Before the lecture begins …

- Keep working on Homework 3! Send us questions – we have answers!

Intro to C++

Lecture 6

Floating Point, Pointers

More variables

- Some variables are ***floating-point***. These variables can represent non-integer numbers and very large numbers (using scientific notation).
- However, note that when using graphics such as pixels, ***floating-point*** numbers must often be truncated because pixels are discrete. This is often a time-consuming procedure.

Floating-point Variables

- ***float***: 32 bits
  - 1.17549-38 to 3.40282e+38

- ***double***: 64 bits
  - 2.22507e-308 to 1.79769e308

Example

```
void line_dda(int x1, int y1, int x2, int y2, DWORD color)
{
    float k,b,y;

    k = float((y2-y1))/(x2-x1);
    b = y1 - k*x1;

    // Note: assumes x1 <= x2
    if (int x=x1; x<x2; x++)
    {
        y = k*x + b;
        putpixel(x, int(y), c);
    }
}
```

Casting Floating Point to Integer

- A ***cast*** looks like the following:

```
float xPos = 0.0;
float dxPos = 0.1;
int x = 0;

while (xPos < 640)
{
    xPos += dxPos;
    itcc->SetPixel(xPos, 0, pixelColor);
}
```

- This is typically a bad situation. A truncated version of xPos will be sent to SetPixel, and the conversion itself is time-consuming.

Pointers

- Generally, a function looks like this:

  Output FuncName(*Inputs*)

- Let's take a look at a function we're used to:

  ```
  void SetPixel (unsigned long x, unsigned long y, unsigned long color);
  ```

- Here, we see that SetPixel does not have any outputs (void), but that it does have three inputs – *x*, *y*, and *color*. We can call the function by replacing *x*, *y*, and *color* with either values or variables.

Note: Technically, a function that does not return anything is called a *subroutine*.

Pointers

- Pointers are often considered one of the most difficult parts of C/C++ programming to understand.
- A ***pointer*** is an address in memory where data lies. In 32-bit processors, ever wonder why there was an upper limit of using about 4Gb of memory? A 32-bit address can refer to about 4.3 billion bytes, assuming that the smallest addressable unit is the byte.

Pointers

- When you allocate memory to store a variable, this memory resides somewhere in a user’s RAM. This RAM is addressable via a 32-bit address into your RAM. This address is simply a number referring to a certain byte in memory.
- Think of there being 4 billion mailboxes numbered sequentially in a long row. If we receive a number referring to one of these mailboxes, then we can go directly to the mailbox in question and retrieve whatever is stored at that address.

Pointers in C/C++

- The asterisk next to a data type means that the data type is a ***pointer***. Pointers are ALWAYS 32 bits in length because an address is ALWAYS 32 bits long (for 32 bit processors).
- A generic pointer to ANY memory address is void *. To declare a variable of this type:

```
void *myPointer = NULL;
void *videoBuffer = 0xA0000000;
```

- NULL is often defined as 0, meaning that the pointer DOES NOT POINT to a memory address. It is a good idea to always set pointers to NULL when they are not in use.

Data Type Pointers

- When we place a variable type in front of the asterisk, the data allocated for the memory address is ALWAYS 32 bits. The data type in front of the * merely means that we can correctly interpret the data stored at that address as that data type. For example:

```
unsigned int *videoBuffer = 0xA000000;
float *bankSalary = NULL;
```

Pointing to a Variable

• When we normally reference a variable, we grab that variable's *value*. When we make a pointer that points to a certain variable, we want the *address* of the variable stored in the pointer, not the variable's value.

• To get the address in memory that a variable is stored at, we use the ampersand symbol (&):

```
int totalMonsters = 100;
int *ptrTotalMonsters = &totalMonsters;
```

• Do you see the difference? If we did not use the ampersand, then the pointer would point to whatever was stored at the hundredth byte in memory!

