Summary

Computer Graphics

WBCS019-05

Bjorn Pijnacker

b.pijnacker.1@student.rug.nl

2021–2022, 2a

Contents

About this summary

This summary is based on the lecture slides for Computer Graphics WBCS019-05 2021–2022 by Jiri Kosinka and on the book Fundamentals of Computer Graphics 4th ed. by S. Marschner and P. Shirley. Chapters in this summary follow the chapters in the fourth edition of the book.

Chapter 1 and 2 are skipped in the summary, since they cannot be summarized well. The reader is encouraged to go through these chapters themselves. This summary was written with having watched the lectures in mind, and as such only serves as a refresher of the material. Sections of the book that are fully comprised of material not covered by the lectures are skipped.

While chapters 22 and 23 are named in the material, they are not included here. Chapter 22 is extremely in-depth w.r.t. implicit modeling, while most that is named in the slides is covered in other chapters. Chapter 23 is not examinable for the most part, so you may wish to read this, but I skipped it here. If you have any feedback or questions, please do let me know.

Chapter 3

Raster Images

Most computer graphics images are presented to the user in some kind of raster display, made up of pixels. Rasters are also prevalent as input devices. For example, a digital camera has a image sensors made up of pixels. The most common way to store images is using raster images, which is simply a 2D array with a value per pixel, usually stored as *red*, *green*, and *blue*. Other ways of describing images is using vector images, which store descriptions of shapes.

3.1 Raster Devices

3.1.1 Displays

Current displays consist of fixed arrays of pixel. We differentiate between emissive displays, where the pixels make light themselves, and transmissive displays, which employ a backlight behind the array. An example of an emissive display is an LED display. An example of a transmissive display is an LCD display.

3.1.2 Hardcopy Devices

Printers do not have a physical array of pixels, rather, the resolution is determined by how small the drops of ink can be made and how often these can be placed on the paper. Ink is mixed to allow for many colors to be printed onto the paper. A printer's resolution is usually measured in "dpi", which stands for "dots per inch".

3.1.3 Input devices

A digital camera is an example of a 2D array input device which captures a raster image. The image has a grid of light-sensitive pixels, which measure the light that fall onto them to store a picture.

Another common input device is a flatbed scanner. This uses a 1D array that sweeps across the page being scanned, making multiple measurements with the same input units. Similar to

a printer, the resolution is fixed by the amount of pixels on the array, and the speed at which the array moves and can capture information.

3.2 Images, Pixels, and Geometry

We can abstract an image as a function $I(x, y) : R \to V$, where $R \subset \mathbb{R}^2$ is a finite rectangular area and V is the set of possible pixel values. An idealized color image with red, green, and blue values for each pixel would have $V = (\mathbb{R}^+)^3$. By convention, a raster image is indexed by the pair (i, j) indicating column i and row j, counting from the bottom left. The integer coordinates of a pixel are in the center of that pixel.

3.2.1 Pixel Values

When storing images, we cannot have a continuous value for each pixel, since we would need infinite storage. Instead, we store the pixel values using a floating point number of some size. Generally, for high quality images, this is 32 bits. This allows us to store enough colors such that humans cannot differentiate between two neighboring color values.

3.2.2 Monitor Intensities and Gamma

All modern monitors take digital input for the value of a pixel and convert this to some intensity. There are some issues with just giving the pixel the lightness value of the input: first of all, monitors are nonlinear with respect to input. An approximate characterization of this nonlinearity is commonly named γ. If we can measure a monitor, we can find a suitable measure of γ that gives us the correct display output. The second issue is that monitors are not uniform. Some pixel value might differ in actual brightness and color on two different parts of the screen. Fortunately, this amount of color accuracy is rarely needed.

Chapter 4

Ray Tracing

We omit the sections about ray-object intersection.

4.5 Shading

Once we know the visible surface for a pixel, we now must calculate its value using a shading mode. Two basic models are described in this chapter. Most shading models are designed to capture the process of light reflection, where surfaces are illuminated by light sources and reflect part of the light to the camera. The important variables to consider are the light direction $\mathbf{l}$ which points from the hit to the light source, the view direction $\mathbf{v}$ which points to the camera, the surface normal $\mathbf{n}$ which is the normal vector at the hit point, and of course the characteristics of the surface.

4.5.1 Lambertian Shading

Lambertian shading is the simplest shading model. It uses the observation that the amount of energy from a light source onto a surface depends on the angle between $\mathbf{l}$ and $\mathbf{n}$. This gives us the Lambertian shading model:

$$L = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

where L is the pixel color, k_d is the diffuse coefficient of the material, and I is the intensity of the light source. Since $\mathbf{l}$ and $\mathbf{n}$ are both unit vectors, $\mathbf{l} \cdot \mathbf{n}$ serves as a shorthand for $\cos\theta$.

