Bitmap (Raster) Images

Multimedia and Computer Graphics

University of Leicester
Overview of Lectures on Images

Part I: Image Transformations
- Examples of images; key attributes/properties.
- The standard computer representations of color.
- Coordinate Geometry: transforming positions.
- Position Transformation in Java.
- And/Or Bit Logic: transforming Color.
- Color Transformation in Java.

Part II: Image Dithering
- Basic Dithering.
- Expansive Dithering.
- Ordered Dithering.
- Example Programs.
Examples of Images
Examples of Images
Examples of Images
Attributes of Images

- An image is a (finite, 2-dimensional) array of colors $c$.
- The $(x, y)$ position, an image coordinate, along with its color, is a pixel (eg $p = ((x, y), c)$).
- $x_{\text{max}} + 1$ is the width and $y_{\text{max}} + 1$ is the height.

We study these types of images:
- 1-bit
  - $2^1$ colors: black and white; $c \in \{0, 1\}$
- 8-bit grayscale
  - $2^8$ colors: grays; $c \in \{0, 1, 2, \ldots, 255\}$
- 24-bit color (RGB)
  - $2^{24}$ colors: see later on . . .
- others . . .
1-Bit Images

- A pixel in a 1-bit image has a color selected from one of \(2^1\), that is, *two* “colors”, \(c \in \{0, 1\}\). Typically 0 indicates *black* and 1 *white*.

- The *(idealised!)* memory size of a 1-bit image is

\[
\frac{\text{height} \times \text{width}}{8} \quad \text{bytes}
\]
8-Bit Grayscale Images

A pixel in an 8-bit (grayscale) image has a color selected from one of \(2^8 = 256\) colors (which denote shades of gray). Each color \(c\) is a computer representation of an integer \(0 \leq c \leq 255\). The (minimal) memory required is a byte.

- color 20
- color 125
- color 232
(Recall) Hexadecimal

- Integers are represented as (finite) sequences of digits; each digit selected from the set \( \{0, \ldots, 9, a, b, c, d, e, f\} \). For example \( 0x: 2b1f \), where \( 0x: \) indicates Hex.

- The symbol \( s \) in position \( i \) denotes \( s \times 16^i \) where \( a = 10, b = 11, c = 12, d = 13, e = 14, f = 15 \).

- The sequence of digits \( d_{n-1} \ldots d_0 \) denotes the integer

\[
\sum_{\text{for } i=n-1 \text{ downto } i=0} d_i \times 16^i = d_{n-1} \times 16^{n-1} + \ldots + d_1 \times 16^1 + d_0
\]

- \( 0x: 2b1f \) denotes \( 2 \times 16^3 + 11 \times 16^2 + 1 \times 16^1 + 15 \times 16^0 = \ldots \)

- IMPORTANT FACT: 8-digit binary numbers correspond exactly to 2-digit hex numbers—they represent the same integers.
24-bit Color Images

- A pixel in a 24-bit color image has a color selected from $2^{24} = 16777216$ colors. Each color $c$ is a computer representation of an integer $0 \leq c \leq 16777215$. The (minimal) memory required is 24 bits, that is, 3 bytes.

- The representation is composed out of a Red, Green and Blue component, each component represented as one of the three bytes—hence this is often called RGB color.

- An example: $00011101\,11010101\,11111101\,0..255\,0..255\,0..255$

- White is $0xff\,ff\,ff\,ff\,ff\,ff$; pure red is $0xff\,00\,00\,00\,00\,00$; pure green is $0x00\,ff\,ff\,ff\,ff\,ff$; pure blue is $0x00\,00\,00\,ff\,ff\,ff$; black is $0x00\,00\,00\,00\,00\,00$. 

24-bit Color Images

- The uncompressed size of a 24-bit color image is
  \[ \text{width} \times \text{height} \times 3 \text{ bytes} \]

So a $512 \times 512$ 24-bit image requires (at least) 768 kilobytes of storage without any compression.

- Many 24-bit color images are actually stored as 32-bit images, with the extra byte of data for storing an $\alpha$ value representing special information. This $\alpha$ component is (sometimes) used to encode “transparency” information of the pixel.

