CSC418
Lecture Notes

Yuchen Wang

July 8, 2020

Contents

1 Raster Images

1. Raster displays show images as rectangular arrays of pixels.
2. Images should be arrays of floating-point numbers, with either one (for **grayscale**, or black and white images), or three (for **RGB** color images) 32-bit floating point numbers stored per pixel.

Images as functions Grayscale: $I(x, y) : \mathbb{R}^2 \to \mathbb{R}$
RGB: $I(x, y) : \mathbb{R}^2 \to \mathbb{R}^3$

Bayer filters A **Bayer filter** is a color filter array for arranging RGB color filters on a square grid of photosensors. The information is stored as a seemingly 1-channel image, but with an understood convention for interpreting each pixel as the red, green or blue intensity value with a fixed pattern.

Demosaic algorithm To demosaic an image, we would like to create a full rgb image without down-sampling the image resolution. For each pixel

1. We'll use the exact color sample when it's available;
2. We average available neighbors (in all 8 directions) to fill in missing colors.

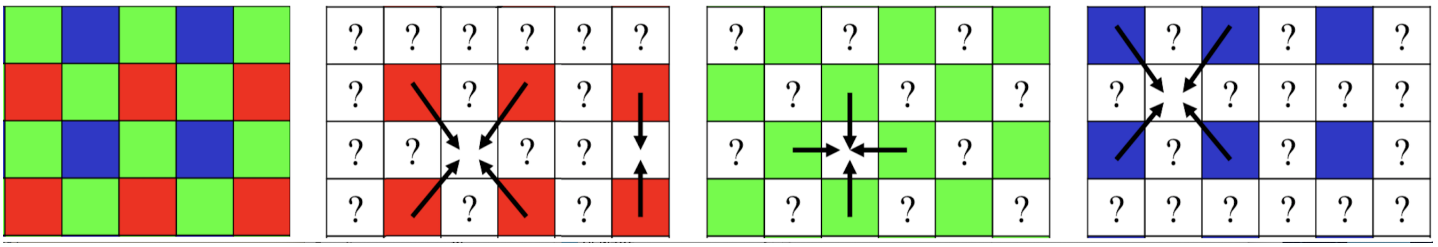


Figure 1: a: Bayer filters; bcd: Demosaic algorithm

Image storage on the computer To reduce the storage requirement, most image formats allow for some kind of compression, either *lossless* or *lossy*.

- **jpeg**: Lossy, compresses image blocks based on thresholds in the human visual system, works well for natural images.
- **tiff**: Losslessly compressed 8- or 16-bit RGB, or hold binary images.
- **ppm**: Lossless, uncompressed, most often used for 8-bit RGB images. Color intensities are represented as an integer between 0 and 255.
- **png**: Lossless, with a good set of open source management tools. can store rgba images

In our assignment we use the convention that the red value of pixel in the top-left corner comes first, then its green value, then its blue value, and then the rgb values of its neighbor to the right and so on across the row of pixels, and then moving to the next row down the columns of rows.

RGB images

HSV images Another useful representation is store the hue, saturation, and value of a color. This "hsv" representation also has 3-channels: typically, the hue or h channel is stored in degrees (i.e., on a periodic scale) in the range $[0°, 360°]$ and the saturation s and value v are given as absolute values in $[0, 1]$.

HSV/RGB conversion **Desaturation**: Given a `factor`, for each pixel,

1. Convert RGB values to HSV
2. Let $s \leftarrow (1 - factor) * s$
3. Convert HSV back to RGB

Hue shifting: **Desaturation**: Given a `shift`, for each pixel,

1. Convert RGB values to HSV
2. Let $h \leftarrow (h + shift) \bmod 360$
3. Convert HSV back to RGB

Grayscale images Take a weighted average of RGB values given higher priority to green:

$$i = 0.2126r + 0.7152g + 0.0722b \tag{1.1}$$

Alpha masking It is often useful to store a value α representing how opaque each pixel is.
When we store rgb + α image as a 4-channel rgba image. Just like rgb images, rgba images are 3D arrays unrolled into a linear array in memory.

2 Ray Casting

2.1 General Ray Casting Algorithm

```
for each pixel in the image {
        Generate a ray
        for each object in the scene {
                if (Intersect ray with object){
                        Set pixel color
                }
        }
}
```

2.2 Basics

Notation 2.1. **s**: the scene
e: the camera
d: the viewing direction $p(t)$: point along ray

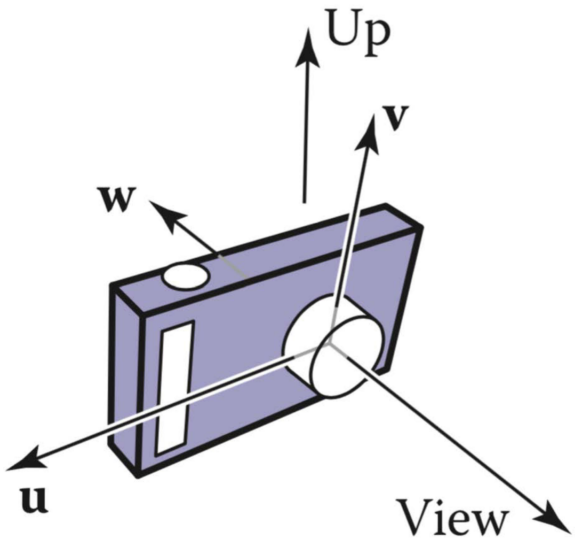
Parametric equation for ray A ray emanating from a point $\mathbf{e} \in \mathbb{R}^3$ in a direction $\mathbf{d} \in \mathbb{R}^3$ can be parameterized by a single number $t \in [0, \infty)$. Changing the value of t picks a different point along the ray. The parametric function for a ray is

$$p(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}) \tag{2.1}$$
$$= \mathbf{e} + t\mathbf{d} \tag{2.2}$$

where $p(0) = \mathbf{e}$ and $p(1) = \mathbf{s}$.
If $0 < t_1 < t_2$, then $p(t_1)$ is closer to the eye than $p(t_2)$.
Also, if $t < 0$, then $p(t)$ is "behind" the eye.

Camera space

$$\mathbf{w} = -\frac{View}{||View||} \tag{2.3}$$
$$\mathbf{u} = View \times Up \tag{2.4}$$
$$\mathbf{v} = \mathbf{w} \times \mathbf{u} \tag{2.5}$$



2.3 Orthographic/perspective projection

The view that is produced is determined by the choice of projection direction and image plane.

