CS 184

Preston Fu

Spring 2024

These are course notes for the Spring 2024 rendition of CS 184, Computer Graphics and Imaging, by Prof. Ren Ng. They cover almost all of the official course slides, but with words instead of pictures. Hopefully this is slightly easier to interpret!

The course ended with some guest lectures, which I attended sparsely. If you're interested in NeRF, check out my notes for CS 294-158.

Contents

1 Introduction (01/16)

The lectures slides are here.

This lecture was an overview of the course. He introduced himself, showed some videos, and reviewed the course policies.

2 Drawing Triangles (01/18)

The lecture slides are here.

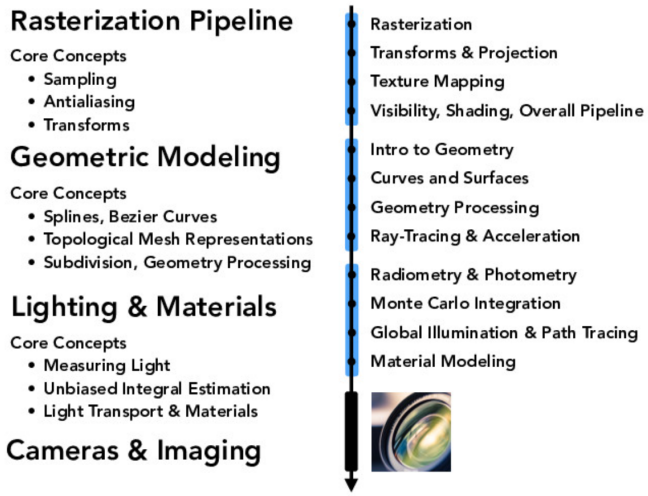


Figure 2.1. Roadmap of the (first part of the) course

The course breakdown is as follows:

- 4 homework assignments, each 12.5 points, in a team of 1-2.
- Two in-person exams, each 10 points.
- A final project, worth 25 points, in a team of 4.
- Participation, worth 5 points.

There are 8 late days for the semester, which apply only to homework assignments. (Max 4 late days on the last assignment.)

A roadmap of the course is in Figure 2.1.

Overview

In computer graphics, we have vector representations of things. We can make art with rasterization, fabrication, or use an oscilloscope with electron arrivals.

A display accepts RGB signals as inputs (frame buffer) and must output intensities (raster display). LCDs block or transmit light by twisting polarization. Alternatively, you can use an LED display, which emits light with diodes, or a DMD projection display, where mirrors get angled specifically to project light in particular directions.

Different smartphones use different screen designs. For instance, iPhone 6S uses red/green/blue in columns, while Galaxy S5 has them interleaved. Thus, there are different requirements for how light should be emitted from each diode.

Typically, polygon meshes are used as representations for 3D objects. Take, for example, a tiger. The idea is to map images onto this mesh, known as **texturing**.

Today, we will focus on constructing the mesh.

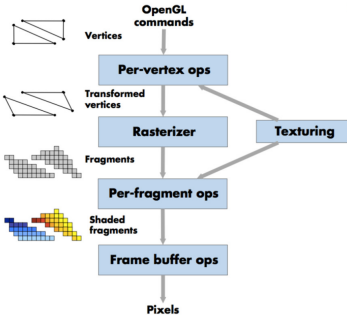


Figure 2.2. OpenGL pipeline

Idea. Specify the vertices and color of each triangle. Finally, project the result onto the screen.

There's lots of difficulties with this, such as objects appearing behind one another, or ambient lighting. This is the topic of this course.

OpenGL (a drawing machine) is an API for our purposes in this lecture; we will discuss it in more depth in future lectures (see fig. 2.2 for a high-level overview). We pass it triangles, lines, points, and images, and it outputs pixels in the frame buffer, with GPU acceleration.

Triangles

Question 2.1. Why do we use triangles?

Answer. There are lots of conceivable types of meshes. In practice, we tend to use triangles because they are the most basic polygon and are guaranteed to be planar (i.e. they define a plane, eliminating the need for specifying which triangles other polygons are constructed from). Furthermore, we can use **barycentric coordinates** to interpolate values at vertices over triangles.

Question 2.2. How do we rasterize a triangle? That is, suppose you put a triangle on the plane, where you have a lattice grid. How can you approximate a triangle with pixel values?

Answer. Use sampling, the standard method for discretizing a function. In fact, you can sample from $n \in \mathbb{N} \cup \{\infty\}$ dimensional spaces. In this case, we only care about sampling in $\mathbb{R}^2$, where you can define

$$\text{INSIDE}(\text{triangle } T; x, y) = \mathbb{1}\{(x, y) \in T\}.$$

Simply evaluate the function at $[x + 0.5, y + 0.5]$ for x, y integers in the desired range to render the image.

To actually evaluate INSIDE, observe that a triangle is the intersection of three half-planes. So you can just check that the point of interest is in the correct half-plane for each edge.

Suppose you have a line with tangent vector $\mathbf{n} = [x, y]$ and tangent vector $\mathbf{t} = [-y, x]$. The idea is that the sign of $L(\mathbf{v}) = \mathbf{v} \cdot \mathbf{n}$ will tell you whether $\mathbf{v}$ is above/on/below the line. If all three of your inequalities satisfy $L \geq 0$, you win.

Question 2.3. Suppose you have a point on an edge between two triangles. Which triangle does it belong to?

Answer. Ideally, it should belong to exactly one. It is ok but inefficient if it belongs to both. It is awful it belongs to neither (your mesh will have cracks).

Question 2.4. Suppose you have a large triangle. On a GPU, how do you efficiently rasterize?

Answer. Process cells in blocks. If the entire block is inside the triangle, then you should be able to update it very quickly.

Question 2.5. The method above will produce **jaggies**, i.e. a staircase-stepping function. If you have triangle edges with weird slopes, you'll end up with terrible irregularities. See, for instance, Figure 2.3.

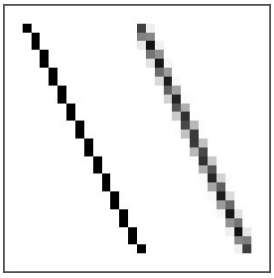


Figure 2.3. Jaggies

3 Sampling and Aliasing (01/23)

The lecture slides are here.

Much of graphics is possible with ray tracing. The idea is that for any 3D object, we can get a pixel value at any point by sampling rays to approximate an integral. However, there are problems due to sampling we need to account for.

Some common examples of artifacts due to sampling, i.e. **aliasing**, are Moiré patterns (weird pixel patterns while downsampling) and jaggies (staircase pattern). The idea of **antialiasing** is to filter out high frequencies before sampling.

Example 3.1. You have a camera that captures an image at some tiny interval of time, say $\varepsilon = 1/4000$ s. However, it operates at 60 fps. So your video camera will produce some motion blur, rather than taking just the image in that ε-second period.

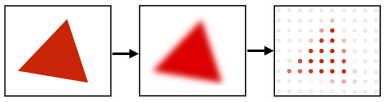


Figure 3.1. Rasterization with antialiasing

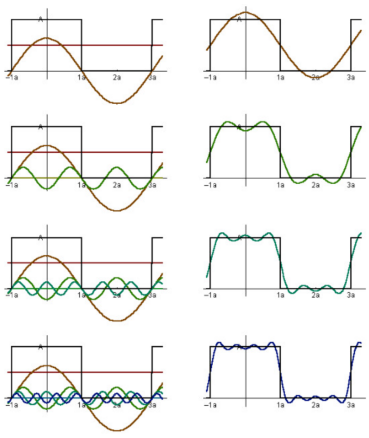


Figure 3.2. Approximating a square wave with sinusoids

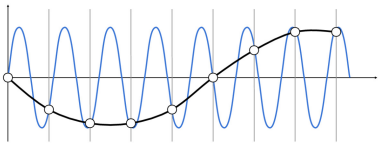


Figure 3.3. Frequency aliasing

Example 3.2. For rasterization, apply some pre-filtering to remove high frequencies before sampling. See Figure 3.1. (Doing it the other way doesn't help; that just makes blurry jaggies.)

The idea of a **Fourier transform** is that you can decompose any (nice) function as a weighted sum of sines and cosines. See, for example, Figure 3.2. Formally,

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \omega x} \mathrm{d}x,$$

and the inverse transform is

$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{2\pi i \omega x} \mathrm{d}\omega.$$

(We won't need to know the math for this class.)

The sampling problem is that undersampling creates **frequency aliases**; that is, high-frequency signals appear indistinguishable from low-frequency samples. So sampling at too low of a frequency will cause high frequencies to get aliased (which could cause any of the problems we mentioned before). See Figure 3.3.

See the slides for some examples of the correspondence between spatial and frequency domains. A key example is that the frequency domain function $\mathbb{1}\{(x,y) = (0,0)\}$ will produce a constant pixel value in the spatial domain. The important thing is that high frequencies will produce pixel variations over a smaller spatial range. For example, if you filter out low frequencies, you'll get edge detection.

Theorem 3.3. *Convolution in the spatial domain is equal to multiplication in the frequency domain, and vice-versa.*

For example, suppose you have an image and a blurring kernel you use for convolution. When you apply the kernel to the image, you'll end up with a blurry version of that image. Or you can consider this in the frequency domain, where you multiply the Fourier transform of the image and that of the kernel, then apply the inverse Fourier transform to get the same thing.

Theorem 3.4. *We get no aliasing from frequencies in the signal that are less than the Nyquist frequency (half of the sampling frequency).*

Example 3.5. Suppose you have the image $\sin(2\pi/32)x$ in the sampling domain, and you sample every 16 pixels. (So they hit alternating white/black). Then the Nyquist frequency of sampling is exactly $\frac{1}{2} \cdot \frac{1}{16} = \frac{1}{32}$, and the maximum signal frequency is $\frac{1}{32}$. So there is no aliasing.

On the other hand, if you have the image $\sin(2\pi/16)x$ and sample

every 16 pixels, you will hit only white. So you'll have aliasing. (One way to fix this is just by sampling every 8 pixels.)

Example 3.6. Gaussian blur just means multiplication by a Gaussian kernel in the frequency domain.

Combining all of this, we use the approach in Figure 3.4. The idea is that we will filter the original image to reduce the maximum signal frequency, then create a low-res image by sampling only every 16 pixels. (In general, we prefer 4, which actually has some aliasing. In practice, we use the Nyquist frequency as the cutoff to perform full anti-aliasing.)

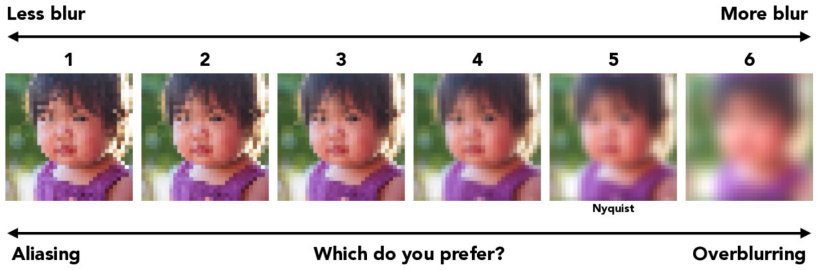


Figure 3.4. Visual example for nyquist frequencies

4 Sampling & Aliasing & Transforms (01/25)

Sampling & Aliasing

A review of last time: to reduce aliasing error, we can:

- Increase the sampling rate (and thus increase the Nyquist frequency), which also requires higher resolution
- Antialiasing: remove signal frequencies above the Nyquist before sampling

A classic example of a pre-filter is a 1 pixel-width box filter, which attenuates frequencies with period at most 1 pixel-width.

Observe the equivalence between the following two approaches:

- Pre-filter by a 1-pixel box-blur, then sample per pixel
- Compute the average value in the pixel (for instance, if you have something on an edge or vertex, the pixel average value will be in $(0,1)$)

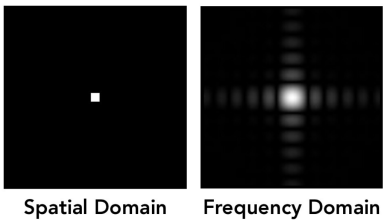


Figure 4.1. One pixel box filter

To accomplish the latter approach, we will **supersample**. That is, we will render a higher-resolution image of the triangle, where in each pixel you take $N \times N$ samples. Of course, this approach is very costly (scales with N^2).

This approach still produces aliasing, though, because we saw last time that with this method of filtering, we still get frequencies out to infinity. See Figure 4.1.

Question 4.1. How should you sample: Uniformly? Use samples inside or outside the picture? How do you supersample multiple triangles?

Remark. A popular stress test is $\sin(x^2 + y^2)$, where you'll end up with some under-sampled and over-sampled points.

Transforms

The lecture slides are here.

There are lots of different types of transforms, e.g. scaling, rotations, and translations. We need them to model coordinates in a convenient representation, enable multiple copies of the same objects, and efficiently represent hierarchical scenes. To display 3D scenes, we need to convert from world to camera coordinates, then project onto a 2D screen.

To represent translations as a linear map in 2D, just augment 1 to your coordinate system. That is, your scaling, rotation, and translation matrices are shown (respectively in Figure 4.2.

$$S(s_x, s_y) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$T(t_x, t_y) = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 4.2. Transformations in 2D

Question 4.2 (Common exam question)**.** How can you decompose a complex translation into the simple methods we described above?

You can use the same thing in 3D. To rotate about an arbitrary axis, you construct an orthonormal frame transformation F, then run FRF^{-1} where R denotes the desired rotation where you line up the axes.

For example, if you'd like to rotate about an axis, your matrices will look like those shown in Figure 4.3.

$$R_x(\alpha) = \begin{pmatrix} 1 & & & \\ & \cos\alpha & -\sin\alpha & \\ & \sin\alpha & \cos\alpha & \\ & & & 1 \end{pmatrix}$$

$$R_y(\alpha) = \begin{pmatrix} \cos\alpha & & \sin\alpha & \\ & 1 & & \\ -\sin\alpha & & \cos\alpha & \\ & & & 1 \end{pmatrix}$$

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & & \\ \sin\alpha & \cos\alpha & & \\ & & 1 & \\ & & & 1 \end{pmatrix}$$

Figure 4.3. Transformations in 3D

5 Transforms & Texture Mapping (01/30)

Transforms

The lecture slides are here.

The idea behind a hierarchical representation is that you can model a rigid body as consisting of multiple parts. When you transform each part, other parts will move with it. For example, if you have a skeleton and apply a transformation to someone's arm, then their hand will move with it. You can just move the entire thing together, rather than worrying about each individual thing separately. See, for example, Figure 5.1.

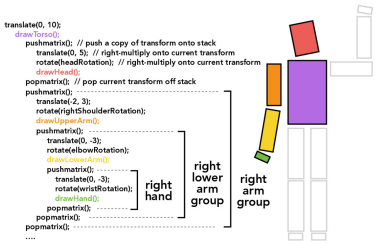


Figure 5.1. Hierarchical transforms

We capture the world through cameras, so we care a lot about camera space. Typically, we will use the following setup: a camera is looking down the (negative) $-z$ axis, up is the y axis, and right is the x axis. The result is that in any picture through a camera, the center of the image is the origin, the x and y axes are natural, and the z axis is out of the page (away from the scene).

Equivalently, you can model this in the real world as follows: $\mathbf{e}$ represents the position of the camera, $\mathbf{u}$ represents up, and $\mathbf{v}$ represents the view direction (where the camera is facing). One can check that the matrix to convert from camera to world space is

$$\text{C2W} = \begin{pmatrix} \mathbf{r} & \mathbf{u} & -\mathbf{v} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \text{with inverse} \quad \text{W2C} = \begin{pmatrix} \mathbf{r}^\top & 0 \\ \mathbf{u}^\top & 0 \\ -\mathbf{v}^\top & 0 \\ \mathbf{0}^\top & 1 \end{pmatrix} \begin{pmatrix} I_3 & -\mathbf{e} \\ \mathbf{0}^\top & 1 \end{pmatrix}.$$

Now, the task remains to project from 3D to 2D. The difficulty here is perspective (which wasn't understood in art for the longest time!). This was only properly figured out in *Delivery of the Keys* by Perugino in 1491; see Figure 5.2.

Figure 5.2. *Delivery of the Keys*

Eventually, people figured out that a camera can work similarly to the eye: if you have a pinhole in a sheet, then its projection onto another sheet will be an inverted copy of the same thing. Suppose you have a camera at the origin, and an object is at $[x, y, z]^\top$. Then the projection onto the plane $z = d$ is the mapping $[x, y, z] \mapsto [x \cdot d/z, y \cdot d/z, d]^\top$. To get this transformation in homogeneous coordinates, multiply by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}.$$

Figure 5.3. The person is in the same place, but the lens is changing.

Remark. Division is hard to do on a chip. People have tried working out alternatives, but this is the best we've been able to do. We isolate the division to this particular matrix.

The parameters for perspective really do matter! See Figure 5.3.

To specify how the perspective works, there are a couple parameters; see Figure 5.4. The idea is that you can specify fovy (vertical angular field of view), aspect ratio (width-height ratio), near (depth of the near clipping plane), and far (depth of the far clipping plane). From this, you can derive: $\text{top} = \text{near} \cdot \tan(\text{fovy})$, $\text{bottom} = -\text{top}$, $\text{right} = \text{top} \cdot \text{aspect}$, and $\text{left} = -\text{right}$.

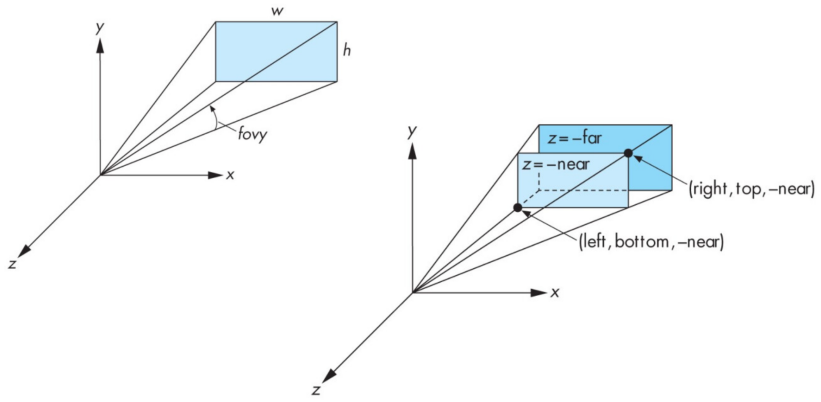


Figure 5.4. Parameterizing perspective viewing volume

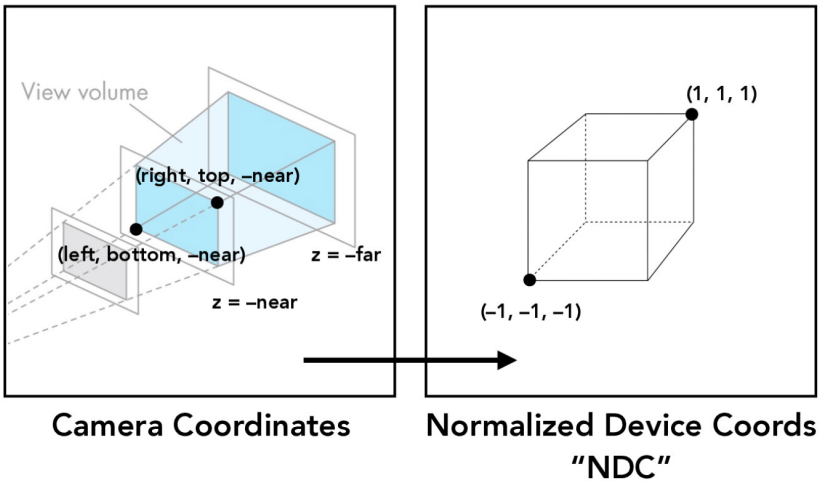


Figure 5.5. Conversion between camera and world coordinates

$$\mathbf{P} = \begin{pmatrix} \frac{\text{near}}{\text{right}} & & & \\ & \frac{\text{near}}{\text{top}} & & \\ & & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} & \frac{-2\text{far}\cdot\text{near}}{\text{far}-\text{near}} \\ & & -1 & \end{pmatrix}$$

Figure 5.6. Perspective transform matrix

Suppose we want to convert from perspective to the real world. We can apply the perspective transform matrix given in Figure 5.6.

