Computer Graphics Lecture Notes

CSC418 / CSCD18 / CSC2504

Computer Science Department
University of Toronto

Version: November 24, 2006

Contents

Conventions and Notation

Vectors have an arrow over their variable name: $\vec{v}$. Points are denoted with a bar instead: $\bar{p}$. Matrices are represented by an uppercase letter.

When written with parentheses and commas separating elements, consider a vector to be a column vector. That is, $(x, y) = \begin{bmatrix} x \\ y \end{bmatrix}$. Row vectors are denoted with square braces and no commas: $\begin{bmatrix} x & y \end{bmatrix} = (x, y)^T = \begin{bmatrix} x \\ y \end{bmatrix}^T$.

The set of real numbers is represented by $\mathbb{R}$. The real Euclidean plane is $\mathbb{R}^2$, and similarly Euclidean three-dimensional space is $\mathbb{R}^3$. The set of natural numbers (non-negative integers) is represented by $\mathbb{N}$.

There are some notable differences between the conventions used in these notes and those found in the course text. Here, coordinates of a point $\bar{p}$ are written as p_x, p_y, and so on, where the book uses the notation x_p, y_p, etc. The same is true for vectors.

> *Aside:*
> Text in "aside" boxes provide extra background or information that you are not required to know for this course.

Acknowledgements

Thanks to Tina Nicholl for feedback on these notes. Alex Kolliopoulos assisted with electronic preparation of the notes, with additional help from Patrick Coleman.

1 Introduction to Graphics

1.1 Raster Displays

The screen is represented by a 2D array of locations called **pixels**.

Zooming in on an image made up of pixels

The convention in these notes will follow that of OpenGL, placing the origin in the lower left corner, with that pixel being at location $(0, 0)$. Be aware that placing the origin in the upper left is another common convention.

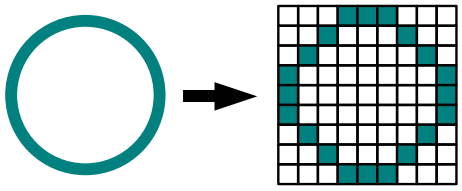
One of 2^N intensities or colors are associated with each pixel, where N is the number of bits per pixel. Greyscale typically has one byte per pixel, for $2^8 = 256$ intensities. Color often requires one byte per channel, with three color channels per pixel: red, green, and blue.

Color data is stored in a **frame buffer**. This is sometimes called an image map or bitmap.

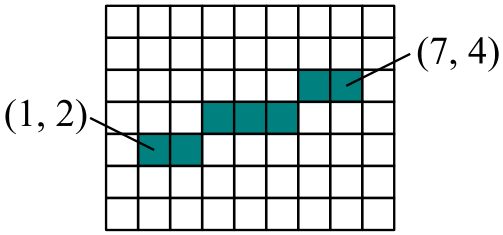
Primitive operations:

- `setpixel(x, y, color)`
  Sets the pixel at position (x, y) to the given color.
- `getpixel(x, y)`
  Gets the color at the pixel at position (x, y).

Scan conversion is the process of converting basic, low level objects into their corresponding pixel map representations. This is often an approximation to the object, since the frame buffer is a discrete grid.



Scan conversion of a circle



1.2 Basic Line Drawing

Set the color of pixels to approximate the appearance of a line from (x_0, y_0) to (x_1, y_1).
It should be

- "straight" and pass through the end points.
- independent of point order.
- uniformly bright, independent of slope.

The explicit equation for a line is $y = mx + b$.

> *Note:*
> Given two points (x_0, y_0) and (x_1, y_1) that lie on a line, we can solve for m and b for the line. Consider $y_0 = mx_0 + b$ and $y_1 = mx_1 + b$.
> Subtract y_0 from y_1 to solve for $m = \frac{y_1 - y_0}{x_1 - x_0}$ and $b = y_0 - mx_0$.
> Substituting in the value for b, this equation can be written as $y = m(x - x_0) + y_0$.

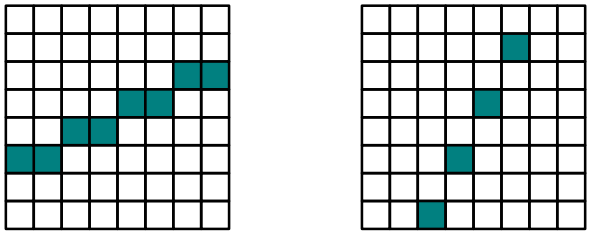
Consider this simple line drawing algorithm:

```
int x
float m, y
m = (y1 - y0) / (x1 - x0)
for (x = x0; x <= x1; ++x) {
  y = m * (x - x0) + y0
  setpixel(x, round(y), linecolor)
}
```


Problems with this algorithm:

- If $x_1 < x_0$ nothing is drawn.
  Solution: Switch the order of the points if $x_1 < x_0$.

- Consider the cases when $m < 1$ and $m > 1$:



(a) $m < 1$ (b) $m > 1$

  A different number of pixels are on, which implies different brightness between the two.
  Solution: When $m > 1$, loop over $y = y_0 \ldots y_1$ instead of x, then $x = \frac{1}{m}(y - y_0) + x_0$.

- Inefficient because of the number of operations and the use of floating point numbers.
  Solution: A more advanced algorithm, called Bresenham's Line Drawing Algorithm.

2 Curves

2.1 Parametric Curves

There are multiple ways to represent curves in two dimensions:

- **Explicit:** $y = f(x)$, given x, find y.

  Example:
  The explicit form of a line is $y = mx + b$. There is a problem with this representation–what about vertical lines?

- **Implicit:** $f(x, y) = 0$, or in vector form, $f(\bar{p}) = 0$.

  Example:
  The implicit equation of a line through $\bar{p}_0$ and $\bar{p}_1$ is

  $$(x - x_0)(y_1 - y_0) - (y - y_0)(x_1 - x_0) = 0.$$

  Intuition:
  - The direction of the line is the vector $\vec{d} = \bar{p}_1 - \bar{p}_0$.
  - So a vector from $\bar{p}_0$ to any point on the line must be parallel to $\vec{d}$.
  - Equivalently, any point on the line must have direction from $\bar{p}_0$ perpendicular to $\vec{d}^{\perp} = (d_y, -d_x) \equiv \vec{n}$.
    This can be checked with $\vec{d} \cdot \vec{d}^{\perp} = (d_x, d_y) \cdot (d_y, -d_x) = 0$.
  - So for any point $\bar{p}$ on the line, $(\bar{p} - \bar{p}_0) \cdot \vec{n} = 0$.
    Here $\vec{n} = (y_1 - y_0, x_0 - x_1)$. This is called a **normal**.
  - Finally, $(\bar{p} - \bar{p}_0) \cdot \vec{n} = (x - x_0, y - y_0) \cdot (y_1 - y_0, x_0 - x_1) = 0$. Hence, the line can also be written as:

    $$(\bar{p} - \bar{p}_0) \cdot \vec{n} = 0$$

  Example:
  The implicit equation for a circle of radius r and center $\bar{p}_c = (x_c, y_c)$ is

  $$(x - x_c)^2 + (y - y_c)^2 = r^2,$$

  or in vector form,

  $$\|\bar{p} - \bar{p}_c\|^2 = r^2.$$

- **Parametric:** $\bar{p} = \bar{f}(\lambda)$ where $\bar{f} : \mathbb{R} \rightarrow \mathbb{R}^2$, may be written as $\bar{p}(\lambda)$ or $(x(\lambda), y(\lambda))$.

Example:
A parametric line through $\bar{p}_0$ and $\bar{p}_1$ is

$$\bar{p}(\lambda) = \bar{p}_0 + \lambda\vec{d},$$

where $\vec{d} = \bar{p}_1 - \bar{p}_0$.
Note that bounds on λ must be specified:

- Line segment from $\bar{p}_0$ to $\bar{p}_1$: $0 \leq \lambda \leq 1$.
- Ray from $\bar{p}_0$ in the direction of $\bar{p}_1$: $0 \leq \lambda < \infty$.
- Line passing through $\bar{p}_0$ and $\bar{p}_1$: $-\infty < \lambda < \infty$

Example:
What's the perpendicular bisector of the line segment between $\bar{p}_0$ and $\bar{p}_1$?

- The midpoint is $\bar{p}(\lambda)$ where $\lambda = \frac{1}{2}$, that is, $\bar{p}_0 + \frac{1}{2}\vec{d} = \frac{\bar{p}_0 + \bar{p}_1}{2}$.
- The line perpendicular to $\bar{p}(\lambda)$ has direction parallel to the normal of $\bar{p}(\lambda)$, which is $\vec{n} = (y_1 - y_0, -(x_1 - x_0))$.

Hence, the perpendicular bisector is the line $\ell(\alpha) = \left(\bar{p}_0 + \frac{1}{2}\vec{d}\right) + \alpha\vec{n}$.

Example:
Find the intersection of the lines $\bar{l}(\lambda) = \bar{p}_0 + \lambda\vec{d}_0$ and $f(\bar{p}) = (\bar{p} - \bar{p}_1) \cdot \vec{n}_1 = 0$.

Substitute $\bar{l}(\lambda)$ into the implicit equation $f(\bar{p})$ to see what value of λ satisfies it:

$$\begin{aligned} f\left(\bar{l}(\lambda)\right) &= \left(\bar{p}_0 + \lambda\vec{d}_0 - \bar{p}_1\right) \cdot \vec{n}_1 \\ &= \lambda\vec{d}_0 \cdot \vec{n}_1 - (\bar{p}_1 - \bar{p}_0) \cdot \vec{n}_1 \\ &= 0 \end{aligned}$$

Therefore, if $\vec{d}_0 \cdot \vec{n}_1 \neq 0$,

$$\lambda^* = \frac{(\bar{p}_1 - \bar{p}_0) \cdot \vec{n}_1}{\vec{d}_0 \cdot \vec{n}_1},$$

and the intersection point is $\bar{l}(\lambda^*)$. If $\vec{d}_0 \cdot \vec{n}_1 = 0$, then the two lines are parallel with no intersection or they are the same line.

Example:
The parametric form of a circle with radius r for $0 \leq \lambda < 1$ is

$$\bar{p}(\lambda) = (r\cos(2\pi\lambda), r\sin(2\pi\lambda)).$$

This is the polar coordinate representation of a circle. There are an infinite number of parametric representations of most curves, such as circles. Can you think of others?

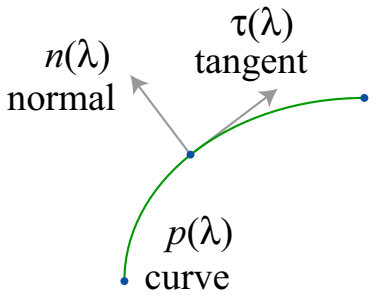
An important property of parametric curves is that it is easy to generate points along a curve by evaluating $\bar{p}(\lambda)$ at a sequence of λ values.

2.1.1 Tangents and Normals

The **tangent** to a curve at a point is the instantaneous direction of the curve. The line containing the tangent intersects the curve at a point. It is given by the derivative of the parametric form $\bar{p}(\lambda)$ with regard to λ. That is,

$$\vec{\tau}(\lambda) = \frac{d\bar{p}(\lambda)}{d\lambda} = \left(\frac{dx(\lambda)}{d\lambda}, \frac{dy(\lambda)}{d\lambda}\right).$$

The **normal** is perpendicular to the tangent direction. Often we normalize the normal to have unit length. For closed curves we often talk about an inward-facing and an outward-facing normal. When the type is unspecified, we are usually dealing with an outward-facing normal.



We can also derive the normal from the implicit form. The normal at a point $\bar{p} = (x, y)$ on a curve defined by $f(\bar{p}) = f(x, y) = 0$ is:

$$\vec{n}(\bar{p}) = \nabla f(\bar{p})|_{\bar{p}} = \left(\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right)$$

Derivation:
For any curve in implicit form, there also exists a parametric representation $\bar{p}(\lambda) =$

$(x(\lambda), y(\lambda))$. All points on the curve must satisfy $f(\bar{p}) = 0$. Therefore, for any choice of λ, we have:

$$0 = f(x(\lambda), y(\lambda))$$

We can differentiate both side with respect to λ:

$$\begin{aligned} 0 &= \frac{d}{d\lambda} f(x(\lambda), y(\lambda)) && (1) \\ 0 &= \frac{\partial f}{\partial x}\frac{dx(\lambda)}{d\lambda} + \frac{\partial f}{\partial y}\frac{dy(\lambda)}{d\lambda} && (2) \\ 0 &= \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \cdot \left(\frac{dx(\lambda)}{d\lambda}, \frac{dy(\lambda)}{d\lambda}\right) && (3) \\ 0 &= \nabla f(\bar{p})|_{\bar{p}} \cdot \vec{\tau}(\lambda) && (4) \end{aligned}$$

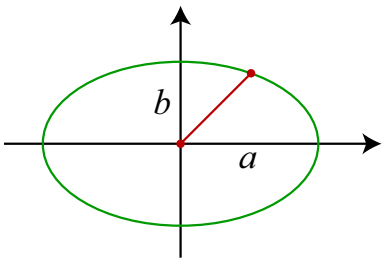
This last line states that the gradient is perpendicular to the curve tangent, which is the definition of the normal vector.

Example:
The implicit form of a circle at the origin is: $f(x, y) = x^2 + y^2 - R^2 = 0$. The normal at a point (x, y) on the circle is: $\nabla f = (2x, 2y)$.

Exercise: show that the normal computed for a line is the same, regardless of whether it is computed using the parametric or implicit forms. Try it for another surface.

2.2 Ellipses

- **Implicit:** $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$. This is only for the special case where the ellipse is centered at the origin with the major and minor axes aligned with $y = 0$ and $x = 0$.



- **Parametric:** $x(\lambda) = a\cos(2\pi\lambda)$, $y(\lambda) = b\sin(2\pi\lambda)$, or in vector form

$$\bar{p}(\lambda) = \begin{bmatrix} a\cos(2\pi\lambda) \\ b\sin(2\pi\lambda) \end{bmatrix}.$$

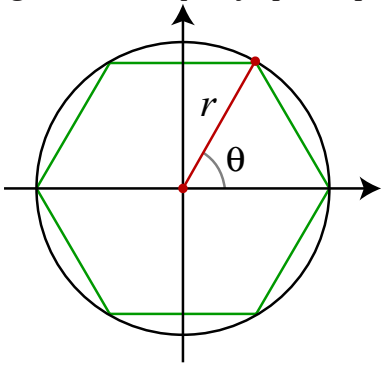
The implicit form of ellipses and circles is common because there is no explicit functional form. This is because y is a multifunction of x.

2.3 Polygons

A **polygon** is a continuous, piecewise linear, closed planar curve.

- A **simple** polygon is non self-intersecting.
- A **regular** polygon is simple, equilateral, and equiangular.
- An **n-gon** is a regular polygon with n sides.
- A polygon is **convex** if, for any two points selected inside the polygon, the line segment between them is completely contained within the polygon.

Example:
To find the vertices of an n-gon, find n equally spaced points on a circle.



In polar coordinates, each vertex $(x_i, y_i) = (r\cos(\theta_i), r\sin(\theta_i))$, where $\theta_i = i\frac{2\pi}{n}$ for $i = 0 \ldots n-1$.

- To translate: Add (x_c, y_c) to each point.
- To scale: Change r.
- To rotate: Add $\Delta\theta$ to each θ_i.

2.4 Rendering Curves in OpenGL

OpenGL does not directly support rendering any curves other that lines and polylines. However, you can sample a curve and draw it as a line strip, e.g.,:

```
float x, y;
glBegin(GL_LINE_STRIP);
for (int t=0 ; t <= 1 ; t += .01)
```



```
    computeCurve( t, &x, &y);
    glVertex2f(x, y);
}
glEnd();
```

You can adjust the step-size to determine how many line segments to draw. Adding line segments will increase the accuracy of the curve, but slow down the rendering.

The GLU does have some specialized libraries to assist with generating and rendering curves. For example, the following code renders a disk with a hole in its center, centered about the z-axis.

```
GLUquadric q = gluNewQuadric();
gluDisk(q, innerRadius, outerRadius, sliceCount, 1);
gluDeleteQuadric(q);
```

See the OpenGL Reference Manual for more information on these routines.

3 Transformations

3.1 2D Transformations

Given a point cloud, polygon, or sampled parametric curve, we can use transformations for several purposes:

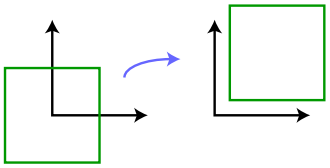
1. Change coordinate frames (world, window, viewport, device, etc).
2. Compose objects of simple parts with local scale/position/orientation of one part defined with regard to other parts. For example, for articulated objects.
3. Use deformation to create new shapes.
4. Useful for animation.

There are three basic classes of transformations:

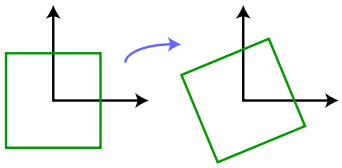
1. **Rigid body** - Preserves distance and angles.
   - Examples: translation and rotation.
2. **Conformal** - Preserves angles.
   - Examples: translation, rotation, and uniform scaling.
3. **Affine** - Preserves parallelism. Lines remain lines.
   - Examples: translation, rotation, scaling, shear, and reflection.

Examples of transformations:

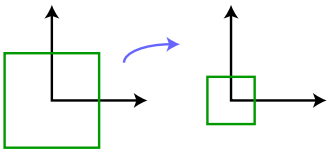
- **Translation** by vector $\vec{t}$: $\bar{p}_1 = \bar{p}_0 + \vec{t}$.



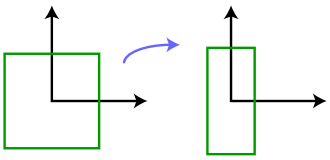
- **Rotation** counterclockwise by θ: $\bar{p}_1 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \bar{p}_0$.



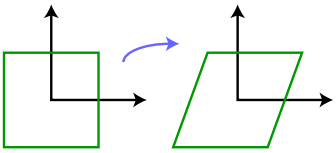
- **Uniform scaling** by scalar a: $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \bar{p}_0$.



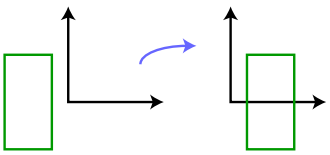
- **Nonuniform scaling** by a and b: $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \bar{p}_0$.



- **Shear** by scalar h: $\bar{p}_1 = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \bar{p}_0$.



- **Reflection** about the y-axis: $\bar{p}_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \bar{p}_0$.



3.2 Affine Transformations

An **affine transformation** takes a point $\bar{p}$ to $\bar{q}$ according to $\bar{q} = F(\bar{p}) = A\bar{p} + \vec{t}$, a linear transformation followed by a translation. You should understand the following proofs.

- The inverse of an affine transformation is also affine, assuming it exists.

Proof:
Let $\bar{q} = A\bar{p} + \vec{t}$ and assume A^{-1} exists, i.e. $\det(A) \neq 0$.
Then $A\bar{p} = \bar{q} - \vec{t}$, so $\bar{p} = A^{-1}\bar{q} - A^{-1}\vec{t}$. This can be rewritten as $\bar{p} = B\bar{q} + \vec{d}$, where $B = A^{-1}$ and $\vec{d} = -A^{-1}\vec{t}$.

Note:
The inverse of a 2D linear transformation is

$$A^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

- Lines and parallelism are preserved under affine transformations.

Proof:
To prove lines are preserved, we must show that $\bar{q}(\lambda) = F(\bar{l}(\lambda))$ is a line, where $F(\bar{p}) = A\bar{p} + \vec{t}$ and $\bar{l}(\lambda) = \bar{p}_0 + \lambda\vec{d}$.

$$\begin{aligned} \bar{q}(\lambda) &= A\bar{l}(\lambda) + \vec{t} \\ &= A(\bar{p}_0 + \lambda\vec{d}) + \vec{t} \\ &= (A\bar{p}_0 + \vec{t}) + \lambda A\vec{d} \end{aligned}$$

This is a parametric form of a line through $A\bar{p}_0 + \vec{t}$ with direction $A\vec{d}$.

- Given a closed region, the area under an affine transformation $A\bar{p} + \vec{t}$ is scaled by $\det(A)$.

Note:

- Rotations and translations have $\det(A) = 1$.
- Scaling $A = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$ has $\det(A) = ab$.
- Singularities have $\det(A) = 0$.

Example:
The matrix $A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ maps all points to the x-axis, so the area of any closed region will become zero. We have $\det(A) = 0$, which verifies that any closed region's area will be scaled by zero.

- A composition of affine transformations is still affine.

Proof:
Let $F_1(\bar{p}) = A_1\bar{p} + \vec{t}_1$ and $F_2(\bar{p}) = A_2\bar{p} + \vec{t}_2$.
Then,

$$\begin{aligned} F(\bar{p}) &= F_2(F_1(\bar{p})) \\ &= A_2(A_1\bar{p} + \vec{t}_1) + \vec{t}_2 \\ &= A_2A_1\bar{p} + (A_2\vec{t}_1 + \vec{t}_2). \end{aligned}$$

Letting $A = A_2A_1$ and $\vec{t} = A_2\vec{t}_1 + \vec{t}_2$, we have $F(\bar{p}) = A\bar{p} + \vec{t}$, and this is an affine transformation.

3.3 Homogeneous Coordinates

Homogeneous coordinates are another way to represent points to simplify the way in which we express affine transformations. Normally, bookkeeping would become tedious when affine transformations of the form $A\bar{p} + \vec{t}$ are composed. With homogeneous coordinates, affine transformations become matrices, and composition of transformations is as simple as matrix multiplication. In future sections of the course we exploit this in much more powerful ways.

With homogeneous coordinates, a point $\bar{p}$ is augmented with a 1, to form $\hat{p} = \begin{bmatrix} \bar{p} \\ 1 \end{bmatrix}$.

All points $(\alpha\bar{p}, \alpha)$ represent the same point $\bar{p}$ for real $\alpha \neq 0$.

Given $\hat{p}$ in homogeneous coordinates, to get $\bar{p}$, we divide $\hat{p}$ by its last component and discard the last component.

Example:
The homogeneous points $(2, 4, 2)$ and $(1, 2, 1)$ both represent the Cartesian point $(1, 2)$. It's the orientation of $\hat{p}$ that matters, not its length.

Many transformations become linear in homogeneous coordinates, including affine transformations:

$$\begin{aligned} \begin{bmatrix} q_x \\ q_y \end{bmatrix} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \\ &= \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} A & \vec{t} \end{bmatrix} \hat{p} \end{aligned}$$

To produce $\hat{q}$ rather than $\bar{q}$, we can add a row to the matrix:

$$\hat{q} = \begin{bmatrix} A & \vec{t} \\ \vec{0}^T & 1 \end{bmatrix} \hat{p} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \hat{p}.$$

This is linear! Bookkeeping becomes simple under composition.

Example:
$F_3(F_2(F_1(\bar{p})))$, where $F_i(\bar{p}) = A_i(\bar{p}) + \vec{t}_i$ becomes $M_3 M_2 M_1 \bar{p}$, where $M_i = \begin{bmatrix} A_i & \vec{t}_i \\ \vec{0}^T & 1 \end{bmatrix}$.

With homogeneous coordinates, the following properties of affine transformations become apparent:

- Affine transformations are associative.
  For affine transformations F_1, F_2, and F_3,

$$(F_3 \circ F_2) \circ F_1 = F_3 \circ (F_2 \circ F_1).$$

- Affine transformations are *not* commutative.
  For affine transformations F_1 and F_2,

$$F_2 \circ F_1 \neq F_1 \circ F_2.$$

3.4 Uses and Abuses of Homogeneous Coordinates

Homogeneous coordinates provide a different representation for Cartesian coordinates, and cannot be treated in quite the same way. For example, consider the midpoint between two points $\bar{p}_1 = (1,1)$ and $\bar{p}_2 = (5,5)$. The midpoint is $(\bar{p}_1 + \bar{p}_2)/2 = (3,3)$. We can represent these points in homogeneous coordinates as $\hat{p}_1 = (1,1,1)$ and $\hat{p}_2 = (5,5,1)$. Directly applying the same computation as above gives the same resulting point: $(3,3,1)$. However, we can *also* represent these points as $\hat{p}'_1 = (2,2,2)$ and $\hat{p}'_2 = (5,5,1)$. We then have $(\hat{p}'_1 + \hat{p}'_2)/2 = (7/2, 7/2, 3/2)$, which cooresponds to the Cartesian point $(7/3, 7/3)$. This is a different point, and illustrates that we cannot blindly apply geometric operations to homogeneous coordinates. The simplest solution is to **always convert homogeneous coordinates to Cartesian coordinates.** That said, there are several important operations that can be performed correctly in terms of homogeneous coordinates, as follows.

Affine transformations. An important case in the previous section is applying an affine transformation to a point in homogeneous coordinates:

$$\bar{q} = F(\bar{p}) = A\bar{p} + \vec{t} \quad (5)$$
$$\hat{q} = \hat{A}\hat{p} = (x', y', 1)^T \quad (6)$$

It is easy to see that this operation is correct, since rescaling $\hat{p}$ does not change the result:

$$\hat{A}(\alpha\hat{p}) = \alpha(\hat{A}\hat{p}) = \alpha\hat{q} = (\alpha x', \alpha y', \alpha)^T \quad (7)$$

which is the same geometric point as $\hat{q} = (x', y', 1)^T$

Vectors. We can represent a vector $\vec{v} = (x, y)$ in homogeneous coordinates by setting the last element of the vector to be zero: $\hat{v} = (x, y, 0)$. However, when adding a vector to a point, the point must have the third component be 1.

$$\hat{q} = \hat{p} + \hat{v} \quad (8)$$
$$(x', y', 1)^T = (x_p, y_p, 1) + (x, y, 0) \quad (9)$$

The result is clearly incorrect if the third component of the vector is not 0.

Aside:
Homogeneous coordinates are a representation of points in **projective geometry.**

3.5 Hierarchical Transformations

It is often convenient to model objects as hierarchically connected parts. For example, a robot arm might be made up of an upper arm, forearm, palm, and fingers. Rotating at the shoulder on the upper arm would affect all of the rest of the arm, but rotating the forearm at the elbow would affect the palm and fingers, but not the upper arm. A reasonable hierarchy, then, would have the upper arm at the root, with the forearm as its only child, which in turn connects only to the palm, and the palm would be the parent to all of the fingers.

Each part in the hierarchy can be modeled in its own local coordinates, independent of the other parts. For a robot, a simple square might be used to model each of the upper arm, forearm, and so on. Rigid body transformations are then applied to each part relative to its parent to achieve the proper alignment and pose of the object. For example, the fingers are positioned to be in the appropriate places in the palm coordinates, the fingers and palm together are positioned in forearm coordinates, and the process continues up the hierarchy. Then a transformation applied to upper arm coordinates is also applied to all parts down the hierarchy.

3.6 Transformations in OpenGL

OpenGL manages two 4×4 transformation matrices: the *modelview matrix*, and the *projection matrix*. Whenever you specify geometry (using `glVertex`), the vertices are transformed by the current modelview matrix and then the current projection matrix. Hence, you don't have to perform these transformations yourself. You can modify the entries of these matrices at any time. OpenGL provides several utilities for modifying these matrices. The modelview matrix is normally used to represent geometric transformations of objects; the projection matrix is normally used to store the camera transformation. For now, we'll focus just on the modelview matrix, and discuss the camera transformation later.

To modify the current matrix, first specify which matrix is going to be manipulated: use `glMatrixMode(GL_MODE` to modify the modelview matrix. The modelview matrix can then be initialized to the identity with `glLoadIdentity()`. The matrix can be manipulated by directly filling its values, multiplying it by an arbitrary matrix, or using the functions OpenGL provides to multiply the matrix by specific transformation matrices (`glRotate`, `glTranslate`, and `glScale`). Note that these transformations **right-multiply** the current matrix; this can be confusing since it means that you specify transformations in the reverse of the obvious order. Exercise: why does OpenGL right-multiply the current matrix?

OpenGL provides a **stacks** to assist with hierarchical transformations. There is one stack for the modelview matrix and one for the projection matrix. OpenGL provides routines for pushing and popping matrices on the stack.

The following example draws an upper arm and forearm with shoulder and elbow joints. The current modelview matrix is pushed onto the stack and popped at the end of the rendering, so, for example, another arm could be rendered without the transformations from rendering this arm affecting its modelview matrix. Since each OpenGL transformation is applied by multiplying a matrix on the right-hand side of the modelview matrix, the transformations occur in reverse order. Here, the upper arm is translated so that its shoulder position is at the origin, then it is rotated, and finally it is translated so that the shoulder is in its appropriate world-space position. Similarly, the forearm is translated to rotate about its elbow position, then it is translated so that the elbow matches its position in upper arm coordinates.

```
glPushMatrix();

glTranslatef(worldShoulderX, worldShoulderY, 0.0f);
drawShoulderJoint();
glRotatef(shoulderRotation, 0.0f, 0.0f, 1.0f);
glTranslatef(-upperArmShoulderX, -upperArmShoulderY, 0.0f);
drawUpperArmShape();

glTranslatef(upperArmElbowX, upperArmElbowY, 0.0f);
```



```
drawElbowJoint();
glRotatef(elbowRotation, 0.0f, 0.0f, 1.0f);
glTranslatef(-forearmElbowX, -forearmElbowY, 0.0f);
drawForearmShape();

glPopMatrix();
```


4 Coordinate Free Geometry

Coordinate free geometry (CFG) is a style of expressing geometric objects and relations that avoids unnecessary reliance on any specific coordinate system. Representing geometric quantities in terms of coordinates can frequently lead to confusion, and to derivations that rely on irrelevant coordinate systems.