1. **Orthographic projection**: The image plane is perpendicular to the view direction
2. **Oblique projection**: The image plane is not perpendicular to the view direction
3. **Perspective projection**: Project alone lines that pass through a single *view point*, rather than along parallel lines

Orthographic views l and r are the positions of the left and right edges of the image, as measured from e along the **u** direction; and b and t are the positions of the bottom and top edges of the image, as measured from **e** along the **v** direction. Usually $l < 0 < r$ and $b < 0 < t$.
To fit an image with $n_x \times n_y$ pixels into a rectangle of size $(r-l) \times (t-b)$, the pixels are spaced a distance

$(r-l)/n_x$ apart horizontally and $(t-b)/n_y$ apart vertically, with a half-pixel space around the edge to center the pixel grid within the image rectangle.

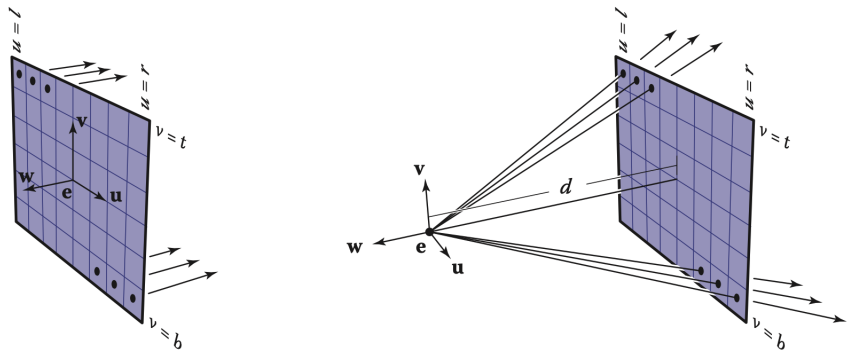
$$u = l + (r-l)(i+0.5)/n_x \tag{2.6}$$
$$v = b + (t-b)(j+0.5)/n_y \tag{2.7}$$

where (u, v) are the coordinates of the pixel's position on the image plane, measured with respect to the origin $\mathbf{e}$ and the basis $\{\mathbf{u}, \mathbf{v}\}$.
The procedure for generating orthographic viewing rays is then:

compute u and v using (4.1)
ray.direction $\leftarrow -\mathbf{w}$
ray.origin $\leftarrow \mathbf{e} + u\,\mathbf{u} + v\,\mathbf{v}$

Perspective views The image plane is some distance d (*image plane distance*) in front of $\mathbf{e}$. The resulting procedure is

compute u and v using (4.1)
ray.direction $\leftarrow -d\,\mathbf{w} + u\,\mathbf{u} + v\,\mathbf{v}$
ray.origin $\leftarrow \mathbf{e}$



Parallel projection
same direction, different origins

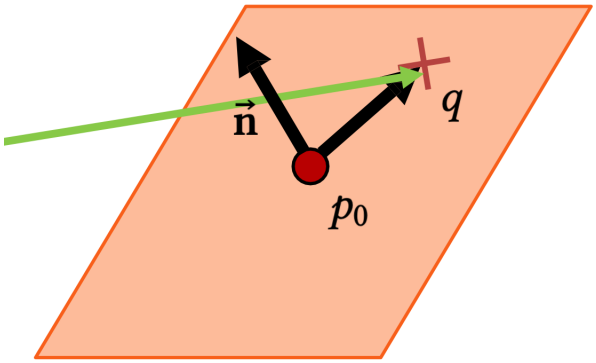
Perspective projection
same origin, different directions

Figure 4.9. Ray generation using the camera frame. Left: In an orthographic view, the rays start at the pixels' locations on the image plane, and all share the same direction, which is equal to the view direction. Right: In a perspective view, the rays start at the viewpoint, and each ray's direction is defined by the line through the viewpoint, $\mathbf{e}$, and the pixel's location on the image plane.

2.4 Ray-Object Intersection

2.4.1 Plane

Plane Equation



Plane equation
Substitute ray equation into it

$$\vec{\mathbf{n}} \cdot (\mathbf{p}(t) - p_0) = 0$$

$$\vec{\mathbf{n}} \cdot ((\mathbf{e} + t\vec{\mathbf{d}}) - p_0) = 0$$

Solve for t

$$t = \frac{-\vec{\mathbf{n}} \cdot (\mathbf{e} - p_0)}{\vec{\mathbf{n}} \cdot \vec{\mathbf{d}}}$$

2.4.2 Sphere

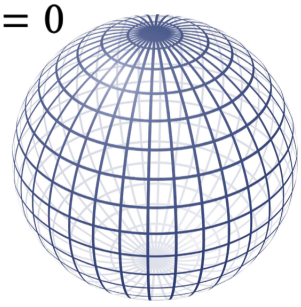
Ray-Sphere Intersection

Substitute ray equation into implicit equation for sphere

$$(\mathbf{e} + t\vec{\mathbf{d}} - \mathbf{c}) \cdot (\mathbf{e} + t\vec{\mathbf{d}} - \mathbf{c}) - r^2 = 0$$

Rearrange

$$(\vec{\mathbf{d}} \cdot \vec{\mathbf{d}})t^2 + 2\vec{\mathbf{d}} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - r^2 = 0$$



Looks familiar…

$$At^2 + Bt + C = 0$$

It's a quadratic! (can use the quadratic equation)

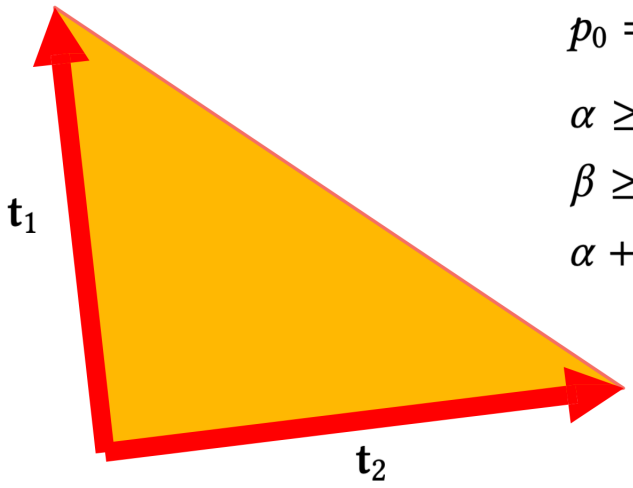
Hint for the homework: the discriminant tells us what kinds of roots the equation has.

