

# CSC418

## Lecture Notes

Yuchen Wang

August 21, 2020

## Contents

<b>1 Raster Images</b>	<b>3</b>
<b>2 Ray Casting</b>	<b>4</b>
2.1 General Ray Casting Algorithm . . . . .	4
2.2 Basics . . . . .	4
2.3 Orthographic/perspective projection . . . . .	5
2.4 Ray-Object Intersection . . . . .	7
2.4.1 Plane . . . . .	7
2.4.2 Sphere . . . . .	7
2.4.3 Triangle . . . . .	8
<b>3 Ray Tracing</b>	<b>9</b>
3.1 General Ray Tracing Algorithm . . . . .	9
3.2 Image-order vs Object-order rendering . . . . .	9
3.3 Two types of lights . . . . .	9
3.4 Shading . . . . .	10
3.4.1 Lambertian Shading . . . . .	10
3.4.2 Blinn-Phong Shading . . . . .	10
3.4.3 Ambient Shading . . . . .	11
3.4.4 Multiple Point Lights . . . . .	11
3.5 A Ray-Tracing Program . . . . .	11
3.6 Shadows . . . . .	12
3.7 Ideal Specular Reflection . . . . .	12
<b>4 Bounding Volume Hierachy</b>	<b>13</b>
4.1 Two types of subdivisions . . . . .	13
4.2 Bounding Boxes . . . . .	13
4.3 Axis-Aligned Bounding Boxes Tree (AABB Tree) . . . . .	15
4.4 Object-Oriented Bounding Box (OOBB) . . . . .	17
4.5 Uniform Spatial Subdivision . . . . .	17
4.6 Axis-Aligned Binary Space Partitioning (BSP Trees) . . . . .	17
<b>5 Triangle Meshes</b>	<b>18</b>
5.1 Mesh Topology . . . . .	19
5.2 Indexed Mesh Storage . . . . .	20
5.3 Triangle Strips and Fans . . . . .	21
5.4 Data Structures for Mesh Connectivity . . . . .	22

5.4.1	The Triangle-Neighbor Structure . . . . .	22
5.5	The Winged-Edge Structure . . . . .	24
5.6	The Half-Edge Structure . . . . .	26
5.7	Different types of normals . . . . .	27
5.8	Catmull-Clark subdivision algorithm . . . . .	28
<b>6</b>	<b>Shader Pipeline</b>	<b>28</b>
6.1	Rasterization . . . . .	31
6.1.1	Triangle Rasterization . . . . .	31
6.2	Operations Before and After Rasterization . . . . .	31
6.2.1	Vertex-processing stage . . . . .	31
6.2.2	Fragment processing stage . . . . .	31
6.2.3	Blending stage . . . . .	32
<b>7</b>	<b>Kinematics</b>	<b>32</b>
<b>8</b>	<b>Mass-Spring Systems</b>	<b>32</b>
8.1	Bones . . . . .	32
8.2	Forward Kinematics . . . . .	33
8.3	Keyframe animation . . . . .	33
8.4	Inverse Kinematics . . . . .	33
8.5	Projected Gradient Descent . . . . .	34
8.6	Line Search . . . . .	34
<b>9</b>	<b>Computational Fabrication</b>	<b>34</b>

# 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:  $I(x, y) : \mathbb{R}^2 \rightarrow \mathbb{R}$

RGB:  $I(x, y) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$

**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.

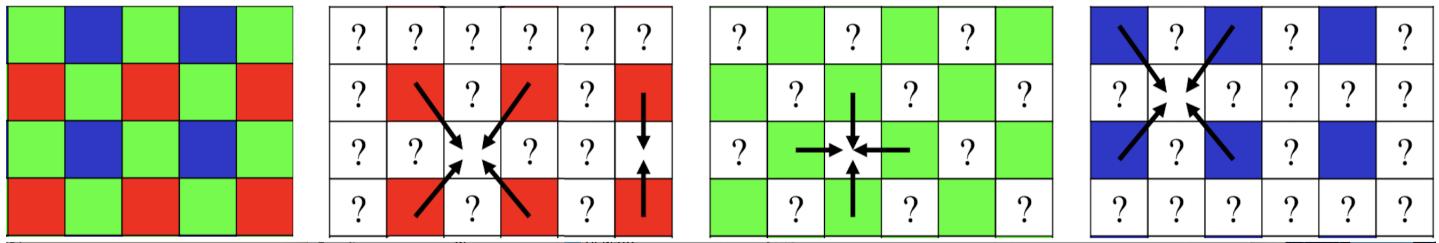


Figure 1: a: Bayer filters; bcd: Demosaic algorithm

**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  $[0^\circ, 360^\circ]$  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:

$$i = 0.2126r + 0.7152g + 0.0722b \quad (1.1)$$

**Alpha masking** It is often useful to store a value  $\alpha$  representing how opaque each pixel is. When we store  $rgb + \alpha$  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

$$p(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}) \quad (2.1)$$

$$= \mathbf{e} + t\mathbf{d} \quad (2.2)$$

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  $p(t_2)$ .

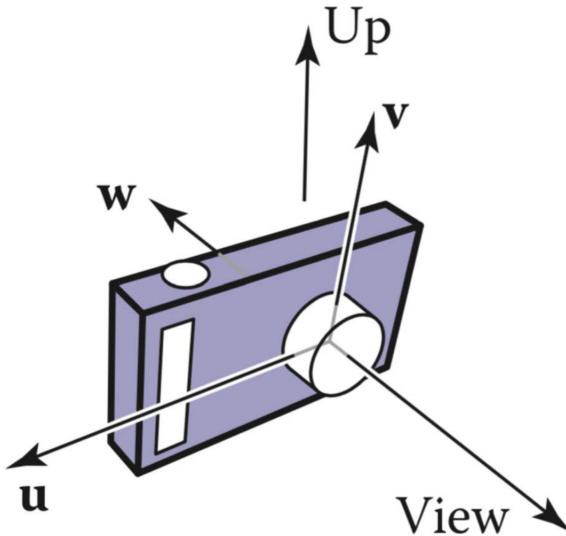
Also, if  $t < 0$ , then  $p(t)$  is “behind” the eye.

