Game Engine Tutorial

Mr. Miyoshi's Beginning Programming / Game Programming Class

Contents

Introduction

The sheer joy of making things... the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles... the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination.

Frederick P. Brooks

Welcome!

This tutorial will walk you through the process of creating your own game, starting with a barebones project and building up to a complete game with user input, game objects, collisions, and more. Along the way we'll explore each piece of the game in-depth. After working through this tutorial you'll be able to use the skills you've learned to make any kind of game you'd like. Let's get started!

How this document is structured

Actually, hold on a second. Before we begin, let's briefly go over the structure of this document. First, the introduction is going to familiarize you with the base project code. Then the document is broken into "parts". Each part is a small unit where we'll build out some piece of the game. The source code for each part can be found in the same place as the base project, which we'll discuss momentarily.

While you could just go and grab the source code for any part, or copy and paste it from this document, if you want to get the most benefit from this tutorial you should read through each part and write the code yourself. The code for each part is there to help you if you get stuck, so you have a baseline reference to compare your work to.

Also, feel free to experiment. Make changes. Add things. Make mistakes. Backtrack. The more you play with the code the more you will learn. You could use the code in this tutorial as a basis for your own games, or you could try something completely different. It's up to you.

At the very end of the document there is a reference section. The first time we mention a concept that has an entry in the reference section, it will have a link to that entry. If you're using Adobe Reader, you can hold ALT and press the back arrow to return where you were before you clicked the link.

Finally, there may be stuff in this tutorial that you already know, and stuff that you may find utterly confusing. Don't worry if you don't understand everything as you go along. Just keep working at it, and by the end you will have a working game, and will know enough to build your own games. You can always come back after you're done and review this tutorial, dig into the reference section, ask for help, or look up stuff online.

OK. Let's go! For reals!

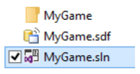
Getting the base project

First things first. You'll need a copy of the base game project. To find it, double-click on the RDC folder on your desktop, then navigate to Public_Programming\GameProgramming\SFML Game Examples\CreatingSFMLGame. There you will find a folder named **Base project**, and inside that you will find **MyGame**.

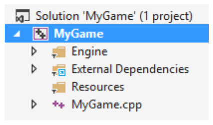
Copy **MyGame** to your **H:** drive. Note that you will need to make some modifications to the base project before it will build and run. See SFMLProjectModification.docx (in the folder above) for details on how to do this.

A quick tour of the base project

OK, now that you have copied the base project to your own folder somewhere, let's open it up and take a look. Double-click on the solution file:



Visual Studio should start up, and you'll see the solution loaded:



Let's take a look at what's inside:

- **MyGame.cpp** is the main source file of the game. We'll be looking at it shortly.
- **GameScene.h** and **GameScene.cpp** together make up the main "scene" of our game. We'll be talking a lot about scenes in just a little while.
- **Engine** contains code for a simple game engine, built on top of the awesome SFML library. We're going to use this engine to build our game. Feel free to peek under the hood and even change things. You can learn from the engine code, and it's *your* engine code now, so if there's something you think it needs, add it!
- **Resources** is a place for you to put images, sounds and fonts you want to use in your game.
- **External Dependencies** shows you the various source files that are needed to build your code. Visual Studio shows this to you in case you are interested, but you can safely ignore it.

MyGame.cpp

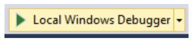
Double-click on **MyGame.cpp**. Let's take a look! Don't worry about understanding it yet, we'll go through it step-by-step.

MyGame.cpp

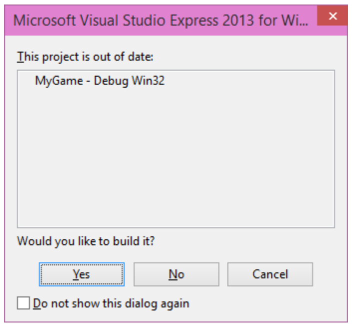
```
#include "Engine/GameEngine.h"
#include "GameScene.h"

const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 600;

const std::string WINDOW_TITLE = "My Awesome Game";

int main()
{
         // Seed the random number generator.
         srand((int)time(NULL));

         // Initialize the game.
         GAME.initialize(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_TITLE);

         // Create our scene.
         GameScenePtr scene = std::make_shared<GameScene>();
         GAME.setScene(scene);

         // Run the game loop.
         GAME.run();
```

```
        return 0;
}
```

We'll talk about each line of code in just a minute. First, let's run this and see what happens. Click the "Local Windows Debugger" button (the one with the green triangle) to see it in action:



You may get a little dialog asking if you want to build your out-of-date project. Usually when you see this dialog you want to click "Yes".

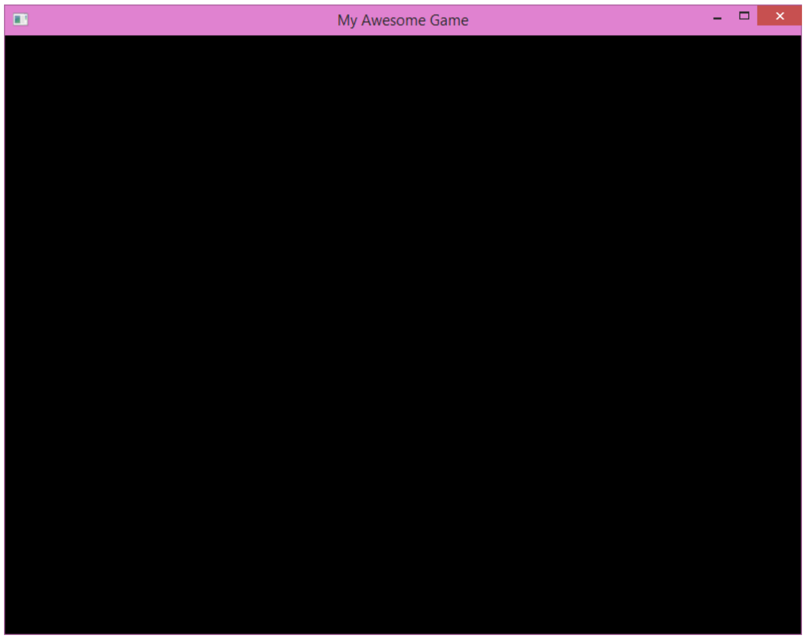

Microsoft Visual Studio Express 2013 for Wi...

This project is out of date:

MyGame - Debug Win32

Would you like to build it?

Once it's running, you should see something very exciting: A completely blank window!



Huzzah! A blank canvas on which to paint your masterpiece. Now, let's take a look at what's going on behind-the-scenes in this game. Close the game by clicking on the close button (✕) to return to the code.

Including the game engine source and the game scene

The first two lines in **MyGame.cpp** are include directives:

```
#include "Engine/GameEngine.h"
#include "GameScene.h"
```

GameEngine.h brings all the game engine files into your project. Instead of having to include each header file in the game engine, you can just include this one and it will include the rest for you.