- The complete pixel data, 8 bits for $\alpha$ and 24 bits for colour, is often stored as a 32-bit integer.
Each pixel has one of $2^8$ colors. Each integer from 0 to 255, denoted by one of the 256 possible 8-bit binary numbers, is used to pick one of 256 different RGB colors from a color lookup table.

Each 8-bit color image is composed from these 256 different colors.
The RGB Model of Color in Java

In the RGB model, colors are stored as *32-bit integers* and we have

- for 8-bit grayscale:

  \[ \text{int } \vec{\text{gray}} \in \mathbb{B}^8, \text{ gray} \in \mathbb{B}^8, \text{ gray} \in \mathbb{B}^8, \text{ gray} \in \mathbb{B}^8. \]

- similarly for 24-bit color and 32-bit color:

  \[ \text{int } \alpha.\text{red}.\text{green}.\text{blue} \]

and these values can be obtained with the following methods (try checking this in the dither examples):
Key methods are

- `img.getRGB(int x, int y)`
  get color of pixel at \((x, y)\)

- `img.setRGB(int x, int y, int col)`
  set color of pixel at \((x, y)\) to \(col\)

- `img.getWidth()`
  NB width is \(x_{Max} + 1\)

- `img.getHeight()`
  NB width is \(y_{Max} + 1\)

for an image `img`.
Coordinate Geometry

- To perform transformations of images, we change from *image coordinates* to *cartesian coordinates*.
- Java 2D and 3D use cartesian coordinates.
- The image coordinates \((i, j)\) correspond to \((i, -j)\) in cartesian coordinates.
- Transformations are often specified by *continuous* functions \(f(x)\) where \(x\) might be a color or a coordinate(s).

( In the coursework we use linear functions \(f\). Such functions take the form \(f(x) = mx + k\). CW1 works with \(m = (P - D)/(O - D)\) and \(k = D \ast (O - P)/(O - D)\) and \(f\) is called *linTrans* (or similar). )
Coordinate Geometry

We will also use some basic trigonometry:

- \( \sin \theta = \frac{o}{h} \) with inverse \( \arcsin \)
- \( \cos \theta = \frac{a}{h} \) with inverse \( \arccos \)
- \( \tan \theta = \frac{o}{a} \) with inverse \( \arctan \)
- The distance of \((I, J)\) from origin \((0, 0)\) is \(\sqrt{I^2 + J^2}\)
Pixel Position Transformations in Java

Suppose a transformation “moves” a pixel \((I,J), c\) in \(img\) to position \((mI,mJ)\): the pixel at \((mI,mJ)\) in \(img\) is updated with color \(c\).

To implement this we might make a copy \(temp\) of \(img\) and for each \((I,J)\) in \(img\) do

\[
\text{temp.setRGB}((mI,mJ), \text{img.getRGB}(I,J))
\]

and return \(temp\), where there is a function \(g\) such that \((mI,mJ) = g(I,J)\).

This is problematic. If \(g\) is continuous we may get rounding errors: the \((mI,mJ)\) may not range over every pixel of \(temp\). These problems are non-examinable!!
Pixel Position Transformations in Java

In fact for every \((I,J)\) in \(img\) we compute \((preI,preJ)\) such that \(g\) “moves” the pixel at \((preI,preJ)\) to \((I,J)\) and do

\[
\text{img.setRGB}((I,J), \text{temp.getRGB}(preI, preJ))
\]

We call \((preI,preJ)\) the \textit{preimage} of \((I,J)\) where \(g(preI,preJ) = (I,J)\).

Since we wish to compute \((preI,preJ)\) from \((I,J)\) we implement \(g^{-1}\):

\[
(preI,preJ) = g^{-1} (I,J)
\]

(The \textit{linTrans} functions in the coursework are examples of the \(g^{-1}\).)
Pixel Position Transformations in Java

- Note that we visit *every* pixel \((I, J)\) of \(\text{img}\) and update its color.

- This is a flexible method; eg if we want a pixel \((A, B)\) to be blue, as a special case, we can do
  \[
  \text{img.setRGB((A, B), 0xff)}
  \]
  with \(0xff\) replacing \(\text{temp.getRGB(preI, preJ)}\).