### Camera space

$$\mathbf{w} = -\frac{\text{View}}{\|\text{View}\|} \quad (2.3)$$

$$\mathbf{u} = \text{View} \times \text{Up} \quad (2.4)$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u} \quad (2.5)$$



### 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 along 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  $\mathbf{e}$  along the  $\mathbf{u}$  direction; and  $b$  and  $t$  are the positions of the bottom and top edges of the image, as measured from  $\mathbf{e}$  along the  $\mathbf{v}$  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.

$$u = l + (r - l)(i + 0.5)/n_x \quad (2.6)$$

$$v = b + (t - b)(j + 0.5)/n_y \quad (2.7)$$

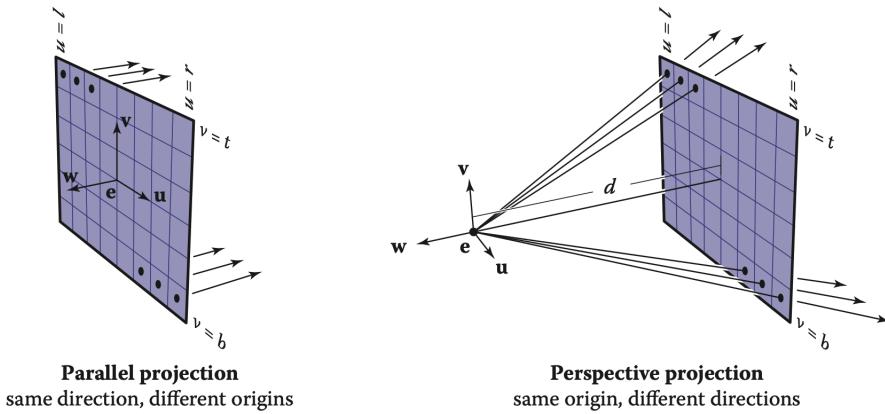
where  $(u, v)$  are the coordinates of the pixel's position on the image plane, measured with respect to the origin  $\mathbf{e}$  and the basis  $\{\mathbf{u}, \mathbf{v}\}$ .