For instance, once can verify that

$$\mathbf{P}\begin{pmatrix}\text{right} & \text{top} & -\text{near} & 1\end{pmatrix}^\top = \begin{pmatrix}1 & 1 & -1\end{pmatrix}^\top,$$

as in Figure 5.5.

Concept. The pipeline is as follows:

- To convert from object coordinates to world coordinates, use a modeling transform.
- To convert from world coordinates to camera coordinates, use a viewing transform.
- To convert from camera coordinates to normalized device coordinates (NDC), use a projection transform and homogeneous division.
- To convert from NDC to screen coordinates, use a screen transform.
- To display the object with screen coordinates, use rasterization.

Texture

The lecture slides are here.

The motivation for texture mapping is depicted in Figure 5.7, or perhaps as a world map. A surface lives in 3D space, and every point (x, y, z) has a corresponding location (u, v) in a 2D texture map.

Figure 5.7. Chocolate wrapper as an analogy for textures

In practice, you'll have some triangle mesh representing your 3D object, and each vertex in the triangle will be associated with some color. To interpolate between points inside the triangles, we use barycentric coordinates: points inside $\triangle ABC$ can be represented as $\alpha A + \beta B + \gamma C$ for $\alpha + \beta + \gamma = 1$. You then interpolate the value V at any point in a triangle in exactly the same way, $\alpha V_A + \beta V_B + \gamma V_C$.

6 Texture Mapping (02/01)

The lecture slides are here.

Last lecture, we discussed barycentric coordinates, and discussed the conversion from barycentric to cartesian coordinates. Now, we will do the other direction.

The key observation is that $(x, y) = Z = \alpha A + \beta B + \gamma C$ satisfies the property that the

$$\frac{\text{dist}(Z, BC)}{\text{dist}(Z, A)} = \frac{\alpha}{1 - \alpha}.$$

The idea, then, is to derive

$$L_{PQ}(x) = -(x - x_P)(y_Q - y_P) + (y - y_P)(x_Q - x_P) \propto \text{dist}(x, PQ).$$

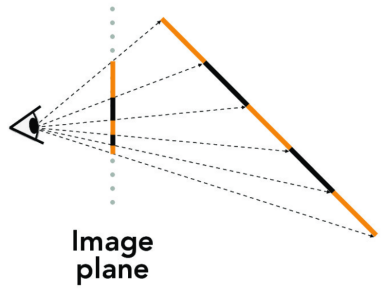


Figure 6.1. Perspective nonlinearity

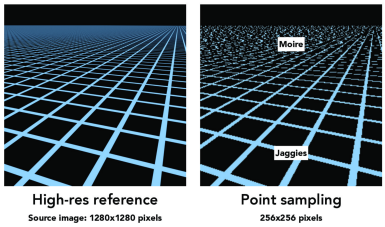


Figure 6.2. Naively sampling points produces aliasing

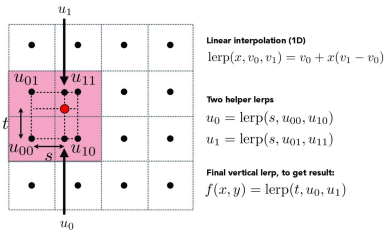


Figure 6.3. Bilinear filtering

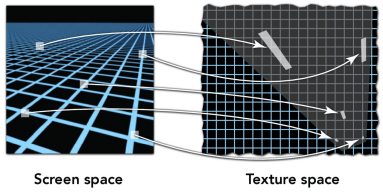


Figure 6.4. Correspondence between screen space and texture space

The derivation was discussed in the second lecture; note that this distance is signed, depending on the relative position of X and PQ. Then

$$\alpha = \frac{L_{BC}(x, y)}{L_{BC}(x_A, y_A)}.$$

Barycentric interpolation is not sufficient; we also have to apply a projection transform, where we scale everything by their depth. (Division is cheap now, so this isn't as much of a problem as it was before.) See Figure 6.1; the idea is that, otherwise, evenly-spaced objects, when projected onto a non-parallel image plane, will become unevenly spaced in the output.

Concept. Applying textures is sampling. The idea is to model samples with some discrete function $f(x, y)$, and to reconstruct a continuous 2D function f_{cont} with some convolution filter $k(x, y)$. Then you can draw your desired sample by evaluating f_{cont} at your point (u, v).

A naive approach looks like Figure 6.2. In this case, we maintain both a pixel space and a texture (texel) space. (We previously discussed this in the 3D case.) When you magnify an image, you upsample from texels, and when you minify, you downsample. Ideally, you will have a 1:1 scale mapping between pixels and texels.

There are many types of filters, e.g. Figure 6.3, where lerp stands for **l**inear int**erp**olation. Some other approaches are nearest-neighbor (just use the pixel value at the closest one; makes it very blocky) and bicubic (fit a cubic to a 4×4 grid; potentially computationally intensive).

The issue with this approach is shown in Figure 6.4; the texture sampling pattern is in fact not rectilinear or isotropic (recall the required transformations from last lecture), and furthermore we want some bounds on the frequency to avoid aliasing. However, we'd like to do this efficiently, where we have $O(1)$ texel lookups per pixel.

The idea here is to take a texture map, apply a low-pass filter, and downsample it. Do this recursively, where at each step you have successively lower maximum signal frequencies. For each sample, we use the texture file whose resolution approximates the screen sampling rate. This approach is known as **mipmap**.

Example 6.1. If you have a 128×128 image (we call this mipmap level 0), this is equivalent to downsampling to a 64×64 image (level 1) by averaging 2×2 blocks, as we discussed in the lecture on frequency domain. Now repeat this for 32×32 (level 2), and so on.

To compute D per pixel, we use the approach of Figure 6.5, where we estimate derivatives using a discrete approximation (take the difference

across neighboring cells), and take the max to avoid aliasing in both dimensions. See Figure 6.6 for a sample visualization of mipmap levels. One technicality is that D need not be an integer, so the idea of **trilinear filtering** is to run bilinear filters on levels $\lfloor D \rfloor$ and $\lceil D \rceil$, and linearly interpolate between the two.

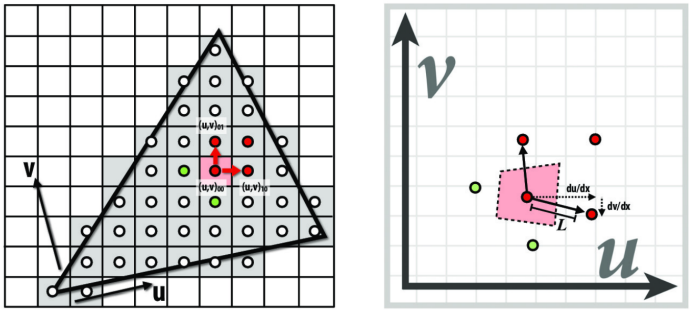
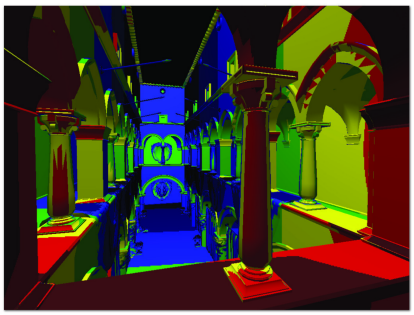


Figure 6.5. Computing mipmap level D

$$D = \log_2 L$$

$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$



D rounded to nearest integer level

Figure 6.6. Example of D in an image, rounded to the nearest integer

7 Texturing & Rasterization & Geometry (02/06)

Advanced texturing methods

The lecture slides are here.

Suppose you have a reflective object, and you want to map a texture onto it. The idea is to consider what direction light will reflect off the object with respect to the camera, and you can map the resulting point in texture coordinates onto the object. See Figure 7.1.

Lat / long texture map | Reflection vector indexes into texture map

Figure 7.1. Mapping a texture via reflection

Recall that we previous mapped points onto a surface by fitting a triangular mesh. This is fine but inefficient: previously, we discussed that points near the north and south poles on a sphere get assigned lots

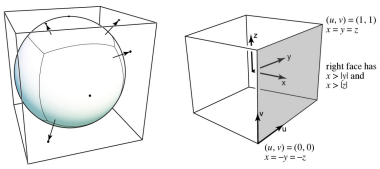


Figure 7.2. Cube map

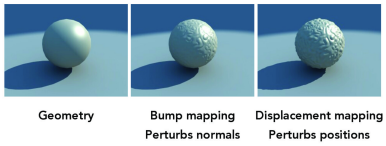


Figure 7.3. Bump map

Figure 7.4. I'm hungry

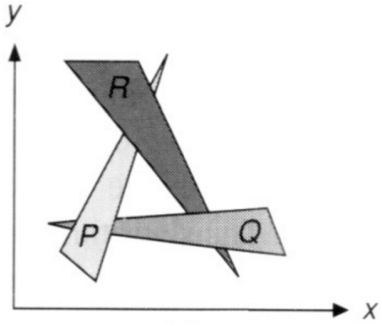


Figure 7.5. Unresolvable depth order

of triangles unnecessarily. The idea is to use a **cube map**, where you map the same direction on a sphere as a cube. See Figure 7.2.

Another hack for representing bumpy surfaces is to use a **bump map**, which perturbs the texture to the surface normal. That is, the normal vectors get displaced a little bit, so that the colors inside the pixels get changed. The **displacement mask** alters the geometric structure of the mesh, i.e. the *positions* get perturbed.

Another method is called **Perlin noise**, where you apply n-dimensional procedural noise and model solids. (For instance, this is used in Minecraft.) There are lots of algorithms here; see Wikipedia for more.

Another method is volume rendering, which is used in NeRF.

The rasterization pipeline

Suppose you have some objects rotating in space. Once the front object (A) rotates behind the back object (B), the frame buffer gets overwritten, and the pixels at the overlap will correspond to A, when really they should correspond to B, which is now in the front.

There's lots of thigs to be improved here. Ultimately, our goal is to get to something like Figure 7.4. There's lots of impressive features here despite simulation, in particular how light it diffuses off the napkin, refracts through the glass, demonstrates the bumpiness in the utensils and smoothness of the mug, and vision/camera-like blur.

We'll work up to this through several parts:

Visibility Suppose you have many objects and want to figure out which ones are visible from the perspective of the camera.

Naively, we might want to use a painter's algorithm, i.e. paint objects back-to-front. This doesn't work directly; see Figure 7.5. One way to get around this is to triangulate small enough, then sort objects in order, but this is quite inefficient (for n triangles, you need $O(n \log n)$).

In practice, we like to use a **Z-buffer**, where you store the current z-value for each sample position. In addition to RGB, you add an additional buffer for depth, which is 16 to 32 bits. When sampling, you take the smallest z-coordinate corresponding to any (x, y). See Figure 7.6. The performance is great: for n triangles, runtime is $O(n)$.

Simple shading We want to achieve characteristics of light indicated in Figure 7.7. Let's do some physics!

Initialize depth buffer to ∞

During rasterization:

```
for (each triangle T)
  for (each sample (x,y,z) in T)
    if (z < zbuffer[x,y])          // closest sample so far
      framebuffer[x,y] = rgb;      // update color
      zbuffer[x,y]     = z;        // update z
    else
      ;     // do nothing, this sample is not closest
```

Figure 7.6. Z-buffer algorithm

Suppose we have the following setup: a surface has a normal vector $\mathbf{n}$, a viewer is in the direction $\mathbf{v}$, and a light source is in the direction $\mathbf{l}$ (for simplicity, all of these vectors have unit norm). The surface has some parameters, such as color, shininess, etc.

In **diffuse reflection**, we assume that light is scattered uniformly in all directions on the same side as the surface. That is, the contribution to surface color is not dependent on $\mathbf{v}$, subject to a direction indicator. **Lambert's cosine law** gives that the light per unit area is proportional to $\cos\theta = \mathbf{l} \cdot \mathbf{n}$, where θ is the angle between $\mathbf{l}$ and $\mathbf{w}$. See, for instance, Figure 7.8.

Suppose you have a light source with intensity I. Distance r from the light source, the intensity is I/r^2. Putting this all together,

$$L_d := k_d(I/r^2)\max(0, \mathbf{n} \cdot \mathbf{l}),$$

where k_d is a diffuse constant.

Specular shading is intense when the viewer is near the reflection of the light source off the surface (i.e. the reflection of $\mathbf{l}$ over $\mathbf{n}$). To measure "near", we will define $\mathbf{h}$ as the angle bisector between $\mathbf{v}$ and $\mathbf{l}$, and use dot product as a matrix. The model (which, unfortunately, is more of a hack than physically inspired) is given by Figure 7.9.

Finally, to accomplish **ambient shading**, for now we will represent this using a constant $L_a = k_a I_a$, and come back to this in another lecture.

The **Blinn-Phong reflection model** aggregates these methods,

$$\begin{aligned} L &= L_a + L_d + L_s \\ &= k_a I_a + k_d(I/r^2)\max(0, \mathbf{n} \cdot \mathbf{l}) + k_s(I/r^2)\max(0, \mathbf{n} \cdot \mathbf{h})^p. \end{aligned}$$

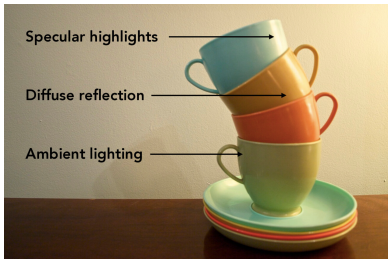


Figure 7.7. Pereptual observations

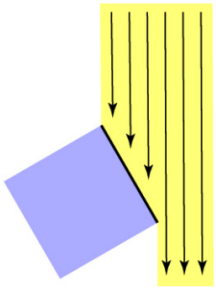


Figure 7.8. Example of Lambert's cosine law

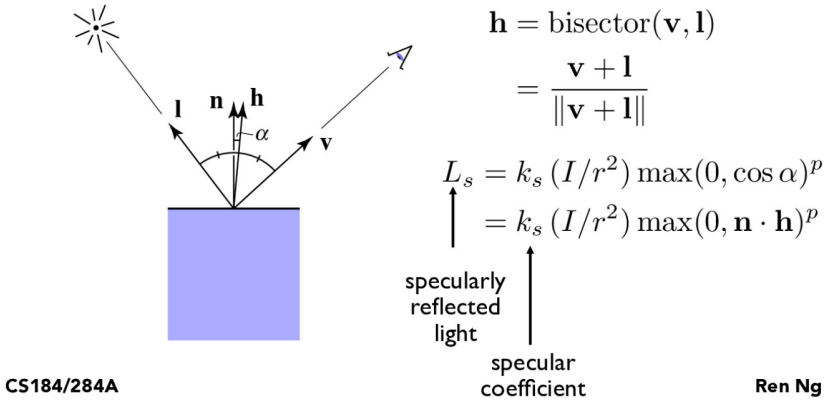


Figure 7.9. Blinn-Phong model for specular shading

Shading triangle meshes There are several methods for shading a triangle mesh: triangle (flat), vertex (Gouraud), and pixel (Phong). In practice, we like to use vertex normals, where you can either use the underlying geometry to acquire a normal, or interpolate based on the incident faces. Then you can get per-pixel normal vectors through barycentric interpolation.

Rasterization pipeline The pipeline is depicted in Figure 7.10.

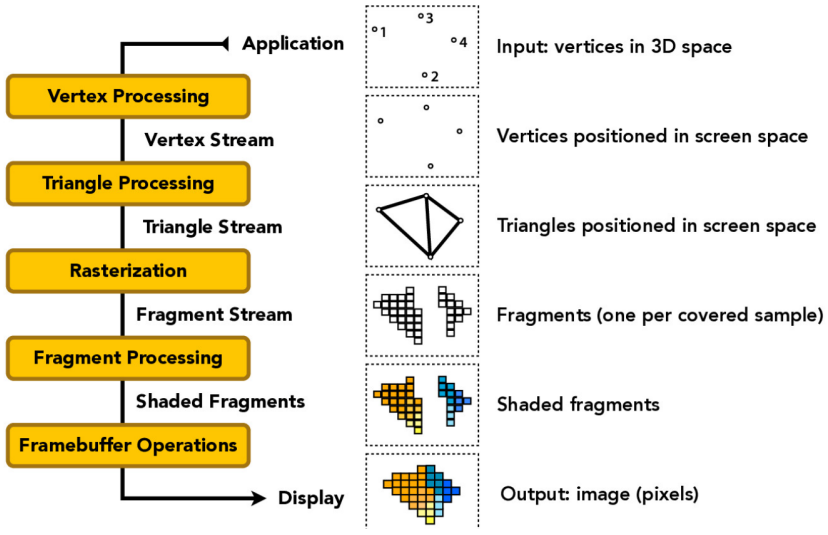


Figure 7.10. Rasterization pipeline

Introduction to geometry

Suppose you have some points and want to interpolate them. The typical approach in graphics is with **cubic hermite interpolation**. The prob-

lem setup is as follows: you are given $h_0 = P(0), h_1 = P(1), h_2 = P'(0)$, and $h_3 = P'(1)$ (hence the choice of a cubic, since there are three parameters). Letting $P(x) = ax^3 + bx^2 + cx + d$, it is easy to check that

$$\begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}.$$

In fact, this matrix is invertible:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}.$$

8 Geometry (02/08)

The lecture slides are here.

Last time, we discussed using cubics to interpolate between any two points.

Catmull-Rom interpolation works similarly: suppose you have four points, $\{(x, y)_i\}_{i=0}^3$. The algorithm works by interleaving, i.e. approximate the slope at x_1 by the secant between (x_0, y_0) and (x_2, y_2), and likewise approximate the slope at x_2 by the secant between (x_1, y_1) and (x_3, y_3). Then for every two neighboring points, fit a cubic to their y and $\frac{dy}{dx}$ values.

