1 Projections

World Coords are *normalized* to **Device Coords**:

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

$$Y_v = \frac{(Y_w - W_{ymin})(V_{ymax} - V_{ymin})}{W_{ymax} - W_{ymin}} + V_{ymin}$$

A **mesh** is a collection of **polygons**, which consist of *vertex* & *face* data structures. To draw, we must **project** 3D coordinates onto a 2D plane:

- **Orthographic**: viewpoint at $z = -\infty$, plane of projection at $z = 0$. All projectors have same direction d, with equation $P = V + \mu d$, hence the projected location is $P = \begin{pmatrix} X_v & Y_v & 0 \end{pmatrix}$.
- **Perspective**: viewpoint at **frustrum** $z = -f$. Then, $\mu_p = f \div V_z$, so $P = \begin{pmatrix} \frac{fV_x}{V_z} & \frac{fV_y}{V_z} & 0 \end{pmatrix}$.

2 Transformations

To draw graphics from any angle, its easier to have the *viewpoint at the origin*, and the *z-axis as the direction of view*. To allow matmul to model all transformations (including translation), use **4D homogeneous coords** $(p_x, p_y, p_z, s) \Leftrightarrow (\frac{p_x}{s}, \frac{p_y}{s}, \frac{p_z}{s})$. Usually, $s = 1$.

Affine Transformations *preserve parallel lines*. Most trans (except perspective projection) are affine. Transformations are not commutative.

2.1 Translation Transformations

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.2 Scaling Transformations

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \\ s_z p_z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.3 Rotation Transformations

Rotation is defined by **axis** and **angle**:

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

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

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

Rotations have **direction** (*anti-clockwise when looking along the axis of rotation*). Rotation is clockwise when looking back towards the origin from the positive axis direction. To find the inverse we rotate by angle $-\theta$.

2.4 Flying Sequences

Let required viewpoint be $C = (C_x, C_y, C_z)$ and required view direction be $d = (d_x, d_y, d_z)^T$. To transform coords into a system centered at the viewpoint, looking in the positive z-direction:

1. Translate abt origin, $\mathcal{A} = \begin{pmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$
2. Rotate about y-axis until d in yz-plane. Define $v = \sqrt{d_x^2 + d_z^2}$, $\cos\theta = \frac{d_z}{v}$, $\sin\theta = \frac{d_x}{v}$. Rotation matrix is: $\mathcal{B} = \begin{pmatrix} \frac{d_z}{v} & 0 & -\frac{d_x}{v} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{d_x}{v} & 0 & \frac{d_z}{v} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.
3. Rotate about x-axis until d points in z-axis. Let $v = \sqrt{d_x^2 + d_z^2}$, $\cos\phi = \frac{v}{|d|}$, $\sin\phi = \frac{d_y}{|d|}$. Rotation matrix is: $\mathcal{C} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{v}{|d|} & -\frac{d_y}{|d|} & 0 \\ 0 & \frac{d_y}{|d|} & \frac{v}{|d|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

These can be combined as $\mathcal{T} = \mathcal{CBA}$, which is applied to every point. The view is now in **canonical** form and we can apply projection.

About general line, first rotate general line to align it with z-axis, then rotate, then rotate back: $\mathcal{T} = \mathcal{A}^{-1}\mathcal{B}^{-1}\mathcal{C}^{-1}\mathcal{R}_z\mathcal{CBA}$. Also similar effects (*scaling obj in-place*) as $\mathcal{T} = \mathcal{A}^{-1}\mathcal{B}^{-1}\mathcal{C}^{-1}\mathcal{SCBA}$.

2.5 Projection Matrices

Canonical orthographic projection simply drops the z coord with $\mathcal{M}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

Canonical perspective projection does $\mathcal{M}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{pmatrix}$ such that $\mathcal{M}_p \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ \frac{p_z}{f} \end{pmatrix}$. We can normalize this to get $\begin{pmatrix} \frac{p_x f}{p_z} \\ \frac{p_y f}{p_z} \\ f \\ 1 \end{pmatrix}$.

All projection matrices are **singular**, or *non-invertible*.

2.6 Transformation Matrices

Homogeneous coords can be:

- **Position Vectors** have non-zero final ordinate $s > 0$. Can be normalized to *Cartesian* form.
- **Direction Vectors** have a zero final ordinate $s = 0$. These have direction & magnitude.

Dir + Dir = Dir; Dir + Pos = Pos; Pos + Pos = Midpoint

$$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In transformation mat, bottom row is always $(0,0,0,1)$. Cols are 3 dir vectors & 1 pos vector.

- If we transform dir vector, its only affected by first 3 cols.
- If we transform pos vector, its affected by all 4 cols.
- If we dont shear, then q, r, s remain orthogonal, $q \cdot r = q \cdot s = r \cdot s = 0$.

Transformation mat cols represent cols of new coord system in terms of the old:

- q is the transformed x-axis.
- r is the transformed y-axis.
- s is the transformed z-axis.
- C is the new origin.

