CP SC 4040/6040 Computer Graphics Images

Joshua Levine
levinej@clemson.edu

Lecture 02
OpenImageIO and OpenGL

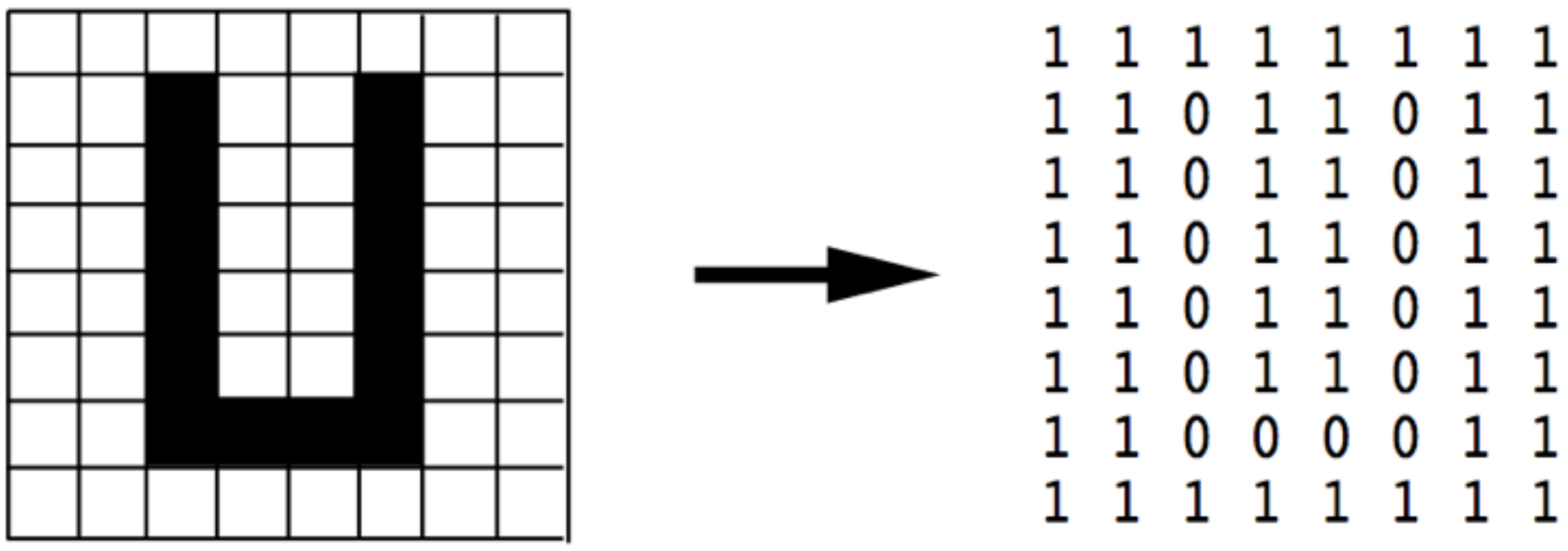
Aug. 25, 2015

Agenda

- Reminder, course webpage:
    - http://people.cs.clemson.edu/~levinej/courses/6040
- Is everyone on Piazza?
- Clarification on Final Project Presentations (only on last day of lecture) vs. Final Exams (Fri., Dec. 11)
- Open Discussion: Quizzes
- Topics for today: Encoding Digital Images, OIIO, OpenGL

Encoding Digital Images

Bitmaps



- Bitmap: digital image that is a 2d array of pixels which store one bit.
  - Simplest digital image, a representation of a black and white image.
- Pixel: **pic**ture **el**ement, individual sample of an image.
- Bit: ones/zeros, convention is 0 = black & 1 = white.
- Scanline: a row in the 2d array (terminology from acquisition).

Digital Images Linearized

```
1 1 1 1 1 1 1 1
1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1
1 1 0 0 0 0 1 1
1 1 1 1 1 1 1 1
```

- While we think of images as 2-dimensional, in memory they are 1-dimensional.

11111111	11011011	11011011	11011011	11011011	11011011	11000011	11111111

“U” in 8 bytes, binary + hexadecimal

FF	DB	DB	DB	DB	DB	C3	FF

Greyscale Images - Pixmaps

- We use 0 for black and 1 for white -- what value should we use for grey?
- Could use floating point numbers
- Instead, one convention is to use 8bits for pixel -- how many different “shades of grey”?
- Can convert to [0.0,1.0] by dividing by 256

Code! Pixmap Declaration

```
const int ROWS = 8;
const int COLS = 8;
unsigned char pixmap[ROWS][COLS];
unsigned char pixmap2[ROWS*COLS];

//top left pixel (x,y) = (0,0)
pixmap[0][0];
pixmap2[0];

//top right pixel (x,y) = (0,7)
pixmap[0][7];
pixmap2[7];

//bottom left pixel, in general [index] = [y*COLS+x]
pixmap[7][0];
pixmap2[56];

//bottom right pixel
pixmap[7][7]
pixmap2[ ?? ];  //fill me in
```

```
void print_greymap(unsigned char *greymap,
                   int width,
                   int height)
{
  // scanline number
  int y;
  // pixel number on scanline
  int x;
  // value of pixel (0 to 255)
  int value;

  for(y = 0; y < height; y++){  // loop for each scanline
    for(x = 0; x < width; x++){   // loop for each pixel on line
      value = greymap[y * width + x];  // fetch pixel value
      printf("%5.3f ", value / 255.0);
    }
    printf("\n");
  }
}
```

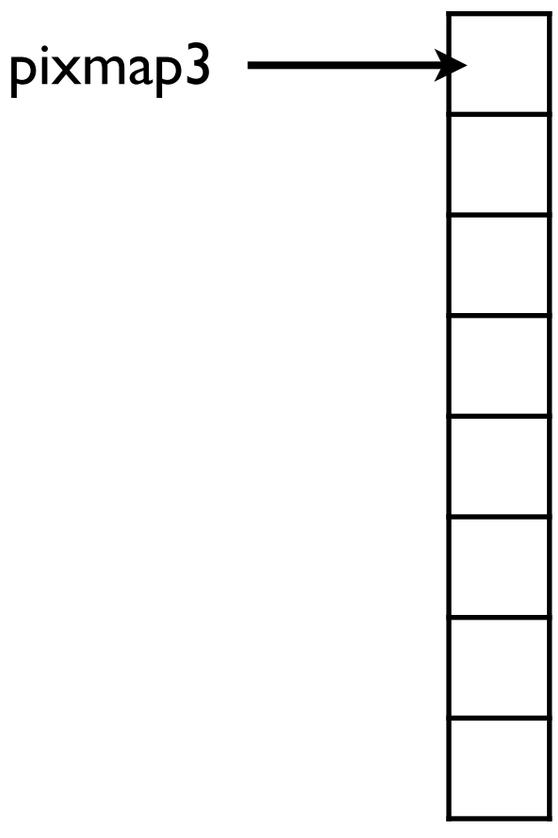
More Code!
Pixmap “Display”

Ever More Code!
Pixmap Declaration #2

```
const int ROWS = 8;
const int COLS = 8;

//like pixmap[][]
unsigned char ** pixmap3;
pixmap3 = new unsigned char*[ROWS];
for (int y=0; y<ROWS; y++) {
  pixmap3[y] = new unsigned char[COLS];
}

//like pixmap2[]
unsigned char * pixmap4;
pixmap4 = new unsigned char[ROWS*COLS];

//contiguous allocation
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
//note y starts with 1!!!
for (int y=1; y<ROWS; y++) {
  pixmap5[y] = pixmap[y-1] + COLS;
}
```

Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap3;  //like pixmap[][]
pixmap3 = new unsigned char*[ROWS];
for (int y=0; y<ROWS; y++) {
  pixmap3[y] = new unsigned char[COLS];
}
```



Rows separated in memory!

Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap3;  //like pixmap[][]
pixmap3 = new unsigned char*[ROWS];
for (int y=0; y<ROWS; y++) {
  pixmap3[y] = new unsigned char[COLS];
}
```

Rows separated in memory!

