flux per unit area. .   |  |  | | --- | --- | |
| **Irradiance**: Flux arriving per unit area. . |
| **Radiance**: Radiosity per direction. . |

Inverse square law:

Cosine rule:

| 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. |  |
| --- | --- |

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.

Shading:

1. **Flat**: Calculated once per polygon.
2. **Smooth / Gouraud**: Calculated once per vertex and interpolated.
3. **Phong**: Normals interpolated and calculated at each fragment.

Average normal:

Phong shading vs. Gouraud shading:

* Handles specular highlights much better.
* Handles Mach bands better.
* More expensive.

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.

A point light source radiating energy equally in all directions (isotropic) has radiant intensity:

(Inverse square law where )

## Lambertian Illumination Model

A surface reflectance can be used instead of a BRDF:

The contribution from a single source is given by:

where:

* is the reflected radiance.
* is the BRDF.
* is the incident radiance.

## Phong Illumination Model

BRDF is approximated by spherical cosine function raised to a power (Phong exponent):

*(Higher ⇒ sharper spike)*

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

## Ambient Illumination Model

Less accurate irradiance model:

where:

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

## Phong Illumination Model

where:

| Reflected Vector: |  |
| --- | --- |

Blinn-Phong:

| The half vector is a vector with a direction half-way between the eye vector and light vector.  The specular term is:  as opposed to |  |
| --- | --- |

Other light source types:

1. Directional: At infinity and is represented by a direction rather than a position.
2. Spot lights: We admit illumination only within a restricted angle.

where is the spotlight attenuation.

New normal matrix:

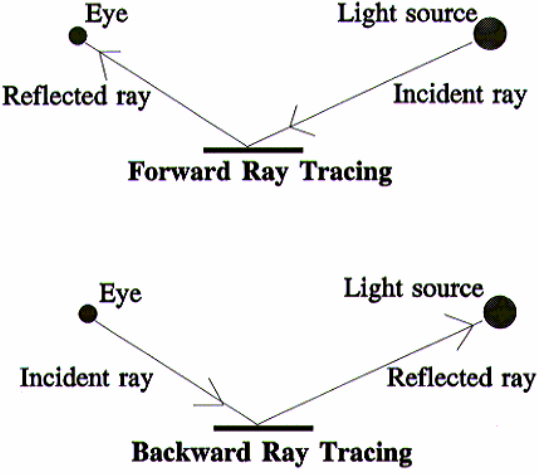
| mat4 normal\_mat = transpose(inverse(view\_mat \* model\_mat)); // or mat4 normal\_mat = inverse(transpose(view\_mat \* model\_mat)); |
| --- |

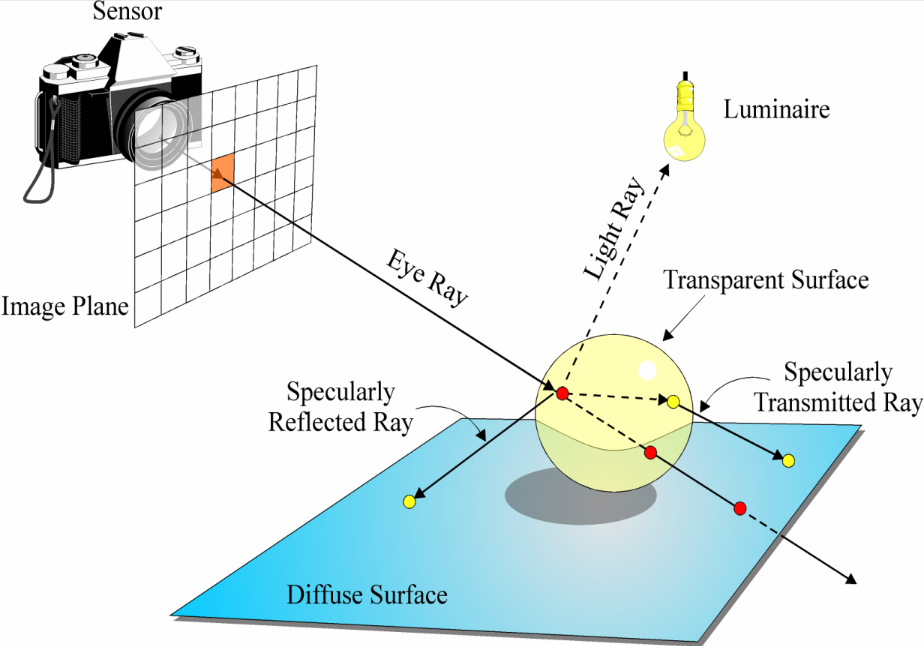
Phong fragment shader:

| // Calculate ambient term. vec3 ambient = light.ambient \* material.ambient;  // Calculate diffuse term. vec3 lightVector = normalize(light.position - position); float dotProduct = max(dot(normal, lightVector), 0); vec3 diffuse = light.diffuse \* dotProduct \* material.diffuse;  // Calculate specular term. vec3 cameraVector = normalize(cameraPosition - position); vec3 reflectionVector = reflect(-lightVector, normal); float dotProduct = max(dot(cameraVector, reflectionVector), 0); float specularComponent = pow(dotProduct, material.shininess); vec3 specular = light.specular \* specularComponent \* material.specular;  return ambient + diffuse + specular; |
| --- |

# Ray Tracing

Ray tracing simulates specular to specular only, without the spread.





**Whitted illumination model**:

where:

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

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

### 

| 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.

We terminaterecursion 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 .

**Ray tracing**:

| 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**.

**Object intersection**:

Sub ray equation into object equation and solve for .

