So far, we only concerned with the rendering of geometric data. Two other important classes of data:

- Bitmaps, typically used for characters in fonts (bilevel)

- Image data, which might have been scanned in or calculated (multiple-level)
Current Raster Position (CRP)

- OpenGL maintains the *current raster position*, a 3D-position in world coordinate
- Modified by:
  - `glRasterPos()`
    - specify the object coordinates (as in `glVertex`); transformed by MODELVIEW & PROJECTION, passed to the clipping stage.
    - If the position is not culled, raster position is updated; otherwise, it is not valid.
  - `glBitmap()`
- `glGetFloatv(GL_CURRENT_RASTER_POSITION, fptr)` to obtain the CRP
- `glGetBooleanv(GL_CURRENT_RASTER_POSITION_VALID, &boolvar)` to test validity of CRP
Bitmaps and Characters

• The most common use of bitmaps for characters

• The commands `glRasterPos*()` and `glBitmap()` position and draw a single bitmap on the screen.

  • Void `glRasterPos{234}{sifd}{v} (TYPE x, TYPE y, TYPE z, TYPE w)`;
    Here (x, y, z, w) are transformed to screen coordinates through the transformation pipeline as an ordinary vertex

  • Void `glWindowPos{23}{sifd}{v} (TYPE x, TYPE y, TYPE z, TYPE w)`;
    Sets window coordinates
Bitmap and Characters (cont)

• A bitmap is a rectangular array of 0s and 1s that serves as a drawing mask for a rectangular portion of the window
• Specification:
  – Width need not be multiples of 8, but specified in hex code
  – Specified in one dimensional array, row by row, starting from lower-left corner

```c
GLubyte rasters[24] = {
    0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
    0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00,
    0xff, 0xc0, 0xff, 0xc0};
```

You can't rotate bitmap fonts because the bitmap is always drawn aligned to the x and y framebuffer axes.
• Draw a bit map at the current raster position:
  \texttt{glBitmap}(w, h, xbo, ybo, xbi, ybi, bitmapPtr)
  
  – \(xbo, ybo\): origin
  
  – \(xbi, ybi\): increment

\begin{align*}
  w &= 10 \\
  h &= 12 \\
  (x_{bo}, y_{bo}) &= (0, 0) \\
  (x_{bi}, y_{bi}) &= (11, 0)
\end{align*}

\texttt{glBitmap}(10, 12, 0, 0, 11, 0, bitmapPtr)
Drawing a Bitmapped Character: drawf.c

#include <GL/glut.h>
#include <stdlib.h>

GLubyte rasters[24] = {
  0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
  0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
  0xff, 0xc0, 0xff, 0xc0
};
void init(void) {
    glPixelStorei (GL_UNPACK_ALIGNMENT, 1); // how the data to be unpacked
    glClearColor (0.0, 0.0, 0.0, 0.0);
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glRasterPos2i (20, 20); // the current raster position
    glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
    glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
    glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
    glFlush();
}

void reshape(int w, int h) {
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho (0, w, 0, h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
Images

• an image can contain the complete (R, G, B, A) quadruple stored at each pixel.

• Images can come from several sources, such as:
  • A photograph that’s digitized with a scanner
  • Generated on the screen by a graphics program using the graphics hardware and then read back, pixel by pixel
  • A software program that generated the image in memory pixel by pixel

• In addition to simply being displayed on the screen, images can be used for texture maps
**An example: J4_5_Image**

```c
void initImage(void) {
    read_stars_image();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}

void drawImage(float x, float y, float z) {
    glRasterPos3f(x, y, z);
    glDrawPixels(stars_pixels, stars_pixels, GL_LUMINANCE,
                 GL_UNSIGNED_BYTE, stars_image);
}

void display(void) {
    drawImage(-2.4*Width, -2.4*Height, -2.0*Width);
    drawRobot(A, B, C, alpha, beta, gama);
    glutSwapBuffers();
}
```
Reading, Writing, and Copying Pixel Data

- OpenGL provides three basic commands that manipulate image data:
  - `glReadPixels()` -- Reads a rectangular array of pixels from the framebuffer and stores the data in processor memory.
  - `glDrawPixels()` -- Writes a rectangular array of pixels into the framebuffer from data kept in processor memory.
  - `glCopyPixels()` -- Copies a rectangular array of pixels from one part of the framebuffer to another.
More on glCopyPixels

• Note that there's no need for a format or data parameter for `glCopyPixels()`, since the data is never copied into processor memory.