2.7 Dot Product as Projection

The dot product is defined as $P \cdot u = |P||u|\cos\theta$, where θ is the angle between P and u. If u is along a coord axis then $P \cdot u$ is the ordinate of P in the direction of u. Hence, the dot product can be used to express a projection, $P'_x = (P - C) \cdot u = P \cdot u - C \cdot u$ where u is a unit direction vector. So, to transform (x, y, z) into (u, v, w) coordinates, we can use matrix:

$$\begin{pmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -C \cdot u \\ v_x & v_y & v_z & -C \cdot v \\ w_x & w_y & w_z & -C \cdot w \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

To solve the flying sequence problem, define:

- $w = \frac{d}{|d|}$ as d is the view direction.
- $u = \frac{p}{|p|}$ where p is a vec in the horizontal dir (i.e. $p_y = 0$).
- $v = \frac{q}{|q|}$ where q is a vec with a +ve y component (i.e. $q_y = 1$).

To find p_x, p_z, q_x and q_z, use the orthogonality conditions $d = p \times q$, so $p = (d_z, 0, -d_x)$. Since $p \cdot q = 0$, we have: $p_x q_x + p_z q_z = 0$ and $d_y = p_z q_x - p_x q_z$.

3 Clipping

Portions outside the **view frustrum** (defined by *6 planes*) are not rendered. We can clip in:

- **Worldspace**, natural & not degenerate.
- **Clipspace** (*after perspective trans, before perspective divide*), canonical & simple to impl.
- **Screenspace**, fails when objs in cam plane.

A plane splits a space into **halfspaces**. A point is inside a halfspace if it is on the inside side of the plane. Given a plane $Ax + By + Cz + D = 0$, with normal vector $(A, B, C, D)^T$ (*in homogeneous coords with normalized (A,B,C)*), point p in /halfspace if $d = H \cdot p = H^T p > 0$. When clipping, test:

$$\begin{aligned} H_{near} &= (0, 0, 1, -near)^T \\ H_{far} &= (0, 0, -1, far)^T \\ H_{bottom} &= (0, near, -bottom, 0)^T \\ H_{top} &= (0, -near, top, 0)^T \\ H_{left} &= (-near, 0, left, 0)^T \\ H_{right} &= (near, 0, -right, 0)^T \end{aligned}$$

3.1 Clipping Lines

To intersect line & plane, use parametric line equation $L(\mu) = p_0 + \mu(p_1 - p_0) = \mu p_1 + (1 - \mu)p_0$:

1. For any vector v in the plane $n \cdot v = 0$, let the intersection point be $\mu p_1 + (1 - \mu)p_0$.
2. A vector in that plane is given by $\mu p_1 + (1 - \mu)p_0 - v$.
3. So, $n \cdot (\mu p_1 + (1 - \mu)p_0 - v) = 0$.
4. Solve for μ and find the point of intersection.

Take two points either side of the plane that make a line, p and q. Then:

- If $H \cdot p > 0$ and $H \cdot q < 0$ then clip q to plane.
- If $H \cdot p < 0$ and $H \cdot q > 0$ then clip p to plane.
- If $H \cdot p < 0$ and $H \cdot q < 0$ then discard line.
- If $H \cdot p > 0$ and $H \cdot q > 0$ then keep line.

This is implemented in hardware & fast.

3.2 Clipping Objects

In **convex object**, a line joining any 2 points on boundary lies inside obj. Obj is the intersect of planar halfspaces. To test convexity, store normal vector n for each face of obj: P is inside face if θ is acute, $\cos\theta > 0$, or $n \cdot (P - A) > 0$, where A is a point on the face.

- To find the normal vector of a face, we can take the cross prouct of two edges of the face (or any vector in the plane).
- We must ensure normal vector points *outwards* from the obj. We can do this by ensuring the vertices of the face are ordered *counter-clockwise* when viewed from outside the object.
- To check this dir, we need to look at the rest of the object.

Clipping for **concave** objects uses ray containment test: find all intersects between ray & polygon edges (faces). If num intersects is odd, point is contained. To find intersects, define ray $R = T + \mu d$, $\mu > 0$, choosing any d. Solve intersect $T + \mu d = V_2 + \nu(V_1 - V_2)$ for μ, ν. Intersect is valid if $\mu > 0$ and $0 \le \nu \le 1$. Extends to 3D:

1. Calc intersect of ray with plane of each face.
2. If $\mu > 0$, check intersect point is contained in the face (*not trivial, but can be reduced to 2D under orthographic projection*).

To clip to concave volumes, we may need to split the lines & volumes into segments & concave parts. Usually, concave volumes are unused.