4.5.2 Blinn-Phong Shading

Blinn-Phong shading introduces specular highlights to create a more realistic shading model. Its idea is to produce reflection at its brightest when $\mathbf{v}$ and $\mathbf{l}$ are symmetric over $\mathbf{n}$. We add this to the Lambertian shading model, giving us a diffuse and a specular component. We find

$$L = k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p,$$

where k_s is the specular coefficient of the surface, $\mathbf{h}$ is the half vector between $\mathbf{h}$ and $\mathbf{l}$, and p is the Phong exponent.

4.5.3 Ambient Shading

The last component we add is the ambient component, so that surfaces without illumination don't appear completely black. We add this to the shading model to find

$$L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p.$$

4.5.4 Multiple Point Lights

If we have multiple lights, we compute the diffuse component and specular component for each light. We then sum these together and add the ambient component to find our final pixel color.

4.7 Shadows

Once we have basic ray tracing, we can easily add shadows. When casting a ray from the camera, we find a hit point. When shading this hit, we cast another ray from this point to the light source. If this ray hits another object before reaching the light source, we know that our current object is in shadow.

If the object is in shadow for a given light, we skip this light in the shading calculation. This means we do add components for other lights for which the object may not be in shadow, and we also add the ambient component in all cases.

4.8 Ideal Specular Reflection

To add ideal specular reflection to a ray tracing program we cast a reflection ray from the hit of the first ray. The direction of this reflection ray $\mathbf{r}$ is calculated using

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}.$$

If we do this recursively to find the reflection perfectly, it may never terminate. As such, most implementations have a maximum recursion depth for calculating reflections.

Chapter 6

Transformation Matrices

Linear algebra is often used to express operations required to change objects in a 3D scene, view them with cameras, and get the onto the screen. We will discuss geometric transformations such as rotation, translation, scaling, and projection using matrix multiplication.

6.1 2D Linear Transformations

To transform a 2D vector, we can use a 2×2 matrix:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{bmatrix}.$$

Such a operation, which takes a 2-vector and produces another 2-vector by simple matrix multiplication is a linear transformation.

6.1.1 Scaling

To scale a 2D vector along coordinate axes we use

$$\text{scale}(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}.$$

This scales x with s_x, and y with s_y.

6.1.2 Shearing

We can shear in two directions: x and y. We find

$$\text{shear-x}(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}, \qquad \text{shear-y}(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}.$$

6.1.3 Rotation

To rotate a vector **a** by an angle ϕ we use the rotation matrix

$$\text{rotate}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}.$$

The rotation will be by the angle ϕ counterclockwise.

6.1.4 Reflection

To reflect a vector across either coordinate axis, we use the identity matrix where the component corresponding to the axis to flip over is negated.

6.1.5 Composition of Transformations

If we want to perform multiple transformations, we can compose these into one. We can multiple all the matrices together in the order we wish to perform then from right to left. We can then multiply this result with our vector. Since matrix multiplication is not commutative, the order that they are placed in *does* matter.

6.2 3D Linear Transformations

In 3D, transformation matrices are a logical extension of those in 2D. As such, we only consider transformations that are actually different here.

One difference is the rotation, because there are multiple axes of rotation. We show three rotation matrices:

$$\text{rotate-z}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$\text{rotate-x}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix},$$

$$\text{rotate-y}(\phi) = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}.$$

6.2.1 Arbitary 3D Rotations

To rotate over some arbitrary vector **w**, we van create a matrix $\mathbf{R}_{uvw}$ to move our **w** to align with **z**. We have the other orthogonals **u** and **v** as well, which are the corresponding 'moved' **x**

and $\mathbf{y}$ to $\mathbf{w}$. We calculate our rotation by doing

$$\begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}.$$

6.2.2 Transforming Normal Vectors

When we transform a surface we must also properly transform its normal vectors. In some cases, such as shearing, the normal vector will not be normal to the surface after transformation. If we have some transformation matrix $\mathbf{M}$, we must multiple the normals by $\mathbf{N} = (\mathbf{M}^{-1})^{\mathsf{T}}$.

Skipping some steps, we find that the matrix $\mathbf{N}$ is, in general,

$$\mathbf{N} = \begin{bmatrix} m^c_{11} & m^c_{12} & m^c_{13} \\ m^c_{21} & m^c_{22} & m^c_{23} \\ m^c_{31} & m^c_{32} & m^c_{33} \end{bmatrix}.$$

6.3 Translation and Affine Transformations

A 3D translation cannot be represented in a 3×3 transformation matrix: we must instead use an extra row in our matrix. We represent our 3D point (x, y, z) by the vector $[\, x \; y \; z \; 1 \,]^{\mathsf{T}}$ and use a 4×4 matrix of the form

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & x_t \\ m_{21} & m_{22} & m_{23} & y_t \\ m_{31} & m_{32} & m_{33} & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We call this kind of transformation an affine transformation, and this way of transforming using an extra dimension is called homogeneous coordinates.