We first define the basic quantities:

1. A **scalar** is just a real number.
2. A **point** is a location in space. It *does not* have any intrinsic coordinates.
3. A **vector** is a direction and a magnitude. It *does not* have any intrinsic coordinates.

A point is not a vector: we cannot add two points together. We cannot compute the magnitude of a point, or the location of a vector.
Coordinate free geometry defines a restricted class of operations on points and vectors, even though both are represented as vectors in matrix algebra. The following operations are the **only** operations allowed in CFG.

1. $\|\vec{v}\|$: magnitude of a vector.
2. $\bar{p}_1 + \vec{v}_1 = \bar{p}_2$, or $\vec{v}_1 = \bar{p}_2 - \bar{p}_1$.: point-vector addition.
3. $\vec{v}_1 + \vec{v}_2 = \vec{v}_3$.: vector addition
4. $\alpha\vec{v}_1 = \vec{v}_2$: vector scaling. If $\alpha > 0$, then $\vec{v}_2$ is a new vector with the same direction as $\vec{v}_1$, but magnitude $\alpha\|\vec{v}_1\|$. If $\alpha < 0$, then the direction of the vector is reversed.
5. $\vec{v}_1 \cdot \vec{v}_2$: dot product $= \|\vec{v}_1\|\|\vec{v}_2\| \cos(\theta)$, where θ is the angle between the vectors.
6. $\vec{v}_1 \times \vec{v}_2$: cross product, where $\vec{v}_1$ and $\vec{v}_2$ are 3D vectors. Produces a new vector perpedicular to $\vec{v}_1$ and to $\vec{v}_2$, with magnitude $\|\vec{v}_1\|\|\vec{v}_2\| \sin(\theta)$. The orientation of the vector is determined by the right-hand rule (see textbook).
7. $\sum_i \alpha_i\vec{v}_i = \vec{v}$: Linear combination of vectors
8. $\sum_i \alpha_i\bar{p}_i = \bar{p}$, **if** $\sum_i \alpha_i = 1$: affine combination of points.
9. $\sum_i \alpha_i\bar{p}_i = \vec{v}$, **if** $\sum_i \alpha_i = 0$

Example:

- $\bar{p}_1 + (\bar{p}_2 - \bar{p}_3) = \bar{p}_1 + \vec{v} = \bar{p}_4$.
- $\alpha\bar{p}_2 - \alpha\bar{p}_1 = \alpha\vec{v}_1 = \vec{v}_2$.
- $\frac{1}{2}(p_1 + p_2) = p_1 + \frac{1}{2}(\bar{p}_2 - \bar{p}_1) = \bar{p}_1 + \frac{1}{2}\vec{v} = \bar{p}_3$.

Note:
In order to understand these formulas, try drawing some pictures to illustrate different cases (like the ones that were drawn in class).

Note that operations that are *not* in the list are undefined.
These operations have a number of basic properties, e.g., commutivity of dot product: $\vec{v}_1 \cdot \vec{v}_2 = \vec{v}_2 \cdot \vec{v}_1$, distributivity of dot product: $\vec{v}_1 \cdot (\vec{v}_2 + \vec{v}_3) = \vec{v}_1 \cdot \vec{v}_2 + \vec{v}_1 \cdot \vec{v}_3$.
CFG helps us reason about geometry in several ways:

1. When reasoning about geometric objects, we only care about the intrinsic geometric properties of the objects, not their coordinates. CFG prevents us from introducing irrelevant concepts into our reasoning.

2. CFG derivations usually provide much more geometric intuition for the steps and for the results. It is often easy to interpret the meaning of a CFG formula, whereas a coordinate-based formula is usually quite opaque.

3. CFG derivations are usually simpler than using coordinates, since introducing coordinates often creates many more variables.

4. CFG provides a sort of "type-checking" for geometric reasoning. For example, if you derive a formula that includes a term $\bar{p} \cdot \vec{v}$, that is, a "point dot vector," then there may be a bug in your reasoning. In this way, CFG is analogous to type-checking in compilers. Although you could do all programming in assembly language — which does not do type-checking and will happily led you add, say, a floating point value to a function pointer — most people would prefer to use a compiler which performs type-checking and can thus find many bugs.

In order to *implement* geometric algorithms we need to use coordinates. These coordinates are part of the representation of geometry — they are not fundamental to reasoning about geometry itself.

Example:
CFG says that we cannot add two points; there is no meaning to this operation. But what happens if we try to do so anyway, using coordinates?
Suppose we have two points: $\bar{p}_0 = (0, 0)$ and $\bar{p}_1 = (1, 1)$, and we add them together coordinate-wise: $\bar{p}_2 = \bar{p}_0 + \bar{p}_1 = (1, 1)$. This is not a valid CFG operation, but we have done it anyway just to tempt fate and see what happens. We see that the

resulting point is the same as one of the original points: $\bar{p}_2 = \bar{p}_1$.
Now, on the other hand, suppose the two points were represented in a different coordinate frame: $\bar{q}_0 = (1, 1)$ and $\bar{q}_1 = (2, 2)$. The points $\bar{q}_0$ and $\bar{q}_1$ are the *same* points as $\bar{p}_0$ and $\bar{p}_1$, with the same vector between them, but we have just represented them in a different coordinate frame, i.e., with a different origin. Adding together the points we get $\bar{q}_2 = \bar{q}_0 + \bar{q}_1 = (3, 3)$. This is a ***different*** point from $\bar{q}_0$ and $\bar{q}_1$, whereas before we got the same point.
The geometric relationship of the result of adding two points depends on the coordinate system. There is no clear geometric interpretation for adding two points.

Aside:
It is actually possible to define CFG with far fewer axioms than the ones listed above. For example, the linear combination of vectors is simply addition and scaling of vectors.

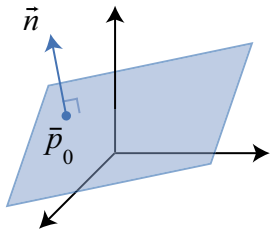
5 3D Objects

5.1 Surface Representations

As with 2D objects, we can represent 3D objects in **parametric** and **implicit** forms. (There are also explicit forms for 3D surfaces — sometimes called "height fields" — but we will not cover them here).

5.2 Planes

- **Implicit:** $(\bar{p} - \bar{p}_0) \cdot \vec{n} = 0$, where $\bar{p}_0$ is a point in $\mathbb{R}^3$ on the plane, and $\vec{n}$ is a normal vector perpendicular to the plane.



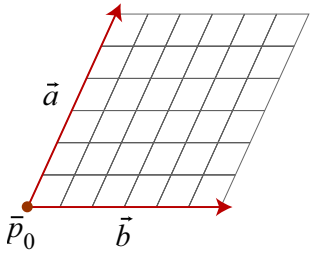
A plane can be defined uniquely by three non-colinear points $\bar{p}_1, \bar{p}_2, \bar{p}_3$. Let $\vec{a} = \bar{p}_2 - \bar{p}_1$ and $\vec{b} = \bar{p}_3 - \bar{p}_1$, so $\vec{a}$ and $\vec{b}$ are vectors in the plane. Then $\vec{n} = \vec{a} \times \vec{b}$. Since the points are not colinear, $\|\vec{n}\| \neq 0$.

- **Parametric:** $\bar{s}(\alpha, \beta) = \bar{p}_0 + \alpha\vec{a} + \beta\vec{b}$, for $\alpha, \beta \in \mathbb{R}$.

> *Note:*
> This is similar to the parametric form of a line: $\bar{l}(\alpha) = \bar{p}_0 + \alpha\vec{a}$.

A **planar patch** is a parallelogram defined by bounds on α and β.

> *Example:*
> Let $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$:



5.3 Surface Tangents and Normals

The **tangent** to a curve at $\bar{p}$ is the instantaneous direction of the curve at $\bar{p}$.

The **tangent plane** to a surface at $\bar{p}$ is analogous. It is defined as the plane containing tangent vectors to all curves on the surface that go through $\bar{p}$.

A **surface normal** at a point $\bar{p}$ is a vector perpendicular to a tangent plane.

5.3.1 Curves on Surfaces

The parametric form $\bar{p}(\alpha, \beta)$ of a surface defines a mapping from 2D points to 3D points: every 2D point (α, β) in $\mathbb{R}^2$ corresponds to a 3D point $\bar{p}$ in $\mathbb{R}^3$. Moreover, consider a curve $\bar{l}(\lambda) = (\alpha(\lambda), \beta(\lambda))$ in 2D — there is a corresponding curve in 3D contained within the surface: $\bar{l}^*(\lambda) = \bar{p}(\bar{l}(\lambda))$.

5.3.2 Parametric Form

For a curve $\bar{c}(\lambda) = (x(\lambda), y(\lambda), z(\lambda))^T$ in 3D, the tangent is

$$\frac{d\bar{c}(\lambda)}{d\lambda} = \left(\frac{dx(\lambda)}{d\lambda}, \frac{dy(\lambda)}{d\lambda}, \frac{dz(\lambda)}{d\lambda} \right). \tag{10}$$

For a surface point $\bar{s}(\alpha, \beta)$, two tangent vectors can be computed:

$$\frac{\partial \bar{s}}{\partial \alpha} \text{ and } \frac{\partial \bar{s}}{\partial \beta}. \tag{11}$$

Derivation:
Consider a point (α_0, β_0) in 2D which corresponds to a 3D point $\bar{s}(\alpha_0, \beta_0)$. Define two straight lines in 2D:

$$\bar{d}(\lambda_1) = (\lambda_1, \beta_0)^T \tag{12}$$
$$\bar{e}(\lambda_2) = (\alpha_0, \lambda_2)^T \tag{13}$$

These lines correspond to curves in 3D:

$$\bar{d}^*(\lambda_1) = \bar{s}(\bar{d}(\lambda_1)) \tag{14}$$
$$\bar{e}^*(\lambda_2) = \bar{s}(\bar{d}(\lambda_2)) \tag{15}$$

Using the chain rule for vector functions, the tangents of these curves are:

$$\frac{\partial \bar{d}^*}{\partial \lambda_1} = \frac{\partial \bar{s}}{\partial \alpha}\frac{\partial \bar{d}_\alpha}{\partial \lambda_1} + \frac{\partial \bar{s}}{\partial \beta}\frac{\partial \bar{d}_\beta}{\partial \lambda_1} \quad = \quad \frac{\partial \bar{s}}{\partial \alpha} \tag{16}$$

$$\frac{\partial \bar{e}^*}{\partial \lambda_2} = \frac{\partial \bar{s}}{\partial \alpha}\frac{\partial \bar{e}_\alpha}{\partial \lambda_2} + \frac{\partial \bar{s}}{\partial \beta}\frac{\partial \bar{e}_\beta}{\partial \lambda_2} \quad = \quad \frac{\partial \bar{s}}{\partial \beta} \tag{17}$$

The normal of $\bar{s}$ at $\alpha = \alpha_0$, $\beta = \beta_0$ is

$$\vec{n}(\alpha_0, \beta_0) \quad = \quad \left(\frac{\partial \bar{s}}{\partial \alpha}\bigg|_{\alpha_0,\beta_0} \right) \times \left(\frac{\partial \bar{s}}{\partial \beta}\bigg|_{\alpha_0,\beta_0} \right). \tag{18}$$

The tangent plane is a plane containing the surface at $\bar{s}(\alpha_0, \beta_0)$ with normal vector equal to the surface normal. The equation for the tangent plane is:

$$\vec{n}(\alpha_0, \beta_0) \cdot (\bar{p} - \bar{s}(\alpha_0, \beta_0)) \quad = \quad 0. \tag{19}$$

What if we used different curves in 2D to define the tangent plane? It can be shown that we get the same tangent plane; in other words, tangent vectors of all 2D curves through a given surface point are contained within a single tangent plane. (Try this as an exercise).

Note:
The normal vector is not unique. If $\vec{n}$ is a normal vector, then any vector $\alpha\vec{n}$ is also normal to the surface, for $\alpha \in \mathbb{R}$. What this means is that the normal can be scaled, and the direction can be reversed.

5.3.3 Implicit Form

In the implicit form, a surface is defined as the set of points $\bar{p}$ that satisfy $f(\bar{p}) = 0$ for some function f. A normal is given by the gradient of f,

$$\vec{n}(\bar{p}) = \nabla f(\bar{p})|_{\bar{p}} \tag{20}$$

where $\nabla f = \left(\frac{\partial f(\bar{p})}{\partial x}, \frac{\partial f(\bar{p})}{\partial y}, \frac{\partial f(\bar{p})}{\partial z} \right)$.

Derivation:
Consider a 3D curve $\bar{c}(\lambda)$ that is contained within the 3D surface, and that passes through $\bar{p}_0$ at λ_0. In other words, $\bar{c}(\lambda_0) = \bar{p}_0$ and

$$f(\bar{c}(\lambda)) \quad = \quad 0 \tag{21}$$

for all λ. Differentiating both sides gives:

$$\frac{\partial f}{\partial \lambda} = 0 \tag{22}$$

Expanding the left-hand side, we see:

$$\frac{\partial f}{\partial \lambda} = \frac{\partial f}{\partial x}\frac{\partial \bar{c}_x}{\partial \lambda} + \frac{\partial f}{\partial y}\frac{\partial \bar{c}_y}{\partial \lambda} + \frac{\partial f}{\partial z}\frac{\partial \bar{c}_z}{\partial \lambda} \tag{23}$$

$$= \nabla f(\bar{p})|_{\bar{p}} \cdot \frac{d\bar{c}}{d\lambda} = 0 \tag{24}$$

This last line states that the gradient is perpendicular to the curve tangent, which is the definition of the normal vector.

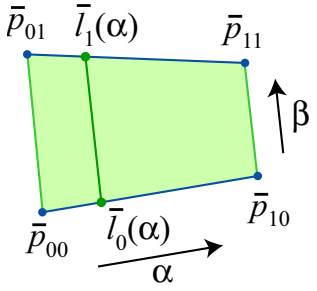
Example:
The implicit form of a sphere is: $f(\bar{p}) = \|\bar{p} - \bar{c}\|^2 - R^2 = 0$. The normal at a point $\bar{p}$ is: $\nabla f = 2(\bar{p} - \bar{c})$.

Exercise: show that the normal computed for a plane is the same, regardless of whether it is computed using the parametric or implicit forms. (This was done in class). Try it for another surface.

5.4 Parametric Surfaces

5.4.1 Bilinear Patch

A **bilinear patch** is defined by four points, no three of which are colinear.



Given $\bar{p}_{00}$, $\bar{p}_{01}$, $\bar{p}_{10}$, $\bar{p}_{11}$, define

$$\bar{l}_0(\alpha) = (1-\alpha)\bar{p}_{00} + \alpha\bar{p}_{10},$$
$$\bar{l}_1(\alpha) = (1-\alpha)\bar{p}_{01} + \alpha\bar{p}_{11}.$$

Then connect $\bar{l}_0(\alpha)$ and $\bar{l}_1(\alpha)$ with a line:

$$\bar{p}(\alpha, \beta) = (1 - \beta)\bar{l}_0(\alpha) + \beta\bar{l}_1(\alpha),$$

for $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$.

Question: when is a bilinear patch not equivalent to a planar patch? Hint: a planar patch is defined by 3 points, but a bilinear patch is defined by 4.

5.4.2 Cylinder

A **cylinder** is constructed by moving a point on a line l along a planar curve $p_0(\alpha)$ such that the direction of the line is held constant.

If the direction of the line l is $\vec{d}$, the cylinder is defined as

$$\bar{p}(\alpha, \beta) = p_0(\alpha) + \beta\vec{d}.$$

A **right cylinder** has $\vec{d}$ perpendicular to the plane containing $p_0(\alpha)$.

A **circular cylinder** is a cylinder where $p_0(\alpha)$ is a circle.

Example:
A right circular cylinder can be defined by $p_0(\alpha) = (r\cos(\alpha), r\sin(\alpha), 0)$, for $0 \leq \alpha < 2\pi$, and $\vec{d} = (0, 0, 1)$.

So $p_0(\alpha, \beta) = (r\cos(\alpha), r\sin(\alpha), \beta)$, for $0 \leq \beta \leq 1$.

To find the normal at a point on this cylinder, we can use the implicit form $f(x, y, z) = x^2 + y^2 - r^2 = 0$ to find $\nabla f = 2(x, y, 0)$.

Using the parametric form directly to find the normal, we have

$$\frac{\partial\bar{p}}{\partial\alpha} = r(-\sin(\alpha), \cos(\alpha), 0), \text{ and } \frac{\partial\bar{p}}{\partial\beta} = (0, 0, 1), \text{ so}$$

$$\frac{\partial\bar{p}}{\partial\alpha} \times \frac{\partial\bar{p}}{\partial\beta} = (r\cos(\alpha)r\sin(\alpha), 0).$$

Note:
The cross product of two vectors $\vec{a} = (a_1, a_2, a_3)$ and $\vec{b} = (b_1, b_2, b_3)$ can

be found by taking the determinant of the matrix,

$$\begin{bmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix}.$$

5.4.3 Surface of Revolution

To form a **surface of revolution**, we revolve a curve in the x-z plane, $\bar{c}(\beta) = (x(\beta), 0, z(\beta))$, about the z-axis.

Hence, each point on $\bar{c}$ traces out a circle parallel to the x-y plane with radius $|x(\beta)|$. Circles then have the form $(r\cos(\alpha), r\sin(\alpha))$, where α is the parameter of revolution. So the rotated surface has the parametric form

$$\bar{s}(\alpha, \beta) = (x(\beta)\cos(\alpha), x(\beta)\sin(\alpha), z(\beta)).$$

Example:
If $\bar{c}(\beta)$ is a line perpendicular to the x-axis, we have a right circular cylinder.

A torus is a surface of revolution:

$$\bar{c}(\beta) = (d + r\cos(\beta), 0, r\sin(\beta)).$$

5.4.4 Quadric

A **quadric** is a generalization of a conic section to 3D. The implicit form of a quadric in the standard position is

$$ax^2 + by^2 + cz^2 + d = 0,$$
$$ax^2 + by^2 + ez = 0,$$

for $a, b, c, d, e \in \mathbb{R}$. There are six basic types of quadric surfaces, which depend on the signs of the parameters.
They are the ellipsoid, hyperboloid of one sheet, hyperboloid of two sheets, elliptic cone, elliptic paraboloid, and hyperbolic paraboloid (saddle). All but the hyperbolic paraboloid may be expressed as a surface of revolution.

Example:
An ellipsoid has the implicit form

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1 = 0.$$

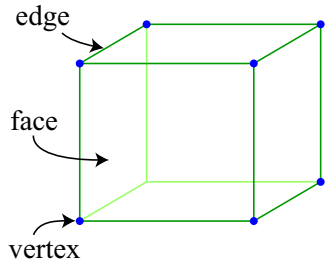
In parametric form, this is

$$\bar{s}(\alpha, \beta) = (a\sin(\beta)\cos(\alpha), b\sin(\beta)\sin(\alpha), c\cos(\beta)),$$

for $\beta \in [0, \pi]$ and $\alpha \in (-\pi, \pi]$.

5.4.5 Polygonal Mesh

A **polygonal mesh** is a collection of polygons (vertices, edges, and faces). As polygons may be used to approximate curves, a polygonal mesh may be used to approximate a surface.



A **polyhedron** is a closed, connected polygonal mesh. Each edge must be shared by two faces.

A **face** refers to a planar polygonal patch within a mesh.

A mesh is **simple** when its topology is equivalent to that of a sphere. That is, it has no holes.

Given a parametric surface, $\bar{s}(\alpha, \beta)$, we can sample values of α and β to generate a polygonal mesh approximating $\bar{s}$.

5.5 3D Affine Transformations

Three dimensional transformations are used for many different purposes, such as coordinate transforms, shape modeling, animation, and camera modeling.

An affine transform in 3D looks the same as in 2D: $F(\bar{p}) = A\bar{p} + \vec{t}$ for $A \in \mathbb{R}^{3\times3}$, $\bar{p}, \vec{t} \in \mathbb{R}^3$. A homogeneous affine transformation is

$$\hat{F}(\hat{p}) = \hat{M}\hat{p}, \text{ where } \hat{p} = \begin{bmatrix} \bar{p} \\ 1 \end{bmatrix}, \hat{M} = \begin{bmatrix} A & \vec{t} \\ \vec{0}^T & 1 \end{bmatrix}.$$

Translation: $A = I$, $\vec{t} = (t_x, t_y, t_z)$.

Scaling: $A = \text{diag}(s_x, s_y, s_z)$, $\vec{t} = \vec{0}$.

Rotation: $A = R$, $\vec{t} = \vec{0}$, and $\det(R) = 1$.

3D rotations are much more complex than 2D rotations, so we will consider only elementary rotations about the x, y, and z axes.

For a rotation about the z-axis, the z coordinate remains unchanged, and the rotation occurs in the x-y plane. So if $\bar{q} = R\bar{p}$, then $q_z = p_z$. That is,

$$\begin{bmatrix} q_x \\ q_y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix}.$$

Including the z coordinate, this becomes

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Similarly, rotation about the x-axis is

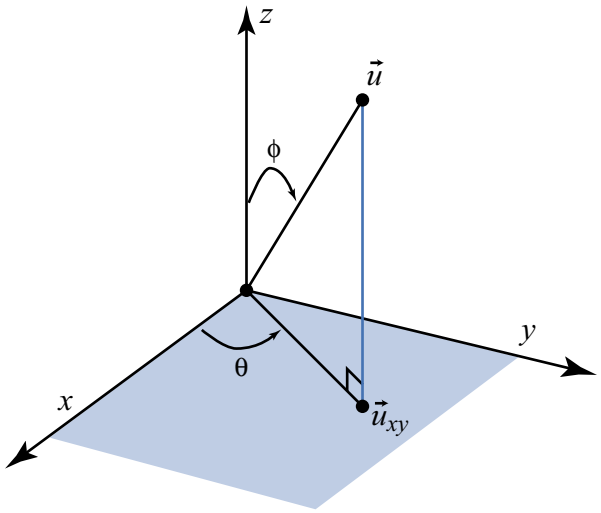
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

For rotation about the y-axis,

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}.$$

5.6 Spherical Coordinates

Any three dimensional vector $\vec{u} = (u_x, u_y, u_z)$ may be represented in **spherical coordinates**. By computing a polar angle ϕ counterclockwise about the y-axis from the z-axis and an azimuthal angle θ counterclockwise about the z-axis from the x-axis, we can define a vector in the appropriate direction. Then it is only a matter of scaling this vector to the correct length $(u_x^2 + u_y^2 + u_z^2)^{-1/2}$ to match $\vec{u}$.



Given angles ϕ and θ, we can find a unit vector as $\vec{u} = (\cos(\theta)\sin(\phi), \sin(\theta)\sin(\phi), cos(\phi))$.

Given a vector $\vec{u}$, its azimuthal angle is given by $\theta = \arctan\left(\frac{u_y}{u_x}\right)$ and its polar angle is $\phi = \arctan\left(\frac{(u_x^2+u_y^2)^{1/2}}{u_z}\right)$. This formula does not require that $\vec{u}$ be a unit vector.

5.6.1 Rotation of a Point About a Line

Spherical coordinates are useful in finding the rotation of a point about an arbitrary line. Let $\bar{l}(\lambda) = \lambda\vec{u}$ with $\|\vec{u}\| = 1$, and $\vec{u}$ having azimuthal angle θ and polar angle ϕ. We may compose elementary rotations to get the effect of rotating a point $\bar{p}$ about $\bar{l}(\lambda)$ by a counterclockwise angle ρ:

1. Align $\vec{u}$ with the z-axis.
   - Rotate by $-\theta$ about the z-axis so $\vec{u}$ goes to the xz-plane.
   - Rotate up to the z-axis by rotating by $-\phi$ about the y-axis.

   Hence, $\bar{q} = R_y(-\phi)R_z(-\theta)\bar{p}$
2. Apply a rotation by ρ about the z-axis: $R_z(\rho)$.

3. Invert the first step to move the z-axis back to $\vec{u}$: $R_z(\theta)R_y(\phi) = (R_y(-\phi)R_z(-\theta))^{-1}$.

Finally, our formula is $\bar{q} = R_{\vec{u}}(\rho)\bar{p} = R_z(\theta)R_y(\phi)R_z(\rho)R_y(-\phi)R_z(-\theta)\bar{p}$.

5.7 Nonlinear Transformations

Affine transformations are a first-order model of shape deformation. With affine transformations, scaling and shear are the simplest nonrigid deformations. Common higher-order deformations include tapering, twisting, and bending.

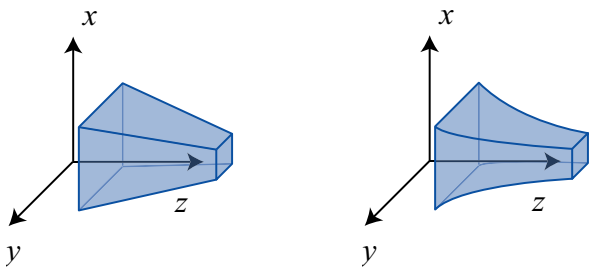
Example:
To create a nonlinear taper, instead of constantly scaling in x and y for all z, as in

$$\bar{q} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix} \bar{p},$$

let a and b be functions of z, so

$$\bar{q} = \begin{bmatrix} a(\bar{p}_z) & 0 & 0 \\ 0 & b(\bar{p}_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \bar{p}.$$

A linear taper looks like $a(z) = \alpha_0 + \alpha_1 z$.
A quadratic taper would be $a(z) = \alpha_0 + \alpha_1 z + \alpha_2 z^2$.



(c) Linear taper (d) Nonlinear taper

5.8 Representing Triangle Meshes

A triangle mesh is often represented with a list of vertices and a list of triangle faces. Each vertex consists of three floating point values for the x, y, and z positions, and a face consists of three

indices of vertices in the vertex list. Representing a mesh this way reduces memory use, since each vertex needs to be stored once, rather than once for every face it is on; and this gives us connectivity information, since it is possible to determine which faces share a common vertex. This can easily be extended to represent polygons with an arbitrary number of vertices, but any polygon can be decomposed into triangles. A tetrahedron can be represented with the following lists:

Vertex index	x	y	z
0	0	0	0
1	1	0	0
2	0	1	0
3	0	0	1

Face index	Vertices
0	0, 1, 2
1	0, 3, 1
2	1, 3, 2
3	2, 3, 0

Notice that vertices are specified in a counter-clockwise order, so that the front of the face and back can be distinguished. This is the default behavior for OpenGL, although it can also be set to take face vertices in clockwise order. Lists of normals and texture coordinates can also be specified, with each face then associated with a list of vertices and corresponding normals and texture coordinates.

5.9 Generating Triangle Meshes

As stated earlier, a parametric surface can be sampled to generate a polygonal mesh. Consider the surface of revolution

$$\bar{S}(\alpha, \beta) = [x(\alpha)\cos\beta, x(\alpha)\sin\beta, z(\alpha)]^T$$

with the profile $\bar{C}(\alpha) = [x(\alpha), 0, z(\alpha)]^T$ and $\beta \in [0, 2\pi]$.

To take a uniform sampling, we can use

$$\Delta\alpha = \frac{\alpha_1 - \alpha_0}{m}, \text{ and } \Delta\beta = \frac{2\pi}{n},$$

where m is the number of patches to take along the z-axis, and n is the number of patches to take around the z-axis.

Each patch would consist of four vertices as follows:

$$S_{ij} = \begin{pmatrix} \bar{S}(i\Delta\alpha, j\Delta\beta) \\ \bar{S}((i+1)\Delta\alpha, j\Delta\beta) \\ \bar{S}((i+1)\Delta\alpha, (j+1)\Delta\beta) \\ \bar{S}(i\Delta\alpha, (j+1)\Delta\beta) \end{pmatrix} = \begin{pmatrix} \bar{S}_{i,j} \\ \bar{S}_{i+1,j} \\ \bar{S}_{i+1,j+1} \\ \bar{S}_{i,j+1} \end{pmatrix}, \quad \text{for } \begin{matrix} i \in [0, m-1], \\ j \in [0, n-1] \end{matrix}$$

To render this as a triangle mesh, we must *tesselate* the sampled quads into triangles. This is accomplished by defining triangles P_{ij} and Q_{ij} given S_{ij} as follows:

$$P_{ij} = (\bar{S}_{i,j}, \bar{S}_{i+1,j}, \bar{S}_{i+1,j+1}), \text{ and } Q_{ij} = (\bar{S}_{i,j}, \bar{S}_{i+1,j+1}, \bar{S}_{i,j+1})$$

6 Camera Models

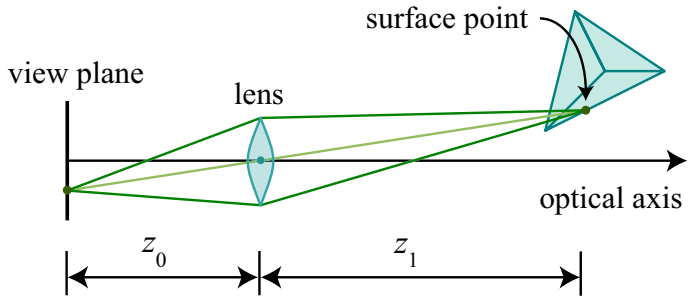
Goal: To model basic geometry of projection of 3D points, curves, and surfaces onto a 2D surface, the **view plane** or **image plane**.

6.1 Thin Lens Model

Most modern cameras use a lens to focus light onto the view plane (i.e., the sensory surface). This is done so that one can capture enough light in a sufficiently short period of time that the objects do not move appreciably, and the image is bright enough to show significant detail over a wide range of intensities and contrasts.

> *Aside:*
> In a conventional camera, the view plane contains either photoreactive chemicals; in a digital camera, the view plane contains a charge-coupled device (CCD) array. (Some cameras use a CMOS-based sensor instead of a CCD). In the human eye, the view plane is a curved surface called the *retina*, and and contains a dense array of cells with photoreactive molecules.