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  $\leftarrow \mathbf{e}$ 
```

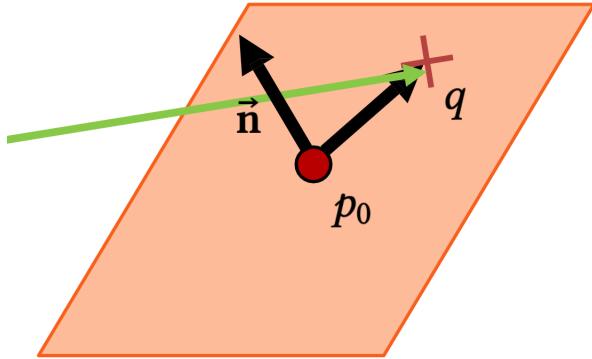


**Figure 4.9.** Ray generation using the camera frame. Left: In an orthographic view, the rays start at the pixels' locations on the image plane, and all share the same direction, which is equal to the view direction. Right: In a perspective view, the rays start at the viewpoint, and each ray's direction 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

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

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

Solve for t

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

### 2.4.2 Sphere

## Ray-Sphere Intersection

Substitute ray equation into implicit equation for sphere

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

Rearrange

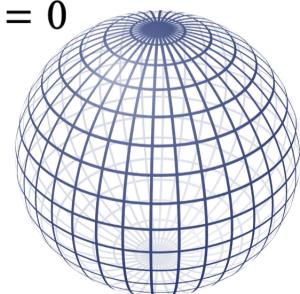
$$(\vec{d} \cdot \vec{d})t^2 + 2\vec{d} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - r^2 = 0$$

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.

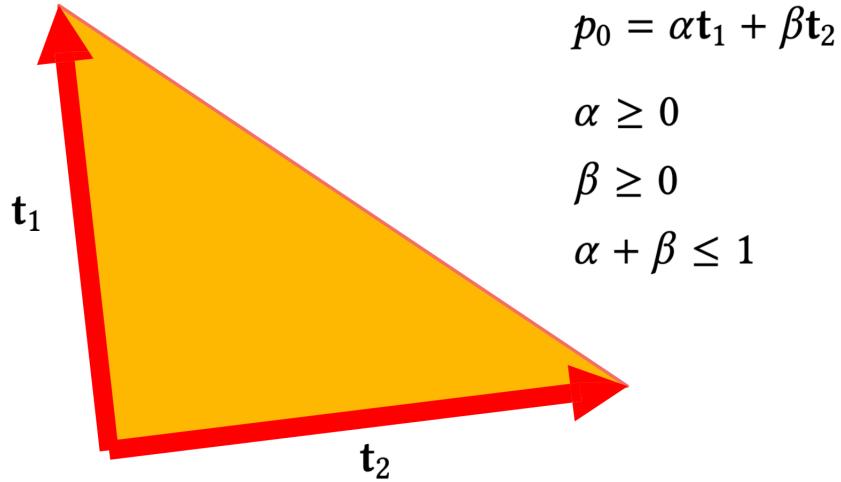


$$\text{intersect} = \mathbf{e} + \mathbf{t} * \mathbf{d} \quad (2.8)$$

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

### 2.4.3 Triangle

## Equations for a Triangle



## Intersection with a Triangle (Parametric Surface)

Check via equating point on surface with point on ray

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

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

Check values of  $\alpha, \beta, t$

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

## 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  $\mathbf{l}$ : a unit vector pointing toward the light source;

View direction  $\mathbf{v}$ : a unit vector pointing toward the eye or camera;

Surface normal  $\mathbf{n}$ : 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  $\mathbf{l}$  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.

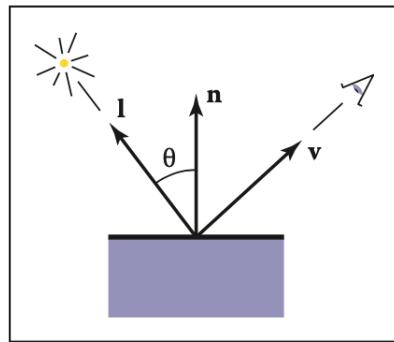


Figure 2: Geometry for Lambertian shading

**Remark 3.1.** Because  $\mathbf{n}$  and  $\mathbf{l}$  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  $\mathbf{v}$  and  $\mathbf{l}$  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  $\mathbf{h}$  with  $\mathbf{n}$ : if  $\mathbf{h}$  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  $p > 1$ .

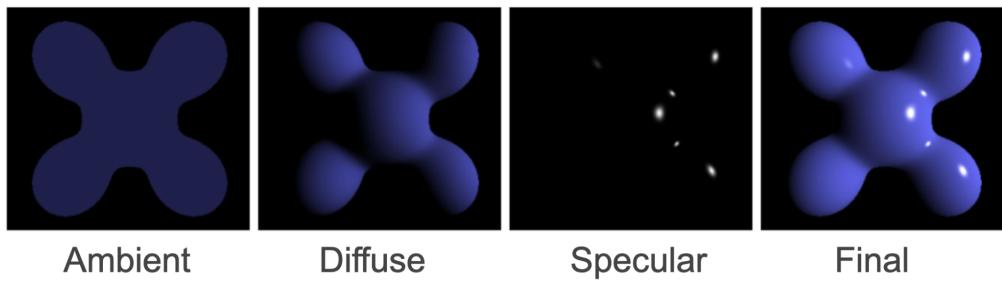
### 3.4.3 Ambient Shading

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  $t \in [0, \infty)$ ) then
        Compute  $\mathbf{n}$ 
        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( $\mathbf{p} + s\mathbf{l}$ ,  $\epsilon$ ,  $\infty$ , srec)) then
        vector3  $\mathbf{h} = \text{normalized}(\text{normalized}(\mathbf{l}) + \text{normalized}(-\mathbf{d}))$ 
         $c = c + \text{rec.}k_d I \max(0, \text{rec.}\mathbf{n} \cdot \mathbf{l}) + (\text{rec.}k_s) I (\text{rec.}\mathbf{n} \cdot \mathbf{h})^{\text{rec.}p}$ 
    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  $\epsilon$  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

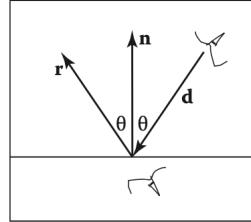


Figure 3: When looking into a perfect mirror, the viewer looking in direction  $\mathbf{d}$  will see whatever the viewer “below” the surface would see in direction  $\mathbf{r}$

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

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:

$$x = x_{\min}$$

$$x = x_{\max}$$

$$y = y_{\min}$$

$$y = y_{\max}$$

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

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

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

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

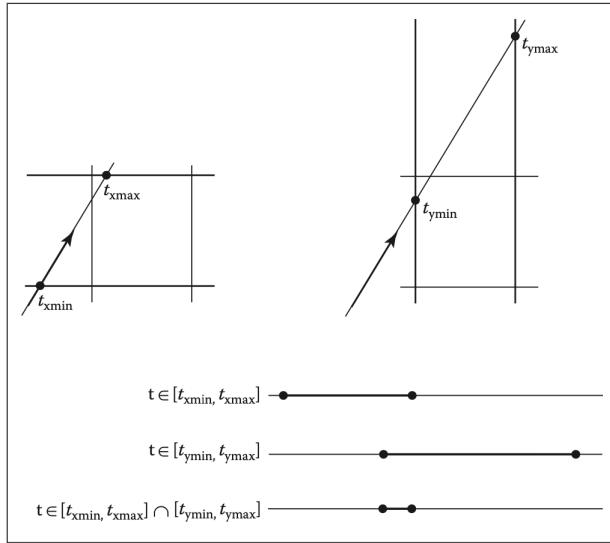
We then make similar computations for  $t_{x\max}$ ,  $t_{y\min}$ , and  $t_{y\max}$ . The ray hits the box if and only if the intervals  $[t_{x\min}, t_{x\max}]$  and  $[t_{y\min}, t_{y\max}]$  overlap.

```

 $t_{x\min} = (x_{\min} - x_e)/x_d$ 
 $t_{x\max} = (x_{\max} - x_e)/x_d$ 
 $t_{y\min} = (y_{\min} - y_e)/y_d$ 
 $t_{y\max} = (y_{\max} - y_e)/y_d$ 
if ( $t_{x\min} > t_{y\max}$ ) or ( $t_{y\min} > t_{x\max}$ ) 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  $t \in [t_{x\min}, t_{x\max}]$ . 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:

```

if ( $x_d \geq 0$ ) then
     $t_{xmin} = (x_{\min} - x_e) / x_d$ 
     $t_{xmax} = (x_{\max} - x_e) / x_d$ 
else
     $t_{xmin} = (x_{\max} - x_e) / x_d$ 
     $t_{xmax} = (x_{\min} - x_e) / x_d$ 