When we wish to transform vectors that are not positions, the extra coordinate should be 0, not 1. This way, the translation has no effect.

6.4 Inverses of Transformation Matrices

Of many common transformations, the inverse is easily found without algebraically inversing the matrix. For example, the inverse of a matrix scale(s_x, s_y, s_z) is scale$(s_x^{-1}, s_y^{-1}, s_z^{-1})$. A rotation is also simple to invert: we transpose the matrix. This is possible, since the matrix is ortogonal.

Chapter 7

Viewing

Matrix transformations, as shown in Chapter 6, are also used for mapping a 3D scene to a 2D view of the scene. This is called a viewing transformation.

7.1 Viewing Transformations

The viewing transformation has the job of mapping 3D locations to coordinates in an image, expressed in units of pixels. This transformation depends on the camera position and orientation, the type of projection, the field of view, and the image resolution.

Most graphics systems use a sequence of three transformations: (1) a camera transformation, which places the camera at the scene origin; (2) a projection transformation, which projects points from camera space so all visible points fall in $[-1, 1]$ in both x and y; (3) a viewport transformation, which maps the unit image rectangle to the desired rectangle in pixel coordinates. The first of these depends only on the camera position and orientation, the second only on the type of projection, and the third only on the size and position of the output image.

7.1.1 The Viewport Transformation

We assume the geometry we want to view is in the canonical view volume, and we wish to view it with an ortographic camera looking in the $-z$ direction. We wish to map the $[-1,1]^2$ rectangle to $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$, since the latter is our pixel space.

We do not need to alter the z-coordinate, only the x and y, so we arrive at

$$M_{vp} = \begin{bmatrix} n_x/2 & 0 & 0 & n_x - 1/2 \\ 0 & n_y/2 & 0 & n_y - 1/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

7.1.2 The Ortographic Projection Transformation

A projection transformation aims to project our scene into the canonical view volume such that arbitrary rectangles can be viewed. We constrain the view volume to an axis-aligned box, and we name the coordinates of its sides such that the view volume is $[l, r] \times [b, t] \times [f, n]$.

The matrix to transform this box to the required one is given by

$$M_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

7.1.3 The Camera Transform

Instead of having our box be axis-aligned, we want to be able to look in any direction. We will use **e** to mean the camera position, **g** to mean the gaze direction, and **t** to mean the view-up vector. These three vectors give us the ability to set up a **uvw** system for the viewing box, independent of the **xyz** axes.

Expressing **u**, **v**, and **w** in **xyz**, we find the following matrix for the camera transformation:

$$M_{cam} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

7.3 Perspective Projection

To do a perspective projection, instead of projecting the entire box linearly, we must have a box that scales with $-z$, so further away objects get scaled to be smaller. We find the perspective matrix

$$M_{per} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

In this case, the relationship between l and r represents the horizontal field of view, and the relationship between t and b represents the vertical field of view.

Chapter 8

The Graphics Pipeline

In rasterization, we have a sequence of operations required to render an image known as the graphics pipeline. This pipeline starts with objects, and ends with pixels in an image. This type of rendering is very efficient, compared to ray tracing.

8.1 Rasterization

Rasterization is central to the graphics pipeline. For any primitive that comes in, the rasterizer enumerates the pixels covered by the primitive, and interpolates values accross the primitive.

8.1.1 Line Drawing

To draw a line given two endpoints (x_0, y_0) and (x_1, y_1) we need to draw a "reasonable" set of pixels that approximate the line. We do this using the implicit equation of the line.

We use the midpoint algorithm. The first step is to find the implicit equation for a line, which is given by

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0.$$

We assume that $x_0 \leq x_1$. The slope m is given by

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

where we assume that $m \in (0, 1]$ in the following steps. Similar discussions can be held for the other cases of m.

We move pixel-wise from the begin- to the endpoint. We have two options: draw a pixel here, or draw a pixel one up. We consider the midpoint between these pixels. Is this below the line? Then draw up. Is this above the line? Draw the current pixel. If we keep doing this until we get to the endpoint, we will have a good approximation of our line.

8.1.2 Triangle Rasterization

To simply draw a triangle, we can consider the barycentric coordinates for that triangle of all pixels. If all three barycentric components are in $[0, 1]$, then we draw the pixel. Otherwise, we do not. Using barycentric coordinates, we can also interpolate vertex values by using the attributes instead of the vertex coordinates in the barycentric calculation.

