Image Resampling

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

• In this lecture we review image resampling:
  - Ideal resampling
  - Mathematical formulation
  - Resampling filter
  - Tradeoffs between accuracy and complexity
  - Software implementation
Definition

- Image Resampling: Process of transformation a sampled image from one coordinate system to another.
Ideal Resampling

- Ideal Resampling: reconstruction, warping, prefiltering, sampling
Mathematical Formulation (1)

<table>
<thead>
<tr>
<th>Stages:</th>
<th>Math Definition:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete Input</td>
<td>$f(u), u \in \mathbb{Z}$</td>
</tr>
<tr>
<td>Reconstruction Input</td>
<td>$f_c = f(u) * r(u) = \sum_{k \in \mathbb{Z}} f(k)r(u-k)$</td>
</tr>
<tr>
<td>Warped Signal</td>
<td>$g_c(x) = f_c(m^{-1}(x))$</td>
</tr>
<tr>
<td>Continuous Output</td>
<td>$g_c^\prime(x) = g_c(x)*h(x) = \int g_c(t)h(x-t)dt$</td>
</tr>
<tr>
<td>Discrete Output</td>
<td>$g(x) = g_c^\prime(x)S(x)$</td>
</tr>
</tbody>
</table>

Two filtering components: reconstruction and prefiltering (bandlimiting warped signal before sampling). Cascade them into one by working backwards from $g(x)$ to $f(u)$:

$g(x) = g_c^\prime(x)$ for $x \in \mathbb{Z}$

$g(x) = \int f_c(m^{-1}(t))h(x-t)dt = \int \left[ \sum_{k \in \mathbb{Z}} f(k)r(m^{-1}(t) - k) \right] h(x-t)dt$

$g(x) = \sum_{k \in \mathbb{Z}} f(k)\rho(x,k)$ where $\rho(x,k) = \int r(m^{-1}(t) - k)h(x-t)dt$
Mathematical Formulation (2)

\[ \rho(x, k) = \int r(m^{-1}(t) - k)h(x - t)dt \]

Selects filter response used to index filter coefficients

Spatially varying resampling filter expressed in terms of output space

(Wide)

(Narrow)
Mathematical Formulation (3)

• We can express $\rho(x,k)$ in terms of input space:

Let $t = m(u)$, we have

$$
\rho(x, k) = \int r(u - k) h(x - m(u)) \left| \frac{\partial m}{\partial u} \right| dt
$$

where $\left| \frac{\partial m}{\partial u} \right|$ is the determinant of Jacobian matrix:

1D: $\left| \frac{\partial m}{\partial u} \right| = \frac{dm}{du} \quad 2D: \left| \frac{\partial m}{\partial u} \right| = \begin{vmatrix} x_u & x_v \\ y_u & y_v \end{vmatrix}$ where $x_u = \frac{\partial x}{\partial u}$
Resampling Filter

Linear warps (space invariant):
\[
\rho(x,k) = h'(u) \ast r(u) = [J|h(uJ)|] \ast r(u)
\]
\[
\rho_{\text{mag}}(x,k) = r(u): \quad \text{Shape of } r(u) \text{ remains the same, independently of } m(u), \text{ (independently of scale factor)}
\]
\[
\rho_{\text{min}}(x,k) = |J|h(uJ)|: \quad \text{Shape of prefilter is based on desired frequency response characteristic (performance in passband and stopband). Unlike } r(u), \text{ though, the prefilter must be scaled proportional to the minification factor (broader and shorter for more minification)}
\]
Fourier Transform Pairs

• The shape of $\rho_{min}$ is a direct consequence of the reciprocal relation between the spatial and frequency domains.

$$\leftrightarrow \text{ denotes Fourier Transform pair}$$

$$H(u) = \int h(u) e^{-i2\pi fu} \, du$$

$$H(m(u)) = \int h(m(u)) e^{-i2\pi fu} \, du$$

Let $x = au = m(u)$ and $dx = \left| \frac{\partial m}{\partial u} \right| \, du$

$$H(m(u)) = \int h(x) e^{-i2\pi m^{-1}(x)} \frac{dx}{\left| \frac{\partial m}{\partial u} \right|}$$

where $m^{-1}(x) = \frac{x}{a}$; $\left| \frac{\partial m}{\partial u} \right| = \left| J \right| = a$

$$h(au) \leftrightarrow \frac{1}{a} \int h(x) e^{-i2\pi \frac{fx}{a}} \, dx$$

$$h(au) \leftrightarrow \frac{1}{a} H\left(\frac{f}{a}\right)$$
Reciprocal Relationship