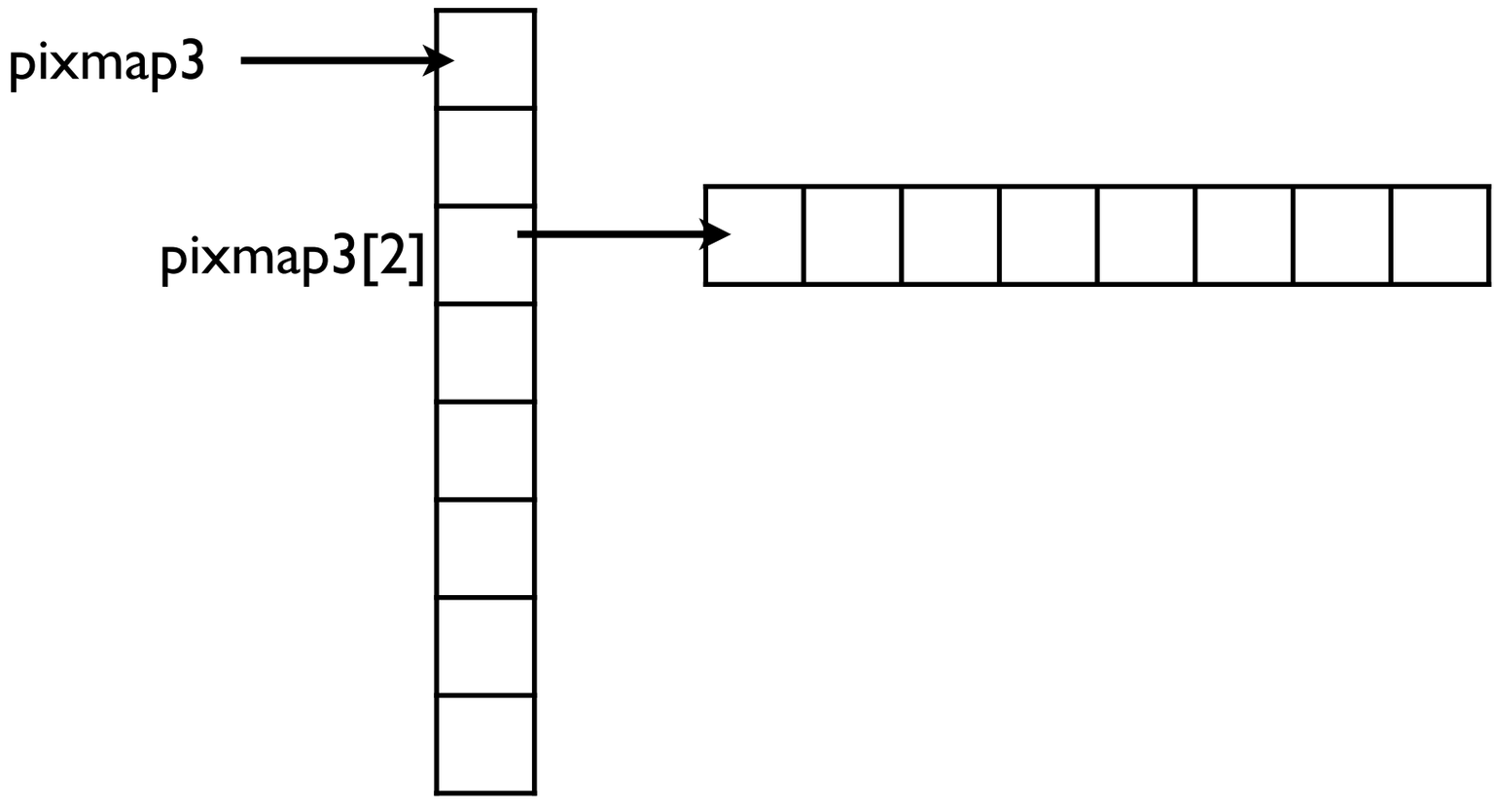


Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap3;  //like pixmap[][]
pixmap3 = new unsigned char*[ROWS];
for (int y=0; y<ROWS; y++) {
  pixmap3[y] = new unsigned char[COLS];
}
```

Rows separated in memory!

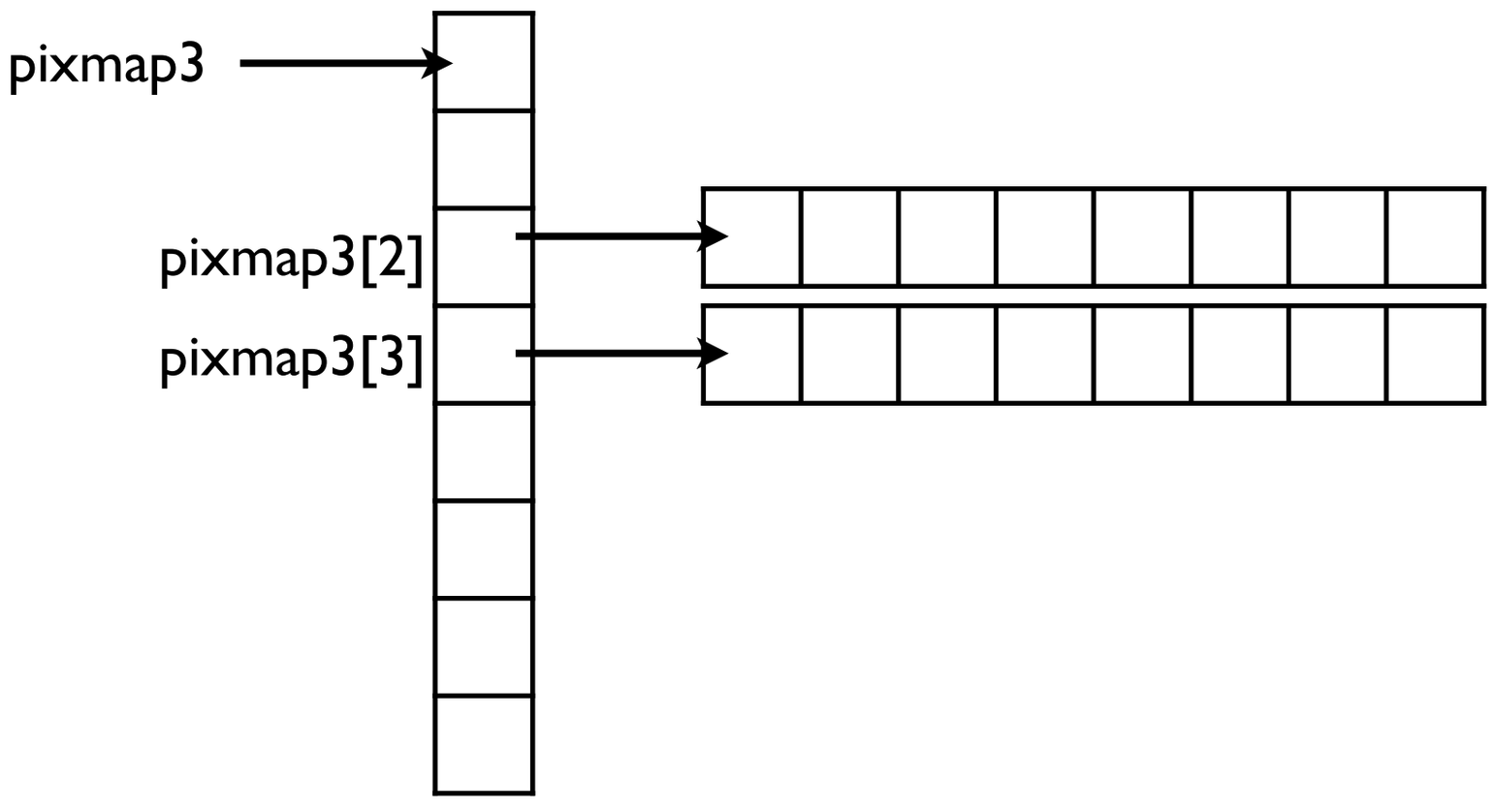


Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap3;  //like pixmap[][]
pixmap3 = new unsigned char*[ROWS];
for (int y=0; y<ROWS; y++) {
  pixmap3[y] = new unsigned char[COLS];
}
```

Rows separated in memory!