8.1.3 Clipping

When rasterizing, we must take care to not rasterize primitives outside of screen space, since doing this can lead to incorrect results. To fix this, we need a clipping operation that removes these first. We have two options for clipping: in world coordinates using the six bounding places of the truncated viewing pyramid, or in the 4D transformed space before the homogeneous divide.

8.1.4 Clipping Before the Transform (Option 1)

For this option we only have to know the six plane equations. We can then check whether the primitive is inside the box and leave it as is; partially inside and clip it; or complete outside and cull it.

8.1.5 Clipping in Homogeneous Coordinates (Option 2)

In this case, the view volume is 4D and is bounded by 3D hyperplanes. Clipping on these hyperplanes is more efficient than option 1.

8.1.6 Clipping against a Plane

The equation for a plane through point $\mathbf{q}$ with normal $\mathbf{n}$ is

$$f(\mathbf{p}) = \mathbf{n} \cdot (\mathbf{p} - \mathbf{q}) = 0,$$

often written as

$$f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + D = 0.$$

If we have a line segment between $\mathbf{a}$ and $\mathbf{b}$, we can clip it against the plane. We check $\mathbf{a}$ and $\mathbf{b}$ to determine whether they are on opposite sides of the plane by checking whether $f(\mathbf{a})$ and $f(\mathbf{b})$ have different signs. If the plane does split the line, we can solve for the intersection point by substituting the equation for the parametric line

$$\mathbf{p} = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$$

into the $f(\mathbf{p}) = 0$ plane. This yields

$$\mathbf{n} \cdot (\mathbf{a} + t(\mathbf{b} - \mathbf{a})) + D = 0,$$

which when solved for t yields

$$t = \frac{\mathbf{n} \cdot \mathbf{a} + D}{\mathbf{n} \cdot (\mathbf{a} - \mathbf{b})}.$$

8.2 Operations Before and After Rasterization

Before a primitive can be rasterized, the vertices and associated attributes must be known. This is done by the vertex-processing stage in the pipeline. After rasterization, further processing is done to compute the color and depth for each fragment. This can be simple or complicated, depending on the shading used.

8.2.1 Simple 2D Drawing

The simplest pipeline lets the rasterizer do all the work. Here, colors are simply interpolated over the shape, and resulting fragments are used as pixels directly.

8.2.2 A Minimal 3D Pipeline

To get a 3D pipeline to work properly, we also need depth information to get occlusion relationships correct. For this, we use the painter's algorithm, where we draw far away objects first, such that nearer objects can be drawn over them.

8.2.3 Using a z-Buffer for Hidden Surfaces

In practice we rarely use the painter's algorithm. Instead, we use a z-buffer algorithm. For each pixel, we keep track of the distance to the closest surface that has been drawn so far, and we throw away any fragments that are further away than that distance. This way, only the closest object will be visible when we are done. This algorithm is implemented in the fragment blending phase, by comparing the depths of each fragment. The z-buffer is initialized to the value of the far viewing plane, so that the first object is closer.

8.2.4 Per-vertex Shading

We can do shading calculations at the vertex stage, having the resulting color be interpolated by the rasterizer. This is sometimes called Gouraud-shading. This has the disadvantage that it cannot produce any details in the shading smaller than the surface primitives.

8.2.5 Per-fragment Shading

Instead of interpolating the color, we interpolate the normals between vertices. We can then calculate the shading in the fragment stage.

8.2.6 Texture Mapping

The most common way to define textures is to simply make the texture coordinate another vertex attribute. Each primitive then knows where it lives in the texture.

8.3 Simple Antialiasing

Rasterization produces ragged lines and triangle edges. The solution to this is to allow pixels to be partially covered by a primitive. We set the image pixel to an average value of the square area belonging to the pixel; an approach knows as box filtering. The easiest way to do this is to implement supersampling: create images at very high resolution, and then downsample it. This is an approximation to the actual box filtered image, but it works well enough.

Chapter 10

Surface Shading

10.1 Diffuse Shading

Diffuse shading is a shading method that results in a "matte" look on the object. Such objects can be considered as behaving as Lambertian objects.

10.1.1 Lambertian Shading Model

A Lambertian object obeys Lambert's cosine law, which states that the color c of a surface is proportional to the angle between the surface normal $\mathbf{n}$ and the direction to the light source $\mathbf{l}$.

A surface can be made lighter or darker by changing the intensity of the light source or by changing the reflectance of the surface. The diffuse reflectance, c_r is the fraction of light reflected by the surface. This c_r can differ per color component of RGB, to give the surface a color. If we model the light color as c_l, we find that the final output color c is given by

$$c = c_r c_l \max(0, \mathbf{n} \cdot \mathbf{l}).$$

In this case we add the $\max(\cdot)$ to account for negative dot products, where $\mathbf{n}$ and $\mathbf{l}$ point away from each other.

