Graphics

Lectured by Bernhard Kainz and Abhijeet Ghosh

Typed by Aris Zhu Yi Qing

March 15, 2022

Contents

1 Projections and Transformations

1.1 Parallel Projection

- For a vertex $\mathbf{V} = (V_x, V_y, V_z)^T$, the **projector** (projection line) is defined by the parametric line equation

$$\mathbf{P} = \mathbf{V} + \mu\mathbf{d}$$

- Assuming the projection plane is $z = 0$, we can establish

$$0 = P_z = V_z + \mu d_z$$

  to obtain μ, thereby computing P_x and P_y.

- **Orthographic projection** is a special type of parallel projection:
  - projection plane: $z = 0$
  - $\mathbf{d} = \begin{pmatrix} 0 & 0 & -1 \end{pmatrix}^T$
  - $P_x = V_x$, $P_y = V_y$

1.2 Perspective Projections

- The **centre of projection** is the viewpoint, which all the projectors pass through, assumed to be at the origin.
- For a vertex $\mathbf{V} = (V_x, V_y, V_z)^T$, the projector $\mathbf{P}$ has the equation

$$\mathbf{P} = \mu\mathbf{V}$$

- Since the projection plane is at a constant z value f, at the point of intersection we have

$$f = P_z = \mu V_z$$

  to obtain μ, thereby computing P_x and P_y.

1.3 Space Transformations

1.3.1 Homogeneous Coordinates

- A **homogeneous coordinate** is a three-dimensional coordinate with a fourth componenet called **ordinate** which acts as a scale factor.

- Assuming a point $\mathbf{P} = (p_x, p_y, p_z)$ in Cartesian coordinate, we introduce s being the ordinate

$$\mathbf{P}' = (p_x, p_y, p_z, s)$$

  to form a homogeneous coordinate.

- To convert $\mathbf{P}'$ back ot Cartesian, we will perform **perspective division**

$$\mathbf{P}'' = \left(\frac{p_x}{s}, \frac{p_y}{s}, \frac{p_z}{s}\right),$$

  i.e. divide x, y and z values by the ordinate. Thus when $s = 1$, $\mathbf{P} = \mathbf{P}''$.

- If $s \neq 0$, we have a **position vector**. If $s = 0$, we have a **direction vector**.

1.3.2 Translation Matrix

To apply a translation vector $\mathbf{t} = (t_x, t_y, t_z)$ to a point $\mathbf{P} = (p_x, p_y, p_z)$, we do

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

with the inverse of the translation matrix as

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

1.3.3 Scaling Matrix

To scale a point from the origin, we can do

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

with the inverse of the scaling matrix as

$$\begin{pmatrix}1/s_x & 0 & 0 & 0\\ 0 & 1/s_y & 0 & 0\\ 0 & 0 & 1/s_z & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

1.3.4 Rotation Matrix

To rotate anti-clockwise when looking along the direction of the axis with a left-hand axis system, we have

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

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

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

to rotate along x, y and z axis respectively. In other words, the right matrices are rotating clockwise.

1.3.5 Projection Matrix

For a perspective projection, placing the centre of projection at the origin and using $z = f$ as before, we can use

$$\mathcal{M}_p = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 1/f & 0\end{pmatrix}$$

For an orthographic projection, with the projection plane at $z = 0$, we can use

$$\mathcal{M}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2 Clipping

- **Clipping** eliminates portions of objects outside the **viewing frustum**, which is the boundaries of the image plane projected in 3D with a near and far clipping plane.
- Why clipping?
  - avoid degeneracy: e.g. don't draw objects behind the camera
  - improve efficiency: e.g. do not process objects which are not visible.
- When to clip?
  - before perspective transform in 3D space:
    * 3D world space
    * use the equation of 6 planes
    * natural, not too degenerate
  - in homogeneous coordinates after perspective transform and before perspective division:
    * clip space
    * canonical, independent of camera
    * simplest to implement, since clipping plane can align with axis so that we can easily discard anything further than the far plane or closer than the near plane
  - in the transformed 3D screen space after perspective division:
    * Normalized Device Coordinates (NDC)
    * The regions extends from -1. to 1. in each axis. Anything outside from the volume is discarded.
    * problem — having negative orginates
- **Halfspace** We can define any plane as a test for a point $\mathbf{p}$:

$$f(x, y, z) = \mathbf{H} \cdot \mathbf{p} = 0$$

  where $\mathbf{H} = (H_x, H_y, H_z, H_s)$ and $\mathbf{p} = (x, y, z, 1)$, such that

$$\begin{cases} \mathbf{H} \cdot \mathbf{p} > 0 & \text{in one halfspace (pass-through)} \\ \mathbf{H} \cdot \mathbf{p} < 0 & \text{in the other halfspace (clip/cull/reject)} \end{cases}$$

  - **Segment Clipping** Similarly we have

$$\begin{cases} \mathbf{H} \cdot \mathbf{p} > 0, \mathbf{H} \cdot \mathbf{q} < 0 & \text{clip } \mathbf{q} \text{ to plane} \\ \mathbf{H} \cdot \mathbf{p} < 0, \mathbf{H} \cdot \mathbf{q} > 0 & \text{clip } \mathbf{p} \text{ to plane} \\ \mathbf{H} \cdot \mathbf{p} > 0, \mathbf{H} \cdot \mathbf{q} > 0 & \text{pass through} \\ \mathbf{H} \cdot \mathbf{p} < 0, \mathbf{H} \cdot \mathbf{q} < 0 & \text{clipped out} \end{cases}$$

  - Test if an object is convex.
    1. For each face of the object, pick a random point.
    2. For this point, compare with points from other faces, check if
       ```
       sign(f(xj,yj,zj)) != sign(f(xi,yi,zi))
       ```
       then it is not convex.
  - Test if a point is contained in a concave object.
    * Cast a ray from the test point in any direction. If the number of intersections with the object is odd, then the test point is inside.

3 Graphics Pipeline

3.1 Application

- executed by the software on the main processor (CPU)
- typical tasks performed: collision detection, animation, morphing, perform spatial subdivision scheme (quadtree, octree).
- to reduce the amount of main memory required at a given time

3.2 Geometry

1. Modelling Transformations
2. Illumination (Shading)
3. Viewing Transformations (Perspective/Orthographic)
4. Clipping
5. Projection (to screen space — window-viewport transformation)

3.3 Rasterization

- **Rasterization** is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (a series of pixels).
- During this process, fragments/raster points are created from continuous primitives. A **fragment** can be thought of as the data needed to shade the pixel (e.g. color, illumination, texture) and to test whether the fragment survives to become a pixel (depth, alpha, etc.)
- Eventually, one or more fragments are merged to become a **pixel**, which is the smallest addressable element in a raster image.
- To prevent from exposing the process of gradual screening of the primitives, double buffering is used so that the rasterization takes place in a special memory, and as soon as the image is completely rastered, it is copied into the visible area of the image memory (frame buffer).

3.4 Shading

Shading refers to the modification of individual vertices or fragments within the graphics pipeline. This is the *programmable* part of the graphics pipeline.