• The read source buffer and the destination buffer of `glCopyPixels()` are specified by `glReadBuffer()` and `glDrawBuffer()` respectively
  – Default:
    • single-buffer: GL_FRONT
    • Double-buffered: GL_BACK
Storing, Transforming, and Mapping Pixels

• An image stored in memory has between 1 or 4 chunks of data (RGBA), called elements, for each pixel.

• The data might consist of just the color index or the luminance, or the RGBA components.

• The possible arrangements of pixel data, or formats, determine the number of elements.

• Image data is typically stored in processor memory in rectangular 2-D or 3-D arrays.
Storing, Transforming, and Mapping Pixels

- An image pixel in memory has between 1 to 4 elements (RGBA).
  - The elements might be the index, luminance, or RGBA components.

- The *formats* determine the number of elements stored for each pixel and their order.

- Image data is transferred from memory into framebuffer, or from framebuffer into memory.
  - OpenGL can perform operations on it: `glPixelTransfer()`, `glPixelMap()`, `glPixelZoom()`
# Pixel Format

<table>
<thead>
<tr>
<th>format Constant</th>
<th>Pixel Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_COLOR_INDEX</td>
<td>A single color index</td>
</tr>
<tr>
<td>GL_RGB</td>
<td>A red color component, followed by a green color component</td>
</tr>
<tr>
<td>GL_RGBA</td>
<td>A red color component, followed by a green color component, followed by a blue color component</td>
</tr>
<tr>
<td>GL_RED</td>
<td>A single red color component</td>
</tr>
<tr>
<td>GL_GREEN</td>
<td>A single green color component</td>
</tr>
<tr>
<td>GL_BLUE</td>
<td>A single blue color component</td>
</tr>
<tr>
<td>GL_ALPHA</td>
<td>A single alpha color component</td>
</tr>
<tr>
<td>GL_LUMINANCE</td>
<td>A single luminance component</td>
</tr>
<tr>
<td>GL_LUMINANCE_ALPHA</td>
<td>A luminance component followed by an alpha color component</td>
</tr>
<tr>
<td>GL_STENCIL_INDEX</td>
<td>A single stencil index</td>
</tr>
<tr>
<td>GL_DEPTH_COMPONENT</td>
<td>A single depth component</td>
</tr>
</tbody>
</table>
# Data Type

<table>
<thead>
<tr>
<th>type Constant</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_UNSIGNED_BYTE</td>
<td>unsigned 8-bit integer</td>
</tr>
<tr>
<td>GL_BYTE</td>
<td>signed 8-bit integer</td>
</tr>
<tr>
<td>GL_BITMAP</td>
<td>single bits in unsigned 8-bit integers using the same format as glBitmap()</td>
</tr>
<tr>
<td>GL_UNSIGNED_SHORT</td>
<td>unsigned 16-bit integer</td>
</tr>
<tr>
<td>GL_SHORT</td>
<td>signed 16-bit integer</td>
</tr>
<tr>
<td>GL_UNSIGNED_INT</td>
<td>unsigned 32-bit integer</td>
</tr>
<tr>
<td>GL_INT</td>
<td>signed 32-bit integer</td>
</tr>
<tr>
<td>GL_FLOAT</td>
<td>single-precision floating point</td>
</tr>
</tbody>
</table>
Pixel Storage Modes

• glPixelStore()
  – Controlling Pixel-Storage Modes
  – glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    • Packing and unpacking refer to the way that pixel data is written to and read from processor memory.
    • tells OpenGL not to skip bytes at the end of a row
Imaging Pipeline
Controlling Pixel-Storage Modes

```c
void glPixelStore{if}(GLenum pname, TYPE param)
```