$$intersect = \mathbf{e} + \mathbf{t} * \mathbf{d} \tag{2.8}$$

$$\mathbf{n} = (intersect - \mathbf{c}).normalized() \tag{2.9}$$

2.4.3 Triangle

Equations for a Triangle



$$p_0 = \alpha \mathbf{t}_1 + \beta \mathbf{t}_2$$

$$\alpha \geq 0$$

$$\beta \geq 0$$

$$\alpha + \beta \leq 1$$

Intersection with a Triangle (Parametric Surface)

Check via equating point on surface with point on ray

$$\mathbf{e} = \alpha \mathbf{t}_1 + \beta \mathbf{t}_2 - t\vec{\mathbf{d}}$$

$$\mathbf{e} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & -\mathbf{d} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ t \end{bmatrix}$$

Check values of α, β, t

$$\mathbf{n} = [(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})].normalized() \tag{2.10}$$

3 Ray Tracing

3.1 General Ray Tracing Algorithm

```
for each pixel in the image {
      pixel colour = rayTrace(viewRay, 0)
}

colour rayTrace(Ray, depth) {
      for each object in the scene {
            if(Intersect ray with object) {
                  colour = shading model
                  if(depth < maxDepth)
                  colour +=rayTrace(reflectedRay,depth+1)
            }
      }
      return colour
}
```

```
colour rayTrace(Ray, depth) {
     for each object in the scene {
          if(Intersect ray with object) {
               colour = shading model
               if(depth < maxDepth) {
                    colour +=
rayTrace(reflectedRay,depth+1)
                    colour +=
rayTrace(refractedRay,depth+1)
               }
          }
     }
     return colour
}
```

3.2 Image-order vs Object-order rendering

Image-order rendering Each pixel is considered in turn, and for each pixel all the objects that influence it are found and the pixel value is computed.
Simpler to get working, more flexible in the effects that can be produced, usually takes much more execution time to produce a comparable image.

Object-order rendering Each object is considered in turn, and for each object all the pixels that it influences are found and updated.

3.3 Two types of lights

Directional Light Direction of light does not depend on the position of the object. Light is very far away.

Point Light Direction of light depends on position of object relative to light.

3.4 Shading

Notation 3.1. The important variables in light reflection are unit vectors
Light direction **l**: a unit vector pointing toward the light source;
View direction **v**: a unit vector pointing toward the eye or camera;
Surface normal **n**: a unit vector perpendicular to the surface at the point where reflection is taking place.

3.4.1 Lambertian Shading

An observation by Lambert in the 18th century: the amount of energy from a light source that falls on an area of surface depends on the angle of the surface to the light.

Definition 3.1 (Lambertian shading model)**.** The vector **l** is computed by subtracting the intersection point of the ray and the surface from the light source position.
The pixel color

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

where k_d is the *diffuse coefficient*, or the surface color; and I is the intensity of the light source.

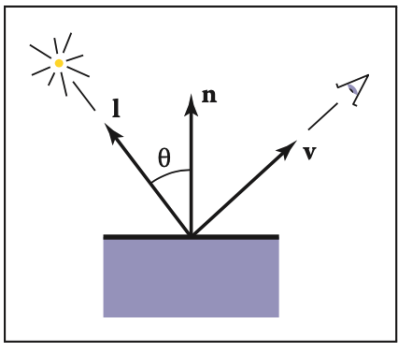


Figure 2: Geometry for Lambertian shading

Remark 3.1. Because **n** and **l** are unit vectors, we can use $\mathbf{n} \cdot \mathbf{l}$ as a convenient shorthand for $\cos\theta$. This equation applies separately to the three color channels.

Remark 3.2. Lambertian shading is *view independent:* the color of a surface does not depend on the direction from which you look. Therefore it does not produce any highlights and leads to a very matte, chalky appearance.

3.4.2 Blinn-Phong Shading

A very simple and widely used model for specular highlights by Phong (1975) and J.F.Blinn (1976).

Idea Produce reflection that is at its brightest when **v** and **l** are symmetrically positioned across the surface normal, which is when mirror reflection would occur; reflection then decreases smoothly as the vectors move away from a mirror configuration.
Compare the half vector **h** with **n**: if **h** is near the surface normal, the specular component should be bright and vice versa.

Definition 3.2 (Blinn-Phong shading model)**.**

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

where k_s is the *specular coefficient*, or the specular color of the surface, and $p > 1$.

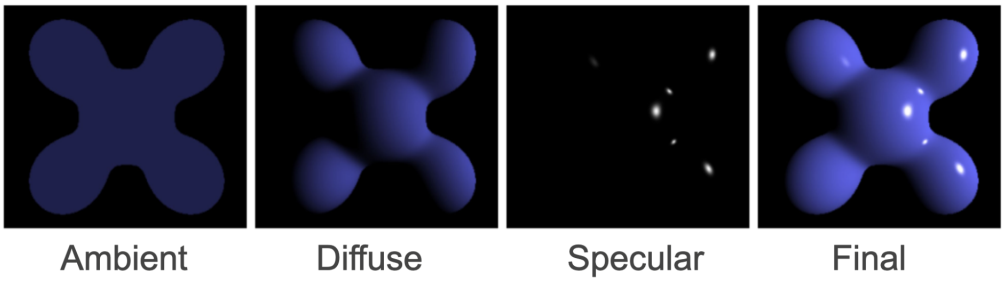
3.4.3 Ambient Shading

A heuristic to avoid black shadows is to add a constant component to the shading model, one whose contribution to the pixel color depends only on the object hit, with no dependence on the surface geometry at all, as if surfaces were illuminated by ambient light that comes equally from everywhere.

Definition 3.3 (simple shading model / Blinn-Phong model with ambient shading)**.**

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

where k_a is the surface's ambient coefficient or "ambient color", and I_a is the ambient light intensity.



3.4.4 Multiple Point Lights

Property 3.1 (superposition)**.** The effect by more than one light source is simply the sum of the effects of the light sources individually.

Definition 3.4 (extended simple shading model)**.**

$$L = k_a I_a + \sum_{i=1}^{N} [k_d I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p]$$

where $I_i, \mathbf{l}_i$ and $\mathbf{h}_i$ are the intensity, direction, and half vector of the i-th light source.