- In a typical image rounding errors are not a problem, *since (preimage) pixels close to each other are likely to have the same color!*
Given binary digits (Booleans) $b, b' \in \mathbb{B}$ then logical AND is written $b \& \& b' \in B$ and logical OR is $b \mid \mid b' \in \mathbb{B}$.

Given binary numbers $\vec{b}, \vec{b}'$ then bitwise logical AND is written $\vec{b} \& \vec{b}'$ and bitwise logical OR is $\vec{b} \mid \vec{b}'$.

Given binary numbers $\vec{b}$ and $n \in \mathbb{N}$ then shiftleft is written $\vec{b} \ll n$, and shiftright is written $\vec{b} \gg n$.

E.g. $1111000011110101 \gg 4 = 0000111110001111$.

We can use these logical operations to extract color components from RGB colors, and to build new RGB colors.
In Java, inputs typically will be length 32 (for integers) or length 8 (for bytes).

**Warning**: We can do bitwise operations on binary numbers of different length! The shorter number is *sign extended* to the length of the longer number. E.g.

Given binary numbers $\vec{b} = 10101010 \in \mathbb{B}^8$ and $\vec{b}' = 11111111.00000000.11110000.10101101 \in \mathbb{B}^{32}$ then

$\vec{b} \& \vec{b}' = 11111111.11111111.11111111.10101010 \&$

$11111111.00000000.11110000.10101101$

$= 11111111.00000000.11110000.10101000$
A Java fragment to convert an RGB color into its components

```java
int red, green, blue, col
...
blue = (col & 0xff);
green = (col & 0xff00) >> 8;
red = (col & 0xff0000) >> 16;
```

And vice versa from the components to an RGB color

```java
col = red << 16 | green << 8 | blue;
// or alternatively
col = red * 16^4 + green * 16^2 + blue;
```
Reading Images in Java

- In practice, often read in an image file to a variable `img` of type `BufferedImage` (a subclass of `Image`): Java gives us a “standardised model” of image data. For the “color” image data this is the RGB model.

- We should specify the correct `imageType` (for the image to be input), such as `TYPE_BYTE_BINARY` (say for inputting an 8-bit grayscale) or `TYPE_INT_RGB` (for inputting an 24-bit RGB color image).

- Try reading about buffered images and image types in the Java API documentation. You do not need to know the details for coursework or examination, but some reading will give you a better overall understanding.
Pixel Color Transformations in Java: Split RGB Program

```java
class SplitRGB {
    private BufferedImage filter(BufferedImage img, int choice) {
        BufferedImage ans = new BufferedImage(
            img.getWidth(), img.getHeight(),
            BufferedImage.TYPE_INT_RGB);

        int graylvl;
        for (int x = 0; x < img.getWidth(); x++) {
            for (int y = 0; y < img.getHeight(); y++) {
                switch (choice) {
                    case BLUE:
                        graylvl = (img.getRGB(x, y) & 0xff);
                        ans.setRGB(x, y, graylvl);
                        break;
                    case GREEN:
                        graylvl = (img.getRGB(x, y) & 0xff00);
                        ans.setRGB(x, y, graylvl);
                        break;
                    case RED:
                        graylvl = (img.getRGB(x, y) & 0xff0000);
                        ans.setRGB(x, y, graylvl);
                        break;
                }
            }
        }

        return ans;
    }
}
```

Pixel Color Transformations in Java: Split Into Color Components
Pixel Color Transformations in Java: Split Into Grays
Image Compression and Dithering

*Compression* is the process of transforming an image into a new image that is *smaller* but whose *quality* is the same, or only slightly poorer, than the original.

*Dithering* is the process of transforming an image into a new image that has *fewer colors* but whose *quality* is representative of, but typically rather worse than, the original.

Exercise: think about exactly what *smaller* and *quality* might mean. Note: this is more subtle than you might at first think.
Basic Dithering from 8 to 1-bit

- How do we dither an 8-bit grayscale image to a 1-bit image?
- A very simple idea:
  A dark gray pixel color in the original image is mapped to black and a light gray pixel color to white.
- Recall black and white are represented by $c \in \{0, 1\}$.
- Recall grays are represented by $c \in \{0, 1, 2, \ldots, 255\}$.
- So light grays are in the range $128 \ldots 255$, that is, $> 127$
Basic Dithering Algorithm

begin
  for x = 0 to x_max
    for y = 0 to y_max
      if (OriginalImageColor(x,y) > 127 )
        DitheredImageColor(x,y) = 1; // White !!
      else
        DitheredImageColor(x,y) = 0; // Black !!
  end
end
Expansive Dithering

Can we do better?