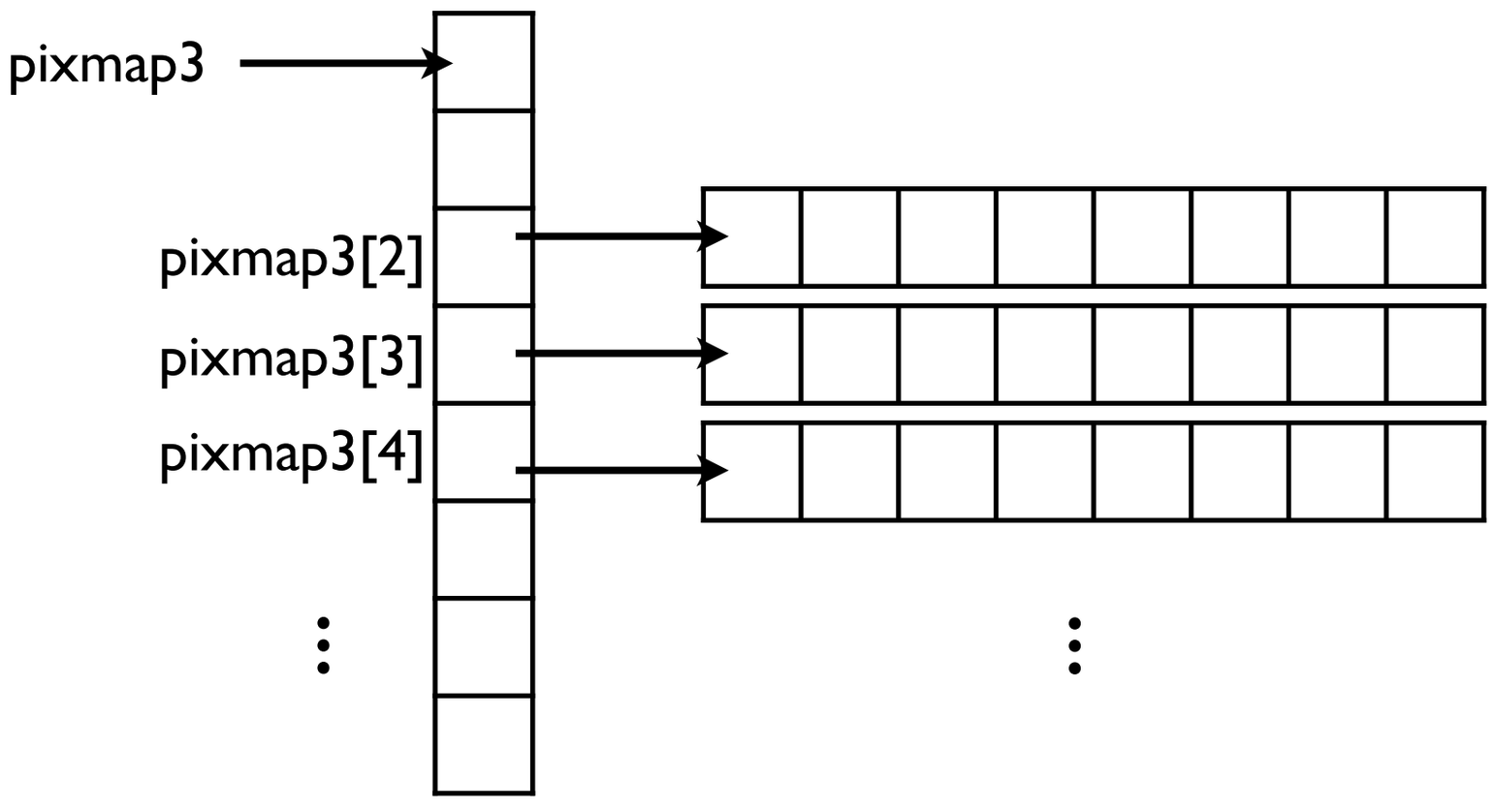


Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap3;  //like pixmap[][]
pixmap3 = new unsigned char*[ROWS];
for (int y=0; y<ROWS; y++) {
  pixmap3[y] = new unsigned char[COLS];
}
```

Rows separated in memory!

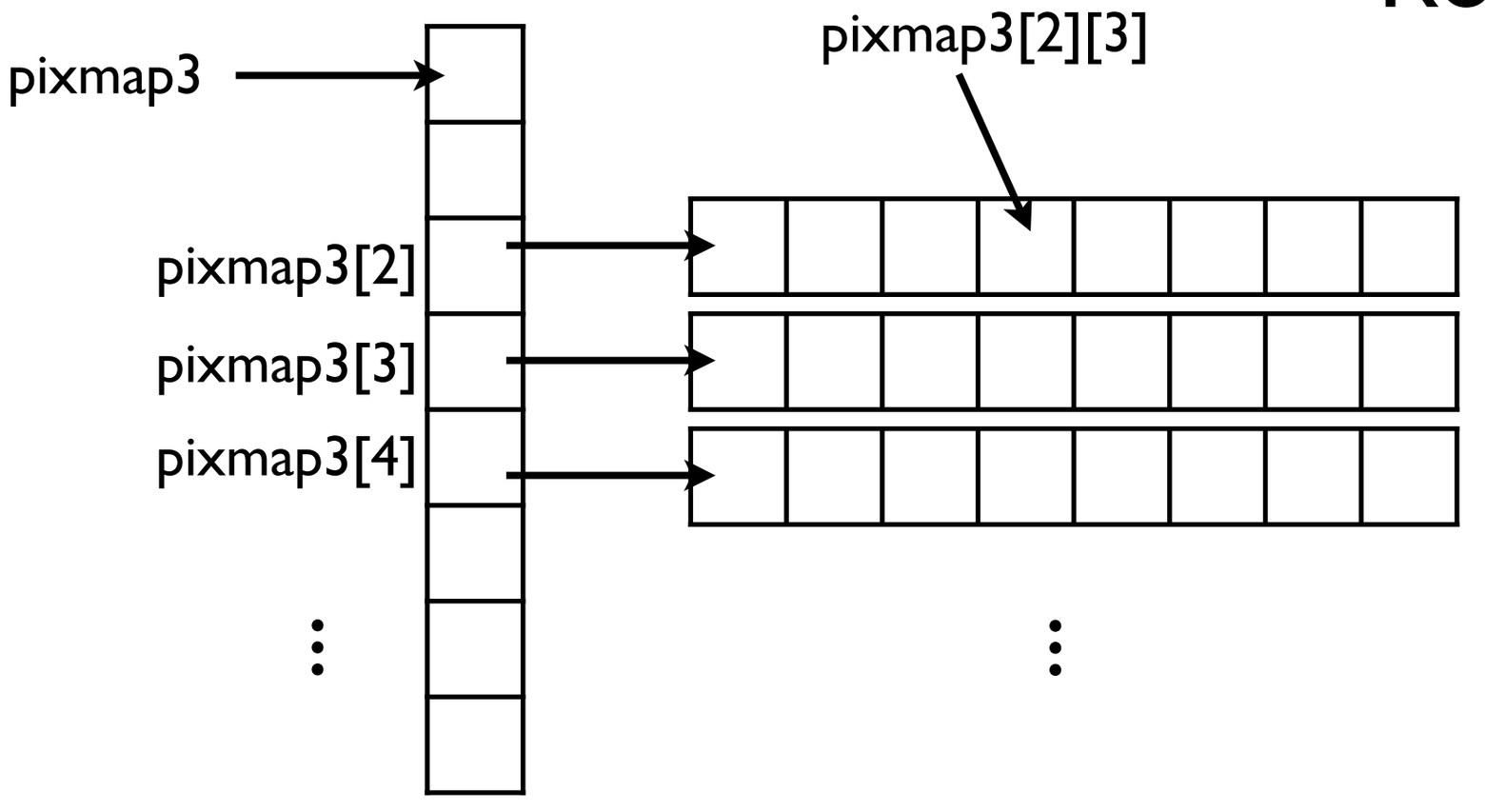
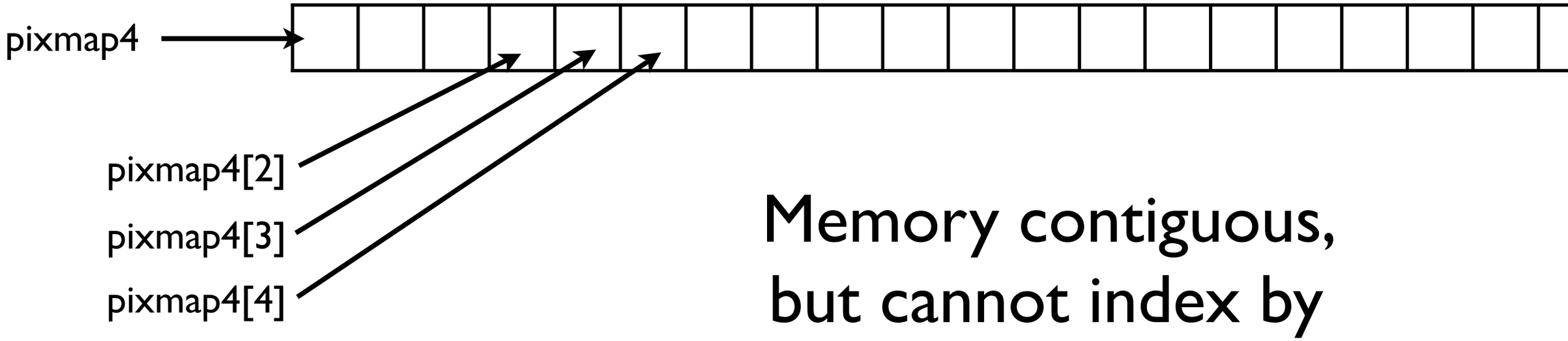


Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char * pixmap4; //like pixmap2[]
pixmap4 = new unsigned char[ROWS*COLS];
```



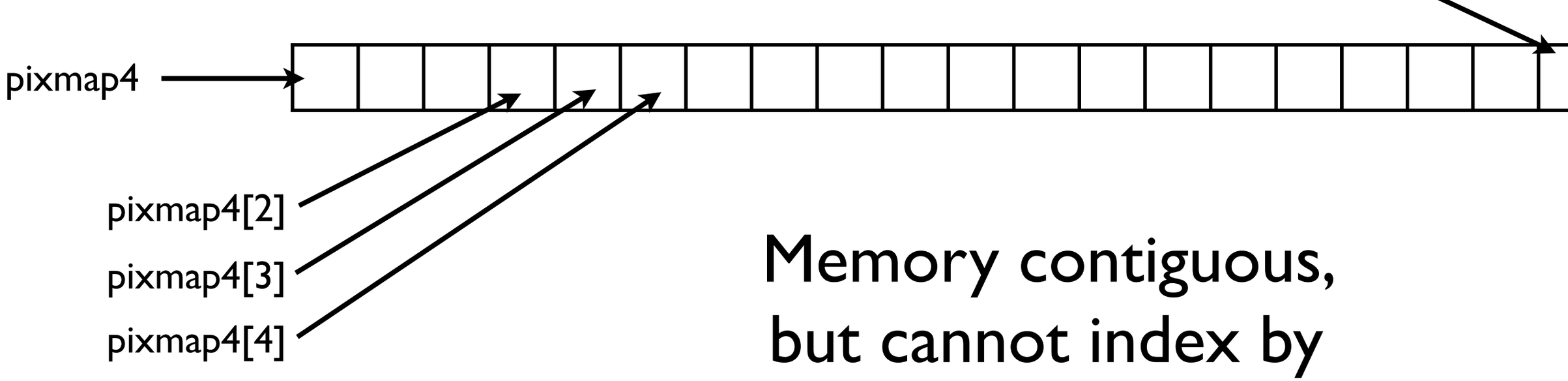
Memory contiguous,
but cannot index by
row / column

Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char * pixmap4; //like pixmap2[]
pixmap4 = new unsigned char[ROWS*COLS];
```

We don't have this

XXXX[2][3]



Memory contiguous,
but cannot index by
row / column

Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
//contiguous allocation
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
for (int y=1; y<ROWS; y++) {  //note index starts with 1!!!
  pixmap5[y] = pixmap5[y-1] + COLS;
}
```

