CS 240: Programming in C

Lecture 23: Graphics, Text Encoding

PURDUE UNIVERSITY

Announcements

- Midterm 2 grades are released
  - Stats on Ed announcement
  - Don’t be afraid to ask for a regrade
    - But only if you have a legitimate reason
- Homework 10 due this Friday

Colors

- Computers display colors as a combination of red, green, and blue color channels
  - Each channel can have from 0% to 100% intensity
  - 0% red, 0% green, 0% blue = black
  - 100% red, 100% green, 100% blue = white
  - 100% red, 0% green, 100% blue = magenta
  - 50% red, 50% green, 50% blue = gray
  - etc.

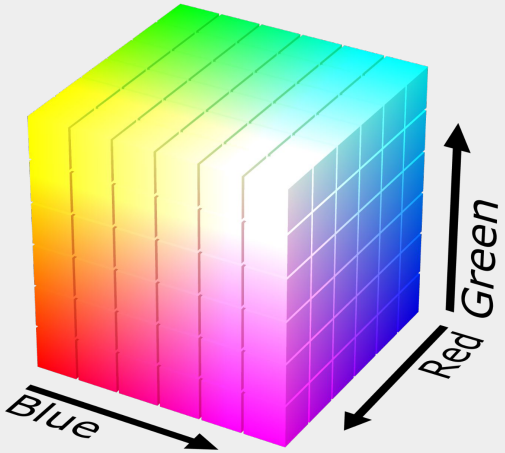
How do we represent a color?

```
typedef struct color {
  unsigned char red;
  unsigned char green;
  unsigned char blue;
} color_t;
```

- Each channel is one byte, or 8 bits
- Altogether we use 24 bits to represent the color
  - aka 24-bit color
- 16,777,216 possible colors
- Values per channel range from 0 to 255

Other color formats

- 16-bit color
  - 5 red, 5 green, 5 blue; ignore the last bit
  - 32,768 colors
- 8-bit color
  - 256 colors
  - Many different representations
    - 6x6x6 color cube + 40 grays
    - or: 3 bits for red, 3 green, 2 blue
  - Alternatively, use a palette



Source: Wikipedia

Color palettes

- Make a table (array) of colors you want to use
- Index into that table with an 8-bit (or lower) value
- You can swap out the palette to change the colors, without changing the image data

Source: sneslab.net

Source: everynesgame.com

What about “alpha”?

- Maybe you’ve heard about 32-bit color, or RGBA
- Colors can come with an extra channel called alpha
- It’s used for transparency when blending images together

```
out_color = alpha * fg_color + (1 - alpha) * bg_color;
```

- You still only get 24-bit color!
  - Your monitor can’t display semi-transparent pixels
  - Alpha is just a blending coefficient

How are images represented?

- An image is just a 2D array of colors

```
typedef struct image {
  color_t *pixels;
  unsigned int width;
  unsigned int height;
} image_t;
```

```
image_t img = /* ... */
img.pixels[img.width * y + x].red = 255;
```

Image representations

- Most APIs that deal with images just use char * instead of a color struct

```
typedef struct image {
  unsigned char *pixels;
  unsigned int width;
  unsigned int height;
  unsigned int bpp;    /* number of bytes per pixel */
  unsigned int pitch;  /* number of bytes per row */
} image_t;
```

```
image_t img = /* ... */
img.pixels[img.pitch * y + img.bpp * x + 0] = 255;
```

Image representations

- Most APIs that deal with images just use char * instead of a color struct

```
typedef struct image {
  unsigned char *pixels;
  unsigned int width;
  unsigned int height;
  unsigned int bpp;    /* number of bytes per pixel */
  unsigned int pitch;  /* number of bytes per ro
} image_t;
```

```
image_t img = /* ... */
img.pixels[img.pitch * y + img.bpp * x + 0] = 255;
```

Be careful with the pixel format!

Images on disk

- There are many image formats
  - bmp, png, jpg, tif, webp
- Most of them are compressed
- Lossy compression
  - jpg, webp
- Lossless compression
  - png, tif, webp
- Uncompressed
  - bmp