3.4.1 Vertex Shader

- executed once for each vertex
- only has access to the vertex and no neighbouring vertices, the topology, or similar

3.4.2 Tessellation Shader

- divides an area (triangle or square) into smaller areas
- advantage: allow detail to be dynamically added and subtracted from a 3D polygon mesh and its silhouette edges based on control parameters (e.g. camera distance)
- The *Tessellation Control Shader* (TCS) determines how much tessellation to do. It is optional; default tessellation values can be used.
- The *tessellation primitive generator* (not programmable) takes the input patch and subdivides it based on values computed by the TCS.
- The *Tessellation Evaluation Shader* (TES) takes the tessellated patch and computes the vertex values for each generated vertex.

3.4.3 Geometry Shader

- takes a single primitive as input and may output zero or more primitives of the same type
- has access to multiple vertices, if the primitive consists of multiple vertices
- A **primitive** can mean
  (a) the interpretation scheme to determine what a stream of vertices represents when being rendered, which can be arbitrarily long
  (b) the *result* of the interpretation of a vertex stream (also called the *base primitive*)
- Use case: in a particle system, the inputs are processed points, and geometry shader generates polygons/cubes/etc. to save computation in the previous pipelines.

3.4.4 Fragment Shader

- executed once for every fragment generated by the rasterization
- it takes in interpolated vertex attributes
- it calculates the color of the corresponding fragment

4 OpenGL

- The interface is platform independent, but the implementation is platform dependent.
- It defines an abstract rendering device and a set of functions to operate the device.
- It is a low-level "immediate mode" graphics API with drawing commands and no concept of permanent objects, operating as a state machine.
- To write an OpenGL programme, we need to
  1. create a render window via library such as glut, Qt, etc.
  2. setup viewport, model transformation and file I/O (shader, textures, etc.)
  3. implement frame-generataion (update/rendering) functions to define what happens in every frame
- Basic concepts:
  - **Context**
    - represents an instance of OpenGL
    - a process can have multiple contexts to share resources
    - one-to-one mapping between a context and a thread
  - **Resources**
    - act as sources of input and sinks for output
    - e.g. texture images(input), buffers(output)
  - **Object Model**
    - Object instances are identified by unique names (unsigned integer handle).
    - Commands work on targets, where each target is bounded by an object.
- **Buffer objects** are regular OpenGL objects that store an array of unformatted memory allocated by the OpenGL context (i.e. GPU).
- has primitive types such as `GL_POINTS`, `GL_LINES`, `GL_POLYGONS`, `GL_TRIANGLES`, etc.

5 Illumination and Shading

5.1 Physics of Shading

- object properties
  - the position of the object relative to the light sources
  - the surface normal vector
  - the albedo of the surface (ability to absorb light energy) and the reflectivity of the surface
- light source properties
  - intensity of the emitted light
  - distance to the point on the surface
- energy (Joule) of a photon is

$$e(\lambda) = \frac{hc}{\lambda}$$

  where $h \approx 6.63 \times 10^{-34} J \cdot s$ and $c \approx 3 \times 10^8 m/s$.
- radiant energy (Joule) of n photons is

$$Q = \sum_{i=1}^{n} e(\lambda_i)$$

- Radiation/radiant/electromagnetic flux (Watts) is

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t}$$

- **Radiance** (Watt/(meter^2 · steradian)) is density of a incident flux falling onto a surface in a particular direction

$$L(\omega) = \frac{\mathrm{d}^2\Phi}{\cos\theta \, \mathrm{d}A\mathrm{d}\omega}$$

- **Irradiance** (Watt/meter^2) is density of the incident flux falling onto a surface

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A}$$

- We define the **Bidirectional Reflectance Distribution Function** (BRDF) (1/steradian)

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\omega_i, \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)}$$

by ignoring other physical phenomenon such as absorption, transmission, fluorescence, diffraction, etc.

  - *Isotropic BRDF* is such that rotation along surface normal does not change reflectance.
  - *Anisotropic BRDF* changes reflectance when rotating along surface normal, which happens on surfaces with strongly oriented microgeometry elements such as brushed metals, hair, cloth, etc.
  - non-negativity: $f_r(\omega_i, \omega_r) \geq 0$
  - energy conservation: $\forall \omega_i, \int_\Omega f_r(\omega_i, \omega_r) \cos\theta_r \mathrm{d}\omega_r \leq 1$
  - reciprocity: $f_r(\omega_i, \omega_r) = f_r(\omega_r, \omega_i)$

- To compute the reflected radiance discretely, with n points light sources, we have

$$L_r(\omega_r) = \sum_{i=1}^{n} f_r(\omega_i, \omega_r) E_i = \sum_{i=1}^{n} f_r(\omega_i, \omega_r) \cos\theta_i \frac{\Phi_i}{4\pi d_i^2}$$

- Ideally, BRDF is constant, so with a single point light source

$$L(\omega_r) = k_d (n \cdot l) \frac{\Phi_s}{4\pi d^2}$$

where k_d is the diffuse reflection coefficient, n is the (normalized) surface normal, and l is the (normalized) light direction from surface.

5.2 The Phong Model

- light sources are assumed to be point-shaped, i.e. no spatial extent
- Reflected radiance calculation is

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

where k_s is the specular reflection coefficient, q is the specular reflection exponent, v is the direction vector from surface to camera, and r is the reflected ray.

- Blinn-Phong variation is that

$$L(\omega_r) = k_s (n \cdot h)^q \frac{\Phi_s}{4\pi d^2} \qquad \text{with} \qquad h = \frac{l + v}{\|l + v\|}$$

- The Phong model is the sum of three components: diffuse, specular and ambient, i.e.

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

- Sometimes using $(d + s)$ instead of d^2 produces better result, where s is a heuristic constant.

5.3 Shading

5.3.1 Flat Shading

- each polygon is shaded uniformly over its surface
- computed by taking a point in the center and at the surface normal
- normally only the diffuse and ambient components are used

5.3.2 Gouraud Shading

- interpolate color using shade value at each vertex
- can interpolate intensity at each vertex from all the polygons that meet at that vertex to create the impression of a smooth surface
- cannot accurately model specular components, since we don't have normal vector at each point on a polygon

5.3.3 Phong Shading

- interpolate normals across triangles at fragment stage
- more accurate modelling of specular components, but slower

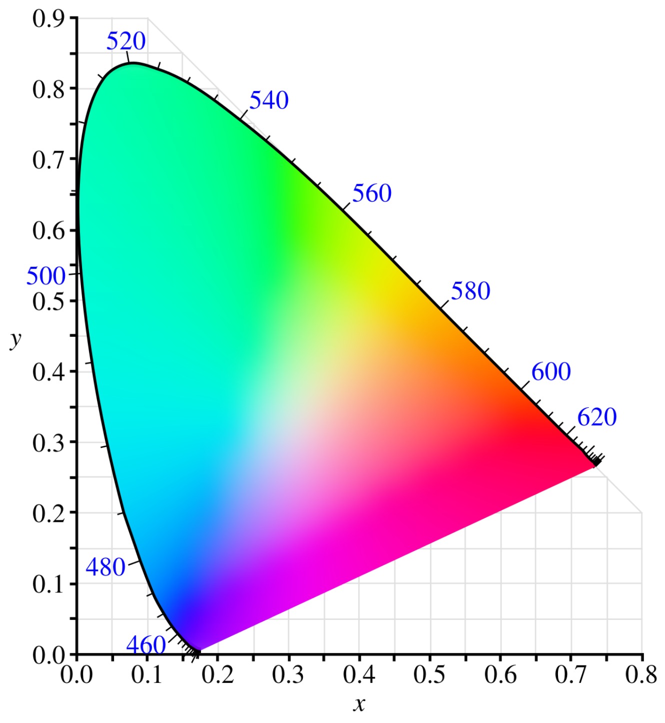


Figure 1: The CIE 1931 color space. Red is at $(0.628, 0.346, 0.026)$, Green is at $(0.268, 0.588, 0.144)$, Blue is at $(0.150, 0.07, 0.780)$