(y,x) access AND contiguous memory

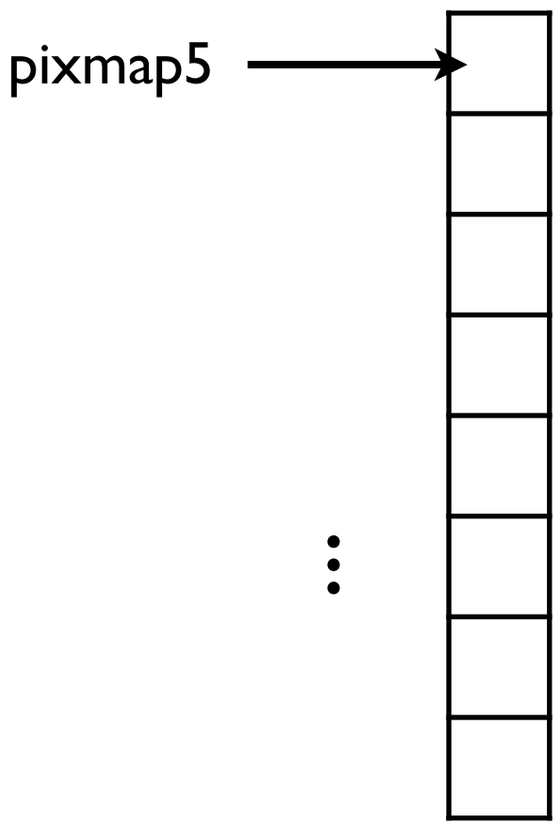


Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
//contiguous allocation
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
for (int y=1; y<ROWS; y++) {  //note index starts with 1!!!
  pixmap5[y] = pixmap5[y-1] + COLS;
}
```

(y,x) access AND contiguous memory

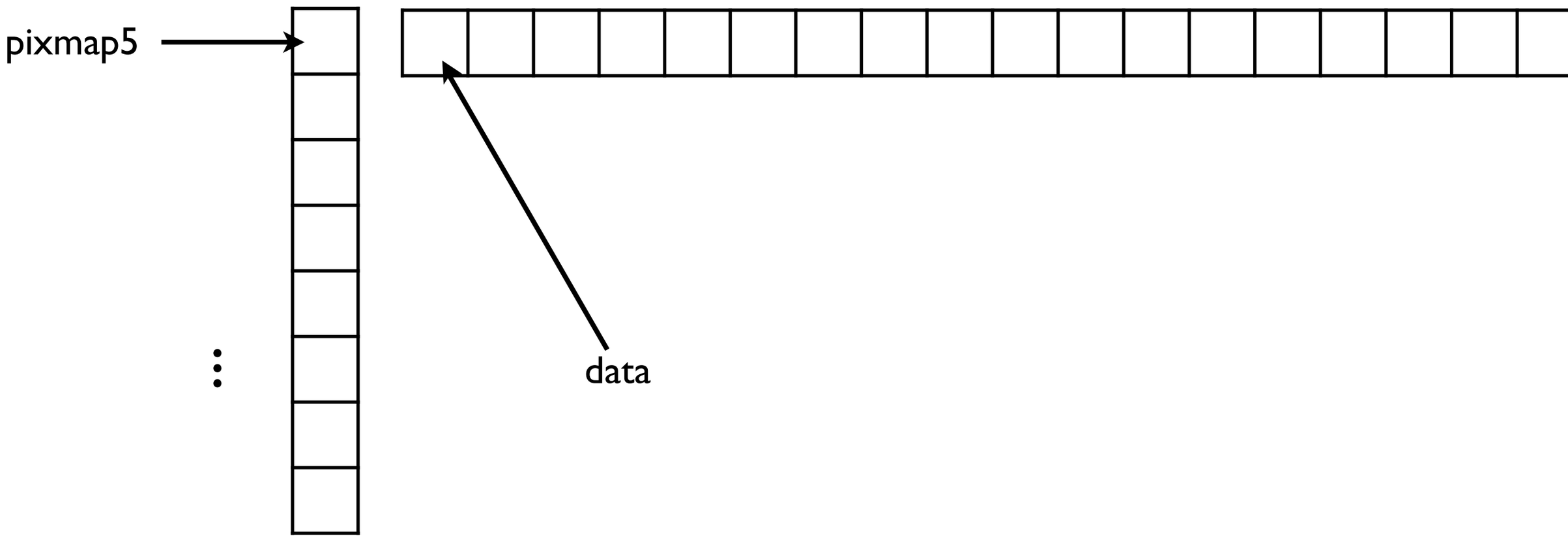


Image Allocation

(y,x) access AND
contiguous memory

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
//contiguous allocation
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
for (int y=1; y<ROWS; y++) {  //note index starts with 1!!!
  pixmap5[y] = pixmap5[y-1] + COLS;
}
```

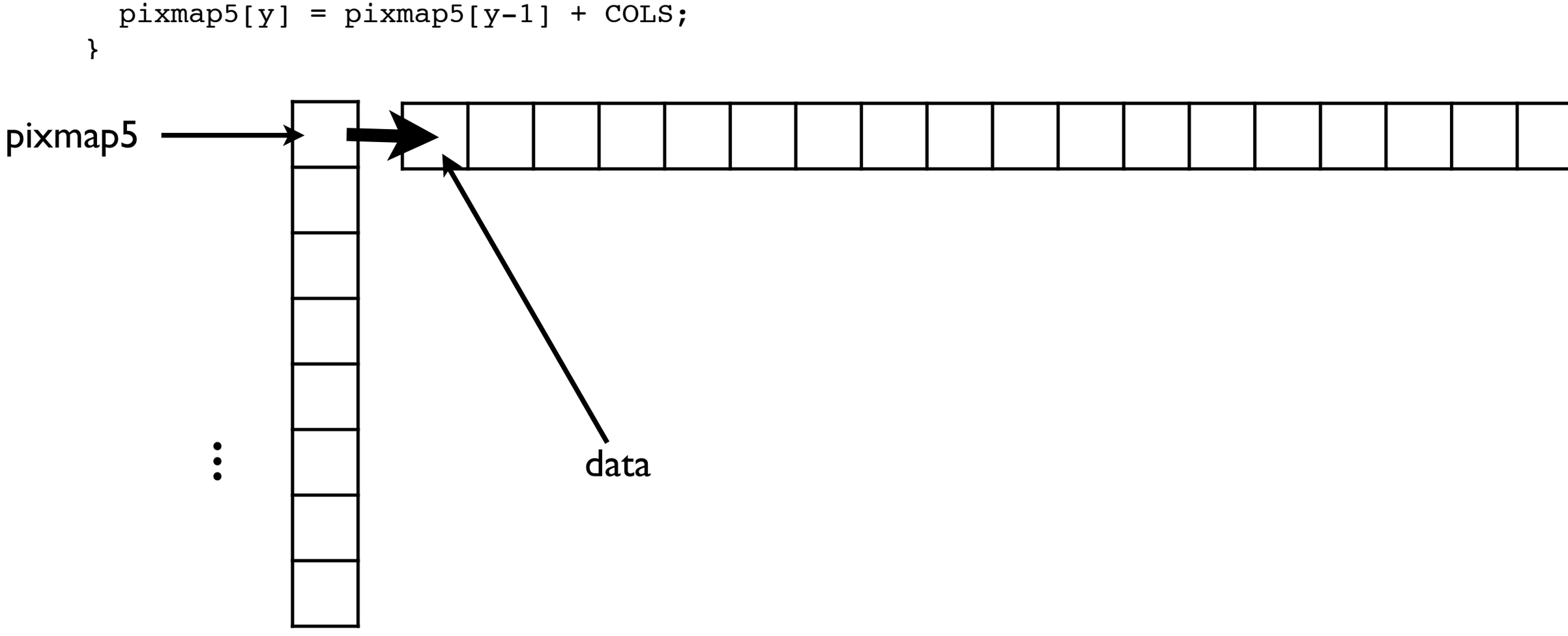


Image Allocation

(y,x) access AND
contiguous memory

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
//contiguous allocation
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
for (int y=1; y<ROWS; y++) {  //note index starts with 1!!!
  pixmap5[y] = pixmap5[y-1] + COLS;
}
```

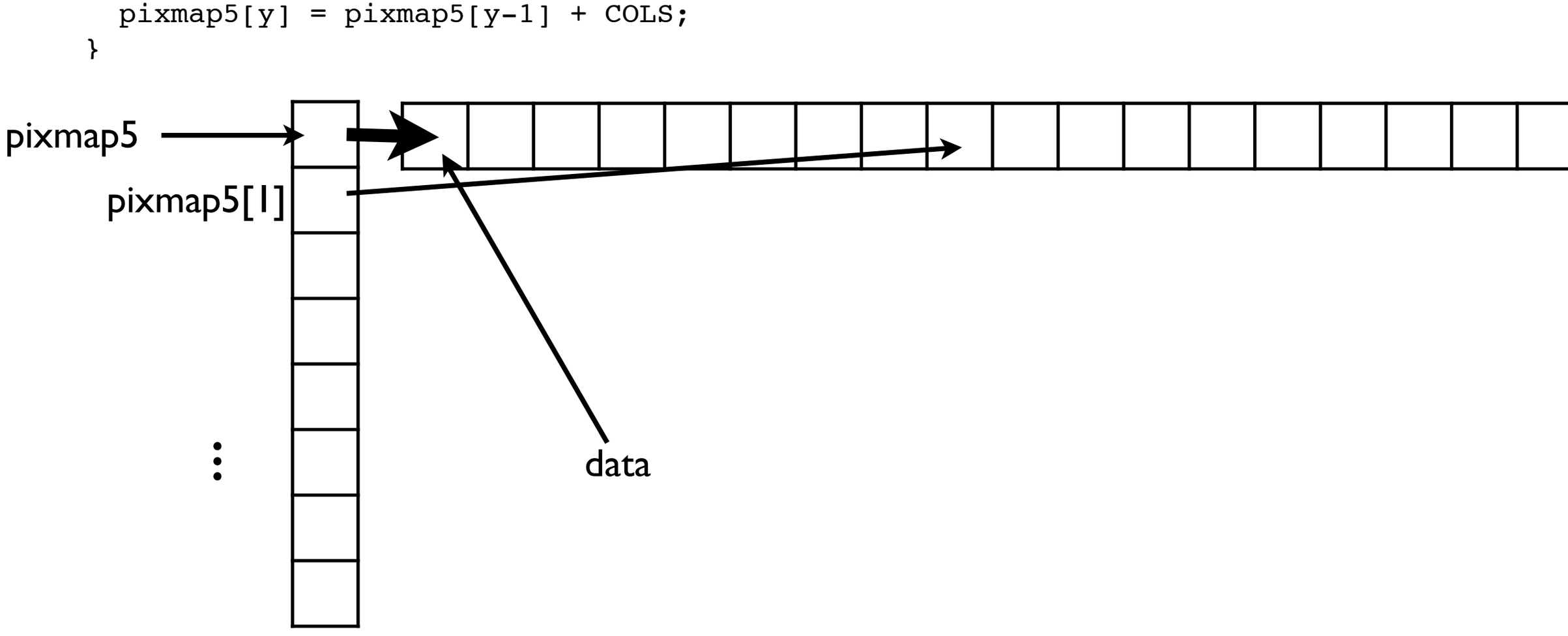


Image Allocation

(y,x) access AND
contiguous memory

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
//contiguous allocation
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
for (int y=1; y<ROWS; y++) {  //note index starts with 1!!!
  pixmap5[y] = pixmap5[y-1] + COLS;
}
```