Lens models can be quite complex, especially for compound lens found in most cameras. Here we consider perhaps the simplist case, known widely as the thin lens model. In the thin lens model, rays of light emitted from a point travel along paths through the lens, convering at a point behind the lens. The key quantity governing this behaviour is called the *focal length* of the lens. The focal length,, $|f|$, can be defined as distance behind the lens to which rays from an infinitely distant source converge in focus.

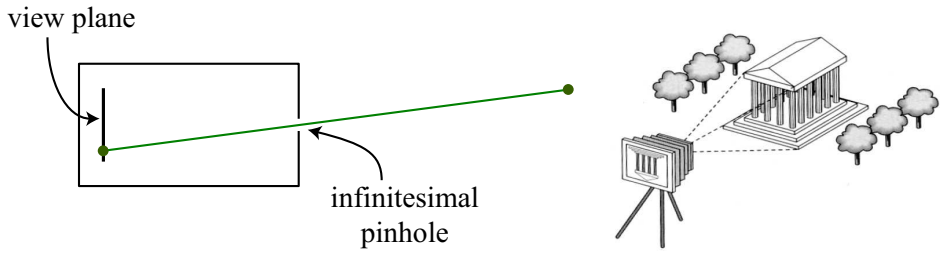


More generally, for the thin lens model, if z_1 is the distance from the center of the lens (i.e., the nodal point) to a surface point on an object, then for a focal length $|f|$, the rays from that surface point will be in focus at a distance z_0 behind the lens center, where z_1 and z_0 satisfy the thin lens equation:

$$\frac{1}{|f|} = \frac{1}{z_0} + \frac{1}{z_1} \tag{25}$$

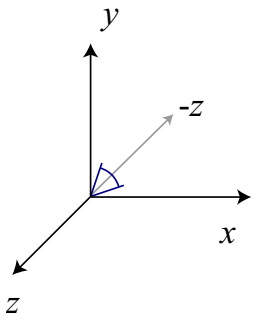
6.2 Pinhole Camera Model

A **pinhole** camera is an idealization of the thin lens as aperture shrinks to zero.

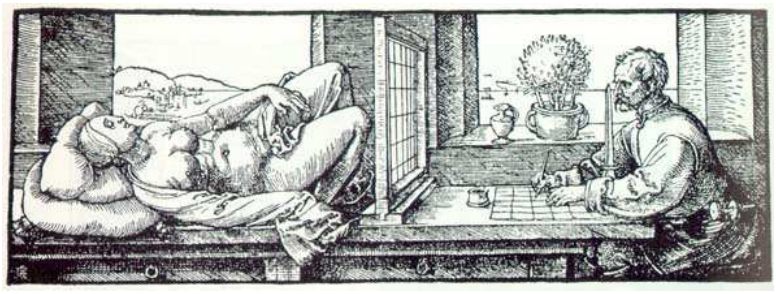


Light from a point travels along a single straight path through a pinhole onto the view plane. The object is imaged upside-down on the image plane.

Note:
We use a right-handed coordinate system for the camera, with the x-axis as the horizontal direction and the y-axis as the vertical direction. This means that the optical axis (gaze direction) is the negative z-axis.

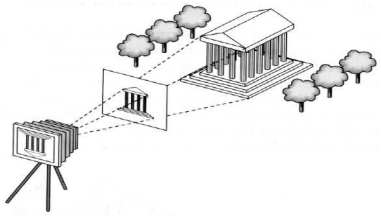


Here is another way of thinking about the pinhole model. Suppose you view a scene with one eye looking through a square window, and draw a picture of what you see through the window:



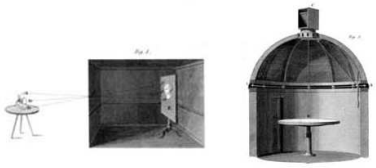
(Engraving by Albrecht Dürer, 1525).

The image you'd get corresponds to drawing a ray from the eye position and intersecting it with the window. This is equivalent to the pinhole camera model, except that the view plane is in front of the eye instead of behind it, and the image appears rightside-up, rather than upside down. (The eye point here replaces the pinhole). To see this, consider tracing rays from scene points through a view plane behind the eye point and one in front of it:



For the remainder of these notes, we will consider this camera model, as it is somewhat easier to think about, and also consistent with the model used by OpenGL.

Aside:
The earliest cameras were room-sized pinhole cameras, called *camera obscura*s. You would walk in the room and see an upside-down projection of the outside world on the far wall. The word *camera* is Latin for "room;" *camera obscura* means "dark room."



18th-century camera obscuras. The camera on the right uses a mirror in the roof to project images of the world onto the table, and viewers may rotate the mirror.

6.3 Camera Projections

Consider a point $\bar{p}$ in 3D space oriented with the camera at the origin, which we want to project onto the view plane. To project p_y to y, we can use similar triangles to get $y = \frac{f}{p_z} p_y$. This is **perspective projection**.

Note that $f < 0$, and the focal length is $|f|$.

In perspective projection, distant objects appear smaller than near objects:

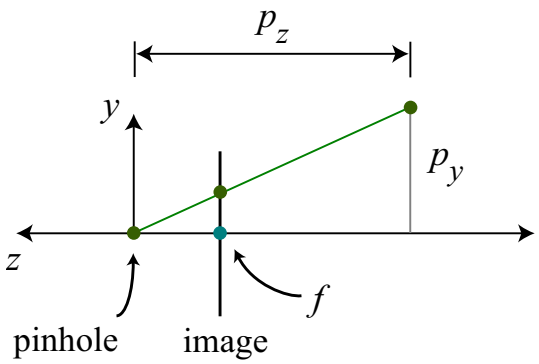
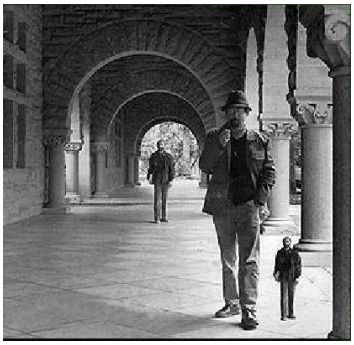


Figure 1: *

Perspective projection



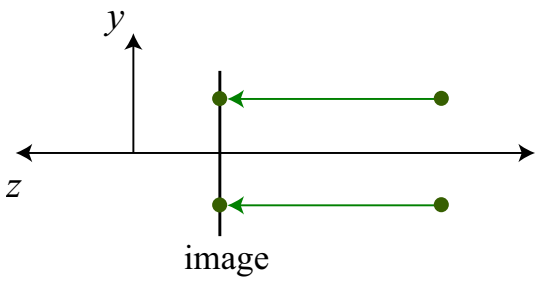
The man without the hat appears to be two different sizes, even though the two images of him have identical sizes when measured in pixels. In 3D, the man without the hat on the left is about 18 feet behind the man with the hat. This shows how much you might expect size to change due to perspective projection.

6.4 Orthographic Projection

For objects sufficiently far away, rays are nearly parallel, and variation in p_z is insignificant.

Here, the baseball players appear to be about the same height in pixels, even though the batter is about 60 feet away from the pitcher. Although this is an example of perspective projection, the camera is so far from the players (relative to the camera focal length) that they appear to be roughly the same size.

In the limit, $y = \alpha p_y$ for some real scalar α. This is **orthographic projection**:



6.5 Camera Position and Orientation

Assume camera coordinates have their origin at the "eye" (pinhole) of the camera, $\bar{e}$.

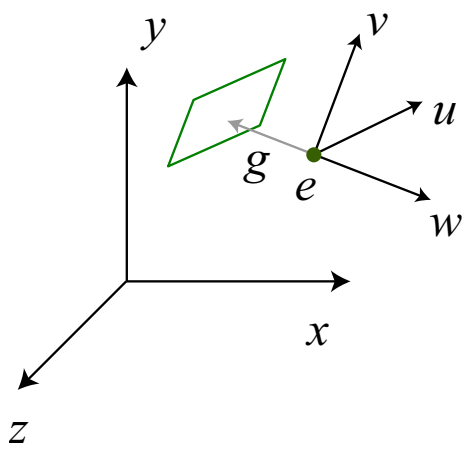


Figure 2:

Let $\vec{g}$ be the gaze direction, so a vector perpendicular to the view plane (parallel to the camera z-axis) is

$$\vec{w} = \frac{-\vec{g}}{\|\vec{g}\|} \tag{26}$$

We need two more orthogonal vectors $\vec{u}$ and $\vec{v}$ to specify a camera coordinate frame, with $\vec{u}$ and $\vec{v}$ parallel to the view plane. It may be unclear how to choose them directly. However, we can instead specify an "up" direction. Of course this up direction will not be perpendicular to the gaze direction.

Let $\vec{t}$ be the "up" direction (e.g., toward the sky so $\vec{t} = (0, 1, 0)$). Then we want $\vec{v}$ to be the closest vector in the viewplane to $\vec{t}$. This is really just the projection of $\vec{t}$ onto the view plane. And of course, $\vec{u}$ must be perpendicular to $\vec{v}$ and $\vec{w}$. In fact, with these definitions it is easy to show that $\vec{u}$ must also be perpendicular to $\vec{t}$, so one way to compute $\vec{u}$ and $\vec{v}$ from $\vec{t}$ and $\vec{g}$ is as follows:

$$\vec{u} = \frac{\vec{t} \times \vec{w}}{\|\vec{t} \times \vec{w}\|} \qquad\qquad \vec{v} = \vec{w} \times \vec{u} \tag{27}$$

Of course, we could have use many different "up" directions, so long as $\vec{t} \times \vec{w} \neq 0$.

Using these three basis vectors, we can define a **camera coordinate system**, in which 3D points are represented with respect to the camera's position and orientation. The camera coordinate system has its origin at the eye point $\bar{e}$ and has basis vectors $\vec{u}$, $\vec{v}$, and $\vec{w}$, corresponding to the x, y, and z axes in the camera's local coordinate system. This explains why we chose $\vec{w}$ to point away from the image plane: the right-handed coordinate system requires that z (and, hence, $\vec{w}$) point away from the image plane.
Now that we know how to represent the camera coordinate frame within the world coordinate frame we need to explicitly formulate the rigid transformation from world to camera coordinates. With this transformation and its inverse we can easily express points either in world coordinates or camera coordinates (both of which are necessary).
To get an understanding of the transformation, it might be helpful to remember the mapping from points in camera coordinates to points in world coordinates. For example, we have the following correspondences between world coordinates and camera coordinates: Using such correspondences

Camera coordinates (x_c, y_c, z_c)	World coordinates (x, y, z)
$(0, 0, 0)$	$\bar{e}$
$(0, 0, f)$	$\bar{e} + f\vec{w}$
$(0, 1, 0)$	$\bar{e} + \vec{v}$
$(0, 1, f)$	$\bar{e} + \vec{v} + f\vec{w}$

it is not hard to show that for a general point expressed in camera coordinates as $\bar{p}^c = (x_c, y_c, z_c)$, the corresponding point in world coordinates is given by

$$\begin{aligned} \bar{p}^w &= \bar{e} + x_c\vec{u} + y_c\vec{v} + z_c\vec{w} && (28)\\ &= \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix}\bar{p}^c + \bar{e} && (29)\\ &= M_{cw}\bar{p}^c + \bar{e}. && (30) \end{aligned}$$

where

$$M_{cw} = \left[\begin{array}{ccc} \vec{u} & \vec{v} & \vec{w} \end{array} \right] = \left[\begin{array}{ccc} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{array} \right] \tag{31}$$

Note: We can define the same transformation for points in homogeneous coordinates:

$$\hat{M}_{cw} = \left[\begin{array}{cc} M_{cw} & \bar{e} \\ \vec{0}^T & 1 \end{array} \right].$$

Now, we also need to find the inverse transformation, i.e., from world to camera coordinates. Toward this end, note that the matrix M_{cw} is orthonormal. To see this, note that vectors $\vec{u}$, $\vec{v}$ and, $\vec{w}$ are all of unit length, and they are perpendicular to one another. You can also verify this by computing $M_{cw}^T M_{cw}$. Because M_{cw} is orthonormal, we can express the inverse transformation (from camera coordinates to world coordinates) as

$$\begin{aligned} \bar{p}^c &= M_{cw}^T(\bar{p}^w - \bar{e}) \\ &= M_{wc}\bar{p}^w - \bar{d}\,, \end{aligned}$$

where $M_{wc} = M_{cw}^T = \left[\begin{array}{c} \vec{u}^T \\ \vec{v}^T \\ \vec{w}^T \end{array} \right]$. (why?), and $\bar{d} = M_{cw}^T \bar{e}$.

In homogeneous coordinates, $\hat{p}^c = \hat{M}_{wc}\hat{p}^w$, where

$$\begin{aligned} \hat{M}_v &= \left[\begin{array}{cc} M_{wc} & -M_{wc}\bar{e} \\ \vec{0}^T & 1 \end{array} \right] \\ &= \left[\begin{array}{cc} M_{wc} & \vec{0} \\ \vec{0}^T & 1 \end{array} \right] \left[\begin{array}{cc} I & -\bar{e} \\ \vec{0}^T & 1 \end{array} \right]. \end{aligned}$$

This transformation takes a point from world to camera-centered coordinates.

6.6 Perspective Projection

Above we found the form of the perspective projection using the idea of similar triangles. Here we consider a complementary algebraic formulation. To begin, we are given

- a point $\bar{p}^c$ in camera coordinates (uvw space),
- center of projection (eye or pinhole) at the origin in camera coordinates,
- image plane perpendicular to the z-axis, through the point $(0, 0, f)$, with $f < 0$, and

- line of sight is in the direction of the negative z-axis (in camera coordinates),

we can find the intersection of the ray from the pinhole to $\bar{p}^c$ with the view plane.

The ray from the pinhole to $\bar{p}^c$ is $\bar{r}(\lambda) = \lambda(\bar{p}^c - \bar{0})$.

The image plane has normal $(0, 0, 1) = \vec{n}$ and contains the point $(0, 0, f) = \bar{f}$. So a point $\bar{x}^c$ is on the plane when $(\bar{x}^c - \bar{f}) \cdot \vec{n} = 0$. If $\bar{x}^c = (x^c, y^c, z^c)$, then the plane satisfies $z^c - f = 0$.

To find the intersection of the plane $z^c = f$ and ray $\vec{r}(\lambda) = \lambda\bar{p}^c$, substitute $\vec{r}$ into the plane equation. With $\bar{p}^c = (p_x^c, p_y^c, p_z^c)$, we have $\lambda p_z^c = f$, so $\lambda^* = f/p_z^c$, and the intersection is

$$\vec{r}(\lambda^*) = \left(f\frac{p_x^c}{p_z^c}, f\frac{p_y^c}{p_z^c}, f \right) = f\left(\frac{p_x^c}{p_z^c}, \frac{p_y^c}{p_z^c}, 1 \right) \equiv \bar{x}^*. \tag{32}$$

The first two coordinates of this intersection $\bar{x}^*$ determine the image coordinates.

2D points in the image plane can therefore be written as

$$\begin{bmatrix} x^* \\ y^* \end{bmatrix} = \frac{f}{p_z^c} \begin{bmatrix} p_x^c \\ p_y^c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \frac{f}{p_z^c} \bar{p}^c.$$

The mapping from $\bar{p}^c$ to $(x^*, y^*, 1)$ is called **perspective projection**.

Note:
Two important properties of perspective projection are:

- Perspective projection preserves linearity. In other words, the projection of a 3D line is a line in 2D. This means that we can render a 3D line segment by projecting the endpoints to 2D, and then draw a line between these points in 2D.
- Perspective projection does not preserve parallelism: two parallel lines in 3D do not necessarily project to parallel lines in 2D. When the projected lines intersect, the intersection is called a **vanishing point**, since it corresponds to a point infinitely far away. Exercise: when do parallel lines project to parallel lines and when do they not?

Aside:
The discovery of linear perspective, including vanishing points, formed a cornerstone of Western painting beginning at the Renaissance. On the other hand, defying realistic perspective was a key feature of Modernist painting.

To see that linearity is preserved, consider that rays from points on a line in 3D through a pinhole all lie on a plane, and the intersection of a plane and the image plane is a line. That means to draw polygons, we need only to project the vertices to the image plane and draw lines between them.

6.7 Homogeneous Perspective

The mapping of $\bar{p}^c = (p_x^c, p_y^c, p_z^c)$ to $\bar{x}^* = \frac{f}{p_z^c}(p_x^c, p_y^c, p_z^c)$ is just a form of scaling transformation. However, the magnitude of the scaling depends on the depth p_z^c. So it's not linear.

Fortunately, the transformation can be expressed linearly (ie as a matrix) in homogeneous coordinates. To see this, remember that $\hat{p} = (\bar{p}, 1) = \alpha(\bar{p}, 1)$ in homogeneous coordinates. Using this property of homogeneous coordinates we can write $\bar{x}^*$ as

$$\hat{x}^* = \left(p_x^c, p_y^c, p_z^c, \frac{p_z^c}{f}\right).$$

As usual with homogeneous coordinates, when you scale the homogeneous vector by the inverse of the last element, when you get in the first three elements is precisely the perspective projection. Accordingly, we can express $\hat{x}^*$ as a linear transformation of $\hat{p}^c$:

$$\hat{x}^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \hat{p}^c \equiv \hat{M}_p \hat{p}^c.$$

Try multiplying this out to convince yourself that this all works.
Finally, $\hat{M}_p$ is called the homogeneous perspective matrix, and since $\hat{p}^c = \hat{M}_{wc}\hat{p}^w$, we have $\hat{x}^* = \hat{M}_p \hat{M}_{wc} \hat{p}^w$.

6.8 Pseudodepth

After dividing by its last element, $\hat{x}^*$ has its first two elements as image plane coordinates, and its third element is f. We would like to be able to alter the homogeneous perspective matrix $\hat{M}_p$ so that the third element of $\frac{p_z^c}{f}\hat{x}^*$ encodes depth while keeping the transformation linear.

Idea: Let $\hat{x}^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1/f & 0 \end{bmatrix} \hat{p}^c$, so $z^* = \frac{f}{p_z^c}(ap_z^c + b)$.

What should a and b be? We would like to have the following two constraints:

$$z^* = \begin{cases} -1 & \text{when } p_z^c = f \\ 1 & \text{when } p_z^c = F \end{cases},$$

where f gives us the position of the **near plane**, and F gives us the z coordinate of the **far plane**.

So $-1 = af + b$ and $1 = af + b\frac{f}{F}$. Then $2 = b\frac{f}{F} - b = b\left(\frac{f}{F} - 1\right)$, and we can find

$$b = \frac{2F}{f - F}.$$

Substituting this value for b back in, we get $-1 = af + \frac{2F}{f-F}$, and we can solve for a:

$$\begin{aligned} a &= -\frac{1}{f}\left(\frac{2F}{f-F} + 1\right) \\ &= -\frac{1}{f}\left(\frac{2F}{f-F} + \frac{f-F}{f-F}\right) \\ &= -\frac{1}{f}\left(\frac{f+F}{f-F}\right). \end{aligned}$$

These values of a and b give us a function $z^*(p_z^c)$ that increases monotonically as p_z^c decreases (since p_z^c is negative for objects in front of the camera). Hence, z^* can be used to sort points by depth.

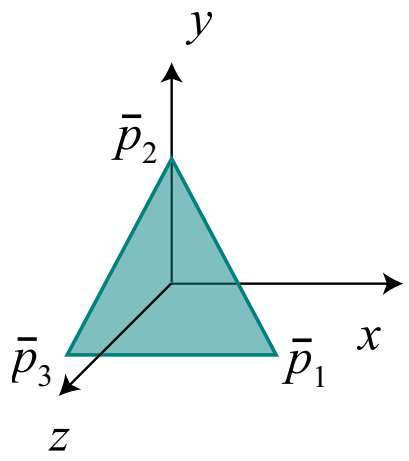
Why did we choose these values for a and b? Mathematically, the specific choices do not matter, but they are convenient for implementation. These are also the values that OpenGL uses.

What is the meaning of the near and far planes? Again, for convenience of implementation, we will say that only objects between the near and far planes are visible. Objects in front of the near plane are behind the camera, and objects behind the far plane are too far away to be visible. Of course, this is only a loose approximation to the real geometry of the world, but it is very convenient for implementation. The range of values between the near and far plane has a number of subtle implications for rendering in practice. For example, if you set the near and far plane to be very far apart in OpenGL, then Z-buffering (discussed later in the course) will be very inaccurate due to numerical precision problems. On the other hand, moving them too close will make distant objects disappear. However, these issues will generally not affect rendering simple scenes. (For homework assignments, we will usually provide some code that avoids these problems).

6.9 Projecting a Triangle

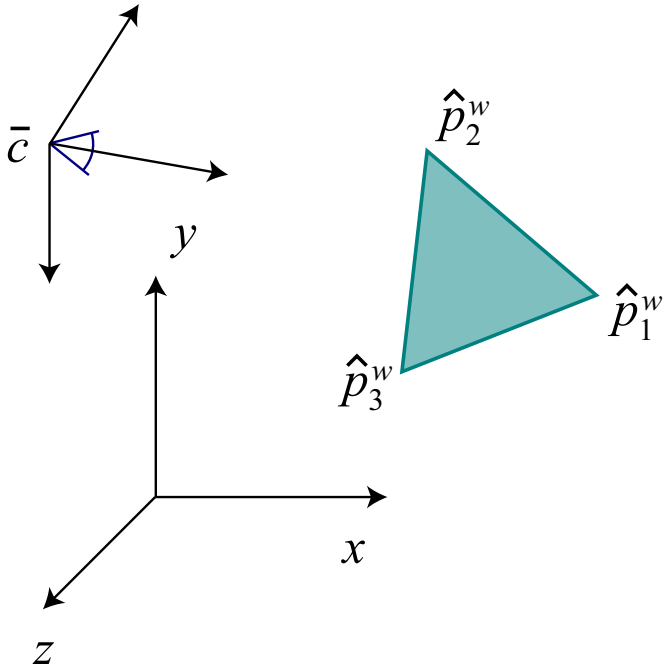
Let's review the steps necessary to project a triangle from object space to the image plane.

1. A triangle is given as three vertices in an object-based coordinate frame: $\bar{p}_1^o$, $\bar{p}_2^o$, $\bar{p}_3^o$.



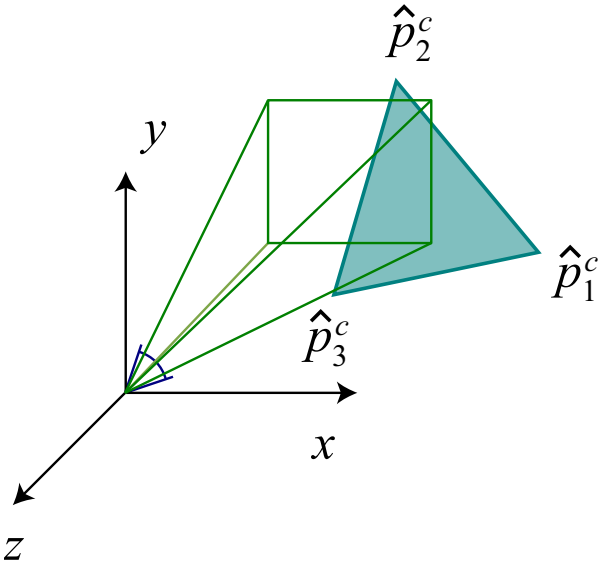
A triangle in object coordinates.

2. Transform to world coordinates based on the object's transformation: $\hat{p}_1^w$, $\hat{p}_2^w$, $\hat{p}_3^w$, where $\hat{p}_i^w = \hat{M}_{ow}\hat{p}_i^o$.



The triangle projected to world coordinates, with a camera at $\bar{c}$.

3. Transform from world to camera coordinates: $\hat{p}_i^c = \hat{M}_{wc}\hat{p}_i^w$.



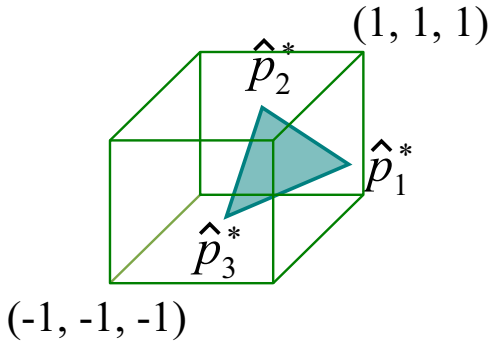
The triangle projected from world to camera coordinates.

4. Homogeneous perspective transformation: $\hat{x}_i^* = \hat{M}_p \hat{p}_i^c$, where

$$\hat{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1/f & 0 \end{bmatrix}, \text{ so } \hat{x}_i^* = \begin{bmatrix} p_x^c \\ p_y^c \\ ap_z^c + b \\ \frac{p_z^c}{f} \end{bmatrix}.$$

5. Divide by the last component:

$$\begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} = f \begin{bmatrix} \frac{p_x^c}{p_z^c} \\ \frac{p_y^c}{p_z^c} \\ \frac{ap_z^c + b}{p_z^c} \end{bmatrix}.$$



The triangle in normalized device coordinates after perspective division.

Now (x^*, y^*) is an image plane coordinate, and z^* is pseudodepth for each vertex of the triangle.