6 Color

6.1 RGB CIE Color Space

- a standard normalized representation of colors ranging from 0 to 1, with

$$x = \frac{r}{r+g+b}, y = \frac{g}{r+g+b}, z = \frac{b}{r+g+b} = 1 - x - y$$

- the actual visible colors are a subset of this as shown in Figure 1, done through manual testing

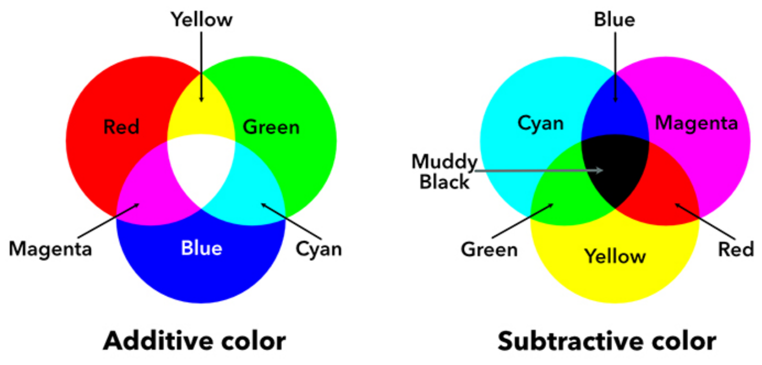
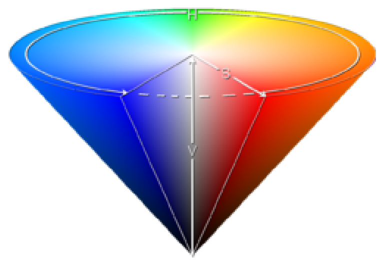


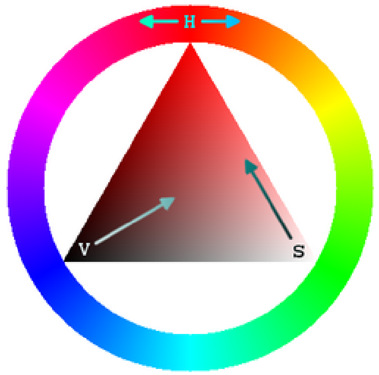
Figure 2: additive and subtractive primaries

- the shape must be convex, since any blend (interpolation) of pure colors should create a color in the visible region.
- the **pure colors** are around the edge of the diagram, also called **fully saturated**
- the line joining purple and red has no pure equivalent; the colours can only be created by blending
- **Saturation** of an arbitrary point is the ratio of its distance to the white point over the distance of the white point to the edge.
- white point: when $x = y = z = 0.\dot{3}$
- The **complement color** of a color is the point diametrically opposite through the white point. Computationally, if the color has value (r, g, b), its complement color is $(255 - r, 255 - g, 255 - b)$.
- The **additive primaries** are RGB (Red, Green, Blue) and the **subtractive primaries** are CMY (Cyan, Magenta, Yellow). Red is the complement color of Cyan, and similarly for Green and Blue, as shown in Figure 2.
- RGB can be converted to CIE by

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0.628 & 0.268 & 0.15 \\ 0.346 & 0.588 & 0.07 \\ 0.026 & 0.144 & 0.78 \end{pmatrix} \begin{pmatrix} r \\ g \\ b \end{pmatrix}$$



(a) The conical representation of the HSV model



(b) The HSV color wheel

Figure 3: HSV model

6.2 HSV Color Representation

- **Hue** corresponds notionally to pure color
- **Saturation** is the proportion of pure color
- **Value** is the brightness/intensity
- We can visualize the perceptual color space in HSV as in Figure 3.
- Conversion between RGB and HSV can be done as

$$V = \max(r, g, b)$$
$$S = \frac{\max(r, g, b) - \min(r, g, b)}{\max(r, g, b)}$$
$$H = \begin{cases} \text{undefined} & r = g = b \\ 120 \cdot \dfrac{g-b}{(r-b)+(g-b)} & (r > b) \wedge (g > b) \\ 120 + 120 \cdot \dfrac{b-r}{(g-r)+(b-r)} & (g > r) \wedge (b > r) \\ 240 + 120 \cdot \dfrac{r-g}{(r-g)+(b-g)} & (r > g) \wedge (b > g) \end{cases}$$

6.3 Transparency

We can model transparency with an α channel, with

- transparent: $\alpha = 0$
- semi-transparent: $0 < \alpha < 1$
- opaque: $\alpha = 1$

Suppose that we put A over B over background G,

- How much of B is blocked by A? α_A
- How much of B shows through A? $(1 - \alpha_A)$
- How much of G shows through both A and B? $(1 - \alpha_A)(1 - \alpha_B)$
- How much does G contribute to the overall color? $(1-\alpha_A)(1-\alpha_B)\alpha_G$

7 Texture

7.1 Definition

- **Texture (map)** is an image applied (mapped) to the surface of a shape or polygon.
- A texture can be 1D, 2D, or 3D, but 2D is the most common for visible surfaces.
- **Raster images** are 2D rectangular matrices or grid of square pixels, often used as textures.
- **Procedural texture** is a texture created using a mathematical description rather than directly stored data (e.g. raster image). Mathematically, it is a function f defiend as

  $$f : \mathbf{p} \mapsto \text{color},$$

  where $\mathbf{p}$ is a coordinate.

  + Very small memory footprint before the texture map is generated. The ultimate way in image compression.

+ No texture memory is really needed since generated “on the fly” in a fragment shader, resulting in the exactly right level of detail for each pixel on the screen.
- Hard to get a formula to get the exact/natural look.
- On-the-fly generation can take a lot of shader program instructions, almost always slower than just loading/looking up one.

7.2 Photo Textures Mapping

7.2.1 Mechanism

- Define a 2D coordinate system on an image mapped onto a 3D object.
- For each fragment on an object’s surface, work out what coordinate needs to be sampled in the image’s 2D space to get the right color.
- Conventionally, texture coordinates are denoted with (s, t) (*texture space*). Canonically it goes from (0,0) to (0,1). The object surface is denoted with (u, v) (*object space*) and the pixel on the screen is denoted with (x, y) (*screen space*). We need to define

  $$\underline{\textbf{Parameterization}} : (s, t) \mapsto (u, v)$$

  which is the process of finding parametric equations of textures and objects so that texture can be mapped onto object surface, and

  $$\underline{\textbf{Rendering}} : (u, v) \mapsto (x, y)$$

  which is the process of generating an image from a model.