4 Graphics Pipeline & APIs

Pipeline is **declarative** (provides abstract scene def, system renders it) or **imperitive** (programmer specifies each rendering step). The pipeline:

1. **Modelling Transformations**: 3D models defined in their own coord system. This step **orients** models within common coord frame (*worldspace*).
2. **Illumination (Shading)**: Vertices are **lit** (*shaded*) according to material properties, surface properties and light sources, using a local lighting model.
3. **Viewing Transformation**: Scene is transformed from worldspace to **camera space** (*viewspace/eyespace*). Viewing pos is transformed to origin, viewing dir is aligned with the -ve z-axis.
4. **Clipping**: Portions of objs outside **view frustrum** removed. Output in **Normalized Device Coords (NDC)**.
5. **Projection**: The 3D scene is projected onto a 2D image plane, using either **perspective** or **orthographic** projection. This converts NDC to **screen space**.
6. **Scan Conversion**: rasterizes objects into pixels, interpolating values inside objects.
7. **Display**: handle occlusion and transparency blending using a **depth buffer** (*z-buffer*) and outputs the final image to the screen.

Pipeline utilises **GPU**, with *programmable shaders*:

- **Vertex**: foreach vertex, allows custom transformations & lighting calc. Take input vertex stream of arbitrary vertex attrs (*pos, normal, col, etc*), transforms it into stream of verts mapped onto screen (using `ModelViewProjection` Matrix), composed of their clipspace coords & varying attrs (interpolated across the primitive).
- **Fragment**: executed for each pixel fragment, allowing custom color and texture calculations.

5 Shading Languages (OpenGL)

- **Contexts**: an *instance* of OpenGL.
- **Resources**: sources of inputs and sinks for outputs.
- **Objects**: are identified with an *ID*. Each command has an object bound to the target. **Buffer objects** store an array of unformatted memory allocated by the `OpenGL` context.
- **Primitve Types**: e.g. *LINES*, *TRIANGLES*, *LINE_STRIP*, etc.

6 Illumination & Shading

Color & shading intensity of a point depends on its *characteristics* (i.e. pos, surf normal, **albedo**) & *light sources* that illuminate it. Defininitions:

- **Albedo** is an object's ability to absorb light energy.
- **Reflection** is the process by which electromagnetic flux incident on a surface leaves the surface without a change in frequency.
- **Reflectance** is the fraction of incident flux that is reflected by a surface.

6.1 BRDFs

Reflectance is given by a **bidirectional reflectance distribution func (BRDF)**: $f_r(\theta_i, \phi_i, \theta_r, \phi_r) = \frac{dL_r(\theta_r, \phi_r)}{dE_i(\theta_i, \phi_i)}$; ratio of reflected radiance in a given dir to the incident irradiance from a given dir.

Isotropic BRDFs: simplified models, assuming reflectance equal in all dirs, $f_r(\theta_i, \theta_r, \phi_r - \phi_i) = f_r(\theta_i, \theta_r, \phi_d) = \frac{dL_r(\theta_r, \phi_d)}{dE_i(\theta_i)}$. Reduces param count to 3, a common assumption.

Anisotropic BRDFs: complex models, accounting for dir dependence of reflectance, *e.g. brushed metal/hair*. Requires more params to describe reflectance behavior. BRDFs may be:

- **Non Negative**: $f_r(\theta_i, \phi_i, \theta_r, \phi_r) \ge 0$.
- **Energy Conserving** $\forall \theta_i, \phi_i . \int_\Omega f_r(\theta_i, \phi_i, \theta_r, \phi_r)\, d\mu(\theta_r, \phi_r) \le 1$
- **Reciprical**: $f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\theta_r, \phi_r, \theta_i, \phi_i)$

A more precise model for reflection is the **bidirectional scattering surface reflectance distribution function (BSSRDF)**, includes both reflection and transmission.

6.2 Computing Reflected Radiance

$$L_r(\omega_r) = \int_\Omega f_r(\omega_i, \omega_r)\, dL_i(\omega_i) \cos(\omega_i \cdot n)\, d\omega_i$$

$$\text{where } \omega = \langle \theta, \phi \rangle$$

Or, in the discrete case, for n point light sources:

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

Where ω is the direction of the incoming light, θ is the angle between the surface normal and the incoming light, Φ_{sj} is the radiant flux of the j-th light source, and d_j is the distance from the surface point to the j-th light source.

6.3 Ideal Reflectance

For **ideal diffuse reflectance**, assume surf reflects equally in all dirs. In reality, a very rough surface (e.g. *chalk, dull paints*). Here, BRDF iss cost, so $L_r(\omega_r) = \int_\Omega f_r(\omega_i, \omega_r)\, dE_i(\omega_i) = f_r E_i$. This reflects light according to **Lambert's Cosine Law** (*intensity of reflected light ∝ cosine of angle between surf norm & incoming light*). For IDR, given the diffuse reflection coeff k_d, surf norm n & light dir l, reflected radiance is given by: $L(\omega_r) = k_d \max(0, n \cdot l) \frac{\Phi_s}{4\pi d^2}$. Here, dot product *clamped* to avoid -ve vals (correspond to light coming from behind surf). $\frac{\Phi_s}{4\pi d^2}$ accounts for the attenuation of light with dist, following inverse square law.