Sets the pixel-storage modes

The GL_UNPACK* parameters control how data is unpacked from memory into framebuffer; The GL_PACK* parameters control how data is packed into memory.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Valid Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_UNPACK_SWAP_BYTES,</td>
<td>GLboolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>GL_PACK_SWAP_BYTES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_UNPACK_LSB_FIRST,</td>
<td>GLboolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>GL_PACK_LSB_FIRST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_UNPACK_ROW_LENGTH,</td>
<td>GLint</td>
<td>0</td>
<td>any nonnegative integer</td>
</tr>
<tr>
<td>GL_PACK_ROW_LENGTH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_UNPACK_SKIP_ROWS,</td>
<td>GLint</td>
<td>0</td>
<td>any nonnegative integer</td>
</tr>
<tr>
<td>GL_PACK_SKIP_ROWS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_UNPACK_SKIP_PIXELS,</td>
<td>GLint</td>
<td>0</td>
<td>any nonnegative integer</td>
</tr>
<tr>
<td>GL_PACK_SKIP_PIXELS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_UNPACK_ALIGNMENT,</td>
<td>GLint</td>
<td>4</td>
<td>1, 2, 4, 8</td>
</tr>
<tr>
<td>GL_PACK_ALIGNMENT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PixelStorei(GL_UNPACK_ALIGNMENT, x)

• Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are:
  – 1 (byte-alignment),
  – 2 (rows aligned to even-numbered bytes),
  – 4 (word-alignment [default]), and
  – 8 (rows start on double-word boundaries).

<table>
<thead>
<tr>
<th>Byte: 8-bit</th>
<th>Word: 16-bit</th>
<th>Double-word: 32-bit</th>
</tr>
</thead>
</table>
Assuming the image is of size 3x2:

Client memory

The start of each row of pixels is …

Byte aligned (1)

Word aligned (4)
Settings

- For RGBA images, it doesn’t matter (each pixel has 4 bytes: RGBARGBA…)

- For RGB and luminance images, it should be set to byte-aligned (data are densely packed)

```c
glPixelStorei(GL_UNPACK_ALIGNMENT, 1)
```

```c
glPixelStorei(GL_UNPACK_ALIGNMENT, 4)
```
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Valid Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH</td>
<td>GLint</td>
<td>0</td>
<td>any nonnegative integer</td>
</tr>
<tr>
<td>GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS</td>
<td>GLint</td>
<td>0</td>
<td>any nonnegative integer</td>
</tr>
<tr>
<td>GL_UNPACK_SKIP_PIXELS, GL_PACK_SKIP_PIXELS</td>
<td>GLint</td>
<td>0</td>
<td>any nonnegative integer</td>
</tr>
<tr>
<td>GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT</td>
<td>GLint</td>
<td>4</td>
<td>1, 2, 4, 8</td>
</tr>
</tbody>
</table>
Pixel-Transfer Operations

You can perform various operations on pixels as they’re transferred from and to the framebuffer: `glPixelTransfer{if}(GLenum pname, TYPEparam);`

Pixel Mapping

All the color components, color indices, and stencil indices can be modified by means of a table lookup to map to different values/ranges:

`void glPixelMapfv(GLenum map, GLsizei mapsize, const GLfloat * values);`

Magnifying, Reducing, or Flipping an Image

• Normally, each pixel in an image is written to a single pixel on the screen. However, you can arbitrarily magnify or reduce an image by using `glPixelZoom(GLfloat xfactor, GLfloat yfactor).`
Improving Pixel Pipeline Performance

- A series of fragment operations is applied to pixels as they are drawn into the framebuffer. For optimum performance, disable all fragment operations.
- While performing pixel operations, disable other costly states, such as texturing and lighting.
- It is usually faster to draw a large pixel rectangle than to draw several small ones, since the cost of transferring the pixel data can be amortized over many pixels.