10.1.2 Ambient Shading

With the above equation, many surfaces might be completely black. In real life, this is rarely the case. This is why often an ambient component is added to the shading. This makes all surfaces non-black, for a more realistic approach to shading. Given the ambient color c_a, we find

$$c = c_r \left(c_a + c_l \max(0, \mathbf{n} \cdot \mathbf{l})\right).$$

10.1.3 Vertex-Based Diffuse Shading

If this shading model is applied to an object made of triangles, the boundaries between these triangles will be clearly visible. To avoid this, we can place surface normals at vertices, and

interpolate the color of these vertices using the barycentric interpolation method.

10.2 Phong Shading

The Phong illumination model concerns non-matte objects. It adds a specular highlight to surfaces, which move across the object as our camera moves.

10.2.1 Phong Lighting Model

The specular highlight is computed as

$$c = c_l \max(0, \mathbf{r} \cdot \mathbf{e})^p,$$

where $\mathbf{r}$ is the reflection of the light ray over the surface normal, $\mathbf{e}$ is the direction towards the camera, and p is the Phong exponent, which determines the width of the specular highlight. We combine this with the Lambertian shading model to find

$$c = c_r \left(c_a + c_l \max(0, \mathbf{n} \cdot \mathbf{l})\right) + c_l \max(0, \mathbf{r} \cdot \mathbf{e})^p.$$

Optionally, we can add a component c_p to the rightmost term so that the user can control the specular color.

10.2.2 Surface Normal Vector Interpolation

Shading at the normal vectors and interpolating the result can yield disturbing artifacts, and cannot accomodate details in the colors. Instead, we interpolate the normals across the polygon and apply Phong shading at each pixel. This allows us to get good images without needing very fine geometry.

10.3 Artistic Shading

The two shading methods above are based on heuristics designed to imitate realism. Artistic shading is designed to mimic drawings made by humans or to provide efficiency and more information in cases such as CAD rendering.

10.3.1 Line Drawing

Line drawing concerns drawing silhouettes around our objects. When we have a set of triangles with shared edges, we can draw the edge as a silhouette when one of the two triangles sharing an edge faces towards the viewer, and the other does not. This is true when $(\mathbf{e} \cdot \mathbf{n}_0)(\mathbf{e} \cdot \mathbf{n}_1) \leq 0$.

10.3.2 Cool-to-Warm Shading

Cool-to-warm shading is a method used to shade line drawings. Surfaces facing in one direction are shaded with a cool color, while surfaces in the other are shades using a warm color.

This helps illustrate angles in a line drawing without losing any necessary details due to dark shading.

Chapter 11

Texture Mapping

The book goes into much more detail than the slides with regards to texture mapping. This section is as such derived from the slides, specifically `05b_Textures.pdf`*, and not the book.*

So far, we have been limited to final color based on materials and lights. All geometry has been described by meshes. With these limitations, we are unable to have fine detail in our objects. The solution to this is to use textures.

Textures can be used to modify the surface material, transparency, normal vectors, geometry, and more.

11.1 Texture Mapping

To use a texture, we must map a 3D (x, y, z) coordinate on our object to a 2D (u, v) coordinate on our texture. This is called texture mapping. This can be mapped on a per-vertex basis, or this can be done procedurally for some shapes.

For a sphere, the u and v coordinates are provided by

$$u = 0.5 + \frac{\arctan2(y, x)}{2\pi} \qquad v = 1 - \frac{\arccos z/r}{\pi}$$

In general, we require mesh 'flattening' with minimal distortion. This operation is not trivial and differs greatly per model used.

11.2 Different Texture Types

Color textures can give us the diffuse coefficient to use in shading. Aside from colors, we can also use textures to provide normals for our model by using a normal map. This allows us to have small details in shading that are not created by geometry but rather by small color fluctuations. This gives for a much more realistic result in shading, as objects do not appear compeltely flat.

We can also add small details in geometry using textures. This is most often done using a displacement map, where small subdivision is required to displace vertices and introduce imperfections in the mesh.

11.3 Texture Sampling

11.3.1 Perspective Interpolation

When sampling a texture, we must take care to correctly interpolate perspective. The u value for proper perspective interpolation is given by

$$u = \frac{(1-a)u_0/z_0 + au_1/z_1}{(1-a)1/z_0 + a1/z_1},$$

where a is the value to interpolate to, u_0 and u_1 represent the value of u at two given vertices, and z_0 and z_1 represent the depth of those same vertices.

11.3.2 Neighbor Interpolation

When an interpolated (u, v) does not line up exactly with a texel in our texture, we have multiple ways of finding the output color:

Nearest neighbor is the simplest. Here, we simply grab the closest texel and use this on our render. Nearest neighbor does not look great, however. Another common method is bilinear interpolation, where the four surrounding texels are used, and the final color is the interpolated value of these texels, weighed by the relative distance of the sample point.