```

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  $a$ ,

$$\begin{aligned} +a/0 &= +\infty \\ -a/0 &= -\infty \end{aligned}$$

```

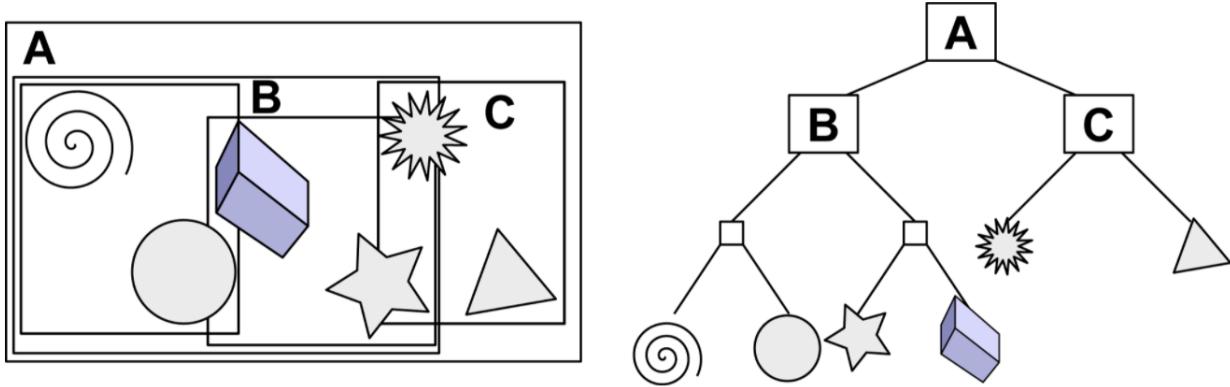
 $a = 1/x_d$ 
if ( $a \geq 0$ ) then
     $t_{\min} = a(x_{\min} - x_e)$ 
     $t_{\max} = a(x_{\max} - x_e)$ 
else
     $t_{\min} = a(x_{\max} - x_e)$ 
     $t_{\max} = a(x_{\min} - x_e)$ 

```

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:**  $\mathcal{O}(n)$

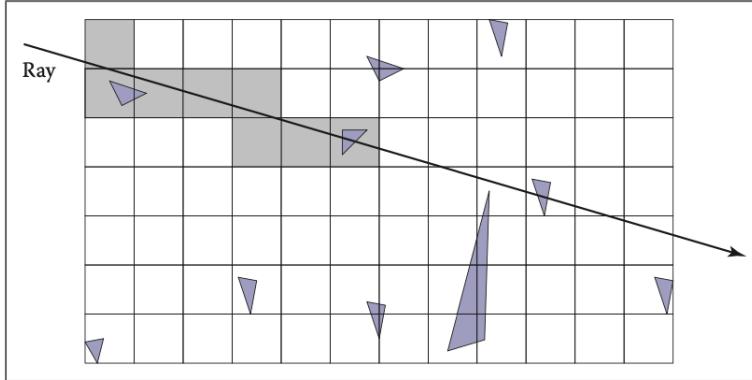
#### 4.4 Object-Oriented Bounding Box (OOBB)

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.

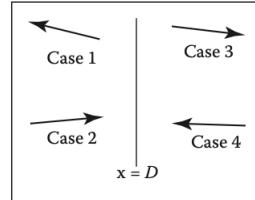
#### 4.6 Axis-Aligned Binary Space Partitioning (BSP Trees)

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 + t d, 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  $t_0$ , real  $t_1$ ,
                           hit-record rec)
     $x_p = x_a + t_0 x_b$ 
    if ( $x_p < D$ ) then
        if ( $x_b < 0$ ) then
            return (left ≠ NULL) and (left→hit(a + tb,  $t_0$ ,  $t_1$ , rec))
         $t = (D - x_a)/x_b$ 
        if ( $t > t_1$ ) then
            return (left ≠ NULL) and (left→hit(a + tb,  $t_0$ ,  $t_1$ , rec))
        if (left ≠ NULL) and (left→hit(a + tb,  $t_0$ ,  $t$ , rec)) then
            return true
        return (right ≠ NULL) and (right→hit(a + tb,  $t$ ,  $t_1$ , rec))
    else
        analogous code for cases 3 and 4

```

## 5 Triangle Meshes

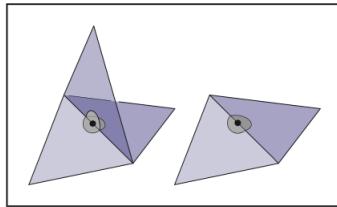
### Motivation

- 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.

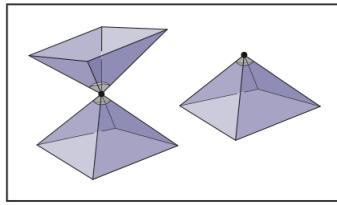
- We'd like to minimize the amount of storage consumed.
- Besides basic operations (storing & drawing), other operations such as *subdivision*, *mesh editing* and *mesh compression*, efficient access to adjacency information is crucial.

## 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.



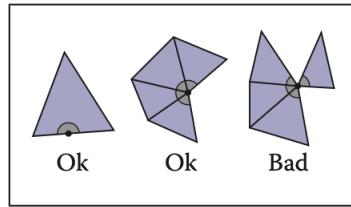
**Figure 12.2.** Non-manifold (left) and manifold (right) interior vertices.

**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:

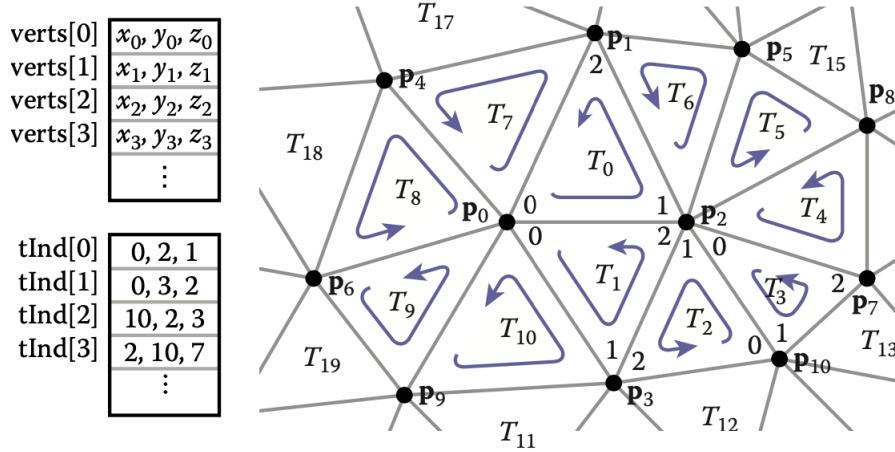
```
Triangle {
    Vertex v[3]
}