3.5 A Ray-Tracing Program

```
for each pixel do
    compute viewing ray
    if (ray hits an object with t ∈ [0, ∞)) then
        Compute n
        Evaluate shading model and set pixel to that color
    else
        set pixel color to background color
```

3.6 Shadows

Recall from 3.4 that light comes from direction $\mathbf{l}$. If we imagine ourselves at a point $\mathbf{p}$ on a surface being shaded, the point is in shadow if we "look" in direction $\mathbf{l}$ and see an object. If there are no objects, then the light is not blocked.

function raycolor(ray $\mathbf{e} + t\mathbf{d}$, real t_0, real t_1)
hit-record rec, srec
if (scene→hit($\mathbf{e} + t\mathbf{d}$, t_0, t_1, rec)) **then**
  $\mathbf{p} = \mathbf{e} + (\text{rec}.t)\,\mathbf{d}$
  color $c = \text{rec}.k_a\ I_a$
  if (not scene→hit($\mathbf{p} + s\mathbf{l}$, ϵ, ∞, srec)) **then**
    vector3 $\mathbf{h}$ = normalized(normalized($\mathbf{l}$) + normalized($-\mathbf{d}$))
    $c = c + \text{rec}.k_d\, I \max(0, \text{rec}.\mathbf{n} \cdot \mathbf{l}) + (\text{rec}.k_s)\, I\, (\text{rec}.\mathbf{n} \cdot \mathbf{h})^{\text{rec}.p}$
  return c
else
  return background-color

Remark 3.3. The usual adjustment to avoid the problem of intersecting $\mathbf{p}$ with the surface that generates it is to make the shadow ray check for $t \in [\epsilon, \infty)$ where ϵ is some small positive constant.

Remark 3.4. The code above assumes that $\mathbf{d}$ and $\mathbf{l}$ are not necessarily unit vectors.

3.7 Ideal Specular Reflection

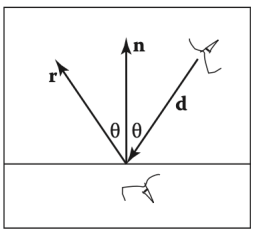


Figure 3: When looking into a perfect mirror, the viewer looking in direction $\mathbf{d}$ will see whatever the viewer "below" the surface would see in direction $\mathbf{r}$

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

In the real world, a surface may reflect some colors more efficiently than others, so it shifts the colors of the objects it reflects (i.e. some energy is lost when the light reflects from the surface).
We implement the reflection by a recursive call of *raycolor*:

```
color c = c + km * raycolor(p + sr, epsilon, max_t);
```

where k_m is the specular RGB color, p is the intersection of the viewing ray and the surface, $s \in [\epsilon, max\_t)$ for the same reason as we did with shadow rays: we don't want the reflection ray to hit the object that generates it.
To make sure that the recursive call will terminate, we need to add a maximum recursion depth.

4 Bounding Volume Hierachy

4.1 Two types of subdivisions

Object-based subdivision:

- AABB Trees

Spatial subdivision:

- Uniform Spatial Subdivision
- Axis-Aligned Spatial Subdivision

4.2 Bounding Boxes

Definition 4.1. We first consider a 2D ray whose direction vector has positive x and y components. The 2D bounding box is defined by two horizontal and two vertical lines:

$$x = x_{\text{min}}$$
$$x = x_{\text{max}}$$
$$y = y_{\text{min}}$$
$$y = y_{\text{max}}$$

The points bounded by these lines can be described in interval notation:

$$(x, y) \in [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]$$

The intersection test can be phrased in terms of these intervals. First, we compute the ray parameter where the ray hits the line $x = x_{\text{min}}$:

$$t_{xmin} = \frac{x_{\text{min}} - x_e}{x_d}$$

We then make similar computations for t_{xmax}, t_{ymin}, and t_{ymax}. The ray hits the box if and only if the intervals $[t_{xmin}, t_{xmax}]$ and $[t_{ymin}, t_{ymax}]$ overlap.

$t_{\text{xmin}} = (x_{\text{min}} - x_e)/x_d$
$t_{\text{xmax}} = (x_{\text{max}} - x_e)/x_d$
$t_{\text{ymin}} = (y_{\text{min}} - y_e)/y_d$
$t_{\text{ymax}} = (y_{\text{max}} - y_e)/y_d$
if $(t_{\text{xmin}} > t_{\text{ymax}})$ or $(t_{\text{ymin}} > t_{\text{xmax}})$ **then**
  return false
else
  return true

Figure 4: The algorithm pseudocode.

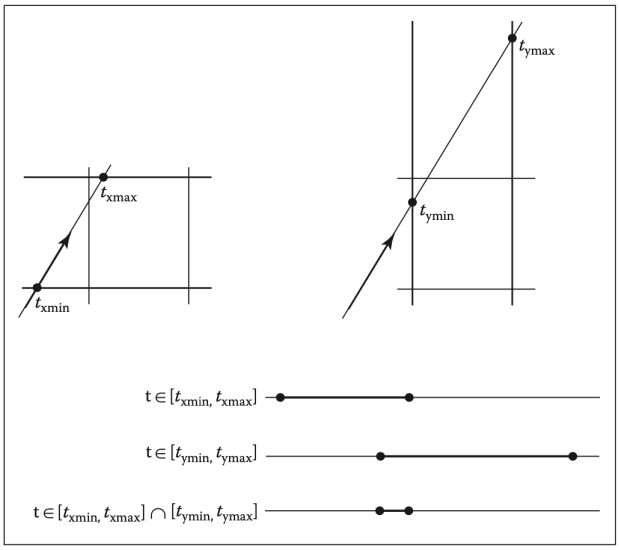


Figure 12.24. The ray will be inside the interval $x \in [x_{\min}, x_{\max}]$ for some interval in its parameter space $t \in [t_{\text{xmin}}, t_{\text{xmax}}]$. A similar interval exists for the y interval. The ray intersects the box if it is in both the x interval and y interval at the same time, i.e., the intersection of the two one-dimensional intervals is not empty.

Ray-AABB Intersection If $\max(t_{xmin}, t_{ymin}, y_{zmin}) < \min(t_{xmax}, t_{ymax}, y_{zmax})$, then the ray intersects with the box.

Issue 1 The first thing we must address is the case when x_d or y_d is negative. If x_d is negative, then the ray will hit $x_{\max}$ before it hits $x_{\min}$. Thus the code for computing t_{xmin} and t_{xmax} expands to:

$$
\begin{aligned}
&\textbf{if } (x_d \geq 0) \textbf{ then} \\
&\quad t_{\text{xmin}} = (x_{\min} - x_e)/x_d \\
&\quad t_{\text{xmax}} = (x_{\max} - x_e)/x_d \\
&\textbf{else} \\
&\quad t_{\text{xmin}} = (x_{\max} - x_e)/x_d \\
&\quad t_{\text{xmax}} = (x_{\min} - x_e)/x_d
\end{aligned}
$$

A similar code expansion must be made for the y cases.

