Computer Graphics - Notes

Matteo Alberici

January 2022

Contents

1 Introduction

Computer graphics deals with generating images with the aid of computers and has four main branches:

- **Rendering**: shading, raytracing, and photon mapping
- **Geometry Processing and Modelling**: description and manipulation of surfaces
- **Animation and Simulation**: physically "correct" behaviours reproduction
- **Scientific Visualization**: graphical representation of data

Three paradigms exist about computer graphics:

- **Raytracing**: represents complex scenes with photorealistic quality and "real" global effects, but it is expensive and targets offline applications
- **Rasterization**: represent efficiently real time applications with "fake" global effects
- **Image-based rendering**: creates novel views by combining images (Neural rending)

2 Raytracing Basics

2.1 Ray Casting

Ray casting determines the color of all pixels of an image: a **ray** is traced from a **camera** to a **scene** passing through an **image** full of pixels, copying the color of the intersected object on the correspondent pixel on the image.

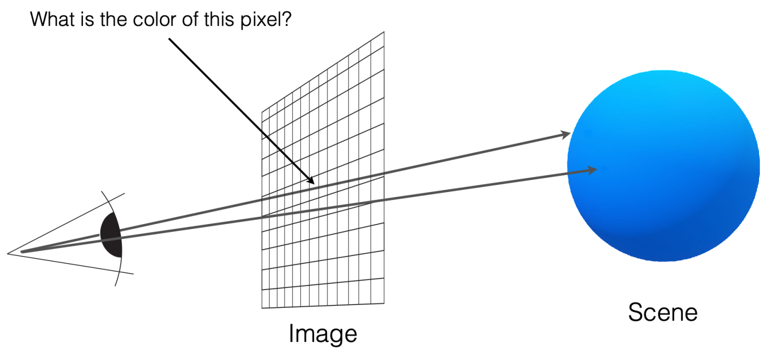

Figure 1: Ray casting representation

2.2 Whitted Ray Tracing

In **whitted ray tracing**, one **primary ray** per pixel is traced, then for each of them:

1. Find the intersection with the scene
2. Consider generating **secondary rays** recursively:
    - Shadow rays
    - Reflection rays
    - Refraction rays
3. Color the pixel with the aggregated result

A ray terminates when it leaves the scene without hitting any object, when the maximal recursion depth is reached, or finally when the contribution to the final color is negligible.

2.3 Raytracing Computation

2.3.1 Camera and Image Definitions

The camera is located at coordinates $(0, 0, 0)$ and creates an opening angle α. The image is on a plane with $z = 1$ and has a resolution of $w \cdot s \times h \cdot s$, where s represents the dimensions of a pixel and is computed as follows:

$$s = \frac{2 \cdot tg(\frac{\alpha}{2})}{w}$$

Each pixel has coordinates $p_{ij} = (x_{ij}, y_{ij}, z_{ij})$.

2.3.2 Ray Computation

Given the ray origin $o \in \mathbb{R}^3$, the distance t between o and the intersection, and the ray direction d, then a ray is defined as follows:

$$\gamma(t) = o + t \cdot d,$$

2.3.3 Per-Pixel Computation

Let's start by the top-left corner with coordinates $(X, Y, 1)$, where X and Y are computed as follows:

$$X = \frac{-w \cdot s}{2} \qquad Y = \frac{h \cdot s}{2}$$

The loop for computing the **per-pixel direction** d is the following:

```
for i = 0 to w - 1
    for j = 0 to h - 1
        dx = X + i * s + 0.5 * s
        dy = Y - j * s - 0.5 * s
        dz = 1
        d = d / ||d||
```

2.3.4 Ray-Sphere Intersection

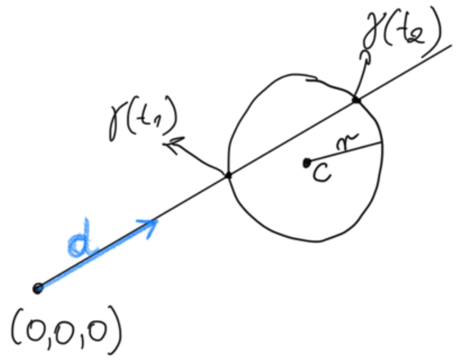

Figure 2: Intersection points on a sphere

The scene holds some objects such as a sphere with center c and ray r. The ray intersects the sphere if there exists some t such that:

$$||\gamma(t) - c|| = r$$

There exist two methods for computing the intersection points t_1 and t_2.
The first method consists in computing points t_1 and t_2 as follows:

$$t_{1,2} = \langle d, c \rangle \pm \sqrt{\langle d, c \rangle^2 - ||c||^2 + r^2}$$

Furthermore, depending on their sign, we obtain a solution:

- $t_1, t_2 < 0$: the sphere is located behind the ray
- t_1 xor $t_2 < 0$: the ray origin is inside the sphere, therefore there is only one point of intersection
- $t_1, t_2 > 0$: there exist two points of intersection

The second method consists in computing D as follows:

$$D = \sqrt{||c||^2 - \langle c, d \rangle^2}$$

Furthermore, we can differ between three cases:

- $D < r$: there exist two solutions
- $D = r$: there exists one solution
- $D > r$: there exists no solution

Finally, we can compute points t_1 and t_2 as follows:

$$t_{1,2} = \langle c, d \rangle \pm \sqrt{r^2 - D^2}$$

3 Lighting Models

There exists a set of rules for computing **color values** of objects' surfaces which models the light sources and the surface itself.

3.1 Illumination Factors

The **intensity** I of a light source depends on many factors: the object's color and reflective properties, the light's position and intensity, the viewer's position, the **normal** n of the surface point p, and the distance from p to the light.

3.2 Phong Lighting Model

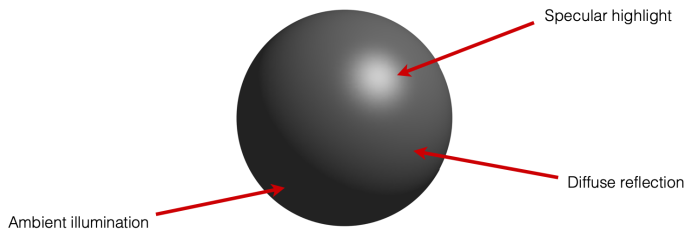

Figure 3: Phong model factors

3.2.1 Diffuse Reflection

Diffuse reflection simulates **Lambertian surfaces** on which light only depends on the light direction l and on the normal n and is reflected evenly in all directions. These surfaces have a material-dependent reflection constant $\rho_d \leq 1$ and follow the **Lambert's Cosine Law**, which states that the intensity I_d coming from a diffuse reflection is proportional to the cosine of the angle between the normal n and the direction l:

$$I_d = \rho_d \cdot \langle n, l \rangle \cdot I$$

If $\langle n, l \rangle < 0$, then the light source is behind the surface and has an intensity $I = 0$.