7.2.2 Parameterization

- Planar mapping: ignore one of the coordinates
- Cylindrical/Spherical mapping: compare to cylindrical/spherical coordinate systems
- Box mapping: 6 planar mapping
- **Unwrapping**: the process of creating manual mapping

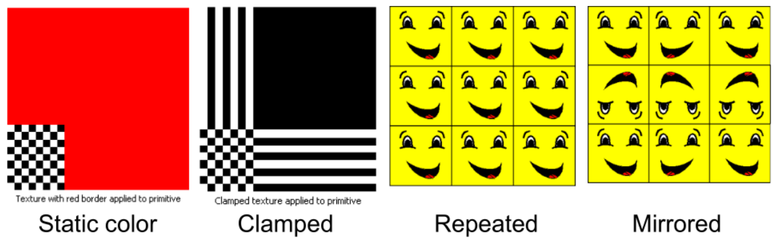

Texture with red border applied to primitive
Static color
Clamped texture applied to primitive
Clamped
Repeated
Mirrored

Figure 4: Different texture addressing

7.2.3 Texture Addressing (Mode)

What happens outside $[0, 1]$? Following the order in Figure 4,

- static color: we use a static color (red in this case)
- clamped: we use the last color in the range
- repeated: wrap back to the first coordinate, repeating the texture
- mirrored: similar to repeated, but go backwards insteads after ending the texture

7.3 Perspective Correct Interpolation

- cannot simply perform linear interpolation as in Gouraud shading on texture coordinates
- The problem:
  – perspective projection **does not preserve** linear combinations of points
  – e.g. equal distances in 3D do not map to equal distances in screen space, as shown in figure 5.
- The solution:
  – Assume image plane is at $z = f = 1$.
  – Let t controls linear blend of texture coordinates of $\mathbf{p}$ and $\mathbf{r}$ and let

  $$t_p = 0, \ t_r = 1.$$

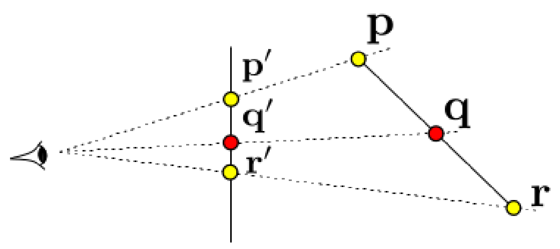


Figure 5: perspective interpolation error

- Let lerp(x, y) be the linear interpolation function, and as such

$$t_q = t_{q'} z_q = \frac{\text{lerp}(t_{p'}, t_{r'})}{\frac{1}{z_q}} = \frac{\text{lerp}(\frac{t_p}{z_p}, \frac{t_r}{z_r})}{\text{lerp}(\frac{1}{z_p}, \frac{1}{z_r})}$$

- The algorithm: given texture parameter t at vertices,
  1. compute $\frac{1}{z}$ for each vertex
  2. linearly interpolate $\frac{1}{z}$ across the triangle
  3. linearly interpolate $\frac{t}{z}$ across the triangle
  4. perform perspective division via dividing $\frac{t}{z}$ by $\frac{1}{z}$ to obtain t

7.4 Texture Mapping and Illumination

- For texture-mapped object, changing the lighting will not show the unevenness of the object's surface.
- **Bump mapping**: textures to alter the surface normal of an object. Shading on the object is changed, but its silhouette is not.
- **Displacement mapping**: textures to change the shading of both the object and its silhouette. It actually moves the surface point — geometry is displaced before determining visibility.
- **Environment mapping**: we can simulate reflections by using the direction of the reflected ray to index a spherical texture map at "infinity".

8 Rasterization, Visibility, and Anti-aliasing

8.1 Background

- **Rasterization** determines which pixels are drawn into the frame-buffer.
- Pixels have unique framebuffer location, but multiple fragments can be at the same address.
- **Alias effects**: an unreal visual artefacts caused by undersampling e.g. straight lines look jagged.

8.2 Barycentric Coordinates

- let vertices of a triangle be **a**, **b**, and **c**, then any point **p** can be specified in the plane as

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

  and (α, β, γ) is called (absolute) barycentric coordinates.

- If **p** is inside the triangle $(\mathbf{a}, \mathbf{b}, \mathbf{c})$, we have $0 < \alpha, \beta, \gamma < 1$; if **p** is on an edge, one coefficient is 0; if **p** is on a vertex, one coefficient is 1.
- Since implicit equation in 2D is defined as

$$f(x, y) = Ax + By + c = 0$$

  and an implicit line through (x_a, y_a) and (x_b, y_b) is

$$f_{ab}(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0,$$

  note that a barycentric coordinate such as β is a **signed distance** from a line (in this case, the line through **a** and **c**), and we can use implicit line equations to evaluate the signed distance, we have

$$\frac{f_{ac}(x_b, y_b)}{f_{ac}(x, y)} = \frac{1}{\beta} \Longrightarrow \beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}$$

  and we can obtain similar results for α and γ, thereby computing the barycentric coordinates with the given cartesian coordinates.

- In general, the barycentric coordinates for $\mathbf{p}$ are the solution of the linear system
$$\begin{pmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- Let a, b, c be the side lengths of the triangle, from barycentric to trilinear coordinates:
$$(\alpha, \beta, \gamma) \mapsto (\frac{\alpha}{a}, \frac{\beta}{b}, \frac{\gamma}{c}),$$
from trilinear to barycentric coordinates:
$$(t_1, t_2, t_3) \mapsto (t_1 a, t_2 b, t_3 c)$$

- **Triangle Rasterization**: we can check if $(0 < \alpha, \beta, \gamma < 1)$ to determine if a fragment of a triangle should be generated.

8.3 Visibility

- **Painter's algorithm**: sort the triangles using the z values in camera space and draw them from back to front.
  - - not efficient due to costly sorting
  - - suffer from correctness issues such as intersections, cycles, etc.

- **depth buffer (z-buffer)**: perform hidden surface removal per-fragment by saving the z value for each fragment and keep only the fragment with the smallest z value at each pixel. Use *z-buffer (2D buffer)* of the same size as image to save z values.
  - + facilitates hardware implementation
  - + handles intersections and cycles
  - + draw opaque polygons in any order

8.4 Anti-aliasing

- apply a degree of blurring to the boundary such that the aliasing effect is reduced.

- **Supersampling**
  1. Compute the picture at a high resolution to that of the display area.
  2. Supersamples are averaged to find the pixel value.
  3. This blurs the boundaries and leaves the coherent areas of color unchanged.

- **Convolution filtering**: use a filter that takes a (weighted) average over a small region around the pixel to blur the image, such as
$$\frac{1}{36}\begin{pmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{pmatrix}$$
  - + very fast, can be done in hardware
  - + generally applicable
  - - degrade the image while enhancing its visual appearance

9 Ray Tracing

9.1 Introduction

- **Direct illumination**: a surface point receives light directly from all light sources in the scene.