11.3.3 MIP map

Mipmapping is a technique where a texture is stored at multiple resolutions. When sampling the texture, the resolution is chosen based on the size of the object on the screen. This yields slightly blurred textures on far away of small objects, which is much more visually appealing than the alternative. For an example, see the slides.

One form of mipmapping is anisotropic mipmapping, where instead of the resolution being reduced in both dimensions, a copy exists of each image where the resolution is only reduced in a single dimension, one for horizontal, and one for vertical. This yields a render with much less visible blurring, but also without the artifacts of no mipmapping at all.

Chapter 12

Data Structures for Graphics

12.1 Triangle Meshes

Triangle meshes are often used to represent surfaces, where edges with shared vertices for a single continuous surface. For a triangle we store its vertices. We need at least its position in 3D space, but often we store additional attributes as well. Most often, we have vertex data which is interpolated over the surface as values change, but we may also wish to store data per edge or per face.

12.1.1 Mesh Topology

We can impose constraints on mesh topology to formalize the idea of a surface. These constraints hold for the way triangles connect together; not for their vertex positions. The simplest and most restrictive algorithm is the requirement of the topology to be manifold. A manifold mesh is closed: it has no gaps, and it separarates the volume inside and outside of the surface.

There is many different algorithms to ensure a mesh is manifold. The verification boils down to checking that all edges and all vertices verify the following conditions: every edge is shared by exactly two triangles; every edge has a single, complete loop of triangles around it.

Manifold meshes cannot have boundaries, while we sometimes do wish to have boundaries in our mesh. As such, we can relax the requirements to obtain a set of requirements for a 'manifold with boundary' mesh. These conditions are: every edge is used by either one or two triangles; every vertex connects to a single edge-connected set of triangles.

Finally, it is also important to distinguish the front (or outside) and back (or inside) of the surface. This is known as its orientation. We define this based on the order in which the vertices are listed. The front is the side from which the vertices are in counterclockwise order. A connected mesh is consistently oriented if all triangles agree on which side is the front.

12.2 Scene Graphs

A complex scene can contain many transformations for a single object. Organizing a scene well can make it much easier to manipulate and work with. Most scenes use a hierarchical organization, where transformations can be managed using a scene graph.

An example of how this makes transformations easier is by considering a car: A car has wheels, which should move with the car. When transforming the wheels, the transformation matrices of objects above it in the hierarchy have already been applied, so they will move with the car without special care needing to be taken as far as the wheels are concerned. This also means that the wheels have their own local coordinate system. This makes it easier, for example, to rotate them over their own driving axis, if they are globally placed at the world origin, and then moved to be with the car by some transformation matrices.

12.3 Spatial Data Structures

In a graphics application, it is important to be able to quickly locate geometric objects in a particular region of space. For example, when interecting a ray, we can optimize by only considering objects we know the ray *might* hit. This need can be supported by various spatial data structures, designed to organize objects in space for efficient lookup.

12.3.1 Bounding Boxes

For intersection-acceleration, a bounding box intersection is a key operation. This allows is to determine whether a ray might hit an object or not, before doing a more complex intersection. For the algorithm of bounding box intersection, please see the book.

12.3.2 Hierarchical Bounding Boxes

Checks for box intersection are not free, so we might also wish to optimize the number of these. We can make the bounding box structure hierarchical by partitioning the set of objects in a box, and then placing another box around a partition. We then obtain a tree of bounding boxes, which reduces the number of box intersections that need to take place from $O(n)$ to $O(\ln n)$.

Important to note is that two subtrees may overlap, since there is no geometric ordering between two subtrees. When creating a tree, it is convenient to make the tree mostly balanced. A heuristic to accomplish this it to sort the surfaces along an axis before dividing them into two sublists. We can carefully choose the axis each time such that the sum of the volumes of the bounding boxes of the two subtrees is minimized. We can optimize this further by just partitioning, instead of sorting completely.

12.3.3 Uniform Spatial Subdivision

Another way to reduce intersection tests is to divide space. In spatial subdivision, each point in space belongs to exactly one node whereas objects may belong to many nodes. The scene is

divided into axis-aligned boxes, a grid if you will. A ray traverses these boxes until an object is hit. For each box, only objects inside that box should be considered.

12.3.4 Axis-Aligned Binary Space Partitioning

A hierarchical data structure similar to the bounding boxes is the binary space partitioning tree, or BSP tree. A node in this structure is a single cutting plane and a left and right subtree. The entire space is divided into two sections. We can do this again for the subtrees, to obtain a hierarchical split of the data. When a ray is cast, we can calculate the intersections with the splitting plane first, to know which side of it we can leave out of consideration.