GameScene.h defines a scene, which we'll discuss shortly.

Constants

Next we have a couple of constants:

```
const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 600;

const std::string WINDOW_TITLE = "My Awesome Game";
```

`WINDOW_WIDTH` and `WINDOW_HEIGHT` specify the size of the window you saw when you ran the game. `WINDOW_TITLE` is a string that specifies the text that's shown in the title bar of the window. Feel free to change width and height to whatever size you want, and definitely give your game a better title!

Note that **`string`** is part of the C++ standard library, and everything in the standard library is inside the **`std`** namespace, so we preface it with **`std::`**.

The main function

Next up, the most important function of them all, the main function! It's where your program begins, it's where you program will end, it starts on line 9, it ends on line 25, it's got stuff inside of it!

```
int main()
{
        // Here there be stuff!
}
```

Uh... let's look at the stuff!

Seeding the random number generator

First up, we seed the random number generator. We only need to do this once in the entire program. If we don't, we'll get the same "random" numbers every time we run the game.

```
// Seed the random number generator.
srand((int)time(NULL));
```

Initialize the game

Next we call the **`initialize`** function of our `GAME`. Like seeding the random number generator, this is something we only have to do once.

```
// initialize the game
GAME.initialize(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_TITLE);
```

`GAME` is a special class in our game engine. Its primary responsibility is to run our game loop. The call to **`initialize`** creates the window all our game objects will be drawn to. This is the blank window you saw when you ran the program.

Creating the scene

Our game engine uses the concept of *scenes*. Each scene stores a bunch of stuff we want to have in our game. We might have different scenes for the start menu, the main game, the "game over" screen, and so on. We'll learn more about scenes as we go along.

We've created a class called `GameScene` to hold all the stuff for our main game scene. In Part 2 we'll begin adding stuff to it and seeing it on the screen.

```
        GameScenePtr scene = std::make_shared<GameScene>();
```

This line of code might not make a lot of sense when you first look at it. That's OK. As we go through the process of making our game we'll cover everything in this line of code, and you'll not only understand it but be able to write lines like this yourself. For now, it's enough to know that this line creates an instance of the `GameScene` class. (Remember, there are many ways to create instances – this is just one!)

```
        GAME.setScene(scene);
```

Then we tell `GAME` about the scene. The game engine can switch from one scene to another. When you call `setScene`, the transition will happen. You also have to call `setScene` before you can run the game at all.

Run the game!

Once our scene is created and `GAME` has been told about it, we can run our game! We do this by asking `GAME` to run the game loop.

```
        // Run the game loop.
        GAME.run();

        return 0;
```

The game loop will keep on running until the window is closed. Once that happens, our `main` function returns and the program exits.

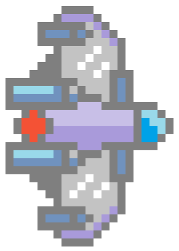
So that's everything in our barebones MyGame.cpp. If you don't understand everything in the code, that's good. It means you're perfectly normal. ☺ As you work through the process of building the game bit by bit, things will begin to click. So, let's start building!

Part 1: Adding a Sprite

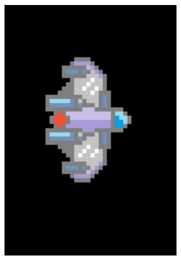
Talk is cheap. Show me the code.
Linus Torvalds

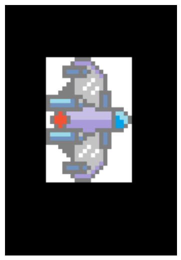
Creating the sprite

Let's get our player sprite on the screen. For this, you can create your own sprite in a program like Paint or Photoshop or Paint.NET, or you can use this one, which you can find in the same place you found the base project, inside the **Art** folder:



One thing to keep in mind if you draw your own sprites is transparency. If you don't use a transparent background, you'll see it in your game. Here's the difference:



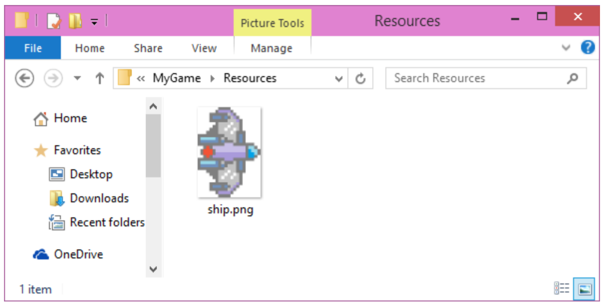


With transparency   *Without transparency*

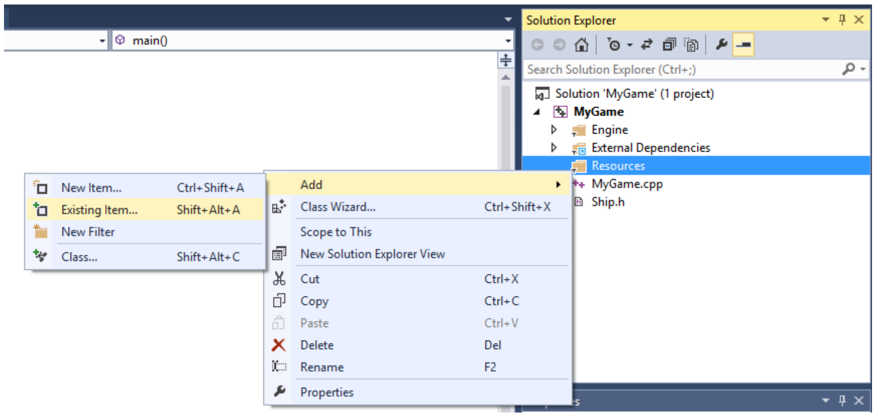
Paint is not great for making transparent sprites, but Photoshop and Paint.NET support transparency. Also, to have transparent sprites you'll have to save to an image file format that supports it, such as PNG. JPEG does not support transparency.

Adding the sprite to your project

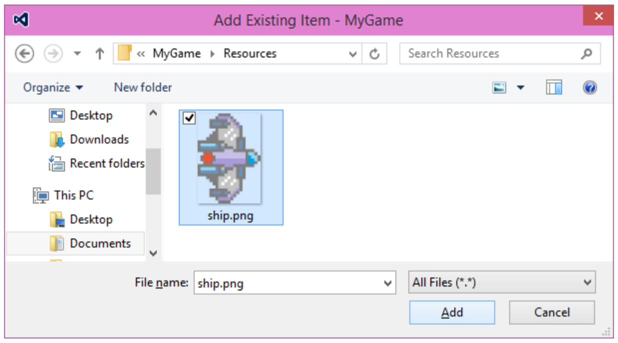
Inside your solution folder (where your source code lives), you'll find the **Resources** folder. Copy the image file you want to use into this folder.