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  $k$ th vertex of the  $i$ th triangle is found in  $\text{tInd}[i][k]$ , and the position of that vertex is stored in the corresponding row of the `verts` array.



**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.

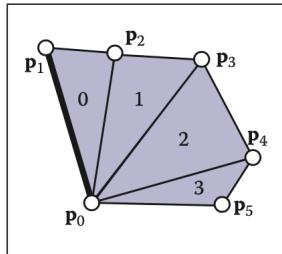


Figure 12.9. A triangle fan.

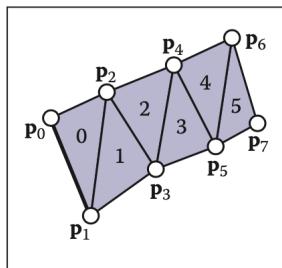


Figure 12.10. A triangle strip.

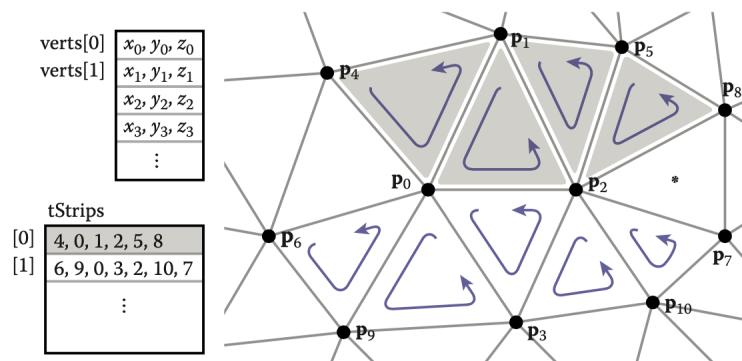


Figure 12.11. Two triangle strips in the context of a larger mesh. Note that neither strip can be extended to include the triangle marked with an asterisk.

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
}

```

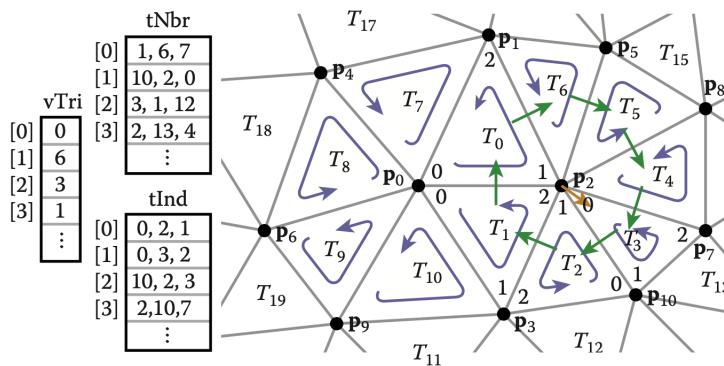
In the array `Triangle.nbr`, the  $k$ th entry points to the neighboring triangle that shares vertices  $k$  and  $k + 1$ .

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  $k$ th 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
}

```

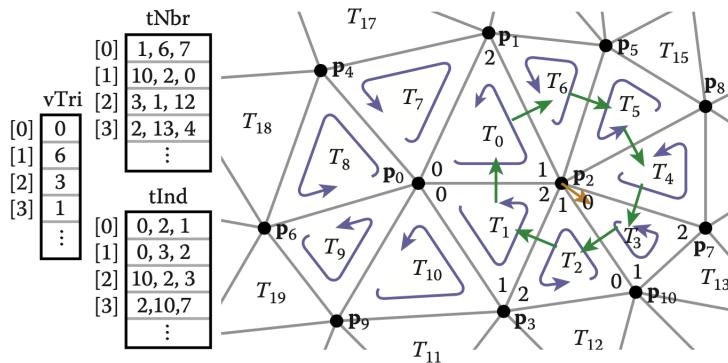
```

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 counterclockwise 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:

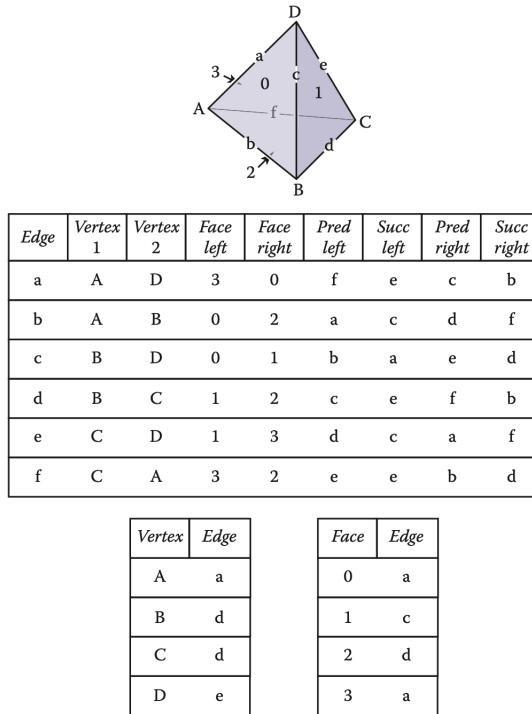


**Figure 12.13.** The triangle-neighbor structure as encoded in arrays, and the sequence that is followed in traversing the neighboring triangles of vertex 2.