6.10 Camera Projections in OpenGL

OpenGL's modelview matrix is used to transform a point from object or world space to camera space. In addition to this, a *projection matrix* is provided to perform the homogeneous perspective transformation from camera coordinates to *clip coordinates* before performing perspective division. After selecting the projection matrix, the `glFrustum` function is used to specify a viewing volume, assuming the camera is at the origin:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(left, right, bottom, top, near, far);
```

For orthographic projection, `glOrtho` can be used instead:

```
glOrtho(left, right, bottom, top, near, far);
```

The GLU library provides a function to simplify specifying a perspective projection viewing frustum:

```
gluPerspective(fieldOfView, aspectRatio, near, far);
```

The field of view is specified in degrees about the x-axis, so it gives the vertical visible angle. The aspect ratio should usually be the viewport width over its height, to determine the horizontal field of view.

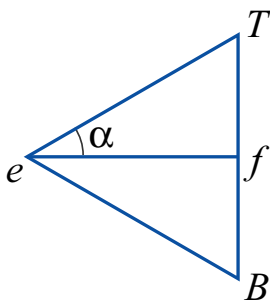
7 Visibility

We have seen so far how to determine how 3D points project to the camera's image plane. Additionally, we can render a triangle by projecting each vertex to 2D, and then filling in the pixels of the 2D triangle. However, what happens if two triangles project to the same pixels, or, more generally, if they overlap? Determining which polygon to render at each pixel is **visibility.** An object is visible if there exists a direct line-of-sight to that point, unobstructed by any other objects. Moreover, some objects may be invisible because they are behind the camera, outside of the field-of-view, or too far away.

7.1 The View Volume and Clipping

The **view volume** is made up of the space between the near plane, f, and far plane, F. It is bounded by B, T, L, and R on the bottom, top, left, and right, respectively.

The angular field of view is determined by f, B, T, L, and R:



From this figure, we can find that $\tan(\alpha) = \frac{1}{2}\frac{T-B}{|f|}$.

Clipping is the process of removing points and parts of objects that are outside the view volume.

We would like to modify our homogeneous perspective transformation matrix to simplify clipping. We have

$$\hat{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{1}{f}\left(\frac{f+F}{f-F}\right) & \frac{2F}{f-F} \\ 0 & 0 & -1/f & 0 \end{bmatrix}.$$

Since this is a homogeneous transformation, it may be multiplied by a constant without changing

its effect. Multiplying $\hat{M}_p$ by f gives us

$$\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\left(\frac{f+F}{f-F}\right) & \frac{2fF}{f-F} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

If we alter the transform in the x and y coordinates to be

$$\hat{x}^* = \begin{bmatrix} \frac{2f}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2f}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -\left(\frac{f+F}{f-F}\right) & \frac{2fF}{f-F} \\ 0 & 0 & 1 & 0 \end{bmatrix} \hat{p}^c,$$

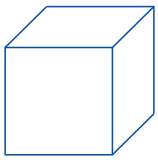
then, after projection, the view volume becomes a cube with sides at -1 and $+1$. This is called the **canonical view volume** and has the advantage of being easy to clip against.

Note:
The OpenGL command glFrustum(l, r, b, t, n, f) takes the distance to the near and far planes rather than the position on the z-axis of the planes. Hence, the n used by glFrustum is our $-f$ and the f used by glFrustum is $-F$. Substituting these values into our matrix gives exactly the perspective transformation matrix used by OpenGL.

7.2 Backface Removal

Consider a closed polyhedral object. Because it is closed, far side of the object will always be invisible, blocked by the near side. This observation can be used to accelerate rendering, by removing **back-faces.**

Example:
For this simple view of a cube, we have three backfacing polygons, the left side, back, and bottom:



Only the near faces are visible.

We can determine if a face is back-facing as follows. Suppose we compute a normals $\vec{n}$ for a mesh face, with the normal chosen so that it points outside the object For a surface point $\bar{p}$ on a planar

patch and eye point $\bar{e}$, if $(\bar{p} - \bar{e}) \cdot \vec{n} > 0$, then the angle between the view direction and normal is less than $90°$, so the surface normal points away from $\bar{e}$. The result will be the same no matter which face point $\bar{p}$ we use.

Hence, if $(\bar{p} - \bar{e}) \cdot \vec{n} > 0$, the patch is backfacing and should be removed. Otherwise, it *might* be visible. This should be calculated in world coordinates so the patch can be removed as early as possible.

Note:
To compute $\vec{n}$, we need three vertices on the patch, in counterclockwise order, as seen from the outside of the object, $\bar{p}_1$, $\bar{p}_1$, and $\bar{p}_3$. Then the unit normal is

$$\frac{(\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)}{\|(\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)\|}.$$

Backface removal is a "quick reject" used to accelerate rendering. It must still be used together with another visibility method. The other methods are more expensive, and removing backfaces just reduces the number of faces that must be considered by a more expensive method.

7.3 The Depth Buffer

Normally when rendering, we compute an image buffer `I(i,j)` that stores the color of the object that projects to pixel (i, j). The depth d of a pixel is the distance from the eye point to the object. The **depth buffer** is an array `zbuf(i, j)` which stores, for each pixel (i, j), the depth of the nearest point drawn so far. It is initialized by setting all depth buffer values to infinite depth: `zbuf(i,j)`$= \infty$.

To draw color c at pixel (i, j) with depth d:

```
if d < zbuf(i, j) then
  putpixel(i, j, c)
  zbuf(i, j) = d
end
```

When drawing a pixel, if the new pixel's depth is greater than the current value of the depth buffer at that pixel, then there must be some object blocking the new pixel, and it is not drawn.

Advantages

- Simple and accurate
- Independent of order of polygons drawn

Disadvantages

- Memory required for depth buffer
- Wasted computation on drawing distant points that are drawn over with closer points that occupy the same pixel

To represent the depth at each pixel, we can use pseudodepth, which is available after the homogeneous perspective transformation.[1] Then the depth buffer should be initialized to 1, since the pseudodepth values are between -1 and 1. Pseudodepth gives a number of numerical advantages over true depth.

To scan convert a triangular polygon with vertices $\bar{x}_1$, $\bar{x}_2$, and $\bar{x}_3$, pseudodepth values d_1, d_2, and d_3, and fill color c, we calculate the x values and pseudodepths for each edge at each scanline. Then for each scanline, interpolate pseudodepth between edges and compare the value at each pixel to the value stored in the depth buffer.

7.4 Painter's Algorithm

The **painter's algorithm** is an alternative to depth buffering to attempt to ensure that the closest points to a viewer occlude points behind them. The idea is to draw the most distant patches of a surface first, allowing nearer surfaces to be drawn over them.

In the heedless painter's algorithm, we first sort faces according to depth of the vertex furthest from the viewer. Then faces are rendered from furthest to nearest.

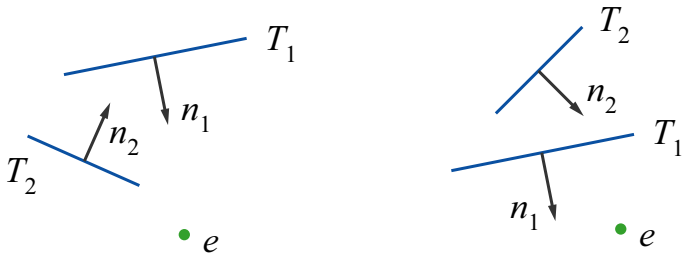
There are problems with this approach, however. In some cases, a face that occludes part of another face can still have its furthest vertex further from the viewer than any vertex of the face it occludes. In this situation, the faces will be rendered out of order. Also, polygons cannot intersect at all as they can when depth buffering is used instead. One solution is to split triangles, but doing this correctly is very complex and slow. Painter's algorithm is rarely used directly in practice; however, a data-structure called BSP trees can be used to make painter's algorithm much more appealing.

7.5 BSP Trees

The idea of **binary space partitioning trees** (BSP trees) is to extend the painter's algorithm to make back-to-front ordering of polygons fast for any eye location and to divide polygons to avoid overlaps.

Imagine two patches, T_1 and T_2, with outward-facing normals $\vec{n}_1$ and $\vec{n}_2$.

[1] The OpenGL documentation is confusing in a few places — "depth" is used to mean pseudodepth, in commands like `glReadPixels` and `gluUnProject`.



If the eye point, $\bar{e}$, and T_2 are on the same side of T_1, then we draw T_1 before T_2. Otherwise, T_2 should be drawn before T_1.

We know if two points are on the same side of a plane containing T_1 by using the implicit equation for T_1,

$$f_1(\bar{x}) \quad = \quad (\bar{x} - \bar{p}_1) \cdot \vec{n}. \tag{33}$$

If $\bar{x}$ is on the plane, $f_1(\bar{x}) = 0$. Otherwise, if $f_1(\bar{x}) > 0$, $\bar{x}$ is on the "outside" of T_1, and if $f_1(\bar{x}) < 0$, $\bar{x}$ is "inside."

Before any rendering can occur, the scene geometry must be processed to build a BSP tree to represent the relative positions of all the facets with respect to their inside/outside half-planes. The same BSP tree can be used for any eye position, so the tree only has to be constructed once if everything other than the eye is static. For a single scene, there are many different BSP trees that can be used to represent it — it's best to try to construct balanced trees.

The tree traversal algorithm to draw a tree with root F is as follows:

```
if eye is in the outside half-space of F
  draw faces on the inside subtree of F
  draw F
  draw faces on the outside subtree of F
else
  draw faces on the outside subtree of F
  draw F (if backfaces are drawn)
  draw faces on the inside subtree of F
end
```

7.6 Visibility in OpenGL

OpenGL directly supports depth buffering, but it is often used in addition to other visibility techniques in interactive applications. For example, many games use a BSP tree to prune the amount of static map geometry that is processed that would otherwise not be visible anyway. Also, when

dealing with blended, translucent materials, these objects often must be drawn from back to front without writing to the depth buffer to get the correct appearance. For simple scenes, however, the depth buffer alone is sufficient.

To use depth buffering in OpenGL with GLUT, the OpenGL context must be initialized with memory allocated for a depth buffer, with a command such as

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
```

Next, depth writing and testing must be enabled in OpenGL:

```
glEnable(GL_DEPTH_TEST);
```

OpenGL will automatically write pseudodepth values to the depth buffer when a primitive is rendered as long as the depth test is enabled. The `glDepthMask` function can be used to disable depth writes, so depth testing will occur without writing to the depth buffer when rendering a primitive.

When clearing the display to render a new frame, the depth buffer should also be cleared:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```


8 Basic Lighting and Reflection

Up to this point, we have considered only the geometry of how objects are transformed and projected to images. We now discuss the *shading* of objects: how the appearance of objects depends, among other things, on the lighting that illuminates the scene, and on the interaction of light with the objects in the scene. Some of the basic qualitative properties of lighting and object reflectance that we need to be able to model include:

Light source - There are different types of sources of light, such as point sources (e.g., a small light at a distance), extended sources (e.g., the sky on a cloudy day), and secondary reflections (e.g., light that bounces from one surface to another).

Reflectance - Different objects reflect light in different ways. For example, diffuse surfaces appear the same when viewed from different directions, whereas a mirror looks very different from different points of view.

In this chapter, we will develop simplified model of lighting that is easy to implement and fast to compute, and used in many real-time systems such as OpenGL. This model will be an approximation and does not fully capture all of the effects we observe in the real world. In later chapters, we will discuss more sophisticated and realistic models.

8.1 Simple Reflection Models

8.1.1 Diffuse Reflection

We begin with the diffuse reflectance model. A diffuse surface is one that appears similarly bright from all viewing directions. That is, the emitted light appears independent of the viewing location. Let $\bar{p}$ be a point on a diffuse surface with normal $\vec{n}$, light by a point light source in direction $\vec{s}$ from the surface. The reflected intensity of light is given by:

$$L_d(\bar{p}) \quad = \quad r_d \, I \, \max(0, \vec{s} \cdot \vec{n}) \tag{34}$$

where I is the intensity of the light source, r_d is the diffuse reflectance (or albedo) of the surface, and $\vec{s}$ is the direction of the light source. This equation requires the vectors to be normalized, i.e., $||\vec{s}|| = 1$, $||\vec{n} = 1||$.

The $\vec{s} \cdot \vec{n}$ term is called the *foreshortening term*. When a light source projects light obliquely at a surface, that light is spread over a large area, and less of the light hits any specific point. For example, imagine pointing a flashlight directly at a wall versus in a direction nearly parallel: in the latter case, the light from the flashlight will spread over a greater area, and individual points on the wall will not be as bright.

For color rendering, we would specify the reflectance in color (as $(r_{d,R}, r_{d,G}, r_{d,B})$), and specify the light source in color as well (I_R, I_G, I_B). The reflected color of the surface is then:

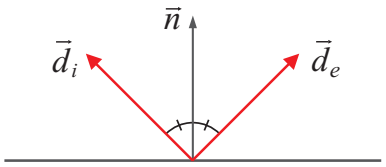
$$L_{d,R}(\bar{p}) = r_{d,R}\, I_R\, \max(0, \vec{s} \cdot \vec{n}) \quad (35)$$

$$L_{d,G}(\bar{p}) = r_{d,G}\, I_G\, \max(0, \vec{s} \cdot \vec{n}) \quad (36)$$

$$L_{d,B}(\bar{p}) = r_{d,B}\, I_B\, \max(0, \vec{s} \cdot \vec{n}) \quad (37)$$

8.1.2 Perfect Specular Reflection

For pure specular (mirror) surfaces, the incident light from each incident direction $\vec{d_i}$ is reflected toward a unique emittant direction $\vec{d_e}$. The emittant direction lies in the same plane as the incident direction $\vec{d_i}$ and the surface normal $\vec{n}$, and the angle between $\vec{n}$ and $\vec{d_e}$ is equal to that between $\vec{n}$ and $\vec{d_i}$. One can show that the emittant direction is given by $\vec{d_e} = 2(\vec{n} \cdot \vec{d_i})\vec{n} - \vec{d_i}$. (The derivation was



covered in class). In perfect specular reflection, the light emitted in direction $\vec{d_e}$ can be computed by reflecting $\vec{d_e}$ across the normal (as $2(\vec{n} \cdot \vec{d_e})\vec{n} - \vec{d_e}$), and determining the incoming light in this direction. (Again, all vectors are required to be normalized in these equations).

8.1.3 General Specular Reflection

Many materials exhibit a significant specular component in their reflectance. But few are perfect mirrors. First, most specular surfaces do not reflect all light, and that is easily handled by introducing a scalar constant to attenuate intensity. Second, most specular surfaces exhibit some form of *off-axis specular reflection*. That is, many polished and shiny surfaces (like plastics and metals) emit light in the perfect mirror direction and in some nearby directions as well. These off-axis specularities look a little blurred. Good examples are *highlights* on plastics and metals.

More precisely, the light from a distant point source in the direction of $\vec{s}$ is reflected into a range of directions about the perfect mirror directions $\vec{m} = 2(\vec{n} \cdot \vec{s})\vec{n} - \vec{s}$. One common model for this is the following:

$$L_s(\vec{d_e}) = r_s I \max(0, \vec{m} \cdot \vec{d_e})^{\alpha}, \quad (38)$$

where r_s is called the specular reflection coefficient I is the incident power from the point source, and $\alpha \geq 0$ is a constant that determines the width of the specular highlights. As α increases, the effective width of the specular reflection decreases. In the limit as α increases, this becomes a mirror.

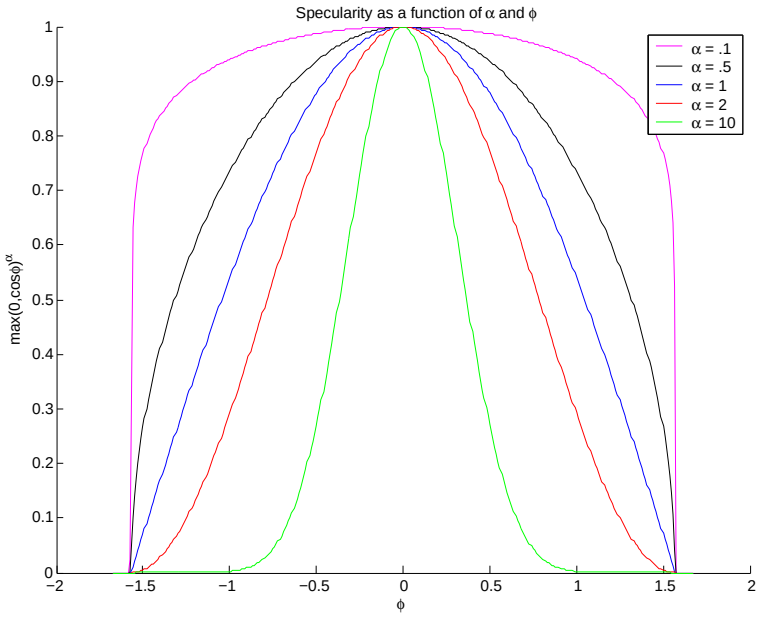


Figure 3: Plot of specular intensity as a function of viewing angle ϕ.

The intensity of the specular region is proportional to $\max(0, \cos\phi)^\alpha$, where ϕ is the angle between $\vec{m}$ and $\vec{d}_e$. One way to understand the nature of specular reflection is to plot this function, see Figure 3.

8.1.4 Ambient Illumination

The diffuse and specular shading models are easy to compute, but often appear artificial. The biggest issue is the point light source assumption, the most obvious consequence of which is that any surface normal pointing away from the light source (i.e., for which $\vec{s} \cdot \vec{n} < 0$) will have a radiance of zero. A better approximation to the light source is a uniform *ambient* term plus a point light source. This is a still a remarkably crude model, but it's much better than the point source by itself. Ambient illumintation is modeled simply by:

$$L_a(\bar{p}) = r_a \, I_a \tag{39}$$

where r_a is often called the ambient reflection coefficient, and I_a denotes the integral of the uniform illuminant.

8.1.5 Phong Reflectance Model

The **Phong reflectance model** is perhaps the simplest widely used shading model in computer graphics. It comprises a diffuse term (Eqn (81)), an ambient term (Eqn (82)), and a specular term

(Eqn (85)):

$$L(\bar{p}, \vec{d_e}) \;=\; r_d\, I_d\, \max(0, \vec{s} \cdot \vec{n}) \;+\; r_a\, I_a \;+\; r_s I_s \max(0, \vec{m} \cdot \vec{d_e})^{\alpha}, \tag{40}$$

where

- I_a, I_d, and I_r are parameters that correspond to the power of the light sources for the ambient, diffuse, and specular terms;
- r_a, r_d and r_s are scalar constants, called reflection coefficients, that determine the relative magnitudes of the three reflection terms;
- α determines the spread of the specurlar highlights;
- $\vec{n}$ is the surface normal at $\bar{p}$;
- $\vec{s}$ is the direction of the distant point source;
- $\vec{m}$ is the perfect mirror direction, given $\vec{n}$ and $\vec{s}$; and
- and $\vec{d_e}$ is the emittant direction of interest (usually the direction of the camera).

In effect, this is a model in which the diffuse and specular components of reflection are due to incident light from a point source. Extended light sources and the bouncing of light from one surface to another are not modeled except through the ambient term. Also, arguably this model has more parameters than the physics might suggest; for example, the model does not constrain the parameters to conserve energy. Nevertheless it is sometimes useful to give computer graphics practitioners more freedom in order to acheive the appearance they're after.

8.2 Lighting in OpenGL

OpenGL provides a slightly modified version of Phong lighting. Lighting and any specific lights to use must be enabled to see its effects:

```
glEnable(GL_LIGHTING);  // enable Phong lighting
glEnable(GL_LIGHT0);    // enable the first light source
glEnable(GL_LIGHT1);    // enable the second light source
...
```

Lights can be directional (infinitely far away) or positional. Positional lights can be either point lights or spotlights. Directional lights have the w component set to 0, and positional lights have w set to 1. Light properties are specified with the `glLight` functions:


```
GLfloat direction[] = {1.0f, 1.0f, 1.0f, 0.0f};
GLfloat position[] = {5.0f, 3.0f, 8.0f, 1.0f};
Glfloat spotDirection[] = {0.0f, 3.0f, 3.0f};
Glfloat diffuseRGBA[] = {1.0f, 1.0f, 1.0f, 1.0f};
Glfloat specularRGBA[] = {1.0f, 1.0f, 1.0f, 1.0f};

// A directional light
glLightfv(GL_LIGHT0, GL_POSITION, direction);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseRGBA);
glLightfv(GL_LIGHT0, GL_SPECULAR, specularRGBA);

// A spotlight
glLightfv(GL_LIGHT1, GL_POSITION, position);
glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuseRGBA);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spotDirection);
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0f);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 30.0f);
```

OpenGL requires you to specify both diffuse and specular components for the light source. This has no physical interpretation (real lights do not have “diffuse” or “specular” properties), but may be useful for some effects. The `glMaterial` functions are used to specify material properties, for example:

```
GLfloat diffuseRGBA = {1.0f, 0.0f, 0.0f, 1.0f};
GLfloat specularRGBA = {1.0f, 1.0f, 1.0f, 1.0f};
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseRGBA);
glMaterialfv(GL_FRONT, GL_SPECULAR, specularRGBA);
glMaterialf(GL_FRONT, GL_SHININESS, 3.0f);
```

Note that both lights and materials have ambient terms. Additionally, there is a global ambient term:

```
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glMaterialfv(GL_FRONT, GL_AMBIENT, ambientMaterial);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientGlobal);
```

The material has an emission term as well, that is meant to model objects that can give off their own light. However, no light is actually cast on other objects in the scene.

```
glMaterialfv(GL_FRONT, GL_EMISSION, em);
```

The global ambient term is multiplied by the current material ambient value and added to the material’s emission value. The contribution from each light is then added to this value.

When rendering an object, normals should be provided for each face or for each vertex so that lighting can be computed:


```
glNormal3f(nx, ny, nz);
glVertex3f(x, y, z);
```


9 Shading

Goal: To use the lighting and reflectance model to shade facets of a polygonal mesh — that is, to assign intensities to pixels to give the impression of opaque surfaces rather than wireframes.

Assume we're given the following:

- $\bar{e}^w$ - center of projection in world coordinates
- $\bar{l}^w$ - point light source location
- I_a, I_d - intensities of ambient and directional light sources
- r_a, r_d, r_s - coefficients for ambient, diffuse, and specular reflections
- α - exponent to control width of highlights

9.1 Flat Shading

With **flat shading**, each triangle of a mesh is filled with a single color.

For a triangle with counterclockwise vertices $\bar{p}_1$, $\bar{p}_2$, and $\bar{p}_3$, as seen from the outside, let the midpoint be $\bar{p} = \frac{1}{3}(\bar{p}_1 + \bar{p}_2 + \bar{p}_3)$ with normal $\vec{n} = \frac{(\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)}{\|(\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)\|}$. Then we may find the intensity at $\bar{p}$ using the Phong model and fill the polygon with that:

$$E = \tilde{I}_a r_a + r_d \tilde{I}_d \max(0, \vec{n} \cdot \vec{s}) + r_s \tilde{I}_d \max(0, \vec{r} \cdot \vec{c})^\alpha, \tag{41}$$

where $\vec{s} = \frac{\bar{l}^w - \bar{p}}{\|\bar{l}^w - \bar{p}\|}$, $\vec{c} = \frac{\bar{e}^w - \bar{p}}{\|\bar{e}^w - \bar{p}\|}$, and $\vec{r} = -\vec{s} + 2(\vec{s} \cdot \vec{n})\vec{n}$.

Flat shading is a simple approach to filling polygons with color, but can be inaccurate for smooth surfaces, and shiny surfaces. For smooth surfaces—which are often tesselated and represented as polyhedra, using flat shading can lead to a very strong faceting effect. In other words, the surface looks very much like a polyhedron, rather than the smooth surface it's supposed to be. This is because our visual system is very sensitive to variations in shading, and so using flat shading makes faces really look flat.

9.2 Interpolative Shading

The idea of **interpolative shading** is to avoid computing the full lighting equation at each pixel by interpolating quantites at the vertices of the faces.

Given vertices $\bar{p}_1$, $\bar{p}_2$, and $\bar{p}_3$, we need to compute the normals for each vertex, compute the radiances for each vertex, project onto the window in device coordinates, and fill the polygon using scan conversion.

There are two methods used for interpolative shading:

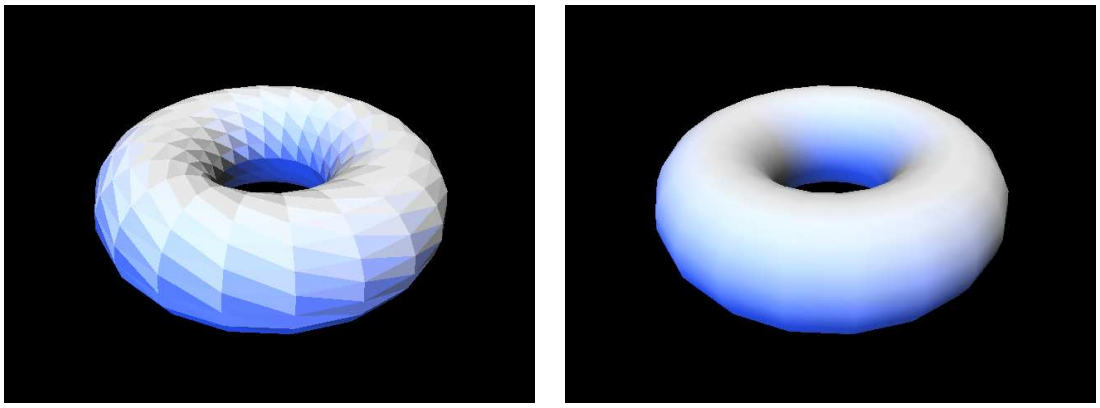
Gouraud Shading The radiance values are computed at the vertices and then linearly interpolated within each triangle. This is the form of shading implemented in OpenGL.

Phong shading The normal values at each vertex are linearly interpolated within each triangle, and the radiance is computed at each pixel.

Gouraud shading is more efficient, but Phong shading is more accurate. When will Gouraud shading give worse results?

9.3 Shading in OpenGL

OpenGL only directly supports Gouraud shading or flat shading. Gouraud is enabled by default, computing vertex colors, and interpolating colors across triangle faces. Flat shading can be enabled with `glShadeModel(GL_FLAT)`. This renders an entire face with the color of a single vertex, giving a faceted appearance.



Left: Flat shading of a triangle mesh in OpenGL. *Right:* Gouraud shading. Note that the mesh appears smooth, although the coarseness of the geometry is visible at the silhouettes of the mesh.

With *pixel shaders* on programmable graphics hardware, it is possible to achieve Phong shading by using a small program to compute the illumination at each pixel with interpolated normals. It is even possible to use a *normal map* to assign arbitrary normals within faces, with a pixel shader using these normals to compute the illumination.

10 Texture Mapping

10.1 Overview

We would like to give objects a more varied and realistic appearance through complex variations in reflectance that convey textures. There are two main sources of natural texture:

- Surface markings — variations in *albedo* (i.e. the total light reflected from ambient and diffuse components of reflection), and
- Surface relief — variations in 3D shape which introduces local variability in shading.

We will focus only on surface markings.

Examples of surface markings and surface relief

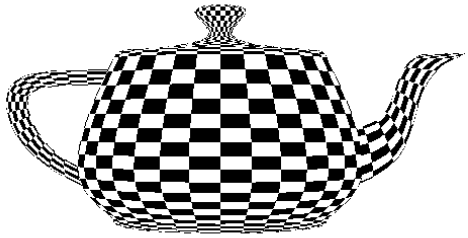
These main issues will be covered:

- Where textures come from,
- How to map textures onto surfaces,
- How texture changes reflectance and shading,
- Scan conversion under perspective warping, and
- Aliasing

10.2 Texture Sources

10.2.1 Texture Procedures

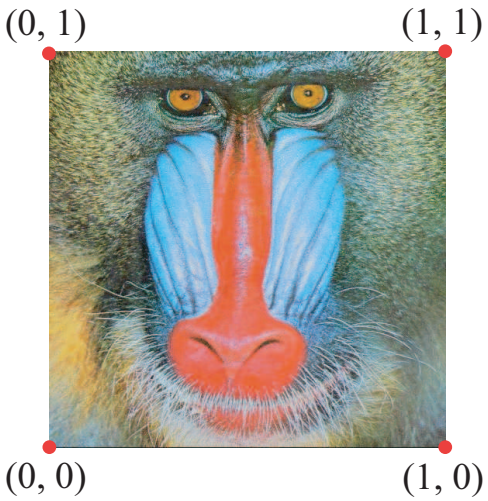
Textures may be defined procedurally. As input, a procedure requires a point on the surface of an object, and it outputs the surface albedo at that point. Examples of procedural textures include checkerboards, fractals, and noise.



A procedural checkerboard pattern applied to a teapot. The checkerboard texture comes from the OpenGL programming guide chapter on texture mapping.

10.2.2 Digital Images

To map an arbitrary digital image to a surface, we can define texture coordinates $(u, v) \in [0, 1]^2$. For each point $[u_0, v_0]$ in texture space, we get a point in the corresponding image.



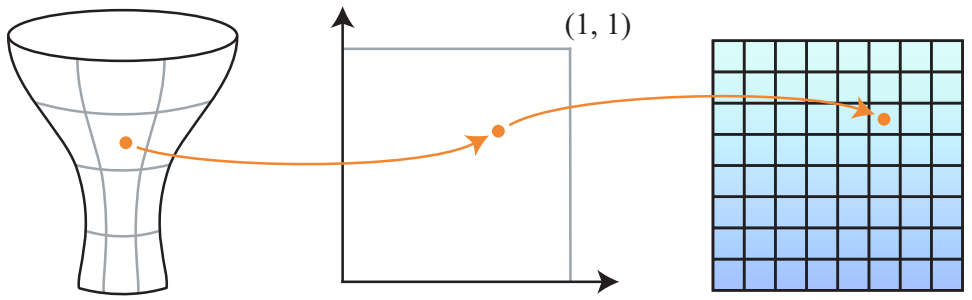
Texture coordinates of a digital image

10.3 Mapping from Surfaces into Texture Space

For each face of a mesh, specify a point (μ_i, ν_i) for vertex $\bar{p}_i$. Then define a continuous mapping from the parametric form of the surface $\bar{s}(\alpha, \beta)$ onto the texture, i.e. define m such that $(\mu, \nu) = m(\alpha, \beta)$.

Example:
For a planar patch $\bar{s}(\alpha, \beta) = \bar{p}_0 + \alpha\vec{a} + \beta\vec{b}$, where $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$.
Then we could use $\mu = \alpha$ and $\nu = \beta$.

Example:
For a surface of revolution, $\bar{s}(\alpha, \beta) = (c_x(\alpha)\cos(\beta), c_x(\alpha)\sin(\beta), c_z(\alpha))$. So let $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 2\pi$.
Then $\mu = \alpha$ and $\nu = \beta/2\pi$.



10.4 Textures and Phong Reflectance

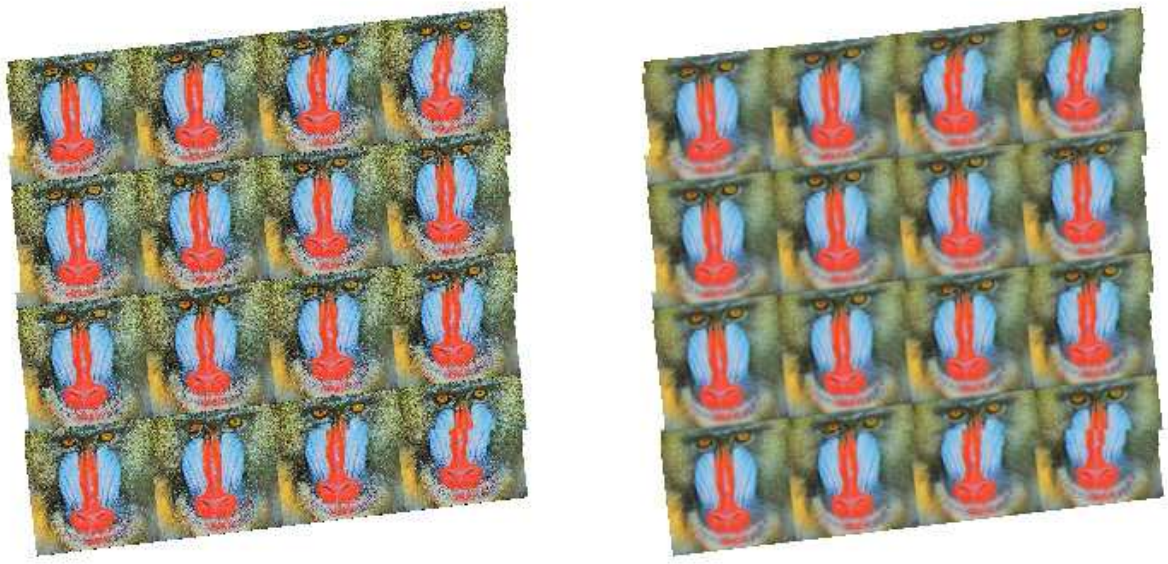
Scale texture values in the source image to be in the range $0 \leq \tau \leq 1$ and use them to scale the reflection coefficients r_d and r_a. That is,

$$\begin{aligned} \tilde{r}_d &= \tau r_d, \\ \tilde{r}_a &= \tau r_a. \end{aligned}$$

We could also multiply τ by the specular reflection, in which case we are simply scaling E from the Phong model.

10.5 Aliasing

A problem with high resolution texturing is aliasing, which occurs when adjacent pixels in a rendered image are sampled from pixels that are far apart in a texture image. By down-sampling—reducing the size of a texture—aliasing can be reduced for far away or small objects, but then textured objects look blurry when close to the viewer. What we really want is a high resolution texture for nearby viewing, and down-sampled textures for distant viewing. A technique called *mipmapping* gives us this by prerendering a texture image at several different scales. For example, a 256x256 image might be down-sampled to 128x128, 64x64, 32x32, 16x16, and so on. Then it is up to the renderer to select the correct mipmap to reduce aliasing artifacts at the scale of the rendered texture.