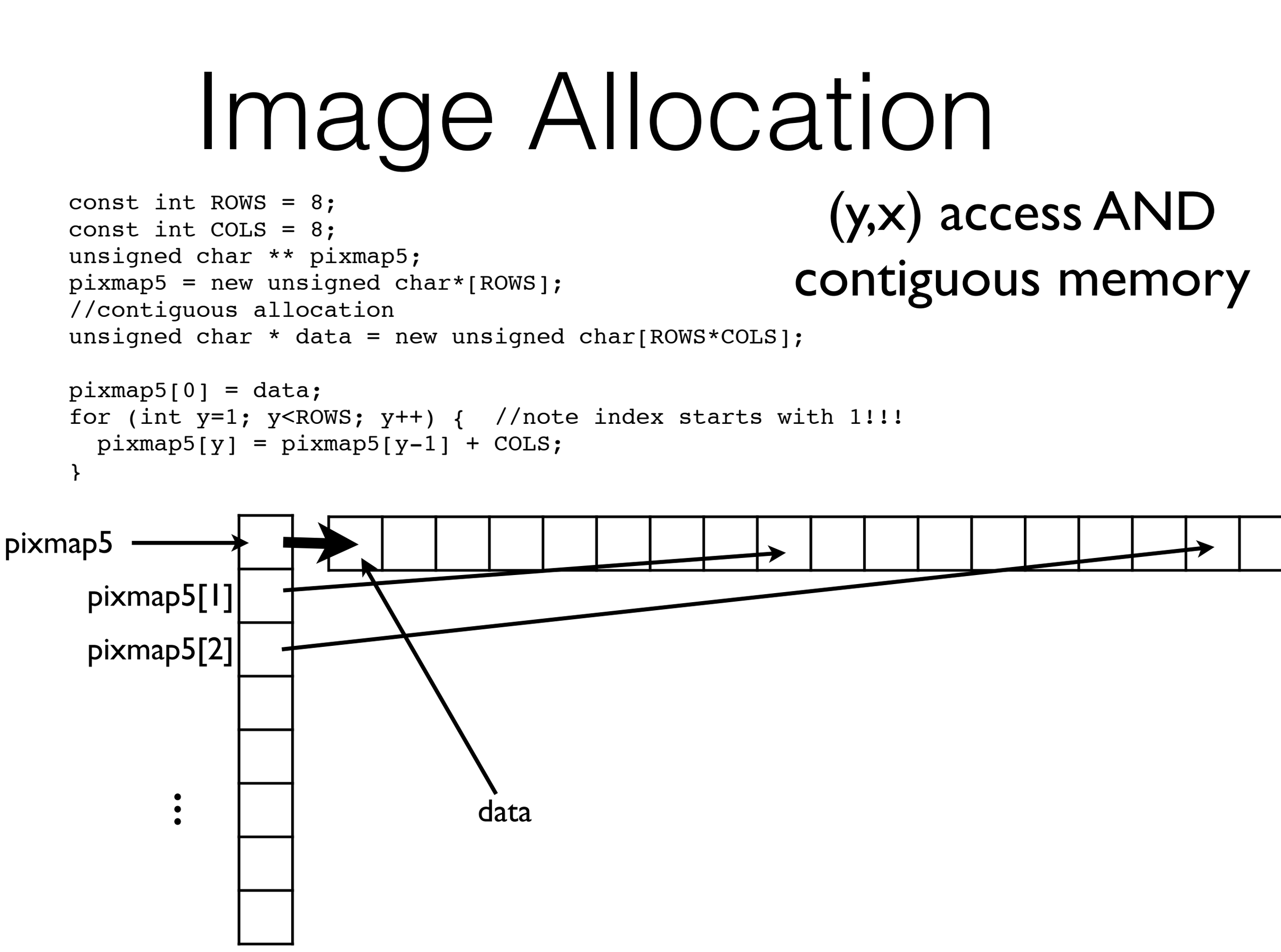


Image Allocation

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
//contiguous allocation
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
for (int y=1; y<ROWS; y++) {  //note index starts with 1!!!
  pixmap5[y] = pixmap5[y-1] + COLS;
}
```

(y,x) access AND contiguous memory

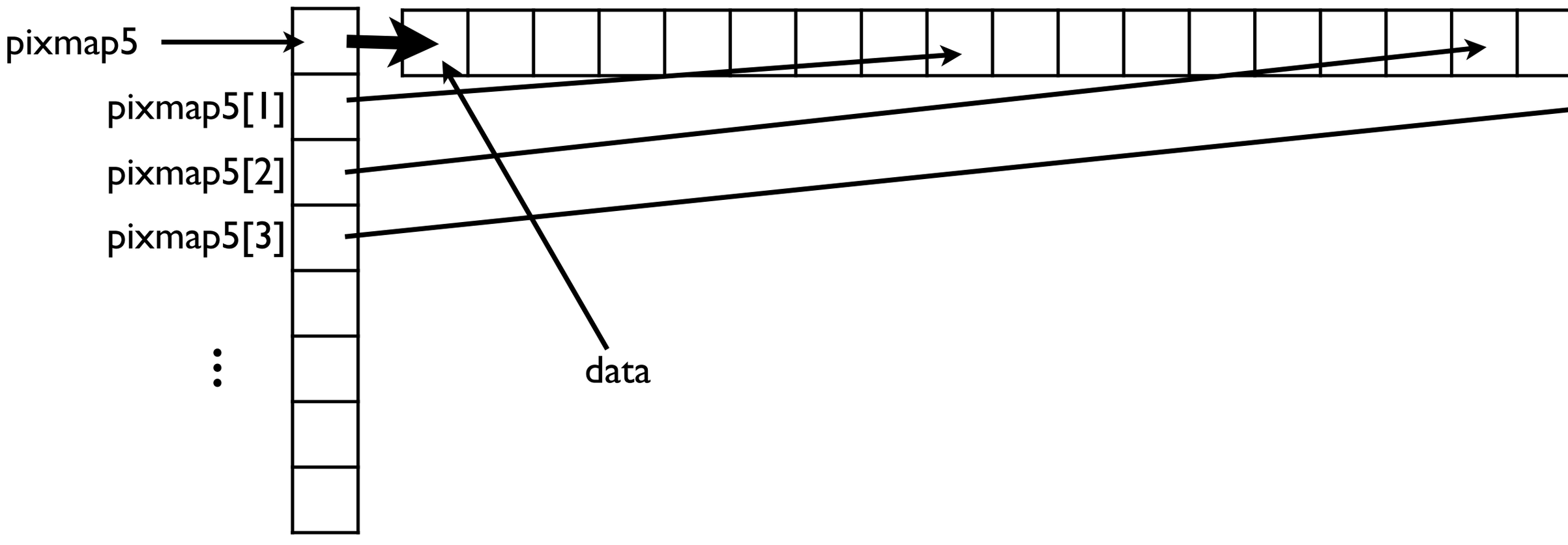
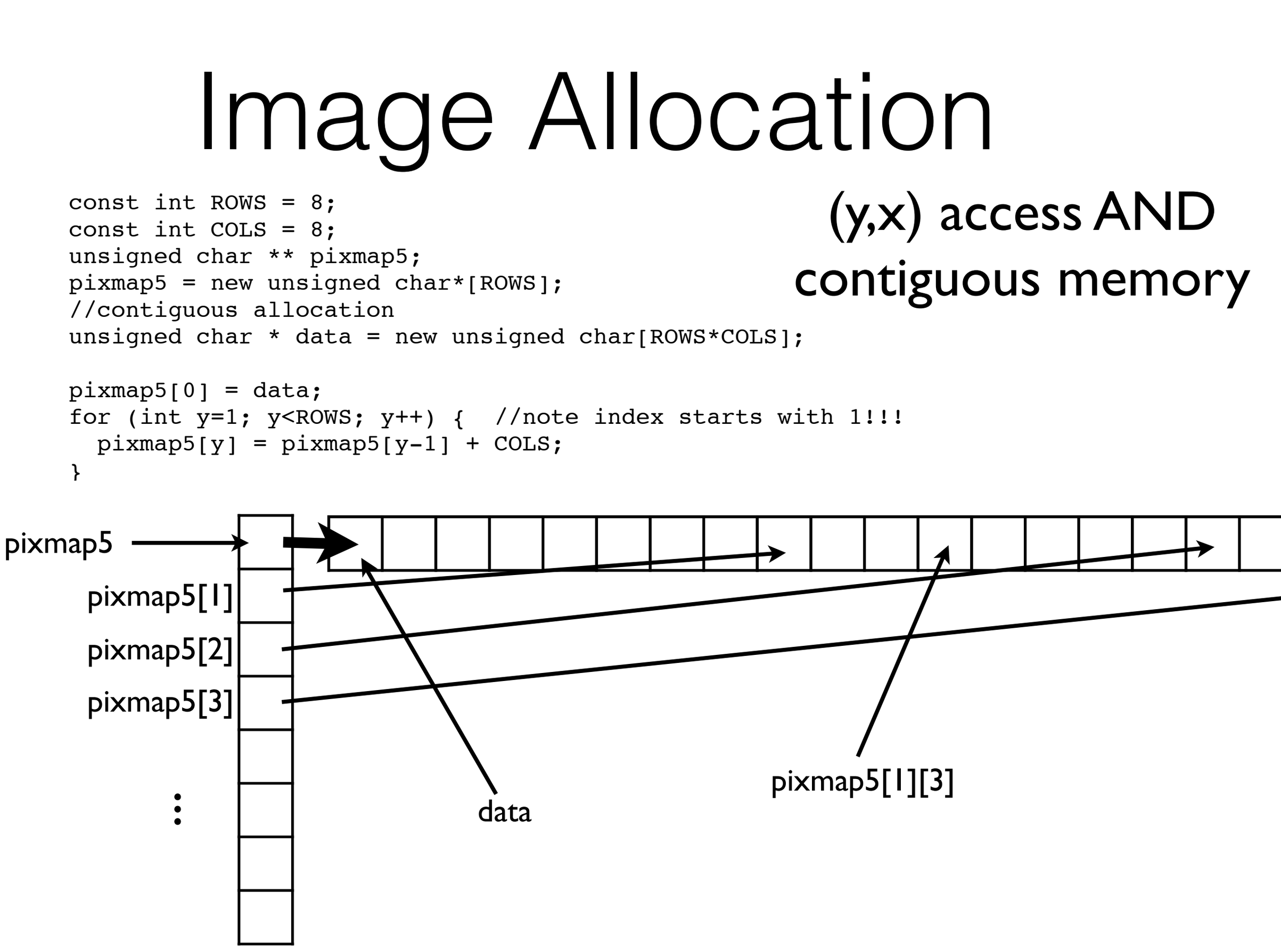


Image Allocation

(y,x) access AND contiguous memory

