Graphics

Lectured by Bernhard Kainz and Abhijeet Ghosh

Typed by Aris Zhu Yi Qing

March 13, 2022

Contents

1 Projections and Transformations **2**
- 1.1 Parallel Projection . . . 2
- 1.2 Perspective Projections . . . 2
- 1.3 Space Transformations . . . 2
  - 1.3.1 Homogeneous Coordinates . . . 2
  - 1.3.2 Translation Matrix . . . 2
  - 1.3.3 Scaling Matrix . . . 2
  - 1.3.4 Rotation Matrix . . . 3
  - 1.3.5 Projection Matrix . . . 3

2 Clipping **3**

3 Graphics Pipeline **4**
- 3.1 Application . . . 4
- 3.2 Geometry . . . 4
- 3.3 Rasterization . . . 4
- 3.4 Shading . . . 4
  - 3.4.1 Vertex Shader . . . 4
  - 3.4.2 Tessellation Shader . . . 5
  - 3.4.3 Geometry Shader . . . 5
  - 3.4.4 Fragment Shader . . . 5

4 OpenGL **5**

5 Illumination and Shading **6**
- 5.1 Physics of Shading . . . 6
- 5.2 The Phong Model . . . 6
- 5.3 Shading . . . 7
  - 5.3.1 Flat Shading . . . 7
  - 5.3.2 Gouraud Shading . . . 7
  - 5.3.3 Phong Shading . . . 7

6 Color **7**
- 6.1 CIE Color Space . . . 7
- 6.2 HSV Color Representation . . . 8

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 <u>anti-clockwise</u> when looking <u>along the direction</u> of the axis with a <u>left-hand</u> 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

- <u>**Clipping**</u> eliminates portions of objects outside the <u>**viewing frustum**</u>, which is the boundaries of the image plane projected in 3D with a near and far clipping plane.
- <u>Why</u> clipping?
  - avoid degeneracy: e.g. don't draw objects behind the camera
  - improve efficiency: e.g. do not process objects which are not visible.
- <u>When</u> 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 <u>before</u> 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 <u>after</u> 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**.
- 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$^2 \cdot$ 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

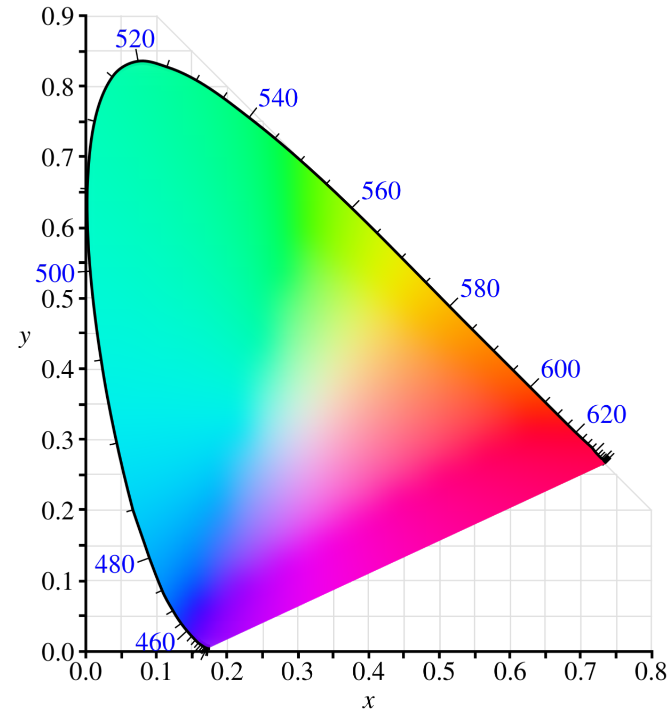

Figure 1: The CIE 1931 color space. Red is at $(0.628, 0.346, 0.026)$, Green is at $(0.268, 0.588, 0.144)$, Blue is at $(0.150, 0.07, 0.780)$

6 Color

6.1 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

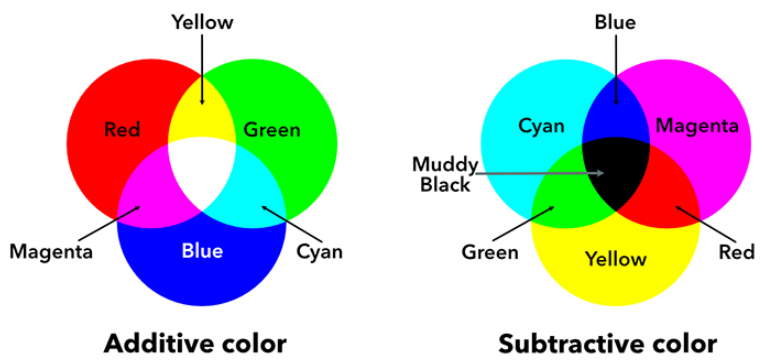

Figure 2: additive and subtractive primaries

- 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

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

6.2 HSV Color Representation

- **Hue** corresponds notionally to pure color
- **Saturation** is the proportion of pure color
- **Value** is the brightness/intensity