We rearrange this into the form

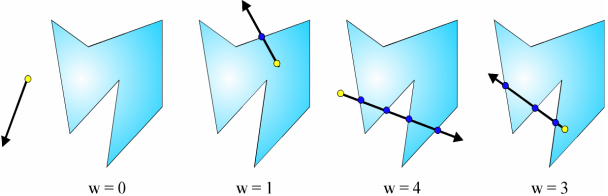
| We use the **discriminant**.  ⇒ **No** real roots - Ray **misses**.  ⇒ **One** root - Ray is **tangent**.  ⇒ **Two** real roots:   * Both positive ⇒ Ray **intersects** sphere. * One negative ⇒ Ray **originates** in sphere. |  |
| --- | --- |

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**:

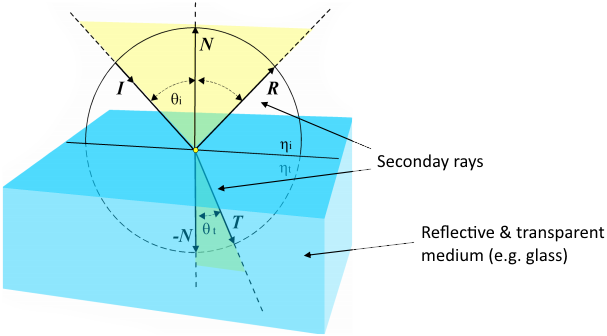
**Jordan Curve Theorem**:

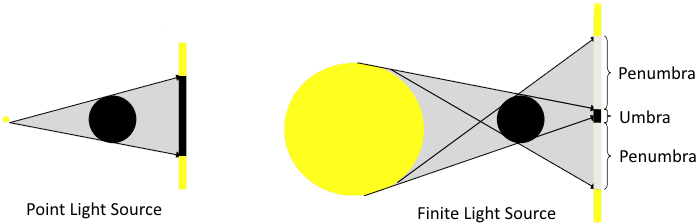
1. Construct any ray with the **intersection point** as an origin.
2. Count the number of polygon **edges** the ray crosses (the **winding number** ).
3. If is **odd**, then the point is in the **interior**.



Whitted illumination model:

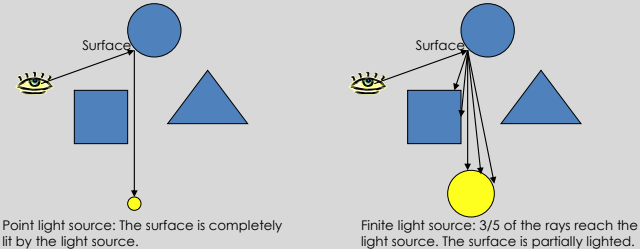
1. A **reflected** ray .
2. A **refracted** ray .





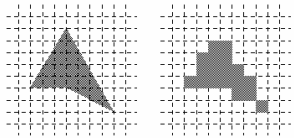
When calculating lighting, we shoot **several rays** to the volumetric light source.

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



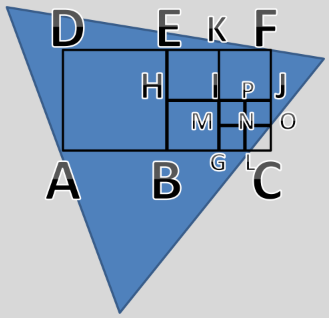
### Aliasing

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



**Adaptive** supersampling:

1. Divide 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.



**Stochastic** sampling:

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

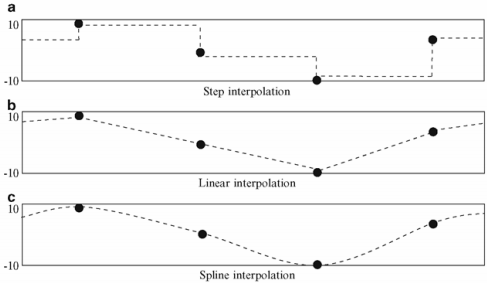
### Speeding Up Ray Tracing

An **octree** is a data structure that describes how the objects a scene are **distributed** throughout the 3D space occupied by the scene.

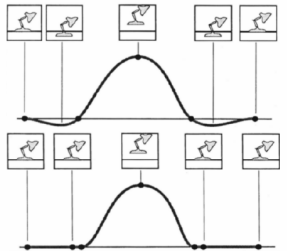
| **Step 1**:  Place a **bounding cube** around **all objects**.  This is the **root** of the tree. |  |
| --- | --- |
| **Step 2**:  **Subdivide** each node into 8 equal cubes.  These will be the **children** of the node.  Place each object in their respective node. |  |
| **Step 3**:  For each node,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

# Animation

Interpolation:



Interpenetration:



Representing general curves:

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

**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:

* A scalar weight is given to each influencing bone.
* The weighted sum gives the vertex position in the new pose.

Forward kinematics:

* Animator specifies all motions.
* Each node in hierarchy inherits movements of ancestry.

Inverse kinematics:

* Animator specifies the end-effector positions.
* Computer computes the joint angles.
* End of limbs are fixed while the body moves.
* Disadvantages:
  + Not always a unique solution.
  + Not always well behaved.
  + Nonlinear problem.
  + Joint limits.

Physically-based animation:

Forces are used to maintain relationships between geometric elements.

Boid neighbour / flock rules:

1. Separation: Keep distance.
2. Alignment: Fly in same direction.
3. Cohesion: Flock-centering tendency.

# Textures

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**.

Magnification:

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

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

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

Bump maps: Greyscale = up or down.

Normal maps: RGB = direction of each normal.