- **Global illumination**: a surface point receives light after the light rays interact with other objects in the scene.
  - – points may be in shadow
  - – rays may refract through transparent material
  - – computed by reflection and transmission rays

- Ray tracing pros and cons
  - + Easy to implement
  - + extends well to global illumination, such as shadows, reflections/refractions, etc.
  - - speed — seconds/frame, instead of frames/second

9.2 Basic Ray Tracing Algorithm

1. cast one ray per pixel
2. compute nearest intersection with an object in the scene
3. check if the ray from the intersection point to the light source is occluded
    - if occlusion happens, stop tracing
    - if no occlusion, compute color at the intersection point, update ray to be the next ray to be traced, and go back to step 2.

9.3 Mathematical Computations

9.3.1 Reflected Ray

Givne that $\mathbf{v}$ is the primary ray, $\mathbf{n}$ is the unit surface normal, we can compute the reflected ray $\mathbf{v}'$ as

$$\mathbf{v}' = \mathbf{v} - (2\mathbf{v}\cdot\mathbf{n})\mathbf{n}.$$

9.3.2 Color computation

We can compute color/shadow as

$$L = k_a + s\left[k_d(\mathbf{n}\times\mathbf{l}) + k_s(\mathbf{v}\times\mathbf{r})^q\right] I_s + k_{\text{reflected}}L_{\text{reflected}} + k_{\text{refracted}}L_{\text{refracted}}$$

where

$$s = \begin{cases} 0 & \text{if light source is obscured} \\ 1 & \text{if light souce is not obscured.} \end{cases}$$

We can use multiple shadow rays to sample an area light source, allowing us to create *soft shadows*.

9.3.3 Refraction

- The angle of the refracted ray can be determined by the Snell's law

  $$\eta_1 \sin\phi_1 = \eta_2 \sin\phi_2$$

  where

  - η_1 is the refractive index for medium 1
  - η_2 is the refractive index for medium 2
  - ϕ_1 is the angle between the incident ray and the surface normal
  - ϕ_2 is the angle between the refracted ray and the surface normal

- In vector notation Snell's law can be written as

  $$\eta_1(\mathbf{v}\cdot\mathbf{n}) = \eta_2(\mathbf{v}'\cdot\mathbf{n})$$

  so that the direction of the refracted ray can be calculated as

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

- The equation only has a solution if

  $$(\mathbf{n}\cdot\mathbf{v})^2 > 1 - \left(\frac{\eta_2}{\eta_1}\right)^2$$

  illustrating the physical phenomenon of the limiting angle (possibility of <u>*total internal reflection*</u>).

9.3.4 Amount of reflection and refraction

To more accurately model the phenomenon of more reflection at grazing angle, use the Fresnel factor

$$L_{\text{secondary}} = k_{\text{fresnel}}L_{\text{reflected}} + (1 - k_{\text{fresnel}})L_{\text{refracted}}$$

with Schlick's approximation

$$k_{\text{fresnel}}(\theta) = k_{\text{fresnel}}(0) + [1 - k_{\text{fresnel}}(0)][1 - (\mathbf{n}\cdot\mathbf{l})]^5$$

where $k_{\text{fresnel}}(0)$ is the Fresnel factor at zero degrees, and choosing $k_{\text{fresnel}}(0) = 0.8$ make the object's material looks like stainless steel.

9.3.5 Intersection Calculation

Given that each primary ray could be expressed with

$$\mathbf{p} = \mathbf{p}_0 + \mu \mathbf{d}$$

where

- $\mathbf{p}_0$ is the origin of the ray
- $\mathbf{d}$ is the direction vector
- $\underline{\mu > 0}$ indicates that $\mathbf{p}$ is visible (in front of the viewing plane),

we can calculate intersections with

- *sphere*
  $$\Longrightarrow \|\mathbf{p}_0 + \mu \mathbf{d} - \mathbf{p}_s\|^2 - \|\mathbf{r}\|^2 = 0,$$
  where $\mathbf{p}_s$ is the center of the sphere, $\mathbf{r}$ is an arbitrary radius vector.

  We can derive
  $$\mu = -\mathbf{d} \cdot \Delta\mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \Delta\mathbf{p})^2 - \|\Delta\mathbf{p}\|^2 + \|\mathbf{r}\|^2}$$
  where $\Delta\mathbf{p} = \mathbf{p}_0 - \mathbf{p}_s$. Assuming there are two solutions with $\mu_1 < \mu_2$ (the ray intersects the sphere),
  - μ_1 corresponds to the point at which the ray enters the sphere
  - μ_2 corresponds to the point at which the ray leaves the sphere.
- *cylinder*
  $$\Longrightarrow \mathbf{p}_1 + \alpha \Delta\mathbf{p} + \mathbf{q} = \mathbf{p}_0 + \mu \mathbf{d}$$
  where $\mathbf{p}_1/\mathbf{p}_2$ is one/the other end point of the long axis of the cylinder, $\Delta\mathbf{p} = \mathbf{p}_1 - \mathbf{p}_2$ is the vector along the long axis of the cylinder (thus $0 < \alpha < 1$), $\mathbf{q}$ is a random radius vector.

  We can dot both sides by $\Delta\mathbf{p}$ to solve for α as
  $$\alpha = \frac{\mathbf{p}_0 \cdot \Delta\mathbf{p} + \mu \mathbf{d} \cdot \Delta\mathbf{p} - \mathbf{p}_1 \cdot \Delta\mathbf{p}}{\Delta\mathbf{p} \cdot \Delta\mathbf{p}}$$
  and obtain expression for $\mathbf{q}$ as
  $$\mathbf{q} = \mathbf{p}_0 + \mu \mathbf{d} - \mathbf{p}_1 - \left(\frac{\mathbf{p}_0 \cdot \Delta\mathbf{p} + \mu \mathbf{d} \cdot \Delta\mathbf{p} - \mathbf{p}_1 \cdot \Delta\mathbf{p}}{\Delta\mathbf{p} \cdot \Delta\mathbf{p}} \right) \Delta\mathbf{p}$$
  by dotting both sides of the above equation by itself, we can solve the quadratic equation for μ
  $$\|\mathbf{q}\|^2 = r^2 = \left[\mathbf{p}_0 + \mu \mathbf{d} - \mathbf{p}_1 - \left(\frac{\mathbf{p}_0 \cdot \Delta\mathbf{p} + \mu \mathbf{d} \cdot \Delta\mathbf{p} - \mathbf{p}_1 \cdot \Delta\mathbf{p}}{\Delta\mathbf{p} \cdot \Delta\mathbf{p}} \right) \Delta\mathbf{p} \right]^2$$
  where r is the radius of the cylinder. Assuming $\exists \mu_1, \mu_2$ and $\mu_1 < \mu_2$, we can compute
  $$\alpha_1 = \frac{\mathbf{p}_0 \cdot \Delta\mathbf{p} + \mu_1 \mathbf{d} \cdot \Delta\mathbf{p} - \mathbf{p}_1 \cdot \Delta\mathbf{p}}{\Delta\mathbf{p} \cdot \Delta\mathbf{p}}$$
  to determine which part of the cylinder the intersection is on.
- *plane*
  $$\Longrightarrow \mathbf{p}_1 + \mathbf{q} = \mathbf{p}_0 + \mu \mathbf{d}$$
  where $j\mathbf{p}_1$ is a point in plane, $\mathbf{q}$ is a vector on the plane.