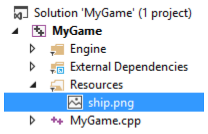
Next, add the sprite in Visual Studio. Left click on your **Resources** folder and select **Add / Existing Item**.



Then navigate to your **Resources** folder and click **Add**.



You should now see the sprite in your solution, inside the **Resources** folder:



And there you have it. You can use this pattern to add any resource into your game, including sounds, sprites and fonts.

The GameObject class

OK, ladies and gentlemen. It's time to get **classy**. Our game engine is object oriented, so everything we create in our game (monsters, space ships, bullets, ponies, etc.) will be represented by a class. And in our game engine, one of the most important classes is `GameObject`. You'll find it inside the **Engine** folder in your project. Let's take a quick look at **GameObject.h** now, and then we'll go over it in a bit more depth. Again, don't stress if you don't understand everything that's going on – we'll revisit a lot of it as we make our game.

GameObject.h

```
#pragma once

#include <set>
#include <string>
#include <memory>
#include <SFML/Graphics.hpp>

// This class represents every object in your game, such as the player, enemies, and so on.
```

```
class GameObject
{
public:
	// Tags let you annotate your objects so you can identify them later
	// (such as "player").
	void assignTag(std::string tag);
	bool hasTag(std::string tag);

	// "Dead" game objects will be removed from the scene.
	bool isDead();
	void makeDead();

	// update is called every frame. Use this to prepare to draw (move, perform AI, etc.).
	virtual void update(sf::Time& elapsed) {}

	// draw is called once per frame. Use this to draw your object to the screen.
	virtual void draw() {}

	// This flag indicates whether this game object should be checked for collisions.
	// The more game objects in the scene that need to be checked, the longer it takes.
	bool isCollisionCheckEnabled();
	void setCollisionCheckEnabled(bool isCollisionCheckEnabled);

	// This function lets you specify a rectangle for collision checks.
	virtual sf::FloatRect getCollisionRect() { return sf::FloatRect(); }

	// Use this to specify what happens when this object collides with another object.
	virtual void handleCollision(GameObject& otherGameObject) {}

	// Put any code here for deal with events such as mouse movement and key presses.
	virtual void handleEvent(sf::Event& event) {}

private:
	// Using a set prevents duplicates.
	std::set<std::string> tags_;

	bool isDead_ = false;
	bool isCollisionCheckEnabled_ = false;
};

typedef std::shared_ptr<GameObject> GameObjectPtr;
```

Boilerplate

At the top of the file we have our directives, #pragma once and #include.

```
#pragma once


#include <set>
#include <string>
#include <memory>
#include <SFML/Graphics.hpp>
```

Tags

Tags are a way of identifying things in your game. Let’s say you create a platformer game, and you want to keep track of which things can hurt the player. You might give them a tag of “enemy”. Or you could keep track of the player with the tag “player”.

`GameObject` has two functions for dealing with tags:

```
void assignTag(std::string tag);
bool hasTag(std::string tag);
```

The `assignTag` function sticks a `string` tag on a `GameObject`, and the `hasTag` function returns a bool that tells you if a tag has been stuck to the object.

Dead Game Objects

Game objects will sometimes need to be removed from whatever scene they are in. The way we do that is with the `isDead` flag.

```
bool isDead();
void makeDead();
```

The `isDead` function will let the scene know that the game object should be removed, and the `makeDead` function will set this flag to `true`. Permanently. Mua ha ha.

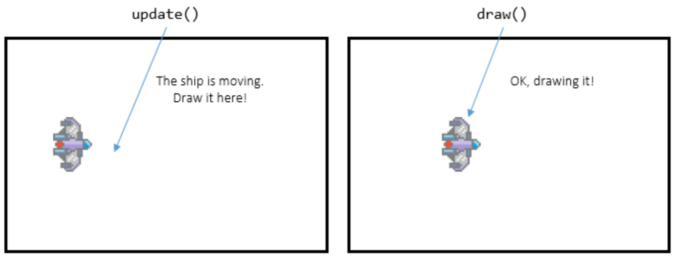
Update and Draw

The `update` and `draw` functions, together, are the two most important functions in any of your game objects. These two virtual functions are called once each frame, one after the other, for every game object in the scene.

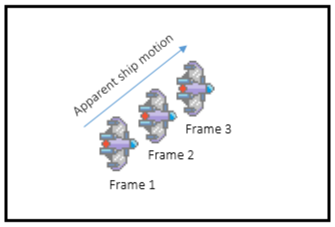
```
virtual void update(sf::Time& elapsed) {}
virtual void draw() {}
```

The `update` function is where you put any code that changes your game object in some way. Examples of this would be moving your game object based on its velocity, changing its size based on whether it's shrinking or growing, choosing which frame of animation to play if it's animated, or seeing if it's run out of health and should be destroyed.

The `draw` function is where you put code to draw the game object to the screen. The `update` function may have changed the game object in some way, and now it's the job of `draw` to put the pixels where they belong.



For a game object to appear on the screen for any given frame, the `draw` function must draw it. You can't just draw the object to the screen once, even if it doesn't change. You have to draw it every single frame. The reason is because the screen is always cleared at the beginning of each frame. This helps to create the illusion of motion. After clearing the screen, all the game objects are drawn in their new positions. This happens so fast that to us it looks like the objects are moving.



Notice that the update function is called with a reference to a `Time` instance. This can be used to determine how much time has passed since the last frame. If we're trying to move game objects on the screen or animate them, time is important because it tells us how far to move the object (perhaps we move 10 pixels per millisecond) or which animation frame to draw (perhaps our animation should run at 12 frames per second). If we didn't take time into consideration, our objects would move or animate at different speeds depending on the speed of the computer. If the computer slowed down for some reason, our game would slow down, too. Or if someone with a really fast computer plays our game, it might run much faster than it should. You can sometimes see this in very old games, where if you run them on a modern, fast computer they look like they're playing in fast-forward. When we take time into consideration, our game will run at the same speed regardless of the speed of the computer.

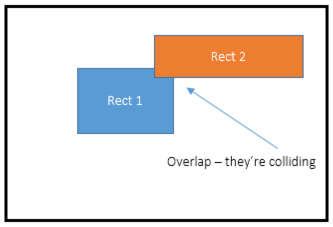
Collisions

The GameObject class has four functions related to collisions:

```
bool isCollisionCheckEnabled ();
void setCollisionCheckEnabled (bool isCollisionCheckEnabled);
virtual sf::FloatRect getCollisionRect() { return sf::FloatRect(); }
virtual void handleCollision(GameObject& otherGameObject) {}
```

Every frame, the current scene will check for collisions between game objects. But it will only check objects that return `true` when their `isCollisionCheckEnabled` function is called. This is because checking for collisions can get expensive if we do it too much. We only want to check objects that really need it. The `setCollisionCheckEnabled` function allows us to say if an object should be checked or not.