```
const int ROWS = 8;
const int COLS = 8;
unsigned char ** pixmap5;
pixmap5 = new unsigned char*[ROWS];
//contiguous allocation
unsigned char * data = new unsigned char[ROWS*COLS];

pixmap5[0] = data;
for (int y=1; y<ROWS; y++) {  //note index starts with 1!!!
  pixmap5[y] = pixmap5[y-1] + COLS;
}
```



Encoding Color Images

- Could encode 256 colors with an unsigned char. But what convention to use?

- One of the most common is to use 3 **channels** or bands

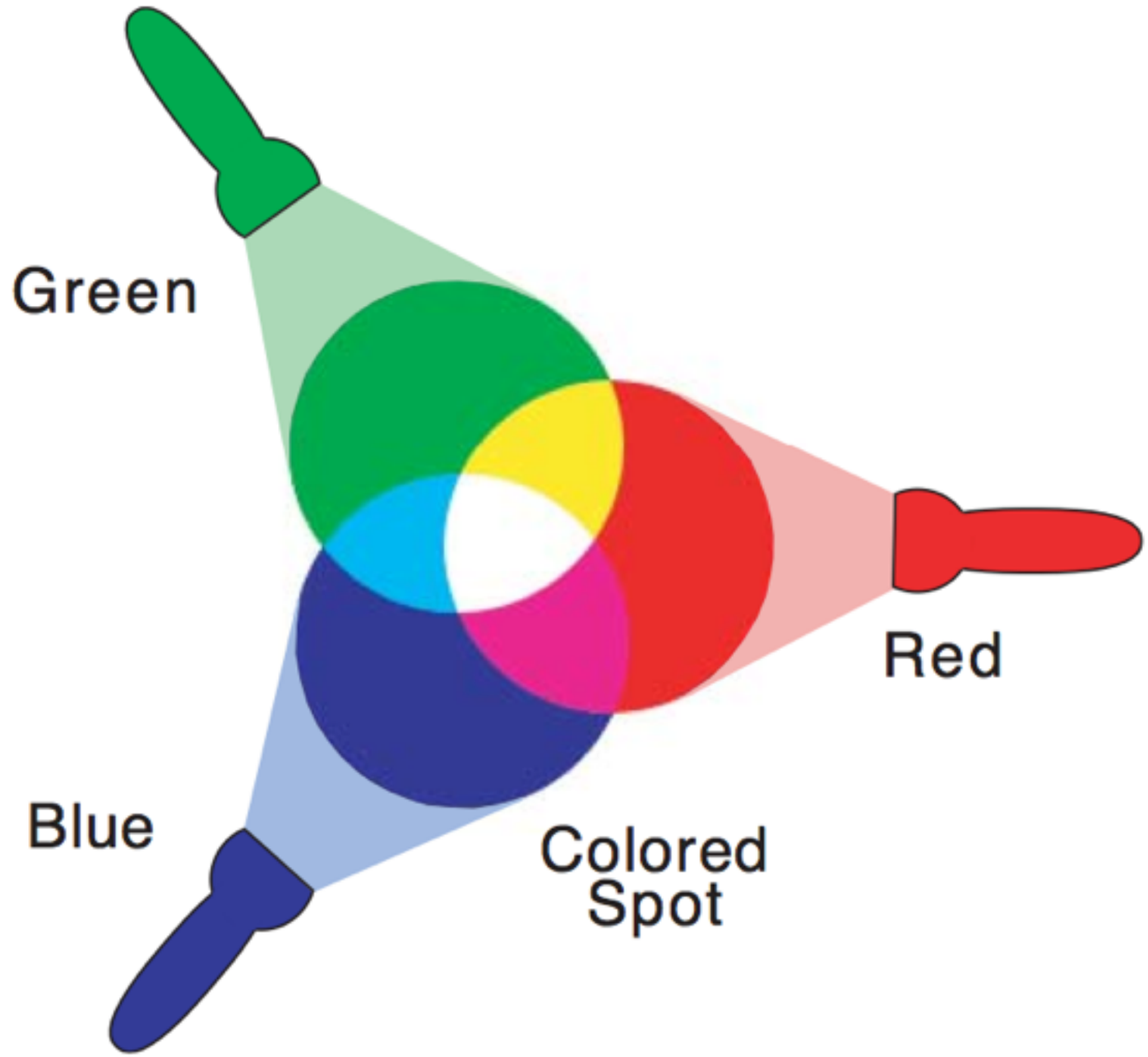
- Red-Green-Blue or RGB color is the most common -- based on how color is represented by lights.

- Coincidentally, this just happens to be related to how our eyes work too.