An aliased high resolution texture image (left) and the same texture after mipmapping (right)

10.6 Texturing in OpenGL

To use texturing in OpenGL, a texturing mode must be enabled. For displaying a 2D texture on polygons, this is accomplished with

```
glEnable(GL_TEXTURE_2D);
```

The dimensions of texture in OpenGL must be powers of 2, and texture coordinates are normalized, so that $(0, 0)$ is the lower left corner, and $(1, 1)$ is always the upper right corner. OpenGL 2.0, however, does allow textures of arbitrary size, in which case texture coordinates are based on the original pixel positions of the texture.

Since multiple textures can be present at any time, the texture to render with must be selected. Use `glGenTextures` to create texture handles and `glBindTexture` to select the texture with a given handle. A texture can then be loaded from main memory with `glTexImage2D` For example:

```
GLuint handles[2];
glGenTextures(2, handles);

glBindTexture(GL_TEXTURE_2D, handles[0]);
// Initialize texture parameters and load a texture with glTexImage2D

glBindTexture(GL_TEXTURE_2D, handles[1]);
// Initialize texture parameters and load another texture
```


There are a number of texture parameters that can be set to affect the behavior of a texture, using `glTexParameteri`. For example, texture wrap repeating can be enabled to allow a texture to be tiled at the borders, or the minifying and magnifying functions can be set to control the quality of textures as they get very close or far away from the camera. The texture environment can be set with `glTexEnvi`, which controls how a texture affects the rendering of the primitives it is attached to. An example of setting parameters and loading an image follows:

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth, imageHeight,
             0, GL_RGB, GL_UNSIGNED_BYTE, imagePointer);
```

Mipmaps can be generated automatically by using the GLU function `gluBuild2DMipmaps` instead of `glTexImage2D`.

Once a texture is bound and texturing is enabled, texture coordinates must be supplied for each vertex, by calling `glTexCoord` before `glVertex`:

```
glTexCoord2f(u, v);
glVertex3f(x, y, z);
```

When textures are no longer needed, they can be removed from the graphics hardware memory with

```
glDeleteTextures(2, handles);
```


11 Basic Ray Tracing

11.1 Basics

- So far, we have considered only *local* models of illumination; they only account for incident light coming directly from the light sources.
- *Global* models include incident light that arrives from other surfaces, and lighting effects that account for global scene geometry. Such effects include:
  - Shadows
  - Secondary illumination (such as color bleeding)
  - Reflections of other objects, in mirrors, for example
- Ray Tracing was developed as one approach to modeling the properties of global illumination.
- The basic idea is as follows:

  For each pixel:
  - Cast a ray from the eye of the camera through the pixel, and find the first surface hit by the ray.
  - Determine the surface radiance at the surface intersection with a combination of local and global models.
  - To estimate the global component, cast rays from the surface point to possible incident directions to determine how much light comes from each direction. This leads to a recursive form for tracing paths of light backwards from the surface to the light sources.

> *Aside:*
> Basic Ray Tracing is also sometimes called Whitted Ray Tracing, after its inventor, Turner Whitted.

Computational Issues

- Form rays.
- Find ray intersections with objects.
- Find closest object intersections.
- Find surface normals at object intersection.
- Evaluate reflectance models at the intersection.

11.2 Ray Casting

We want to find the ray from the eye through pixel (i, j).

- Camera Model
  $\bar{e}^W$ is the origin of the camera, in world space.
  $\vec{u}$, $\vec{v}$, and $\vec{w}$ are the world space directions corresponding to the $\vec{x}$, $\vec{y}$, and $\vec{z}$ axes in eye space.
  The image plane is defined by $(\bar{p} - \bar{r}) \cdot \vec{w} = 0$, or $\bar{r} + a\vec{u} + b\vec{v}$, where $\bar{r} = \bar{e}^W + f\vec{w}$.

- Window
  A window in the view-plane is defined by its boundaries in camera coordinates: w_l, w_r, w_t, and w_b. (In other words, the left-most edge is the line (w_l, λ, f).)

- Viewport
  Let the viewport (i.e., output image) have columns $0...n_c - 1$ and rows $0...n_r - 1$. $(0, 0)$ is the upper left entry.
  The camera coordinates of pixel (i, j) are as follows:

$$\bar{p}^C_{i,j} = (w_l + i\Delta u, w_t + j\Delta v, f)$$

$$\Delta u = \frac{w_r - w_l}{n_c - 1}$$

$$\Delta v = \frac{w_b - w_t}{n_r - 1}$$

  In world coordinates, this is:

$$\bar{p}^W_{i,j} = \begin{pmatrix} | & | & | \\ \vec{u} & \vec{v} & \vec{w} \\ | & | & | \end{pmatrix} \bar{p}^C_{i,j} + \bar{e}^W$$

- Ray: Finally, the ray is then defined in world coordinates as follows:

$$\bar{r}(\lambda) = \bar{p}^W_{i,j} + \lambda\vec{d}_{i,j}$$

  where $\vec{d}_{i,j} = \bar{p}^W_{i,j} - \bar{e}^W$. For $\lambda > 0$, all points on the ray lie in front of the viewplane along a single line of sight.

11.3 Intersections

In this section, we denote a ray as $\bar{r}(\lambda) = \bar{a} + \lambda\vec{d}$, $\lambda > 0$.

11.3.1 Triangles

Define a triangle with three points, $\bar{p}_1$, $\bar{p}_2$, and $\bar{p}_3$. Here are two ways to solve for the ray-triangle intersection.

- Intersect $\bar{r}(\lambda)$ with the plane $(\bar{p} - \bar{p}_1) \cdot \vec{n} = 0$ for $\vec{n} = (\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)$ by substituting $\bar{r}(\lambda)$ for $\bar{p}$ and solving for λ. Then test the half-planes for constraints. For example:

$$(\bar{a} + \lambda\vec{d} - \bar{p}_1) \cdot \vec{n} = 0$$

$$\lambda^* = \frac{(\bar{p}_1 - \bar{a}) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}$$

  What does it mean when $\vec{d} \cdot \vec{n} = 0$? What does it mean when $\vec{d} \cdot \vec{n} = 0$ and $(\bar{p}_1 - \bar{a}) \cdot \vec{n} = 0$?

- Solve for α and β where $\bar{p}(\alpha, \beta)\bar{p}_1 + \alpha(\bar{p}_2 - \bar{p}_1) + \beta(\bar{p}_3 - \bar{p}_1)$, i.e. $\bar{r}(\lambda) = \bar{a} + \lambda\vec{d} = \bar{p}_1 + \alpha(\bar{p}_2 - \bar{p}_1) + \beta(\bar{p}_3 - \bar{p}_1)$. This leads to the 3x3 system

$$\begin{pmatrix} | & | & | \\ -(\bar{p}_2 - \bar{p}_1) & -(\bar{p}_3 - \bar{p}_1) & \vec{d} \\ | & | & | \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \lambda \end{pmatrix} = (\bar{p}_1 - \bar{a})$$

  Invert the matrix and solve for α, β, and λ. The intersection is in the triangle when the following conditions are all true:

$$\alpha \geq 0$$
$$\beta \geq 0$$
$$\alpha + \beta \leq 1$$

11.3.2 General Planar Polygons

For general planar polygons, solve for the intersection with the plane. Then form a ray $\mathbf{s}(t)$ in the plane, starting at the intersection $\bar{p}(\lambda^*)$. Measure the number of intersections with the polygon sides for $t > 0$. If there is an even number of intersections, the intersection is inside. If the number of intersection is odd, it is outside.

> *Aside:*
> This is a consequence of the Jordan Curve Theorem. As related to this problem, it states that two points are both inside or both outside when the number of intersections on a line between them is even.

11.3.3 Spheres

Define the unit sphere centered at $\bar{c}$ by $||\bar{p} - \bar{c}||^2 = 1$.
Substitute a point on the ray $\bar{r}(\lambda)$ into this equation:

$$(\bar{a} + \lambda\vec{d} - \bar{c}) \cdot (\bar{a} + \lambda\vec{d} - \bar{c}) - 1 = 0$$

Expand this equation and write it in terms of the quadratic form:

$$A\lambda^2 + 2B\lambda + C = 0$$
$$A = \vec{d} \cdot \vec{d}$$
$$B = (\bar{a} - \bar{c}) \cdot \vec{d}$$
$$C = (\bar{a} - \bar{c}) \cdot (\bar{a} - \bar{c}) - 1$$

The solution is then:

$$\lambda = \frac{-2B \pm \sqrt{4B^2 - 4AC}}{2A} = -\frac{B}{A} \pm \frac{\sqrt{D}}{A}, D = B^2 - AC$$

If $D < 0$, there are no intersections. If $D = 0$, there is one intersection; the ray grazes the sphere. If $D > 0$, there are two intersections with two values for λ, λ_1 and λ_2.

When $D > 0$, three cases of interest exist:

- $\lambda_1 < 0$ and $\lambda_2 < 0$. Both intersections are behind the view-plane, and are not visible.
- $\lambda_1 > 0$ and $\lambda_2 < 0$. The $\bar{p}(\lambda_1)$ is a visible intersection, but $\bar{p}(\lambda_1)$ is not.
- $\lambda_1 > \lambda_2$ and $\lambda_2 > 0$. Both intersections are in front of the view-plane. $\bar{p}(\lambda_2)$ is the closest intersection.

11.3.4 Affinely Deformed Objects

Proposition: Given an intersection method for an object, it is easy to intersect rays with affinely deformed versions of the object. We assume here that the affine transformation is invertible.

- Let $F(\bar{y}) = 0$ be the deformed version of $f(\bar{x}) = 0$, where $\bar{y} = \mathbf{A}\bar{x} + \vec{t}$.
  i.e. $F(\bar{y}) = f(\mathbf{A}^{-1}(\bar{y} - \vec{t})) = 0$, so $F(\bar{y}) = 0$ iff $f(\bar{x}) = 0$.
- Given an intersection method for $f(\bar{x}) = 0$, find the intersection of $\bar{r}(\lambda) = \bar{a} + \lambda\vec{d}$ and $F(\bar{y}) = 0$, where $\lambda > 0$.
- **Solution:** Substitute $\bar{r}(\lambda)$ into the implicit equation $f = F(\bar{y})$:

$$\begin{aligned} F\left(\bar{r}(\lambda)\right) &= f\left(\mathbf{A}^{-1}\left(\bar{r}\left(\lambda\right) - \vec{t}\right)\right) \\ &= f(\mathbf{A}^{-1}(\bar{a} + \lambda\vec{d} - \vec{t})) \\ &= f(\bar{a}' + \lambda\vec{d}') \\ &= f(\bar{r}'(\lambda)) \end{aligned}$$

where

$$\bar{a}' = \mathbf{A}^{-1}(\bar{a} - \vec{t})$$
$$\vec{d}' = \mathbf{A}^{-1}\vec{d}$$

i.e. intersecting $F(\bar{y})$ with $\bar{r}(\lambda)$ is like intersecting $f(\mathbf{x})$ with $\bar{r}'(\lambda) = \bar{a}' + \lambda\vec{d}'$ where $\lambda > 0$. The value of λ found is the same in both cases.

- **Exercise:** Verify that, at the solution λ^*, with an affine deformation $\bar{y} = \mathbf{A}\bar{x} + \vec{t}$, that $\bar{r}(\lambda^*) = \mathbf{A}\bar{r}'(\lambda^*) + \vec{t}$.

11.3.5 Cylinders and Cones

A right-circular cylinder may be defined by $x^2 + y^2 = 1$ for $|z| \leq 1$. A cone may be defined by $x^2 + y^2 - \frac{1}{4}(1 - z^2) = 0$ for $0 \leq z \leq 1$.

- Find intersection with "quadratic wall," ignoring constraints on z, e.g. using $x^2 + y^2 = 1$ or $x^2 + y^2 - \frac{1}{4}(1 - z^2) = 0$. Then test the z component of $\bar{p}(\lambda^*)$ against the constraint on z, e.g. $z \leq 1$ or $z < 1$.

- Intersect the ray with the planes containing the base or cap (e.g. $z = 1$ for the cylinder). Then test the x and y components of $\bar{p}(\lambda^*)$ to see if they satisfy interior constraints (e.g. $x^2 + y^2 < 1$ for the cylinder).

- If there are multiple intersections, then take the intersection with the smallest positive λ (i.e., closest to the start of the ray).

11.4 The Scene Signature

The scene signature is a simple way to test geometry intersection methods.

- Create an image in which pixel (i, j) has intensity k if object k is first object hit from ray through (i, j).
- Each object gets one unique color.

Note:
Pseudo-Code: Scene Signature