The `getCollisionRect` function is something you can override to specify the rectangle around your game object (or inside of it) where collisions should occur. This is specified as a `FloatRect`. When checking to see if two objects collide, `Scene` will call this function on each of the objects to get their collision rectangles, and then see if the rectangles overlap.



When this happens, a collision has occurred. When two game objects collide, each of them has their `handleCollision` function called, and is provided a reference to the other game object they are colliding with. This function is where you can put any special logic or behavior, such as changing direction, losing health points, etc. We'll cover collisions in more detail in Part 6.

Events

The `GameObject` has a function for dealing with events, such as user input:

```
virtual void handleEvent(sf::Event& event) {}
```

If you implement this function in your game objects it will let them respond to any events that occur. As we'll see in Part 2, this will let us do stuff like make our ship move when the player presses the arrow keys. In Part 3, we'll add some code to fire a laser when the space bar is pressed.

GameObjectPtr

Finally, **GameObject.h** uses typedef to define the `GameObjectPtr` type:

```
typedef shared_ptr<GameObject> GameObjectPtr;
```

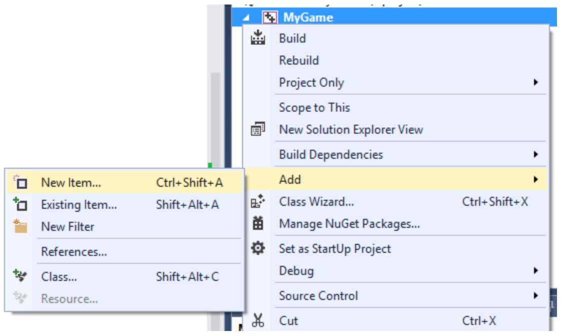
This definition says that `GameObjectPtr` is a shared pointer for `GameObject` instances. This is simply a convenience. It lets us type "`GameObjectPtr`" instead of the much longer "`shared_ptr<GameObject>`" everywhere.

Creating a game object for the ship

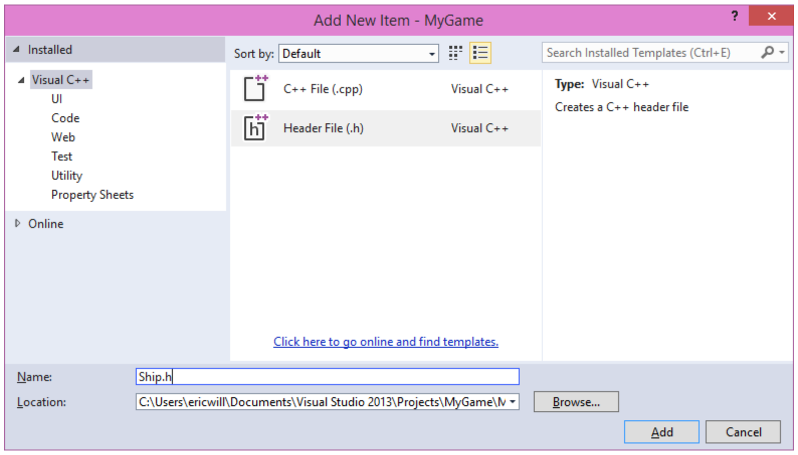
OK, now that we know a little more about the `GameObject` class, it's time to write one of our own.

Create the header file

In your solution explorer, right click on your project and select **Add / New Item**.

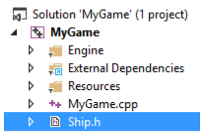


The **Add New Item** dialog should pop up. Select **Header File** and name it **Ship.h**.


Add New Item - MyGame
Installed
Visual C++
UI
Code
Web
Test
Utility
Property Sheets
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
Type: Visual C++
Creates a C++ header file
Click here to go online and find templates.
Browse...
Add
Cancel

Name: Ship.h
Location: C:\Users\ericwill\Documents\Visual Studio 2013\Projects\MyGame\M

You should see the header file in your solution explorer:



Now let's write the code for our **Ship** game object.

Ship.h

```
#pragma once

#include "Engine/GameEngine.h"

class Ship : public GameObject
{
public:
	// Creates our ship.
	Ship();

	// Functions overridden from GameObject:
	void draw();
private:
	sf::Sprite sprite_;
};

typedef std::shared_ptr<Ship> ShipPtr;
```

Notice that the `Ship` class extends `GameObject`. This means that any instance of `Ship` is also an instance of `GameObject`. Of all the functions we saw when we looked at game object (stuff for tags, the update/draw stuff, and stuff for collisions and events), the only one we need to implement for now is the **`draw`** function, so we put it in our class. We also need a constructor. In the private section of the class, we declare an instance variable for our `Sprite`, which we'll need to implement the draw method. Finally, we create a nice `typedef` for `ShipPtr` to save us some typing later.

By the way, it's interesting to note that `sprite_` is an instance of the `Sprite` class, and is created when the `Ship` is created. Yet another way to create an instance!

Finally, if you're wondering why `sprite_` has an underscore at the end, it's just a convention that some programmers like to use to differentiate instance variables from other kinds of variables. Sometimes people put the underscore at the beginning, sometimes they put a lowercase `m` in front of the name, sometimes they do nothing!

Create the implementation file

As you did with the header file, add a new item to the solution. This time, choose C++ File and name it Ship.cpp. Here's the code for the class:

Ship.cpp

```
#include "Ship.h"

Ship::Ship()
{
	sprite_.setTexture(GAME.getTexture("Resources/ship.png"));
	sprite_.setPosition(sf::Vector2f(100, 100));
}

void Ship::draw()
{
	GAME.getRenderWindow().draw(sprite_);
}
```

Our constructor first applies a `Texture` (the image resource file we added earlier) to our `Sprite` via its `setTexture` function. Without a texture, a sprite has no pixels to draw to the screen. To load the texture, we call `GAME`'s `getTexture` function and provide it the file name of the texture we want. Next we put the `Sprite` at x, y position 100, 100 via its `setPosition` function, which taxes a `Vector2f`.

Our `draw` function simply draws the sprite to the window via the `draw` function of `RenderWindow`. We get the `RenderWindow` by asking `GAME` for it via its `getRenderWindow` function. Our `draw` function will do this every single frame. Notice that we didn't need to implement the `update` function in this game object. Our ship, as it's currently implemented, doesn't change from frame to frame. There's nothing to update! We'll take care of that in Part 2, when we add some keyboard controls to move our ship around.