3.2.2 Ambient Illumination

Ambient illumination simulates indirect lightning with multiple reflections between objects and is independent of the light source and the viewpoint.
Given a scene constant I_a and a material-dependent reflection constant $\rho_a \leq 1$, the ambient term is computed as follows:

$$\rho_a \cdot I_a$$

3.2.3 Specular Reflection

Specular reflection simulates shiny surfaces on which light is reflected in exactly one reflection direction with maximum intensity.

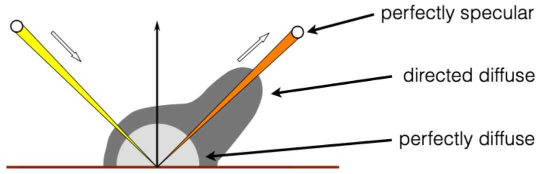

Figure 4: Specular reflection representation

The **reflection vector** r is computed as follows:

$$r = 2 \cdot n \ \cdot \langle n, l \rangle - \ l$$

Finally, given a surface specular coefficient ρ_s and a **shininess** $k \geq 1$, the specular term I_s is computed as follows:

$$I_s = \rho_s \ \cdot \langle v, r \rangle^k \cdot \ I$$

3.2.4 Phong Lighting Model Computation

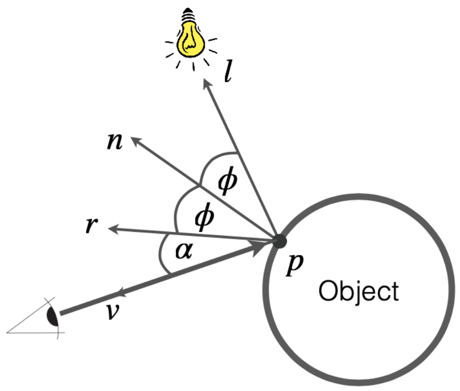

Figure 5: Phong lighting model representation

The **Phong lightning model** consists of the superposition of diffuse reflection, ambient illumination, and specular reflection.
Given n light sources and a self-emitting intensity I_e, the model is defined as follows:

$$I = I_e + \rho_a \cdot I_a + \sum_{j=1}^{n} (\rho_d \ \cdot \langle n_j, l_j \rangle + \ \rho_s \ \cdot \langle v, r_j \rangle^k) \cdot I_j$$

3.2.5 Blinn-Phong Specular Reflection

In order to simplify the computations for the Phong model, we can use the **half vector** h:

$$h = \frac{1}{2} \cdot (l + v)$$

We can compute the specular term I_s as follows:

$$I_s = \ p_s \ \cdot \langle n, h \rangle^{4k} \cdot \ I$$

3.3 Light Sources

We can distinguish between three types of light sources: point sources, directional sources, and spot sources.

3.3.1 Point Light Sources

Point light sources have an intensity I and are specified by their position, from which they radiate evenly.

3.3.2 Directional Light Sources

Directional light sources consist of an infinite set of point sources and are specified by their direction.

3.3.3 Spot Light Sources

Spot light sources generate light cones and are specified by their position p, their direction d, and their opening angle Θ_L. The maximal intensity is found along direction d, otherwise it decreases as follows:

$$I'(\Theta) = cos^k \ \Theta \cdot I$$

If $\Theta > \Theta_L$, then $I'(\Theta) = 0$.

3.4 Distance Attenuation

Light intensity **attenuation** is proportional to r^2 and there exist two ways two compute it. The first one is the following:

$$att(r) = \frac{1}{max(r, r_{min})^2}$$

The second one needs the extra parameters a_1, a_2, and a_3:

$$att(r) = \frac{1}{a_1 + a_2 \cdot r + a_3 \cdot r^2}$$

Now we can define the **extended Phong model** as follows:

$$I = I_e + \rho_a \cdot I_a + \sum_{j=1}^{n} (\rho_d \ \cdot \langle n_j, l_j \rangle + \ \rho_s \ \cdot \langle v, r_j \rangle^k) \cdot I_j \cdot att(d)$$

While computing lightning, all direction vectors must be normalized and we must handle cases in which cosines are negative.

3.5 Bidirectional Reflectance Distribution Function

Bidirectional Reflectance Distribution Function (BRDF)

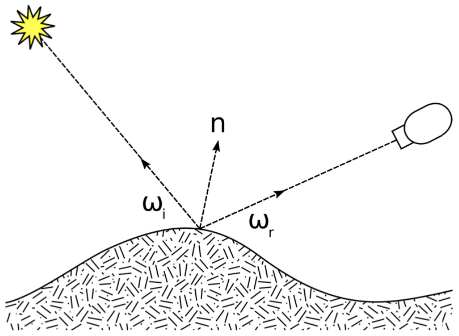

Figure 6: Ray casting representation

4 Light and Color

Light consists of charged particles emitting electromagnetic radiations (EMs). Humans can only see some EMs depending on the **wavelength** $\gamma[nm]$. The contribution of each γ is described using **Spectral Power Distribution** (**SPD**).

4.1 Color Perception

Eyes have two kinds of photoreceptor cells: **rods**, which perceive light intensity, and **cones**, which perceive colors. Cones can be distinguished in three types: S for short wavelengths, M for medium ones, and L for long ones. Given a cone sensitivity w and the incoming SPD I, the **cone stimulus** is the following:

$$\int w(\gamma) \cdot I(\gamma)\ d\gamma$$

4.2 Displays

Displays stimulate cones using several sub-pixels. Given sub-pixels' intensities R, G, and B, and SPDs $\bar{r}$, $\bar{g}$, and $\bar{b}$, the emitted light is computed as follows:

$$I(\gamma) = R \cdot \bar{r}(\gamma) + G \cdot \bar{g}(\gamma) + B \cdot \bar{b}(\gamma)$$

4.3 Gamma Correction

The relation between **display input** I_{in} and **intensity shown** I_{out} is non-linear, thus we should apply **inverse gamma correction** to the intensity I:

$$I_{in} = I^{\frac{1}{\gamma}} \qquad 1.8 \leq \gamma \leq 2.4$$

We assign more bits to dark regions to which we are more sensitive. In short:

1. Compute intensities
2. Apply inverse gamma correction for perceptual encoding
3. Display applies gamma
4. Get the desired intensities on the screen

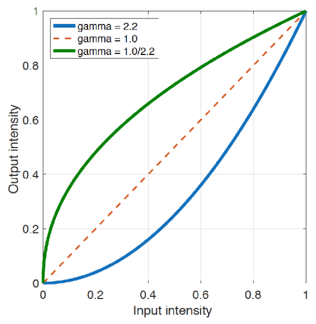

Figure 7: Ray casting representation

4.4 Tone Mapping

Displays show a range of intensities (I_{black}, I_{white}) expressed in **luminance** (cd/m^2). Since it is impossible to reproduce the real luminance range on a screen, we can use simple **tone mapping** followed by gamma correction with parameters α and β:

$$I_{in} = max((\alpha \cdot I^{\beta})^{\frac{1}{\gamma}}, 1.0)$$

Values outside the range must be clamped.

5 Meshes

Complex surfaces can be approximated using **triangle meshes**.