Chapter 13

More Ray Tracing

13.1 Transparency and Refraction

Rays can travel through transparent objects, but when the refractive index of two volumes is different, the ray changes direction when it passes from one to the other. When a ray travels from a medium with refractive index n_i into one with refractive index n_t, Snell's Law tells us that

$$n_i \sin\theta = n_t \sin\phi,$$

where θ is the angle between the incoming ray and the surface normal, and ϕ is the angle between the ray inside the medium and the reflection of the surface normal over the surface. When a ray exists the medium again, the same holds, but with n_i and n_t reversed.

In transparent object we have the notion of total internal reflection. This happens when the angle θ nears 90 degrees. We can precisely model this using Fresnel equations, but we can approximate it using Schlicks approximation, which tells us that

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5,$$

where R_0 is given by

$$R_0 = \left(\frac{n_t - 1}{n_t + 1}\right)^2.$$

13.4 Distribution Ray Tracing

Distribution Ray Tracing is used to mitigate the "clean" look of ray-traced images.

13.4.1 Antialiasing

Usually in supersampling antialiasing we render each pixel by sending multiple rays through a pixel and averaging them. Normally, these rays were equally spaced throughout the pixel. We call this regular sampling. Instead, we can do random sampling, where rays are distributed

throughout the pixel randomly. This helps mitigate artificats that can arise due to regular sampling, such as moiré patterns.

13.4.2 Soft Shadows

In reality, lights are not infinitely small points. This means not all light rays come from the same location, and thus shadows do not have hard edges, since a surface can also be partially occluded from a light source. Distribution ray tracing can be used to model soft shadows. We represent the area light as an infinite cluster of point lights, and choose a random one for each viewing ray. This yields slighly different shadow calculations for each pixel, given the illusion of soft shadows.

13.4.3 Depth of Field

In real life, a camera is also not infinitely small. Since a sensor has a surface and uses a lens, there is a distance at which objects are in focus, where other objects are not. To model depth-of-field, we randomly select the eye position from a plane that represents the camera, instead of using a fixed **e**.

13.4.4 Glossy Reflection

Some surfaces are somewhere between an ideal reflector and a diffuse surface. To model this, we randomly scatter reflection rays from the surface. Instead of casting **r**, we perturb the ray to $\mathbf{r}'$, which we then cast.

To choose $\mathbf{r}'$, we again sample a random square. This square will be perpendicular to **r** and has a width which controls the degree of blur. If we have 2D sample points $(\xi, \xi') \in [0,1]^2$ then the analogous point on the square is

$$u = -\frac{a}{2} + \xi a \qquad v = -\frac{a}{2} + \xi' a.$$

Because the square is parallel to both **u** and **v**, the ray $\mathbf{r}'$ is simply

$$\mathbf{r}' = \mathbf{r} + u\mathbf{u} + v\mathbf{v}.$$

13.4.5 Motion Blur

To model motion blur with respect to time we choose a random time in the scene to cast each ray, between T_0 and T_1, using

$$T = T_0 + \xi(T_1 - T_0).$$

Of course, for motion blur to be visible, objects must also move with time.

Chapter 15

Curves

This chapter does not cover all that is named in the slides. It is important to go over the slides as well. Similarly, this chapter goes into a lot more detail in some sections compared to the slides; these details are not covered here.

15.1 Curves

A curve is an infinitely large set of points. All points have two neighbors, except the endpoints, which have a single neighbor. Some curves have no endpoints, either because they are infinite, or because they are closed. There are three main ways to specify curves mathematically:

(1) Implicit curve representations define the set of points on a curve by giving a boundary condition for a multivariable function one dimension above the curve. An example is $f(x, y) = 0$ which is a boundary condition on a function in 3D, which produces a curve in 2D. In this case, f must be a scalar function.

(2) Parametric curve representations provide a mapping from a free parameter to the set of points on the curve. This parameter is the index to the points on the curve. Intuitively this parameter can be seen as the time, if you consider drawing the curve on paper using a pen. An example of such a function is $(x, y) = \mathbf{f}(t)$. The dimension of $\mathbf{f}$ determines the dimensionality of the curve.

(3) Generative or procedural curve representations that can generate the points on the curve do not fall into either two catagories above. Examples of these are fractals.

15.1.1 Parameterizations and Reparameterizations

When we have a parametric curve, we usually want the parameter to run over the unit interval from 0 to 1. We consider $u = 0$ to be the start of the curve, and $u = 1$ to be the end. If we have some function $\mathbf{f}(t)$ where t runs over $[a, b]$, we define $g(u) = a + (b - a)u$ and then $\mathbf{f}_2(u) = \mathbf{f}(g(u))$. These two functions provide the same curve, but using different parametrizations. This is called reparameterization. The function g is called the reparameterization function.

