Programming in C++ / *FASTTRACK TUTORIALS*

PART 12: “Classes”

Introduction

In this episode we will build on the code from episode 11. In case you need a starting point, here is some code that will do for today:

```
#include "precomp.h"

Surface tiles( "assets/nc2tiles.png" );
Sprite tank( new Surface( "assets/ctankbase.tga" ), 16 );
int px = 0, py = 0;

void Game::Init() {}
void Game::Shutdown() {}

char map[5][30] = {
    "kc kc kc kc kc kc kc kc kc kc",
    "kc fb fb fb kc kc kc kc kc kc",
    "kc fb fb fb fb fb kc kc kc kc",
    "kc lc lc fb fb fb kc kc kc kc",
    "kc kc kc lc lc lc kc kc kc kc" };

void DrawTile( int tx, int ty, Surface* screen, int x, int y )
{
    Pixel* src = tiles.GetBuffer() + 1 + tx * 33 + (1 + ty * 33) * 595;
    Pixel* dst = screen->GetBuffer() + x + y * 800;
    for( int i = 0; i < 32; i++, src += 595, dst += 800 )
        for( int j = 0; j < 32; j++ ) dst[j] = src[j];
}

void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    for( int y = 0; y < 5; y++ ) for( int x = 0; x < 10; x++ )
    {
        int tx = map[y][x * 3] - 'a', ty = map[y][x * 3 + 1] - 'a';
        DrawTile( tx, ty, screen, x * 32, y * 32 );
    }
    if (GetAsyncKeyState( VK_LEFT )) { px--; tank.SetFrame( 12 ); }
    if (GetAsyncKeyState( VK_RIGHT )) { px++; tank.SetFrame( 4 ); }
    if (GetAsyncKeyState( VK_UP )) { py--; tank.SetFrame( 0 ); }
    if (GetAsyncKeyState( VK_DOWN )) { py++; tank.SetFrame( 8 ); }
    tank.Draw( screen, px, py );
}
```

There are some problems with this code, and we will fix those problems using *classes*, which allow us to encapsulate functionality.

Objects

Your game world consists of objects. Take the last episode: we have a player tank, a backdrop (which consists of tiles), a screen that we draw to. Later on, we might want to add bullets, enemies, more levels and so on. This is a list of *things*, not a list of operations. You could actually argue that the most logical way to start thinking about what a game will do is actually: what *objects* do I need? And, slightly more detailed: what do these objects do? And what properties do these objects have?

Note that this is something you have already been using. Consider the default code in the Tick function:

```
void Game::Tick( float deltaTime )
{
    // clear the graphics window
    screen->Clear( 0 );
    ...
}
```

When you open up `game.h` you will see that your game has a variable `screen`, which is an object. The type of this object is 'Surface'. In the default template code we are telling this surface to do things: it's asked to clear itself, and to print something. A surface can do more: you can find out what by looking at `surface.h`, line 37. There you will see that a surface can also `Resize` itself, amongst other things. A Surface also has some properties: a width, a height, a bunch of Pixels, and some other things. The things that the Surface can *do* are called *methods.* The properties of an object are called *member variables*, or just *properties*.

Tank object

Let's apply this in a more interesting way. In the previous tutorial, you added a player-controlled tank, with collision detection, to a tile-based backdrop. In this tutorial, we'll make the tank move by itself, using three simple rules:

1. If the tank *can* move to the right, it will
2. If the tank *cannot* move the the right, it will:
    a. Move up, if it is in the lower half of the screen;
    b. Move down, if it is in the top half of the screen.
3. When the tank reaches the right side of the screen, it is respawned at the left side.