```
<  Construct scene model = { obj, (A, t⃗), objID }  >
sig: array[nc, nr] of objID
for  j = 0 to nr-1 (loop over rows)
    for  i = 0 to nc-1 (loop over columns)
        <  Construct ray r⃗_ij(λ) = p̄_ij + λ(p̄_ij − ē) through pixel p̄_ij  >
        λ_{i,j} ← ∞
        loop  over all objects in scene, with object identifiers objID_k
            <  find λ* for the closest intersection of the ray r⃗_ij(λ) and the object  >
            if  λ* > 0  and  λ* < λ_{i,j}  then
                λ_{i,j}  ←  λ*
                sig[i,j].objID  ←  objID_k
            end if
        end loop
    end for
end for
```

11.5 Efficiency

Intersection tests are expensive when there are large numbers of objects, and when the objects are quite complex! Fortunately, data structures can be used to avoid testing intersections with objects that are not likely to be significant.

Example: We could bound a 3D mesh or object with a simple bounding volume (e.g. sphere or cube). Then we would only test intersections with objects if there exists a positive intersection with the bounding volume.

Example: We could project the extent onto the image plane so you don't need to cast rays to determine potential for intersections.

11.6 Surface Normals at Intersection Points

Once we find intersections of rays and scene surfaces, and we select the first surface hit by the ray, we want to compute the shading of the surface as seen from the ray. That is, we cast a ray out from a pixel and find the first surface hit, and then we want to know how much light leave the surface along the same ray but in the reverse direction, back to the camera.
Toward this end, one critical property of the surface geometry that we need to compute is the surface normal at the hit point.

- For mesh surfaces, we might interpolate smoothly from face normals (like we did to get normals at a vertex). This assumes the underlying surface is smooth.
- Otherwise we can just use the face normal.
- For smooth surfaces (e.g. with implicit forms $f(\bar{p}) = 0$ or parametric forms $\mathbf{s}(\alpha, \beta)$), either take

$$\vec{n} = \frac{\nabla f(\bar{p})}{||\nabla f(\bar{p})||}$$

or

$$\vec{n} = \frac{\frac{\partial \mathbf{s}}{\partial \alpha} \times \frac{\partial \mathbf{s}}{\partial \beta}}{||\frac{\partial \mathbf{s}}{\partial \alpha} \times \frac{\partial \mathbf{s}}{\partial \beta}||}.$$

11.6.1 Affinely-deformed surfaces.

Let $f(\bar{p}) = 0$ be an implicit surface, and let $Q(\bar{p}) = \mathbf{A}\bar{p} + \vec{t}$ be an affine transformation, where $\mathbf{A}$ is invertible. The affinely-deformed surface is

$$F(\bar{q}) = f(Q^{-1}(\bar{p})) = f(\mathbf{A}^{-1}(\bar{p} - \vec{t})) = 0 \tag{42}$$

A normal of F at a point $\bar{q}$ is given by

$$\frac{\mathbf{A}^{-T}\vec{n}}{||\mathbf{A}^{-T}\vec{n}||} \tag{43}$$

where $\mathbf{A}^{-T} = (\mathbf{A}^{-1})^T$ and $\vec{n}$ is the normal of f at $\bar{p} = Q^{-1}(\bar{q})$.

Derivation:
Let $\bar{s} = \bar{r}(\lambda^*)$ be the intersection point, and let $(\bar{p} - \bar{s}) \cdot \vec{n} = 0$ be the tangent plane at the intersection point. We can also write this as:

$$(\bar{p} - \bar{s})^T \vec{n} = 0 \tag{44}$$

Substituting in $\bar{q} = \mathbf{A}\bar{p} + \vec{t}$ and solving gives:

$$(\bar{p} - \bar{s})^T \vec{n} \;=\; (\mathbf{A}^{-1}(\bar{q} - \vec{t}) - \bar{s})^T \vec{n} \tag{45}$$

$$= \quad (\bar{q} - (\mathbf{A}\bar{s} + \vec{t}))^T \mathbf{A}^{-T} \vec{n} \tag{46}$$

In other words, the tangent plane at the transformed point has normal $\mathbf{A}^{-T}\vec{n}$ and passes through point $(\mathbf{A}\bar{s} + \vec{t})$.
preserved so the tangent plane on the deformed surface is given by $(\mathbf{A}^{-1}(\bar{q} - \vec{t}))^T \vec{n} = D$.

This is the equation of a plane with *unit* normal $\frac{\mathbf{A}^{-T}\vec{n}}{||\mathbf{A}^{-T}\vec{n}||}$.

11.7 Shading

Once we have cast a ray through pixel $\bar{p}_{i,j}$ in the direction $\vec{d}_{i,j}$, and we've found the closest hit point $\bar{p}$ with surface normal $\vec{n}$, we wish to determine how much light leaves the surface at $\bar{p}$ into the direction $-\vec{d}_{i,j}$ (i.e., back towards the camera pixel). Further we want reflect both the light from light sources that directly illuminate the surface as well as secondary illumination, where light from other surfaces shines on the surface at $\bar{p}$. This is a complex task since it involves all of the ways in which light could illuminate the surface from all different directions, and the myriad ways such light interacts with the surface and it then emitted or reflected by the surface. Here we will deal first with the simplest case, known widely as Whitted Ray Tracing.

Aside:
First, note that if we were to ignore all secondary reflection, then we could just compute the Phong reflectance model at $\bar{p}$ and then color the pixel with that value. Such scenes would look similar to those that we have rendered using shading techniques seen earlier in the course. The main differences from earlier rendering techniques are the way in which hidden surfaces are handled and the lack of interpolation.

11.7.1 Basic (Whitted) Ray Tracing

In basic ray tracing we assume that that the light reflected from the surface is a combination of the reflection computed by the Phong model, along with one component due to specular secondary reflection. That is, the only reflection we consider is that due to perfect mirror reflection. We only consider perfect specular reflection for computational efficiency; i.e., rather than consider secondary illumination at $\bar{p}$ from all different directions, with perfect specular reflection we know that the only incoming light at $\bar{p}$ that will be reflected in the direction $-\vec{d}_{i,j}$ will be that coming from the corresponding mirror direction (i.e., $\vec{m}_s = -2(\vec{d}_{i,j} \cdot \vec{n}))\vec{n} + \vec{d}_{i,j}$). We can find out how much light is incoming from direction $\vec{m}_s$ be casting another ray into that direction from $\bar{p}$ and calculating the light reflected from the first surface hit. Note that we have just described a recursive ray tracer; i.e., in order to calculate the reflectance at a hit point we need to cast more rays and compute the reflectance at the new hit points so we can calculate the incoming light at the original hit point.

In summary, for basic (Whitted) ray tracing, the reflectance model calculation comprises:

- A local model (e.g., Phong) to account for diffuse and off-axis specular reflection (highlights) due to light sources.
- An ambient term to approximate the global diffuse components.
- Cast rays from $\bar{p}$ into direction $\vec{m}_s = -2(\vec{d}_{i,j} \cdot \vec{n}))\vec{n} + \vec{d}_{i,j}$ to estimate ideal mirror reflections due to light coming from other ojects (i.e., secondary reflection).

For a ray $r(\lambda) = \bar{a} + \lambda\vec{d}$ which hits a surface point point $\bar{p}$ with normal $\vec{n}$, the reflectance is given by

$$E = r_a I_a + r_d I_d \max(0, \vec{n} \cdot \vec{s}) + r_s I_s \max(0, \vec{c} \cdot \vec{m})^\alpha + r_g I_{spec}$$

where r_a, r_d, and r_s are the reflection coefficients of the Phong model, I_a, I_d, and I_s are the light source intensities for the ambient, diffuse and specular terms of the Phong model, $\vec{s}$ is the light source direction from $\bar{p}$, the emittant direction of interest is $\vec{c} = -\vec{d}_{i,j}$, and $\vec{m} = 2(\vec{s} \cdot \vec{n}))\vec{n} - \vec{s}$ is the perfect mirror direction for the local specular reflection. Finally, I_{spec} is the light obtained from the recursive ray cast into the direction $\vec{m}_s$ to find secondary illumination, and r_g is the reflection coefficient that determines the fraction of secondary illumination that is reflected by the surface at $\bar{p}$

11.7.2 Texture

- Texture can be used to modulate diffuse and a mbient reflection coefficients, as with Gouraud shading.
- We simply need a way to map each point on the surface to a point in texture space, as above, e.g. given an intersection point $\bar{p}(\lambda^*)$, convert into parametric form $\mathbf{s}(\alpha, \beta)$ and use (α, β) to find texture coordinates (μ, ν).
- Unlike Gouraud shading, we don't need to interpolate (μ, ν) over polygons. We get a new (μ, ν) for each intersection point.
- Anti-aliasing and super-sampling are covered in the Distribution Ray Tracing notes.

11.7.3 Transmission/Refraction

- Light that penetrates a (partially or wholly) transparent surface/material is refracted (bent), owing to a change in the speed of light in different media.
- Snell's Law governs refraction:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{c_1}{c_2}$$

- The index of refraction is the ratio of light speeds c_1/c_2. For example, the index of refraction for passing from air to water is $\frac{c_{air}}{c_{water}} = 1.33$, and for passing from air to glass, it is $\frac{c_{air}}{c_{glass}} = 1.8$.
  Note: There is also a wavelength dependence. We ignore this here.

- Example:
  - If $c_2 < c_1$, light bends towards the normal (e.g. air to water). If $c_2 < c_1$, light bends away from the normal (e.g. water to air).
  - The critical angle θ_c, when $c_2 > c_1$, is when $\theta_1 \rightarrow \theta_c$ and $\theta_2 \rightarrow 90$. Beyond θ_c, $\theta_1 > \theta_c$, and total internal reflection occurs. No light enters the material.

- Remarks:
  - The outgoing direction is in the plane of the incoming direction and $\vec{n}$. This is similar to the perfect specular direction.
  - When $\theta_1 = 0$, then $\theta_2 = 0$, i.e. there is no bending.

- For ray tracing:
  - Treat global transmission like global specular, i.e. cast one ray.
  - Need to keep track of the speed of light in the current medium.

11.7.4 Shadows

- A simple way to include some global effects with minimal work is to turn off local reflection when the surface point $\bar{p}$ cannot see light sources, i.e. when $\bar{p}$ is in shadow.

- When computing E at $\bar{p}$, cast a ray toward the light source, i.e. in the direction $\mathbf{s} = (\mathbf{l} - \bar{p})$.

$$\bar{p}^W(\lambda) = \bar{p}^W + \lambda(\mathbf{l}^W - \bar{p}^W)$$

- Find the first intersection with a surface in the scene. If λ^* at the first intersection point is $0 \leq \lambda \leq 1$, then there exists a surface that occludes the light source from $\bar{p}$.
  - We should omit diffuse and specular terms from the local Phong model.
  - The surface radiance at $\bar{p}$ becomes

$$E = r_a I_a + r_g I_{spec}$$

Note:

Pseudo-Code: Recursive Ray Tracer

for each pixel (i,j)
  $<$ compute ray $\vec{\mathbf{r}}_{ij}(\lambda) = \bar{\mathbf{p}}_{ij} + \lambda\vec{\mathbf{d}}_{ij}$ where $\vec{\mathbf{d}}_{ij} = \bar{\mathbf{p}}_{ij} - \vec{\mathbf{e}}$ $>$
  I = rayTrace($\bar{\mathbf{p}}_{ij}$, $\vec{d}_{ij}$, 1);
  setpixel(i, j, I)
end **for**

rayTrace($\bar{\mathbf{a}}$, $\vec{\mathbf{b}}$, depth)
  findFirstHit($\bar{\mathbf{a}}$, $\vec{\mathbf{b}}$, *output var* obj, λ, $\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$)
  if $\lambda > 0$ then
    I = rtShade(obj, $\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$, $-\vec{\mathbf{b}}$, depth)
  else
    I = background;
  end **if**
  return(I)

findFirstHit($\bar{\mathbf{a}}$, $\vec{\mathbf{b}}$, *output var* OBJ, λ_h, $\bar{\mathbf{p}}_h$, $\vec{\mathbf{n}}_h$)
  $\lambda_h = -1$;
  loop over all objects in scene, with object identifiers objID_k
    $<$ find $\lambda*$ for the closest legitimate intersection of ray $\vec{\mathbf{r}}_{ij}(\lambda)$ and object $>$
    if ($\lambda_h < 0$ or $\lambda^* < \lambda_h$) and $\lambda^* > 0$ then
      $\lambda_h = \lambda^*$
      $\bar{\mathbf{p}}_h$ = $\bar{\mathbf{a}} + \lambda^*\vec{\mathbf{b}}$;
      $<$ determine normal at hit point $\vec{\mathbf{n}}_h$ $>$
      OBJ = objID_k
    end **if**
  end **loop**

Note:

rtShade(OBJ, $\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$, $\vec{\mathbf{d}}_e$, depth)
    /* Local Component */
    findFirstHit($\bar{\mathbf{p}}$, $\vec{\mathbf{l}}^w - \bar{\mathbf{p}}$, *output var* temp, λ_h);
    if $0 < \lambda_h < 1$ then
        I_l = ambientTerm;
    else
        I_l = phongModel($\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$, $\vec{\mathbf{d}}_e$, OBJ.localparams)
    end **if**
    /* Global Component */
    if depth $<$ maxDepth then
        if OBJ has specular reflection then
            $<$ calculate mirror direction $\vec{\mathbf{m}}_s = -\vec{\mathbf{d}}_e + 2\vec{\mathbf{n}} \cdot \vec{\mathbf{d}}_e\vec{\mathbf{n}}$ $>$
            I_{spec} = rayTrace($\bar{\mathbf{p}}$, $\vec{\mathbf{m}}_s$, depth+1)
            $<$ scale I_{spec} by OBJ.specularReflCoef $>$
        end **if**
        if OBJ is refractive then
            $<$ calculate refractive direction $\vec{\mathbf{t}}$ $>$
            if not total internal reflection then
                I_{refr} = rayTrace($\bar{\mathbf{p}}$, $\vec{\mathbf{t}}$, depth+1)
                $<$ scale I_{refr} by OBJ.refractiveReflCoef $>$
            end **if**
        end **if**
        $I_g = I_{spec} + I_{refr}$
    else
        $I_g = 0$
    end **if**
    return($I_l + I_g$)

12 Radiometry and Reflection

Until now, we have considered highly simplified models and algorithms for computing lighting and reflection. These algorithms are easy to understand and can be implemented very efficiently; however, they also lack realism and cannot achieve many important visual effects. In this chapter, we introduce the fundamentals of radiometry and surface reflectance that underly more sophisticated models. In the following chapter, we will describe more advanced ray tracing algorithms that take advantage of these models to produce very realistic and simulate many real-world phenomena.

12.1 Geometry of lighting

In our discussion of lighting and reflectance we will make several simplifying assumptions. First, we will ignore time delays in light propagation from one place to another. Second, we will assume that light is not scattered nor absorbed by the median through which it travels, i.e., we will ignore light scattering due to fog. These assumptions allow us to focus on the *geometry* of lighting; i.e., we can assume that light travels along straight lines, and is conserved as it travels (e.g., see Fig. 1).

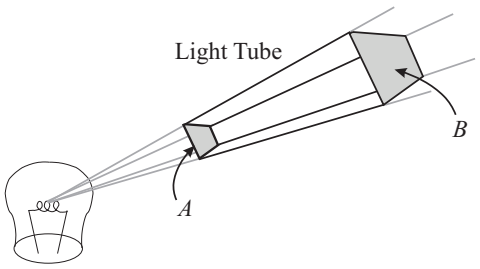


Figure 4: Given a set of rays within a tube, passing through A and B but not the sides of the tube, the flux (radiant power) at A along these rays is equal to that at B along the same set of rays.

Before getting into the details of lighting, it will be useful to introduce three key geometric concepts, namely, *differential areas*, *solid angle* and *foreshortening*. Each of these geometric concepts is related to the dependence of light on the distance and orientation between surfaces in a scene that receive or emit light.

Area differentials: We will need to be able describe the amount of lighting that hitting an area on a surface or passing through a region of space. Integrating functions over a surface requires that we introduce an *area differential* over the surface, denoted dA. Just as a 1D differential (dx) represents an infinitesimal region of the real line, an area differential represents an infinitesimal region on a 2D surface.

Example:
Consider a rectangular patch S in the $x - y$ plane. We can specify points in the patch in terms of an x coordinate and a y coordinate, with $x \in [x_0, x_1], y \in [y_0, y_1]$. We can

divide the plane into NM rectangular subpatches, the ij-th subpatch bounded by

$$x_i \leq x \leq x_i + \Delta x \tag{47}$$
$$y_j \leq y \leq y_j + \Delta y \tag{48}$$

where $i \in [0...N-1], j \in [0...M-1]$, $\Delta x = (x_1 - x_0)/N$ and $\Delta y = (y_1 - y_0)/M$. The area of each subpatch is $A_{i,j} = \Delta x \Delta y$. In the limit as $N \to \infty$ and $M \to \infty$,

$$dA = dxdy \tag{49}$$

To compute the area of a smooth surface S, we can break the surface into many tiny patches (i, j), each with area $A_{i,j}$, and add up these individual areas:

$$\text{Area}(S) = \sum_{i,j} A_{i,j} \tag{50}$$

In the planar patch above, the area of the patch is:

$$\text{Area}(S) = \sum_{i,j} A_{i,j} = NM\Delta x \Delta y = (x_1 - x_0)(y_1 - y_0) \tag{51}$$

Computing these individual patch areas for other surfaces is difficult. However, taking the infinite limit we get the general formula:

$$\text{Area}(S) = \int_S dA \tag{52}$$

For the planar patch, this becomes:

$$\int_S dA = \int_{y_0}^{y_1} \int_{x_0}^{x_1} dxdy = (x_1 - x_0)(y_1 - y_0) \tag{53}$$

We can create area differentials for any smooth surface. Fortunately, in most radiometry applications, we do not actually need to be able to do so for anything other than a plane. We will use area differentials when we integrate light on the image sensor, which, happily, is planar. However, area differentials are essential to many key definitions and concepts in radiometry.

Solid angle: We need to have a measure of *angular extent* in 3D. For example, we need to be able to talk about what we mean by the field of view of a camera, and we need a way to quantitfy the width of a directional light (e.g., a spot light).

Let's consider the situation in 2D first. In 2D, *angular extent* is just the angle between two directions, and we normally specify angular extent in *radians*. In particular, the angular extent between two rays emanating from a point $\bar{q}$ can be measured using a circle centered at $\bar{q}$; that is, the angular extent (in radians) is just the circular arc length l of the circle between the two directions, divided by radius r of the circle, l/r (see Fig. 5). For example, the angular extent of an entire circle having circumference $2\pi r$ is just 2π radians. A half-circle has arclength πr and spans π radians.

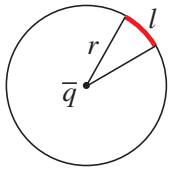


Figure 5: Angular extent in 2D is given by l/r (radians).

In 3D, the corresponding quantity to 2D angular extent is called *solid angle*. Analogous to the 2D case, solid angle is measured as the area a of a patch on a sphere, divided by the squared radius of the sphere (Figure 6); i.e.,

$$\omega = \frac{a}{r^2} \tag{54}$$

The unit of measure for solid angle is the *steradian* (sr). A solid angle of 2π steradians corresponds to a hemisphere of directions. The entire sphere has a solid angle of 4π sr. As depicted in Figure 2, to find the solid angle of a surface S with respect to a point $\bar{q}$, one projects S onto a sphere of radius r, centered at $\bar{q}$, along lines through $\bar{q}$. This gives us a, so we then divide by r^2 to find the solid angle subtended by the surface. Note that the solid angle of a patch does not depend on the radius r, since the projected area a is proportional to r^2.

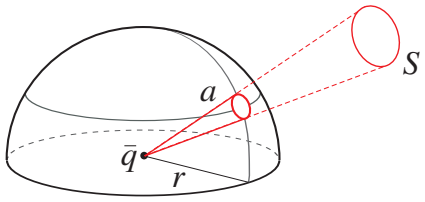


Figure 6: The solid angle of a patch S is given by the area a of its projection onto a sphere of radius r, divided by the squared radius, r^2.

Note:
At a surface point with normal $\vec{n}$, we express the hemisphere of incident and emittant directions in spherical coordinates. That is, directions in the hemisphere $\vec{d}$ are

$$\vec{d} = (\sin\theta\cos\phi,\ \sin\theta\sin\phi,\ \cos\theta)^T \tag{55}$$

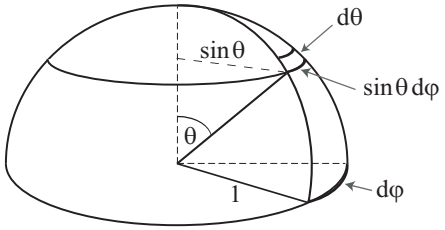
where $\theta \in [0, \pi/2]$ denotes the angle between $\vec{d}$ and the normal, and $\phi \in [-\pi, \pi)$ measures the direction projected onto the surface.

With direction expressed in this way one can write the infinitesimal solid angle as

$$d\omega = \sin\theta \, d\theta \, d\phi \tag{56}$$

The infinitesimal solid angle is an area differential for the unit sphere.

To see this, note that for θ held fixed, if we vary ϕ we trace out a circle of radius $\sin\theta$ that is perpendicular to $\vec{n}$. For a small change $d\phi$, the circular arc has length $\sin\theta \, d\phi$, and therefore the area of a small ribbon of angular width $d\theta$ is just $\sin\theta \, d\theta \, d\phi$.



This also allows us to compute the finite solid angle for a a range of visual direction, such as $\theta_0 \leq \theta \leq \theta_1$ and $\phi_0 \leq \phi \leq \phi_1$. That is, to compute the solid angle we just integrate the differential solid angle over this region on a unit sphere ($r = 1$):

$$\omega = \int_{\phi_0}^{\phi_1} \int_{\theta_0}^{\theta_1} \sin\theta \; d\theta \; d\phi \tag{57}$$

$$= \int_{\phi_0}^{\phi_1} -\cos\theta \Big|_{\theta_0}^{\theta_1} \; d\phi \tag{58}$$

$$= (\phi_1 - \phi_0)(\cos\theta_0 - \cos\theta_1) \tag{59}$$

(Assuming we are in the quadrant where this quantity is positive)

Foreshortening: Another important geometric property is *foreshortening*, the reduction in the (projected) area of a surface patch as seen from a particular point or viewer. When the surface normal points directly at the viewer its effective size (solid angle) is maximal. As the surface normal rotates away from the viewer it appears smaller (Figure 7). Eventually when the normal is pointing perpendicular to the viewing direction you see the patch "edge on"; so its projection is just a line (with zero area).

Putting it all together: Not surprisingly, the solid angle of a small surface patch, with respect to a specific viewing location, depends on both on the distance from the viewing location to the patch, and on the orientation of the patch with respect to the viewing direction.

Figure 7: Foreshortening in 2D. *Left:* For a patch with area A, seen from a point $\bar{q}$, the patch's foreshortened area is approximately $A\cos\theta$. This is an approximation, since the distance r varies over the patch. The angle θ is the angle between the patch normal and the direction to $\bar{q}$. *Right:* For an infinitesimal patch with area dA, the foreshortened area is exactly $dA\cos\theta$.

Let $\bar{q}$ be the point (such as a light source or a viewer) about which we want to compute solid angle. Let $\bar{p}$ be the location of a small planar surface patch S with area A at distance $r = ||\bar{q} - \bar{p}||$ from $\bar{q}$. Additionally, suppose the surface normal points directly at $\bar{q}$ (Figure 8). In this case, we can imagine drawing a hemisphere about $\bar{q}$ with radius r, and the projected area a of this patch will be approximately A. Hence, the solid angle $\omega \approx A/r^2$. In other words, the solid angle is inversely proportional to distance squared; a more distant object obscures less of $\bar{q}$'s "field of view." This is an approximation, however, since the distance r varies over the patch. Nevertheless, if we consider the limit of an infinitesimal patch with area dA, then the solid angle is exactly $d\omega = dA/r^2$.

When the surface normal does not point directly at $\bar{q}$, foreshortening plays a significant role. As the surface normal rotates away from the direction of $\bar{q} - \bar{p}$, the surface, as viewed from point $\bar{q}$, becomes smaller; it projects onto a smaller area on a sphere centered at $\bar{q}$. sphere. So, we say that the area of the patch, as seen from $\bar{q}$, is *foreshortened*. More formally, let θ be the angle between the normal $\vec{n}$ and direction, $\bar{q} - \bar{p}$. Then, for our infinitesimal surface with area dA, the solid angle subtended by the tilted patch is

$$d\omega = \frac{dA\cos\theta}{r^2}\,, \tag{60}$$

The cosine term should look familiar; this is the same cosine term used in Lambertian shading within the Phong model.

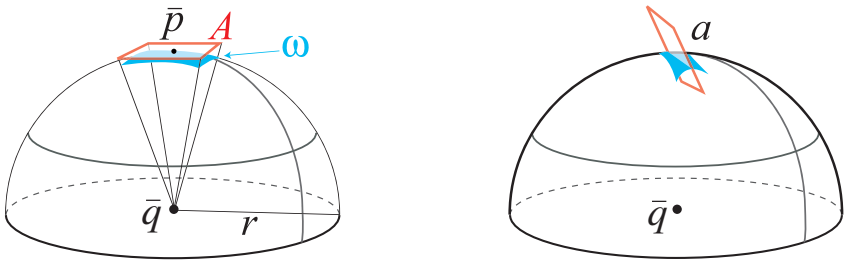


Figure 8: Solid angle of a patch. *Left:* A patch with normal pointing at $\bar{l}$. *Right:* A patch with arbitrary orientation.

12.2 Elements of Radiometry

The field of radiometry concerns the measurement of light (electromagnetic radiation), usually restricted to the visible wavelengths, in the range 400-700 nm. Light is often measured in discrete units called photons. It is difficult to talk about the number of photons that illuminate a point on a surface at a particular time (as it is almost always zero). Instead, we talk about the average number of photons in small (infinitesimal) intervals of space or time, that is, we talk about photon density, and thereby treat light as a continuous quantity rather than a photon count. In effect, we are assuming that there is enough light in the scene so that we can treat light as a continuous function of space-time. For example, we will talk about the light hitting a specific surface patch as a continuous function over the patch, rather than discuss the discrete photons of light.

12.2.1 Basic Radiometric Quantities

Formally, we describe light in terms of *radiant energy*. You can think of radiant energy as the totality of the photons emitted from a body over its entire surface and over the entire period of time it emits light. Radiant energy is denoted by $Q(t)$ and measured in Joules (J). You can think of radiant energy as describing how much light has been emitted from (or received by) a surface up to a time t, starting from some initial time 0. [2]

The main quantity of interest in radiometry is *power*, that is, the rate at which light energy is emitted or absorbed by an object. This time-varying quantity, usually called ***flux***, is measured in Joules per second ($\mathrm{J} \cdot \mathrm{s}^{-1}$). Here we denote flux by $\Phi(t)$:

$$\Phi(t) = \frac{dQ(t)}{dt} \tag{61}$$

We can compute the total light that hits a surface up to time t as:

$$Q(t) = \int_0^t \Phi(\tau)\, d\tau \tag{62}$$

Flux is sufficiently important that we define a special unit of measure for it, namely, watts (W). One watt is one Joule per second; so a 50 watt light bulb draws 50J of energy per second. Most of this radiant energy is emitted as visible light. The rest is converted to thermal energy (heat). Higher wattage means a brighter light bulb.

Not surprisingly, the light received or emitted by an object varies over the surface of the object. This is important since the appearance of an object is often based on how the light reflected from

[2] Of course, radiant energy depends on wavelength λ, so it is common to express energy as a function of wavelength; the resulting density function, $Q(\lambda)$, is called spectral energy. This is important since different wavelengths are seen as different colours. Nevertheless, our next major simplification will be to ignore the dependence of radiant energy on wavelength. In computer graphics, colours are controlled by the relative amounts of power in three separate spectral bands, namely, Red, Green, and Blue. What we describe in this chapter can be applied to each colour channel.

its surface depends on surface position. Formally, light *received* at the surface of an object, as a function of image position is called *irradiance*. The light *emitted* from a surface, as a function of surface position, is often called *radiant exitance* (or *radiosity*).

Irradiance, the incident flux, as a function of surface position $\bar{p}$, is denoted by $H(\bar{p})$. Remember, we cannot talk about the amount of light received at a single point on a surface because the number of photons received at a single point is generally zero. Instead, irradiance is the spatial density of flux, i.e., the amount of light per unit surface area. The integral of irradiance over the surface of an object gives us the total incident flux (i.e., received by) the object. Accordingly, irradiance is the spatial derivative of flux. For smooth surfaces we write

$$H(\bar{p}) = \frac{d\Phi}{dA} \tag{63}$$

where dA refers to differential surface area. Irradiance is just power per unit surface area ($\mathrm{W}\cdot\mathrm{m}^{-2}$).

Example:
For a planar patch in the $x-y$ plane, we can write irradiance as a function of (x, y) position on the patch. Also, we have $dA = dxdy$. In this case:

$$H(x, y) = \frac{d^2\Phi}{dxdy} \tag{64}$$

These terms are all functions of time t, since lighting Φ may change over time t. However, we will leave the dependence on time t implicit in the equations that follow for notational simplicity.

Example:
What is the irradiance, owing to a point light source, on an infinitesimal patch S with area dA? Let's say we have a point light source at $\bar{l}$ emitting I watts per steradian into all directions:

$$d\Phi = I d\omega \tag{65}$$

In other words, the amount of light from this source is proportional to solid angle, and independent of direction. Our goal is to compute the irradiance H on the patch, which can be done by subtitution of formulas from this chapter:

$$H = \frac{d\Phi}{dA} = \frac{I d\omega}{dA} = \frac{I dA \cos\theta}{dA r^2} = \frac{I \cos\theta}{r^2} \tag{66}$$

where $\bar{p}$ is the position of S, $r = ||\bar{l} - \bar{p}||$, and θ is the angle between the surface normal and the vector $\bar{l} - \bar{p}$. This formula illustrates the importance of solid angle: *the amount of light hitting a surface is proportional to its solid angle with respect to*

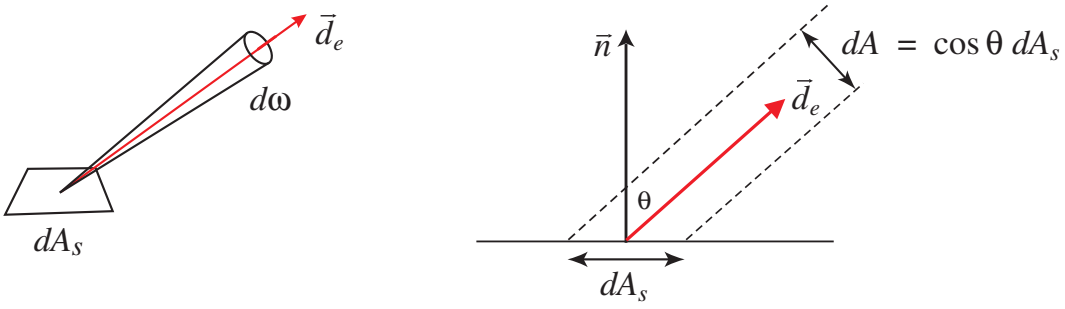
the light source. A distant patch (with large r) receives less light than a nearby patch, and a foreshortened patch receives less light than a frontal patch. Furthermore, the amount of light hitting the patch is proportional to the intensity I of the light source.

12.2.2 Radiance

Of course the light emitted or received by an object depends on visual direction as well as surface position. For example, objects are often illuminated more from above (the sky) than below (the ground). As a consequence, when the direction of light propagation is important, we will express flux as a function of visual direction. This leads to the central quantity in radiometry, namely, *radiance*. Radiance is a measure of the rate at which light energy is emitted from a surface in a particular direction. It is a function of position and direction, and it is often denoted by L (or $L(\bar{p}, \vec{d})$). Formally, it is defined as power per steradian per surface area ($\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}$), where the surface area is defined with respect to a surface patch at $\bar{p}$ that is perpendicular to the direction $\vec{d}$.

Normally, one might think of radiance as a measure of the light emitted from a particular surface location into a particular direction. The definition above is more general however. It allows us to talk about the light travelling in a particular direction through an arbitrary point in space. In this case we are measuring surface area with respect to a *virtual* surface, but we can talk about surface area nonetheless.