5.1 Anatomy of Triangle Meshes

Triangle meshes are composed of:

- **Vertices**: $V = \{v_1, v_2, ..., v_n\},\ v_i \in \mathbb{R}^3$
- **Edges**: $E = \{e_1, e_2, ..., e_l\},\ e_i = [v_{i1}, v_{i2}]$
- **Faces**: $F = \{f_1, f_2, ..., f_m\},\ f_i = [v_{i1}, v_{i2}, v_{i3}]$

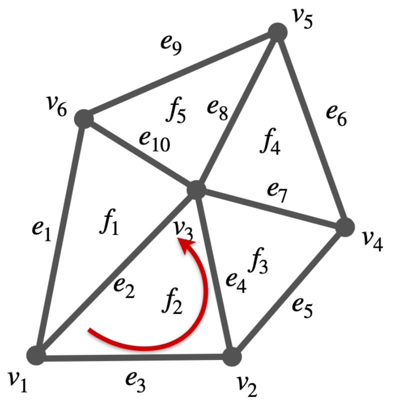

Figure 8: Triangle mesh representation

5.2 Ray-Triangle Intersection

First, we define the ray-plane intersection, where t is computed as follows:

$$t = \frac{\langle p - o, N \rangle}{\langle d, N \rangle}$$

A triangle is defined by three points p_1, p_2, and p_3. In order to compute the intersection, we must find a t such that:

$$p = \gamma(t) \text{ is coplanar with } p_1,\ p_2,\ p_3$$

If $t > 0$ and p is inside the triangle, then we compute lighting at point p.
The normal vector n at p is computed as follows:

$$n = \frac{(p_2 - p_1)\ \times\ (p_3 - p_1)}{||(p_2 - p_1)\ \times\ (p_3 - p_1)||}$$

5.3 Barycentric Coordinates

Given a triangle $[p_1, p_2, p_3]$ and a point p inside:

1. Compute the area W of the triangle
2. Compute the areas w_1, w_2, and w_3 formed by p and the edges opposite p_1, p_2, and p_3
3. Normalize the areas: $\lambda_i(p) = \frac{w_i}{W}$

The obtained values are the **barycentric coordinates** of p with respect to triangle $[p_1, p_2, p_3]$ and have two properties:

- Partition of unity: $\sum_{i=1}^{3} \lambda_i(p) \;= 1$
- Non-negativity: $\lambda_i(p) \geq 0 \;$ for $\; p \in [p_1, p_2, p_3]$

If p is not in the triangle, then the second property does not hold.

5.3.1 Computation in 2D

1. Define $p_i = (x_i, y_i)$ and $p = (x, y)$
2. Compute the area of triangle $[p1, p2, p3]$ via $2D$ determinant:

$$2W = (x_2 - x_1) \cdot (y_3 - y_1) - (x_3 - x_1) \cdot (y_2 - y_1)$$

3. Compute similarly w_1, w_2, and w_3

5.4 Computation in 3D

1. Define $p_i = (x_i, y_i, z_i)$ and $p = (x, y, z)$
2. Compute the area of triangle $[p1, p2, p3]$ via $3D$ cross-product:

$$n = (p_2 - p_1) \times (p_3 - p_1) \qquad 2W = ||n||$$

3. Compute w_1, w_2, and w_3:

$$n_i = (p_{i+1} - p) \times (p_{i-1} - p) \qquad 2w_i = ||n_i|| \cdot sign(\langle n_i, n \rangle)$$

5.5 Procedural Textures

In **procedural textures**, the color of an object depends on the coordinates of its surface point:

- Sphere: reflection coefficients as function of the normal vector coordinates
- Triangle: reflection coefficients as function of the barycentric coordinates

Let's define a function $f(u, v) : \mathbb{R}^2 \to \mathbb{R}^3$:

$$f(u, v) = (\lfloor n \cdot u \rfloor + \lfloor n \cdot v \rfloor)\%2$$

In order to texture a triangle, we can use two of its barycentric coordinates:

$$f(u, v) = f(\lambda(p_3), \lambda(p_2))$$

In order to texture a sphere, we need to transform the position $p = (x, y, z)$ to the spherical coordinates. First we compute the angles θ and ϕ:

$$\theta = arcsin(\frac{y}{r}) \in [-\frac{\pi}{2}, \frac{\pi}{2}], \qquad \phi = arctan(\frac{z}{x}) \in [-\pi, \pi]$$

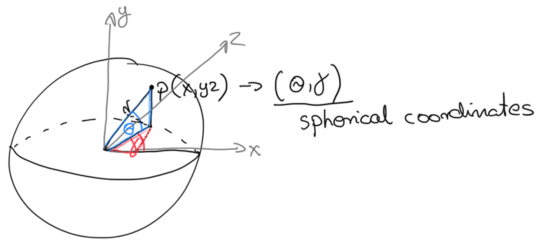

Figure 9: Sphere texturing representation

Then we can compute the spherical coordinates (u, v):

$$u = \frac{\phi + \pi}{2 \cdot \pi}, \qquad v = \frac{\theta + \pi/2}{\pi}$$

6 Transformation

Objects are described by $3D$ points: a sphere with center c and radius r is defined as follows:

$$r : S(c, r) = \{p = (x, y, z) : ||p - c|| = r\}$$

Translations are described by a translation vector t:

$$t = (t_x, t_y, t_z) \to p' = p + t$$

$$\text{Sphere } S(c, r) \ \to \ S(c + t, r)$$

Rotations are described by a transformation matrix R:

$$p' = Rp$$

$$\text{Axis } \{c + \lambda v : \lambda \in \mathbb{R}\} \ \to \ \{Rc + \lambda \cdot (Rv) : \lambda \in \mathbb{R}\}$$

6.1 Global and Local Coordinates

There exist two ways to describe motions: using **global coordinates**, meaning the camera ones, or using **local coordinates**, meaning the object ones. From camera's point of view, the object's coordinates change, while from the object's point of view, the ray origin coordinates change in the opposite direction.
Initially, every object is born at the origin and the local coordinates are identical to the global ones, then objects are moved around: if an object is rotated by R and translated by T, then from its perspective the world is translated by $-T$ and rotated by $-R$.

6.2 Rotations

In order to describe rotations, we must define an object center c and a **rotation axis**. In global coordinates, the rotation axis is defined as follows:

$$\{c + \lambda \cdot (1, 0, 0) : \lambda \in \mathbb{R}\}$$

In local coordinates, we rotate the ray and its origin in the opposite direction with the following rotation axis:

$$\{\lambda \cdot (1, 0, 0) : \lambda \in \mathbb{R}\}$$

6.3 Homogeneous Coordinates

Since translations are described by additions and rotations by multiplications, we add a coordinate and work in 4 dimensions. For any point p and direction d:

$$\begin{array}{lcl} p = (x, y, z) & \to & p = (x, y, z, 1) \\ d = (x, y, z) & \to & d = (x, y, z, 0) \end{array}$$

The new coordinates are called the **homogeneous coordinates**.
We need to introduce some arithmetic meanings:

- Position + Displacement = Position
- Position − Position = Displacement
- Displacement + Displacement = Displacement
- Position + Position = Position

7 Shadows, Reflection, and Refraction

7.1 Shadows

Shadows are dark spots for which the light is occluded and consist of an **umbra**, which is a complete shadow, and a **penumbra**, which is a partial shadow. They convey information such as objects' relative positions, depth, and lights positions.
After tracing the primary ray, we trace a **shadow ray** from the intersection point towards the light source: if the intersection is closer than the light source, then the object is in shadow.
The Phong lighting model is extended by the **shadow term** $s_j(p)$ which evaluates to 0 if the shadow ray hits an object, otherwise evaluates to 1

$$I = I_e + \rho_a \cdot I_a + \sum_{j=1}^{n} (\rho_d \cdot \langle n_j, l_j \rangle + \rho_s \cdot \langle v, r_j \rangle^k) \cdot I_j \cdot s_j(p)$$

7.2 Reflections

Reflection occurs when a ray hits a point p on a mirror surface.