```

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
}

```



**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 awkwardness—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.

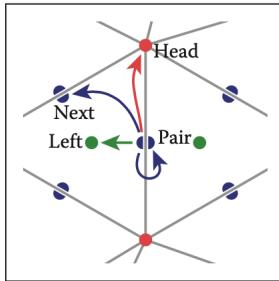
```

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:

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

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:

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

where  $\epsilon$  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”.

**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  $P$ .

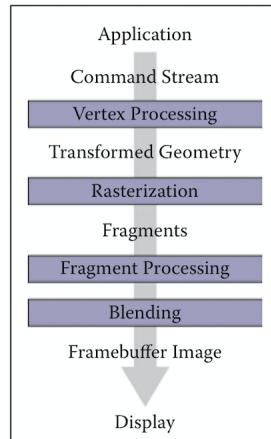
5. Connect up original points to make facets
6. Repeat

If texture coordinates is off in the test, look up textbook p249

## 6 Shader Pipeline

**Definition 6.1** (rasterization). The process of finding all the pixels in an image that are occupied by a geometric primitive is called **rasterization**.

**Definition 6.2** (graphics pipeline). The sequence of operations that is required, starting with objects and ending by updating pixels in the image, is known as the **graphics pipeline**.



**Figure 8.1.** The stages of a graphics pipeline.

**Tessellation control shader** Determines how to subdivide each input patch. The subdivision is determined independently for each triangle and not called recursively.

Unlike the vertex or fragment shader, the tessellation control shader has access to attribute information at all of the vertices of a triangle.

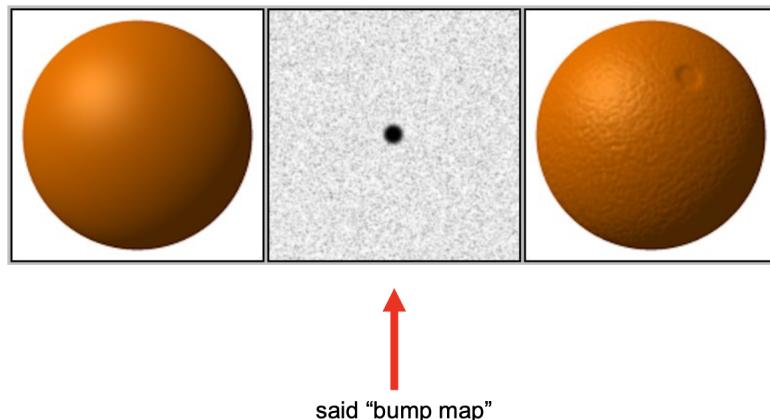
**Tessellation evaluation shader** Takes the result of the tessellation that the “tessellation control shader” has specified. This shader is called once for every vertex output during tessellation. It has access to the attribute information of the original corners and a special variable `gl_TessCoord` containing the barycentric coordinates of the current vertex. Using this information, it is possible to interpolate information stored at the original corners onto the current vertex (e.g. the 3D position).

**Definition 6.3** (Bump map). A **bump map** is a mapping from a surface point to a displacement along the normal direction. Each point  $\mathbf{p} \in \mathbb{R}^3$  on the surface is moved to a new position  $\tilde{\mathbf{p}} \in \mathbb{R}^3$ :

$$\tilde{\mathbf{p}}(\mathbf{p}) := \mathbf{p} + h(\mathbf{p})\hat{\mathbf{n}}(\mathbf{p}) \quad (6.1)$$

where  $h : \mathbb{R}^3 \rightarrow \mathbb{R}$  is the bump height amount function (could be negative) and  $\hat{\mathbf{n}}(\mathbf{p}) : \mathbb{R}^3 \rightarrow \mathbb{R}^3$  is the mathematically correct normal at  $\mathbf{p}$ .

**Remark 6.1.** The idea is that a bump map is a height field: a function that give the local height of the detailed surface above the smooth surface.



**Definition 6.4** (Normal map). A **normal map** is a mapping from a surface point to a unit normal vector.

To create a consistent and believable looking normal map, we can first generate a plausible bump map. If our bump height  $h$  is a smooth function over the surface, we can compute the perceived normal vector  $\tilde{\mathbf{n}}$  by taking a small finite difference of the 3D position:

$$\tilde{\mathbf{n}} = \frac{\partial \mathbf{p}}{\partial T} \times \frac{\partial \mathbf{p}}{\partial B} \approx \left( \frac{\tilde{\mathbf{p}}(\mathbf{p} + \epsilon T) - \tilde{\mathbf{p}}(\mathbf{p})}{\epsilon} \right) \times \left( \frac{\tilde{\mathbf{p}}(\mathbf{p} + \epsilon B) - \tilde{\mathbf{p}}(\mathbf{p})}{\epsilon} \right) \quad (6.2)$$

where  $T, B \in \mathbb{R}^3$  are orthogonal tangent and bi-tangent vectors in the tangent plane at  $\mathbf{p}$  and  $\epsilon$  is a small number (e.g. 0.0001).

We'll make sure that this approximate perceived normal is unit length by dividing by its length:

$$\tilde{\mathbf{n}} \leftarrow \frac{\tilde{\mathbf{n}}}{\|\tilde{\mathbf{n}}\|} \quad (6.3)$$

**Remark 6.2.** Normal mapping is useful in computer graphics because we can drape the appearance of a complex surface on top a low resolution and simple one.

