7. Inheritance, 2D graphics

Ján Dugáček

September 20, 2017

Table of Contents

2D Graphics: How does it work?

- An image is typically compressed using a long and complicated algorithm that should better be avoided using a library like FreeImage
- It is planned that C++ would be added `io2d` to deal with this, but the proposal is not finished yet
- The function to read and save a `.bmp` file is quite simple and available
- An opened image is a 3D array of type `unsigned char` (or `uint8_t`) with dimensions *height*, *width* and *colour* (2D without *colour* if it's greyscale)
- All operations are done on the array until it's saved and this array is the same for all libraries (though it might be flipped)

A black image

```
struct image300x200 {
        unsigned char[300][200][3];
        unsigned char& at(int x, int y, int col) {
                return char[x][y][col];
        }
}
image300x200 pic;
for (int i = 0; i < 300; i++)
        for (int j = 0; j < 200; j++)
                for (int k = 0; k < 3; k++)
                        pic.at(i, j, k) = 0;
```

- To make the image black, we set all values to 0
- Arrays of higher dimension are not very practical when passed as function arguments (a 3D array of `unsigned char` is *not* the same as `unsigned char***`), so it's wrapped in a `struct`
- Images are large objects, large images might not fit on stack and it's better to have them dynamically allocated

A rectangle

```
// pic is as written on previous slide
for (int i = 100; i < 200; i++)
        for (int j = 66; j < 133; j++)
                pic.at(i, j, 1) = 255;
```

- Drawing a rectangle is as simple as locating the pixels whose colour will be changed
- The colour will most likely be green, but it may depend on implementation

A line

```
// pic is as written on previous slide
auto line = [&] (int x1, int x2, int y1, int y2) {
        int length = sqrt((x2 - x1) * (x2 - x1)
                + (y2 - y1) * (y2 - y1));
        float xIncr = (x2 - x1) / length;
        float yIncr = (y2 - y1) / length;
        for (int i = 0; i <= length; i++)
                pic.at(x1 + xIncr * i,
                        y1 + yIncr * i, 2) = 255;
}
```

- This is not the most efficient algorithm, by the way

Fast square root

```
float fsqrt(float x) {
        float xhalf = 0.5f * x;
        int i = *(int*)&x;
        i = 0x5f375a86 - (i >> 1);
        x = *(float*)&i;
        x = 1 / (x * (1.5f - xhalf * x * x));
        return x;
}
```

- This very fast algorithm computes the square root with a decent precision (better than 1%), the imprecision is not visible to naked eye
- Don't ask why it works or how it works
- It was invented by Id Software for game Quake for normalising vectors (apparent colour of a surface is the light intensity multiplied by the surface's colour multiplied by the normalised dot product of the vector of incident light and normal of the surface)

Circle

```
// pic is as written on previous slide
auto circle = [&] (int x, int y, int radius) {
        for (int i = 0; i < radius; i++) {
                int width = fsqrt(radius * radius - i * i);
                for (int j = -width; j <= width; j++) {
                        pic.at(i + x, j, 0) = 255;
                        pic.at(i - x, j, 0) = 255;
                }
        }
}
```

- This algorithm uses the circle equation $f(x) = \sqrt{r^2 - x^2}$

Exercises

1. Use the file available with these slides to create an image class that has methods for drawing of dots, lines, rectangles and circles
2. Add methods for drawing ellipses, arrows, empty circles and empty triangles
3. Challenge: Add a method to draw a filled triangle

Advanced:

1. Use the file available with these slides to create a program that reads a file containing data in two columns, x and $f(x)$ and draws a graph of the function into a picture

Inheritance: Why?

```
struct a {
        int val;
        void increment() { val++; }
};
struct b {
        int val;
        int multiplier;
        int operator*(int n) { return multiplier * n; }
};
b* orig = new b;
a* changed = (a*)orig;
```

- If we convert `b` to `a`, nothing bad happens, the `increment()` method works as it should, the `multiplier` field is not changed
- If `b` had some dynamically allocated stuff, it would leak because its destructor would not be called