In **ideal specular reflectance**, reflection is only at a mirror angle. This is view dependent, and is a very smooth surface (e.g. *polished metal*). Real-world mats are not perfect reflectors.

We expect most of the reflected light to be reflected in the direction of the ideal ray. Because of microscopic surface variations we might expect some light to be reflected at a slight offset to the ideal reflected ray. As we move farther away (in the angular sense) from the reflected ray we expect to see less light reflected.

6.4 Phong Model

Decides how much light is reflected.

- n is the surface normal.
- l is the light direction.
- r is the ideal reflected ray.
- θ is the angle between l and n, or between n and r.
- v is the view direction.
- α is the angle between r and v.
- k_s is the specular reflection coefficient.
- q is the specular reflection exponent. Decides how glossy/blurry the reflection is. When $q = 1$, we approximate a diffuse material. As $q \to \infty$, we approximate a perfect mirror.

Then a regular **Phong Model**:

$$L(\omega_r) = k_s (\cos\alpha)^q \frac{\Phi_s}{4\pi d^2} = k_s (v \cdot (2(n \cdot l)n - l))^q \frac{\Phi_s}{4\pi d^2}$$

A **Blinn-Phong Model** uses the **halfway vector** $h = \frac{l+v}{\|l+v\|}$ instead of the ideal reflected ray. When β is the angle between n and h, the specular term is given by:

$$L(\omega_r) = k_s (\cos\beta)^q \frac{\Phi_s}{4\pi d^2} = k_s (n \cdot h)^q \frac{\Phi_s}{4\pi d^2}$$

Ambient illumination is a hack to represent the reflection of all indirect illumination, avoiding the complexity of global illumination: $L(\omega_r) = k_a$. Adding it all together gives the **Phong illumation model**:

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

The $\frac{1}{4\pi d^2}$ term is responsible to make sure light falls off according to an inverse square law. Although physically correct, can be visually unappealing. Common heuristic is to replace with $\frac{1}{4\pi(d+s)}$, where s is a heuristic const.

6.5 Shading

Flat shading: each polygon is shaded uniformly over surf. Computed by taking point in the centre & surf norm vector. Only diffuse & ambient components used. **Gourard shading** (aka *interpolation shading*),: compute shade value at each vertex, interpolate for shade value at each pixel. **Phong shading**: interpolate norms across tris, compute shade value at each pixel.

This interpolation is **barycentric interpolation**. Any point on a triangle V can be represented in terms of its three vertices:

$$V = V_1 + \mu_1(V_2 - V_1) + \mu_2(V_3 - V_1)$$

The average normal vec at that point is given by:

$$a = n_1 + \mu_1(n_2 - n_1) + \mu_2(n_3 - n_1) \quad n = \frac{a}{\|a\|}$$

These formulae come from barycentric coords where:

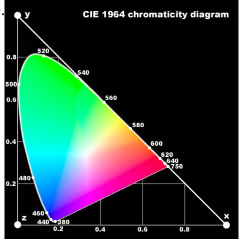
$$\begin{aligned} p &= a + \beta(b - a) + \gamma(c - a) \\ &= \alpha a + \beta b + \gamma c \quad \text{where } \alpha = 1 - \beta - \gamma \end{aligned}$$

In other words, $p(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$, where α, β, and γ are the barycentric coordinates of p with respect to the triangle defined by a, b, and c. *The interpolation calculates may be done in either or 2D or 3D. For specular reflection, it must be done in 3D.*

7 Colour

Pure color is a wave with wavelength λ and intensity I. In practice light has a mixture of wavelengths with an energy dist. Most perceivable colours can be represented as linear combination of basis funcs R, G, B.

CIE diagram: 2D repr of color space. Each point corresponds to color. x, y coords represent chromaticity, z represents luminance.

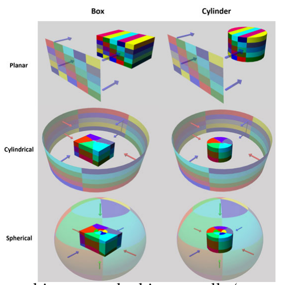


The edges of the horseshoe are coherent λ, and the colours inside can only be represented as a mixture of the pure colours on the edge. The **white point** is the point where all three basis functions are equal, and represents the color white. It is located when $R = G = B$ at $x = y = z = \frac{1}{3}$. *Displays produce a triangular subset of the colors in the CIE diagram, and the vertices of the triangle correspond to the primary colors used by the display.*

8 Texture Mapping

A **texture** can be a **2D function** (a u, v mapping), a **raster image** (texels) or even a **3D function** (a u, v, w mapping to a volume). We have tex coords (t, s) and surface coords (u, v). Textures are unwrapped from 3D to 2D, and then mapped back to 3D. This is done by defining a mapping from surface coords to tex coords:

- **Planar Mapping**: The texture is projected onto the surface along a plane. This is the simplest mapping, but it can cause severe distortion if the surface is not flat.
- **Cylindrical Mapping**: The texture is projected onto the surface along a cylinder. This is suitable for surfaces that are roughly cylindrical, but can still cause distortion.
- Also **Cube Mapping** & **Spherical Mapping**.