When we talk about the light (radiance) emitted from a particular surface into a particular emittant direction $\vec{d}_e$ we have to be a little more careful because radiance is defined with respect to a surface perpendicular to the emittant direction, which is usually not the same orientation as the actual real surface in question. Accordingly, often radiance is defined as power per unit *foreshortened* surface area per solid angle to make explicit the fact that we are using a virtual surface and not the real surface to measure area. That is, we are measuring surface area as seen by someone looking at the surface from somewhere along a ray in the emittant direction.



Note:
Computing radiant exitance (radiosity): As mentioned above, radiant exitance is the total amount of flux leaving a surface into the entire hemisphere of emittant di-

rections, as a function of surface position. Intuitively, it is the integral of surface radiance, but we have to be careful; radiance is defined with respect to unit area on a surface perpendicular to the emittant direction rather than unit area on the real surface of interest. Before we can integrate radiance we need to specify all radiance quantities in terms of unit surface area on the real surface. To do this one needs to multiply radiance for emittant direction $\vec{d_e}$ by the ratio of the surface area normal to $\vec{d_e}$ (i.e., dA), to the real surface area, denoted dA_s. As discussed above, for an infinitesimal patch the ratio of these areas is just the foreshortening factor, i.e.,

$$dA \;=\; \cos\theta \, dA_s \;=\; \vec{n} \cdot \vec{d_e} \, dA_s \;, \tag{67}$$

where θ is the angle between the unit vectors $\vec{n}$ and $\vec{d_e}$.

Taking this foreshortening factor into account, the relation between radiant exitance $E(\bar{p})$ and radiance $L(\bar{p}, \vec{d})$ is given by

$$E(\bar{p}) = \int_{\vec{d} \in \Omega_e} L(\bar{p}, \vec{d}) \; \vec{n} \cdot \vec{d} \, d\omega \tag{68}$$

The domain of integration, Ω_e, is the hemisphere of possible emittant directions.

Note:
Computing Irradiance: Above we showed that the irradiance on an infinitesimal surface patch S at point $\bar{p}$ owing to a point light source at $\bar{q}$ with radiant intensity I is given by

$$H \;=\; \frac{I \cos\theta}{r^2} \tag{69}$$

where $r = ||\bar{q} - \bar{p}||$ is the distance between the light source and the surface patch, and θ is the angle between the surface normal and the direction of the light source from the surface patch, $\bar{q} - \bar{p}$.
In this case, the radiance at $\bar{p}$ from the point light source direction $\vec{d} = \bar{p} - \bar{q}/r$, i.e., $L(\bar{p}, \vec{d})$, is simply I/r^2. The factor $\cos\theta$ is the foreshortening factor to convert from area perpendicular to the direction $\vec{d}$ to area on the surface S.
Accordingly, if we consider radiance at $\bar{p}$ from the entire hemisphere of possible incident directions, then the total irradiance at $\bar{p}$ is given by

$$H(\bar{p}) = \int_{\vec{d} \in \Omega_i} L(\bar{p}, -\vec{d}) \; \vec{n} \cdot \vec{d} \, d\omega \tag{70}$$

(Note that incident directions here are outward facing from $\bar{p}$.)

Note:
Radiance vs. Irradiance. Radiance and irradiance are very similar concepts — both describe an amount of light transmitted in space — but it is important to recognize the distinctions between them. There are several ways of thinking about the difference:

- Radiance is a function of direction; it is power per foreshortened surface area per steradian in a specific direction. Irradiance is incident power per surface area (not foreshortened); it is not a directional quantity.
- Radiance ($W \cdot sr^{-1} \cdot m^{-2}$) and irradiance ($W \cdot m^{-2}$) have different units.
- Radiance describes light emitted from a surface. Irradiance describes light incident on a surface. Indeed, from the radiance emitted from one surface we can compute the incident irradiance at a nearby surface.

12.3 Bidirectional Reflectance Distribution Function

We are now ready to explore how to model the reflectance properties of different materials. Different objects will interact with light in different ways. Some surfaces are mirror-like, while others scatter light in a wide variety of directions. Surfaces that scatter light often look matte, and appear similar from different viewing directions. Some objects absorb a significant amount of light; the colour of an object is largely a result of which wavelengths it absorbs and which wavelengths it reflects.

One simple model of surface reflectance is refered to as the bidirectional reflectance distribution function (*BRDF*). The BRDF describes how light interacts with a surface for a relatively wide range of common materials. In intuitive terms, it specifies what fraction of the incoming light from a given incident direction will be reflected toward a given emittant direction. When multiplied by the incident power (i.e., the irradiance), one obtains the desired emittant (i.e., reflected) power.

More precisely, the BRDF is a function of emittant and incident directions $\vec{d}_e$ and $\vec{d}_i$. It is defined to be the ratio of radiance to irradiance:

$$\rho(\vec{d}_e, \vec{d}_i) = \frac{L}{H} \tag{71}$$

For most common materials the only way to determine the BRDF is with measurements. That is, for a wide range of incident and emittant directions, a material is illuminated from one direction while the reflected light is measured from another direction. This is often a tedious procedure. In computer graphics it is more common to design (i.e., make up) parametric BRDF formulae, and then vary the parameters of such models to achieve the desired appearance. Most parametric models are based on analytic models of certain idealized materials, as discussed below.

12.4 Computing Surface Radiance

When rendering an image of an object or scene, one wants to know how much light is incident at each pixel of the image plane. (In effect, one wants to compute the image irradiance.) Fortunately it can be shown that this quantity is linearly related to the scene radiance. In particular, for a point on an opaque object in a given visual direction, one simply needs to compute the radiance from that point on the surface in the direction of the camera. Based on the BRDF model of reflectance, the surface radiance depends on the incident illumination (irradiance) at the surface, and the BRDF of course.

Point Light Sources

For example, consider a single point source with radiant intensity I. To compute the irradiance at a small surface patch we can compute the total flux arriving at the surface, and then divide by the area of the surface to find flux per unit area. More precisely, radiant intensity for the source is given by $I = d\Phi/d\omega$. We multiply by the solid angle subtended by the patch $d\omega$ to obtain the flux on the surface $d\Phi$, and then we divide by the surface area dA to obtain $d\Phi/dA$, that is, irradiance as in Eqn (63). For a point light source this was shown above (see Eqn. (66)) to be given by

$$H \;=\; I\,\frac{\vec{n}\cdot\vec{d_i}}{r^2} \tag{72}$$

where $\vec{n}$ is the unit surface normal, $\vec{d_i}$ is the unit vector in the direction of hte light source from the surface patch, and r is the distance from the patch to the light source.

We now want to compute the radiance from the surface (e.g., toward the camera). Toward this end, we multiply the irradiance H by the BRDF, $\rho(\vec{d_e},\,\vec{d_i})$, in order to find radiance as a function of the emittant direction:

$$L(\bar{p},\vec{d_e}) = \rho(\vec{d_e},\,\vec{d_i})\,I\,\frac{\vec{n}\cdot\vec{d_i}}{r^2} \tag{73}$$

This perspective generalizes naturally to multiple light sources. That is, the radiance from a point $\mathbf{p}$ on a surface in the direction of the camera is the sum of radiances due to individual light sources. For J point light sources, at locations $\mathbf{l}_j$, with intensities I_j, the radiance is given by

$$L(\bar{p},\vec{d_e}) = \sum_{j=1}^{J} \rho(\vec{d_e},\,\vec{d_j})\,I_j\,\frac{\vec{n}\cdot\vec{d_j}}{r_j^2} \tag{74}$$

where $r_j = ||\bar{l}_j - \bar{p}||$ is the distance to the j^{th} source, and $\vec{d_j} = (\bar{l}_j - \bar{p})/r_j$ is the incident direction of the j^{th} source.

Extended Light Sources

Many light sources are not infinitesimal point sources. Rather, in the general case we need to be able to work with extended light sources for which the incident light is a continuous function of incident direction. One way to think of this is to let the number of discrete light sources go to infinity so that the sum in Eqn (74) becomes an integral.

Here we take a slightly different, but equivalent approach. As discussed above, radiance can be used to express the light energy transport through any point in space, in any direction of interest. Thus, given a point $\bar{p}$ on a surface with unit normal $\vec{n}$, we can express the radiance through $\bar{p}$ along the hemisphere of possible incident directions as $L(\bar{p}, \vec{d_i})$ for $\vec{d_i} \in \Omega_i$ where Ω_i denotes the domain of plausible incident directions at $\bar{p}$.

Note:
As above, we can erect a spherical coordinate system at $\bar{p}$. Toward this end, let θ_i denote an angle measured from the surface normal, and let ϕ_i be an angle in the surface tangent plane about the normal relative to some Cartesian $x - y$ coordinate system in the plane. Then all directions

$$\vec{d_i} \equiv (\sin\theta_i \cos\phi_i,\ \sin\theta_i \sin\phi_i,\ \cos\theta_i)^T \tag{75}$$

contained in Ω_i satisfy $\theta_i \in [0, \pi/2]$ and $\phi_i \in [-\pi, \pi)$.

One problem with radiance is the fact that it expresses the light flux in terms of power per unit area on a surface perpendicular to the direction of interest. Thus, for each incident direction we are using a different plane orientation. In our case we want to express the power per unit area on our surface S, and therefore we need to rescale the radiance in direction $\vec{d_i}$ by the ratio of foreshortened surface area to surface area. One can show that this is accomplished by multiplying $L(\bar{p}, \vec{d_i})$ by $\cos\theta_i = \vec{d_i} \cdot \vec{n}$, for normal $\vec{n}$. The result is now the incident power per unit surface area (not foreshortened) per solid angle. We multiply this by solid angle $d\omega$ to obtain irradiance:

$$H = L(\bar{p}, -\vec{d_i})\ \cos\theta_i\, d\omega_i \tag{76}$$

Therefore, the resulting surface radiance in the direction of the camera due to this irradiance is just

$$\rho(\vec{d_e},\ \vec{d_i})\, L(\bar{p}, -\vec{d_i})\ \cos\theta_i\, d\omega_i$$

If we then accumulate the total radiance from the incident illumination over the entire hemisphere of possible incident directions we obtain

$$L(\vec{d_e}) = \int_{\vec{d_i} \in \Omega_i} \rho(\vec{d_e},\ \vec{d_i})\, L(\bar{p}, -\vec{d_i})\ \cos\theta_i\, d\omega_i \tag{77}$$

where, as above, the infinitesimal solid angle is $d\omega_i = \sin\theta_i \, d\theta_i \, d\phi_i$.

Light sources vary greatly from scene to scene. In effect, when you take a photograph you are measuring irradiance at the image plane of the camera for a limited field of view (angular extent). This shows how complex illumination sources can be.

Note:
The ideal point light source can also be cast in the framework of a continuous, extended source. To do this we assume that the distribution of incident light can be modeled by a scaled Dirac delta function. A Dirac delta function $\delta(x)$ is defined by:

$$\delta(x) = 0 \quad \text{for } x \neq 0 \quad , \quad \text{and} \quad \int_x \delta(x)\, f(x)\, dx = f(0) \tag{78}$$

With the light source defined as a delta function, Eqn (77) reduces to Eqn (73).

12.5 Idealized Lighting and Reflectance Models

We now consider several important special instances of BRDF models. In particular, we are interested in combinations of lighting and BRDF models that facilitate efficient shading algorithms. We discuss how diffuse and specular surfaces can be represented as BRDFs.

12.5.1 Diffuse Reflection

A diffuse (or matte) surface is one for which the pattern of shading over the surface appears the same from different viewpoints. The ideal diffusely reflecting surface is known as a perfect Lambertian surface. Its radiance is independent of the emittant direction, its BRDF is a constant, and it reflects all of the incident light (i.e., it absorbs zero power). The only factor that determines the appearance (radiance) of a Lambertian surface is therefore the irradiance (the incident light). In this case, with the BRDF constant, $\rho(\vec{d_e}, \vec{d_i}) = \rho_0$, the (constant) radiance L_e has the form:

$$L_d(\bar{p}, \vec{d_e}) = \rho_0 \int_{\vec{d_i} \in \Omega_i} L(\bar{p}, -\vec{d_i}) \, \cos\theta_i \, d\omega_i \tag{79}$$

Note:
A perfect Lambertian surface reflects all incident light, absorbing none. Therefore, the total irradiance over the hemisphere of incident directions must equal the radiant exitance. Setting these quantities to be equal, one can show that $\rho_0 = 1/\pi$. The BRDF for any diffuse surface must therefore have a value between 0 and $1/\pi$.

Despite the simplicity of the BRDF, it is not that simple to compute the radiance because we still have an integral over the hemisphere of incident directions. So let's simplify the model further.

Let's assume a single point light source with intensity I at location $\bar{l}$. This gives us

$$L_d(\bar{p}, \vec{d_e}) = \rho_0 \, I \, \frac{\vec{n} \cdot \vec{d_i}}{r^2} \tag{80}$$

where $r = ||\bar{l} - \bar{p}||$ is the distance to the light source from $\bar{p}$, and $\vec{d_i} = (\bar{l} - \bar{p})/r$ is the direction of the source from $\bar{p}$. Of course, the surface normal $\vec{n}$ also changes with $\bar{p}$.

Eqn (80) is much easier to compute, but we can actually make the computation even easier. Let's assume that the point source is sufficiently far away that r and $\vec{d_i}$ do not change much with points $\bar{p}$ on the object surface. That is, let's treat them as constant. Then we can simplify Eqn (80) to

$$L_d(\bar{p}) = r_d \, I \, \vec{s} \cdot \vec{n} \tag{81}$$

where r_d is often called the diffuse reflection coefficient, and $\vec{s}$ is the direction of the source. Then the only quantity that depends on surface position $\bar{p}$ is the surface normal $\vec{n}$.

> *Note:*
> The value $\vec{s} \cdot \vec{n}$ should actually be $\max(0, \vec{s} \cdot \vec{n})$. Why? Consider the relationship of the light source and surface when this dot product is negative.

12.5.2 Ambient Illumination

The diffuse shading model in Eqn (80) is easy to compute, but often appears artificial. The biggest issue is the point light source assumption, the most obvious consequence of which is that any surface normal pointing away from the light source (i.e., for which $\vec{s} \cdot \vec{n} < 0$) will have a radiance of zero. A better approximation to the light source is a uniform *ambient* term plus a point light source. This is a still a remarkably crude model, but it's much better than the point source by itself.

With a uniform illuminant and a constant BRDF, it is easy to see that the integral in Eqn (79) becomes a constant. That is, the radiance does not depend on the orientation of the surface because the illumination is invariant to surface orientation. As a result we can write the radiance under a uniform illuminant as

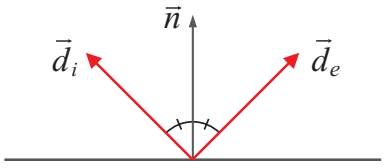
$$L_a(\bar{p}) = r_a \, I_a \tag{82}$$

where r_a is often called the ambient reflection coefficient, and I_a denotes the integral of the uniform illuminant.

> *Note:*
> If the illuminant is the sum of a point source and a uniform source, then the resulting radiance is the sum of the radiances to the individual sources, that is, the sum of Eqns (82) and (81).

12.5.3 Specular Reflection

For specular (mirror) surfaces, the incident light from each incident direction $\vec{d_i}$ is reflected toward a unique emittant direction $\vec{d_e}$. The emittant direction lies in the same plane as the incident direction $\vec{d_i}$ and the surface normal $\vec{n}$, and the angle between $\vec{n}$ and $\vec{d_e}$ is equal to that between $\vec{n}$ and $\vec{d_i}$. One



can show that the emittant direction is given by $\vec{d_e} = 2(\vec{n} \cdot \vec{d_i})\vec{n} - \vec{d_i}$. For all power from $\vec{d_i}$ be reflected into a single emittant direction the BRDF for a perfect mirror must be proportional to a delta function, $\rho(\vec{d_e}, \vec{d_i}) \propto \delta(\vec{d_i} - (2(\vec{n} \cdot \vec{d_e})\vec{n} - \vec{d_e}))$.

In particular, if we choose the constant of proportionality so that the radiant emittance is equal to the total incident power, then the BRDF becomes:

$$\rho(\vec{d_e}, \vec{d_i}) = \frac{1}{\vec{n} \cdot \vec{d_i}} \, \delta(\vec{d_i} - (2(\vec{n} \cdot \vec{d_e})\vec{n} - \vec{d_e})) \tag{83}$$

In this case, Eqn (77) reduces to

$$L_s(\bar{p}, \vec{d_e}) = L(\bar{p}, -(2(\vec{n} \cdot \vec{d_e})\vec{n} - \vec{d_e})) \tag{84}$$

This equation plays a major role in ray tracing.

Off-Axis Specularity: Many materials exhibit a significant specular component in their reflectance. But few are perfect mirrors. First, most specular surfaces do not reflect all light, and that is easily handled by introducing a scalar constant in Eqn (84) to attenuate surface radiance L_s. Second, most specular surfaces exhibit some form of *off-axis specular reflection*. That is, many polished and shiny surfaces (like plastics and metals) emit light in the perfect mirror direction and in some nearby directions as well. These off-axis specularities look a little blurred. Good examples are *highlights* on plastics and metals.

The problem with off-axis specularities is that the BRDF is no longer a simple delta function. The radiance into a particular emittant direction will now be affected from the incident power over a range of incident directions about the perfect specular direction. This means that, unlike the simple radiance function in Eqn (84) for perfect measures, we need to return to the integral in Eqn (77). Therefore it is not easy to compute radiance in this case.

Like the diffuse case above, one way to simplify the model with off-axis specularities is to assume a point light source. With a point light source we can do away with the integral. In that case the

light from a distant point source in the direction of $\vec{s}$ is reflected into a range of directions about the perfect mirror directions $\vec{m} = 2(\vec{n} \cdot \vec{s})\vec{n} - \vec{s}$. One common model for this is the following:

$$L_s(\vec{d_e}) = r_s I \max(0, \vec{m} \cdot \vec{d_e})^\alpha, \tag{85}$$

where r_s is called the specular reflection coefficient (often equal to $1 - r_d$), I is the incident power from the point source, and $\alpha \geq 0$ is a constant that determines the width of the specular highlights. As α increases, the effective width of the specular reflection decreases. In the limit as α increases, this becomes a mirror.

12.5.4 Phong Reflectance Model

The above components, taken together, give us the well-known Phong reflectance model that was introduced earlier:

$$L(\bar{p}, \vec{d_e}) \;=\; r_d\, I_d\, \max(0, \vec{s} \cdot \vec{n}) \;+\; r_a\, I_a \;+\; r_s I_s \max(0, \vec{m} \cdot \vec{d_e})^\alpha, \tag{86}$$

where

- I_a, I_d, and I_r are parameters that correspond to the power of the light sources for the ambient, diffuse, and specular terms;
- r_a, r_d and r_s are scalar constants, called reflection coefficients, that determine the relative magnitudes of the three reflection terms;
- α determines the spread of the specurlar highlights;
- $\vec{n}$ is the surface normal at $\bar{p}$;
- $\vec{s}$ is the direction of the distant point source;
- $\vec{m}$ is the perfect mirror direction, given $\vec{n}$ and $\vec{s}$; and
- and $\vec{d_e}$ is the emittant direction of interest (usually the direction of the camera).

13 Distribution Ray Tracing

In Distribution Ray Tracing (hereafter abbreviated as "DRT"), our goal is to render a scene as accurately as possible. Whereas Basic Ray Tracing computed a very crude approximation to radiance at a point, in DRT we will attempt to compute the integral as accurately as possible. Additionally, the intensity at each pixel will be properly modeled as an integral as well. Since these integrals cannot be computed exactly, we must resort to numerical integration techniques to get approximate solutions.

> *Aside:*
> When originally introduced, DRT was known as "Distributed Ray Tracing." We will avoid this name to avoid confusion with distributed computing, especially because some ray-tracers are implemented as parallel algorithms.

13.1 Problem statement

Recall that, shading at a surface point is given by:

$$L(\vec{d}_e) = \int_\Omega \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n}\cdot\vec{d}_i)\, d\omega \tag{87}$$

This equation says that the radiance emitted in direction $\vec{d}_e$ is given by integrating over the hemisphere Ω the BRDF ρ times the incoming radiance $L(-\vec{d}_i(\phi,\theta))$. Directions on the hemisphere are parameterized as

$$\vec{d}_i = (\sin\theta\sin\phi, \sin\theta\cos\phi, \cos\theta) \tag{88}$$

The differential solid angle $d\omega$ is given by:

$$d\omega = \sin\theta d\theta d\phi \tag{89}$$

and so:

$$L(\vec{d}_e) = \int_{\phi\in[0,2\pi]}\int_{\theta\in[0,\pi/2]} \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n}\cdot\vec{d}_i)\, \sin\theta d\theta d\phi \tag{90}$$

This is an integral over all incoming light directions, and we cannot compute these integrals in closed-form. Hence, we need to develop numerical techniques to compute approximations.

Intensity of a pixel. Up to now, we've been engaged in a fiction, namely, that the intensity of a pixel is the light passing through a single point on an image plane. However, real sensors — including cameras and the human eye — cannot gather light at an infinitesimal point, due both to the nature of light and the physical properties of the sensors. The actual amount of light passing through any infinitesimal region (a point) is infinitesimal (approaching zero) and cannot be measured. Instead light must be measured within a region. Specifically, the image plane (or

retina) is divided up into an array of tiny sensors, each of which measures the total light incident on the area of the sensor.

As derived previously, the image plane can be parameterized as $\bar{p}(\alpha, \beta) = \bar{p}_0 + \alpha\vec{u} + \beta\vec{v}$. In camera coordinates, $\bar{p}_0^c = (0, 0, f)$, and the axes correspond to the x and y axes: $\vec{u}^c = (1, 0, 0)$ and $\vec{v}^c = (0, 1, 0)$. Then, we placed pixel coordinates on a grid: $\bar{p}_{i,j}^c = (L + i\Delta i, T + j\Delta j, f) = \bar{p}_0 + \alpha$, where $\Delta i = (R - L)/n_c$ and $\Delta j = (B - T)/n_r$, and L, T, B, R are the boundaries of the image plane.

We will now view each pixel as an area on the screen, rather than a single point. In other words, pixel (i, j) is all values $\bar{p}(\alpha, \beta)$ for $\alpha_{min} \leq \alpha < \alpha_{max}$, $\beta_{min} \leq \beta < \beta_{max}$. The bounds of each pixel are: $\alpha_{min} = L + i\Delta i, \alpha_{max} = L + (i + 1)\Delta i, \beta_{min} = T + j\Delta j$, and $\beta_{max} = T + (j + 1)\Delta j$. (In general, we will set things up so that this rectangle is a square in world-space.) For each point on the image plane, we can write the ray passing through this pixel as

$$\vec{d}(\alpha, \beta) = \frac{\bar{p}(\alpha, \beta) - \bar{e}}{||\bar{p}(\alpha, \beta) - \bar{e}||} \tag{91}$$

To compute the color of a pixel, we should compute the total light energy passing through this rectangle, i.e., the flux at that pixel:

$$\Phi_{i,j} = \int_{\alpha_{min} \leq \alpha < \alpha_{max}} \int_{\beta_{min} \leq \beta < \beta_{max}} H(\alpha, \beta) d\alpha d\beta \tag{92}$$

where $H(\alpha, \beta)$ is the incoming light (irradiance) on the image at position α, β. For color images, this integration is computed for each color channel. Again, we cannot compute this integral exactly.

> *Aside:*
> An even more accurate model of a pixel intensity is to weight rays according to how close they are to the center of the pixel, using a Gaussian weighting function.

13.2 Numerical integration

We begin by considering the general problem of computing an integral in 1D. Suppose we wish to integrate a function $f(x)$ from 0 to D:

$$S = \int_0^D f(x) dx \tag{93}$$

Visually, this corresponds to computing the area under a curve. Recall the definition of the integral. We can break the real line into a set of intervals centered at uniformly-spaced points $x_1, ..., x_N$. We can then define one rectangle on each interval, each width D/N and height $f(x_i)$. The total area

of these rectangles will be approximately the same as the area under the curve. The area of each rectangle is $f(x_i)D/N$, and thus the total area of all rectangles together is:

$$S_N = \frac{D}{N} \sum_i f(x_i) \tag{94}$$

Hence, we can use S_N as an approximation to S. Moreover, we will get more accuracy as we increase the number of points:

$$\lim_{N \to \infty} S_N = S \tag{95}$$

There are two problems with using uniformly-spaced samples for numerical integration:

- Some parts of the function may be much more "important" than others. For example, we don't want to have to evaluate $f(x)$ in areas where it is almost zero. Hence, you need to generate many, many x_i values, which can be extremely slow.
- Uniformly-spaced samples can lead to *aliasing artifacts*. These are especially noticable when the scene or textures contain repeated (periodic) patterns.

In ray-tracing, each evaluation of $f(x)$ requires performing a ray-casting operation and a recursive call to the shading procedure, and is thus very, very expensive. Hence, we would like to design integration procedures that use as few evaluations of $f(x)$ as possible.

To address these problems, randomized techniques known as **Monte Carlo integration** can be used.

13.3 Simple Monte Carlo integration

Simple Monte Carlo addresses the problem of aliasing, and works as follows. We randomly sample N values x_i in the interval $[0, D]$, and then evaluate the same sum just as before:

$$S_N = \frac{D}{N} \sum_i f(x_i) \tag{96}$$

It turns out that, if we have enough samples, we will get just as accurate a result as before; moreover, aliasing problems will be reduced.

> *Aside:*
> Formally, it can be shown that the expected value of S_N is S. Moreover, the variance of S_N is proportional to N, i.e., more samples leads to better estimates of the integral.

In the C programming language, the random sampling can be computed as `rand() * D`.

> *Aside:*
> Monte Carlo is a city near France and Italy famous for a big casino. Hence, the name of the Monte Carlo algorithm, since you randomly sample some points and gamble that they are representative of the function.

13.4 Integration at a pixel

To compute the intensity of an individual pixel, we need to evaluate Equation 92). This is a 2D integral, so we need to determine K 2D points (α_i, β_i), and compute:

$$\Phi_{i,j} \approx \frac{(\alpha_{max} - \alpha_{min})(\beta_{max} - \beta_{min})}{K} \sum_{i=1}^{K} H(\alpha_i, \beta_i) \tag{97}$$

In other words, we pick N points withnin the pixel, cast a ray through each point, and then average the intensities of the rays (scaled by the pixel's area $(\alpha_{max} - \alpha_{min})(\beta_{max} - \beta_{min})$. These samples can be chosen randomly, or uniformly-spaced.

Example:
The simplest way to compute this is by uniformly-spaced samples (α_m, β_n):

$$\alpha_m = (m-1)\Delta\alpha, \quad \Delta\alpha = (\alpha_{max} - \alpha_{min})/M \tag{98}$$
$$\beta_n = (n-1)\Delta\beta, \quad \Delta\beta = (\beta_{max} - \beta_{min})/N \tag{99}$$

and then sum:

$$\Phi_{i,j} \approx \Delta\alpha\Delta\beta \sum_{m=1}^{M}\sum_{n=1}^{N} H(\alpha_m, \beta_n) \tag{100}$$

However, Monte Carlo sampling — in which the samples are randomly-spaced — will usually give better results.

13.5 Shading integration

Our goal in shading a point is to compute the integral:

$$L(\vec{d}_e) = \int_{\phi\in[0,2\pi]} \int_{\theta\in[0,\pi/2]} \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n}\cdot\vec{d}_i)\, \sin\theta\, d\theta d\phi \tag{101}$$

We can choose uniformly-spaced values of ϕ and θ values as follows:

$$\theta_m = (m-1)\Delta\theta, \quad \Delta\theta = (\pi/2)/M \tag{102}$$
$$\phi_n = (n-1)\Delta\phi, \quad \Delta\phi = 2\pi/N \tag{103}$$

This divides up the unit hemisphere into MN solid angles, each with area approximately equal to $\sin\theta\Delta\theta\Delta\phi$. Applying 2D numerical integration gives:

$$L(\vec{d}_e) \approx \sum_{m=1}^{M}\sum_{n=1}^{N} \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n}\cdot\vec{d}_i)\, \sin\theta\, \Delta\theta\, \Delta\phi \tag{104}$$

Once you have all the elements in place (e.g., the ray-tracer, the BRDF model, etc.), evaluating this equation is actually quite simple, and doesn't require all the treatment of special cases required for basic ray-tracing (such as specular, diffuse, mirror, etc.). However, it is potentially much slower to compute.

13.6 Stratified Sampling

A problem with Simple Monte Carlo is that, if you use a small number of samples, these samples will be spaced very irregularly. For example, you might be very unlucky and get samples that don't place any samples in some parts of the space. This can be addressed by a technique called stratified sampling: divide the domain into K-uniformly sized regions, and randomly sample J points x_i within each region; then sum $\frac{D}{N}\sum_i f(x_i)$ as before.

13.7 Non-uniformly spaced points

Quite often, most of the radiance will come from a small part of the integral. For example, if the scene is lit by a bright point light source, then most of the energy comes from the direction to this source. If the surface is very shiny and not very diffuse, then most of the energy comes from the reflected direction. In general, it is desirable to sample more densely in regions where the function changes faster and where the function values are large. The general equation for this is:

$$S_N = \sum_i f(x_i) d_i \tag{105}$$

where d_i is the size of the region around point x_i. Alternatively, we can use stratified sampling, and randomly sample J values within each region. How we choose to define the region sizes and spaces depends on the specific integration problem. Doing so can be very difficult, and, as a consequence, deterministic non-uniform spacing is normally used in graphics; instead, importance sampling (below) is used instead.

13.8 Importance sampling

The method of **importance sampling** is a more sophisticated form of Monte Carlo that allows non-uniform sample spacing. Instead of sampling the points x_i uniformly, we sample them from another probability distribution function (PDF) $p(x)$. We need to design this PDF so that it gives us more samples in regions of x that are more "important," e.g., values of $f(x)$ are larger. We can then approximate the integral S as:

$$S_N = \frac{1}{N}\sum_i \frac{f(x_i)}{p(x_i)} \tag{106}$$

If we use a uniform distribution: $p(x) = 1/D$ for $x \in [0, D]$, then it is easy to see that this procedure reduces to Simple Monte Carlo. However, we can also use something more sophisticated, such as a Gaussian distribution centered around the point we expect to provide the greatest contribution to the intensity.

13.9 Distribution Ray Tracer

for each pixel (i,j)
  $<$ choose N points $\bar{x}_k$ within the pixel's area $>$
  for each sample k
    $<$ compute ray $\vec{\mathbf{r}}_k(\lambda) = \vec{\mathbf{p}}_k + \lambda\vec{\mathbf{d}}_k$ where $\vec{\mathbf{d}}_k = \vec{\mathbf{p}}_k - \vec{\mathbf{e}}$ $>$
    I_k = rayTrace($\vec{\mathbf{p}}_k$, $\vec{\mathbf{d}}_k$, 1)
  end **for**
  setpixel(i, j, $\Delta i \Delta j \sum_k I_k / N$)
end **for**

The rayTrace and findFirstHit procedures are the same as for Basic Ray Tracing. However, the new shading procedure uses numerical integration:

distRtShade(OBJ, $\vec{\mathbf{p}}$, $\vec{\mathbf{n}}$, $\vec{\mathbf{d}}_e$, depth)
  $<$ choose N directions (ϕ_k, θ_k) on the hemisphere $>$
  for each direction k
    I_k = rayTrace($\vec{\mathbf{p}}$, $\vec{\mathbf{d}}_k$, depth+1)
  end **for**
  return $\Delta\theta\Delta\phi \sum_k \rho(\vec{\mathbf{d}}_e, \vec{d}_i(\phi_k, \theta_k)) I_k \sin\theta_k$

14 Interpolation

14.1 Interpolation Basics

Goal: We would like to be able to define curves in a way that meets the following criteria:

1. Interaction should be natural and intuitive.
2. Smoothness should be controllable.
3. Analytic derivatives should exist and be easy to compute.
4. Representation should be compact.

Interpolation is when a curve passes through a set of "control points."

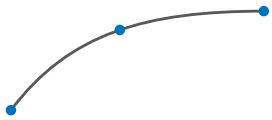


Figure 9: *
Interpolation

Approximation is when a curve approximates but doesn't necessarily contain its control points.

Figure 10: *
Approximation

Extrapolation is extending a curve beyond the domain of its control points.

Continuity - A curve is is C^n when it is continuous in up to its n^{th}-order derivatives. For example, a curve is in C^1 if it is continuous and its first derivative is also continuous.

Consider a cubic interpolant — a 2D curve, $\bar{c}(t) = \begin{bmatrix} x(t) & y(t) \end{bmatrix}$ where

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3, \quad (107)$$
$$y(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3, \quad (108)$$

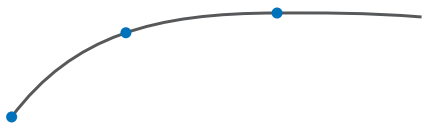


Figure 11: *
Extrapolation

so

$$x(t) = \sum_{i=0}^{3} a_i t^i = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \vec{t}^T \vec{a}. \tag{109}$$

Here, $\vec{t}$ is the basis and $\vec{a}$ is the coefficient vector. Hence, $\bar{c}(t) = \vec{t}^T \begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix}$. (Note: $T\begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix}$ is a 4×2 matrix).

There are eight unknowns, four a_i values and four b_i values. The constraints are the values of $\bar{c}(t)$ at known values of t.

Example:
For $t \in (0, 1)$, suppose we know $\bar{c}_j \equiv \bar{c}(t_j)$ for $t_j = 0, \frac{1}{3}, \frac{2}{3}, 1$ as $j = 1, 2, 3, 4$. That is,

$$\bar{c}_1 = \begin{bmatrix} x_1 & y_1 \end{bmatrix} \equiv \begin{bmatrix} x(0) & y(0) \end{bmatrix}, \tag{110}$$
$$\bar{c}_2 = \begin{bmatrix} x_2 & y_2 \end{bmatrix} \equiv \begin{bmatrix} x(1/3) & y(1/3) \end{bmatrix}, \tag{111}$$
$$\bar{c}_3 = \begin{bmatrix} x_3 & y_3 \end{bmatrix} \equiv \begin{bmatrix} x(2/3) & y(2/3) \end{bmatrix}, \tag{112}$$
$$\bar{c}_4 = \begin{bmatrix} x_4 & y_4 \end{bmatrix} \equiv \begin{bmatrix} x(1) & y(1) \end{bmatrix}. \tag{113}$$

So we have the following linear system,

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1/3 & (1/3)^2 & (1/3)^3 \\ 1 & 2/3 & (2/3)^2 & (2/3)^3 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix}, \tag{114}$$

or more compactly, $\begin{bmatrix} \vec{x} & \vec{y} \end{bmatrix} = C \begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix}$. Then, $\begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix} = C^{-1} \begin{bmatrix} \vec{x} & \vec{y} \end{bmatrix}$. From this we can find $\vec{a}$ and $\vec{b}$, to calculate the cubic curve that passes through the given points.

We can also place derivative constraints on interpolant curves. Let

$$\vec{\tau}(t) = \frac{d\bar{c}(t)}{dt} = \frac{d}{dt}\begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix}\begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix} \quad (115)$$

$$= \begin{bmatrix} 0 & 1 & t & t^2 \end{bmatrix}\begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix}, \quad (116)$$

that is, a different basis with the same coefficients.

Example:
Suppose we are given three points, $t_j = 0, \frac{1}{2}, 1$, and the derivative at a point, $\vec{\tau}_2(\frac{1}{2})$. So we can write this as

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_2' & y_2' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1/2 & (1/2)^2 & (1/2)^3 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2(1/2) & 3(1/2)^2 \end{bmatrix}\begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix}, \quad (117)$$

and

$$\begin{bmatrix} \bar{c}_1 \\ \bar{c}_2 \\ \bar{c}_3 \\ \vec{\tau}_2 \end{bmatrix} = C\begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix}, \quad (118)$$

which we can use to find $\vec{a}$ and $\vec{b}$:

$$\begin{bmatrix} \vec{a} & \vec{b} \end{bmatrix} = C^{-1}\begin{bmatrix} \bar{c}_1 \\ \bar{c}_2 \\ \bar{c}_3 \\ \vec{\tau}_2 \end{bmatrix}. \quad (119)$$

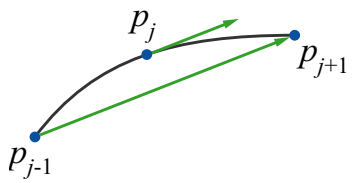
Unfortunately, polynomial interpolation yields unintuitive results when interpolating large numbers of control points; you can easily get curves that pass through the control points, but oscillate in very unexpected ways. Hence, direct polynomial interpolation is rarely used except in combination with other techniques.

14.2 Catmull-Rom Splines

Catmull-Rom Splines interpolate degree-3 curves with C^1 continuity and are made up of cubic curves.

A user specifies only the points $[\bar{p}_1, ...\bar{p}_N]$ for interpolation, and the tangent at each point is set to be parallel to the vector between adjacent points. So the tangent at $\bar{p}_j$ is $\kappa(\bar{p}_{j+1} - \bar{p}_{j-1})$ (for

endpoints, the tangent is instead parallel to the vector from the endpoint to its only neighbor). The value of κ is set by the user, determining the "tension" of the curve.



Between two points, $\bar{p}_j$ and $\bar{p}_{j+1}$, we draw a cubic curve using $\bar{p}_j$, $\bar{p}_{j+1}$, and two auxiliary points on the tangents, $\kappa(\bar{p}_{j+1} - \bar{p}_{j-1})$ and $\kappa(\bar{p}_{j+2} - \bar{p}_j)$.

We want to find the coefficients a_j when $x(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \end{bmatrix}^T$, where the curve is defined as $\bar{c}(t) = \begin{bmatrix} c(t) & y(t) \end{bmatrix}$ (similarly for $y(t)$ and b_j). For the curve between $\bar{p}_j$ and $\bar{p}_{j+1}$, assume we know two end points, $\bar{c}(0)$ and $\bar{c}(1)$ and their tangents, $\bar{c}'(0)$ and $\bar{c}'(1)$. That is,

$$\begin{aligned} x(0) &= x_j, && (120)\\ x(1) &= x_{j+1}, && (121)\\ x'(0) &= \kappa(x_{j+1} - x_{j-1}), && (122)\\ x'(1) &= \kappa(x_{j+2} - x_j). && (123) \end{aligned}$$

To solve for $\vec{a}$, set up the linear system,

$$\begin{bmatrix} x(0) \\ x(1) \\ x'(0) \\ x'(1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}. \quad (124)$$

Then $\vec{x} = M\vec{a}$, so $\vec{a} = M^{-1}\vec{x}$. Substituting $\vec{a}$ in $x(t)$ yields

$$\begin{aligned} x(t) &= \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_j \\ x_{j+1} \\ \kappa(x_{j+1} - x_{j-1}) \\ \kappa(x_{j+2} - x_j) \end{bmatrix} && (125)\\ &= \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\kappa & 0 & \kappa & 0 \\ 2\kappa & \kappa - 3 & 3 - 2\kappa & -\kappa \\ -\kappa & 2 - \kappa & \kappa - 2 & \kappa \end{bmatrix} \begin{bmatrix} x_{j-1} \\ x_j \\ x_{j+1} \\ x_{j+2} \end{bmatrix}. && (126) \end{aligned}$$

For the first tangent in the curve, we cannot use the above formula. Instead, we use:

$$\vec{\tau}_1 = \kappa(\bar{p}_2 - \bar{p}_1) \tag{127}$$

and, for the last tangent:

$$\vec{\tau}_N = \kappa(\bar{p}_N - \bar{p}_{N-1}) \tag{128}$$

15 Parametric Curves And Surfaces

15.1 Parametric Curves

Designing Curves

- We don't want only polygons.
- Curves are used for design. Users require a simple set of controls to allow them to edit and design curves easily.
- Curves should have infinite resolution, so we can zoom in and still see a smooth curve.
- We want to have a compact representation.

Parametric functions are of the form $x(t) = f(t)$ and $y(t) = g(t)$ in two dimensions. This can be extended for arbitrary dimensions. They can be used to model curves that are *not* functions of any axis in the plane.

Curves can be defined as polynomials, for example $x(t) = 5t^{10} + 4t^9 + 3t^8 + ...$ However, coefficients are not intuitive editing parameters, and these curves are difficult to control. Hence, we will consider more intuitive parameterizations.

15.2 Bézier curves

We can define a set of curves called Bézier curves by a procedure called the de Casteljau algorithm. Given a sequence of control points $\bar{p}_k$, de Casteljau evaluation provides a construction of smooth parametric curves. Evaluation proceeds by repeatedly defining new, smaller point sequences until we have a single point at the value for t for which we are evaluating the curve.

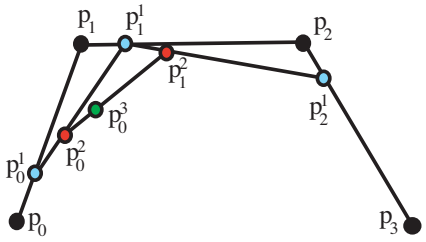


Figure 12: de Casteljau evaluation for $t = 0.25$.

$$\bar{p}_0^1(t) \quad = \quad (1-t)\bar{p}_0 + t\bar{p}_1 \tag{129}$$

$$\begin{aligned}
\bar{p}_1^1(t) &= (1-t)\bar{p}_1 + t\bar{p}_2 && (130)\\
\bar{p}_2^1(t) &= (1-t)\bar{p}_2 + t\bar{p}_3 && (131)
\end{aligned}$$

$$\begin{aligned}
\bar{p}_0^2(t) &= (1-t)\bar{p}_0^1(t) + t\bar{p}_1^1(t) && (132)\\
&= (1-t)^2\bar{p}_0 + 2t(1-t)\bar{p}_1 + t^2\bar{p}_2 && (133)\\
\bar{p}_1^2(t) &= (1-t)\bar{p}_1^1(t) + t\bar{p}_2^1(t) && (134)\\
&= (1-t)^2\bar{p}_1 + 2t(1-t)\bar{p}_2 + t^2\bar{p}_3 && (135)
\end{aligned}$$

$$\begin{aligned}
\bar{p}_0^3(t) &= (1-t)\bar{p}_0^2(t) + t\bar{p}_1^2(t) && (136)\\
&= (1-t)^3\bar{p}_0 + 3(1-t)^2t\bar{p}_1 + 3(1-t)t^2\bar{p}_2 + t^3\bar{p}_3 && (137)
\end{aligned}$$

The resulting curve $\bar{p}_0^3$ is the cubic Bézier defined by the four control points. The curves $\bar{p}_0^2$ and $\bar{p}_1^2$ are quadratic Bézier curves, each defined by three control points. For all Bézier curves, we keep t in the range $[0...1]$.

15.3 Control Point Coefficients

Given a sequence of points $\bar{p}_0$, $\bar{p}_1$, ..., $\bar{p}_n$, we can directly evaluate the coefficient of each point. For a class of curves known as Bézier curves, the coefficients are defined by the Bernstein polynomials:

$$\bar{p}_0^n(t) = \sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) (1-t)^{n-i} t^i \bar{p}_i = \sum_{i=0}^{n} B_i^n(t)\bar{p}_i \qquad (138)$$

where

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i \qquad (139)$$

are called the *Bernstein basis functions.*

For example, cubic Bézier curves have the following coefficients:

$$\begin{aligned}
B_0^3(t) &= (1-t)^3 && (140)\\
B_1^3(t) &= 3(1-t)^2t && (141)\\
B_2^3(t) &= 3(1-t)t^2 && (142)\\
B_3^3(t) &= t^3 && (143)
\end{aligned}$$

Figure 13 is an illustration of the cubic Bernstein basis functions.

Similarly, we define basis functions for a linear curve, which is equivalent to the interpolation $\bar{p}(t) = \bar{p}_0(1-t) + \bar{p}_1 t$. These are shown in Figure 3.

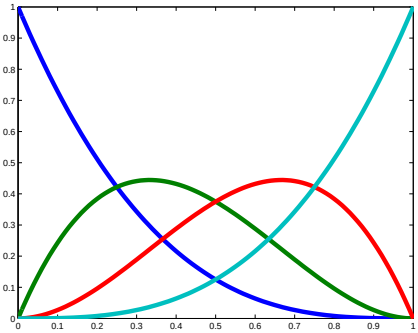


Figure 13: Degree three basis functions for Bézier curves. $B_0^3(t)$ (dark blue), $B_1^3(t)$ (green), $B_2^3(t)$ (red), and $B_3^3(t)$ (light blue).

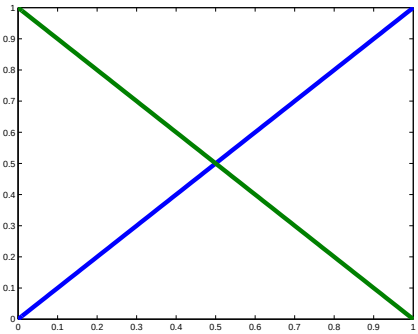


Figure 14: Degree one basis functions for Bézier curves. $B_0^1(t)$ (green) and $B_1^1(t)$ (blue).

15.4 Bézier Curve Properties

- **Convexity of the basis functions.** For all values of $t \in [0...1]$, the basis functions sum to 1:

$$\sum_{i=0}^{n} B_i^n(t) = 1 \tag{144}$$

In the cubic case, this can be shown as follows:

$$((1-t)+t)^3 = (1-t)^3 + 3(1-t)^2 t + 3(1-t)t^2 + t^3 = 1 \tag{145}$$

In the general case, we have:

$$((1-t)+t)^n = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i = 1 \tag{146}$$

Similarly, it is easy to show that the basis functions are always non-negative: $B_i^n(t) \geq 0$.

- **Affine Invariance**

  What happens if we apply an affine transformation to a Bézier curve?

  Let $\bar{c}(t) = \sum_{j=0}^{n} \bar{p}_j B_j^n(t)$, and let $F(\bar{p}) = \mathbf{A}\bar{p} + \vec{d}$ be an affine transformation. Then we have the following:

$$\begin{aligned} F(\bar{c}(t)) &= \mathbf{A}\bar{c}(t) + \vec{d} && (147)\\ &= \mathbf{A}\left(\sum \bar{p}_j B_j^n(t)\right) + \vec{d} && (148)\\ &= \sum \left(\mathbf{A}\bar{p}_j\right) B_j^n(t) + \vec{d} && (149)\\ &= \sum \left(\mathbf{A}\bar{p}_j + \vec{d}\right) B_j^n(t) && (150)\\ &= \sum B_j^n(t)\bar{q}_j && (151) \end{aligned}$$

  $\bar{q}_j = \mathbf{A}\bar{p}_j + \vec{d}$ denotes the transformed points. This illustrates that the transformed curve we get is the same as what we get by transforming the control points. (The third statement follows from the fact that $\sum_{j=0}^{n} B_j^n(t) = 1$.)

- **Convex Hull Property**

  Since $B_i^N(t) \geq 0$, $\bar{p}(t)$ is a convex combination of the control points. Thus, Bézier curves *always* lie within the convex hull of the control points.

- **Linear Precision**

  When the control points lie on a straight line, then the corresponding Bézier curve will also be a straight line. This follows from the convex hull property.

- **Variation Diminishing**

  No straight line can have more intersections with the Bézier curve than it has with the control polygon. (The control polygon is defined as the line segments $\overline{p_j p_{j+1}}$.)

- **Derivative Evaluation**

  Letting $\bar{c}(t) = \sum_{j=0}^{N} \bar{p}_j B_j^N(t)$, we want to find the following:

$$\bar{c}'(t) = \frac{d\bar{c}(t)}{dt} = \left(\frac{dx(t)}{dt}, \frac{dy(t)}{dt}\right) \quad (152)$$

  Letting $\vec{d_j} = \bar{p}_{j+1} - \bar{p}_j$, it can be shown that:

$$\tau(t) = \frac{d}{dt}\bar{c}(t) = \frac{d}{dt}\sum_{j=0}^{N} \bar{p}_j B_j^N(t) = N\sum_{j=0}^{N-1} \vec{d} B_j^{N-1}(t) \quad (153)$$

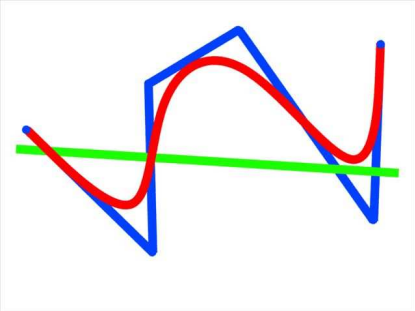


Figure 15: The line (green) will always intersect the curve less often than or as many times as the control polygon.

Thus, $\bar{c}(t)$ is a convex sum of the points $\bar{p}_j$ and is a point itself. $\tau(t)$ is a convex sum of vectors and is a vector.

Example: What is $\tau(0)$ when $N = 3$, given $(\bar{p}_0, \bar{p}_1, \bar{p}_2, \bar{p}_3)$?

Since $B_j^3(0) = 0$ for all $j \neq 0$ and $B_0^3(0) = 1$,

$$\tau(0) = N \sum \vec{d}_j B_j^{N-1}(t) = 3\vec{d}_j = 3\,(\bar{p}_1 - \bar{p}_0) \tag{154}$$

Therefore, the tangent vector at the endpoint is parallel to the vector from the endpoint to the adjacent point.

- **Global vs. Local Control**

  Bézier curves that approximate a long sequence of points produce high-degree polynomials. They have global basis functions; that is, modifying any point changes the entire curve. This results in curves that can be hard to control.

15.5 Rendering Parametric Curves

Given a parameter range $t \in [0, 1]$, sample t by some partition Δt, and draw a line connecting each pair of adjacent samples.

- This is an expensive algorithm.
- This does not adapt to regions of a curve that do not require as many samples.
- It's difficult to determine a sufficient number of samples to render the curve such that it appears smooth.

There are faster algorithms based on adaptive refinement and subdivision.

15.6 Bézier Surfaces

Cubic Bézier patches are the most common parametric surfaces used for modeling. They are of the following form:

$$\mathbf{s}(\alpha,\beta) = \sum_{k=0}^{3}\sum_{j=0}^{3} B_j^3(\alpha)B_k^3(\beta)\bar{p}_{j,k} = \sum_k B_k^3(\beta)\bar{p}_k(\alpha) \tag{155}$$

where each $\bar{p}_k(\alpha)$ is a Bézier curve:

$$\bar{p}_k(\alpha) = \sum_j B_j^3(\alpha)\bar{p}_{j,k} \tag{156}$$

Rather than considering only four points as in a cubic Bézier curve, consider 16 control points arranged as a 4 x 4 grid:

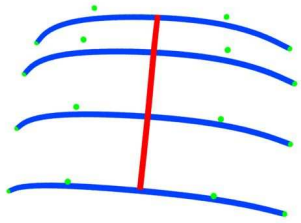


Figure 16: Evaluation of any point can be done by evaluating curves along one direction (blue), and evaluating a curve among points on these curves with corresponding parameter values.

For any given α, generate four points on curves and then approximate them with a Bézier curve along β.

$$\bar{p}_k(\alpha) = \sum_{j=0}^{3} B_j^3(\alpha)\bar{p}_{jk} \tag{157}$$

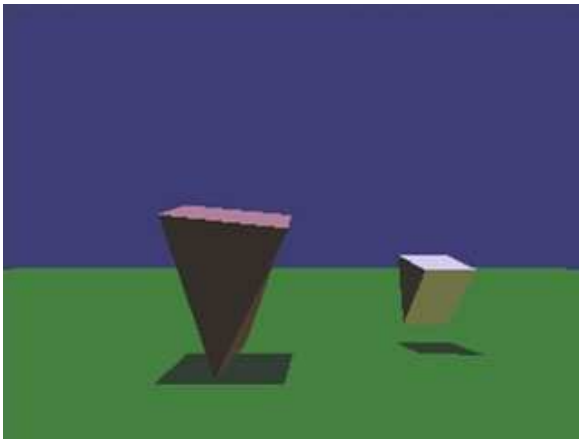
To connect multiple patches, we align adjacent control points. to ensure C^1 continuity, we also have to enforce colinearity of the neighboring points.
The surface can also be written in terms of 2D basis functions $B_{j,k}^3(\alpha,\beta) = B_j^3(\alpha)B_k^3(\beta)$:

$$\mathbf{s}(\alpha,\beta) = \sum_{k=0}^{3}\sum_{j=0}^{3} B_{j,k}^3(\alpha,\beta)\bar{p}_{j,k} \tag{158}$$

16 Animation

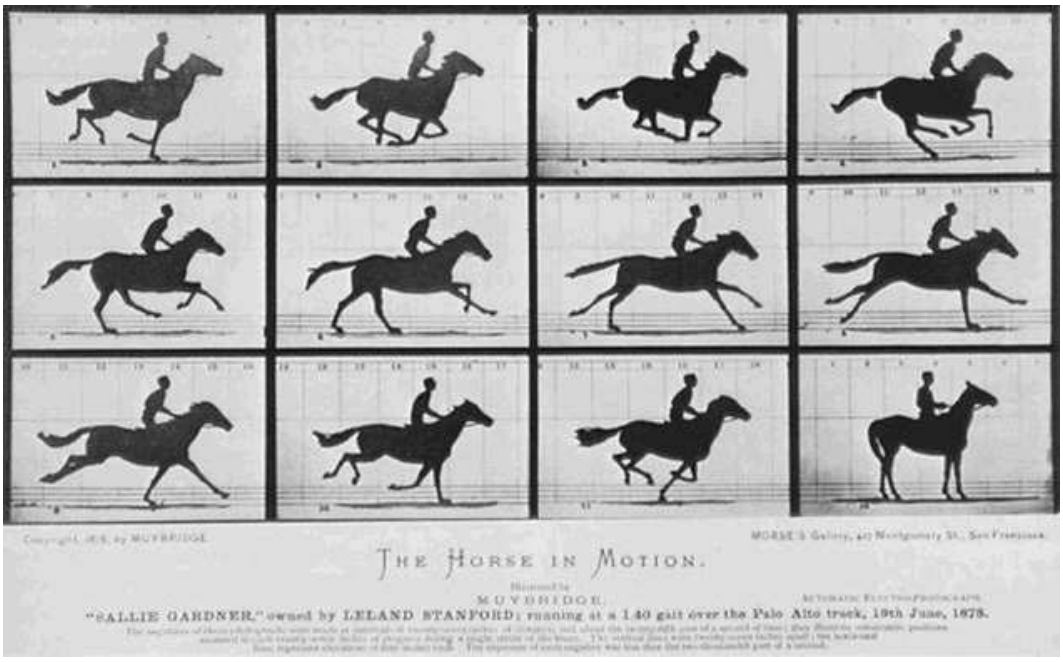
16.1 Overview

Motion can bring the simplest of characters to life. Even simple polygonal shapes can convey a number of human qualities when animated: identity, character, gender, mood, intention, emotion, and so on.



Very simple characters (image by Ken Perlin)

A movie is a sequence of frames of still images. For video, the frame rate is typically 24 frames per second. For film, this is 30 frames per second.



In general, animation may be achieved by specifying a model with n parameters that identify degrees of freedom that an animator may be interested in such as

- polygon vertices,
- spline control,
- joint angles,
- muscle contraction,
- camera parameters, or
- color.

With n parameters, this results in a vector $\vec{q}$ in n-dimensional state space. Parameters may be varied to generate animation. A model's motion is a trajectory through its state space or a set of motion curves for each parameter over time, i.e. $\vec{q}(t)$, where t is the time of the current frame. Every animation technique reduces to specifying the state space trajectory.

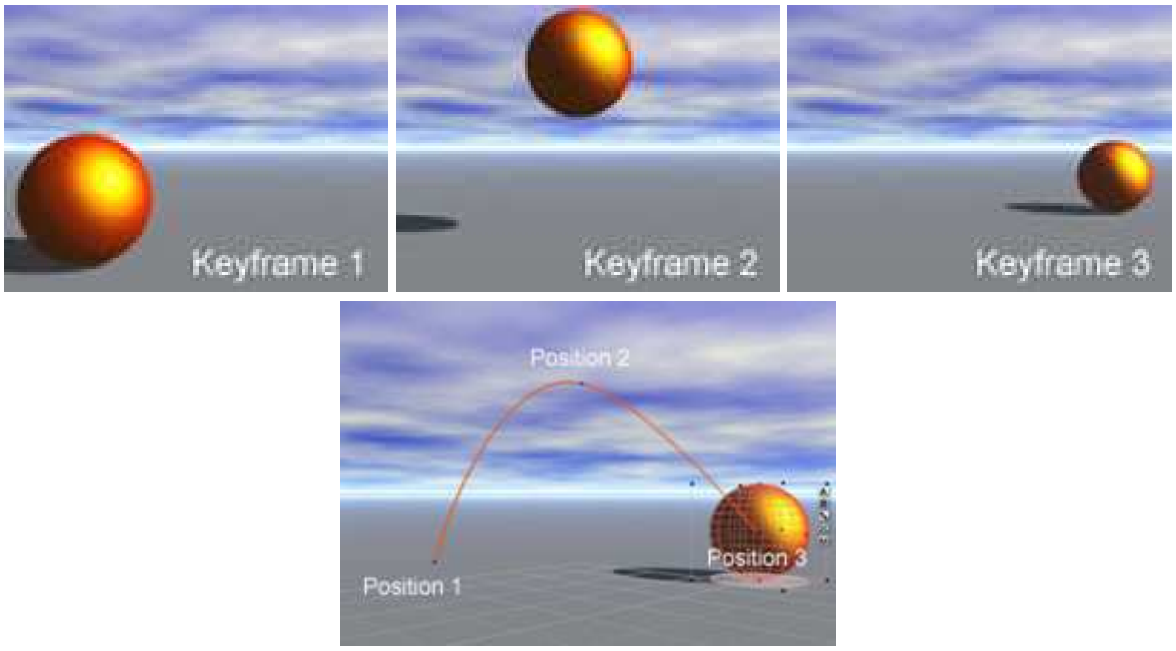
The basic animation algorithm is then: `for t=`$\mathtt{t}_1$ `to` $\mathtt{t}_{\mathtt{end}}$`: render(`$\vec{\mathtt{q}}$`(t))`.

Modeling and animation are loosely coupled. Modeling describes control values and their actions. Animation describes how to vary the control values. There are a number of animation techniques, including the following:

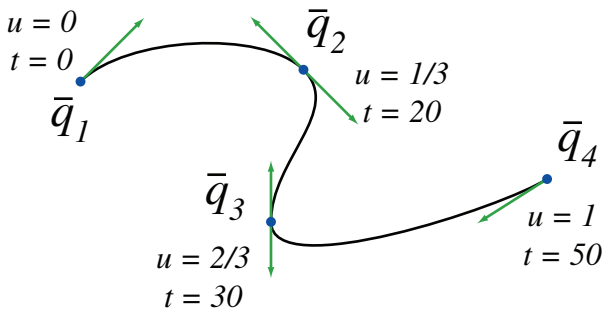
- User driven animation
  - Keyframing
  - Motion capture
- Procedural animation
  - Physical simulation
  - Particle systems
  - Crowd behaviors
- Data-driven animation

16.2 Keyframing

Keyframing is an animation technique where motion curves are interpolated through states at times, $(\vec{q}_1, ..., \vec{q}_T)$, called keyframes, specified by a user.



Catmull-Rom splines are well suited for keyframe animation because they pass through their control points.



- Pros:
  - Very expressive
  - Animator has complete control over all motion parameters
- Cons:

  - Very labor intensive
  - Difficult to create convincing physical realism

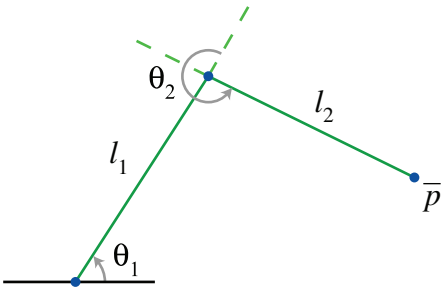
- Uses:
  - Potentially everything except complex physical phenomena such as smoke, water, or fire

16.3 Kinematics

Kinematics describe the properties of shape and motion independent of physical forces that cause motion. Kinematic techniques are used often in keyframing, with an animator either setting joint parameters explicitly with **forward kinematics** or specifying a few key joint orientations and having the rest computed automatically with **inverse kinematics**.

16.3.1 Forward Kinematics

With forward kinematics, a point $\bar{p}$ is positioned by $\bar{p} = f(\Theta)$ where Θ is a state vector $(\theta_1, \theta_2, ...\theta_n)$ specifying the position, orientation, and rotation of all joints.



For the above example, $\bar{p} = (l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2), l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2))$.

16.3.2 Inverse Kinematics

With inverse kinematics, a user specifies the position of the end effector, $\bar{p}$, and the algorithm has to evaluate the required Θ give $\bar{p}$. That is, $\Theta = f^{-1}(\bar{p})$.

Usually, numerical methods are used to solve this problem, as it is often nonlinear and either underdetermined or overdetermined. A system is underdetermined when there is not a unique solution, such as when there are more equations than unknowns. A system is overdetermined when it is inconsistent and has no solutions.

Extra constraints are necessary to obtain unique and stable solutions. For example, constraints may be placed on the range of joint motion and the solution may be required to minimize the kinetic energy of the system.

16.4 Motion Capture

In motion capture, an actor has a number of small, round markers attached to his or her body that reflect light in frequency ranges that motion capture cameras are specifically designed to pick up.

(image from movement.nyu.edu)

With enough cameras, it is possible to reconstruct the position of the markers accurately in 3D. In practice, this is a laborious process. Markers tend to be hidden from cameras and 3D reconstructions fail, requiring a user to manually fix such drop outs. The resulting motion curves are often noisy, requiring yet more effort to clean up the motion data to more accurately match what an animator wants.

Despite the labor involved, motion capture has become a popular technique in the movie and game industries, as it allows fairly accurate animations to be created from the motion of actors. However, this is limited by the density of markers that can be placed on a single actor. Faces, for example, are still very difficult to convincingly reconstruct.

- Pros:
  - Captures specific style of real actors
- Cons:
  - Often not expressive enough
  - Time consuming and expensive
  - Difficult to edit
- Uses:
  - Character animation
  - Medicine, such as kinesiology and biomechanics

16.5 Physically-Based Animation

It is possible to simulate the physics of the natural world to generate realistic motions, interactions, and deformations. **Dynamics** rely on the time evolution of a physical system in response to forces.

Newton's second law of motion states $f = ma$, where f is force, m is mass, and a is acceleration. If $x(t)$ is the path of an object or point mass, then $v(t) = \frac{dx(t)}{dt}$ is velocity and $a(t) = \frac{dv(t)}{dt} = \frac{d^2x(t)}{dt^2}$ is acceleration. Forces and mass combine to determine acceleration, i.e. any change in motion.

In **forward simulation** or **forward dynamics**, we specify the initial values for position and velocity, $x(0)$ and $v(0)$, and the forces. Then we compute $a(t)$, $v(t)$, $x(t)$ where $a(t) = \frac{f(t)}{m}$, $v(t) = \int_0^t a(t)dt + v(0)$, and $x(t) = \int_0^t v(t)dt + x(0)$.

Forward simulation has the advantage of being reasonably easy to simulate. However, a simulation is often very sensitive to initial conditions, and it is often difficult to predict paths $x(t)$ without running a simulation—in other words, control is hard.

With **inverse dynamics**, constraints on a path $x(t)$ are specified. Then we attempt to solve for the forces required to produce the desired path. This technique can be very difficult computationally.

Physically-based animation has the advantages of:

- Realism,
- Long simulations are easy to create,
- Natural secondary effects such as wiggles, bending, and so on—materials behave naturally,

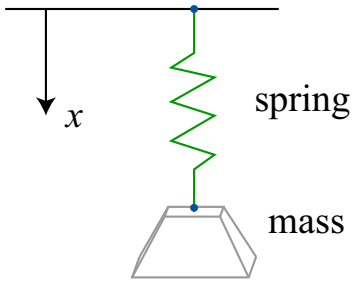
- Interactions between objects are also natural.

The main disadvantage of physically-based animation is the lack of control, which can be critical, for example, when a complicated series of events needs to be modeled or when an artist needs precise control over elements in a scene.

- Pros:
  - Very realistic motion
- Cons:
  - Very slow
  - Very difficult to control
  - Not expressive
- Uses:
  - Complex physical phenomena

16.5.1 Single 1D Spring-Mass System

Spring-mass systems are widely used to model basic physical systems. In a 1D spring, $x(t)$ represents the position of mass, increasing downwards.



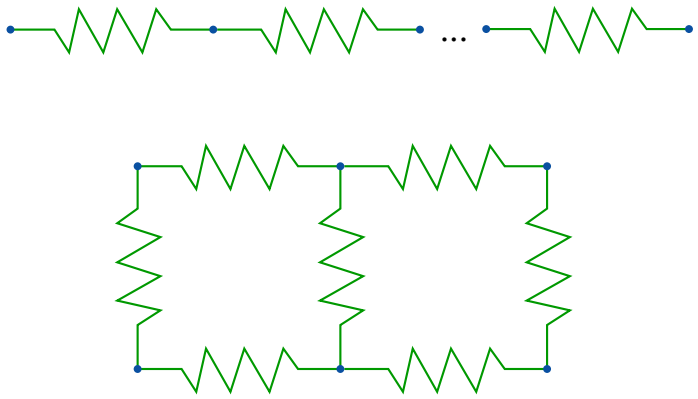
A spring has resting length l and stiffness k. Deformation force is linear in the difference from the resting length. Hence, a spring's internal force, according to Hooke's Law, is $f^s(t) = k(l - x(t))$.

The external forces acting on a spring include gravity and the friction of the medium. That is, $f^g = mg$ and $f^d(t) = -\rho v(t) = -\rho\frac{dx(t)}{dt}$, where ρ is the damping constant.

Hence, the total force acting on a spring is $f(t) = f^s(t) + f^g + f^d(t)$. Then we may use $a(t) = \frac{f(t)}{m}$ with initial conditions $x(0) = x_0$ and $v(0) = v_0$ to find the position, velocity, and acceleration of a spring at a given time t.

16.5.2 3D Spring-Mass Systems

Mass-spring systems may be used to model approximations to more complicated physical systems. Rope or string may be modeled by placing a number of springs end-to-end, and cloth or rubber sheets may be modeled by placing masses on a grid and connecting adjacent masses by springs.



Let the ith mass, m_i, be at location $\bar{p}_i(t)$, with elements $x_i(t)$, $y_i(t)$, $z_i(t)$. Let l_{ij} denote the resting length and k_{ij} the stiffness of the spring between masses i and j.

The **internal force** for mass i is

$$f^s_{ij}(t) = -k_{ij} e_{ij} \frac{p_i - p_j}{\|p_i - p_j\|},$$

where $e_{ij} = l_{ij} - \|p_i - p_j\|$.

> *Note:*
> It is the case that $f^s_{ji}(t) = -f^s_{ij}(t)$.

The net total internal force on a mass i is then

$$f^s_i(t) = \sum_{j \in N_i} f^s_{ij}(t),$$

where N_i is the set of indices of neighbors of mass i.

16.5.3 Simulation and Discretization

A common approach to discretizing over time in a physical simulation is to use a numerical ordinary differential equation solver, such as the Runge-Kutta method, with finite difference approximations to derivatives.

To find an approximation to $a(t)$, we choose a time increment Δt so the solution is computed at $t_i = i\Delta t$.

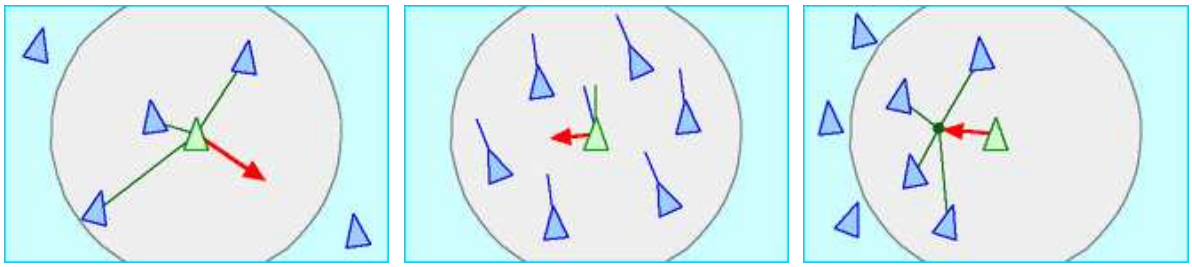
The simplest approach is the use Euler time integration with forward differences:

- Compute $\vec{a}_i(t) = f_i(t)/m_i$.
- Update $\vec{v}_i(t + \Delta t) = \vec{v}_i(t) + \Delta t \vec{a}_i(t)$.
- Update $\bar{p}_i(t + \Delta t) = \bar{p}_i(t) + \Delta t \vec{v}_i(t)$.

16.5.4 Particle Systems

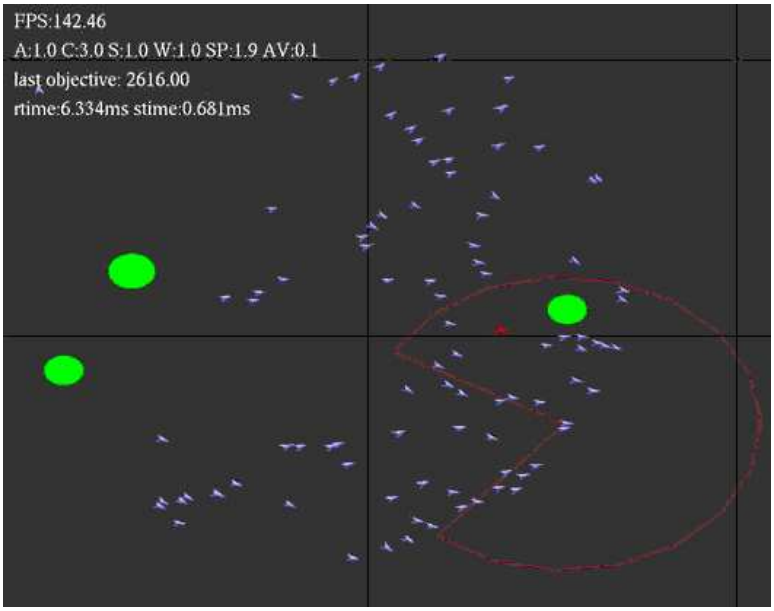
A particle system fakes passive dynamics to quickly render complex systems such as fire, flowing water, and sparks. A particle is a point in space with some associated parameters such as velocity, time to live, color, or whatever else might be appropriate for the given application. During a simulation loop, particles are created by emitters that determine their initial properties, and existing particles are removed if their time to live has been exceeded. The physical rules of the system are then applied to each of the remaining particles, and they are rendered to the display. Particles are usually rendered as flat textures, but they may be rendered procedurally or with a small mesh as well.

16.6 Behavioral Animation



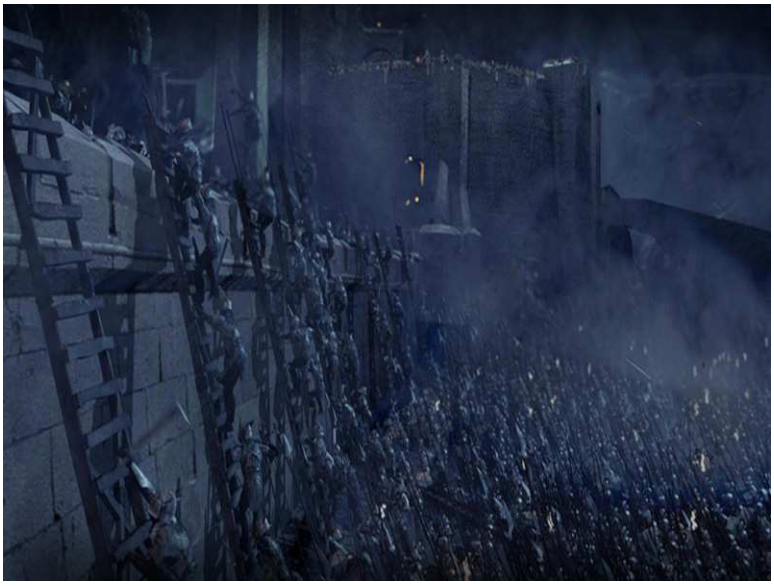
Flocking behaviors

Particle systems don't have to model physics, since rules may be arbitrarily specified. Individual particles can be assigned rules that depend on their relationship to the world and other particles, effectively giving them behaviors that model group interactions. To create particles that seem to flock together, only three rules are necessary to simulate separation between particles, alignment of particle steering direction, and the cohesion of a group of particles.



Particles that flock and steer around obstacles

More complicated rules of behavior can be designed to control large crowds of detailed characters that would be nearly impossible to manually animate by hand. However, it is difficult to program characters to handle all but simple tasks automatically. Such techniques are usually limited to animating background characters in large crowds and characters in games.



A crowd with rule-based behaviors

- Pros:

  - Automatic animation
  - Real-time generation

- Cons:
  - Human behavior is difficult to program
- Uses:
  - Crowds, flocks, game characters

16.7 Data-Driven Animation

Data-driven animation uses information captured from the real world, such as video or captured motion data, to generate animation. The technique of video textures finds points in a video sequence that are similar enough that a transition may be made without appearing unnatural to a viewer, allowing for arbitrarily long and varied animation from video. A similar approach may be taken to allow for arbitrary paths of motion for a 3D character by automatically finding frames in motion capture data or keyframed sequences that are similar to other frames. An animator can then trace out a path on the ground for a character to follow, and the animation is automatically generated from a database of motion.

- Pros:
  - Captures specific style of real actors
  - Very flexible
  - Can generate new motion in real-time
- Cons:
  - Requires good data, and possibly lots of it
- Uses:
  - Character animation