Intuition: \( f = \frac{1}{T} \)

Consequences: narrow filters in spatial domain (desirable) yield wide frequency spectrums (undesirable). Tradeoff between accuracy and complexity.
Software (1)

```c
resample1D(IN, OUT, INlen, OUTlen, filtertype, offset)
unsigned char *IN, *OUT;
int INlen, OUTlen, filtertype, offset;
{
    int i;
    int left, right;       // kernel extent in input
    int pixel;             // input pixel value
    double u, x;           // input (u), output (x)
    double scale;          // resampling scale factor
    double (*filter)();    // pointer to filter fct
    double fwidth;         // filter width (support)
    double fscale;         // filter amplitude scale
    double weight;         // kernel weight
    double acc;            // convolution accumulator

    scale = (double) OUTlen / INlen;
```
switch(filtertype) {
  case 0: filter = boxFilter; // box filter (nearest nbr)
      fwidth = .5;
      break;
  case 1: filter = triFilter; // triangle filter (lin intrp)
      fwidth = 1;
      break;
  case 2: filter = cubicConv; // cubic convolution filter
      fwidth = 2;
      break;
  case 3: filter = lanczos3; // Lanczos3 windowed sinc fct
      fwidth = 3;
      break;
  case 4: filter = hann4; // Hann windowed sinc function
      fwidth = 4;
      // 8-point kernel
      break;
}
if(scale < 1.0) {  // minification: h(x) -> h(x*scale)*scale
    fwidth = fwidth / scale;  // broaden filter
    fscale = scale;  // lower amplitude

    /* roundoff fwidth to int to avoid intensity modulation */
    if(filtertype == 0) {
        fwidth = CEILING(fwidth);
        fscale = 1.0 / (2*fwidth);
    }
} else fscale = 1.0;

// project each output pixel to input, center kernel, and convolve
for(x=0; x<OUTlen; x++) {
    /* map output x to input u: inverse mapping */
    u = x / scale;

    /* left and right extent of kernel centered at u */
    if(u - fwidth < 0) {
        left = FLOOR (u - fwidth);
    } else
        left = CEILING(u - fwidth);

    right = FLOOR(u + fwidth);
/* reset acc for collecting convolution products */
acc = 0;

/* weigh input pixels around u with kernel */
for(i=left; i <= right; i++) {
    pixel = IN[ CLAMP(i, 0, INlen-1)*offset];
    weight = (*filter)((u - i) * fscale);
    acc   += (pixel * weight);
}

/* assign weighted accumulator to OUT */
OUT[x*offset] = acc * fscale;
double boxFilter(double t)
{
    if((t > -.5) && (t <= .5)) return(1.0);
    return(0.0);
}

double triFilter(double t)
{
    if(t < 0) t = -t;
    if(t < 1.0) return(1.0 - t);
    return(0.0);
}
double cubicConv(double t)
{
    double A, t2, t3;

    if(t < 0) t = -t;
    t2 = t * t;
    t3 = t2 * t;

    A = -1.0;  // user-specified free parameter
    if(t < 1.0) return((A+2)*t3 - (A+3)*t2 + 1);
    if(t < 2.0) return(A*(t3 - 5*t2 + 8*t - 4));
    return(0.0);
}
Software (7)

double sinc(double t)
{
    t *= PI;
    if(t != 0) return(sin(t) / t);
    return(1.0);
}

double lanczos3(double t)
{
    if(t < 0) t = -t;
    if(t < 3.0) return(sinc(t) * sinc(t/3.0));
    return(0.0);
}
double hann4(double t)
{
    int N = 4;  // fixed filter width
    if(t < 0) t = -t;
    if(t < N)
        return(sinc(t) * (.5 + .5*cos(PI*t / N)));
    return(0.0);
}