  Subtracting $\mathbf{p}_1$ and dot both sides by $\mathbf{n}$, the normal of the plane, we can solve for μ yielding
  $$\mu = \frac{(\mathbf{p}_1 - \mathbf{p}_0) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$
- *triangles* — parameterized by vertices $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$, we need to test
  - front-facing: $\mathbf{d} \cdot \mathbf{n} < 0$
  - plane of triangle intersects ray: same as before
  - intersection point is inside triangle: using the concept of barycentric coordinates, we can formulate the point as
    $$\mathbf{q} = \alpha \mathbf{a} + \beta \mathbf{b}$$
    with
    $$\alpha = \frac{(\mathbf{b} \cdot \mathbf{b})(\mathbf{q} \cdot \mathbf{a}) - (\mathbf{a} \cdot \mathbf{b})(\mathbf{q} \cdot \mathbf{b})}{(\mathbf{a} \cdot \mathbf{a})(\mathbf{b} \cdot \mathbf{b}) - (\mathbf{a} \cdot \mathbf{b})^2}$$
    $$\beta = \frac{\mathbf{q} \cdot \mathbf{b} - \alpha(\mathbf{a} \cdot \mathbf{b})}{\mathbf{b} \cdot \mathbf{b}}$$
    and test if all of below hold
    * $0 \le \alpha \le 1$
    * $0 \le \beta \le 1$
    * $\alpha + \beta \le 1$

9.4 Engineering Problems

9.4.1 Precision Problem

- It is possible that due to precision error, the intersection point resides *within* the object instead of on/above the surface of the object.
- This means the object is reflecting within itself, resulting in erroneous shadow calculation.
- To prevent this from happening, simply add an empirical ϵ tolerance to correct the intersection position so that it is guaranteed to be on/above the surface of the object.

9.4.2 Randomness Simulation

- **Monte-Carlo Ray Tracing**: cast *one* primary ray and compute *multiple* secondary rays with some randomness in their direction.
  - very costly since the amount of computation grows exponentially
- **Monte-Carlo Path Tracing**: cast *multiple* primary rays and compute *one* secondary ray for each primary ray with some randomness in its direction.

9.4.3 Acceleration of Ray Tracing

- **Bounding regions**
- **Regular grid**
  - + easy to construct
  - + easy to traverse
  - - may be only sparsely filled
  - - geometry may still be clumped
- **Adaptive grid**
  - + grid complexity matches geometric density
  - - more expensive to traverse
- **Binary Space Partition(BSP) tree**
  - + enable simple front-to-back traversal
  - + solves the problem of painter's algorithm by
    - ∗ providing a rapid method of sorting polygons w.r.t. given viewport
    - ∗ subdividing overlapping polygons to avoid errors
  - - construction of BSP tree is expensive

10 Spline Curves

10.1 Definition

- *Interpolating* spline: defined by a start point, an end point, and a small set of points called **knots** or **control points** that they go through smoothly
- *Approximating* spline: same points but do *not* need to pass through them except for start and end points
- Non-parametric spline has only one curve that fits all the data.

10.2 Spline Parameterization

- spline in vector form in 2D is
$$\mathbf{P} = \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$
with a parameter μ and three vector constants $\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2 \in \mathbb{R}^2$.
- Assuming we have three points:
  - $\mathbf{P}_0$: at the start point, i.e. $\mu = 0$
  - $\mathbf{P}_1$: somewhere in the middle, i.e. $\mu = \mu_1$
  - $\mathbf{P}_2$: at the end point, i.e.$\mu = 1$

  we can obtain the following system
$$\begin{cases} \mathbf{P}_0 = \mathbf{a}_0 \\ \mathbf{P}_1 = \mathbf{a}_2\mu_1^2 + \mathbf{a}_1\mu_1 + \mathbf{a}_0 \\ \mathbf{P}_2 = \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{P}_0 \end{cases}$$
  and solve for the three vector constants.

10.3 Spline Patches

- The simpliest and most effective way to calculate spline patches is to use a cubic polynomial

$$\mathbf{P} = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$

with

$$\mathbf{P}' = 3\mathbf{a}_3\mu^2 + 2\mathbf{a}_2\mu + \mathbf{a}_1$$

so as to join the patches smoothly

- By using the value of $\mathbf{P}_{i-1}$, $\mathbf{P}_i$, $\mathbf{P}_{i+1}$, $\mathbf{P}_{i+2}$, we can obtain the following system with the spline patch between $\mathbf{P}_i$ and $\mathbf{P}_{i+1}$

$$\begin{aligned}
\mathbf{P}_i &= \mathbf{a}_0 \\
\mathbf{P}_{i+1} &= \mathbf{a}_3 + \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{a}_0 \\
\mathbf{P}'_i &= \mathbf{a}_1 = \frac{1}{2}(\mathbf{P}_{i+1} - \mathbf{P}_{i-1}) \\
\mathbf{P}'_{i+1} &= 3\mathbf{a}_3 + 2\mathbf{a}_2 + \mathbf{a}_1 = \frac{1}{2}(\mathbf{P}_{i+2} - \mathbf{P}_i)
\end{aligned}$$

and in matrix form, we have

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_i \\ \mathbf{P}'_i \\ \mathbf{P}_{i+1} \\ \mathbf{P}'_{i+1} \end{pmatrix}$$

therefore solving the four vector constants as

$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_i \\ \mathbf{P}'_i \\ \mathbf{P}_{i+1} \\ \mathbf{P}'_{i+1} \end{pmatrix}$$

10.4 Smoothness

10.4.1 Parametric Continuity

- C^0 continuous — (x, y) values of the curve agree
- C^1 continuous — (x, y) and first order derivatives $\left(\frac{\mathrm{d}x}{\mathrm{d}s}, \frac{\mathrm{d}y}{\mathrm{d}s}\right)$ agree
- C^2 continuous — (x, y), first and second order derivatives agree

$\vdots$

10.4.2 Geometric Continuity

- G^0 continuous — same as C^0
- G^1 continuous — G^0 and proportional first order derivatives
- G^2 continuous — G^1 and proportional second order derivatives

$\vdots$

10.5 Bezier Curve

For $N + 1$ control points, we have

$$\mathbf{P}(\mu) = \sum_{i=0}^{N} W(N, i, \mu)\mathbf{P}_i$$

where

$$W(N, i, \mu) = \binom{N}{i} \mu^i (1 - \mu)^{N-i}$$

is the Berstein blending function. Specially, we can show that

4-point Bezier curve = Cubic patch going through $\mathbf{P}_0$ and $\mathbf{P}_3$.

with $\mathbf{P}_1$ and $\mathbf{P}_2$ being the control points, since

$$\begin{aligned}
\mathbf{P}(\mu) &= (1 - \mu)^3\mathbf{P}_0 + 3\mu(1 - \mu)^2\mathbf{P}_1 + 3\mu^2(1 - \mu)\mathbf{P}_2 + \mu^3\mathbf{P}_3 \\
&= \mathbf{a}_0\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{a}_0 &= \mathbf{P}_0 \\
\mathbf{a}_1 &= 3\mathbf{P}_1 - 3\mathbf{P}_0 \\
\mathbf{a}_2 &= 3\mathbf{P}_2 - 6\mathbf{P}_1 + 3\mathbf{P}_0 \\
\mathbf{a}_3 &= \mathbf{P}_3 - 3\mathbf{P}_2 + 3\mathbf{P}_1 - \mathbf{P}_0
\end{aligned}$$

thereby deriving that

$$\mathbf{P}'_0 = 3\mathbf{P}_1 - 3\mathbf{P}_0$$
$$\mathbf{P}'_3 = 3\mathbf{P}_3 - 3\mathbf{P}_2.$$

In short, if you simply want to connect two splines with C^1 continuity, use a spline patch; otherwise if custom control on one spline is required, use Bezier curve.