The same principle works in $\mathbb{R}^n$: just fit vectors instead of values. In general, we can define interpolation in terms of basis functions for the interpolation scheme:

$$\mathbf{p}(t) = \sum_{i=0}^{n} \mathbf{p}_i F_i(t).$$

For instance, we've already looked at the **Hermite interpolation scheme** $\{H_i(t)\}$ earlier; we will derive $C_i(t)$ for Catmull-Rom and $B_i(t)$ for Bézier curves.

The inputs will be

$$h_0 = p_1, \quad h_1 = p_2, \quad h_2 = \frac{1}{2}(p_2 - p_0), \quad h_3 = \frac{1}{2}(p_3 - p_1).$$

To implement this, just use the same format,

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \mathbf{H} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix},$$

where **H** denotes the Hermite interpolation matrix.

Our methods above have interpolated by forcing curves through the data. In practice, this might not be a good choice because real-world data is noisy. The idea of **Bézier curves** is to define some "control points" $p_{0:3}$ and define a curve that passes through p_0 and p_3 and define the rest with tangent vectors $t_0 = 3(p_1 - p_0)$ and $t_1 = 3(p_3 - p_2)$.

Exercise 8.1. Derive the Bézier matrix.

Remark. In graphic design, the control points are the things you drag to specify curves. To make smooth curves, you constrain control points to be collinear with the vertex. To make sharp points, move them off the line.

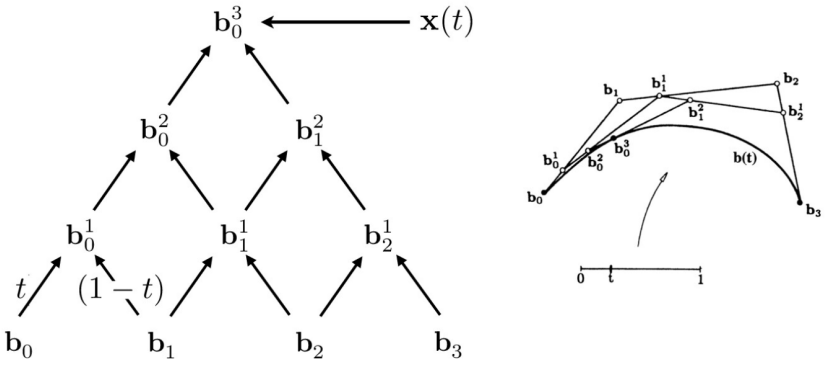


Figure 8.1. A computation graph for the de Casteljau algorithm

The motivation for Bézier curves is given by the **de Casteljau Algorithm**. Let

$$\begin{aligned} b_0^1 &= (1-t)b_0 + tb_1 \\ b_1^1 &= (1-t)b_1 + tb_2 \\ b_0^2 &= (1-t)b_0^1 + tb_1^1, \end{aligned}$$

i.e. repeatedly take a point proportion t along the way from each segment until you can no longer do so (run linear interpolation in succession). Then the Bézier curve passes through it. This algorithm is much more intuitive than the tangent line approach and generalizes easily to an arbitrary number of points!

The algorithm is effectively Pascal's triangle; see Figure 8.1. We can evaluate the **Bernstein form** of a Bézier curve of order n, namely

$$\mathbf{b}^n(t) = \mathbf{b}_0^n(t) = \sum_{j=0}^{n} \mathbf{b}_j \underbrace{\binom{n}{j} t^j (1-t)^{n-j}}_{B_j^n(t)},$$

where $\mathbf{b}_j$ are the control points.

Bézier curves work well in low dimensions, but in high dimensions it becomes very hard to control (high-order polynomials become much more sensitive to changes in coefficients, and naturally may be numerically unstable from a user perspective). In practice, we like to chain together may low-order Bezier curves.

Note. In the last remark, we said that the way to enforce C^0 continuity is to constrain points to match. To enforce C^1 continuity is to constrain points to collinear and equidistant from the vertex. To enforce C^2 continuity, use an A-frame construction. See Figures 8.2 to 8.4.

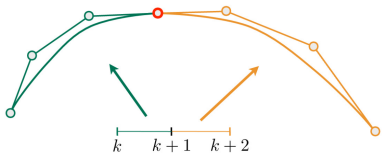


Figure 8.2. Enforcing continuity constraints on Bézier curves: C^0, C^1, and C^2 cases, respectively

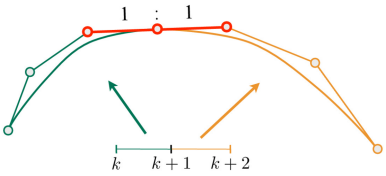


Figure 8.3. C^1 case

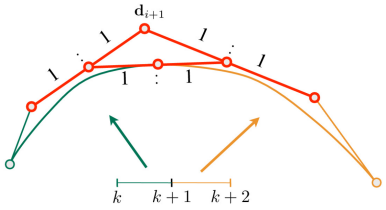


Figure 8.4. C^2 case

Bézier curves satisfy the following useful properties:

- **Affine transformations**: you can transform the curve by transforming the control points.
- **Convex hull**: the curve lies within the convex hull of the control points.

Likewise, higher-dimensional surfaces can be approximated by composing several Bézier surfaces, which we call patches. Now we will derive this multivariate case. The idea is to just do the computation one dimension at a time: produce Bézier curves in the x direction, and use the resulting points as controls for the y direction. That is, just take the outer product:

$$\mathbf{b}^{m,n}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{n} \mathbf{b}_{i,j} B_i^m(u) B_j^n(v).$$

9 Mesh Representations and Geometry Processing (02/13)

The lecture slides are here.

Last time, we discussed that we might be interested in upsampling, downsampling, or regularizing a triangle mesh. In this lecture, we will discuss representations of triangles that allow us to implement this.

In general, we will use an implementation according to Figure 9.1, where we store the coordinates corresponding to each vertex, and the

vertices corresponding to each triangle. This avoids redundant information, and we can efficiently move a vertex and accordingly move its containing triangles.

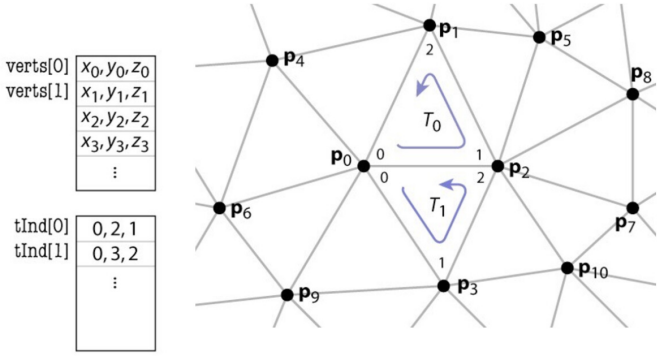


Figure 9.1. Indexing triangles

Topology and **geometry** are depicted in Figure 9.2. The geometry refers to the shape of the object, while topology refers to the arrangement of the triangles. In general, it is much easier for topology to generalize to different geometries: one idea to interpolate between two different geometries with the same topology is to linearly interpolate the triangles.

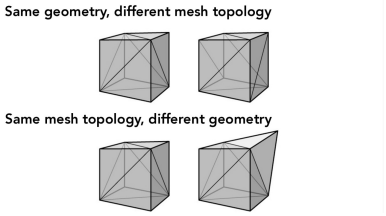


Figure 9.2. Topology vs. geometry

For our purposes, a **manifold** is a surface that, when cut with a sphere, always yields something topologically equivalent to a disk. See Figure 9.3. Assuming that every surface we care about to be a manifold, our algorithms become much cleaner. In particular, we can regard them as polyhedra and apply similar intuition.

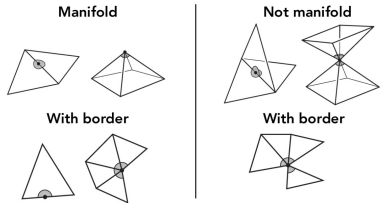


Figure 9.3. Definition of a manifold

Furthermore, we will assume that we are dealing with orientable structures. That is, each triangle can be assigned a consistent winding order determining the front and back. For example, a polyhedron is orientable, while a Möbius strip is not.

Typically, we care about the following data structures:

```
Triangle:
  - Stores the vertices and neighboring triangles

Vertex:
  - Stores the vertex's location and incident triangles

Half-edge:
  - Stores its "twin" half-edge, the "next" half-edge around the face, the
    ↪ start vertex, the edge, and the face (see Figure 9.4)
```

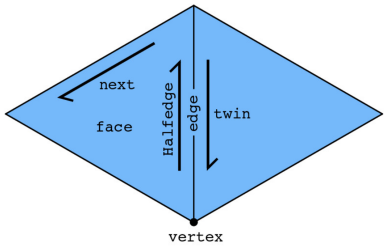


Figure 9.4. Specifications for `Half-Edge`

The benefit of this approach is that we can easily traverse edges around a face, or edges around a vertex. By iterating, we can traverse a mesh.

There are several kinds of base mesh editing operations, i.e. flip, split, and collapse edges, where you just update every relevant pointer.

These operations will be the basis for larger operations.

The idea for **Loop subdivision** is that for every triangle, split it into four parts, and assign new vertex positions according to weights. For vertices with degree n, assign weight

$$u = \begin{cases} 3/16 & n = 3 \\ 3/(8n) & \text{else} \end{cases}$$

to each incident vertex and $1 - nu$ to the new vertex.

Then, split each edge arbitrarily, and connect the edges. Finally, flip each new edge incident to both a new and old vertex. See Figure 9.5.

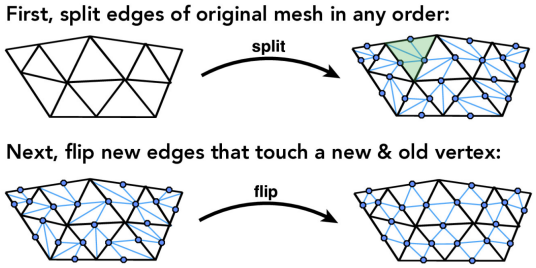


Figure 9.5. Edge operations in loop subdivision

Remark. In most meshes, every vertex will have degree 6. One can use Euler's formula to show that if a mesh is topologically equivalent to a sphere, there must exist points with degree not equal to 6.

The **Catmull–Clark algorithm** does subdivision for quad meshes. See the slides for more details; like Loop, there is no derivation on the slides, so it is just a matter of implementation.

10 Meshes & Ray Tracing (02/15)

Mesh simplification

The lecture slides are here.

Last time, we talked about the basic mesh resampling functions: splits for local upsampling, collapse for local downsampling, and flips for local resampling. Our objective is to this on a larger scale, i.e. reducing the number of mesh elements while maintaining the overall shape.

Let's quantify this. Let $u = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^\top$ be a point in homogeneous coordinates and $ax + by + cz + d = 0$ be a plane parameterized by $v = \begin{bmatrix} a & b & c & d \end{bmatrix}^\top$. The idea is that the signed distance between u and the plane can be written as $ax + by + cz + d = u^\top v$, so the squared

distance can be written as $u^\top vv^\top u = u^\top Qu$, where $Q = vv^\top$ is called the **quadric error matrix**.

The quadric error at a vertex as a proxy for how much the local geometry differs from a simpler representation. In math, it is the sum of the quadric distances to each adjacent triangle's planes, or equivalently the quadric distance that arises when the adjacent error matrices are summed. Accordingly, a greedy algorithm to determine which edges to collapse is to minimize the quadric error at the midpoint across all edges. See Figure 10.1.

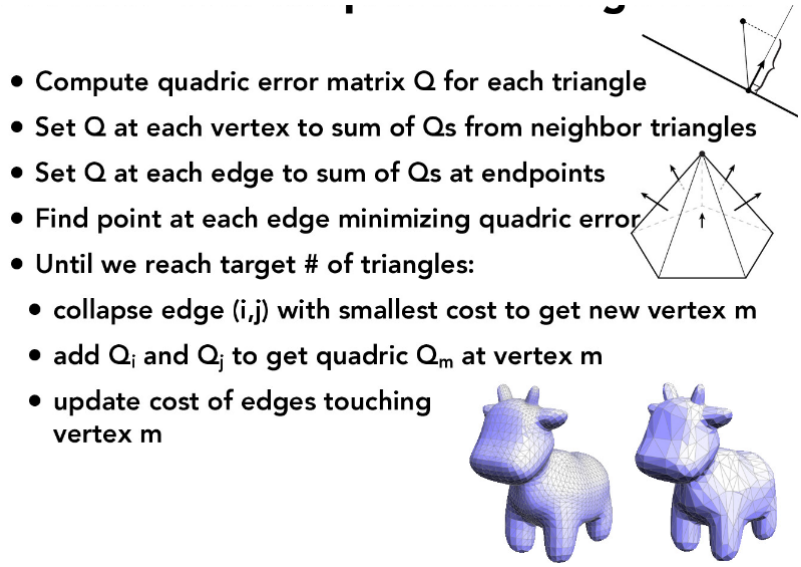


Figure 10.1. Quadric error algorithm

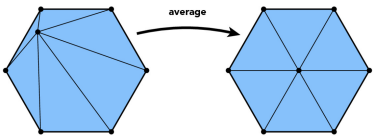


Figure 10.2. Centering vertices

Mesh regularization

It is often the case that with the greedy algorithm alone, we end up with "bad" meshes. To fix this, we regularize, i.e. try to make our triangles "round" (equilateral-like) rather than narrow. Some examples of common conditions for a regular mesh are:

- The interior of the circumcircle of any triangle contains no other vertices (forces acute)
- Every vertex should have degree close to 6. (Flip edges until this is the case.)
- Triangles might be unnecessarily irregular, e.g. Figure 10.2. To fix this, just move each vertex to the average of its neighbors.

  A remeshing algorithm combines all of these tricks.

Algorithm 1. Isotropic remeshing algorithm

while true **do**
  split edges with length $> \frac{4}{3}\times$ the mean edge length
  collapse edges with length $< \frac{4}{5}\times$ the mean edge length
  flip edges to improve vertex degree
  center vertices
end while

Ray tracing

The lecture slides are here.

Our discussion on geometry has largely concerned representing geometric shapes. In graphics, we care about how we use the geometry to produce images. In Figure 7.4, we indicated some desirable properties of images. One property we haven't discussed yet are shadows, among other things.

The idea of ray tracing is to cast a ray per pixel and to check for shadows by sending a ray to the light. The benefit of this approach over previous work because it enables a more generalizable approach to light. For instance, we previously discussed z-buffer for rasterization. Here, we can determine depth by marching along the ray, and this allows us to determine the effects of specular reflection, refraction, and so on by tracing rays recursively until you hit a non-mirror or upon hitting an upper bound.

Example 10.1. To determine whether a ray hits a triangle, we will determine the ray's intersection with the containing plane, then determine whether the intersection is in the triangle using the method from homework 1.

Parameterize a ray by $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ for $t \geq 0$. Furthermore, we parameterize a plane by its normal vector $\mathbf{n}$ and a point $\mathbf{p}'$ on the plane. The plane is then $\{\mathbf{p} : (\mathbf{p} - \mathbf{p}') \cdot \mathbf{n} = 0\}$. The intersection between the ray and the plane occurs at

$$t = \frac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

if $t \geq 0$, or does not exist otherwise.

Example 10.2. The same method is compatible with spheres, which we can parameterize as $\{\mathbf{p} : (\mathbf{p} - \mathbf{c})^2 - R^2 = 0\}$. You'll end up with a quadratic in t, which could have $\{0, 1, 2\}$ roots.

Example 10.3. We might have a surface $\{\mathbf{p} : f(\mathbf{p}) = 0\}$. As long as you can solve for the roots of f efficiently, the same approach generalizes.

Naively, ray tracing involves determining intersection points between each ray and each triangle. An algorithm with complexity $O(PT)$, where $P = \#$ pixels is on the order of a million and $T = \#$ triangles is on the order of a billion, is very slow, and we require acceleration to get this to work well in practice.

One idea is to use bounding boxes: use a bounding box around triangles. If a ray doesn't intersect a box, then there is no need to check any of the triangles inside the box. We can check intersections using the algorithm in Figure 10.3: for axis-aligned boxes, compute the t_{min} and t_{max} corresponding to each dimension, and take the max and min respectively to get the desired values for the multi-dimensional box.

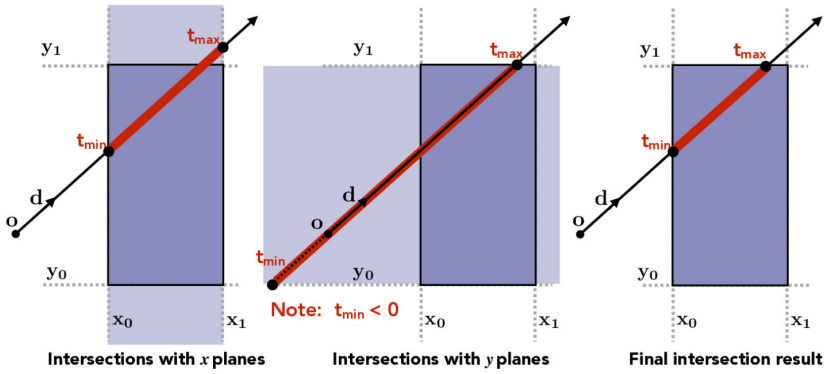


Figure 10.3. Intersection points between ray and boxes

A naive approach is to use bounding boxes as a uniform grid. To do this, check the first cell that the ray hits, then take the nearest intersection point among objects within that cell. Unfortunately this doesn't work: see Figure 10.4, where the ray hits the dark blue square, which points to the circle on the right, when really we care about the one on the left.

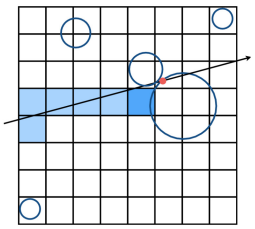


Figure 10.4. A problem with uniform grid cells

A more general approach is to use a hierarchical representation, where a partition of the image can be represented as a tree structure (see Figure 10.5). In this class, we will discuss a specific algorithm for this, **KD-trees**.

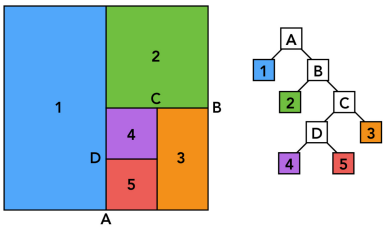


Figure 10.5. Spatial hierarchy

11 Ray Tracing & Radiometry and Photometry (02/20)

Ray Tracing and Acceleration

The lecture slides are here.

The premise of a KD tree is the hierarchical representation we discussed last time. The algorithm is to choose an axis to split on (typically

we will alternate between x and y), and split on a plane along that axis at the median of the corresponding points.

To make our data structure useful, we must specify an algorithm to trace a ray. We illustrate the recursive step in Figure 11.2 and a demonstration of the algorithm in Figure 11.1. Intuitively, KD-trees trace the ray by moving $t_{\min}$ and $t_{\max}$ down along the tree. If the ray does not intersect the splitting plane, we can eliminate half of the space. Otherwise, we need to check intersections for both sides of the tree.