By allowing the size of the dithered image to be bigger than the original, we can “preserve more of the original image”. Such a dithered image is a better quality than the simple dithered image.

Each pixel in the original image will correspond to 4 pixels (2 x 2) in the new image. Note all original pixels are 8-bit and all new ones are 1-bit pixels.

Depending on the darkness of the original pixel the resulting four pixels (called a 4-pixel gray) contain either \( il = 0, 1, 2, 3, 4 \) white pixels (the other ones are black) in a random arrangement. We call \( il \) the intensity level.
Principle of Expansive Dithering

First, linearly map the grayscale “colors” 0..255 into the intensities 0..4:

<table>
<thead>
<tr>
<th>grayscale value</th>
<th>intensity level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..51</td>
<td>0</td>
</tr>
<tr>
<td>52..102</td>
<td>1</td>
</tr>
<tr>
<td>103..153</td>
<td>2</td>
</tr>
<tr>
<td>154..204</td>
<td>3</td>
</tr>
<tr>
<td>205..255</td>
<td>4</td>
</tr>
</tbody>
</table>
Principle of Expansive Dithering

Then, map the intensities into “4-pixel grays” . . . refer to lecture explanations!

\[ il \]

\[
\begin{align*}
0 & \mapsto B\ B\ B\ B \\
1 & \mapsto W\ B\ B\ B & \text{any permutation} \\
2 & \mapsto W\ W\ B\ B & \text{any permutation} \\
3 & \mapsto W\ W\ W\ B & \text{any permutation} \\
4 & \mapsto W\ W\ W\ W
\end{align*}
\]

Given the original image, for each intensity, a *fixed choice of* permutation 4-pixel gray is chosen. *Why?*
Principle of Expansive Dithering

There is a cunning way in which to compute such 4-pixel grays . . . . Think of $B$ as falsity and $W$ as truth!

$$dm(i, j)$$

$$\begin{array}{c|cccc}
    il & 0 & 1 & 2 & 3 \\
    \hline
    0 & \rightarrow & B & B & B & B \\
    1 & \rightarrow & W & B & B & B \\
    2 & \rightarrow & W & W & B & B \\
    3 & \rightarrow & W & W & W & B \\
    4 & \rightarrow & W & W & W & W
\end{array}$$

It is intuitive to arrange the values $dm(i, j) = 0, 1, 2, 3$ from the $il > dm(i, j)$ computations as a $2 \times 2$ dithering matrix.
A 2 × 2 Example

Example of a 2 x 2 dithering matrix \( \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix} \). Each pixel of the original image yields an intensity \( i_l \). We then “map” \( e \mapsto (i_l > e) \) “over the matrix elements \( e \)” to obtain one 4-pixel gray from each pixel \( i_l \) (see Step 2 code):

<table>
<thead>
<tr>
<th>grayscale value</th>
<th>intensity level</th>
<th>4-pixel gray</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..51</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>52..102</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>103..153</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>104..204</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>205..255</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Final Observations on Expansive Dithering

- An $n \times n$ dithering matrix can represent $n^2 + 1$ levels of intensity.

- The new image created by an $n \times n$ matrix used for expansive dithering is $n$ times wider and $n$ times higher than the original. So the new image is $n^2$ larger than the original one.
Final Observations on Expansive Dithering

- Note that $il = \text{(int)}(((n^2 + 1)/256) \times gs)$. Why?
  - Try drawing line $y = f(x) = mx + k$ where $k = 0$ and $m = (n^2 + 1)/256$ and $x = gs$. Then draw a picture of the effect of Java (int) coercion to understand computation of $il$.