**Remark 6.3.** Normal mapping makes the shading normal depend on values read from a texture map.

**Remark 6.4.** It does not change the surface at all, just a shadowing trick that create the illusion of depth detail on the surface of a model.

A type of gradient noise used to produce natural appearing textures.

1. Grid definition: Define an  $n$ -dimensional grid where each point has a random  $n$ -dimensional unit-length gradient vector.
2. Dot product: An  $n$ -dimensional candidate point can be viewed as falling into an  $n$ -dimensional grid cell. Fetch the  $2^n$  closest gradient values, located at the  $2^n$  corners of the grid cell. Then for each gradient value, compute a distance vector from each corner node to the candidate point. Then compute the dot product between each pair of gradient vector and distance vector. **This function has a value of 0 at the node and a gradient equal to the precomputed node gradient.**
3. Interpolation: Performed using a function that has zero first derivative at the  $2^n$  grid nodes.

If  $n = 1$ , an example of a function that interpolates between value  $a_0$  at grid node 0 and value  $a_1$  at grid node 1 is

$$f(x) = a_0 + \text{smoothstep}(x) \cdot (a_1 - a_0) \quad (6.4)$$

for  $0 \leq x \leq 1$ , where

$$\text{smoothstep}(x) = \begin{cases} 0 & x \leq 0 \\ 3x^2 - 2x^3 & 0 \leq x \leq 1 \\ 1 & x \geq 1 \end{cases} \quad (6.5)$$

**Definition 6.5.** Tangent and bitangent vector Let  $s(x, y, z)$  and  $t(x, y, z)$  be differentiable scalar functions defined at all points on a surface  $S$ . In computer graphics, the functions  $s$  and  $t$  often represent texture coordinates for a 3-dimensional polygonal model. In this context, the **tangent vector**  $T(p_x, p_y, p_z)$  is specifically defined to be the unit vector lying in the tangent plane for which

$$\text{grad}_T t(p_x, p_y, p_z) = 0, \text{grad}_T s(p_x, p_y, p_z) > 0 \quad (6.6)$$

The **bitangent vector**  $B(p_x, p_y, p_z)$  is defined to be the unit vector lying in the tangent plane for which

$$\text{grad}_T s(p_x, p_y, p_z) = 0, \text{grad}_T t(p_x, p_y, p_z) > 0 \quad (6.7)$$

**Remark 6.5.** The vectors  $T$  and  $B$  are not necessarily orthogonal and may not exist for poorly conditioned functions  $s$  and  $t$ .

## 6.1 Rasterization

For each primitive that comes in, the rasterizer has two jobs:

1. It enumerates the pixels that are covered by the primitive.
2. It interpolates values, called attributes, across the primitive.

The output is called **fragments**, one for each pixel covered by the primitive. Each fragment “lives” at a particular pixel and carries its own set of attribute values.

### 6.1.1 Triangle Rasterization

**Definition 6.6** (Gouraud interpolation). If the vertices have colors  $\mathbf{c}_0, \mathbf{c}_1$  and  $\mathbf{c}_2$ , the color at a point in the triangle with barycentric coordinates  $(\alpha, \beta, \gamma)$  is

$$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2 \quad (6.8)$$

We are usually rasterizing triangles that share vertices and edges. To draw them with no holes, we can use the midpoint algorithm to draw the outline of each triangle and then fill in the interior pixels.

Pixels are drawn iff their centers are inside the triangle, i.e., the barycentric coordinates of the pixel center are all in the interval  $(0, 1)$ .

```

 $x_{\min} = \text{floor}(x_i)$ 
 $x_{\max} = \text{ceiling}(x_i)$ 
 $y_{\min} = \text{floor}(y_i)$ 

 $y_{\max} = \text{ceiling}(y_i)$ 
 $f_\alpha = f_{12}(x_0, y_0)$ 
 $f_\beta = f_{20}(x_1, y_1)$ 
 $f_\gamma = f_{01}(x_2, y_2)$ 
 $\text{for } y = y_{\min} \text{ to } y_{\max} \text{ do}$ 
     $\text{for } x = x_{\min} \text{ to } x_{\max} \text{ do}$ 
         $\alpha = f_{12}(x, y)/f_\alpha$ 
         $\beta = f_{20}(x, y)/f_\beta$ 
         $\gamma = f_{01}(x, y)/f_\gamma$ 
         $\text{if } (\alpha \geq 0 \text{ and } \beta \geq 0 \text{ and } \gamma \geq 0) \text{ then}$ 
             $\text{if } (\alpha > 0 \text{ or } f_\alpha f_{12}(-1, -1) > 0) \text{ and}$ 
                 $(\beta > 0 \text{ or } f_\beta f_{20}(-1, -1) > 0) \text{ and}$ 
                 $(\gamma > 0 \text{ or } f_\gamma f_{01}(-1, -1) > 0) \text{ then}$ 
                     $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$ 
                     $\text{drawpixel } (x, y) \text{ with color } \mathbf{c}$ 

```

Figure 6: Algorithm for triangle rasterization.

## 6.2 Operations Before and After Rasterization

### 6.2.1 Vertex-processing stage

1. Incoming vertices are transformed by the modeling, viewing, and projection transformations, mapping them from their original coordinates into screen space.
2. Other information, such as colors, surface normals, or texture coordinates, is transformed as needed.

### 6.2.2 Fragment processing stage

- 1.

### 6.2.3 Blending stage

Combines the fragments generated by the primitives that overlapped each pixel to compute the final color. The most common blending approach is to choose the color of the fragment with the smallest depth (closest to the eye).