Algorithm 2. Ray tracing with KD tree

```
function RAYTRACE(ray r, KD-Tree T)
    if T is empty or r misses T.bbox then
        return null
    end if
    if T is a leaf then
        return INTERSECTOBJECTS(r, T.objects)
    end if
    if r intersects T.splitting_plane then
        near ← T.left
        far ← T.right
    else
        near ← T.right
        far ← T.left
    end if
    P_near ← RAYTRACE(r, near)
    P_far ← RAYTRACE(r, far)
    return CLOSER(P_near, P_far)
end function
```

Another approach is to use object partitions or **bounding volume hierarchies (BVH)**. (We'll implement this for homework!) Instead of partitioning spatially (i.e. partition your space into non-overlapping regions), we are partitioning our set of objects into disjoint subsets of objects. Analogously, while KD trees can result in objects in multiple regions, BVH can result in overlapping bounding boxes.

The algorithm is conceptually simple: compute the bounding boxes of each triangle, and recursively split into two subsets. This likewise produces a tree structure, and you stop when there are a few objects in each step. The pseudocode is shown in Figure 11.3.

Remark. It looks like we're traversing the whole tree. In practice, this is not the case: since we are partitioning spatially, only logarithmically many elements will make it past the `ray misses node.bbox` line. We do need to check both boxes, for the same reason as in Figure 10.4,

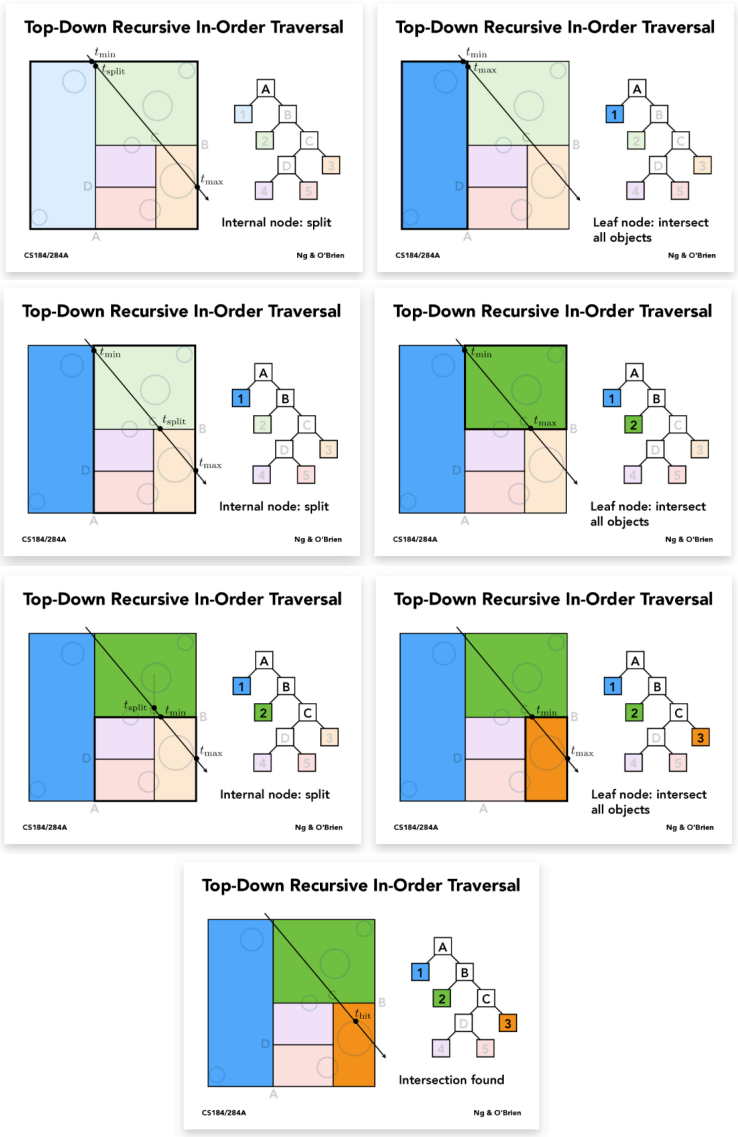


Figure 11.1. KD tree algorithm for ray tracing

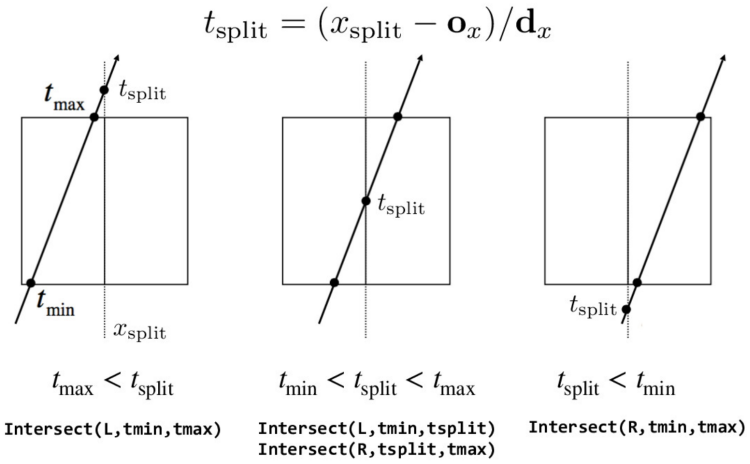


Figure 11.2. KD tree recursive step

due to overlapping bounding boxes.

```
Intersect (Ray ray, BVH node)
  if (ray misses node.bbox) return;
  if (node is a leaf node)
     test intersection with all objs;
     return closest intersection;
  hit1 = Intersect (ray, node.child1);
  hit2 = Intersect (ray, node.child2);
  return closer of hit1, hit2;
```

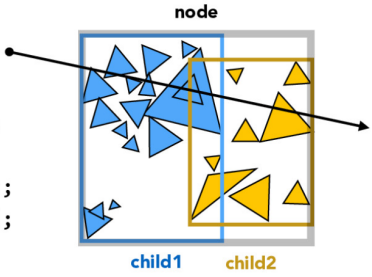


Figure 11.3. BVH pseudocode

In KD trees, we chose to split on the median. For BVH, we have to decide on how to make a good split. For each node, its cost is

$$C_{\text{trav}} + \mathbb{P}(\text{hit } L) \cdot C_L + \mathbb{P}(\text{hit } R) \cdot C_R,$$

where C_{trav} is the cost of traversal[1] and C_L and C_R are the costs of traversing its left and right children.

Assuming a uniform prior, for convex A contained in convex B, we can define

$$\mathbb{P}(\text{hit } A \mid \text{hit } B) = S_A / S_B,$$

the ratio of the surface areas, and we can define C_* as number of triangles.

Remark. C_{trav} is a constant. The base algorithm gives something like $C_{\text{triangle}}/C_{\text{trav}} \approx 8$, and an highly optimized algorithm gives something like 1.5.

[1] In the example of KD trees, this might be something like determining split points, choosing which direction, pushing to stacks, etc.

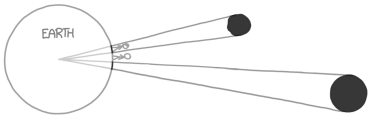
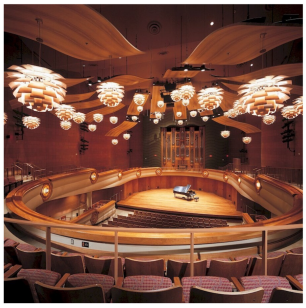


Figure 11.4. We get eclipses on Earth because the sun and moon have similar perceived size, as quantified by solid angle (xkcd.com/1276).



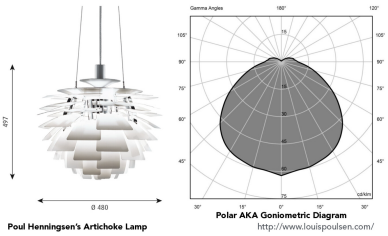


Figure 11.5. Artichoke lamp

[2] Candela is an SI unit!

Radiometry and Photometry

Now, we will move into the next part of the course on lighting and materials. Previously, we handled this in a rather handwavy way, but we'll do the real thing now. Physics time!

Physical processes covert energy into photos, which carry a small amount of energy. A light bulb, for example, consumes some energy which is released as heat, while a lot is released as photons. **Exposure** is the energy of photos hitting an object. In graphics, we will generally assume that light has "steady state" flow, i.e. it travels infinitely fast.

Definition 11.1. Radiant (luminous) energy Q is the energy of electromagnetic radiation in Joules.

Definition 11.2. Radiant (luminous) flux is the rate of energy over time, $\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t}$, as measured in watts (lumens).

Photometry accounts for the impact of the human visual system V (which we'll discuss more later on in the color lectures). In general, these quantities will vary over wavelength λ, and the flux we perceive is given by

$$\Phi_v = \int_0^\infty \Phi_e(\lambda) V(\lambda) \mathrm{d}\lambda.$$

Definition 11.3. A circle has angle $\theta = \ell / r$, where the entire circle consists of 2π radians. Likewise, a sphere has **solid angle** $\omega = A/r^2$, where the entire circle consists of 4π **steradians** (sr). See Figure 11.4.

Exercise 11.4. We can verify that $\Omega = \int_{S^2} \mathrm{d}\omega = 4\pi$.

Definition 11.5. The **radiant (luminous) intensity** is the power per unit solid angle emitted by a point light source:

$$I(\omega) = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega},$$

in units of W/sr (lm/sr = cd = candela).[2]

Example 11.6. Consider an **isotropic** point source (i.e. uniform intensity). Then $\Phi = 4\pi I$.

Remark. It's possible to design light sources to match certain light distributions. See Figure 11.5.

12 Photometry & Monte Carlo Integration (02/22)

Radiometry and Photometry

The lecture slides are here.

Definition 12.1. Irradiance (illuminance) is the power per unit area incident on a surface point, $E(x) = \frac{\mathrm{d}\Phi(x)}{\mathrm{d}A}$. The units are W/m^2 ($\mathrm{lm}/\mathrm{m}^2 =$ lux).

One approach of computing this is **Lambert's cosine law** (see Section 7), where in general $E = \frac{\Phi}{A}\cos\theta$, where $\cos\theta = l \cdot n$.

Remark. The Earth's axis of rotation is around 23.5° off the vertical axis, the cosine is lower in the winter than in summer — hence why we have seasons.

As we discussed previously in the Blinn-Phong model (Section 7), light emitting flux Φ in a uniform angular distribution will have $E = \Phi/(4\pi)$.

Definition 12.2. The **radiance (luminance)** is the power emitted, reflected, transmitted, or received by a surface, per unnit solid angle, per unit projected area. In math,

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

The units are $\mathrm{W}/(\mathrm{sr}\cdot\mathrm{m}^2)$ ($\mathrm{cd}/\mathrm{m}^2 =$ nit).

In general, the incident and exitant radiance functions on a point on a surface are not necessarily equal, so we distinguish them as L_i and L_o.

Consider a surface S. Then the flux per unit area on the surface can be modeled as due to the incoming light from all directions:

$$E(p) = \int_S L_i(p,\omega)\cos\theta\mathrm{d}\omega,$$

where the $\cos\theta$ is from Lambert's law.

Example 12.3. We can compute the irradiance from a uniform hemispherical light:

$$\begin{aligned} E(p) &= \int_{H^2} L\cos\theta\mathrm{d}\omega \\ &= L\int_0^{2\pi}\int_0^{\pi/2}\cos\theta\sin\theta\mathrm{d}\theta\mathrm{d}\varphi \\ &= L\pi. \end{aligned}$$

Remark. One way of measuring radiance is with a pinhole camera: measure the radiance for rays passing through the point at each point on the back of a box.

If you do this with a sphere and collect the light field at every point, you can compute views of an object at the center by tracing the rays back to the corresponding points on the sphere.

Monte Carlo integration

Typically, we approximate integrals via sampling. We did this for antialiasing, for example, where we selected some points and took averages.

The motivation for using this approach instead of a numerical approximation (i.e. Riemann sums) is that the numerical error is relatively lower as dimensionality increases. That is, getting a complete set of samples for $\{1, \dots, n\}^d$ requires $N = n^d$ samples. Numerical integration scales with $\frac{1}{n} = N^{-1/d}$, while random sampling scales with $N^{-1/2}$. Furthermore, it works well with any general function, but it produces noise.

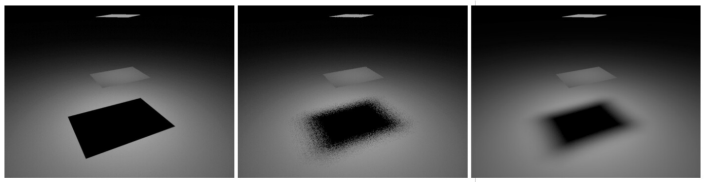


Figure 12.1. Shadows with Monte Carlo integration. Sampling the light source is much better than sampling one point, which yields sharp shadows.

Suppose we have some true data distribution $p(X)$, and we sample $X_i \overset{\text{iid}}{\sim} p$, and would like to approximate $\int_X f(x)\mathrm{d}x$. The estimator is given by

$$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}.$$

Example 12.4. If $p(x) = U(a, b)$, then $F_N = \frac{b-a}{N}\sum_{i=1}^{N} f(X_i)$. (This looks familiar!)

The following claim provides that MC sampling is actually useful:

Claim 12.5. F_N is unbiased.

Proof.

$$\mathbb{E}_{X_i\sim p}[F_N] = \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}_{X_i\sim p}\left[\frac{f(X_i)}{p(X_i)}\right] = \frac{1}{N}\sum_{i=1}^{N}\int\frac{f(x_i)}{p(x_i)}p(x_i)\mathrm{d}x_i = \int f(x_i).$$

■

We know that the variance will decrease as N^{-1}, so as we collect more samples, our estimate gets better in probability.

13 Monte Carlo integration & Global Illumination (02/27)

Monte Carlo integration

The lecture slides are here.

Last time, we talked about MC methods for approximating general integrals. This time, we'll discuss it in the context of graphics.

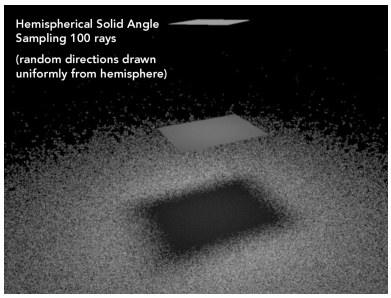


Figure 13.1. Noisy ray tracing

Example 13.1. Suppose we have a unit hemisphere and want to estimate the irradiance. Recall that

$$E(\mathbf{p}) = \int L(\mathbf{p}, \omega) \cos\theta \mathrm{d}\omega,$$

where θ is the angle between ω and the surface normal. We will estimate $X_i \sim p(\omega)$, where $p(\omega) = 1/(2\pi)$ uniformly, as

$$F_N = \frac{2\pi}{N} \sum_{i=1}^{N} L(\mathbf{p}, \omega_i) \cos\theta_i.$$

So for each point $\mathbf{p}$, we will sample N directions and evaluate the incoming radiance $L(\mathbf{p}, \omega_i)$ from those directions.

The problem with this approach is that sampling ω_i uniformly will produce high-variance outputs; see Figure 13.1. The problem is that the incoming radiance is zero for most directions in the scene. One idea is to change the sampling distribution p (domain-specific), rather than using a uniform prior, to produce better estimates for our integrals.

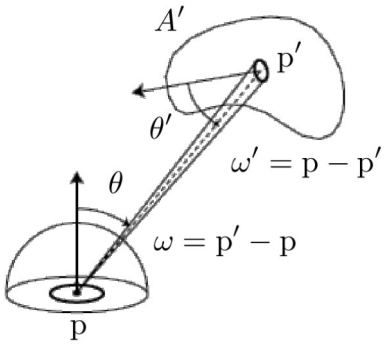


Figure 13.2. Sampling from a light source

Example 13.2. One example of a prior might be only integrating over the directions of light sources (see Figure 13.2). To implement this, one would approximate

$$E(\mathbf{p}) = \int_{A'} L_o(\mathbf{p}', \omega') V(\mathbf{p}, \mathbf{p}') \frac{\cos\theta \cos\theta'}{\|\mathbf{p} - \mathbf{p}'\|^2},$$

where $L_o(\mathbf{p}', \omega')$ denotes the outgoing radiance from $\mathbf{p}'$ at solid angle ω' (the direction towards $\mathbf{p}$), and $V(\mathbf{p}, \mathbf{p}')$ is an indicator for whether $\mathbf{p}$ is visible from $\mathbf{p}'$.

We can approximate this integral by MC with a uniform prior over A.

Remark. To draw samples X from a probability distribution with cdf G, recall the technique discussed in CS 70. It is equivalent to sample $U \sim U(0,1)$ and take $X = G^{-1}(U)$.

For a discrete variable, we can evaluate this using binary search.

Global illumination and path tracing

The lecture slides are here.

There's lots of components of Figure 13.3 we'd like to implement, and we already have all of the required tools. These include:

- Soft shadows (from the balls),
- Caustics (light passing through the glass ball and reflecting off various surfaces),
- Inter-reflections (reflections from each surface to every other surface).

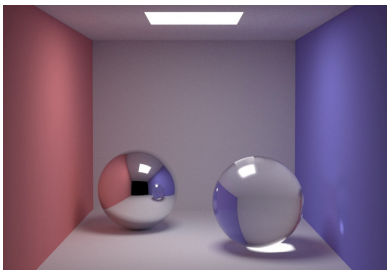


Figure 13.3. Global illumination

We will concern ourselves with **reflections**, which are the processes by which light incident on a surface leaves the same side without changing in frequency. We will represent this using a **bidirectional reflectance distribution function (BRDF)**, representing how much light is reflected into each outgoing direction from each incoming direction. See Figure 13.4.

The idea is that we can compute

$$f_r(\omega_i \to \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)\cos\theta_i \mathrm{d}\omega_i},$$

which has units $1/\text{sr}$. Then we can evaluate the incoming irradiance as

$$L_r(\mathbf{p}, \omega_r) = \int_{H^2} f_r(\mathbf{p}, \omega_i \to \omega_r) L_i(\mathbf{p}, \omega_i) \cos\theta_i \mathrm{d}\omega_i, \tag{1}$$

which we approximate with MC sampling.

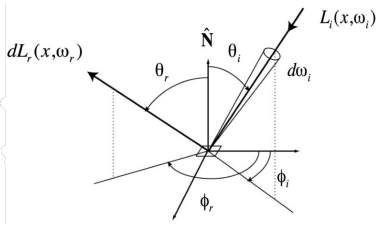


Figure 13.4. BRDF

Looking back at our earlier stuff, we want to use importance sampling here, too. Naively, we can do this over all light sources. But observe that in (1), our setup is naturally recursive. This makes sense, since we want light to reflect off lots of things. The following discussion is from Kajiya [1986].