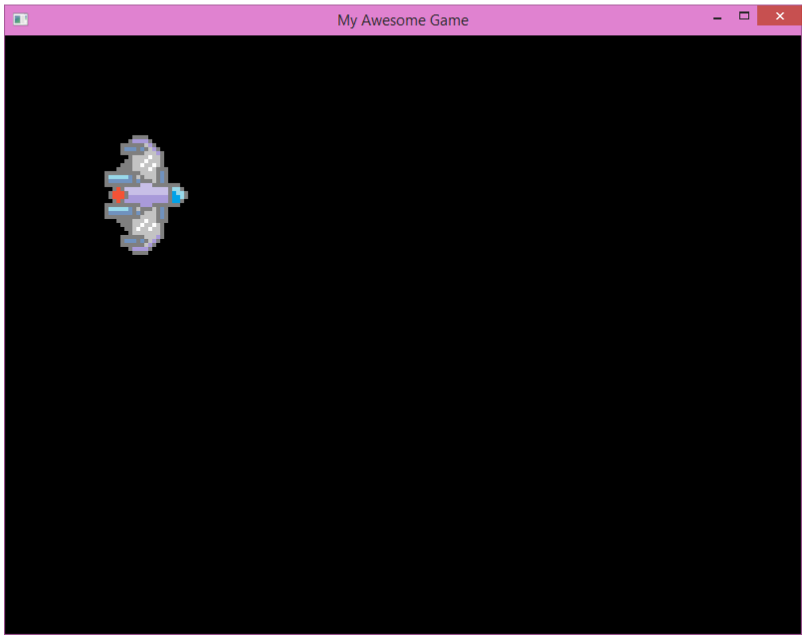
Adding the ship to the scene

OK, the last thing we need to do before we're done with part 1 is get our ship in the scene. To do this, we need to make some changes to GameScene.cpp. (Note that when describing changes to an existing file, we won't always show all the source code. We'll call out the areas you have to change. Remember you can always look at the full source code for each part if you get lost.)

GameScene.cpp

```
#include "GameScene.h"
#include "Ship.h"

GameScene::GameScene()
{
	ShipPtr ship = std::make_shared<Ship>();
	addGameObject(ship);
}
```

Here we have created a `shared_ptr` to a new `Ship` instance, and added it to our scene. Don't forget to `#include "Ship.h"`. Let's run the program and check out the result!



If you don't see the happy little ship floating in space, compare your code with the reference code for Part 1.

Challenge: Customize it

At the end of each part you'll be offered a challenge. It's up to you if you want to take them on. They're a good opportunity to build skills by stepping away from the safety net of the tutorial and making your own mistakes. Trying a challenge and failing is better than not trying at all. That said, some folks won't be interested in them, and that's OK too. Either way, remember that the reference code for each part is available in case you get lost.

So, the challenge for Part 1 is to customize the ship. Either modify the one provided, or make your own. You could make one big ship, or you could even make multiple small ships that the player could control as one. Have fun with it!

Part 2: Making the Sprite Move

To me games have an extremely great and still unrealized potential to influence man. I want to bring joy and excitement to people's lives in my games, while at the same time communicate aspects of this journey of life we are all going through. Games have a larger potential for this than linear movies or any other form of media.

Philip Price

Implement the update function

Now that we have the ship on the screen, making it move is actually very simple. To do this, we're going to implement the `update` function. Recall that `Ship` is a `GameObject`, and game objects have `update` and `draw` functions. Right now we've implemented the `draw` function, but if we want our object to change over time we need to implement the `update` function as well. Let's do it.

Override the update function in Ship.h

Open up Ship.h. In the definition of the `Ship` class, add the `update` function. This tells the compiler you are overriding the virtual method in the base class (`GameObject`).

Ship.h

```
#pragma once

#include "Engine/GameEngine.h"

class Ship : public GameObject
{
public:
	// Creates our ship.
	Ship();

	// Functions overridden from GameObject:
	void draw();
	void update(sf::Time& elapsed);
private:
	sf::Sprite sprite_;
};

typedef std::shared_ptr<Ship> ShipPtr;
```

Implement the update function in Ship.cpp

Open up Ship.cpp and let's add an implementation of the update function.

Ship.cpp

```
#include "Ship.h"

const float SPEED = 0.3f;

// Omitted code...

void Ship::update(sf::Time& elapsed) {
	sf::Vector2f pos = sprite_.getPosition();
	float x = pos.x;
```

```
        float y = pos.y;

        int msElapsed = elapsed.asMilliseconds();

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))    y -= SPEED * msElapsed;
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))  y += SPEED * msElapsed;
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))  x -= SPEED * msElapsed;
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) x += SPEED * msElapsed;

        sprite_.setPosition(sf::Vector2f(x, y));
}
```

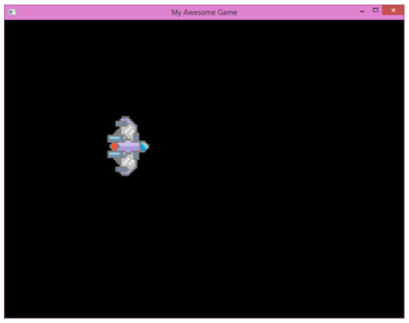
First, this function gets the current position of our sprite and its x and y values. Then it determines how many milliseconds have elapsed since the last frame. Next, it updates the values of x and y depending on which keys are pressed, which it determines via the `Keyboard` class. Finally, it updates the sprite's position.

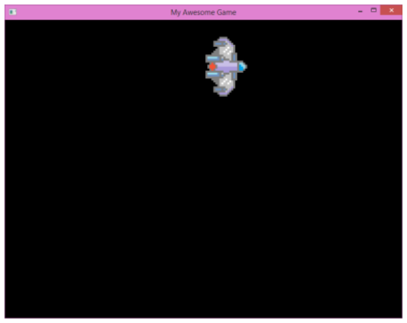
Notice how we're using the elapsed time to set the position of the ship. The number of pixels the ship moves in a single millisecond is defined by a constant called `SPEED`, which we've put right at the top of Ship.cpp.

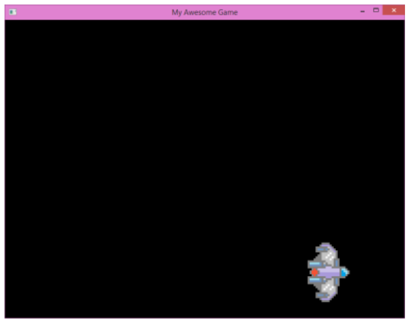
With a speed of 0.3, our ship will move 0.3 pixels per millisecond. If our game is running at 60 frames per second, each frame will last approximately 16.67 milliseconds, and in a single frame the ship will move 5 pixels, or 300 pixels in one second. In our 800 x 600 `RenderWindow`, that means it will take the ship approximately 2.67 seconds to move from one side of the window to the other. But here's the neat thing: Because we are moving based on time, it should *always* take about 2.67 seconds for the ship to move 800 pixels, no matter what the frame rate.

Are we really done with part 2?

Could it truly have been so easy to add keyboard control to our ship? Well... YES. It was. Run the code. Enjoy the fruits of your labor. Fly your ship about the screen, you intrepid space captain you!







Behind the scenes

We're done with the code for part 2, but if you're feeling brave now may be a good time to take a peek under the hood and learn a little bit more about how the game engine works (actually it's not that scary). How is it that your ship's `update` function gets called every frame? What is the relationship between `GAME` and `Scene` and `GameObject`? Feel free to skip this if you're antsy to get through the tutorial.