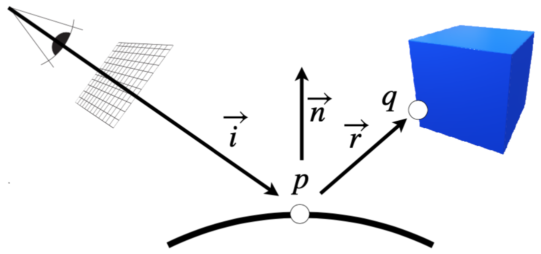

Figure 10: Reflection representation

The procedure is the following:

1. Compute the **reflection ray** r as follows:

$$r = i - 2n \cdot \langle n, i \rangle$$

2. Trace ray r from p towards the reflection direction
3. Find the intersection point q with the first object
4. Compute the color at point q
5. Reflect the color towards p

For partial reflections, we use the constant $\alpha_{reflect} \in [0, 1]$.

7.3 Refractions

Refraction occurs when light propagates through different materials. Given the speed of light in vacuum c and the speed of light in the medium v, then the index of refraction δ is computed as follows:

$$\delta = \frac{c}{v}$$

The **Snell's law** is defined using the refraction indices δ_1 and δ_2 and the velocities in the medium v_1 and v_2:

$$\delta_1 \cdot sin\ \theta_1 = \delta_2 \cdot sin\ \theta_2 \ \rightarrow\ \frac{sin\ \theta_1}{sin\ \theta_2} = \frac{\delta_2}{\delta_1}$$

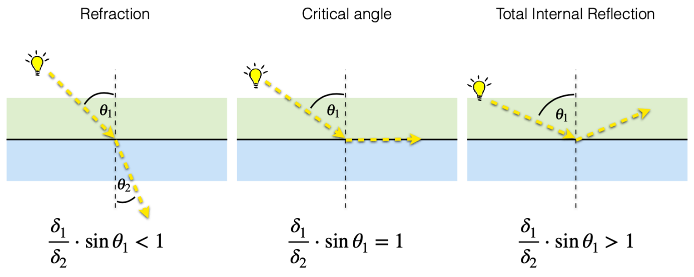
Refraction
Critical angle
Total Internal Reflection
θ_1
θ_2
θ_1
θ_1
$\frac{\delta_1}{\delta_2} \cdot \sin\theta_1 < 1$
$\frac{\delta_1}{\delta_2} \cdot \sin\theta_1 = 1$
$\frac{\delta_1}{\delta_2} \cdot \sin\theta_1 > 1$

Figure 11: Refraction constraint

As in reflection, we compute a **refraction ray** r and evaluate the color of the point it intersects. The refraction ray r is computed as follows:

$$a = n \cdot \langle n, i \rangle \qquad b = i - a \qquad \beta = \frac{\delta_1}{\delta_2} \qquad \alpha = \sqrt{1 + (1 - \beta^2) \cdot \frac{||b||^2}{||a||^2}}$$

$$r = \alpha \cdot a + \beta \cdot b$$

7.4 Recursive Raytracing Algorithm

The following is a recursive raytracing algorithm:

```
trace(origin o, direction d):
    p = findFirstIntersection(o, d)
    n = surfaceNormal(p)
    s = reflectionDirection(p, n)
    t = refractionDirection(p, n)
    I_{direct} = phongLighting(p, n, d)
    I_{reflect} = \alpha_{reflect} \cdot trace(p, s)
    I_{refract} = \alpha_{refract} \cdot trace(p, t)
    return (I_{direct} + I_{reflect} + I_{refract})
```

The algorithm terminates when the ray leaves the scene, if the maximal recursion depth is reached, or if the intensity is smaller than some threshold.

7.5 Fresnel Effect

Due to the **Fresnel effect**, the amount of light that is reflected of refracted on a surface depends on the viewing angle. The Fresnel reflection ray $F_{reflection}$ is computed as follows:

$$F_{reflection} = \frac{1}{2} \cdot \left(\left(\frac{\delta_2 \cdot cos\Theta_1 - \delta_1 \cdot cos\Theta_2}{\delta_2 \cdot cos\Theta_1 + \delta_1 \cdot cos\Theta_2} \right)^2 + \left(\frac{\delta_1 \cdot cos\Theta_1 - \delta_2 \cdot cos\Theta_2}{\delta_1 \cdot cos\Theta_1 + \delta_2 \cdot cos\Theta_2} \right)^2 \right)$$

The Fresnel refraction ray $F_{refraction}$ is computed as follows:

$$F_{refraction} = 1 - F_{reflection}$$

8 Advanced Raytracing

Raytracing is expensive since we generate at least one ray for each pixel and recursively more secondary rays for each intersection.

8.1 Efficient Raytracing

Efficient raytracing uses space partitioning to reference singularly to each grid cell and to the objects contained in it.

8.1.1 Bounding Volume Hierarchy

In **Bounding Volume Hierarchy** (**BVH**), neighbouring objects are gathered through simple bouncing primitives, starting from the biggest primitives.

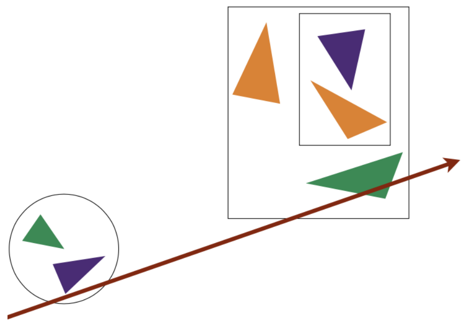

Figure 12: BVH representation

8.1.2 Binary Space Partitioning

In **Binary Space Partitioning** (**BSP**), the space is recursively divided with planes. It has a runtime of $O(logn)$, but the resulting space is hard to traverse.