Let $tr(p, \omega)$ be a **transport function**, returning the first surface intersection function in the scene along ray (p, ω). Observe invariance along a ray: $L_o((\mathbf{p}', \omega), -\omega) = L_i(\mathbf{p}, \omega)$, where $\mathbf{p}' = tr(p, \omega_i)$. (Intuitively, the outgoing radiance from a ray is the incoming radiance at its destination.)

With this in mind, we will rewrite (1) in terms of the outgoing radiance from every ray $(\mathbf{p}, \omega)$, and derive a recursive formulation:

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{H^2} f_r(\mathbf{p}, \omega_i \to \omega_o) L_i(\mathbf{p}, \omega_i) \cos\theta_i \mathrm{d}\omega_i. \tag{2}$$

To solve this, we will proceed via linear **operators** (recall Math 110!). We will define the **reflecion operator** as

$$R(g)(\mathbf{p},\omega) = \int_{H^2} f_r(\mathbf{p},\omega_i \to \omega_o) g(\mathbf{p},\omega_i) \cos\theta_i \mathrm{d}\omega_i,$$

so for example $R(L_i) = L_o$. We will also let the **transport operator** be

$$T(f)(\mathbf{p},\omega) = f(tr(\mathbf{p},\omega), -\omega),$$

so for example $T(L_o) = L_i$.

Substituting into (2), we find that

$$L_o = L_e + (R \circ T)(L_o).$$

14 Light Transport & Materials (02/29)

Light transport

The lecture slides are here.

Last time, we found that

$$L_o = L_e + (R \circ T)(L_o) =: L_e + K(L_o),$$

where R is the reflection operator, T is the transport operator, and K is the **full one-bounce light transport operator**.

Thus, we can compute

$$L_o = (I - K)^{-1}(L_e) = L_e + K(L_e) + K^2(L_e) + \cdots,$$

where we enforce $\|K\| \leq 1$ (a function norm; this has shown to be the case in a variety of physical systems due to energy dissipation). Indeed, this follows intuitively because L_e is the originally emitted image, K is the result of one bounce, K^2 of two bounces, and so on.

Remark. The Blinn–Phong shading model is based on only one bounce.

Remark. Ambient light is highly dependent on the geometry of the scene. It often needs a few bounces to work on diffuse surfaces, and around 8–16 bounces for transparent or reflective ones.

So we have a way of representing this process algebraically, and it remains to actually compute this efficiently. (Doing this function composition directly is expensive.) We can regard this bounce structure as the integral over all ray paths, and our task is to find an unbiased estimator.

Our first idea is to use Figure 14.1, where we just evaluate this infinite sum directly via sampling. The problem, of course, is that there is no base case in the recursion.

Figure 14.1. Idea 1: Naive MC path sampling

```
EstRadianceIn(x, ω)
   p = intersectScene(x, ω);
   L = p.emittedLight(-ω);
   ωi, pdf = p.brdf.sampleDirection(-ω);
   L += EstRadianceIn(p, ωi) * p.brdf(ωi, -ω) * costheta / pdf;
   return L;
```

So we will use **Russian roulette sampling**, where we will continue a path with probability p_{rr} at each step. To make our estimate unbiased, we let

$$X_{\text{rr}} = \begin{cases} X/p_{\text{rr}} & \text{w.p. } p_{\text{rr}} \\ 0 & \text{else.} \end{cases}$$

(It is easy to verify that the expectations are the same.[3])

[3] This looks like dropout!

The problem with this approach alone is that the distributions only converge in expectation by LLN, and as a result there is high variance.

To fix this, we'll partition this recursive radiance evaluation into two steps:

- Direct lighting: importance sampling lights
- Indirect lighting: recursive with BRDF importance sampling.

Material modeling

The lecture slides are here.

The material properties of an object are given by **microfacets**. In general, concentrated microfacets will give glossy surfaces (where we regard glossy as materials that reflect high concentrations of specular light), and spread-out microfacets will give diffuse surfaces. The idea is to treat microfacets as mirrors, and model them accordingly.

```
EstRadianceIn(x, ω)                // incoming at x from dir ω
   p = intersectScene(x, ω);
   return ZeroBounceRadiance(p, -ω)
          + AtLeastOneBounceRadiance(p, -ω);

ZeroBounceRadiance(p, ωo)          // outgoing at p in dir ωo
   return p.emittedLight(ωo);
```

```
AtLeastOneBounceRadiance(p, ωo)              // out at p, dir ωo
   L = OneBounceRadiance(p, ωo);             // direct illum

   ωi, pdf = p.brdf.sampleDirection(ωo);       // Imp. sampling
   p' =  intersectScene(p, ωi);
   cpdf = continuationProbability(p.brdf, ωi, ωo);
   if (random01() < cpdf)                    // Russ. Roulette
      L += AtLeastOneBounceRadiance(p', -ωi)  // Recursive est. of
      * p.brdf(ωi, ωo) * costheta / pdf / cpdf;// indirect illum
   return L;

OneBounceRadiance(p, ωo)                     // out at p, dir ωo
    return DirectLightingSampleLights(p, ωo); // direct illum
```

```
DirectLightingSamplingLights(p, ωo)
   L, ωi, pdf = lights.sampleDirection(p);   // Imp. sampling

   if (scene.shadowIntersection(p, ωi))      // Shadow ray
      return 0;
   else
      return L * p.brdf(ωi, ωo) * costheta / pdf;
```

Figure 14.2. Idea 2.1: Path tracing with Russian roulette

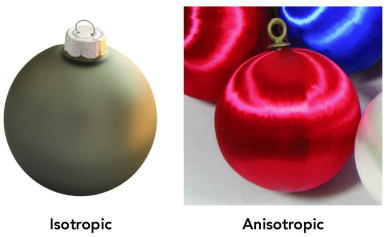


Figure 14.3. Isotropic and anisotropic materials

Figure 14.4. Rendering highly specular materials

In general, we will model this using BRDFs. The formula is given by

$$f(\mathbf{i},\mathbf{o}) = \frac{F(\mathbf{i},\mathbf{h})G(\mathbf{i},\mathbf{o},\mathbf{h})D(\mathbf{h})}{4(\mathbf{n}\cdot\mathbf{i})(\mathbf{n}\cdot\mathbf{o})},$$

where:

- we are considering microfacets that reflect ω_i to ω_o, and let $\mathbf{h} = \frac{\mathbf{i}+\mathbf{o}}{\|\mathbf{i}+\mathbf{o}\|}$ be the half vector,
- the **Fresnel reflection** term F indicates that reflectance depends on the incident angle and polarization of light,
- the shadow-masking term G is a binary indicator,
- and the distribution of normals D is a density function over vectors.

In practice, there are lots of complexities with modeling D. A naive choice would be to take it as a (multivariate) Gaussian, but in many cases this is not feasible:

- **Anisotropic** materials have directional surface normals, whereas **isotropic** materials do not. See Figure 14.3.
- To provide a realistic model for surfaces, we also need to provide a systematic method for "noising" our distributions (Gaussians are overly smooth). See Figure 14.4.[4,5] Indeed, we can follow this same approach in a variety of settings: a more concentrated distribution can be useful for metallic flakes/sparkles, for example, whereas modeling ocean waves needs a more continuous representation.

[4] Ling-Qi Yan, Miloš Hašan, Wenzel Jakob, Jason Lawrence, Steve Marschner, and Ravi Ramamoorthi. Rendering glints on high-resolution normal-mapped specular surfaces. *ACM Transactions on Graphics (TOG)*, 33(4):1–9, 2014

[5] Ling-Qi Yan, Miloš Hašan, Steve Marschner, and Ravi Ramamoorthi. Position-normal distributions for efficient rendering of specular microstructure. *ACM Transactions on Graphics (TOG)*, 35(4):1–9, 2016

15 Material modeling & Cameras and Lenses (03/05)

Material modeling

The lecture slides are here.

A common task is modeling hair and fur. The original approach was the Kajiya-Kay model[6], which proposed modeling hair as cylinders and coloring them according to specular highlights. The Marschner model[7] corrects these by considering the transmission into multiple cones, which accounts for direct reflection, transmission through the fiber, and scattering within the fiber. See Figure 15.1.

Indeed, there are lots of ways in which light can interact with various materials, e.g. smoke, fog, or water, as in Figure 15.2. This violates the

[6] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23 (3):271–280, jul 1989. ISSN 0097-8930. DOI: 10.1145/74334.74361. URL https://doi.org/10.1145/74334.74361

[7] Stephen R Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. Light scattering from human hair fibers. *ACM Transactions on Graphics (TOG)*, 22(3):780–791, 2003

assumption of BRDF, where a ray may exit an object at a different point than it enters. To account for this, we use a BSSRDF as a function of four parameters $(x_i, \omega_i, x_o, \omega_o)$. Of course, running path tracing through every material is computationally expensive, so we'll want some way to do this efficiently. (Out of scope for this class.)

Remark. Most things are actually rendered translucently. For example, skin and food have very rough textures, and modeling them translucently enables a much smoother visualization.

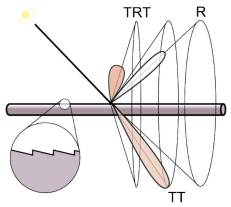


Figure 15.1. Marschner model for hair/fur interaction (R is reflection, T is transmission)

Cameras and lenses

The lecture slides are here.

How are images actually captured from the world? Figure 15.3 shows an example of an old-style camera. There's lots of things going on, but the most important component is that there are many (on the order of 10–15) lenses that collectively bring an image into focus on a plane. The sensor accumulates irradiance during exposure. Pixels provide a visual representation for this. We'll look at image processing later; for now, we'll look at the optics.

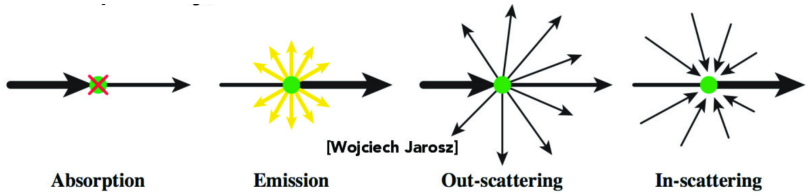


Figure 15.2. Participating medium

In a camera, we have a fixed sensor size h and hope to be able to vary the focal length f. The relationship between these variables is shown in Figure 15.4, where we can write

$$\text{FOV} = 2 \arctan\left(\frac{h}{2f}\right).$$

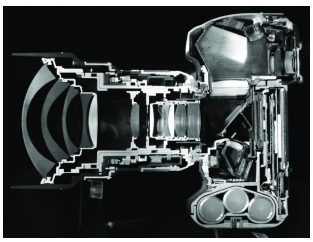


Figure 15.3. Cross-section of a camera

Example 15.1. There are lots of sensor sizes, depending on your camera. A "medium format" corresponds to a sensor area of 22cm^2, whereas a smartphone will have a sensor area of around 0.15cm^2.

Example 15.2. A common sensor size might be 36×24mm, which is commonly referred to as the 35mm-format film. There are many common focal lengths: 17mm is a wide angle at 104°, 50mm is a "normal" lens at 47°, and 200mm is a telephoto lens at 12° (you'll need this for good pictures at a football game, for example).

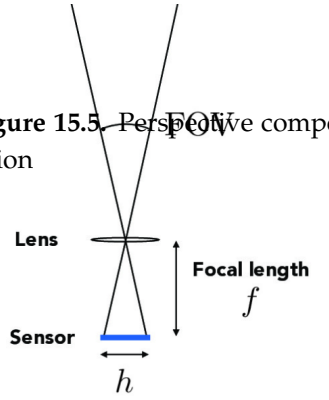


Figure 15.4. Focal length vs. FOV

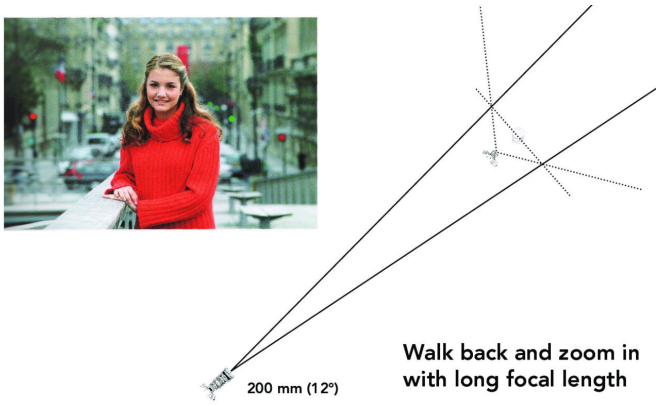


Figure 15.5. Perspective composition

From a photo, you can gauge the field of view. In general, if objects in the back look small (e.g. fisheye), the FOV is large, while if they are large, the FOV is small. This is just what you get with similar triangles. This is useful for photo composition; recall Figure 5.3, for example. From a photographer's perspective, maintaining the same image size means that the subject should occupy the same proportion of the width of the image. See Figure 15.5 for example: using a $\tan x \approx x$ approximation for small x, we have to move a distance roughly proportional to the focal length and zoom accordingly.

Remark. We can use this as tips for our own photography!

- The subject should be prominent, occupying around 1/3 of the image.
- There should be a good perspective relationship (relative size) between the subject, foreground, and background. This requires moving physically or zooming in.

In general, perspective matters more than pixels! This works on a smartphone too.

Concept. Exposure = irradiance × time × gain, where:

- Irradiance is the power of light falling on image sensor pixel, and is affected by factors like scene brightness or lens aperture (as given by F-stops or lens iris control)
- Time is determined by the shutter opening and closing (as given by the inverse of shutter speed)
- Gain is the amplification of sensor pixel values (ISO gain).

The key is that since these factors are multiplicative, changing the exposure corresponds to a logarithmic number of "stops" on the camera.

Note. ISO gain trades off sensitivity for noise; in general, we set the gain to the lowest value that works to minimize the amount of noise.

Definition 15.3. The **F-number** of a lens is the quotient

$$\text{focal length}/\text{diameter of aperture}.$$

Some common F-numbers are shown in Figure 15.6, and incrementing by 1 stop doubles the exposure.

The F-number is the maximum for that lens; you can stop down a lens to a smaller aperture (larger F-number).

F-Stop	1.4	2.0	2.8	4.0	5.6	8.0	11.0	16.0	22.0	32.0
Shutter	1/500	1/250	1/125	1/60	1/30	1/15	1/8	1/4	1/2	1

Figure 15.6. There are multiple combinations of apertures and shutter speeds that yield the same exposure (but different images)

Remark. Typically, cameras follow this procedure:

- Reset pixels to start exposure
- Reading out a pixel electronically ends the exposure
- Sensors are read out sequentially.

So it takes some time (up to 1/30 sec) to read out the entire sensor. If we reset all pixels at the same time, then the last pixel will correspond to a longer exposure, so in practice they are reset on a schedule.

This results in the rolling shutter artifact! Different parts of the image correspond to different points in time. See Figure 15.7. The only "good" fix for this is increasing the shutter speed, or using a global shutter sensor.

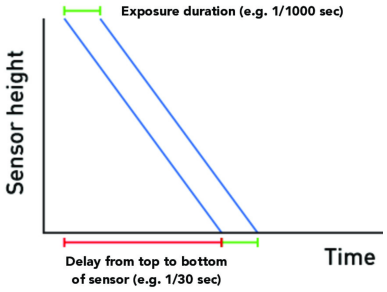


Figure 15.7. Rolling shutter effect

16 Optics (03/07)

The lecture slides are here.

The key benefit of a lens is that it is able to focus objects well. First, we will discuss an approximation to a thin lens in an ideal setting, where all rays through an object will get focused on the same point on a plane. As you ray trace different points on an object, you produce an image. See Figure 16.1.

Remark. In practice, our lenses are not ideal. In fact, Maxwell proved that it is physically impossible. That is, for a rotationally symmetric optical system, it is impossible to eliminate all primary aberrations for all angles of incidence and wavelengths, since eliminating each type of aberration results in conflicts. We can get around this problem by having multiple lenses, which collectively approximate an ideal lens.

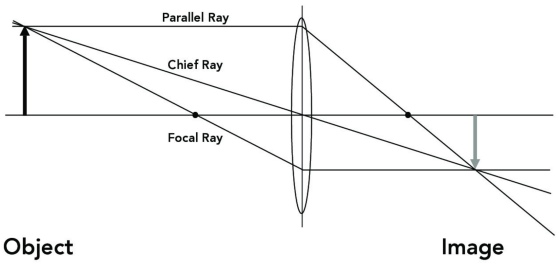


Figure 16.1. Under an approximation, an object is focused exactly onto an image plane

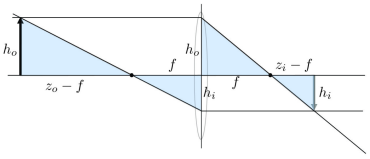


Figure 16.2. Deriving the thin lens equation

In this setting, let f denote the focal length of the lens. We will determine the relationship between the depths of the object and image, which we denote z_o and z_i, respectively. Let h_o and h_i denote the heights of the object and image, respectively. See Figure 16.2.

By similar triangles, we have

$$\frac{h_o}{z_o - f} = \frac{h_i}{f} \implies \frac{h_o}{h_i} = \frac{z_o - f}{f}$$
$$\frac{h_o}{f} = \frac{h_i}{z_i - f} \implies \frac{h_o}{h_i} = \frac{f}{z_i - f}.$$

Thus, $(z_o - f)(z_i - f) = f^2$, or

$$\frac{1}{f} = \frac{1}{z_i} + \frac{1}{z_o}.$$

The **magnification** is given by $m = h_i/h_o = z_i/z_o$ by similar triangles.

Example 16.1. If you have a faraway object (e.g. a mountain a few miles away), you can approximate $z_o \approx \infty$, and accordingly $m \approx 0$. (As expected! The mountain is much bigger than the image.)

Example 16.2. Suppose you want $z_i = z_o$. Then $z_i = 2f$ by the thin lens equation.

Remark. To increment focus linearly while maintaining the same-size object in a photo, z should scale as n^{-1}. (So for faraway objects, the required change is tiny!)

Example 16.3. When you take an image containing a point light, it gets blurred as a disc. We can compute the size of this disc: see Figure 16.3.

It follows from similar triangles that

$$\frac{C}{A} = \frac{d'}{z_i} = \frac{|z_s - z_i|}{z_i}.$$

This **blurring kernel** applies to all points with the same depth, but point light sources are particularly obvious because a single point gets mapped to a disc, whereas everything else gets interpolated and gets smoothed out.

Example 16.4. Suppose you have a 50mm f/2 lens, which focuses at 1m. The background at 10m and foreground at 0.3m will naturally be blurred, and we'd like to compute their respective circles of confusion.

We are given that $f = 50\text{mm}$ and the aperture $A = f/2 = 25\text{mm}$. Since the lens focuses at 1m, the sensor plane z_s satisfies $\frac{1}{50\text{mm}} = \frac{1}{1000\text{mm}} + \frac{1}{z_s}$, so $z_s \approx 52.6\text{mm}$. Then the background at 10m is mapped to a circle of confusion with diameter $C = A|z_s - z_i|/z_i \approx 1.18\text{mm}$, and the foreground at 0.3m has $C \approx 3.1\text{mm}$.