Issue 2 A major concern is that horizontal and vertical rays have a zero value for y_d and x_d, respectively. This will cause divide by zero which may be a problem. We define the **rules for divide by zero**:
For any positive real number a,

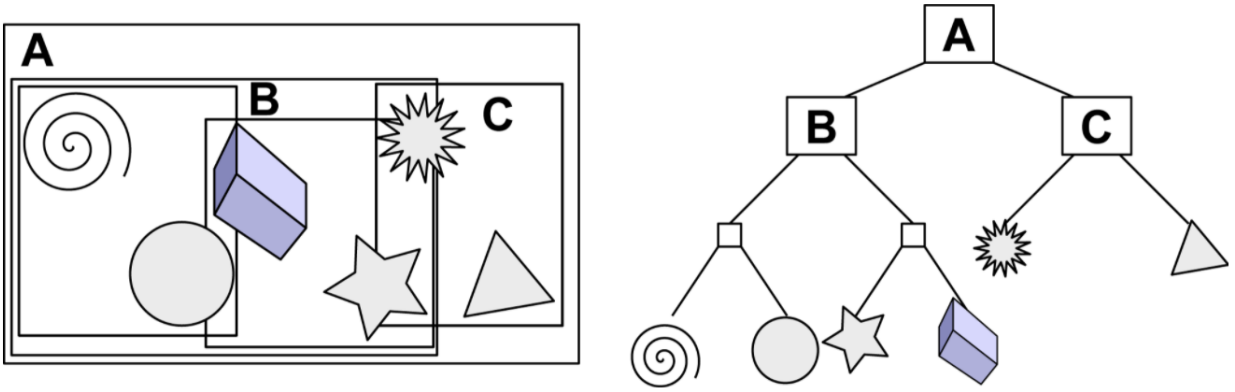
$$+a/0 = +\infty$$
$$-a/0 = -\infty$$

$$
\begin{aligned}
&a = 1/x_d \\
&\textbf{if } (a \geq 0) \textbf{ then} \\
&\quad t_{\text{min}} = a(x_{\text{min}} - x_e) \\
&\quad t_{\text{max}} = a(x_{\text{max}} - x_e) \\
&\textbf{else} \\
&\quad t_{\text{min}} = a(x_{\text{max}} - x_e) \\
&\quad t_{\text{max}} = a(x_{\text{min}} - x_e)
\end{aligned}
$$

Figure 5: The new code segment.

4.3 Axis-Aligned Bounding Boxes Tree (AABB Tree)

In our assignment, we will build a binary tree. Conducting queries on the tree will be reminiscent of searching for values in a *binary search tree*.



Basic idea: Place an axis-aligned 3D bounding box around all the objects. Since testing for intersection with the box is not free, rays that hit the bounding box will be more expensive to compute than in a brute force search, but rays that miss the box are cheaper. Such bounding boxes can be made hierarchical by partitioning the set of objects in a box and placing a box around each partition.

Remark 4.1. Each object belongs to one of two sibling nodes, whereas a point in space may be inside both sibling nodes.

Remark 4.2 (pros and cons)**.** As follows:

- pros: Operations (e.g. growing the bounding box, testing ray intersection or determining closest-point distances with an axis-aligned bounding box) usually reduce to trivial per-component arithmetic. This means the code is simple to write/debug and also inexpensive to evaluate.
- cons: In general, AABBs will not tightly enclose a set of objects.

Ray intersection algorithm The ray intersection algorithm is essentially performing a depth first search.

```
intersect(bvNode, ray,t)
{
   if (bvNode== null || !bvNode.intersect(ray,t))
       return false;
   else
   {
      i1=intersect(bvNode.left, ray,t1);  //check left BV
      i2=intersect(bvNode.right, ray,t2); //check right BV
      if (i1 && i2) {  t=min(t1,t2); return true;  }
      if (i1) {  t=t1; return true;  }
      if (i2) {  t=t2; return true;  }
      return false;
   }
}
```

Distance query algorithm In this algorithm, we are interested in which box in the AABB tree is the closest to a fixed point.

```
minDistance(bvNode, point, currentMin)
{
  d1=minDistance(bvNode.left, point, currentMin);
  d2=minDistance(bvNode.right, point, currentMin);

   if(min(d1,d2) > currentMin) {
     return currentMin
   }

   return min(d1,d2)
}
```

Remark 4.3 (pros and cons of DFS)**.** As follows:

- pros: The search usually doesn't have to visit the entire tree because most boxes are not hit by the given ray. In this way, many search paths are quickly aborted.
- cons: Every box has some closest point to our query. A naive depth-first search could end up searching over every box before finding the one with the smallest query.
- BFS is a much better structure since it makes use of a priority queue to explore the current best looking path in our tree.

Remark 4.4. Tree construction complexity: $\mathcal{O}(n \log n)$
Tree traversal complexity: $\mathcal{O}(n)$ (worst case)
Brute force complexity: $\mathcal{O}(n)$

4.4 Object-Oriented Bounding Box (OOBB)

Find directions of maximum variance using PCA.

4.5 Uniform Spatial Subdivision

In spatial division, each point in space belongs to exactly one node, whereas objects may belong to many nodes.
The scene is partitioned into axis-aligned boxes. These boxes are all the same size, although they are not necessarily cubes. The ray traverses these boxes. When an object is hit, the traversal ends.

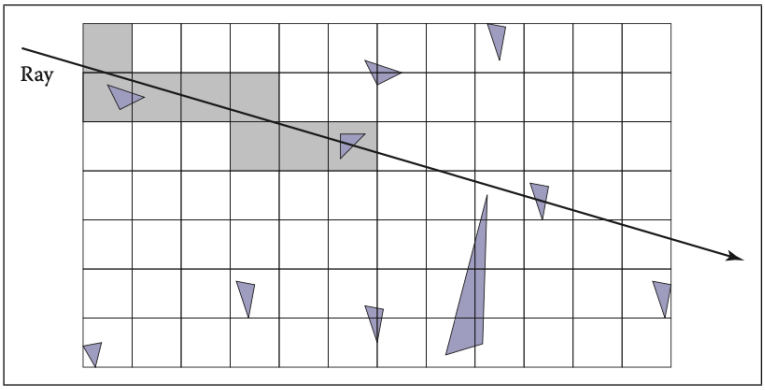


Figure 12.28. In uniform spatial subdivision, the ray is tracked forward through cells until an object in one of those cells is hit. In this example, only objects in the shaded cells are checked.

This traversal is done in an increment fashion. We need to find the index (i, j) of the first cell hit by the ray $\mathbf{e} + t\mathbf{d}$. Then we need to traverse the cells in an appropriate order.