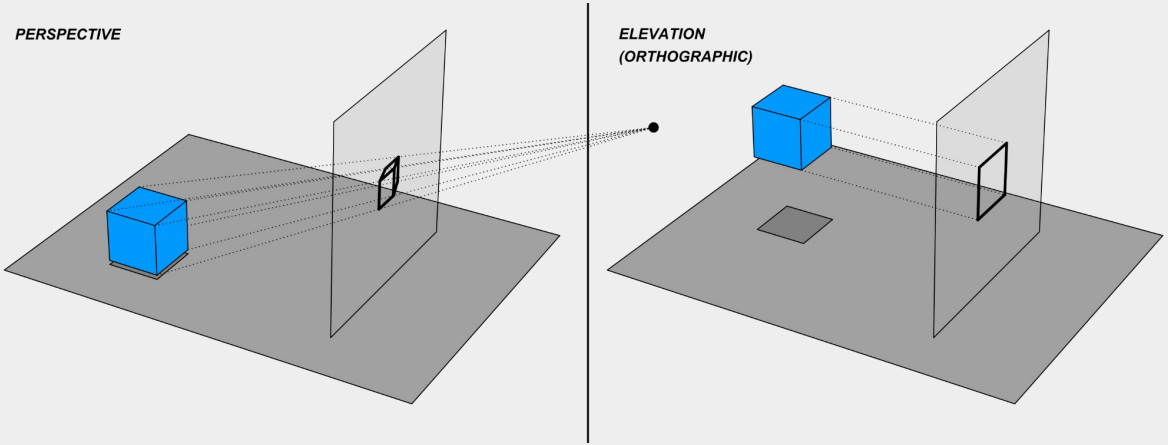
Source: stackexchange.com

Images in memory

- Images get decompressed when read into memory
- Many libraries exist to handle this

3D graphics

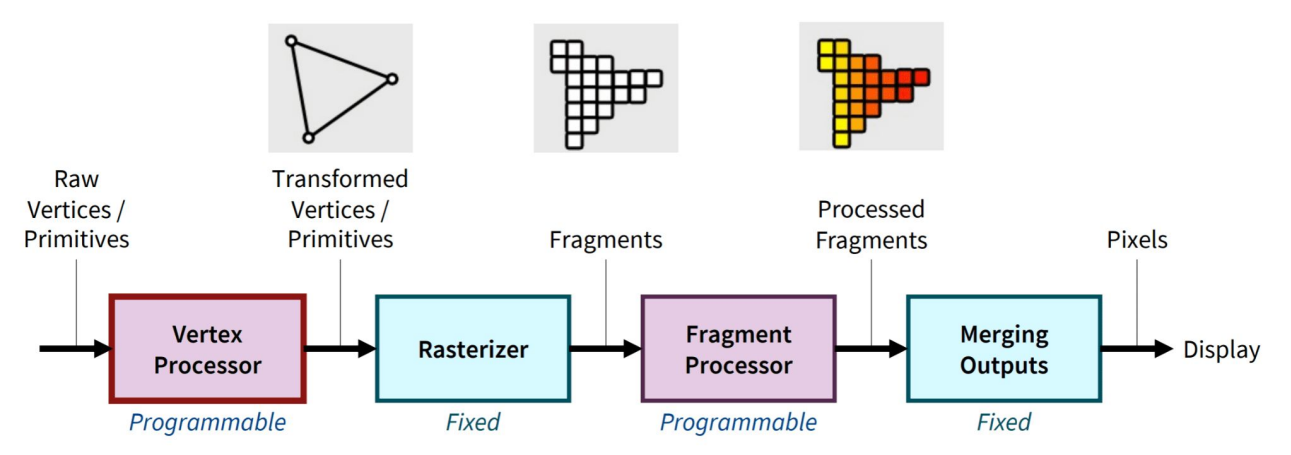
- 3D geometry is represented by points in 3D space (vertices) that are connected by polygons (usually triangles).
- A 2D image of a 3D scene is created by “projecting” the geometry onto an image plane



Source: Wikipedia

3D graphics pipeline

- There are several steps in the process
  - Transform / project vertices
  - Rasterize polygons
  - Compute pixel colors (e.g., lighting, texturing, etc.)



Source: leeyngdo.github.io

OpenGL

- OpenGL is a specification for interacting with a GPU
- Upload vertices, triangles, textures
- Specify rendering behavior using “shader programs”

OpenGL vertex data

```
typedef struct vertex {
  float position[3];
  float color[3];
} vertex_t;

/* Create three vertices with position and color */
vertex verts[3] = {
  { { -0.433, -0.25, 0.0 }, { 1.0, 0.0, 0.0 } },
  { {  0.433, -0.25, 0.0 }, { 0.0, 1.0, 0.0 } },
  { {  0.0,    0.5,  0.0 }, { 0.0, 0.0, 1.0 } },
};
```

OpenGL vertex specification

```
/* Upload vertex data */
GLuint vbuf;
glGenBuffers(1, &vbuf);
glBindBuffer(GL_ARRAY_BUFFER, vbuf);
glBufferData(GL_ARRAY_BUFFER, 3 * sizeof(vertex_t),
             verts, GL_STATIC_DRAW);

/* Tell OpenGL what format it's in */
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                      sizeof(vertex_t), 0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
                      sizeof(vertex_t),
                      (GLvoid *)(sizeof(float) * 3));
```

OpenGL drawing

```
/* Prepare for drawing */
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClearDepth(1.0f);
glUseProgram(shader);

/* Clear the screen */
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/* Draw the geometry */
glDrawArrays(GL_TRIANGLES, 0, 3);

/* Swap the buffers! */
SDL_Flip(screen);
```

I love graphics

- If you’re into graphics, come talk to me!