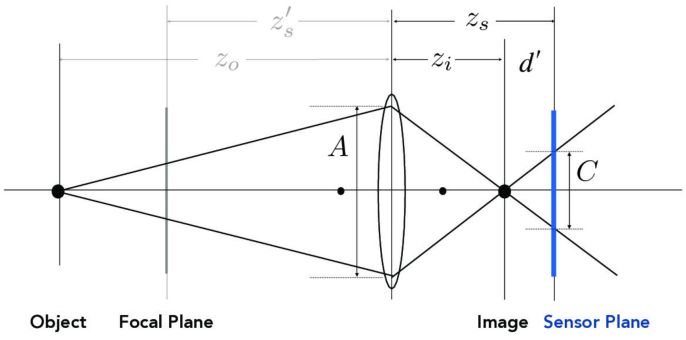


Figure 16.3. An object gets mapped to an image, and points on the focal plane get mapped to the sensor plane

Figure 16.4. As in Example 16.4, there are circles of confusion in the foreground and background

Definition 16.5. The **depth of field** for a lens is the range of depths for which objects get mapped to a sharp image, as measured by the size of the circle of confusion.

The threshold for this size is typically 1 viewing pixel for an image on the web. See Figure 16.5. The ideas are quite simple but the math is messy.

Remark. If for some reason you have a view camera (i.e. your lens and sensor planes are not parallel), your camera will need to satisfy the **Scheimpflug rule** to keep your image sharp: your lens plane, image plane, and subject plane must intersect along a common line.

Concept. Suppose we have a plane at depth z_i, and want to evaluate the pixel value at some point x'. To do this, integrate over points x''

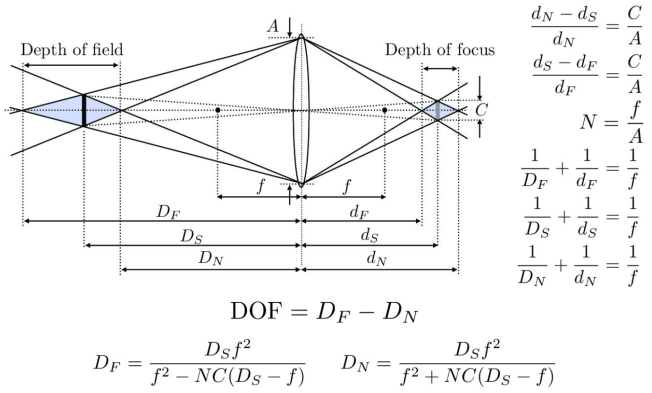


Figure 16.5. Depth of field and depth of focus

on the lens, and trace the rays to intersect at a point x''' with depth z_o as given by the thin lens equation. In practice, we approximate this integral with Monte Carlo sampling, estimating the radiance on these rays via path tracing.

Remark. **Bokeh** is the shape and quality of out-of-focus blur. For small, out-of-focus lights, bokeh takes on the shape of the lens aperture.

17 Cameras and Lenses & Physical Simulation (03/12)

Cameras and Lenses

The lecture slides are here.

An algorithm for getting a pixel value is as follows:

- Sample N random positions in the pixel $\{x'\}$.
- For each position x', choose a random position x'' on the back of the lens.
- Trace a ray from x' to x'' until it either misses an element or enters the scene (then path trace through the scene).
- Weight each ray as

$$L(x'' \to x') \frac{\cos\theta' \cos\theta''}{\|x'' - x'\|^2}.$$

Prof. Ng did a startup called Lytro, which captured 4D light fields (i.e. the radiance flowing on every ray) rather than your typical 2D photographs (irradiance at every pixel on a plane).

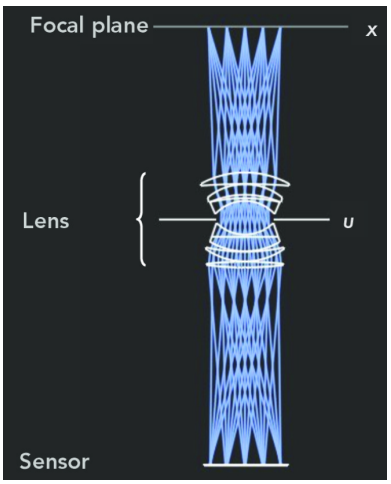


Figure 17.1. 4D light field

The idea is that you can consider points (x, y) on a focal plane and consider their mapping (u, v) on the lens. A plenoptic camera just samples the light field, i.e. for each (x, y) they MC sample over all (u, v). To get a pixel value, they equivalently sample over (x, y), so the portion of the light field that is collected for a pixel is a grid cell in (x, y, u, v) space. Equivalently, a plenoptic camera estimates the irradiance at each point on the sensor, while a light field camera estimates the radiance along all incoming rays. See Figure 17.1.

The idea of a light field camera is to capture this directly. At a sensor level, they use microlenses, which appears directly on top of the sensor in the camera. The benefit of this approach is that you can computationally refocus, i.e. you can take one image and focus on lots of different things. To do so, just move the sensor plane computationally and ray-trace to the new plane. See Figure 17.2.

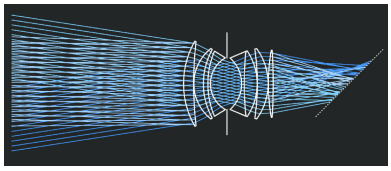


Figure 17.2. Tilted focal plane

Cameras have a **microlens array** between the color glass and CMOS pixels. Light field cameras have different microlens and pixel arrangements, plus some digital changes.

Physical Simulation

To generate the motion of objects in numerical simulation, we can use kinematics. Considering the interactions of a large number of particles or objects is more challenging, since there are lots of interesting interactions between them.

We know from mechanics that

$$F = k_s(\mathbf{b} - \mathbf{a}).$$

In real life, this would make for a spring that oscillates infinitely. To account for damping, we can use

$$f_a = -k_d \frac{\mathbf{b} - \mathbf{a}}{\|\mathbf{b} - \mathbf{a}\|}(\dot{\mathbf{b}} - \dot{\mathbf{a}}) \cdot \frac{\mathbf{b} - \mathbf{a}}{\|\mathbf{b} - \mathbf{a}\|}.$$

Suppose we are simulating cloths, which could be either 10×10 or 20×20. Using the same spring constant to model them will make the 20×20 cloth harder to stretch proportionally, since more force is required to stretch it double the distance. To have the cloths satisfy the same physical properties, we instead use a constant strain $\varepsilon = \Delta\ell / \ell_0$.

In practice, modeling with a straightforward grid pattern will have problems with shearing. A fix is to follow the design in Figure 17.3, where the red springs are much weaker than the blue ones, to avoid directional bias.

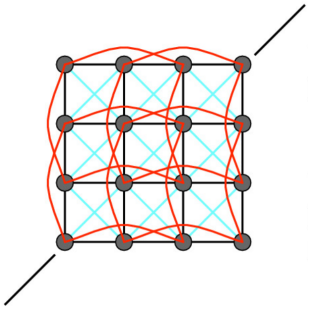


Figure 17.3. Spring structure resistant to shearing

To update positions with kinematics, typically we use an Euler approximation, i.e. $\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\dot{\mathbf{x}}(t)$ and $\dot{\mathbf{x}}(t + \Delta t) = \dot{\mathbf{x}}(t) + \Delta t\ddot{\mathbf{x}}(t)$. This method often works poorly in practice due to numerical instability, so some hacks to get it to converge better are:

- **Adaptive step size**: use some hybrid of Δt and two $\Delta t/2$'s.
- **Momentum (implicit Euler)**, i.e. use the velocity and acceleration from time $t + \Delta t$.
- **Verlet integration**, i.e. use momentum and constrain the positions of the particles to prevent instability. This results in some energy loss, but can work quite well in practice.

18 Physical Simulation & Animation (03/14)

Physical Simulation

The lecture slides are here.

In general, you can model many dynamical systems as a collection of particles that satisfy some properties. For example, you can model a flock of birds as an ODE, where each bird is subject to attraction to the center of its neighbors, repulsion from individual neighbors, and alignment toward the average trajectories of its neighbors. The same sort of thing works for crowds, water, and other large systems.

Another idea for cloths is to use finite element method.[8] The idea is to solve PDEs by subdividing a system into smaller parts called finite elements via constructing a mesh, then to solve this discretized system with algebra. (In practice, one problem is that lots of triangles are required, and they may be nonuniform.)

[8] Here's a Wikipedia page on it; looks pretty cool!

Animation

There are lots of popular techniques for making animations visually appealing and compelling.

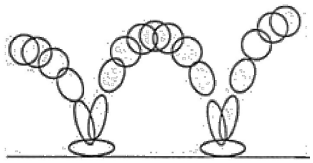


Figure 18.1. Squash and stretch

- **Squash and stretch** refer to the rigidity and mass of an object by distorting its shape during a motion. The shape may change, but not the volume. See Figure 18.1.
- **Anticipation** means preparation for a movement. For instance, when throwing a punch, a person must first wind up.

- **Staging** should ensure that the audience is looking at the right place and makes for a clear situation. Parts are often exaggerated.
- **Follow through** is motion blur, where often different components move at different rates.
- **Ease-in** and **ease-out** means that motion shouldn't start and stop abruptly.
- Characters move in arcs, rather than straight lines.
- There are **secondary actions**, in which motion results from or complements a primary action.

Typically, animation development follows a hierarchical structure, where animators develop **keyframes** before filling in be**tween** them. Ideally we'd like to automate the latter process, where we parameterize the scene and interpolate between them, but in practice the keyframes are too far apart to allow complex scenes to naively interpolate.

In general, this problem is pretty hard. Let's consider the following case: you have an arm consisting of N segments, which has some joint angles. Based on those joint angles, you can determine the end effector's position (**forward kinematics**). The problem at hand is to do the opposite (**inverse kinematics**): how to recover the joint angles based on end effector positions.

Example 18.1. Consider the $N = 2$ case. If your two links have lengths ℓ_1 and ℓ_2 and relative angles θ_1 and θ_2, then you can parameterize the location of the end effector as

$$(p_z, p_x) = (\ell_1 \cos\theta_1 + \ell_2 \cos(\theta_1 + \theta_2)).$$

To make this useful, we will specify the location of the end effector and determine the required joint angles:

$$\theta_2 = \cos^{-1}\left(\frac{p_z^2 + p_x^2 - \ell_1^2 - \ell_2^2}{2\ell_1\ell_2}\right)$$
$$\theta_1 = \frac{-p_z\ell_2 \sin\theta_2 + p_x(\ell_1 + \ell_2\cos\theta_2)}{p_x\ell_2 \sin\theta_2 + p_z(\ell_1 + \ell_2\cos\theta_2)}$$

This is already pretty complicated!

Note. In general, solving this system is pretty hard, and it will almost always be underconstrained. That is, it's possible for there to be multiple continuous solutions, multiple discontinuous solutions, or no solutions at all. People have worked on using learned models for inverse kinematics to make motion appear natural.

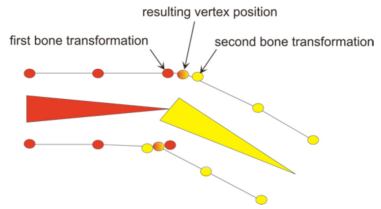


Figure 18.2. Move surface along with assigned handles

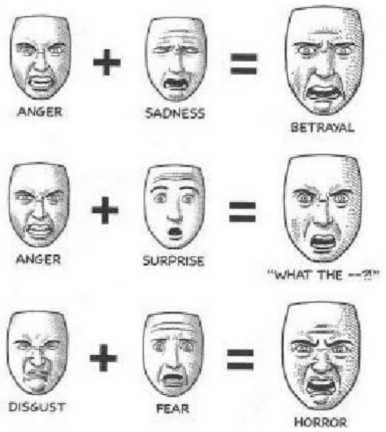


Figure 18.3. Blending facial expressions

Figure 18.4. Rigging

Figure 18.5. Motion capture

Now, suppose we've solved this problem, and we'd like to move on to coming up with a parameterization that is well-suited for interpolation, then actually creating a reasonable animation based on the joint angles. (For example, we have lots of muscles on our bodies, and specifying the amount of tension exerted by each one to make a single frame is a lot of work.) Below are some common techniques:

- Suppose we have a model for an object's (e.g. a humanoid) motion, and its joints are moving. Skinning this model will be pretty hard: you can imagine that if it bends its arm, the inside of its elbow will self-intersect, and the outside will have holes. The fix is to use blending, where you interpolate between the corresponding points on the transformed links. A simple approach is to use **linear blend skinning**, which is quite easy to implement.
- **Blending** is a common way of handling facial expressions. You only actually need to implement a few expressions, and everything else can be achieved with linear interpolation.
- **Rigging** is a common way to augment a character with controls to easily change its pose, create facial expressions, bulge muscles, etc. Typically, this is sophisticated enough to operate in a UI, where you can just drag dots and the overall mesh geometry will get updated.
- **Motion capture** is a data-driven approach to creating animation sequences. The idea is to produce an animation by extracting pose as a function of time from raw data, then mapping those poses onto a character.

  Common approaches are optical, magnetic, and mechanical. Among these, optical is the most widely used, where each marker is triangulated from multiple cameras. The main benefit is that motion is very natural, but it may need touchup and is costly to set up.

19 Color Science (03/19)

The lecture slides are here.

Remark. Up to half of women have four color receptors rather than three. (It's X-linked.)

Example 19.1. In very rare cases, there exist subjects who have one trichromatic eye and one deuteranopic eye. One might conduct a study where a divider is placed between their eyes, and they can match one rainbow to the other. In these cases, people have generally tuned toward yellows and blues.

Example 19.2. Humans have unique color processing systems. One example is shown in Figure 19.1, where the same blue filter is applied to a banana or an entire scene. In the case of just the banana, it looks very green or blue. In the case of the whole scene, it is very clearly yellow.

Figure 19.1. The banana is the same color in both photos

One approach is to apply automatic white balance, where you normalize each channel independently. That is, if you have some input (R', G', B') and a white object (R'_W, G'_W, B'_W), applying

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \operatorname{diag}\left(\frac{1}{R'_W}, \frac{1}{G'_W}, \frac{1}{B'_W}\right) \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

balances the color channels.

Example 19.3. But even this isn't sufficient. In Figure 19.2, for example, the squares labeled A and B have exactly the same RGB, even though in the context of a checkerboard they look very different.

Example 19.4. See the slides for **afterimages**. Stare at the black dot for some time, and without blinking, switch to the next slide. For a second or so, you'll see color mapped inversely to a grayscale image.

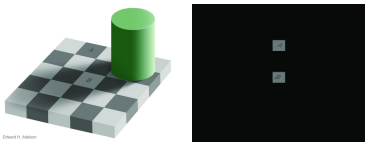


Figure 19.2. Adelson's checkerboard

Example 19.5. The **watercolor illusion** shows that human perception implements floodfill. In Figure 19.3, the only change between the left and right images is that the yellow border is applied. The fill looks cream-colored, but really it's white.

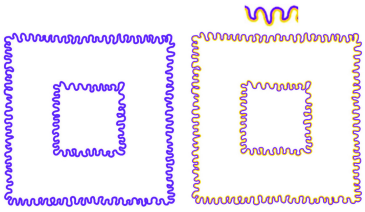


Figure 19.3. Watercolor illusion

For now, our goal will be to reproduce colors. That is, from an array of pixel values, how can we output those colors on an RGB display such that a human will perceive it the same way?

Concept. Color is a phenomenon of human perception, not a universal property of light.

Remark. From white light, you can produce a rainbow with a prism (Newton first demonstrated this). You can't further subdivide it.

Light has various wavelengths, with some subset on the **visible electromagnetic spectrum**, which ranges from red to blue (but not purple!).

20 Color Science (03/21)

The lecture slides are here.

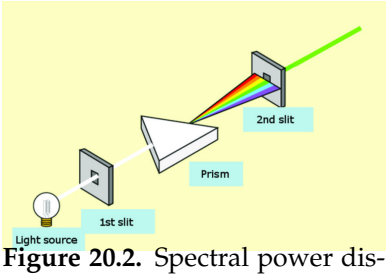


Figure 20.1. Monochromator

Physical basis of color

A monochromator delivers light of a single wavelength from a light source with broad spectrum. The idea is that a prism splits light into different frequencies, so rotating the prism and tracking its output through a slit will yield light of one frequency.

In general, we are interested in the spectral power distributions, which model the "amount" of light present at each wavelength. Formally, the units are W/nm; in practice, we tend to care about ratios, so often we drop the units. Some examples are in Figure 20.2; there are many kinds of "white" light, which possess different amounts of varying-frequency lights.

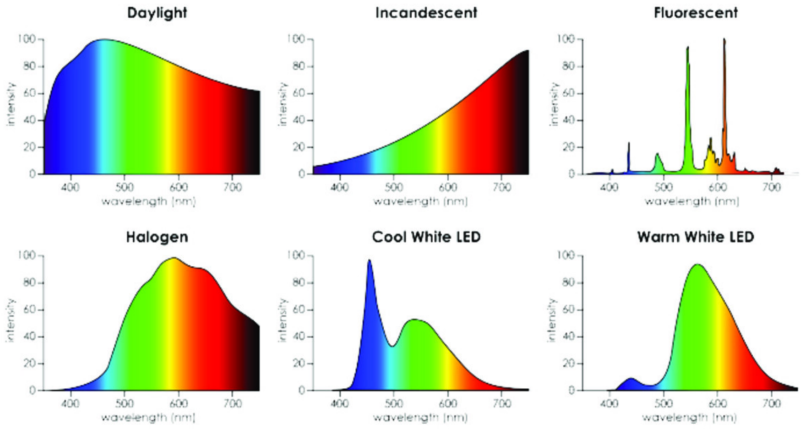


Figure 20.2. Spectral power distributions

Remark. Spectral power distributions are additive. That is, if you have two lights, the SPD of the lights collectively equals the sum of the SPDs of the lights.

A simple model for light detectors is to assign a scalar value proportional to the number of photons detected. This is quite lossy, since there is no way to distinguish between light of different frequencies. But in practice (e.g. in cameras or in human eyes), it is sufficient.

An easy way to do this is to let $n(\lambda)$ be the number of incident photons as a function of wavelength, and $p(\lambda)$ be the "detection efficiency." Then the total number of photos is

$$X = \int n(\lambda)p(\lambda)\mathrm{d}\lambda.$$

We'll do something similar for spectral power distributions:

$$X = \int s(\lambda)r(\lambda)\mathrm{d}\lambda,$$

where $s(\lambda)$ is the input spectrum and $r(\lambda)$ is the detector's sensitivity.

Computationally, we will estimate this integral by discretizing it, i.e. only take some finite set $\{\lambda\}$.

Example 20.1. Maxwell did a color-matching experiment. The idea is that you have two annuli, one with adjustable RGB colors (just pieces of paper) and black and white. The idea is that if you spin the handle super fast, the colors blend, and you get humans to evaluate whether the colors are the same.