To generate this, we can do this manually (**unwrapping**) or procedurally. Since we texture map after projection to 2D, we get perspective distortion (i.e. *equal distances in world space do not map to equal distances in screen space*). To avoid this, we can do **perspective correct texture mapping** by interpolating the texture coordinates in screen space and then dividing by the depth.

8.1 Tiling

- **Static Color**: The texture is extended with a fixed color.
- **Clamped**: The texture is extended with the edge color.
- **Repeat**: Texture is repeated.
- **Mirrored Repeat**: The texture is repeated and mirrored to hide repetitions (*seams*).

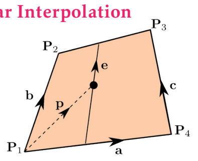
Texture Synthesis algorithms can generate larger textures from smaller samples, by analyzing the patterns in the sample and creating new textures that maintain the same visual characteristics.

8.2 Perspective Correct Interpolation

We must specify texcoords for vertex, then linearly interpolate these in screenspace. Simple interpolation causes perspective distortion (*equal dists in 3D dont map to equal dists in screenspace*). To fix, use **perspective correct projection**:

1. Assign paramtere t to 3D vertices p, r.
2. t controls linear blend of texcoords of p and r s.t. $t = 0$ at p and $t = 1$ at r.
3. Assume for simplicity, wlog $z = 1$.
4. Compute $t'_p = t_p/z_p$ and $t'_r = t_r/z_r$.
5. $t'_q = \text{lerp}(t'_p, t'_r, t)$. Hence, $t_q = z_q \cdot t'_q$.
6. In summary, $t_q = \text{lerp}(\frac{t_p}{z_p}, \frac{t_r}{z_r}) \div \text{lerp}(\frac{1}{z_p}, \frac{1}{z_r})$.

8.3 Bilinear Interpolation



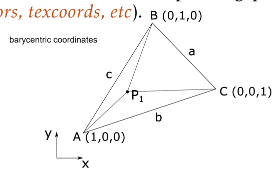
If p is the pixel to be centered, we know $p = \alpha a + \beta b + \alpha\beta(c - b)$. If the map is to a parallelogram, then $b = c$, and it simplifies to $p = \alpha a + \beta b$.

8.4 Uses

A **bump map** is a black and white image. Partial derivatives of the bump map are used to perturb the normal vector, which is then used in lighting calculations to create the illusion of surface detail without increasing geometric complexity. A **displacement map** actually modifies the geometry of the surface based on the texture, creating real geometric detail. This can be more computationally expensive than bump mapping, but it can produce more realistic results. We can simulate reflections by using the direction of the ray to index a spherical texture map at "*infinity*". This is an **environment map**.

9 Rasterization

Rasterization determines which pixels are drawn into the framebuffer, interpolating parameters (*colors, texcoords, etc*).



We can interpolate using barycentric coords:

$$p = a + \beta(b - a) + \gamma(c - a)$$
$$= \alpha a + \beta b + \gamma c \quad \text{where } \alpha = 1 - \beta - \gamma$$

In other words, $p(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$, where α, β, and γ are the barycentric coordinates of p with respect to the triangle defined by a, b, and c. Now, to rasterize a point in the triangle, we check $0 < \alpha, \beta, \gamma < 1$.

The **signed distance** (the distance to the edge, positive if inside, negative if outside) can be computed with an implicit line equation:

$$f_{ab}(x,y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a$$
$$f_{bc}(x,y) = (y_b - y_c)x + (x_c - x_b)y + x_b y_c - x_c y_b$$
$$f_{ca}(x,y) = (y_c - y_a)x + (x_a - x_c)y + x_a y_c - x_c y_a$$

To compute a barycentric coordinate for p, we should choose k_{ac} such that $k_{ac} f_{ac}(x,y) = \beta$, etc for k_{ab} and k_{bc}. In example:

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

So, in general, the barycentric coordinates of $p = (x, y)$ solve:

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

Also, the *trilinear coordinates* $P_t(t_1, t_2, t_3)$ are easily converted into barycentric coordinates as (at_1, bt_2, ct_3) where a, b, and c are the side lengths of the triangle. Similarly, barycentric coordinates can be converted into trilinear coordinates as $(\alpha/a, \beta/b, \gamma/c)$. Instead of checking $0 < \alpha, \beta, \gamma < 1$, graphics card used optimized incremental methods for **triangle rasterization**.