10.6 Surface Parameterization

- We can extend the formulation to simple parametric surfaces using the vector equation with 6 unknown vector constants

$$\mathbf{P}(\mu,\nu) = \begin{pmatrix}\mu & \nu & 1\end{pmatrix}\begin{pmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c}\\ \mathbf{b} & \mathbf{d} & \mathbf{e}\\ \mathbf{c} & \mathbf{e} & \mathbf{f}\end{pmatrix}\begin{pmatrix}\mu\\ \nu\\ 1\end{pmatrix}$$
$$= \mathbf{a}\mu^2 + \mathbf{d}\nu^2 + 2\mathbf{b}\mu\nu + 2\mathbf{c}\mu + 2\mathbf{e}\nu + \mathbf{f}.$$

  Similarly, if we know 6 points with known μs, we can solve the equation.

- To find the contours that bound the patch, we substitute 0 or 1 for one of μ or ν in the equation, for instance

$$\mathbf{P}(0,\nu) = \mathbf{d}\nu^2 + 2\mathbf{e}\nu + \mathbf{f}.$$

- We can use higher orders, e.g.

$$\mathbf{P}(\mu,\nu) = \begin{pmatrix}\mu^3 & \mu^2 & \mu & 1\end{pmatrix}\begin{pmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d}\\ \mathbf{b} & \mathbf{e} & \mathbf{f} & \mathbf{g}\\ \mathbf{c} & \mathbf{f} & \mathbf{h} & \mathbf{j}\\ \mathbf{d} & \mathbf{g} & \mathbf{j} & \mathbf{k}\end{pmatrix}\begin{pmatrix}\nu^3\\ \nu^2\\ \nu\\ 1\end{pmatrix}$$

  + gives more variety in shape and better control

  - hard to apply and generalize

10.7 Surface Patches

- We need to match three values at each corner:

$$\mathbf{P}(\mu,\nu) \qquad \frac{\partial\mathbf{P}(\mu,\nu)}{\partial\mu} \qquad \frac{\partial\mathbf{P}(\mu,\nu)}{\partial\nu}$$

- We could compute the four boundaries $\mathbf{P}(\mu,0)$, $\mathbf{P}(\mu,1)$, $\mathbf{P}(0,\nu)$, $\mathbf{P}(1,\nu)$ with the four end points and the partial derivatives, using the approach of computing spline patch.

- The **Coons patch** defines the internal points by linear interpoloation of the edge curves as

$$\mathbf{P}(\mu,\nu) = \mathbf{P}(\mu,0)(1-\nu) + \mathbf{P}(\mu,1)\nu + \mathbf{P}(0,\nu)(1-\mu) + \mathbf{P}(1,\nu)\mu$$
$$\underbrace{-\mathbf{P}(0,1)(1-\mu)\nu - \mathbf{P}(1,0)\mu(1-\nu) - \mathbf{P}(0,0)(1-\mu)(1-\nu) - \mathbf{P}(1,1)\mu\nu}_{\text{subtract corners}}$$

- Numerical Ray-Patch Algorithm

  1. Polygonize the patch at a low resolution (say 4×4).
  2. Calculate the ray intersection wtih the (32) triangles and find the nearest intersection.
  3. Polygonize the immediate area of the intersection and calculate a better estimate of the intersection.
  4. Continue until the best estimate is found

- Example question:

$$\begin{array}{cc|cccc}
 & & & y,\nu \longrightarrow & & \\
 & & 3 & 4 & 5 & 6\\
\hline
 & 8 & \cdot & 10 & 9 & \cdot\\
x,\mu & 9 & 14 & 12 & 11 & 10\\
\downarrow & 10 & 15 & 13 & 14 & 10\\
 & 11 & \cdot & 10 & 9 & \cdot
\end{array}$$

with

$$\mathbf{P}(0,0) = (9,4,12),$$
$$\mathbf{P}(0,1) = (9,5,11),$$
$$\mathbf{P}(1,0) = (10,4,13),$$
$$\mathbf{P}(1,1) = (10,4,14),$$

we can determine gradients for all four corners for μ and ν. For instance,

$$\left.\frac{\partial \mathbf{P}}{\partial \mu}\right|_{(0,0)} = \frac{\begin{pmatrix}10\\4\\13\end{pmatrix} - \begin{pmatrix}8\\4\\10\end{pmatrix}}{2} = \begin{pmatrix}1\\0\\1.5\end{pmatrix}$$

11 Radiosity

11.1 Reflectance

In practice we only calculate the illumination from light sources and did not attempt to calculate the illumination of neighbouring objects, using

$$I_{\text{reflected}} = k_a + k_d(\mathbf{n}\cdot\mathbf{l})I_{\text{incident}} + k_s(\mathbf{r}\cdot\mathbf{v})^q I_{\text{incident}}.$$

A better approximation to the reflectance equation is to make the ambient light term a function of the incident light as well:

$$\begin{aligned} I_{\text{reflected}} &= k_a I_{\text{incident}} + k_d(\mathbf{n}\cdot\mathbf{l})I_{\text{incident}} + k_s(\mathbf{r}\cdot\mathbf{v})^q I_{\text{incident}} \\ &= (k_a + k_d(\mathbf{n}\cdot\mathbf{l}) + k_s(\mathbf{r}\cdot\mathbf{v})^q)I_{\text{incident}} \\ &:= R I_{\text{incident}} \end{aligned}$$

where R is a viewpoint-dependent reflectance function.

11.2 Radiosity

- **Radiosity**: the energy per unit area leaving a surface
- For each patch i the total energy leaving the patch is the sum of the energy it emits and the light energy it reflects, as

$$B_i = E_i + R_i I_i$$

- We can compute the total incident light at patch i from all the patches as

$$I_i = \sum_{j=1}^{n} B_j F_{ij}$$

where n is the number of patches in the scene, F_{ij} is the **form factor**, a constant linking surface patch i and j, and $F_{ii} = 0$.