The key discovery is that a very small number of primary lights is required to capture a large range of colors.

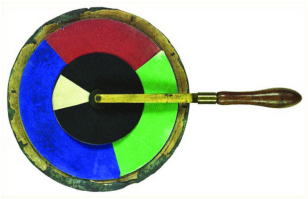


Figure 20.3. Maxwell's experiment

Remark. There was lots of work circulating around at the time that inspired Maxwell to choose red, green, and blue, e.g. the Young–Helmholtz theory (trichromatic color vision).

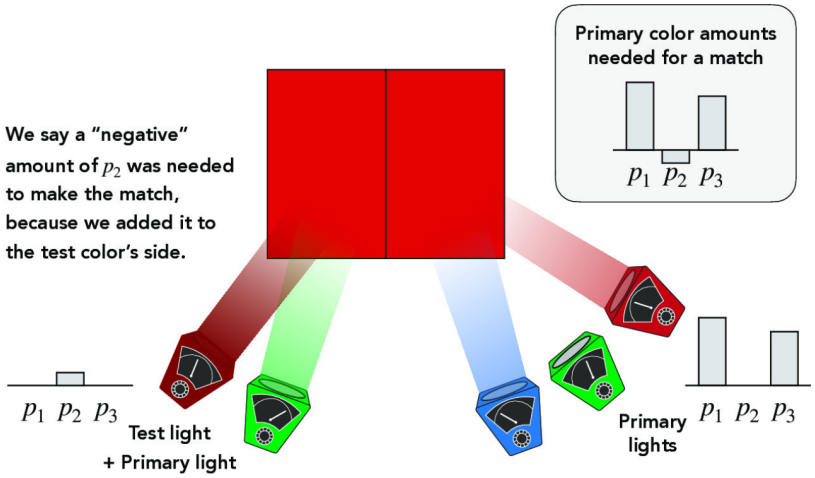


Figure 20.4. The dark red is out of gamut of the primary colors

In some cases, you will need "negative" amounts of color for matching, i.e. you need to add light of some frequency to the test side. See Figure 20.4. You can think of a **gamut** as a nonnegative linear combination.

Note. Three colors is sufficient to cover every color. Two colors is insufficient for normal vision. But for people with red-green color-blindness, two is enough.

Example 20.2. The CIE RGB color-matching experiment used specific wavelengths of red, green, and blue, and tested on a monochromatic light. We'll define the required "amounts" of each color corresponding to the wavelength λ as **color matching functions**, and compute the

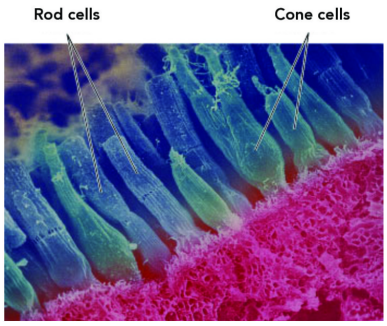


Figure 20.5. Retinal photoreceptor cells

required weights of $r(\lambda)$, $g(\lambda)$, and $b(\lambda)$ to match the desired distribution. (Thus, it's possible for r, g, b to be negative.)

Biological basis of color

The retina has **rods** and **cones**; see Figure 20.5. Rods are usually used in very low light, and perceive only shades of gray. There are around 120 million rods in each eye. Cones are the primary receptors in typical light levels, and there are around 6-7 million cones in the eye. There are three kinds of cones with different spectral sensitivities.

Remark. There are no rods in the fovea. When you're stargazing, you are primarily using your rods, though, which is why people recommend using an averted gaze.

The three types of cone cells are short, medium, and large wavelengths, with corresponding detector sensitivies r_S, r_M, and r_L. Again, we will discretize these, so that

$$\begin{pmatrix} S \\ M \\ L \end{pmatrix} = \begin{pmatrix} r_S^\top \\ r_M^\top \\ r_L^\top \end{pmatrix} s.$$

As before, this representation is still lossy, but it is (as far as we know) what happens in visual processing. Previously, mapping to one scalar gave an $\infty \to 1$ dimensionality reduction, and now it is $\infty \to 3$.

This loss means that there are **metamers**, i.e. different spectra that correspond to the same (S, M, L) response, or equivalently look the same to a human. This is why our screens feel real, even though they are just a layout of RGB pixels. [9]

In graphics, our goal is to choose RGB values for a display such that the output color

$$s_{\text{disp}}(\lambda) := Rs_R(\lambda) + Gs_G(\lambda) + Bs_B(\lambda) = \begin{pmatrix} s_R & s_G & s_B \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

matches the appearance of the target $s(\lambda)$ in the real world.

Thus,

$$\begin{pmatrix} S \\ M \\ L \end{pmatrix}_{\text{disp}} = \begin{pmatrix} S \\ M \\ L \end{pmatrix}_{\text{real}} \implies \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \left(\begin{pmatrix} r_s^\top \\ r_M^\top \\ r_L^\top \end{pmatrix} \begin{pmatrix} s_R & s_G & s_B \end{pmatrix} \right)^{-1} \begin{pmatrix} r_S^\top \\ r_M^\top \\ r_L^\top \end{pmatrix} s.$$

In some cases, we require negative colors, but this isn't physically

[9] There are a lot fewer S cone cells than M and L cells, which have very similar sensitivity functions. The true distributions are unknown, and they cannot be exactly computed. They are just vision scientists' best estimates.

possible (they are just outside of our gamut). Figure 20.6 depicts the **spectral locus** of human cone cells' (S, M, L) response to monochromatic light. The space of all possible responses must be a linear combination of points on this curve.

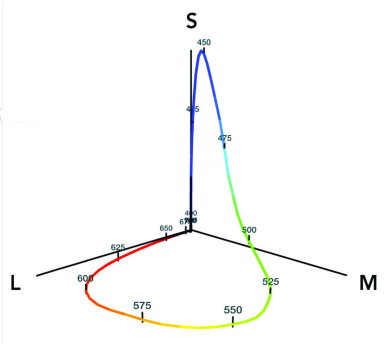


Figure 20.6. Spectral locus in LMS space

21 Color Science & Image Sensors (04/02)

Color Science

The lecture slides are here.

Generally, we will represent color spaces as a linear combination of three colors. There are many possible color spaces (e.g. RGB, HSV), and the same color will have different coordinates in each space.

The classical example is CIE XYZ, which provides luminance curves $X(\lambda)$, $Y(\lambda)$, and $Z(\lambda)$. Separating them just requires barycentric interpolation.

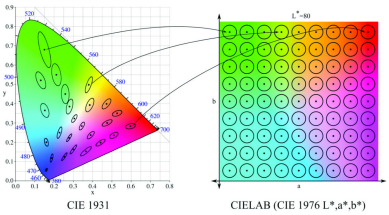


Figure 21.1. CIELAB is a perceptually-organized color space that aims for perceptual uniformity. In CIE, the indicated ellipses are perceived similarly by humans.

Image Sensors

The lecture slides are here.

There were several key discoveries in producing sensors:

- Photoelectric effect: electrons are emitted due to incoming photons.
- Charge coupled devices (CCD): an initial imaging semiconductor circuit. They are an array of linked capacitors, with pixels represented by MOS capacitors.
- CMOS is much more popular now in image sensors, along with a vast array of other common electronics (microprocessors, microcontrollers, memory chips).

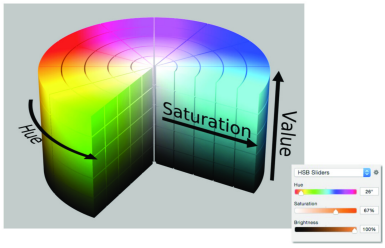


Figure 21.2. HSV has three axes corresponding to artistic characteristics of colors.

A CMOS sensor consists of many pixel sensor photodiodes. One example is in Figure 21.3. It consists of the following components:

- Microlens: focus light onto the photodiode.
- Color filter: allow only one of $\{\text{red}, \text{green}, \text{blue}\}$ to pass through.
- Reset transistor: reset the transistor to a known state before the exposure period.
- Buses: (row) control line and (column) switch, allowing data to be read out.

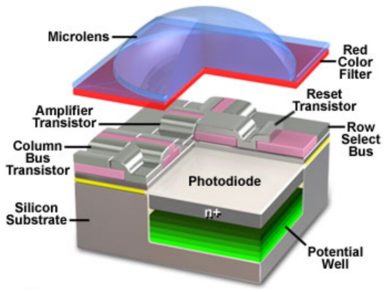


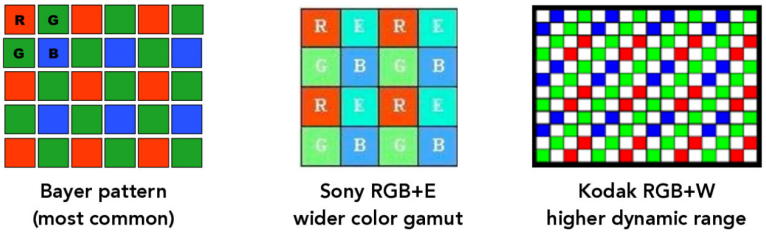
Figure 21.3. Pixel photodiode

- Potential wall: a depletion region, where electron-"hole" pairs are separated and stored as charge carriers. The depth determines the full-well capacity of the photodiode.

There are lots of different possible color architectures. In general, there are more green pixels because humans are more sensitive to green light (as given by the luminous efficiency curve $V(\lambda)$).

Now, given this color architecture, actual pixel values in images are given by **demosaicking** algorithms, i.e. a way to interpolate between pixels. (As a result of this, the majority of images we see are actually made up!) A naive approach is just to lerp, but this results in some noisiness; these days, we tend to use a convnet to do this instead.

Figure 21.4. Color architectures



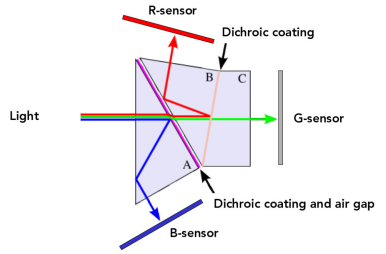


Figure 21.5. Internals of a dichoric prism

Another approach is to use prismatic optics. One option is to use a dichoric prism, as in Figure 21.5, to explicitly collect the RGB values at each pixel with three independent sensors. Thus, no demosaicking is required, at the cost of more (and smaller) sensors to maintain the same resolution.

In silicon, the photoelectric effect gives that the CMOS response function from photons to electrons is linear (though near 0 there is some potential noise). In practice, however, real sensors saturate (due to the description of the potential well we described above), and the signal gets capped after some light density threshold. The result is something we've all seen, e.g. in Figure 21.6.

Figure 21.6. Oversaturation

To fix this, there are two common approaches:

- Take multiple photos at different exposures. Then use Photoshop or some other software to process the images after the fact.
- Use pixel mosaicking (the modern approach). These cameras are higher-end, and can capture multiple exposures at once, then use a learned model to merge this data (can be executed either on the camera or externally).

Usually, around 50% of the board is actually covered with photodi-

odes (this is known as the **fill factor**), and per-pixel microlenses handle the rest.

22 Image sensors & Image processing (04/04)

Image sensors

The lecture slides are here.

In general, there are many sorts of noise that could impact an image. Typically, we quantify it as

$$\text{SNR} = \frac{\text{mean pixel value}}{\text{standard deviation of pixel value}} = \frac{\mu}{\sigma}.$$

(Alternatively, you might hear SNR being used in dB, i.e. $20\log_{10}(\mu/\sigma)$.)

We will provide an overview of these many sources of noise.

- **Pixel structure and micro optics**: One problem with lenses in a standard CMOS architecture is that light will enter one lens and exit at another sensor, known as **cross-talk** (see Figure 22.1). The issue is that since adjacent pixels have differently-colored lenses, colors will become desaturated (see Figure 22.3).

  A fix is to use back-side illumination (see Figure 22.2), with the idea to increase the amount of light captured and improve low-light performance. The more traditional approach, front-side illumination, is constructed in a fashion similar to the human eye.

- **Pixel aliasing**: Pixels get aliased at the sensor level. Some of the pixel board is dedicated to photodiodes, while some is non-photosensitive circuitry (as we discussed, maybe around 50% of a board is actually photoreceptors). This non-uniform sampling, particularly with color filter array paterns, can cause aliasing. A fix is to use an optical low-pass filter, where you can use two layers of birefrigerant material oriented orthogonally such that you split each ray into 2×2 subsamples.

- **Photon shot noise**: This noise is unavoidable. It is given by the independent arrivals of photons during an exposure, which can be modeled by a Poisson distribution. In this case, $\text{SNR} = \mu/\sigma = \lambda/\sqrt{\lambda} = \sqrt{\lambda}$.

- The sensor itself has some noise too, due to physical limits and manufacturing variation.

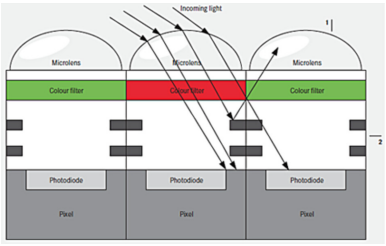


Figure 22.1. Cross-talk

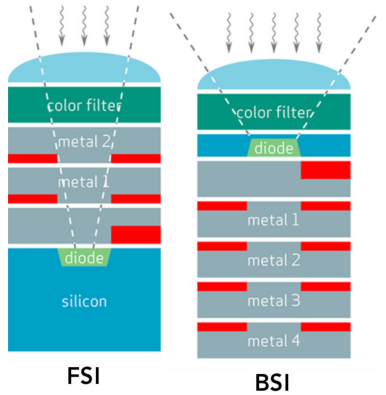


Figure 22.2. Front- vs back- side illumination

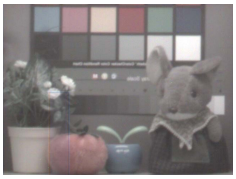


Figure 22.3. Desaturation

- Dark current: Random thermal activity causes electrons to get dislodged, which increases linearly with exposure time and exponentially with temperature.
- Electrons may leak into the well, with number proportional to exposure time.
- Manufacturing variations are common in CMOS sensors, and remains fixed with respect to other factors. Typically, these are handled by accounting for this noise and subtracting it.
- Sensor values may be misread due to thermal noise. The most common solution is cooling.

JPEG image processing

The lecture slides are here.

Step 1: Y'CbCr color space We rely on some image priors:

- Low-frequency content is predominant in most images.
- Humans are less sensitive to high-frequency sources of error and to detail in chromaticity than in luminance.

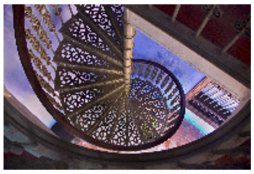
So the idea is to use these as design choices to maximize compression. The Y'CbCr color space has three channels: Y' (luma/lightness), Cb (blue-yellow), and Cr (red-green). In Figure 22.4, we see that these observations generally match, where downsampling in CbCr is much less noticeable than downsampling in Y'.

Step 2: Downsampling Some common representations are as follows:

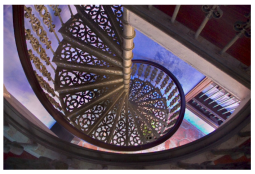
- 4:2:2 means retaining 2/3 of the values. Y' is stored at full resolution as usual, while Cb, Cr are stored at half resolution horizontally.
- 4:2:0 means retaining 1/2 of the values. Y' is stored at full resolution as usual, while Cb, Cr are stored at half resolution in both dimensions.

(This step loses information.)

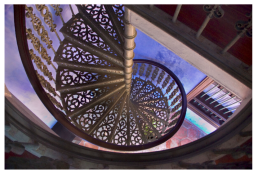
Step 3: Encode channel-wise Typically, this is implemented with a **discrete cosine transformation** (DCT). That is, for each 8×8 block of image values,



(a) 4×4 downsampled Y', full-resolution CbCr



(b) Full-resolution Y', 4×4 downsampled CbCr



(c) Original

Figure 22.4. Different mechanisms for compression

1. Project them onto the basis of 64 basis functions defined by

$$\text{basis}[i,j] = \cos\left(\pi \frac{i}{N}\left(x+\frac{1}{2}\right)\right) \cdot \cos\left(\pi \frac{j}{N}\left(y+\frac{1}{2}\right)\right)$$

for $0 \leq i, j \leq 7$. These are visualized in Figure 22.5.

Remark. This approach often results in noticeable errors along block boundaries, particularly across large color gradients, e.g. text. (But low-frequency regions are usually represented pretty well.)

2. Now, we have a matrix of DCT coefficients. We floordivide it entrywise by a **quantization matrix** (which is just an 8×8 matrix of constants determined empirically from visual perception principles, and assigns lower numbers [or, equivalently, higher weight] to lower frequencies). Note that due to the floordivide, this process, called **quantization**, produces small values for most coefficients on the order of 6 bits, and zeros out most of them.

3. Now, we'd like to store these quantized DCT values, with the key insight that most of them are zeros. We use the optimal encoding under the information theory formulation. That is:

   - Consider a reasonable reordering of these values. In our case, we use diagonal zigzags. Referring back to Figure 22.5, this makes some sense intuitively, since (i, j) along the same diagonal handle roughly the same frequencies.
   - Consider pairs given by the length of a run of zeros and the nonzero value after the run ends.
   - Encode these pairs with Huffman.

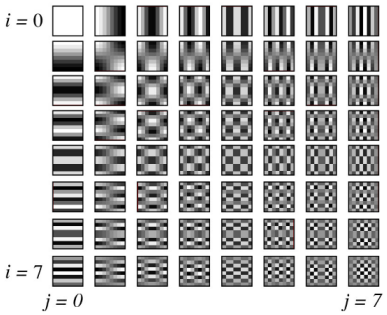


Figure 22.5. Discrete cosine transform

Filters

There are many ways to process an image. The simplest type is a convolution, where each output entry is a linear combination of the local input. Table 22.1 lists some common examples.

Table 22.1. Common image filters

Category	Type	Matrix
Blur	3×3 box	$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$
Blur	Gaussian	$f(i,j) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{i^2+j^2}{2\sigma^2}\right)$ (truncate)
Sharpen	3×3	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
Edge detection	Horizontal	$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
Edge dtection	Vertical	$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
Edge dtection	Sobel	$\sqrt{G_x^2 + G_y^2}$ (applied entrywise)

Since we are working with large images of size $W \times H$ and potentially large filters of size $N \times N$, one potential concern is runtime. There are a couple of ways of handling this:

- **Naive approach:** Doing each of the adds and multiplies takes $O(N^2WH)$.
- **Separable filters:** A key benefit of the two blur filters is that they are **separable**, i.e. they can be written as the product of two other filters. The 3×3 box blur can be written as

$$\frac{1}{3}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \cdot \frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \end{bmatrix},$$

  and Gaussian blur can be written as

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{i^2}{2\sigma^2}\right) \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{j^2}{2\sigma^2}\right).$$

  This works great because this approach runs in $O(NWH)$ (with some minimal memory overhead).
- Recall the convolution theorem (Theorem 3.3). Suppose our image has size $M \times M$, i.e. $M = H = W$, for simplicity. One can check that the runtime is $O(M^2 \log M)$ due to the overhead of FFT, compared to $O(N^2M^2)$ in the spatial domain. Thus, doing computations in the frequency domain is better when the filter is large and worse when the filter is small.