**Using a z-buffer for hidden surfaces** At each pixel, we keep track of the distance to the closest surface that has been drawn so far, and we throw away fragments that are farther away than that distance. The closest distance is stored by allocating an extra value for each pixel, in addition to the rgb values, which is known as the depth, or z-value. The **z-buffer** is the name for the grid of depth values.

**Remark 6.6.** Of course there can be ties in the depth test, in which case the order may well matter.

## 7 Kinematics

Study of motion without consideration of what causes that motion.

## 8 Mass-Spring Systems

### 8.1 Bones

**Definition 8.1** (skeleton). A **skeleton** is a tree of rigid bones. Each bone in the skeleton is a UI widget for visualizing and controlling a 3D rigid transformation. A common visualization of 3D bone in computer graphics is a long, pointed pyramid shape.

**Definition 8.2** (Canonical bones). The “canonical bone” of length  $l$  lies along the  $x$ -axis with its “tail” at the origin  $(0, 0, 0)$ , its “tip” at  $(l, 0, 0)$ .

**Twisting** is rotating about the  $x$ -axis in the canonical frame.

**Bending** is rotating about the  $z$ -axis in the canonical frame.

**Remark 8.1.** Composing a twist, bend and another twist spans all possible 3D rotations. We call the three angles composed into a rotation this way, **Euler angles**.

**Definition 8.3** (Rest bones). For each bone, there is a rigid transformation that places its tail and tip to the desired positions in the model.

$$\hat{T} = (\hat{R}\hat{\mathbf{t}}) \in \mathbb{R}^{3 \times 4} \quad (8.1)$$

where  $T$  = transformation,  $R$  = rotation and  $t$  = translation. We use the convention that the “canonical tail” (the origin  $(0, 0, 0)$ ) is mapped to the “rest tail” inside the model. This means that the translation part of the matrix  $\hat{T}$  is simply the rest tail position  $\hat{\mathbf{s}} \in \mathbb{R}^3$ :

$$\hat{\mathbf{s}} = \hat{T} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \hat{R} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} + \hat{\mathbf{t}}_1 = \hat{\mathbf{t}} \quad (8.2)$$

The bone’s rotation is chosen so that the “canonical tip” maps to the “rest tip”  $\hat{\mathbf{d}} \in \mathbb{R}^3$ :

$$\hat{\mathbf{d}} = \hat{T} \begin{pmatrix} l \\ 0 \\ 0 \\ 1 \end{pmatrix} = \hat{R} \begin{pmatrix} l \\ 0 \\ 0 \\ 1 \end{pmatrix} + \hat{\mathbf{t}}_1 = \hat{\mathbf{t}} \quad (8.3)$$

Typically the “rest tail” of a bone is coincident with the “rest tip” of its parent (if it exists)

$$\hat{\mathbf{d}}_p = \hat{\mathbf{s}} \quad (8.4)$$

**Remark 8.2.** The **pose** of each bone will be measured relative to the “rest bone”.

**Definition 8.4** (Pose bones). In general, each rest bone undergoes a rigid transformation  $T \in \mathbb{R}^{3 \times 4}$ , composed of a rotation  $R \in \mathbb{R}^{3 \times 3}$  and a translation  $\mathbf{t} \in \mathbb{R}^3$ , mapping each of its rest points  $\hat{x} \in \mathbb{R}^3$  to its corresponding posed position  $\mathbf{x} \in \mathbb{R}^3$ :

$$\mathbf{x} = T\hat{\mathbf{x}} \quad (8.5)$$

**Remark 8.3.** In particular, we would like each bone to rotate about its parent’s tip, but this position is determined by the parent’s pose transformation  $T_p$ , which in turn should rotate about the grandparent’s tip and so on.

## 8.2 Forward Kinematics

For each bone  $i$  in a skeleton, we aggregate relative rotations  $\bar{R}_i \in \mathbb{R}^{3 \times 3}$  between a bone  $i$  and its parent bone  $p_i$  in the skeletal tree.

For each bone:

1. First undo the map from canonical to rest (inverting  $\hat{T}_i$ );
2. Then rotate by the relative rotation  $\bar{R}_i$ ;
3. Then map back to rest (via  $\hat{T}_i$ );
4. Continue up the tree recursively applying the parent’s relative transformation.

$$T_i = T_{p_i}(\hat{T}_i) \quad (8.6)$$

$$\mathbf{T}_i = \mathbf{T}_{p_i} \begin{pmatrix} \hat{\mathbf{T}}_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ \bar{\mathbf{R}}_i \\ 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{\mathbf{T}}_i \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

**Definition 8.5** (Linear Blend Skinning). Skinning is the process of defining how bones are attached to the meshes. In linear blend skinning, each vertex  $i$  of the mesh is assigned a weight  $w_{i,j}$  for each bone  $j$  corresponding to how much it is “attached” to that bone on a scale of 0% to 100%:

$$\mathbf{v}_i = \sum_{j=1}^m w_{i,j} T_j \begin{pmatrix} \hat{\mathbf{v}}_i \\ 1 \end{pmatrix} \quad (8.7)$$

## 8.3 Keyframe animation

To create a long animation, specifying three Euler angles for every bone for every frame manually would be too difficult. The standard way to determine the relative bone transformations for each frame is to interpolate values specified at a small number of “key” times during the animation.

**Definition 8.6** (Catmull-Rom interpolation).  $\theta = \mathbf{c}(t)$  is a curve in the pose space. A **cubic Hermite spline** is a spline where each piece is a third-degree polynomial specified by its values and first derivatives.

incomplete

---

## 8.4 Inverse Kinematics

---

A6

## 8.5 Projected Gradient Descent

A6

## 8.6 Line Search

A6

# 9 Computational Fabrication