- We can thus obtain the following equation

$$B_i = E_i + R_i \sum_j B_j F_{ij},$$

re-write the equation to be

$$B_i - \sum_j R_i B_j F_{ij} = E_i,$$

and formulate the problem in matrix form as

$$\begin{pmatrix} 1 & -R_1F_{12} & -R_1F_{13} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1 & -R_2F_{23} & \dots & -R_2F_{2n} \\ -R_3F_{31} & -R_3F_{32} & 1 & \dots & -R_3F_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -R_nF_{n1} & -R_nF_{n2} & -R_nF_{n3} & \dots & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ E_3 \\ \vdots \\ E_n \end{pmatrix}.$$

- The above system can be solved using **Gauss Seidel** method: use

$$B_i^k = E_i + R_i \sum_j B_j^{k-1} F_{ij}$$

and set initial values to e.g. $B_i^0 = 0$ to give successive estimates $B_i^0, B_i^1, \dots$. This evaluation process of B_i^k is called *gathering*, equivalent to evaluate the matrix row-by-row.

  – The method is stable and converges.
  – The radiosity matrix is 'diagonally dominant', which is sufficient to guarantee convergence.
  – At the first iteration, the emitted light energy is distributed to those patches that are illuminated.
  – Those patches illuminate others in the next cycle and so on.
  – Images start dark and progressively illuminate as the iteration proceeds.

- To make Gauss Seidel method faster, suppose in one iteration, B_i changes by $\Delta B_i = B_i^k - B_i^{k-1}$ called the **unshot radiosity**. Every other patch in the scene will change (if form factor is not zero) by

$$\Delta B_j = R_j F_{ji} \Delta B_i \tag{1}$$

modifying the iterative scheme to be

$$B_j^k = B_j^{k-1} + R_j F_{ji} \Delta B_j^{k-1}. \tag{2}$$

This evaluation process is called *shooting*, equivalent to evaluate the matrix column wise.

1. The patches with the largest ΔB are evaluated first
2. shoot the radiosity for the chosen patch and update all other patches with equataion (1) and (2).
3. set $\Delta B_i = 0$ and iterate

11.3 The Form Factor

The form factor F_{ij} can be computed as illustrated in figure 6 using the following equation

$$F_{ij} = \frac{1}{|A_i|} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} \mathrm{d}A_j \mathrm{d}A_i$$

where $|A_i|$ is the area of patch A_i. By assuming $|A_i| \ll r$ and the constancy

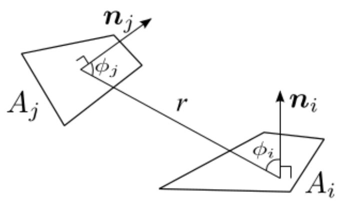


Figure 6: A form factor couples each pair of patches

across patch A_j, we can simplify to the approximate solution

$$F_{ij} = \frac{\cos\phi_i \cos\phi_j |A_j|}{\pi r^2},$$

giving the form factor repciprocity

$$F_{ji} = \frac{F_{ij}|A_i|}{|A_j|}$$

so form factors for only half the patches need to be computed.

But the amount of storage is still massive. If there are 6e5 patches, there are 3.6e9 form factors and need 7GB. As many of them are 0 we can save space by using an indexing scheme, e.g. use 1 bit per form factor, bit=0 indicates form factor=0 and not stored.

11.4 The Hemicube Method

- **Hemicube**: the half unit cube that bounds the hemisphere
  - All patches that project onto the same area of the hemisphere have the same form factor.
  - The hemicube is preferred to the hemisphere since computing intersections with planes is computationally less demanding.
  - A hemicube of side 1 unit is placed over the centre of a patch whose form factors are to be computed.
  - Projection of other *nearest visible* patches onto the hemicube could be done by ray tracing (neatly solves occlusion problem) or projection (requires z-buffering).
- **Hemicube pixels**: the set of square patches which each of the 5 faces of the hemicube is divided regularly into
  - the *larger* the size of the hemicube pixel
  - the *worse* the estimate of the form factors
  - the *faster* the algorithm
- **Delta form factor**: the form factor between the hemicube pixels and the patch under consideration
  - If the area of a hemicube pixel is $|A|$, the delta form factor is

$$\frac{\cos\phi_i \cos\phi_j |A|}{\pi r^2}.$$

  - These delta form factors can be stored in a look-up table.
  - They can then be applied to each patch without any more form factor calculations.

- To compute the delta form factors,

$$\begin{cases} \dfrac{|A|}{\pi r^4} & \text{since for a top face, } \cos\phi_i = \cos\phi_j = \dfrac{1}{r} \\ \dfrac{z_p|A|}{\pi r^4} & \text{since for a side face, } \cos\phi_i = \dfrac{1}{r}, \cos\phi_j = \dfrac{z_p}{r} \end{cases}$$

  where z_p is the height of the side face relative to the patch.
- Once we found the projection of patches, we compute the form factor for each patch by summing the delta form factors of the hemicube pixels to which it projects.

11.5 Meshing

- **Meshing**: the process of dividing the scene into patches
- Meshing artefacts are scene dependent.
- The most obvious are called D^0 artefacts, caused by discontinuities in the radiosity function.
- **Discontinuity Meshing** (*a priori*)
  - compute discontinuities in advance, e.g. object boundaries, albedo/reflectivity discontinuities, shadows, etc.
  - align the patches with them and don't interpolate them so that it is not smoothened
- **Adaptive Meshing** (*a posteriori*)
  - re-compute the mesh during the radiosity calculation
  - if two adjacent patches have a strong discontinuity in radiosity value, we can
    * put more patches into areas with high discontinuity OR
    * move the mesh boundary to coincide with the greatest change
- **h-refinement**: subdivision of patches
  1. Compute the radiosity at the vertices of the coarse grid.
  2. Subvdivide patches if the discontinuities exceed a threshold.

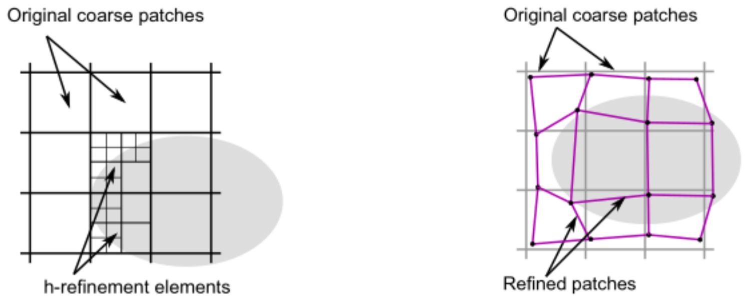


Figure 7: Adaptive Meshing using h-refinement on the left; adaptive meshing using r-refinement on the right

  3. not needed to re-compute complete radiosity by assuming that radiosity of a patch is the sum of that of its elements
- **r-refinement**: patch refinement
  1. Compute the radiosity at the vertices of the coarse grid.
  2. Move the patch boundaries closer together if they have high radiosity changes.
  3. Need to re-compute the entire radiosity solution for each refinement.

11.6 Radiosity Method and Issues

1. Divide the graphics world into discrete patches.
   Meshing strategies, meshing errors
2. Compute form factors by the hemicube method.
   Alias errors
   - due to discrete sampling of a continuous environment
   - not significant since the errors are averaged over a large number of pixels
3. Solve the matrix equation for the radiosity of each patch.
   Computational strategies — see parts regarding Guass Seidle method

4. Average the radiosity values at the corners of each patch.
   Interpolation approximations
   - may not be significant for small patches
5. Compute a texture map of each point or render directly.
   At least this stage is relatively easy!