23 Image processing & VR (04/09)

The lecture slides are here.

In some cases, it is beneficial to use data-dependent filters, which can alleviate issues with outliers. Some examples are as follows:

- **Median filter:** Replace a pixel with the median of its neighbors. A benefit is noise reduction, i.e. unlike Gaussian blur, one bright pixel doesn't bias a large region. In practice, however, it is quite inefficient, since the filter is neither linear nor separable.
- **Bilateral filter:** Define

$$\mathrm{BF}[I](x,y) = \sum_{i,j} f(\|I(x-i,y-j) - I(x,y)\|)G(i,j)I(x-i,y-j),$$

  where:

  - The sum is over all i, j in the support of the Gaussian kernel,
  - $f(\cdot)$ is a spatial weight term (e.g. a Gaussian or a distance threshold),
  - I is the original image,
  - G is a Gaussian blur kernel.

  That is, the weight is a function of the spatial distance and intensity difference. Intuitively, pixels with color very far off from the target pixel are probably on the opposite side of the edge, so their contribution to the weighted sum should be small. See Figure 23.1.

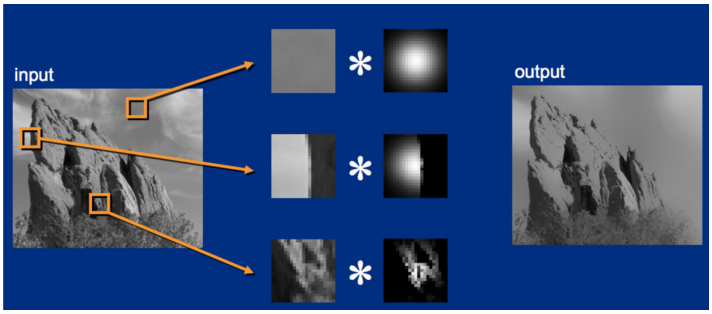


Figure 23.1. Kernel depends on image content

These days, we are more interested in using data-driven methods for image synthesis. A canonical example in graphics is texture synthesis, where we are given a low-resolution texture image and are asked to produce a high-resolution texture that "looks like" the input.

Let's formulate this more rigorously. Suppose we have a pixel p, and let N_p be the $N \times N$ neighborhood around p. Our objective is to

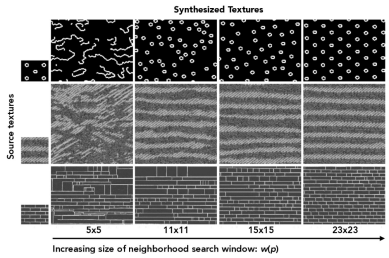


Figure 23.2. Non-parametric texture synthesis

approximate the distribution for possible values of p, based on the rest of its neighborhood. Here is an algorithm that has shown to work well in simple settings:

- Take pixels p as the centers of the closest unsynthesized patches.
- To synthesize each p, find $N \times N$ patches $\{N_q\}$ in the image that are most similar to the patch N_p containing p.
- Sample patches from $\{N_q\}$ with weights given by $d(N_p, N_q)$.

The results are visualized in Figure 23.2.

These days, we like to solve this problem with machine learning. We know that a common way of learning useful representations is to mask out some patches and learn to reconstruct them. This works much better in general.

Virtual Reality

The lecture slides are here.

Virtual reality immerses a user completely in a virtual world, whereas augmented reality is a display that augments the user's normal view of the real world. Mixed reality is a blend of VR and AR. The typical approach now (e.g. Apple Vision Pro) is pass-through, i.e. overlay things on top of camera views.

VR Displays

What should physical VR displays support? The physical displays consist of multiple components.

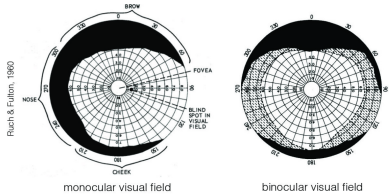


Figure 23.3. FOV for human vision

- Regular 2D camera views have windowed FOV, whereas VR/AR displays provide 360° FOV. Your head's orientation is tracked physically, and your rendered view is synchronized in real time.
- The panels are designed to provide 3D cues: z-buffer, lighting calculations, occlusion, perspective, lens calculations, and so on. They also handle user motion and different views from the left and right eyes.
- The Oculus Quest 2 headset, for instance, supports a fan, 4 high-performance cores, 4 low-performance cores, image and digital signal processors, a GPU that can run a 3k $\times$ 3k display at 90Hz (but the actual headset only has 7M pixels), and can process input streams

from 7 cameras for motion tracking and gesture recognition. A ton of stuff gets packed into this small product to make everything possible!

- The eyepiece lenses are designed to (i) create a wide FOV and (ii) create a focal plane several meters away (for our purposes, at infinity). The point of the latter is that parallel lines reaching the eye should converge on the display at the same point, similarly to what is done on an actual eye.

How many pixels do we need? Much like the displays for our other devices, VR displays that mimic human vision must satisfy properties driven by aspects of human perception.

- Colors must be reproduced as usual.
- Humans have around 160° FOV in each eye, or around 200° with both eyes collectively (before accounting for the eye's ability to move in the socket). See Figure 23.3.
- From the FOV, you can compute the required resolution. Humans can discern changes up to half an arc-minute (1/120 degrees) due to the distance between any two adjacent photoreceptors; by comparison, letters in the 20/20 vision chart occupy around 5 arc-minutes. Given the FOV determined in Figure 23.3, we can find that the required resolution is around $8\text{k} \times 8\text{k}$.

In practice, however, this is quite expensive. The HTC Vive Pro 2 uses an FOV of around 100° per eye, 2448×2448 pixel display, and about 24 pixels per degree (rather than around the 60 required to match 20/20 vision).

What to display? This is a rather challenging problem, since humans have two eyes, which produce different views.

- Naturally, simply superimposing the images will produce double-images. This is expected behavior! If you put your finger in front of your face, focusing on your finger (for most people) produces two background images, and vice-versa.

  Humans' focusing on particular targets is known as **vergence**. They will rotate physically in their sockets to bring closer and further objects into physical alignment on the retina.

  Anatomically, our eyes depend on several muscles to accomplish **verging** (rotation to focus on target angles) and **accommodation** (focusing on target distances).

  - Verging is achieved with **extraocular muscles** (which look like strings pulling around the eyeball), and requires **oculomotor cues**. This is performed passively, to ensure that the projection of the object is centered on the retina.
  - Accommodation is achieved with **ciliary muscles** (which look like strings pulling on the sides of the eyeball); muscle contractions cause the lenses to increase in curvature and enable focusing on nearby objects, and relaxing does the opposite.
- With this in mind, designing a reasonable VR display is challenging. Users' eyes must remain accommodated to far distances, but the eyes must converge to fuse stereoscopic images of objects up close. This leads to conflicting depth cues, which causes fatigue and nausea.

Remark. This problem is resolved if you can just directly emit the light field that would be produced by a virtual scene.

Remark. As you age, the proteins in your eyeballs harden at a constant rate: you begin with 80 diopters (unit with dimensions m^{-1}) and lose around 2 diopters per year. By the time you're 40, your vision cannot accommodate small focal distances, which is why you'll need reading glasses.

24 Virtual Reality (04/09)

Accounting for motion

Following our discussion from last time, an additional issue we must resolve is tracking the user's motions. The approaches outlined below are depicted in **??**.

Figure 24.1. Google cardboard

Figure 24.2. Environment-supported vision-based tracking

Example 24.1. Google cardboard uses gyro and a rear facing camera to estimate the user's viewpoint. Generally, tracking 2D rotation this way works pretty well, but 3D rotation doesn't.

Example 24.2. Figure 24.2 shows an early VR test room at Valve, where markers similar to QR codes are positioned throughout the environment.

Example 24.3. Oculus rift IR LED tracking has an external 60Hz camera, which consists of an IR filter, camera lens, and CMOS sensor. The photo in Figure 24.3 is taken with an IR-sensitive camera (it doesn't look like that to us). It implements **active optical motion capture**, where each LED marker emits a unique blinking pattern, resolving some ambiguities. The downside is that there is inevitably some lag.

Suppose you have:

- The relative positions of the 3D markers on the headset. Due to active tracking, these are known constants.
- The camera viewpoint (a 4×4 projective mapping from 3D to 2D). Due to the camera remaining in place while the headset is in use, this is also a constant.
- Each frame provides a 2D position of each headset marker in an image.

Now, we'd like to solve for the head pose, which has 6 degrees of freedom (3 for position, 3 for direction). This is pretty easy: just use least-squares.

Figure 24.3. Oculus rift IR LED tracking hardware

Remark. This is in contrast to passive optical motion capture, which we discussed in our animation lecture (Section 18). The downside of passive tracking is that you need multiple cameras to handle occlusions (and who wants to buy those?).

Example 24.4. Figure 24.4 depicts HTC Vive's tracking system. As with Oculus, the headset and controller are covered with IR photodiodes. The light emitter is an array of LEDs along with two spinning wheels with lasers. During each frame, the lighthouse emits an LED pulse, horizontal laser sweep, LED pulse, and vertical laser sweep. The photodiode measures the time offset between the LED pulse and the laser arrival, which determines the x and y offset in the lighthouse's field of view. So in some sense, the lighthouse acts as a virtual camera.

Example 24.5. Nowadays, the external camera is too clunky, and we tend to use "inside-out" tracking. Wide angle cameras look outward from the headset, and we use SLAM to estimate the 3D structure of the world and the position and orientation of the viewer, along with the position and orientation of the controllers (which, in the case of Meta Quest 2, also have infrared LEDs to help with tracking).

Rendering challenges

The previous section tackled the problem of the camera view changing. We also need to handle the problem of doing rendering well, in terms of correctness and speed.

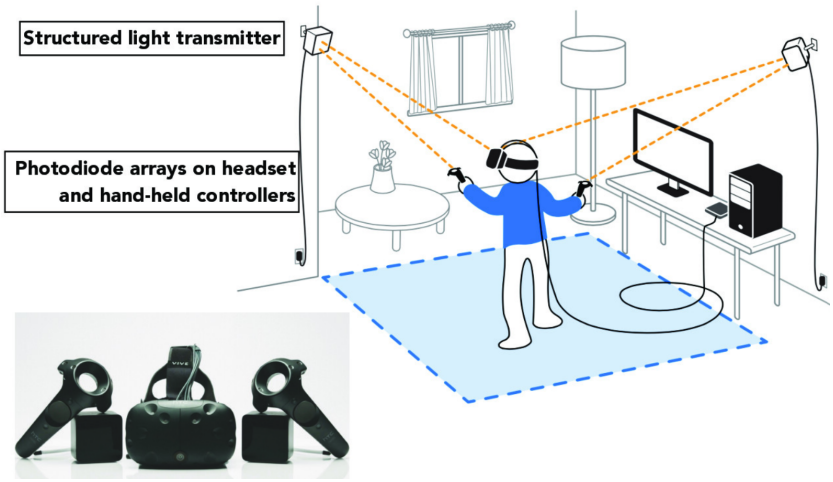


Figure 24.4. HTC Vive tracking system

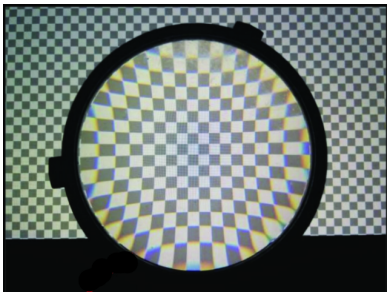


Figure 24.5. View of checkerboard through Oculus Rift lens

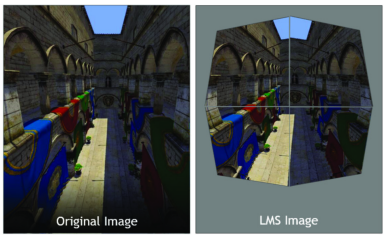


Figure 24.6. Lens-matched shading

Example 24.6. VR headsets require wide FOV. As a result, the lens introduces distortion, namely **pincushion distortion** due to the shape of the lens and **chromatic aberration** due to different wavelengths of light refracting by different amounts. See Figure 24.5. Fixing this turns out to be pretty easy. For pincushion, just do the usual interpolation we need for putting a texture on a mesh. For chromatic aberration, you can use separate distortions for each channel R, G, B to get a decent approximation.

Example 24.7. Projecting things onto planes doesn't quite work, since objects in the periphery will get stretched (pixels span a larger angle in the center of the image). Some solutions have included curved displays, adjusted ray casting, and rendering scenes with four viewports with distinct projection matrices (see Figure 24.6).

Example 24.8. Arguably the most difficult challenge with VR is optimizing end-to-end latency, i.e. the time from moving your head to new photons arriving at your eyes, which includes updating head movement, camera position, rendering the new image, transferring it to the headset, and displaying it. Ideally, a VR system will accomplish all of these, end-to-end, in 10-25ms.

Consider the following case study: Suppose we have a 2k $\times$ 2k display spanning 100° FOV. If you move your head 90° in one second (a fairly reasonable speed), and your end-to-end latency is 33ms, then the displayed image is off by around 60 pixels, or 3°, from the ground truth with zero latency.

This problem can be alleviated with **foveated rendering**, i.e. we al-

locate our resources to where they're most useful. The eye can only perceive detail in a $\sim 5^\circ$ region about the gaze point, which enables us to focus our compute on a very small region. The practical implementation turns out to be quite simple: just maintain three images (one high-res, one medium-res, and one low-res), and blend them for the display.

Example 24.9. Suppose a user's eye is moving to track a moving object.[10] That is, the eye is moving continuously relative to the display, as shown in Figure 24.7. The problem is that the image lags behind the eye motion, which results in **judder**. There are two solutions to this:

- Increase the frame rate. This can be costly.
- Use a low-persistence display. That is, the pixels should only emit light for a small fraction of the frame. If this flickering occurs at a rate higher than around 60Hz, it is imperceptible to the human eye. Oculus DK2 OLED does this: their 75Hz frame rate means that each frame lasts around 13ms, and the pixel persistence is only around 2-3ms.

[10] To actually implement this, add inward-facing cameras to image the eyes and face. One line of work here is enabling VR teleconference, which requires detecting gaze points for UIs and facial expressions.

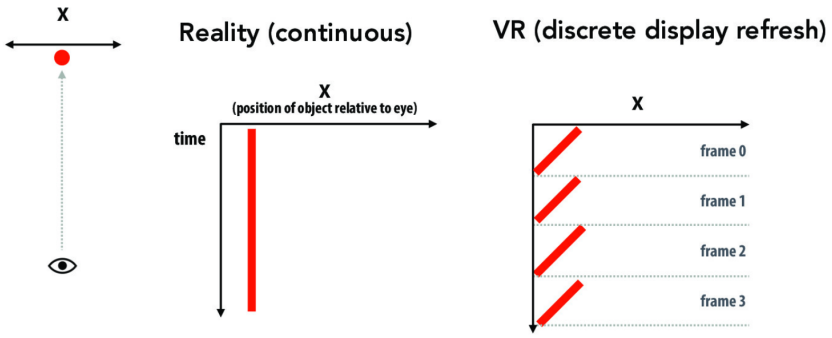


Figure 24.7. Eye moving to track moving object

Example 24.10. Producing high-quality renderings is hard, and increasing the latency will cause motion sickness. **Reprojection** aims to mitigate this problem. The GPU renders a new frame as usual, using the latest position/orientation data. In parallel, motion tracking continues, and if the frame is not generated in time, the most recently rendered frame is reprojected, warped to match the current head position and orientation, and displayed. After the next frame is rendered, it is displayed.

Imaging

There have been lots of problems with naively stitching together views from multiple cameras. In particular, doing this with two spherical cameras is pretty hard. Suppose your two cameras are a distance b apart (called a **baseline**). Then they both rotate by angle θ. Now they are effectively distance $b\cos\theta$ apart. Furthermore, one camera is in view of the other, causing occlusion.

There have been lots of attempts to hack this problem. The natural solution to incorrect baselines is to spin the cameras around a circle, rather than fixing their positions. With this in mind, you might as well rotate them all the way around the circle to produce panoramic views, i.e. create an omni-directional stereo representation.

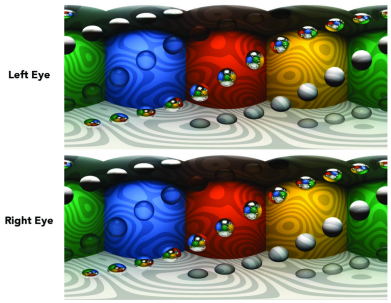


Figure 24.8. Left and right views

Now, to get the two desired views back, project this panoramic view onto the inside of a virtual sphere, and render the left and right eye viewpoints independently with corrections for distortions and chromatic aberration.

This approach works pretty well (see Figure 24.8)! This representation is only an approximation, however; straight lines may appear slightly curve, and vertical disparities for close objects may be wrong.

Example 24.11. A specific case of moving-viewpoint imaging is with 4D light fields. Suppose we'd like a camera to capture views sufficiently densely such that we can capture depth well. That is, consider the following arrangement: a camera and two points at depths $z_n = 0.3$m and $z_f = 0.6$m are collinear. If you consider the view Δx to the right of the center of the camera, you should be able to distinguish the close and far features.

Previously, we saw that 20/20 vision requires that $\theta \approx (1/60)^\circ$. In this case, we require $\Delta x \approx (1/1719)$ft, or millions of views per square foot. Current VR devices allow for $\theta \approx (1/10)^\circ$, or $\Delta x \approx (1/286)$ft, or under a hundred thousand views per square foot.

Multi-camera arrays and plenoptic cameras are much worse at this, allowing on the order of 100 views.

Recently, work on NeRF has allowed for novel view interpolation: sparsely sample images of a scene, and get out new views of the same scene. But it is still much too slow to be of use for VR. There is a lot of work to be done here!

References

This section is unfortunately brief because I'm too lazy to backfill references.

J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23(3):271–280, jul 1989. ISSN 0097-8930. DOI: 10.1145/74334.74361. URL https://doi.org/10.1145/74334.74361.

James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986. ISSN 0097-8930. DOI: 10.1145/15886.15902. URL https://doi.org/10.1145/15886.15902.

Stephen R Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. Light scattering from human hair fibers. *ACM Transactions on Graphics (TOG)*, 22(3):780–791, 2003.

Ling-Qi Yan, Miloš Hašan, Wenzel Jakob, Jason Lawrence, Steve Marschner, and Ravi Ramamoorthi. Rendering glints on high-resolution normal-mapped specular surfaces. *ACM Transactions on Graphics (TOG)*, 33(4):1–9, 2014.

Ling-Qi Yan, Miloš Hašan, Steve Marschner, and Ravi Ramamoorthi. Position-normal distributions for efficient rendering of specular microstructure. *ACM Transactions on Graphics (TOG)*, 35(4):1–9, 2016.

Index