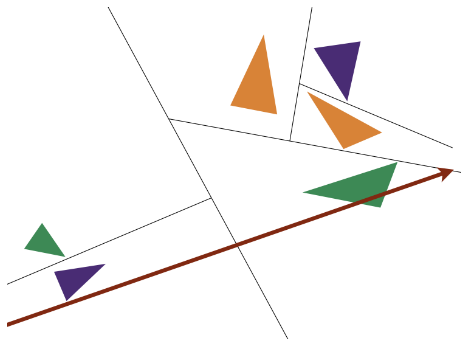

Figure 13: BSP representation

8.1.3 kd-Tree

In **kd-tree** partitioning, the space is recursively divided with axis aligned planes. It uses the following parametric ray equation:

$$\gamma(t) = 0 + t \cdot d,$$

recursively considering the active ray segment $[t_{min}, t_{max}]$. The following is a traversing algorithm that runs in $O(logn)$:

```
float recTraverse(node, t_min, t_max):
    if (node.isLeaf):
        intersectTrianglesInLeaf(node)
        return t_closestHit
    u = (node.s - o[node.a]) / d[node.a]
    if (u <= t_min):
        return recTraverse(node.b, t_min, t_max)
    else if (u >= t_max):
        return recTraverse(node.f, t_min, t_max)
    else:
        t_hit = recTraverse(node.f, t_min, u)
        if (t_hit <= u):
            return t_hit
        return recTraverse(node.b, u, t_max)

void traverse():x
    (t_min, t_max) = clip(0, max)
    recTraverse(kdRoot, t_min, t_max)
```

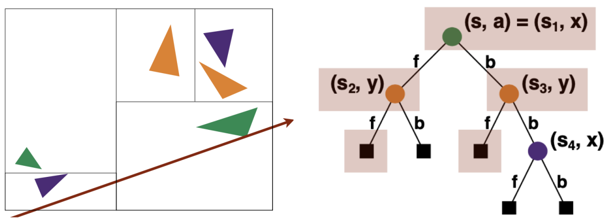

Figure 14: BSP representation

8.2 Antialiasing

With **antialiasing**, we perform super-sampling tracing $k \times k$ rays per pixel and averaging the colors.

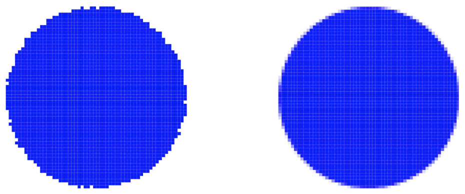

Figure 15: Antialiasing effect

There are two types of super-sampling: in the **stochastic one**, many rays are traced to handle random distribution, while in the **adaptive one**, we start with 5 rays per pixel and, if the color difference is too big, we compute 4 sub-pixels.

8.3 Thin Lens Camera Model

In **thin lens camera model**, there are some additional inputs to the raytracer: the focal distance f, the aperture size r, and the number of samples n.
The following algorithm represents the model working:

```
color = 0
focal_point = f * d / d.z
for i = 0 to n:
    offset = randDisk(r)    // random offset within aperture
    new_o = o + (offset, 0)     // z component stays the same
    new_d = normalize(focal_point - new_o)
    color += traceRay(Ray(new_o, new_d))
color = color / n
```

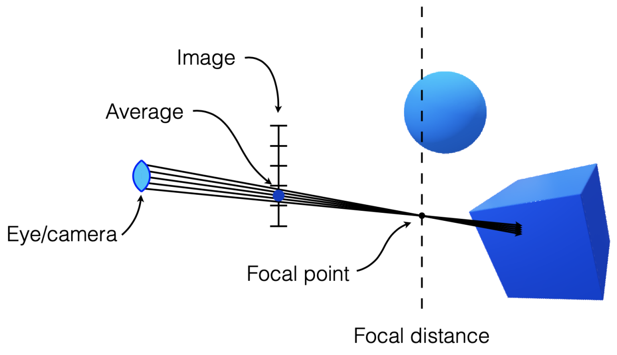

Figure 16: Thin lens model representation

9 Rasterization

Rasterization consists of coloring all visible pixels.

9.1 Drawing Lines

Since trying to draw a line from point $p_1 \in \mathbb{R}^3$ to point $p_2 \in \mathbb{R}^3$ with raytracing would result in all rays missing the line, we must follow an inverse approach:

1. Consider rays from p_1 and p_2 to the camera
2. Compute the intersections with the image
3. Obtain the screen coordinates (x_1, y_1) and (x_2, y_2) for p_1 and p_2
4. Draw the line from (x_1, y_1) to (x_2, y_2)

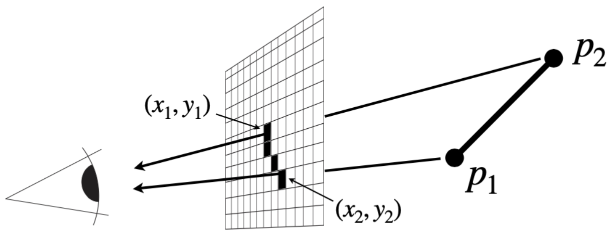

Figure 17: Drawing a line

The closest pixels to the "ideal" line must be colored.
A line in $2D$ is defined as follows:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \qquad d = y_1 - m \cdot x_1 \qquad y = m \cdot x + d$$

Without loss of generalization:

- $0 \leq m \leq 1$
- $x_1 < x_2$
- $0° \leq$ line slop $\leq 45°$

9.2 Midpoint Algorithm

While drawing a line, at each $x-$step there are two options for the $y-$coordinate: either it stays as it is or it increases by 1.

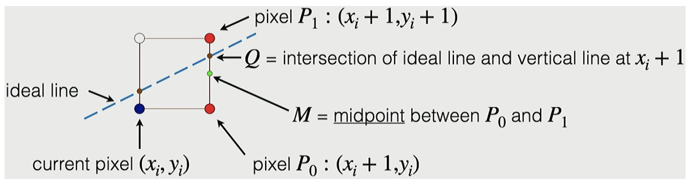
pixel $P_1 : (x_i + 1, y_i + 1)$
Q = intersection of ideal line and vertical line at $x_i + 1$
ideal line
M = midpoint between P_0 and P_1
current pixel (x_i, y_i)
pixel $P_0 : (x_i + 1, y_i)$

Figure 18: Midpoint algorithm

We must determine where the **midpoint** M lies with respect to Q, which is the intersection of the ideal line and the vertical line at $x_i + 1$: if M lies above Q, then we take pixel P_0, while if M lies below Q, then we take pixel P_1.
The procedure to determine whether a point is above or below a line is the following:

1. Compute dx and dy as follows:

$$dx = x_2 - x_1 \qquad dy = y_2 - y_1$$

2. Compute $F(x, y)$ as follows:

$$F(x, y) = y \cdot dx - x \cdot dy + x_1 \cdot dy - y_1 \cdot dx$$

3. Check the sign of $F(x, y)$:

   - if $F(x, y) = 0$, then (x, y) lies on the line
   - if $F(x, y) > 0$, then (x, y) lies above the line
   - if $F(x, y) < 0$, then (x, y) lies below the line

The midpoint decider f is computed as follows:

$$f = F(x_i + 1, y_i + 0.5)$$

If $f \geq 0$, then we choose P_0, otherwise we choose P_1.