10 Visibility

When triangles overlap, perform **hidden surface removal** (keep track of visible surfaces). One option is the **painters algorithm** (sorting triangles by z-value), but this fails when triangles intersect. Alternatively, a **depth buffer** (*z-buffer*) is used. Each fragment gets a z value in screen space, but only the value with the smallest z value is drawn. This is implemented in hardware and is very efficient. Every time something is drawn to the framebuffer, the depth buffer is also updated.

11 Anti-Aliasing

Due to visual texture mapping, we get aliasing artifacts at boundaries and in textures. The solution is to **blur** boundaries to reduce the effect. **Supersampling** computes the picture at higher resolutions and downsampling at boundaries. More common solution is a **convolution filter** that blurs the image. For example, a **box filter** averages the 4 nearest pixels, while a **Gaussian filter** gives more weight to closer pixels. This is very fast, but does degrade the image quality. To antialias textures, we identify a point in the texture map and return an average around that point (*mipmapping*).

12 Raytracing

- **Direct Illum**: surf point recieves light directly from all light sources in scene.
- **Global Illum**: surf point recieves light after rays interact with other objs in scene (*also computes shadows, reflections, refractions, etc*).

12.1 Raycasting

1. $\forall$ pixel, draw ray from camera thru pixel into scene.
2. $\forall$ ray, check for intersections with objs in scene.
3. If intersection found, compute pixel col based on material props of obj & lighting in scene.
4. Otherwise, set pixel color to background color.

12.2 Raytracing

- **Shadows**: shadow rays (from intersection point to light sources) determine if point in shadow.
- **Reflections**: reflection rays from intersection point to compute color contrib using material's *reflectivity*.
- **Refractions**: refraction rays from intersection point to compute col contrib using mat's *refractive index*.

A recursive algorithm. Stop at max depth, or if reflected or transmitted contrub below threshold.

For each ray, test which objs intersect ray. If obj intersects, calc dist between viewpoint & intersection. Smallest dist is the visible surface. At that point, local illum computed as before, $L = k_a + (k_d(n \cdot l) + k_s(v \cdot r)^q)I_s$.

12.3 Secondary Rays

Secondary rays originate at intersection points, due to *reflections*, *refractions* & *shadows*: $L = k_a + (k_d(n \cdot l) + k_s(v \cdot r)^q)I_s + k_{\texttt{reflect}} L_{\texttt{reflect}} + k_{\texttt{refract}} L_{\texttt{refract}}$.

Floating point errors cause artifacts: **shadow acne** (self-shadowing) & **light leaks** (missed shadows). To mitigate, use small bias (in the correct dir) when generating secondary rays.

- Obj's **reflection coeff** $k_{\texttt{reflect}}$ determines how much light is reflected. The **mirror reflection** dir is symmetrical wrt normal: $v' = v - (2v \cdot n)n$, v = incoming ray dir, n = unit surf norm.
- Obj's **transparency coeff** $k_{\texttt{transp}}$ determines how much light is transmitted through the obj. The **refraction** dir is given by Snell's law: $\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$ where η_1, η_2 are refractive indexes of the two media, θ_1, θ_2 are angles of incidence & refraction, respectively. So, the dir of the refracted ray is:

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

$$\text{has solution when } (n \cdot v)^2 > 1 - \left(\frac{\eta_2}{\eta_1}\right)^2$$

12.4 Fresnel Factor

Can compute $k_{\texttt{reflect}}$, $k_{\texttt{transp}}$ based on incidence angle using **Fresnel equations**, such that $L = k_{\texttt{fresnel}} L_{\texttt{reflect}} + (1 - k_{\texttt{fresnel}}) L_{\texttt{refract}}$, *more reflection at grazing angles, less at normal incidence.* The **Fresnel coeff** $k_{\texttt{fres}}$ can be approxed using **Schlick's approx**:

$$k_{\texttt{fres}}(\theta) = k_{\texttt{fres}}(0) + (1 - k_{\texttt{fres}}(0))(1 - (n \cdot l))^5$$

Where $k_{\texttt{fresnel}}(0)$ is the Fresnel coefficient at normal incidence, a property of the material .

12.5 Shadows

Adding in **shadow ray** hit s: $L = k_a + s(k_d(n \cdot l) + k_s(v \cdot r)^q)I_s + k_{\texttt{fres}} L_{\texttt{refl}} + (1 - k_{\texttt{fres}}) L_{\texttt{refr}}$. *Soft shadows*: use **area light**, cast many shadow rays to diff points on source, avg results to s.

12.6 Glossy Objects

Glossy objects have blurred reflection. Cast many reflection rays in cone around mirror refl dir, avg results.

12.7 Monte-Carlo Raytracing, Pathtracing

Cast ray from eye to each pixel, then randomly cast rays in hemisphere above the surface point, average results to compute color. Simulates global illum, indirect lighting, caustics, etc. Very computationally exp method, but realistic. **Path tracing**: several primary rays to reduce noise. More efficient than MC, gives better results.