Challenge: Improved movement

The ship's movement is OK, but, well... it's a little boring. If you feel the same way and are up to the challenge, maybe you can improve it. Perhaps you could add a bit of acceleration and deceleration to the movement, or maybe you could add thrust and momentum for a realistic feeling. Perhaps you can do something about the ship being able to go off the edge of the screen. Maybe you could force it to stop on the edge, or go old skool and simply wrap the ship around to the other side of the screen. What kind of movement do you think would be the most interesting?

Part 3: Pew Pew Pew

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler

Adding a laser

What fun is a space ship without a laser cannon? Let's add one.

Add laser.png to resources

As you did with **ship.png**, add **laser.png** into your project's resources.

Create Laser.h

Let's add a header file for a new game object called `Laser`.

Laser.h

```cpp
#pragma once

#include "Engine/GameEngine.h"

class Laser : public GameObject
{
public:
	// Creates our Laser.
	Laser(sf::Vector2f pos);

	// Functions overridden from GameObject:
	void draw();
	void update(sf::Time& elapsed);
private:
	sf::Sprite sprite_;
};

typedef std::shared_ptr<Laser> LaserPtr;
```

Our `Laser` class is a lot like our `Ship` class. It derives from `GameObject`, and overrides the `draw` and `update` functions. Unlike `Ship`, the constructor for `Laser` takes a `Vector2f` to specify its position on the screen. This is necessary because we want the laser to start from the current position of the ship. We'll cover how that happens in a bit. First, let's add the implementation for our `Laser` class.

Laser.cpp

```cpp
#include "Laser.h"

const float SPEED = 1.2f;

Laser::Laser(sf::Vector2f pos)
{
	sprite_.setTexture(GAME.getTexture("Resources/laser.png"));
	sprite_.setPosition(pos);
	assignTag("laser");
}

void Laser::draw()
{
	GAME.getRenderWindow().draw(sprite_);
}

void Laser::update(sf::Time& elapsed) {
	int msElapsed = elapsed.asMilliseconds();
	sf::Vector2f pos = sprite_.getPosition();

	if (pos.x > GAME.getRenderWindow().getSize().x)
	{
		makeDead();
	}
	else
	{
		sprite_.setPosition(sf::Vector2f(pos.x + SPEED * msElapsed, pos.y));
	}
}
```

The constructor for `Laser` is a lot like the constructor for `Ship`. It loads the texture and sets it on the sprite. But it also sets the sprite's position to the position provided, and assigns the "laser" tag (heh heh). This tag will come in handy later when we want to check for collisions.

The `draw` function is identical to the one in Ship.cpp. It just draws the sprite to the screen.

Our `update` function is different from the one in `Ship`, because this is where the behavior of a laser is defined. What we do here is move the laser along the x axis. We determine how far to move by multiplying the elapsed milliseconds by `SPEED`, which represents how many pixels the laser should move per millisecond.

The new thing we're introducing with the laser is a call to `makeDead`. Recall that this function indicates to the scene that the game object should be removed. Here we're saying this should happen when the laser has moved past the right edge of the screen.

sUpdate Ship.h and Ship.cpp

Next we have to add some logic to the ship code so that the laser is fired. First let's modify Ship.h:

Ship.h

```
#pragma once

#include "Engine/GameEngine.h"

class Ship : public GameObject
{
	// Omitted code...
private:
	sf::Sprite sprite_;
	int fireTimer_ = 0;
};

typedef std::shared_ptr<Ship> ShipPtr;
```

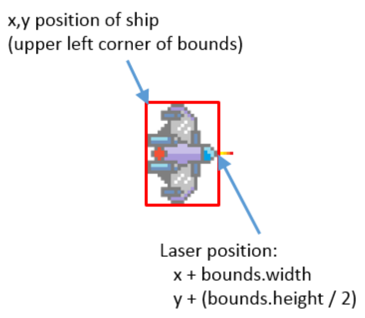
We've added an instance variable to the `Ship` class called `fireTimer`, of type `int`. We'll use this in our implementation to put a little delay between each laser shot.

Ship.cpp

```
#include "Ship.h"

#include <memory>
#include "Laser.h"

const float SPEED = 0.3f;
const int FIRE_DELAY = 200;

// Omitted code...

void Ship::update(sf::Time& elapsed) {

	// Omitted code...

	sprite_.setPosition(sf::Vector2f(x, y));

	if (fireTimer_ > 0)
	{
		fireTimer_ -= msElapsed;
	}

	if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space) && fireTimer_ <= 0)
	{
		fireTimer_ = FIRE_DELAY;

		sf::FloatRect bounds = sprite_.getGlobalBounds();

		float laserX = x + bounds.width;
		float laserY = y + (bounds.height / 2.0f);

		LaserPtr laser = std::make_shared<Laser>(sf::Vector2f(laserX, laserY));
		GAME.getCurrentScene().addGameObject(laser);
	}
}
```

At the bottom of Ship's update function we've added some code to fire the laser. Here you can see how we're using the `fireTimer` instance variable. Each frame we will decrease this value by `msElapsed` until it hits 0 or below. Once this happens, if the space key is being pressed, we can fire the laser.

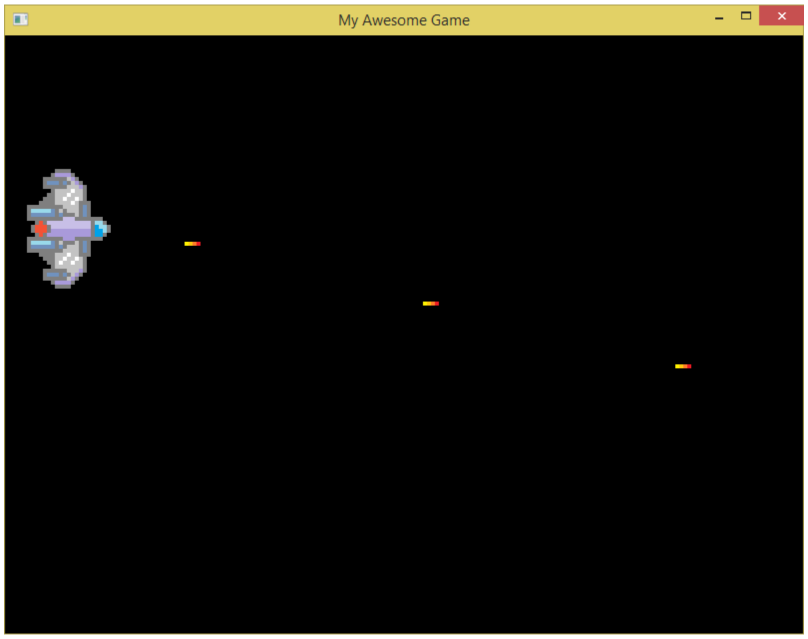
To actually fire the laser, we create a new instance of `Laser` by using `make_shared` and then add it to the scene. In the call to `Laser`'s constructor we pass the `Vector2f` position indicating where it should appear on the screen. We want the laser to show up right in front of the ship, so we compute the position by taking the global bounds of our ship sprite and adding the width (moving the laser to the ship's right edge) and ½ the height (moving the laser half way between the top and bottom of the ship).