The midpoint decider is computed incrementally:

$$F(M_0) = F(M) - 2 \cdot dy$$
$$F(M_1) = F(M) - 2 \cdot dy + 2 \cdot dx$$
$$\text{Initial value: } F(M) = -2 \cdot dy + dx$$

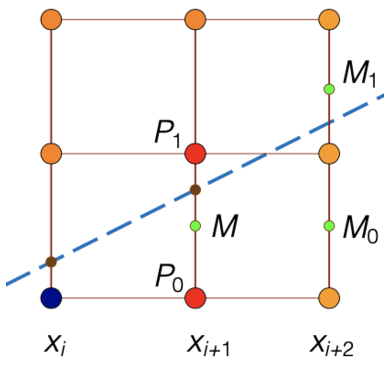

Figure 19: Midpoint decider

The implementation of the midpoint algorithm is the following:

```
x = x1
y = y1
dx = x2 - x1
dy = y2 - y1
f = -2 * dy + dx
for i = 0 to dx:
    setPixel(x,y)
    x += 1
    if (f < 0):
        y += 1
        f += 2 * dx
    f -= 2 * dy
```

Lines drawn with the algorithm appear aliased since we set only one pixel per column, thus we perform antialiasing by setting two pixels P_0 and P_1 per column. The pixels intensities are proportional to the distance to the ideal line:

$$I_0 = \frac{F(P_1)}{F(P_1) - F(P_0)}$$
$$I_1 = \frac{-F(P_0)}{F(P_1) - F(P_0)}$$

The sum of the intensities is 1.

9.3 z-Buffer

9.4 Perspective Interpolation

9.5 Rasterization of Triangles

Given a $3D$ triangle $[p_1, p_2, p_3]$ in global coordinates, we perform the following procedure:

1. For each point p_i compute the screen coordinates s_i:

$$s_i = (x_i, y_i)$$

2. Compute the reciprocal $z-$values z_i as follows:

$$z_i = \frac{1}{p_i^z}$$

3. Found a bounding box defined as follows:

$$\begin{aligned} x_{min} &= min(x_1, x_2, x_3) \\ y_{min} &= min(y_1, y_2, y_3) \\ x_{max} &= max(x_1, x_2, x_3) \\ y_{max} &= max(y_1, y_2, y_3) \end{aligned}$$

4. Check for each pixel s_i within the box if it is in the triangle as follows:
   (a) Compute the barycentric coordinates λ_1, λ_2, and λ_3 of s
   (b) Pixel s is in the triangle if all the barycentric coordinates are non-negative

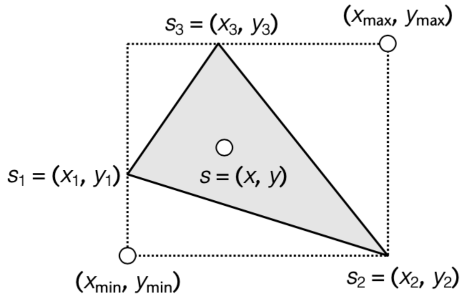

Figure 20: Rasterization of triangles

10 Graphics Pipeline and WebGL

In the **graphics rendering pipeline**, everything runs in parallel according to the following procedure:

1. **Application**: determines the composition of the scene
2. **Geometry Processing**: puts the geometry in the common space, performs clipping and screen mapping, and computes per-vertex shading
3. **Rasterization**: performs primitive setup and traversal and generates fragments with interpolated per-vertex data
4. **Pixel processing**: colors each fragment and merge them into an image

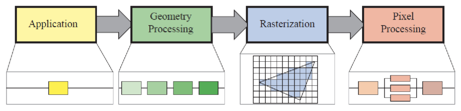

Figure 21: Graphics rendering pipeline

Let's have a closer look at the pipeline marking the fixed stages in orange, which can be configured, and in blue the programmable stages, whose task must be specified:

1. Vertex specification: setting up geometry
2. Vertex shader: performing transformations
3. Vertex post-processing: clipping and outputting geometry
4. Primitive assembly: creating geometry from vertices and face culling
5. Rasterization: creating fragments and interpolating data
6. Fragment shader: shading
7. Per-sample operations: merging and depth test

10.1 Depth Test and Face Culling

The **depth test** is necessary to find which object occludes which other object. **Face culling** prevents the rendering of the faces that are not visible to the viewer.

10.2 WebGL and GLSL

The graphics pipeline in **WebGL** performs the following procedure:

1. Create a WebGL canvas
2. Generate and send the geometry to **Graphics Processing Unit** (**GPU**)
3. Define vertex shaders (per-vertex operations)
4. Define fragment shaders (per-pixel operations)
5. Use the geometry and the shader program to draw the scene

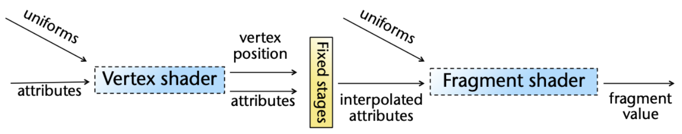

Figure 22: Pipeline data flow

10.3 Vertex Attributes

Geometric objects are stored as vertices, meaning collections of the following attributes in space: position, color, normal vector, and more.
Vertices data is stored in **Vertex Buffer Objects** (**VBOs**), which are arrays of concatenated elements of each vertex. Moreover, the **Vertex Array Objects** (**VAOs**) describe the state of attributes, which VBOs to use, and how to pull the data from it.

11 Transformation Pipeline

The **transformation pipeline** performs different types of transformations in order to convert the given coordinates.

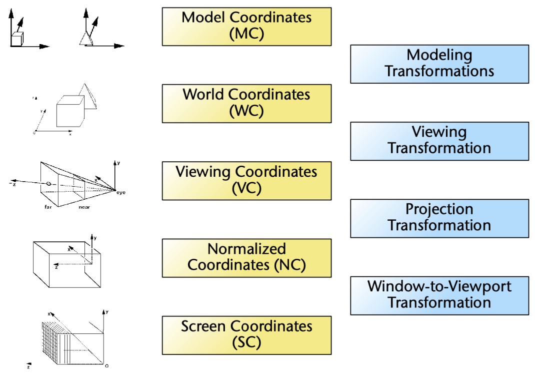

Figure 23: Transformation pipeline steps

11.1 Model Transformations

Model transformations convert local **model coordinates** (**MC**) into global **world coordinates** (**WC**) through a multiplication in homogeneous coordinates with a **model matrix** M.

11.2 Viewing Transformations

Viewing transformations convert global world coordinates into camera **viewing coordinates** (**VC**). Let VPN the view plane normal and VUP the view up vector, then we can define the following variables:

$$z' = \frac{VPN}{||VPN||} \qquad x' = \frac{VUP \times z'}{||VUP \times z'||} \qquad y' = z' \times x'$$

Since we must express everything using the new coordinate system, we can transform the world such that the new coordinate system overlaps with the old one. Let VP be the view point, then the **view matrix** is defined as follows:

$$V = \begin{pmatrix} x'^T_x & x'^T_y & x'^T_z & -x'^T VP \\ y'^T_x & y'^T_y & y'^T_z & -y'^T VP \\ z'^T_x & z'^T_y & z'^T_z & -z'^T VP \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

11.3 Projection Transformations

Projection transformations convert viewing coordinates into **normalized coordinates** (**NC**) $[-1,1]^3$. We need to specify the **viewing frustum**, meaning the range in which objects are visible:

- **Near plane**: $z = -n$
- **Far plane**: $z = -f$
- $0 < n < f$

These transformation can be divided in perspective and orthographic.