12.8 Intersection Calculations

$\forall$ ray, calc possible intersects with each obj. Ray described parametrically with origin p_0, dir d as $p(\mu) = p_0 + \mu d$. Dir can be calced from viewpoint p_v as $d = \frac{p_0 - p_v}{\|p_0 - p_v\|}$.

- $\mu > 0$: part of the ray behind the viewing plane. All intersection points must have $\mu > 0$ to be visible.
- $\mu < 0$: part of the ray in front of the viewing plane.

12.9 Ray-Sphere Intersection

Point q on surface of sphere satisfies $\|q - p_s\|^2 - r^2 = 0$ where p_s= center, r= radius. Subbing $q = p_0 + \mu d$ gives: $\|p_0 + \mu d - p_s\|^2 - r^2 = 0$. Setting $\Delta p = p_0 - p_s$ we get:

$$\mu^2 + 2\mu(d \cdot \Delta p) + \|\Delta p\|^2 - r^2 = 0$$

$$\mu = -d \cdot \Delta p \pm \sqrt{(d \cdot \Delta p)^2 - \|\Delta p\|^2 + r^2}$$

- If μ has no solutions, there is no intersecton.
- If μ has one solution, the ray is tangent to the sphere.
- If μ has two solutions, ray intersects sphere at 2 points. Use smaller +ve solution (*closer point*).

12.10 Ray-Cylinder Intersection

$p_1 + \alpha\Delta p + q = p_0 + \mu d$ where p_1, p_2 are cylinder endpoints, $\Delta p = p_2 - p_1$, q is vector perpendicular to Δp. α is the param along cylinder axis, μ is the param along the ray. Since $q \cdot \Delta p = 0$, we can say:

$$\alpha(\Delta p \cdot \Delta p) = p_0 \cdot \Delta p + \mu d \cdot \Delta p - p_1 \cdot \Delta p$$

After solving for μ, we can find α_1, α_2 as:

$$\alpha_k = \frac{p_0 \cdot \Delta p + \mu_k d \cdot \Delta p - p_1 \cdot \Delta p}{\Delta p \cdot \Delta p}$$

- If $0 < \alpha_1 < 1$ then were on outside surface of cylinder.
- If $0 < \alpha_2 < 1$ then were on inside surface of cylinder.

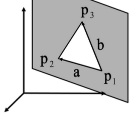
12.11 Ray-Plane Intersection

Given by $p_1 + q = p_0 + \mu d$ where p_1= point on plane, q= vector on plane. Given plane normal n & knowing $q \cdot n = 0$, solve for μ as:

$$(p_0 - p_1) \cdot n + \mu d \cdot n = 0 \quad \mu = \frac{(p_1 - p_0) \cdot n}{d \cdot n}$$

12.12 Ray-Triangle Intersection

Test if: triangle is **front facing**; its **plane intersects** ray; **intersection point inside** triangle.



1. **Front facing** if $d \cdot n < 0$, n is tri's normal. If $d \cdot n = 0$ then ray parallel to triangle plane $\Rightarrow$ no intersection.
2. **Plane equation**: $p_2 - p_1 = a$, $p_3 - p_1 = b$, $n = a \times b$. Then **ray-plane intersect** to get intersect point q.
3. Check q inside triangle: $q = \alpha a + \beta b$ where a, b are tri's edges. *Then, check $\alpha > 0$, $\beta > 0$, & $\alpha + \beta < 1$.*

$$\alpha = \frac{(b \cdot b)(q \cdot a) - (a \cdot b)(q \cdot b)}{(a \cdot a)(b \cdot b) - (a \cdot b)^2} \qquad \beta = \frac{(q \cdot b) - \alpha(a \cdot b)}{b \cdot b}$$

Instead, use **Moller Trumbore Algorithm**:

```
bool i(vec3 v1, vec3 v2, vec3 v3, vec3 origin,
        vec3 dir, float* out):
vec3 edge1, edge2 = v2 - v1, v3 - v1;
vec3 p = cross(dir, edge2);
float det = dot(edge1, p);
if (det > -EPSILON && det < EPSILON)
  return false; // parallel to tri plane
float inv_det = 1.0 / det;
// Calculate barycentric u, v:
vec3 t = origin - v1;
float u = dot(t, p) * inv_det;
if (u < 0.0 || u > 1.0) return false;
vec3 q = cross(t, edge1);
float v = dot(dir, q) * inv_det;
if (v < 0.0 || u + v > 1.0) return false
// Calculate t, ray intersects triangle:
float mu = dot(edge2, q) * inv_det;
if (mu > EPSILON) // ray intersection
  *out = mu; return true;
return false; // No hit, no win
```

12.13 Raytracing Optimisations

Easy to impl, extends to *global illum*, but **slow**. To optimize, use **accel structs**: **Binary Space Parition Trees** (bounding volume hierarchies (BVH) or k-d trees), reducing intersect tests. Grids **regular** (*easy to construct & traverse, but sparse with clustered geom*) or **adaptive** (*complexity matches geom density, expensive to traverse*).