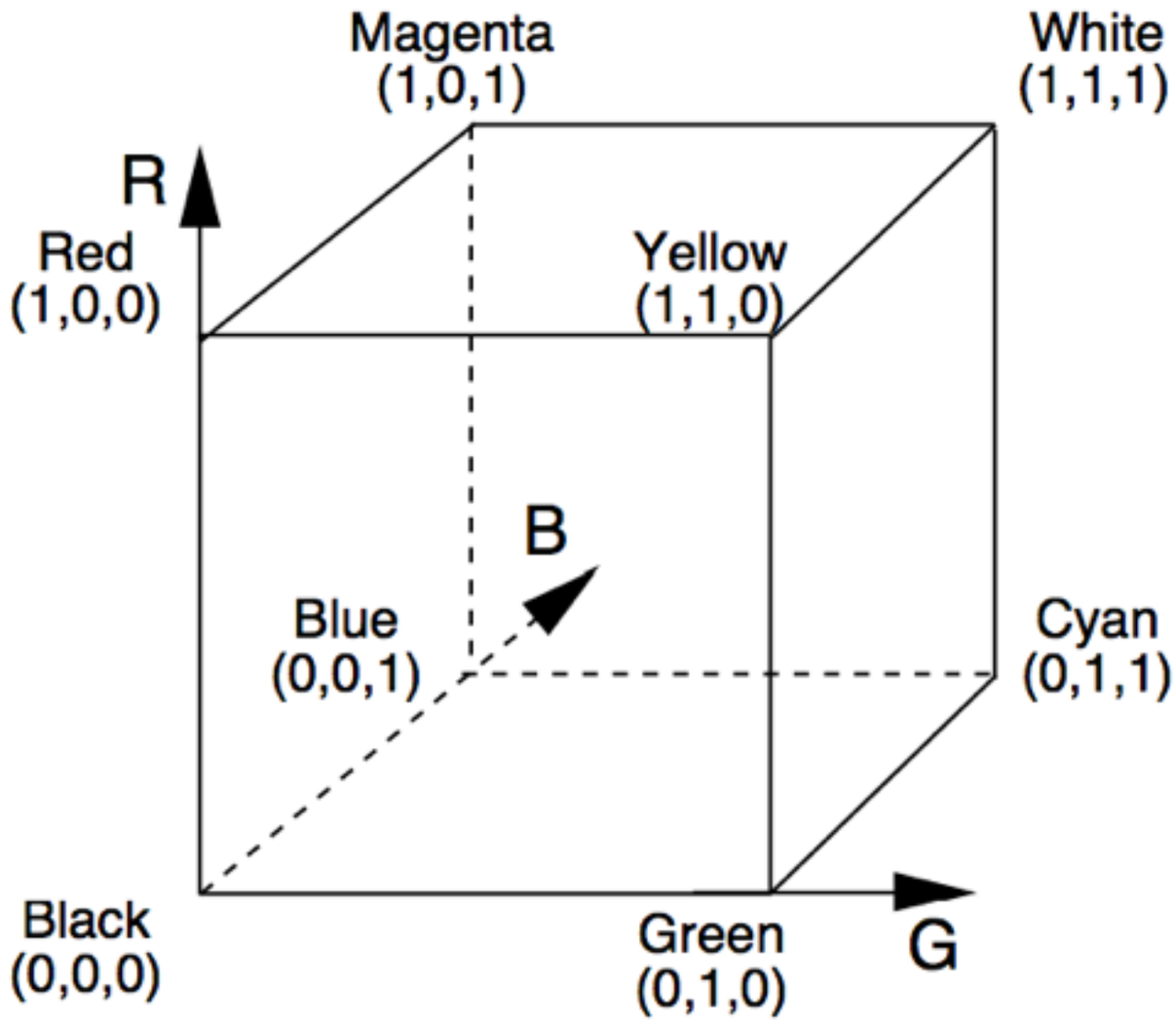
NOTE : There are many schemes to represent color, most use 3 channels. We'll come back to them in a future lecture

RGB Colors

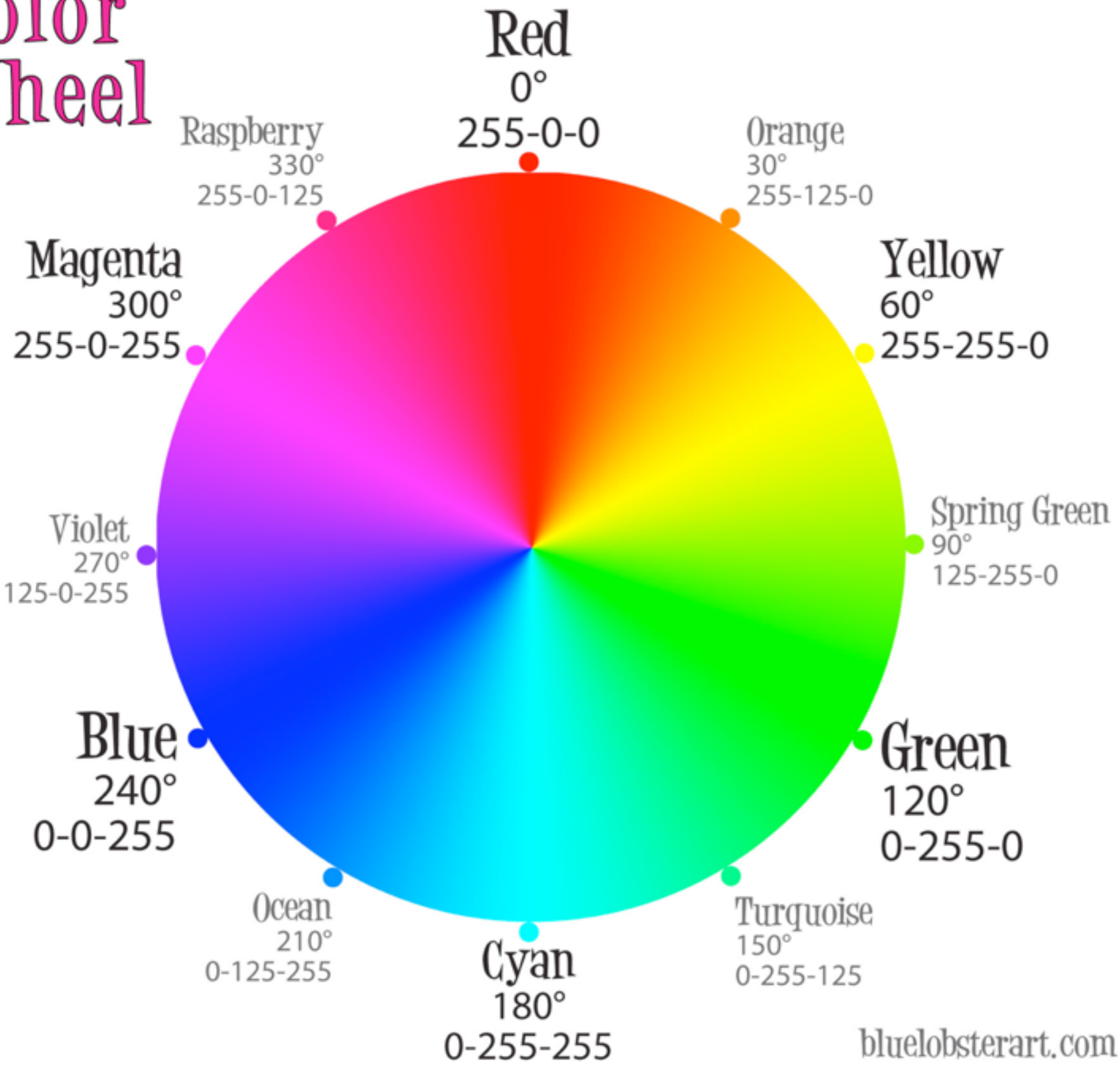
- Additive Mixing of 3 Lights



RGB Cube Code Demo



RGB Color Wheel



bluelobsterart.com

Encoding Color Channels

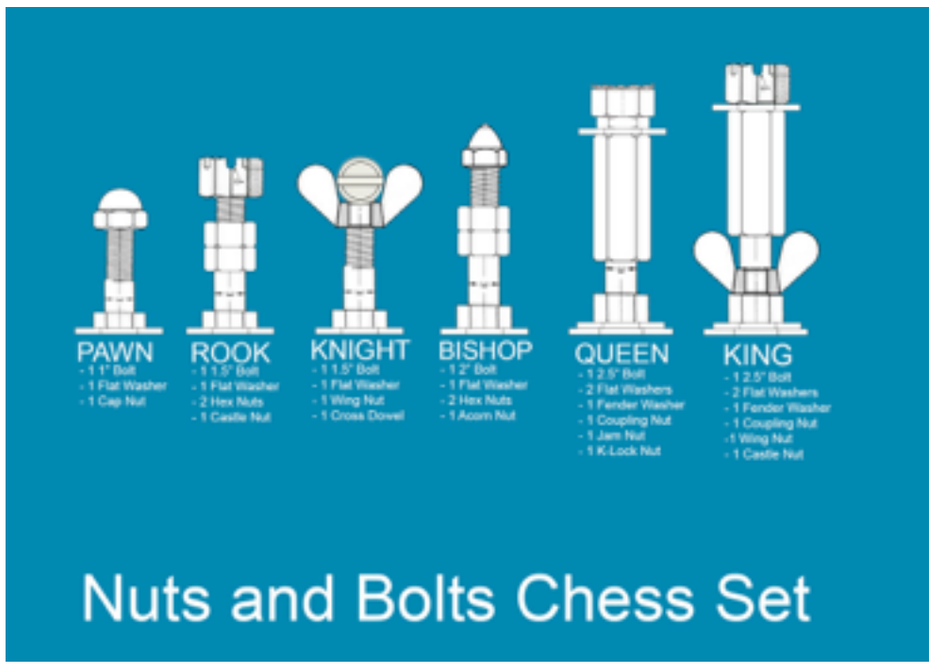
- Could use 8-bits, spread across all 3 channels (a bit ugly…)

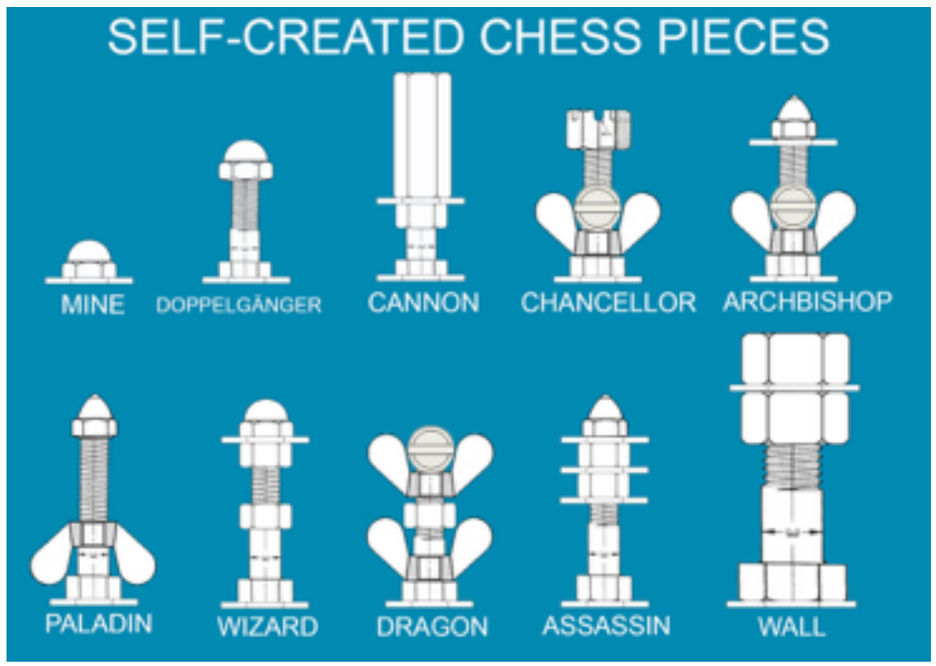
$$59_{16} = \underset{\begin{array}{ccc}\mathrm{R} & \mathrm{G} & \mathrm{B}\end{array}}{\begin{array}{|c|c|c|}\hline 010 & 110 & 01 \\ \hline\end{array}} = (2/7, 6/7, 1/3) = (0.286, 0.757, 0.333)$$

RGB Pixmap Encoding