11.3.1 Perspective Projections

In **perspective projections**, all rays go through the camera and the viewing frustum is mapped to a unit cube defined as follows:

$$[-1,1] \times [-1,1] \times [-1,1]$$

Given a vertical opening angle β and an aspect ratio γ, then we have the following relations:

$$\frac{n}{r} = cot(\frac{\beta/2}{\gamma}) \quad \frac{n}{t} = cot(\frac{\beta}{2})$$

The **perspective matrix** is computed as follows:

$$P_{persp} = \begin{pmatrix} \frac{n}{r} & & & \\ & \frac{n}{t} & & \\ & & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ & & -1 & 0 \end{pmatrix}$$

11.3.2 Orthographic Projections

In **orthographic projections**, all rays are parallel to the viewing direction, thus the viewing frustum is an axis-aligned cuboid mapped to a unit cube as follows:

$$[-r,r] \times [-t,t] \times [-n,-f] \ \rightarrow \ [-1,1]^3$$

The **orthographic matrix** is defined as follows:

$$P_{ortho} = \begin{pmatrix} \frac{1}{r} & & & \\ & \frac{1}{t} & & \\ & & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ & & 0 & 1 \end{pmatrix}$$

11.4 Window-to-Viewport Transformations

Window-to-Viewport transformations convert normalized coordinates into **screen coordinates** (**SC**) by scaling and translating such that:

- x is mapped linearly: $[-1, 1] \rightarrow [0, w]$
- y is mapped linearly: $[-1, 1] \rightarrow [0, h]$
- z is mapped linearly: $[-1, 1] \rightarrow [0, 1]$

11.5 Transformations in Rasterization

Matrix transformations are per-vertex operations, meaning they should be implemented in a vertex shader:

$$v_{out} = PVMv_{in}$$

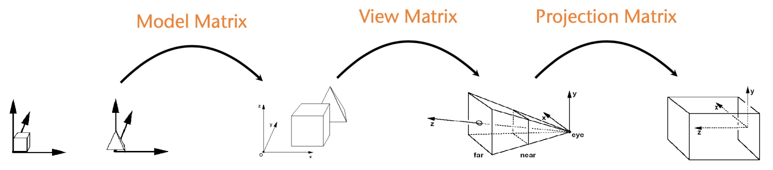

Figure 24: Transformation matrices effects

12 Light Computation

Inputs for light computation such as position, color, and normal vector are defined per vertex. The Phong reflectance model is defined as follows:

$$I = k_a \cdot i_a + k_d(\vec{L} \cdot \vec{N}) \cdot i_d + k_s(\vec{R} \cdot \vec{V})^s \cdot i_s, \text{ where}$$

- $\vec{N}$: normal vector transformed with model and view matrices
- $\vec{L}$: light vector transformed with a view matrix
- $\vec{V}$: viewer position transformed with model and view matrices
- $\vec{R}$: reflection direction computed in a fragment shader from $\vec{L}$ and $\vec{N}$

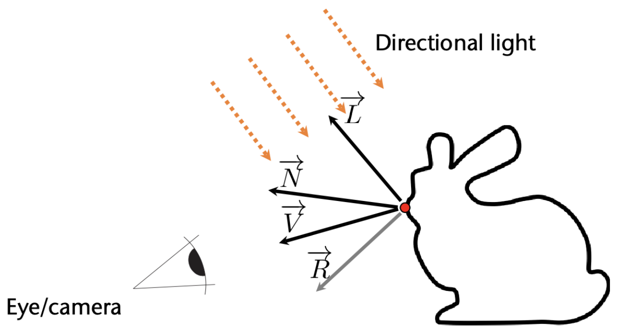

Figure 25: Phong reflectance model

12.1 Shading and Illumination

In **Gouraud shading**, the color is computed per vertex and then interpolated; the Phong model is computed in a vertex shader. On the other side, in **Phong shading**, all information for light computation is interpolated, then the color is computed per fragment; the Phong model is computed in a fragment shader. The most efficient method is the Phong shading since each fragment is shaded separately in a fragment shader.

13 Texture Mapping

In order to implement **texture mapping**, we augment simple geometry with extra information while shading.

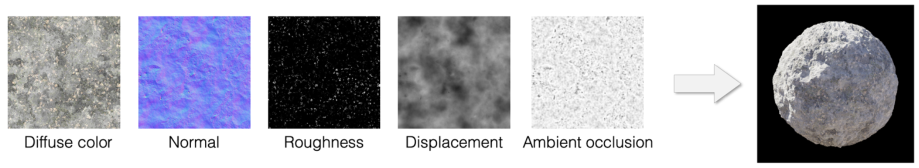

Figure 26: Texture mapping concept

The texture itself is an image composed of **texels** and with a parameter space Ω. It obeys to the following procedural pattern:

$$M : (u, v) \to color$$

Given an object S, usually a triangle mesh, we find the correspondences between S and the $2D$ plane through the following **parameterization** f:

$$f : \Omega \leftrightarrow S$$

The following example computes $f(u, v)$ for a sphere:

$$\Omega = \{(u, v) \in [0, 1]^2\}$$
$$S = \{(x, y, z) \in \mathbb{R}^3 : x^2 + y^2 + z^2 = 1\}$$
$$f(u, v) = (t \cdot cos(2\pi u),\ t \cdot sin(2\pi v)$$

13.1 Texturing Triangles

Given a triangle $[p_0, p_1, p_2]$ and a point $p_i \in \mathbb{R}^3$, we associate the following texture coordinates:

$$t_i = (u_i, v_i) \in \mathbb{R}^2,$$

then we map the texture triangle $[t_0, t_1, t_2]$ linearly to the triangle.
First, we interpolate (u, v) from uv-coordinates of the vertices, then we fetch the corresponding texture value.

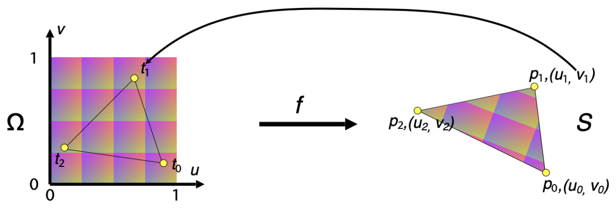

Figure 27: Texturing a triangle

13.2 Texture Access

In order to texture the fragments s in the screen space, we take the four closest texels through nearest neighbors interpolation and interpolate between them through **bilinear filtering**.

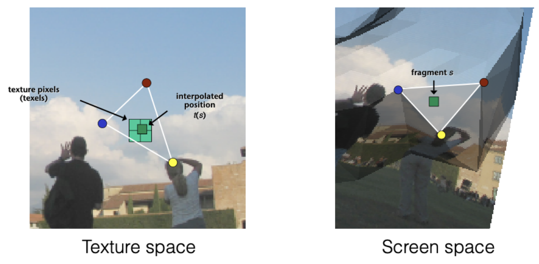

Figure 28: Texture access

13.3 Idea Behind Texture Mapping

We read new **diffuse colors** from the texture and then compute the Phong model using them. Moreover, textures supply high resolution **displacement** information which is used to displace vertices along the normal direction.