Inheritance: Why? #2

```
struct a {
        int val;
};
struct b {
        float val;
};
struct c {
        int asA;
        float asB;
};
c* orig = new c;
a* changed = (a*)orig;
b* changed2 = (b*)&orig.asB;
```

- Now it gets even more impractical

Inheritance

```
struct a {
        int val;
        void increment() { val++; }
};
struct b : public a {
        int multiplier;
        int operator*(int n) { return multiplier * n; }
};
b* orig = new b;
a* changed = orig;
```

- Inheritance is a way to expand a class to a new one that has added functionality
- In this case, `a` called *parent class* and `b` is called *child class*
- Conversion to parent class is done implicitly

Inheritance #2

- Inheritance is mostly `public`, but there is also `private` inheritance, that makes all parent classes' contents private
- Child classes can't access their parents classes' private methods and attributes (they are a different class), but can access their protected members (they are the same object)
- Friend methods and classes are not inherited
- Conversion from child class to parent class is not checked in any way and may cause problems if the new type is incorrect

```
b* orig = new b;
a* changed = orig;
b* reconstructed = static_cast<b*>(changed);
```

Virtual methods

```
struct a {
        int val;
        void increment() { val++; }
};
struct b : public a {
        void increment() { val += 2; }
};
b orig;
a& changed = orig;
changed.increment();
```

- In this case, the compiler calls a's method because the type it's looking through is a

Virtual methods #2

```
struct a {
        int val;
        virtual void increment() { val++; }
};
struct b : public a {
        virtual void increment() { val += 2; }
};
b orig;
a& changed = orig;
changed.increment();
```

- In this case, the compiler calls `b`'s method because it checks the underlying type in runtime and learns it's `b`
- Virtual function calls are inherently slower because they require additional checks and can't be inlined
- It allows us to create a child class from an existing class and use an existing function on it that will call our code
- Note: it's not necessary to declare the child's method as virtual, but it makes the code clearer

Pure virtual methods

```
struct a {
        int val;
        virtual void increment() = 0;
};
struct b : public a {
        virtual void increment() { val += 2; }
};
```

- In this case, class a does not even have a definition of the virtual method, so it's called *pure virtual*
- Class a is called *abstract* and cannot be created, only other types can be changed to it; it is only a way to use multiple classes by the same code

Constructors and destructors

```
struct a {
        int val;
        a(int set) : val(set) {}
        virtual ~a() { val = 0; } // destroy the evidence
};
struct b : public a {
        int val2;
        b(int set1, int set2) : a(set1), val2(set2) {}
        virtual ~b() { val = 0; val2 = 0; }
};
```

- Child classes' constructors may call parent classes' constructors to construct the parent class (is mandatory if the parent class has no default constructor)
- We need to call the right destructor, so **all destructors must be virtual** if inheritance is used

Exercises

1. Create a `ball` class that has its direction, speed and weight as attributes, a `bigBall` class that has all the properties of `ball`, but also size and aerial friction coefficient
2. Write a program that calculates the path of thrown balls, acting differently if the ball is small enough to neglect the aerial friction or not

Advanced:

1. Create a `particle` class that has its direction, speed and size as attributes, an `atom` class that has all the properties of `particle`, but also mass and internal energy and a `photon` class that has all the properties of `particle`, but also wavelength
2. Write a program that simulates interaction of thousands of atoms and photons in a cube limited by mirror walls, making use of inheritance (neglect photon momentum, assume that atoms can absorb anything and will radiate it out into a random direction after some time)

Homework

- Write a library for manipulating images; its main object is `image`, it can contain other `images`, squares, circles and lines, all at any position; squares, circles and lines also have colours besides their geometric properties; it must have a method to create an image of all the content on it (and save it)
- Use inheritance, I recommend using a class `abstractShape` (attributes position, scale,) that has subclasses `shape` and `image`, `shape` has further subclasses `square`, `circle` and `line`

Advanced homework:

- Same as the regular one, but implement also a `triangle` class, add scaling to `image` and transparency to all
- You can also add functionality to delete all shapes whose container was deleted