Text encoding

- Up until now, we've only really considered Latin alphanumeric characters
- But, there are thousands of languages in the world and hundreds of different writing systems
- How do we support them?

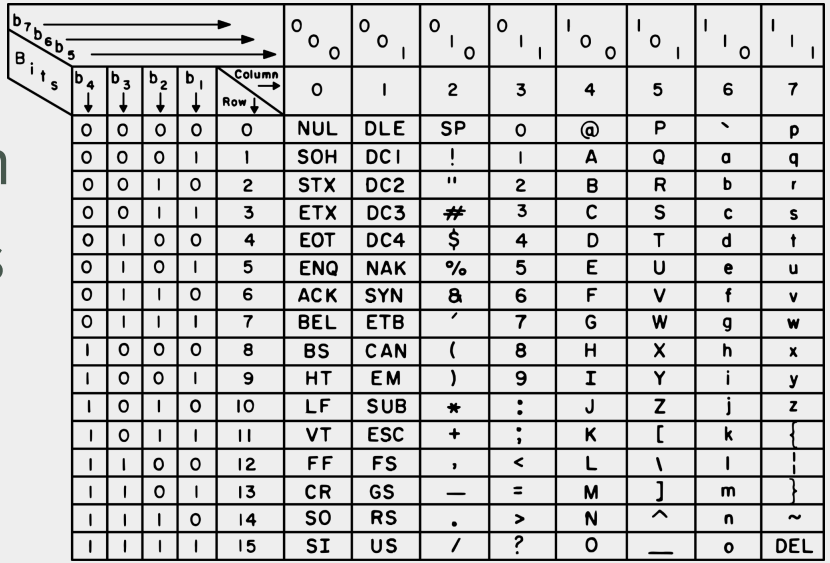
The char datatype

- A char is only 1 byte
  - We can only represent 256 different characters with a char!
- Which numbers represent which characters?
- An encoding specifies this mapping between numbers and characters
- Historically, computers used “code pages”
  - IBM437

Source: Wikipedia

ASCII

- American Standard Code for Information Interchange
- 7-bit encoding
  - Extended ASCII used 8-bits
- Very widely used standard
- Usually what we think of when we talk about character codes



Bits $b_7 b_6 b_5$ →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	—	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Source: Wikipedia

Unicode

- Modern text encoding
- Designed to support ALL the world's writing systems
  - Including emoji, accented characters, even hieroglyphics
- Extendable, under active development
- Widely adopted -- most web pages use unicode

Unicode encodings

- The specification defines 3 encodings
  - UTF-8
    - 1 to 4 bytes per code-point
    - Backwards compatible with ASCII
    - Most common encoding
  - UTF-16
    - 2 or 4 bytes per code-point
  - UTF-32
    - 4 bytes per code-point

Unicode in C

- You're already using it!
- If you do not need to manipulate strings, you don't need to do anything special
  - Reading a Unicode string and printing a Unicode string work exactly the same way
- C provides a type, wchar_t, but it's not very useful
  - 2 bytes per wchar_t, could be used for UTF-16
  - Doesn't help when a code point requires 4 bytes
- Instead, just use char and assume UTF-8

Unicode characters

```
char *str = “こんにちは”;
printf(“%s\n”, str);
printf(“strlen: %d\n”, strlen(str));
```

```
こんにちは
strlen: 15
```

Pitfalls of Unicode in C

- A “single character” can take up to 4 bytes
- Where does each character start and end?
  - The Unicode specification defines this
- strncpy() *et al.* might cut off a multi-byte code point
- Some symbols require multiple code-points
  - e.g. y + ˇ = ǔ
  - These are called “grapheme clusters”
- For robust handling of Unicode, use a library (e.g., ICU)

Terminal colors

- How do we get colored text output?
  - This is not covered by Unicode

```
Your function returns an incorrect value. (-11 points)
```

- Colors (and other effects) are implemented with special byte sequences called ANSI escape sequences
  - We'll talk about that next week!