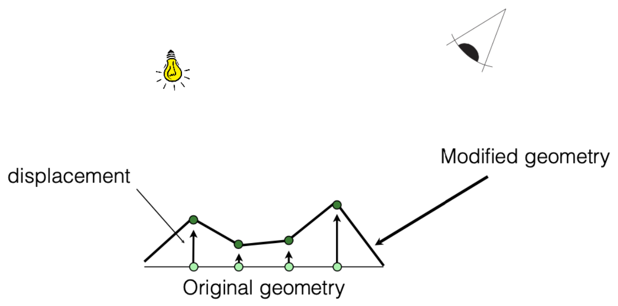

Figure 29: Displacement representation

14 Shadows in Rasterization

Since the graphics pipeline only allows to do local computations, i.e. per-vertex and per-fragment, we need a sort of preprocessing.

14.1 Shadows Preprocessing

First, we compute all the distances z_i from points to a light source L and store them in the z-buffer. While rendering, we test the current distance to L with the stored smallest distance z_{min}. The z-buffer is thus used as a **shadow map**, one per light source.
In per-fragment computation, we transform its coordinates into the local coordinates system of L. The fragment is in shadow if:

$$z > z_{min}(x, y)$$

15 Introduction to Physics-based Animation

Objects motions are computed using transformation matrices. In order to obtain an **animation**, we must compute each frame's **simulation state**, consisting of a **position** $x(t)$ and a **velocity** $v(t)$, as follows:

$$\begin{gathered} x(t) \\ v(t) = \dot{x}(t) \end{gathered}$$

15.1 Simulation Step

According to Newton's first, no external forces are considered. Thus, given a time step Δt, we have the following relations:

$$\begin{gathered} v(t + \Delta t) = v(t) \\ x(t + \Delta t) = x(t) + \Delta t \cdot v(t) \end{gathered}$$

Moreover, according to Newton's second law and given a force F, a mass m, and an acceleration a, we have the following relations:

$$\begin{gathered} F(t) = m \cdot a(t) \ \rightarrow \ a(t) = F(t)/m \\ a(t) = \ddot{x}(t) \end{gathered}$$

15.2 Particle Simulation

In **particle simulation**, we perform the following steps:

1. Initialize position x, velocity v, and mass m of each particle
2. Set the time step Δt
3. Until simulation ends:
    (a) Compute the force F acting on the particle
    (b) For each particle:
        - Update the velocity: $v = v + \Delta t \cdot F/m$
        - Update the position: $x = x + \Delta t \cdot v$

15.3 Mass-Spring System

In the **mass-spring system**, we can model complex objects as particle systems connected with springs. The interaction with the environment is modeled by external forces. According to the **Hooke's law**, a spring in its rest shape with its rest length r does not exert force; otherwise, the force is proportional to the expansion. Defining the spring stiffness with k and its expansion/compression as x, we have the following relation:

$$F = k \cdot x$$

Moreover, defining the spring end points as x_p and x_q respectively, we obtain the following relation:

$$F_p = k \cdot \big(\frac{||x_q - x_p||}{r} - 1 \big) \cdot \frac{x_q - x_p}{||x_q - x_p||}$$

We perform the following steps:

1. Initialize position x, velocity v, and mass m of each particle
2. Set the time step Δt
3. Define springs: p, q, r, k, where q, p are the indices of vertices
4. Until simulation ends:
    (a) Initiate the force F acting on the particle
    (b) For each spring:
        - Compute the forces exert by the spring on p and q
        - Add the forces to F_p and F_q
    (c) For each particle:
        - Update the velocity: $v = v + \Delta t \cdot F/m$
        - Update the position: $x = x + \Delta t \cdot v$

In case of static particles, the position should not be updated and the velocity should be zero.
Now we can introduce the concept of **damping**. Defining the damping coefficient as d, we have the following relation:

$$\hat{F} = d \cdot \langle \frac{v_q - v_p}{r}, \frac{x_q - x_p}{||x_q - x_p||} \rangle \cdot \frac{x_q - x_p}{||x_q - x_p||}$$

16 Appendix A - CG Fundamentals

16.1 Trigonometry

16.1.1 Pythagorean Identity

For any $\alpha \in \mathbb{R}$:

$$sin^2\alpha + cos^2\alpha = 1$$

16.1.2 Half-Angles

For any $\alpha \in \mathbb{R}$:

$$sin\frac{\alpha}{2} = \sqrt{\frac{1 - cos\ \alpha}{2}}, \qquad cos\frac{\alpha}{2} = \sqrt{\frac{1 + cos\ \alpha}{2}}, \qquad tan\frac{\alpha}{2} = \frac{sin\ \alpha}{1 + cos\ \alpha}$$

16.2 Linear Algebra

16.2.1 Vectors

In vector spaces $\mathbb{R}^n$ with coordinates $x_1, x_2, \dots, x_n$:

$$x = (x_1, x_2, \dots, x_n)^T$$

16.2.2 Matrices

A matrix $A \in \mathbb{R}^{mxn}$ with m rows and n columns is an array of $m{\cdot}n$ real numbers $a_{i,j}$, for $i = 1, \dots, m$ and $j = 1, \dots, n$:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

16.2.3 Determinant

In a 2-by-2 matrix $A \in \mathbb{R}^{2x2}$ is:

$$detA = det\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} = \begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} = a_{1,1} \cdot a_{2,2} - a_{1,2} \cdot a_{2,1}$$

16.2.4 Normalization

The norm of a vector $x = (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n$ is:

$$||x|| = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

To normalize a vector, we can compute:

$$y = \frac{x}{||x||} \rightarrow ||y|| = 1$$

16.2.5 Dot Product

The dot product of two vectors $x = (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n$ and $y = (y_1, y_2, \dots, y_n)^T \in \mathbb{R}^n$ is defined as:

$$< x, y >= \sum_{i=1}^{n} x_i \cdot y_i = x_1 \cdot y_1 + x_2 \cdot y_2 + \cdots + x_n \cdot y_n$$

Denoting the angle between x and y by α, we obtain:

$$< x, y >= cos\ \alpha \cdot ||x|| \cdot ||y||$$

16.2.6 Cross Product

The cross product of two vectors $x = (x_1, x_2, x_3)^T \in \mathbb{R}^3$ and $y = (y_1, y_2, y_3)^T \in \mathbb{R}^3$ is defined as:

$$z = x \times y = (z_1, z_2, z_3)^T,\ z_1 = \begin{vmatrix} x_2 & y_2 \\ x_3 & y_3 \end{vmatrix},\ z_2 = - \begin{vmatrix} x_1 & y_1 \\ x_3 & y_3 \end{vmatrix},\ z_3 = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}$$

Denoting the angle between x and y by α, we obtain:

$$||x \times y|| = sin\ \alpha \cdot ||x|| \cdot ||y||$$