4.6 Axis-Aligned Binary Space Partitioning (BSP Trees)

We can also partition space in a hierarchical data structure such as a *binary space partitioning tree* (BSP tree). It's common to use axis-aligned cutting planes for ray intersection.
A node in this structure contains a single cutting plane and a left and right subtree. Each subtree contains all the objects on one side of the cutting plane. Objects that pass through the plane are stored in both subtrees. If we assume the cutting plane is parallel to the yz plane at $x = D$, then the node class is:

```
class bsp-node subclass of surface
  virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
  virtual box bounding-box()
  surface-pointer left
  surface-pointer right
  real D
```

The intersection code can then be called recursively in an object-oriented style. The code considers the four cases:

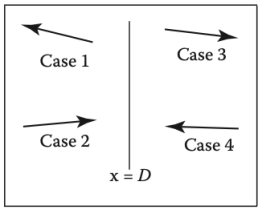


Figure 12.32. The four cases of how a ray relates to the BSP cutting plane $x = D$.

The origin of these rays is a point at parameter t_0:

$$\mathbf{p} = \mathbf{a} + t_0\mathbf{b}$$

The four cases are:

1. The ray only interacts with the left subtree, and we need not test it for intersection with the cutting plane. It occurs for $x_p < D$ and $x_b < 0$.
2. The ray is tested against the left subtree, and if there are no hits, it is then tested against the right subtree. We need to find the ray parameter at $x = D$, so we can make sure we only test for intersections within the subtree. This case occurs for $x_p < D$ and $x_b > 0$.
3. This case is analogous to case 1 and occurs for $x_p > D$ and $x_b > 0$.
4. This case is analogous to case 2 and occurs for $x_p > D$ and $x_b < 0$.

The resulting traversal code handling these cases in order is:

```
function bool bsp-node::hit(ray a + tb, real t0, real t1,
                            hit-record rec)
  xp = xa + t0xb
  if (xp < D) then
    if (xb < 0) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    t = (D − xa)/xb
    if (t > t1) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    if (left ≠ NULL) and (left→hit(a + tb, t0, t, rec)) then
      return true
    return (right ≠ NULL) and (right→hit(a + tb, t, t1, rec))
  else
    analogous code for cases 3 and 4
```

5 Triangle Meshes

Motivation

- Mesh is a basic computer graphics data structure.
- Most real-world models are composed of complexes of triangles with shared vertices. These are usually known as **triangle meshes**, and handling them efficiently is crucial to the performance of many graphics programs.

- We'd like to minimize the amount of storage consumed.
- Besides basic operations (storing & drawing), other operations such as *subdivision*, *mesh editing* and *mesh compression*, efficient access to adjacency information is crucial.

5.1 Mesh Topology

Property 5.1 (Manifold meshes)**.** The topology of a mesh is a **manifold** surface: A surface in which a small neighborhood around any point could be smoothed out into a bit of flat surface.

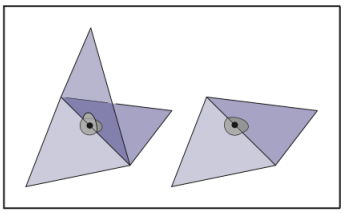


Figure 12.1. Non-manifold (left) and manifold (right) interior edges.

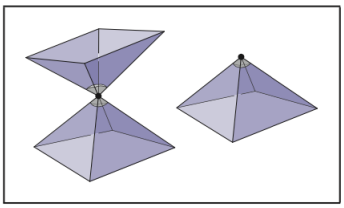


Figure 12.2. Non-manifold (left) and manifold (right) interior vertices.

Verifying the property Prevent crashes or infinite loops by checking:

1. Every edge is shared by exactly two triangles. (as in figure 12.1)
2. Every vertex has a single, complete loop of triangles around it. (as in figure 12.2)

Property 5.2 (Manifold with boundary meshes)**.** Sometimes it's necessary to allow meshes to have boundaries. Such meshes are not manifolds. However, we can relax the requirements of a manifold mesh to those for a **manifold with boundary** without causing problems for most mesh processing algorithms. The relaxed conditions are:

1. Every edge is used by either one or two triangles.
2. Every vertex connects to a single edge-connected set of triangles.

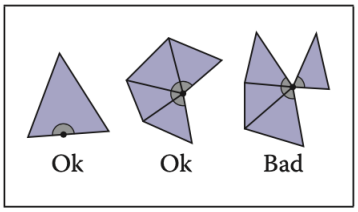


Figure 12.3. Conditions at the edge of a manifold with boundary.

Figure 12.3 illustrates these conditions.

Property 5.3 (orientation)**.** For a single triangle, we define orientation based on the order in which the vertices are listed: the front is the side from which the triangle's three vertices are arranged in counterclockwise order.
A connected mesh is **consistently oriented** if its triangles all agree on which side is the front.

5.2 Indexed Mesh Storage

Definition 5.1 (Separate triangles)**.** Store each triangle as an independent entity.

```
Triangle {
        vector3 vertexPosition[3]
}
```

Definition 5.2 (Indexed Shared-vertex mesh)**.** This data structure has triangles which **point to** vertices which contain the vertex data:

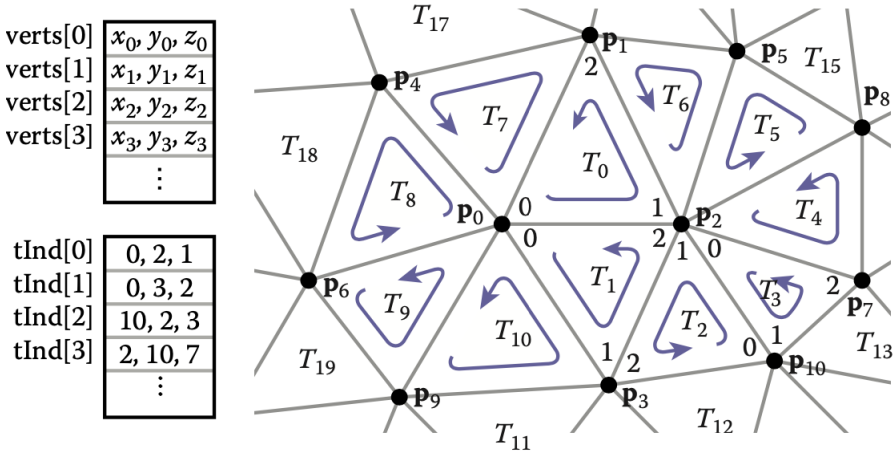
```
Triangle {
        Vertex v[3]
}

Vertex {
        vector3 position
}
```

In implementation, the vertices and triangles are normally stored in arrays:

```
IndexedMesh{
        int tInd[nt][3]
        vector3 verts[nv]
}
```

The index of the kth vertex of the ith triangle is found in `tInd[i][k]`, and the position of that vertex is stored in the corresponding row of the `verts` arary.