Using a Pointer

• When we refer to a ***pointer***, then we are referring to an address stored in memory. If we place an asterisk in front of the ***pointer*** variable, then we are referring to what is *stored* at the address stored in the pointer:

```
int totalMonsters = 100;
int *ptrTotalMonsters = &totalMonsters;

cout << totalMonsters << endl;          // 100
cout << ptrTotalMonsters << endl;       // [a memory address]
cout << *ptrTotalMonsters << endl;      // 100
```

Passing by Value

• When we normally call a function, all of the variables we pass to the function are *copied*, and these copies are used within the function body.

• If this occurs, how would we ever actually modify the values of variables passed to the function?

• We can use pointers! Even if these are copied, we still have the *address* of the original variable in-tact so we can still modify its contents.

Passing by Address

- Note how this technique of *passing by address* also saves time by not copying the entire object being passed, instead only copying 32 bits per parameter, regardless of how large the parameter is.

- Here's how it works (addNum2 is correct):

```
void addNum1(int num1, int num2, int *sum)
{
          sum = num1 + num2;            // Stores the total as the ADDRESS sum points to!
                                        // DO NOT DO THIS!!!
}

void addNum2(int num1, int num2, int *sum)
{
          *sum = num1 + num2;           // Stores the total as the VALUE of sum
}
```

Passing By Address

```
int main(void)
{
        int num1 = 5;
        int sum = 0;
        int *ptrSum = NULL;

        AddNums(num1, 10, &sum);      // Both statements are EQUIVALENT!

        ptrSum = &sum;
        AddNums(num1, 10, ptrSum);

        return 0;
}

int AddNums(int num1, int num2, int *sum)
{
        *sum = num1 + num2;
}
```

Passing Arrays to Functions

- Here is how we might pass an array to a function:

```
void DrawEnemies(long x[], long y[], long numEnemies);
```

- It would be called as follows:

```
long xPos[5], yPos[5];
DrawEnemies(xPos, yPos, 5);
```

Lighting a Pixel

- Let's assume we have a pointer to the start of video memory:

```
unsigned int *vidBuf = NULL;
```

- Knowing that each 32-bit pixel occupies four bytes of memory, we can compute the location where we should store a certain pixel and place the color at that spot. We do this using ***pointer arithmetic***.

Pointer Arithmetic

- What if we execute:

```
*vidBuf = colorRed;
```

- This will store a pixel in the upper left-most corner of the screen (hopefully, the color red!) since we just stored a 32-bit value there!

- What if we execute:

```
*(vidBuf + 1) = colorRed;
```

- This will move forward one unsigned int (4 bytes) from the start of vidBuf and store the color red there, lighting up the second pixel on the top row.

Pointer Arithmetic

- We know that the video buffer is made up of xRes by yRes pixels, and that there are xRes pixels per row. However, the memory is all ***linear***, meaning that although we imagine it being stored two-dimensionally, it is actually like one big one-dimensional array!
- Thus, we can write an equation that specifies where our pixel should be stored. Since there are xRes pixels per row, to get to the *start* of the right row, we just multiply the y by the xRes, then add on x (assuming that each pixel is 32 bits):

```
*(vidBuf + y*yRes + x) = colorRed;
```

The SetPixel Function

- Thus, we obtain the following simple SetPixel32 function:

```
void SetPixel32(ULONG x, ULONG y, ULONG color)
{
    *(itcc->VidBuf32() + y*itcc->YRes() + x) = color;
}
```

TODO

- **Do your best to finish and submit ITCC_HW3.zip by 6:00am PST Wednesday, July 19. If you need help, just ask for it!**
- Download ITCC_HW4.zip from the site (will be available soon) http://www.pclx.com/itcc/, and complete the homework exercises, emailing them (FOR THIS WEEK ONLY) to itcc_teachers@pclx.com. Please do not resubmit solutions, even if they are revised. All homework must be submitted by 6:00am PST Wednesday, July 14.
- Look over the slides for the seventh lecture.
- If you finish with the homework, experiment!