Fire at will!

OK, you've made a laser! All we needed to do was add a new game object for the laser, and then modify the ship game object to create the laser game object when certain conditions were met (the right key was pressed, enough time had passed).

Hopefully you are starting to see some nice patterns here. Anything we add to our game can be a game object, and each game object can have its own unique behavior. Working together, the game objects can create something that is more than the sum of their parts. Also, each game object can be relatively simple, because the code for how it behaves is just inside itself. The ship game object doesn't need to know anything about how lasers move, for example. All it needs to do is create them, and off they go!



Challenge: Moar lasers!

One laser is cool, but three lasers would be even cooler. Do you think you can modify the code to shoot one laser from each end of the ship and the middle, instead of just one laser from its center? If you do, you'll have one well-armed ship!

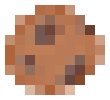
Part 4: Adding a Meteor

People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.

Donald Knuth

Stuff to shoot!

If having a ship without lasers is no fun, having a ship with lasers but without anything to shoot is even less fun! Let's add a meteor to the scene.



Yes, it's a meteor, not a chocolate chip cookie.

Let's have the meteor behave like the laser, except for moving forward until it goes off the right edge of the screen, let's have it move backwards until it goes off the left edge. And let's make it move a bit more slowly. The code for the meteor will be really similar to the code for the laser.

Think you can write the code on your own? Give it a shot. Use **Laser.h** and **Laser.cpp** as a basis for **Meteor.h** and **Meteor.cpp**. If you want to test out your meteor simply edit the constructor in **GameScene.cpp**. Add a couple lines to create a meteor and add it to the scene (don't forget to include **Meteor.h**).

Turn the page to see the code!

Meet the new code, same as the old code, only slightly different

Meteor.h

```
#pragma once

#include "Engine/GameEngine.h"

class Meteor : public GameObject
{
public:
	// Creates our Meteor.
	Meteor(sf::Vector2f pos);

	// Functions overridden from GameObject:
	void draw();
	void update(sf::Time& elapsed);
private:
	sf::Sprite sprite_;
};

typedef std::shared_ptr<Meteor> MeteorPtr;
```

Meteor.cpp

```
#include "Meteor.h"

const float SPEED = 0.25f;

Meteor::Meteor(sf::Vector2f pos)
{
	sprite_.setTexture(GAME.getTexture("Resources/meteor.png"));
	sprite_.setPosition(pos);
	assignTag("meteor");
}

void Meteor::draw()
{
	GAME.getRenderWindow().draw(sprite_);
}

void Meteor::update(sf::Time& elapsed) {
	int msElapsed = elapsed.asMilliseconds();
	sf::Vector2f pos = sprite_.getPosition();

	if (pos.x < sprite_.getGlobalBounds().width * -1)
	{
		makeDead();
	}
	else
	{
		sprite_.setPosition(sf::Vector2f(pos.x - SPEED * msElapsed, pos.y));
	}
}
```

Challenge: More meteor movement

Right now our meteor will just move right to left across the screen. Do you think you could modify the class so that sometimes it moves diagonally, either from top right to bottom left or vis-à-vis?

Part 5: Meteor Shower!

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.
Douglas Hofstadter

Lots of stuff to shoot!

OK, now that we have code to create a meteor, let's add some code to spawn meteors for us to shoot as we go along. (If you added temporary code to GameScene.cpp to test out your meteor, now is a good time to remove it.)

To accomplish the task of continuously hurtling meteors at the spaceship, we're going to do something interesting: We're going to create a `GameObject` that does not draw itself to the screen. Its whole purpose in life will be to spawn meteors at a regular interval.

It may seem odd to have a game object that does not draw itself to the screen, but doing it this way takes advantage of the fact that game objects are able to update themselves once every frame. This is perfect when you want something to happen at regular intervals in time, such as creating meteors. It also demonstrates the flexibility of putting our game code into different game objects, instead of trying to put the code into the core game engine itself.

The Meteor Spawner

OK, here's the code:

MeteorSpawner.h

```
#pragma once

#include "Engine/GameEngine.h"
#include "Meteor.h"

class MeteorSpawner : public GameObject
{
public:
	void update(sf::Time& elapsed);
private:
	int timer_ = 0;
};

Typedef std::shared_ptr<MeteorSpawner> MeteorSpawnerPtr;
```

The spawner only implements the **`update`** function because it's not going to draw itself to the screen. Note that we have a private **`timer_`** value, which we'll use in the implementation to control when we spawn a meteor.

MeteorSpawner.cpp

```
#include "MeteorSpawner.h"

// The number of milliseconds between meteor spawns.
const int SPAWN_DELAY = 1000;

void MeteorSpawner::update(sf::Time& elapsed) {
	// Determine how much time has passed and adjust our timer.
	int msElapsed = elapsed.asMilliseconds();
	timer_ -= msElapsed;
```

```
        // If our timer has elapsed, reset it and spawn a meteor.
        if (timer_ <= 0)
        {
                timer_ = SPAWN_DELAY;

                sf::Vector2u size = GAME.getRenderWindow().getSize();

                // Spawn the meteor off the right side of the screen.
                // We're assuming the meteor isn't more than 100 pixels wide.
                float meteorX = (float)(size.x + 100);

                // Spawn the meteor somewhere along the height of window, randomly.
                float meteorY = (float)(rand() % size.y);

                // Create a meteor and add it to the scene
                MeteorPtr meteor = std::make_shared<Meteor>(sf::Vector2f(meteorX, meteorY));
                GAME.getCurrentScene().addGameObject(meteor);
        }
}
```

Our Meteor Spawner uses a simple timer mechanism to determine when to spawn a meteor. Each frame, we subtract the time that has elapsed since the last frame from our timer. Once the timer value is equal or less than 0, we reset the timer to its default value and spawn a meteor. To spawn one, we simply create a new `Meteor` at a random position off the right side of the screen and add it to the current scene. The meteor will handle everything from there: It will move to the left a little each frame, and remove itself from the frame once it's off the left side of the screen.