15.2 Curve Properties

15.2.1 Continuity

It is very important to understand the local properties of a curve in the place where two parametric pieces come together. If we have two curves of which the endpoints join up, we would like to know whether or not these are continuous or not.

For two curves, we can first check whether the positions where they join—$\mathbf{f}_1(1)$ and $\mathbf{f}_2(0)$—are the same. If not, the curve is not continuous. Additionally, we can check whether the curves move in the same direction, by checking that $\mathbf{f}_1'(1) = \mathbf{f}_2'(0)$. In general, we say a curve is C^n continuous if all of its n-derivates match up. The position itself is considered the zeroth derivative. C^n-continuity is commonly referred to as parametric continuity, since it depends on the parameterization of the two curve pieces.

If we only care about the shape of the curve, we must consider geometric continuity that requires the derivates match when the curves are parameterized equivalently. This means that the corresponding derivates must have the same direction, but not the same magnitude: the vector must be equal up to some scalar k. The definition of G^n continuity is then the same as C^n continuity, except the used comparison is $\mathbf{f}_1'(1) = k\mathbf{f}_2'(0)$. A curve that is C^n is also G^n, since in that case $k = 1$.

15.6 Approximating Curves

15.6.1 Bézier Curves

Bézier curves are one of the most common representations for free-form curves in computer graphics. It is a polynomial curve that approximates its control points. These curves can be polynomial to any degree, where a curve of degree d is controlled by $d + 1$ control points.

To solve Bézier curves exactly, we need blending functions for each of the control points. These blending functions are given, in general, by

$$b_{k,n}(u) = \binom{n}{k} u^k (1-u)^{(u-k)},$$

which is known as the Bernstien basis polynomials. In this equation, n is the order of the Bézier curve and k is the blending function number in $[0, n]$.

Geometric Intuitition for Bézier Curves

Imagine we have a set of control points. Connecting these points with straight lines will lead to something that is not smooth. We can smooth this by cutting off the sharp corners continuously, yielding a smooth curve after infinitely many iterations.

The de Casteljau Algorithm

The de Casteljau algorithm is a very simple and general method for computing and subdividing Bézier curves. It uses a sequence of linear interpolations to compute the positions of Bézier curves, analogous to the corner-cutting described above.

The algorithm begins by connecting every adjacent set of points with lines, and finding the points on these lines that is the u interpolation, giving us $n - 1$ points. For this set of points, this is done again, giving $n - 2$ points. This is repeated until just one point is left. This final point is the value of the Bézier curve at u.

Chapter 19

Color

19.1 Colorimetry

Colorimetry is the science of color measurement and description. Since color is a human response, color measurement should begin with human observation. As such, color is measured using the human response to patches of color. Experiments such as these have resulted in standardized observers, which are approximations of the average human observer.

Since humans have three cone types, color matching can be summed up as the trichromatic generalization, which states that any color stimulus can be completely matched with an additive mixture of three modulated color sources. This provides us the notion of RGB pixels, too.

19.1.2 Cone Responses

Each cone type is sensitive to a range of wavelengths, spanning most of the visible range. This sensitivity is not evenly distributed, but instead contains a peak wavelength. The location of this peak is different for each cone type, which allows humans to percieve color.

19.1.4 Standard Observers

Using a color matching experiment for a large range of colors, it is possible to generate an average color matching dataset. The CEI has defined three primaries to be monochromatic light sources of 435.8 nm, 546.1 nm, and 700 nm. Using these, all other visible wavelengths can be matched by adding different amounts of each. However, to match some wavelenghts, a negative component is needed, which is obviously not physically possible. To solve this, we can shift both the light we are trying to match and the three primaries to such a value that no negative component is necessary.

19.1.5 Chromaticity Coordinates

Each color can be represented by a set of three tristimulus values (X, Y, Z). We can define an orthogonal coordinate system with three axes and plot each color in the resulting 3D space. This is called a color space. The spatial extent of this is named the color gamut.

Since plotting color in 3D is fairly difficult, we can project tristimulus values to a 2D space which approximate chromoatic information, and removes the luminance information. This is obtained using the following equations. First,

$$x = \frac{X}{X+Y+Z},$$
$$y = \frac{Y}{X+Y+Z},$$
$$z = \frac{Z}{X+Y+Z}.$$

then,

$$X = \frac{x}{y}Y,$$
$$Z = \frac{1-x-y}{y}Y.$$

19.2 Color Spaces

For a set of tristimulus values that can produce every color in the visible spectrum, we need imaginary primaries, which is not very practical. For that reason, image encodings and computations on images use a so-called color space. A color space is often specified by a specific transform from CIE XYZ. For instance, linear and additive trichromatic display devices can be transformed to CIE XYZ using a simple 3×3 matrix.