```
//separate channel encoding

unsigned char red_pixmap[ROWS][COLS];
unsigned char green_pixmap[ROWS][COLS];
unsigned char blue_pixmap[ROWS][COLS];

//all together, could use an 32-bit uint,
//might be useful if we have 4 channels
//see House for how to access individual channels.
unsigned int rgb_pixmap[ROWS][COLS];

//or pack the bits with a struct
struct pixel {
  unsigned char red;
  unsigned char green;
  unsigned char blue;
};

pixel * pixmap = new pixel[ROWS*COLS];

//can we do better???
```

OIIO Nuts and Bolts





http://retro-gamer.deviantart.com/gallery/

OIIO Includes

- At the top of every file which uses OIIO:

```
#include <OpenImageIO/imageio.h>
```

  - Includes the declarations of main OIIO classes

`OIIO_NAMESPACE_USING`

  - Sets up the namespace appropriately

OIIO Classes

`ImageSpec`

- Describes the type of data in an image

`ImageOutput`

- Provides functionality to write images

`ImageInput`

- Provides functionality to read images

`ImageSpec` Members

- **`width`**, **`height`**, **`depth`** - resolution of the image
- **`x`**, **`y`**, **`z`** - origin of image
- **`nchannels`** - number of channels
- `TypeDesc` format and **`std::vector<TypeDesc> channelformats`**
  - **`format`** used if all channels the same format, otherwise **`channelformats`** describes each channel
  - `TypeDesc` is a class which stores different data depths, e.g. **`TypeDesc::UINT8`**, **`TypeDesc::FLOAT`**
- **`std::vector<std::string> channelnames`**

`ImageOutput` Methods

- `ImageOutput::create(string fn)`
  - Static creation of an image output. Checks that provided filename fn is a valid type
- `open(string fn, ImageSpec s)`
  - Opens a file at filename fn and write data to with a particular spec s
- `write_image(TypeDesc t, void* d)`
  - Writes data d to file, given a type t to interpret it with
- `close()`
  - Closes the file and finishes writing

Example Write

```
#include <OpenImageIO/imageio.h>
OIIO_NAMESPACE_USING

...

const char *filename = "foo.jpg";
const int xres = 640, yres = 480;
const int channels = 3;  // RGB

//assume this vector is populated with what we want to write
unsigned char pixels[xres*yres*channels];

ImageOutput *out = ImageOutput::create(filename);
if (!out) {
   std::cerr << “Could not create: “ << geterror();
   exit(-1);
}

//create the ImageSpec that describes how you will write the output data
ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);
out->open(filename, spec);

//it is possible that this TypeDesc does not match the ImageSpec,
//in which case it will convert the raw data into the spec.
//But it MUST match the datatype of raw data
out->write_image(TypeDesc::UINT8, pixels);

out->close();
delete out;
```

`ImageInput` Methods

- `ImageInput::open(string fn)`
  - Static creation of an image input. Checks that provided filename fn is a valid type
- `read_image(TypeDesc t, void* d)`
  - Writes data d to file, given a type t to interpret it with
- `close()`
  - Closes the file and finishes writing

Example Read

```
#include <OpenImageIO/imageio.h>
OIIO_NAMESPACE_USING

...

ImageInput *in = ImageInput::open(filename);
if (!in) {
   std::cerr << “Could not create: “ << geterror();
   exit(-1);
}

//after opening the image we can access
//information about it
const ImageSpec &spec = in->spec();
int xres = spec.width;
int yres = spec.height;
int channels = spec.nchannels;

//declare memory, open and read it
unsigned char* pixels = new unsigned char[xres*yres*channels];

//TypeDesc::UINT8 maps the data into a desired type (unsigned char),
//even if it wasn’t originally of that type
in->read_image(TypeDesc::UINT8, &pixels[0]);
in->close();
delete in;
```

Error Checking

- **`OpenImageIO::geterror()`** returns strings with informative error messages

- Also, so does **`geterror()`** for the **`ImageInput`** and **`ImageOutput`** classes.

- You should check this if an **`open()`**, **`write()`**, **`read()`**, or **`close()`** fails

Error Checking Examples

```
ImageInput *in = ImageInput::open (filename);
if (!in) {
   std::cerr << "Could not open " << filename
    << ", error = " << OpenImageIO::geterror() << "\n";
   exit(-1);
}
```

```
if (!in->read_image (TypeDesc::UINT8, pixels)) {
   std::cerr << "Could not read pixels from " << filename
    << ", error = " << in->geterror() << "\n";
   delete in;
   exit(-1);
}

if (!in->close ()) {
   std::cerr << "Error closing " << filename
    << ", error = " << in->geterror() << "\n";
   delete in;
   exit(-1);
}
```

OpenGL Intro

OpenGL Includes

- At the top of your code that uses OpenGL:

```
#include <GL/gl.h>
```

    - Includes the declarations of main OIIO classes

- For GLUT (which by default includes gl.h as well, so you could be lazy and skip the above):

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

    - On an Mac:

```
#include <OpenGL/gl.h>

#include <GLUT/glut.h>
```

- All library function calls with begin `gl*`, `glu*`, `glut*`, depending on where they come from.

OpenGL / GLUT Structure

`glutInit();`

- GLUT Initialization function, must be called before glutMainLoop();

`glutDisplayFunc(display_func);`

- Sets the callback function (pointer) to be called in the draw loop.

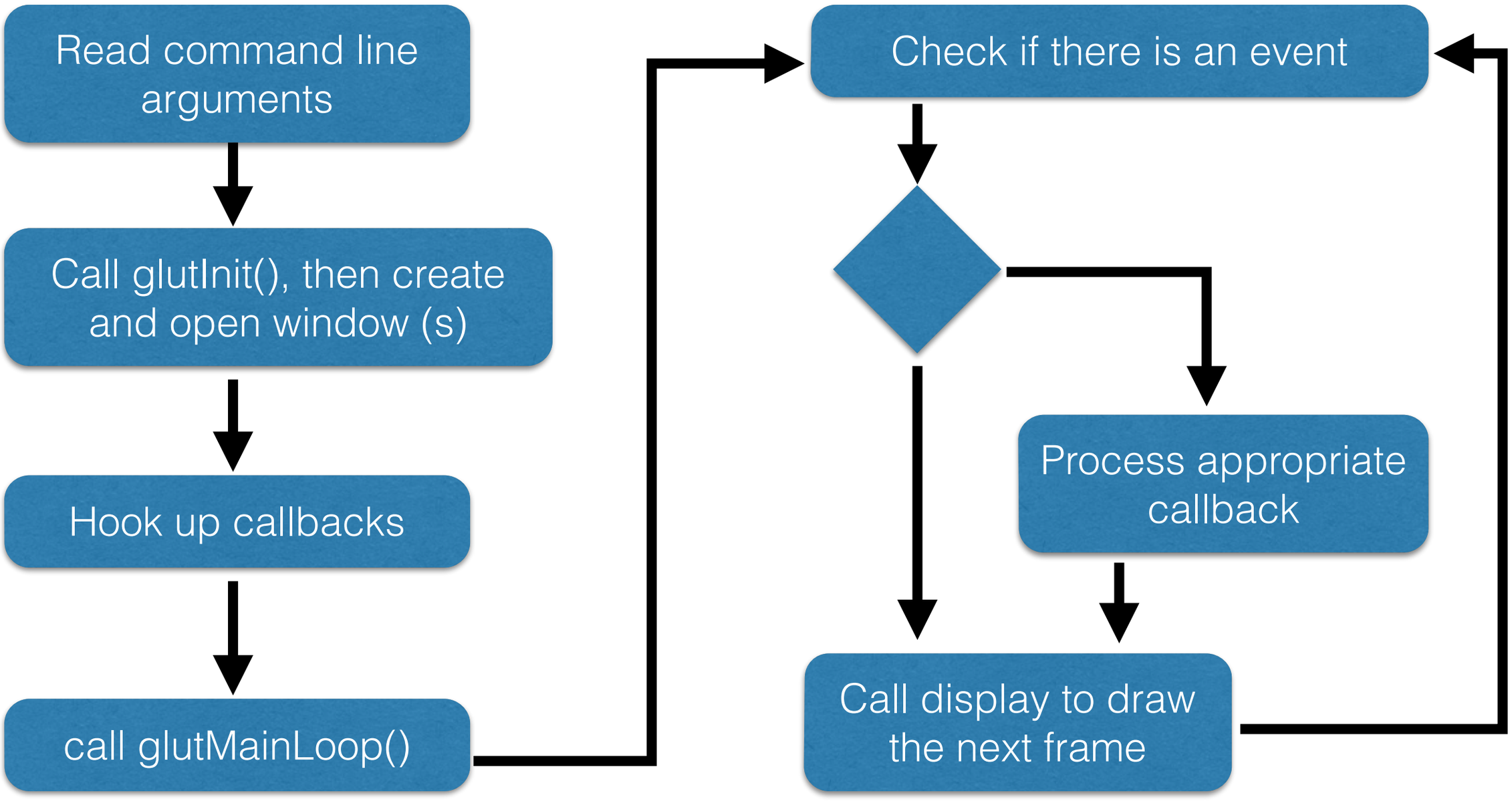
`glutKeyboardFunc(keyboard_func);`

- Sets the keyboard callback — will be triggered if there is a keyboard event. This is also function pointer. Also callbacks for mouse, motion, window resizing, etc..

`glutMainLoop();`

- Goes into (infinite) draw loop

Program Structure with glutMainLoop()



Note: This is greatly simplified. For example, display only redraws if there's a signal too…

glSquare Demo

Lec03 Required Reading

- Hunt, 3.1, 3.3, 4.2, 10.2
- House, 5.1, 5.2