Space requirements If our mesh has n_v vertices and n_t triangles, and if we assume that the data for floats, pointers, and ints all require the same storage, the space requirements are as follows:

- **Triangle**: Three vectors per triangle, for $9n_t$ units of storage;
- **IndexedMesh**: One vector per vertex and three ints per triangle, for $3n_v + 3n_t$ units of storage.

Empirically, $n_t \approx 2n_v$.

5.3 Triangle Strips and Fans

In applications where even more compactness is desirable, the triangle vertex indices can be expressed more efficiently using **triangle strips** and **triangle fans**.

Definition 5.3 (Triangle fan)**.** All the triangles share one common vertex, and the other vertices generate a set of triangles like the vanes of a collapsible fan (Figure 12.9). The fan in the figure could be specified with the sequence $[0, 1, 2, 3, 4, 5]$: the first vertex establishes the center, and subsequently each pair of adjacent vertices creates a triangle.

Definition 5.4 (Triangle strip)**.** Here, vertices are added alternating top and bottom in a linear strip as shown in Figure 12.10. The triangle strip in the figure could be specified by the sequence [0 1 2 3 4 5 6 7], and every subsequence of three adjacent vertices creates a triangle. For consistent orientation, every other triangle needs to have its order reversed. For each new vertex that comes in, the oldest vertex is forgotten and the order of the two remaining vertices is swapped.

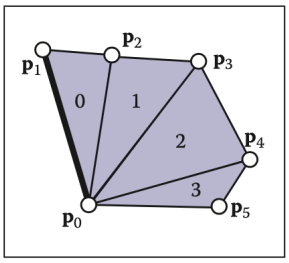


Figure 12.9. A triangle fan.

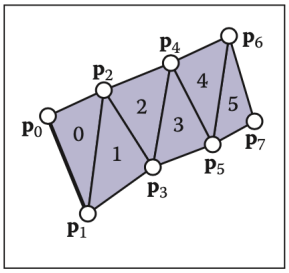


Figure 12.10. A triangle strip.

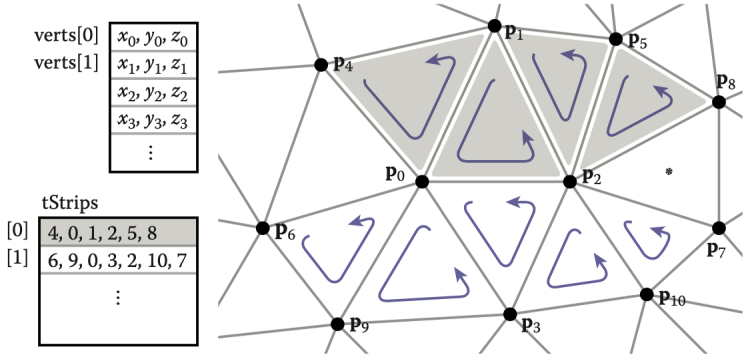


Figure 12.11. Two triangle strips in the context of a larger mesh. Note that neither strip can be extended to include the triangle marked with an asterisk.

In both strips and fans, $n + 2$ vertices suffice to describe n triangles.

5.4 Data Structures for Mesh Connectivity

5.4.1 The Triangle-Neighbor Structure

Augmenting the basic shared-vertex mesh with pointers from the triangles to the three neighboring triangles, and a pointer from each vertex to one of the adjacent triangles (it doesn't matter which one)

```
Triangle {
        Triangle nbr[3];
```

```
        Vertex v[3];
}

Vertex {
        // ... per-vertex data ...
        Triangle t; // any adjacent tri
}
```

In the array `Triangle.nbr`, the kth entry points to the neighboring triangle that shares vertices k and $k+1$.
Implementation:

```
Mesh{
        vector3 verts[nv] // per-vertex data
        int tInd[nt][3]; // vertex indices
        int tNbr[nt][3]; // indices of neighbor triangles
        int vTri[nv]; // index of any adjacent triangle
}
```

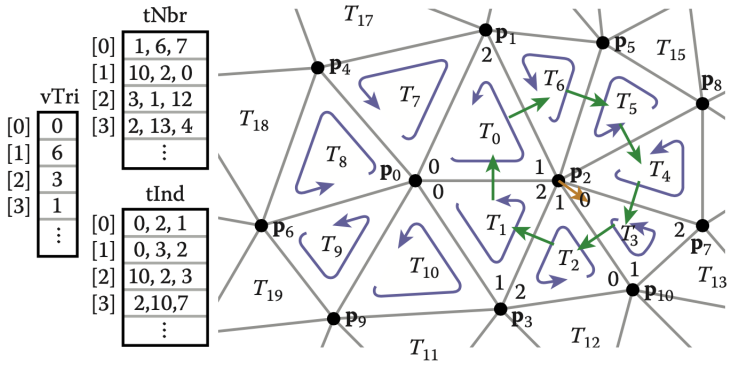


Figure 12.13. The triangle-neighbor structure as encoded in arrays, and the sequence that is followed in traversing the neighboring triangles of vertex 2.

The idea is to move from triangle to triangle, visiting only the triangles adjacent to the relevant vertex. If triangle t has vertex v as its kth vertex, then the triangle t.nbr[k] is the next triangle around v in the clockwise direction. This observation leads to the following algorithm to traverse all the triangles adjacent to a given vertex:

```
TrianglesOfVertex(v) { t = v.t
do {
find i such that (t.v[i] == v) t = t.nbr[i]
} while (t != v.t) }
```

A small refinement to let us find the previous triangle:

```
Triangle {
   Edge nbr[3];
   Vertex v[3];
}
Edge { // the i-th edge of triangle t Triangle t;
   int i;  // in {0,1,2}
}
Vertex {
   // ... per-vertex data ...
   Edge e; // any edge leaving vertex
}
```

```
TrianglesOfVertex(v) {
        {t, i} = v.e;
        do {
                {t, i} = t.nbr[i];
                i = (i+1) mod 3;
        } while (t != v.e.t);
}
```

5.5 The Winged-Edge Structure

In a winged-edge mesh, each edge stores pointers to the two vertices it connects (the head and tail vertices), the two faces it is part of (the left and right faces), and, most importantly, the next and previous edges in the counterclock- wise traversal of its left and right faces (Figure 12.16). Each vertex and face also stores a pointer to a single, arbitrary edge that connects to it:

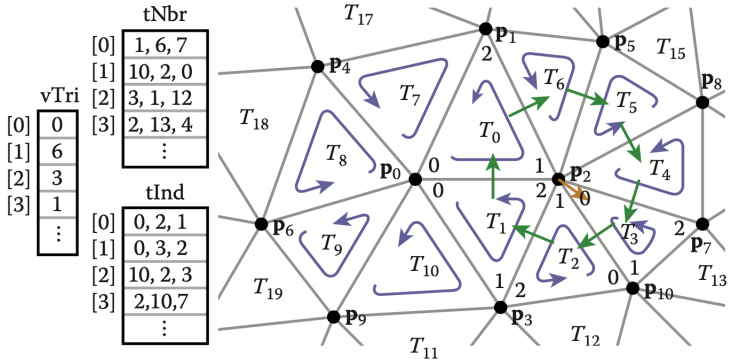
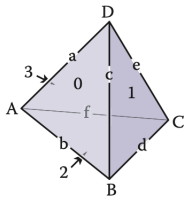


Figure 12.13. The triangle-neighbor structure as encoded in arrays, and the sequence that is followed in traversing the neighboring triangles of vertex 2.

```
Edge {
        Edge lprev, lnext, rprev, rnext;
        Vertex head, tail;
        Face left, right;
}
Face {
        // ... per-face data ...
        Edge e; // any adjacent edge
}
Vertex {
        // ... per-vertex data ... Edge e; // any incident edge
}
```



Edge	*Vertex 1*	*Vertex 2*	*Face left*	*Face right*	*Pred left*	*Succ left*	*Pred right*	*Succ right*
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	e	b	d

Vertex	*Edge*
A	a
B	d
C	d
D	e

Face	*Edge*
0	a
1	c
2	d
3	a

Figure 12.15. A tetrahedron and the associated elements for a winged-edge data structure. The two small tables are not unique; each vertex and face stores any one of the edges with which it is associated.

The search algorithm:

```
EdgesOfVertex(v) {
   e = v.e;
   do {
      if (e.tail == v)
         e = e.lprev;
      else
         e = e.rprev;
   } while (e != v.e);
}

EdgesOfFace(f) {
   e = f.e;
   do {
      if (e.left == f)
         e = e.lnext;
      else
         e = e.rnext;
   } while (e != f.e);
}
```

5.6 The Half-Edge Structure

The winged-edge structure is quite elegant, but it has one remaining awkward- ness—the need to constantly check which way the edge is oriented before moving to the next edge.
Solution: Store data for each **half-edge**. There is one half-edge for each of the two triangles that share an edge, and the two half-edges are oriented oppositely, each oriented consistently with its own triangle. The data normally stored in an edge is split between the two half-edges. Each half-edge points to the face on its side of the edge and to the vertex at its head, and each contains the edge pointers for its face. It also points to its neighbor on the other side of the edge, from which the other half of the information can be found.

```
HEdge {
   HEdge pair, next;
   Vertex v;
   Face f;
}
Face {
        // ... per-face data ...
        HEdge h; // any h-edge of this face
}
Vertex {
        // ... per-vertex data ...
        HEdge h; // any h-edge pointing toward this vertex
}
```

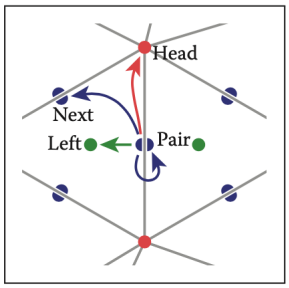


Figure 12.17. The references from a half-edge to its neighboring mesh components.

The search algorithm:

```
EdgesOfVertex(v) {
        h = v.h;
        do {
                h = h.pair.next;
        } while (h != v.h);
}

EdgesOfFace(f) {
        h = f.h;
        do {
                h = h.next;
        } while (h != f.h);
}
```

The vertex traversal here is clockwise, which is necessary because of omitting the prev pointer from the structure.
Because half-edges are generally allocated in pairs (at least in a mesh with no boundaries), many implementations can do away with the pair pointers. For instance, in an implementation based on array indexing (such as shown in Figure 12.18), the array can be arranged so that an even-numbered edge i always pairs with edge $i + 1$ and an odd-numbered edge j always pairs with edge $j - 1$.
In addition to the simple traversal algorithms shown in this chapter, all three of these mesh topology structures can support "mesh surgery" operations of various sorts, such as splitting or collapsing vertices, swapping edges, adding or removing triangles, etc.

5.7 Different types of normals

Per-Face Normals The surface will have a **faceted appearance**. This appearance is mathematically correct, but not necessarily desired if we wish to display a smooth looking surface.

Per-Vertex Normals Corners of triangles located at the same vertex should share the same normal vector. **Smooth appearance**. A common way to define per-vertex normals is to take a **area-weighted**

average of normals from incident faces:

$$\mathbf{n}_v = \frac{\sum_{f \in N(v)} a_f \mathbf{n}_f}{||\sum_{f \in N(v)} a_f \mathbf{n}_f||} \tag{5.1}$$

where $N(v)$ is the set of faces neighboring the v-th vertex, a_f is the area of face f, and $\mathbf{n}_f$ is the normal vector of face f.

Per-Corner Normals For surfaces with a mixture of smooth-looking parts and creases, it is useful to define normals independently for each triangle corner (as opposed to each mesh vertex). For each corner, we'll again compute an area-weighted average of normals triangles incident on the shared vertex at this corner, but we'll ignore triangle's whose normal is too different from the corner's face's normal:

$$\mathbf{n}_{f,c} = \frac{\sum_{g \in N(v) | \mathbf{n}_g \cdot \mathbf{n}_f > \epsilon} a_g \mathbf{n}_g}{||\sum_{g \in N(v) | \mathbf{n}_g \cdot \mathbf{n}_f > \epsilon} a_g \mathbf{n}_g||} \tag{5.2}$$

where ϵ is the minimum dot product between two face normals before we declare there is a crease between them.

5.8 Catmull-Clark subdivision algorithm

Subdivision Surfaces A Recursive refinement of polygonal mesh that results in smooth "limit surface".

5.9 Catmull-Clark subdivision algorithm

The first and still popular subdivision scheme.

1. Set the face point for each facet to be the average of its vertices.
2. Add edge points - average of two neighboring face points and edge end points.
3. Add edges between face points and edge points.
4. Move each original vertex according to new position given by

   $$\frac{F + 2R + (n-3)P}{n}$$

   where F is average of all n created face points adjacent to P and R is average of all original edge midpoints touching P.
5. Connect up original points to make facets
6. Repeat

If texture coordinates is on the test, look up textbook p249