13 Inverse Rendering

Input many images of static scene (unchanging lighting). Output func that renders scene from any point.

13.1 Optimization with SGD

Stochastic grad descent $\min_\theta \|\mathcal{R}_\theta(c) - I^*_c\|$, where:

- I^*_c: **posed** images of scene (camera position c known).
- $\mathcal{R}_\theta$ rendering function parameterized by θ.

Goal: minimize diff between rendered & actual image across all positions. The **initial conditions** very important for reaching a good local minimum.

13.2 Volume Rendering

Functions for volumetric rendering happen to be convenient for SGD optimization. Simplify by ignoring scattering, assuming all particles either absorb/emit light:

$$L(x, \omega) = \int_0^z T(x, x_t)\sigma(x_t)L_e(x_t, \omega)dt$$

- x_t: point inside volume along ray from cam to scene.
- x is the camera position.
- ω: direction of the ray from the camera to the scene.
- $L_e(x_t, \omega)$: **emitted radiance** from point x_t in dir ω.
- $\sigma(x_t)$ is the **density** at point x_t.
- $T(x, x_t)$: **transmittance** from point x to x_t, accounting how much light is absorbed along the path.

Dir component ω allows capturing view-dependent effects, *e.g. specular highlights, reflections*. To approx this, take samples along ray & sum their contributions, (*efficiently computed using numerical integration techniques*).

Also, $T(x, x_t) = \exp(-\int_0^t \sigma(x_s)ds)$ accounts for cumulative absorption along ray. So $L(x, \omega) = \sum_{i=1}^n T_i \alpha_i c_i$:

- $T_i = \prod_{j=1}^{i-1}(1 - \alpha_j)$: transmittance up to i-th sample.
- α_i: **segment opacity** for i-th sample, computed as $\alpha_i = 1 - \exp(-\sigma(x_i)\delta t)$, δt= dist between samples.
- c_i: **color** emitted from i-th sample, $c_i = L_e(x_i, \omega)$.

13.3 3D Representations

Explicit repr defines each point, **implicit** states point using a condition func. With **c-level sets**, represent surface as set of points where $f(x) = c$, allows representing complex geom *implicitly*. $f(x)$ is **signed dist func (SDF)**, which gives dist from point to surface. Sign indicates whether point is inside/outside surface. **Constructive solid geom** represents surfaces by boolean operations on primitives. **Neural net** represents a continuous func of entire scene.

- **Gaussians:** less primitives than voxels; better view-dep than point clouds; fast splat rendering (> ray tracing).
- **Voxel grids:** regular voxel grid; easy to implement, $O(1)$ access, but high memory requirements & limited by Nyquist.
- **NeRF:** continuous scene fn; captures fine detail + view-dep; slow train/render (complex scenes).
- **Hash Grids:** sparse voxel grid with hash-based indexing; fast access, reduced memory; good for large scenes, but may struggle with fine details.

13.4 NeRFs

Start with volumetric cloud, & set of posed images. To represent vol, use **Neural Radiance Field (NeRF)**: a fully connected NN that takes in 3D pos & view dir, outputs density emitted radiance at that point. **Positional Encoding** used to help net learn high-freq details.

13.5 Gaussian Splatting

Primitive Based repr use rasterization. **Point clouds** (*unstructured geom*) have **multi-chart manifold** (represents complex surfs w/o needing single continuous parameterization). Independent to permutations, so can sort points in screen space. Its **view-dependent**.

13.6 Surface Splatting

Consider **oriented points** (*surfels*) as discrete samples of a texture function on a surface. Gaussian reconstruction kernel used to recover continuous signal. This is then sampled in screen space.

- Points scaled with cam dist so objects have no holes.
- Slanted normals appear as ellipses, so we can create good edges.
- Each sample can be processed in parallel.

13.7 Volume Splatting

Instead, use **oriented ellipsoids** as primitives, representing volumetric data. Allows for better handling of complex scenes with varying levels of detail, can capture view-dependent effects more effectively. To blend points in screen space, use **alpha blending** (*differentiable*): giving each point opacity value to create smooth transitions & capture fine details. To render:

1. **Splat**: compute the shape of the Gaussian after projection. The center is projected as before, but the shape's (*covariance matrix*) trans must be approxed using first terms of Taylor Series to ensure affine trans (Gaussians closed under affine trans).
2. **Sort**: globally sort the points by depth.
3. **Blend**: alpha composite.

To optimise cov mat, reparameterize with *rotation* & *scaling* mat, (easier to optimize): $\Sigma = RSS^TR^T$. To splat:

1. Given point cloud $P = \{P_1, \cdots, P_N\}$, and point Q on surface.
2. Create local parameterization for k neighbours of Q.
3. Each 3D point associated to local 2D coord $Q \to u$, $P_k \to u_k$.
4. Continuous surf func $f_c(u) = \sum_{k \in \mathbb{N}} w_k r_k(u - u_k)$, where w_k is a weight and r_k is a Gaussian kernel.