The main object that we will be working on is a tank:

```
class Tank
{
public:
};
```

Put this code in game.cpp, right above `Game::Init()`. Once you have that in place, you can create your tank:

```
Tank mytank;
```

So: you can now make variables of type `Tank`. The object does not yet have any properties though, and it can't do anything yet… Our particular tank will need to perform one task: move. We want to draw it once every time `Game::Tick` is executed, so when it moves, it should do one step. In terms of properties for our tank, there are a few obvious ones: position (x and y), and orientation. Considering this, the tank class now becomes:

```
class Tank
{
public:
	void Move();
	int x, y, rotation;
};
```

When we first create a tank, we need to set its x and y and rotation. Until now, you would have done this in the `Game::Init()` function, but now there is a better way. It is called a *constructor*, and it looks like this:

```
class Tank
{
public:
	Tank()
	{
		x = 0;
		y = 4 * 64;
		rotation = 0;
	}
	void Move();
	int x, y, rotation;
};
```

The good thing is that when you *create* your tank (by creating a variable of type `Tank`), the constructor is executed. So, basically the constructor is the Init function of your class. And, best of all, each class can have its own, and it's executed automatically for you.

Tank behavior

Now that we have a tank (called `mytank`), we can use it. First of all, we can access its properties: `mytank.x`, `mytank.y` and `mytank.rotation`. We can also make it do something. Add the following line to your `Game::Tick` function:

```
mytank.Move();
```

When you compile the program, you will get an error: we told C++ that there exists a tank, and that it can be told to `Move()`, but we didn't specify what happens in that case. Let's fix that. In the tank class, replace `void Move();` by:

```
void Move()
{
	x++;
	if (x > 800) x = 0;
	tank.Draw( screen, x, y );
}
```

Note that this does not implement all the rules specified earlier, we'll safe that for the assignment. ☺ When you try to compile this code, you will get an error. The above code assumes that `screen` can be used in our tank class, but apparently it can not… There is a reason for that: `screen` belongs to another object, which is called `Game`, and we can't just access it. Even though this is annoying right now, this is actually good: member variables belong to their own object. This allows us to use x and y in a tank class, and x and y in a bullet class as well: the tank x and y will be referred to (in our program) as `mytank.x` and `mytank.y`; x for a bullet might be `mightybullet.x`. And, if there are multiple bullets, they all have their own x and y: `bullet1.x`, `bullet2.x`, and so on.

That doesn't solve the surface problem, obviously. Lucikly, the solution is not hard. The game does know about `screen`, and the game moves the tank. So, the game should tell the tank about the Surface as well. Like this:

```
mytank.Move( screen );
```

And, the tank should listen to that:

```
void Move( Surface* gameScreen )
{
	x++;
	if (x > (16 * 64)) x = 0;
	tank.Draw( gameScreen, x, y );
}
```

Note that it's not called `screen` anymore, because it is a different variable now: it's a function argument. This time, all is well, and the tank does its limited behavior, which you get to fix in the assignment.

Conclusion

A few final words before you start hacking away:

You have been using classes without knowing. There is a class `Game`, a `Surface`, and a `Sprite`. The template has some more objects, which you didn't use yet. Having a class means nothing by the way; you merely tell C++ what something looks like. To actually create something, you create a variable of that type, such as `mytank`. This variable is called an *instance.* Each instance has its own set of *member variables*, as defined in the *class definition*.

You use classes for many reasons:

- It allows you to think in high-level concepts before you get to the details;
- It groups data on a per-object basis rather than in one big messy pool;
- It groups data and the code that operates on that data.

We will dive deeper in the subject at a later time.

For now, you know enough for the…

Assignment

BASIC:

1. Correctly implement the three rules for the tank object, using the collision code and full-screen map from episode 11.

2. Make an array of tanks. Each tank starts at a random tile, and executes the same rules.

INTERMEDIATE:

3. Expand task 2 so that no two tanks start at the same tile.

4. Move the tank class to its own set of files: a `tank.h` for the class definition, and a class.cpp for the implementation of the functions. Add `tank.h` to `precomp.h` to make it accessible from `game.cpp`.

HARD:

5. Expand task 3 so that no two tanks occupy the same tile while walking the scene. In other words: thanks should detect each other and avoid collisions.

6. Make a class for the tile map rendering code so that you can separate its data (tiles, width, height, …) and functionality (draw, collisions, …) from the game code. This will also allow you to easily reuse the code in another project.

END OF PART 12

Next part: "Data Structures"