- Example of $4 \times 4$ dithering matrix (17 intensity levels, $il = 0 \ldots 16$)

\[
\begin{pmatrix}
0 & 8 & 2 & 10 \\
12 & 4 & 14 & 6 \\
3 & 11 & 1 & 9 \\
15 & 7 & 13 & 5 \\
\end{pmatrix}
\]
Ordered Dithering

- We now perform 8-bit to 1-bit image dithering which uses a $n \times n$ dithering matrix, but the output size equals that of the input.
- First map each pixel gray-color to its intensity.
- By sliding the dithering matrix over the image ($n$ pixels in the horizontal and vertical direction at a time) each pixel has a corresponding entry in the dithering matrix.
- A pixel with intensity level higher than the corresponding dithering matrix entry is mapped to a white pixel and otherwise a black pixel.
- This technique is called ordered dithering.
Ordered Dithering Example

<table>
<thead>
<tr>
<th>120</th>
<th>110</th>
<th>160</th>
<th>180</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>75</td>
<td>120</td>
<td>130</td>
</tr>
<tr>
<td>250</td>
<td>220</td>
<td>75</td>
<td>170</td>
</tr>
<tr>
<td>120</td>
<td>30</td>
<td>30</td>
<td>75</td>
</tr>
</tbody>
</table>

Dithering matrix: \[
\begin{pmatrix}
3 & 1 \\
0 & 2
\end{pmatrix}
\]

Image:

Result:

Why does ordered dithering work?
The ordered dither algorithm:

```plaintext
for x = 0 to x_max
    for y = 0 to y_max
        // note row i correspond to coordinate y !!!!
        i = y mod n
        j = x mod n
        if IntensityLevelOf_OriginalImageColor(x, y) > DM(i, j)
            DitheredImageColor(x, y) = 1
        else
            DitheredImageColor(x, y) = 0
```
Basic Dithering Program (Step 1)

```java
private BufferedImage basicDither (BufferedImage img, int b) {
    BufferedImage ans = new BufferedImage (
        img.getWidth(), img.getHeight(),
        BufferedImage.TYPE_BYTE_BINARY);

    for ( int i = 0; i < img.getWidth(); i ++ ) {
        for ( int j = 0; j < img.getHeight(); j ++ ) {
            // select 8-bit gray data
            int intensityLevel = ( img.getRGB( x, y ) & 0xff );
            if ( intensityLevel > b )
                ans.setRGB( i, j, 0xffffff ); // set output color to white
            else
                ans.setRGB( i, j, 0x000000 ); // set output color to black
        }
    }

    return ans;
}
```
private BufferedImage expansiveDither(BufferedImage img, int [][] dm) {
    int n = dm.length; int intensityLevel;
    BufferedImage ans = new BufferedImage(
        n*img.getWidth(), n*img.getHeight(),
        BufferedImage.TYPE_BYTE_BINARY);

    for ( int x=0; x<img.getWidth(); x++ ) {
        for ( int y=0; y<img.getHeight(); y++ ) {
            // select 8-bit gray data; linearly map to 0 to n*n
            intensityLevel = (int)((img.getRGB(x,y) & 0xff)*(n*n+1)/256));

            for ( int i=0; i<n; ++i ) {
                for ( int j=0; j<n; ++j ) {
                    if ( intensityLevel > dm[i][j] )
                        ans.setRGB(n*x+i, n*y+j, 0xffffff);
                    else
                        ans.setRGB(n*x+i, n*y+j, 0x000000);
                }
            }
        }
    }

    return ans;
}
```java
private BufferedImage orderedDither (BufferedImage img, int[][] dm) {
    BufferedImage ans = new BufferedImage(
        img.getWidth(), img.getHeight(),
        BufferedImage.TYPE_BYTE_BINARY);

    int i, j;
    int n = dm.length;
    for (int x=0; x<img.getWidth(); x++) {
        for (int y=0; y<img.getHeight(); y++) {
            int intensityLevel = (int)((img.getRGB(x, y) & 0xff) * ((n*n+1)/256));
            // why would i = x%n; j = y%n; still yield a correct program?
            i = y%n; j = x%n;
            if (intensityLevel > dm[i][j])
                ans.setRGB(x, y, 0xffffff);
            else
                ans.setRGB(x, y, 0x000000);
        }
    }

    return ans;
}
```
Further Topics

- dithering from 8 to 4 bits
- dithering on color images
- resizing
- gamma correction
- compression
More resources

- Java Documentation