Add the Meteor Spawner to our scene

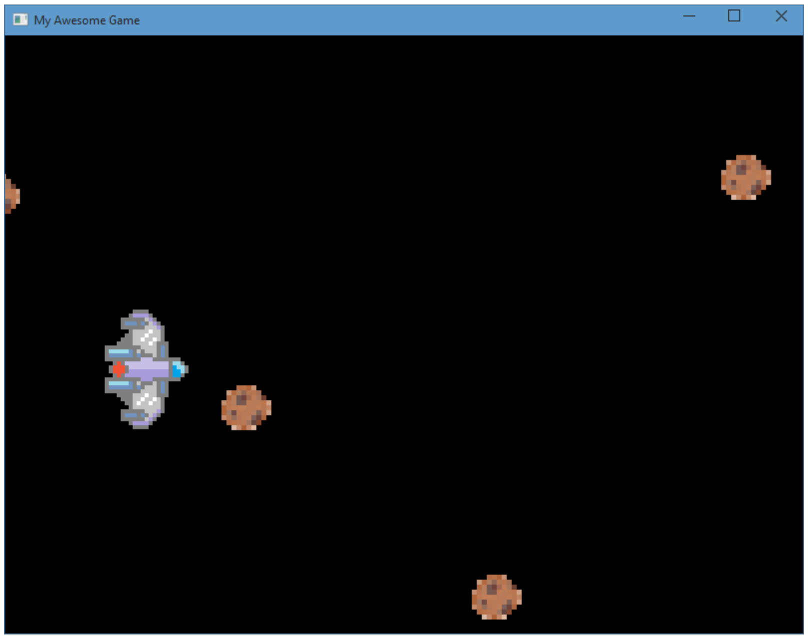
Modify **GameScene.cpp** to add the Meteor Spawner:

GameScene.cpp

```
#include "GameScene.h"
#include "Ship.h"
#include "MeteorSpawner.h"

GameScene::GameScene()
{
        ShipPtr ship = std::make_shared<Ship>();
        addGameObject(ship);

        MeteorSpawnerPtr meteorSpawner = std::make_shared<MeteorSpawner>();
        addGameObject(meteorSpawner);
}
```

Watch out! Cookies ahead!

Try your skill flying through the intense ~~cookie~~ meteor field!



Challenge: Different meteor speeds

It would be neat if the Meteor Spawner could randomly choose a speed for each meteor. Do you think you could modify the `Meteor` constructor to accept a parameter for speed, and modify `MeteorSpawner` to supply it?

Part 6: Shooting the Meteors

Measuring programming progress by lines of code is like measuring aircraft building progress by weight.

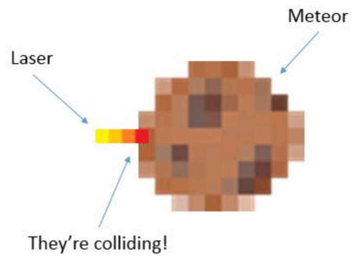
Bill Gates

Time for collisions

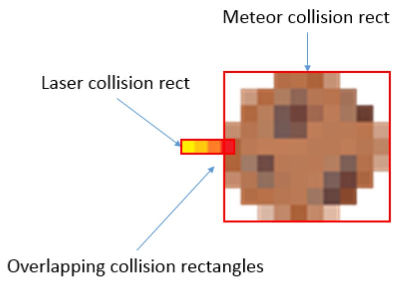
It would be fun to blast these meteors into atoms, don't you think? Having the lasers destroy the meteors is actually really easy. All we need to do is handle the collisions that happen between them, and when a collision occurs, remove both. We'll make a couple small changes to our `Laser` and `Meteor` classes to accomplish this.

How do collisions work?

But first, let's take a moment to talk about collisions. In our game engine, a collision can happen between two `GameObject`s any time they touch.



But how do we know that these two objects are touching? Each GameObject has a "collision rectangle", which you can get by calling its `getCollisionRect` function. When two objects have overlapping collision rectangles, they are colliding. Here's what the game engine sees when it's comparing these rectangles:



The collision rectangle returned from a game object's `getCollisionRect` function is of type `FloatRect`. This type has float values `rectLeft`, `rectTop`, `rectWidth` and `rectHeight`. The left and top values give us a position on the screen for the upper left corner of the rectangle, and the width and height values give us its size (and thereby the locations of the other corners).

If we want collisions between game objects in our game, there are a couple things we need to do:

Make sure game objects that can collide have collision rectangles

We need to make sure both objects return a collision rect that has nonzero area. The default collision rect returned by `GameObject` has 0 for all values, and therefore can't collide with anything. So we'll need to override `getCollisionRect` in any game object we want to participate in collisions. We'll see this below in the code for both `Laser` and `Meteor`.

Tell the engine to check for collisions and do something when they happen

If we want the game engine to check if our game object is colliding with other objects, we must first tell it to do so. The engine won't check if game objects are colliding by default because it would hurt game performance (a lot of unnecessary checking would be done).

To tell the engine to check our game object for collisions, we simply call the `setCollisionCheckEnabled` function (this is in the base `GameObject` class) with a value of `true`. Typically we do this in our game object's constructor:

```
        setCollisionCheckEnabled(true);
```

Once the game object indicates that it should be checked for collisions, the engine will do so each frame. To check, the engine asks whether the collision rectangle of the game object it's checking overlaps with the collision rectangle of any other game object in the scene. (If you'd like to check out the code, look at the `handleCollisions` function in Scene.cpp.)

When the game engine finds an overlap it means a collision has happened. The engine will then call the `handleCollision` method of both game objects. In our code for `Meteor`, below, you'll see how it overrides `handleCollision` to respond to being hit by a laser.

One important thing to note is that for a collision to happen, only one of the two colliding objects needs to have called `setCollisionCheckEnabled(true)`. In the code below, only `Meteor` does this. `Laser` does not. But both `Laser` and `Meteor` can respond to the collision if they want to, because the game engine will call `handleCollision` on both.

OK, enough with the long winded explanations. Let's look at the code!

Modify the Laser class

Since we're going to want to check collisions between lasers and meteors, we need to make sure that our `Laser` class has a collision rectangle with nonzero area. So let's override the base implementation of `getCollisionRect`.

Laser.h

```
#pragma once

#include "Engine/GameEngine.h"

class Laser : public GameObject
{
public:
        // Creates our Laser.
        Laser(sf::Vector2f pos);

        // Functions overridden from GameObject:
        void draw();
        void update(sf::Time& elapsed);
        sf::FloatRect getCollisionRect();
private:
        sf::Sprite sprite_;
};
```

```
typedef  std::shared_ptr<Laser> LaserPtr;
```

Laser.cpp

```
#include "Laser.h"

// Omitted code ...

sf::FloatRect Laser::getCollisionRect()
{
	return sprite_.getGlobalBounds();
}
```

Notice how the collision rect we're returning in `Laser::getCollisionRect` comes from the laser sprite's `getGlobalBounds` function. This function returns a `FloatRect` that specifies the location and size of the sprite on the screen.

Modify the Meteor class

Like `Laser`, `Meteor` needs to specify a collision rectangle. In addition, it needs to implement `handleCollision` and remove itself and any colliding laser from the scene.

Meteor.h

```
#pragma once

#include "Engine/GameEngine.h"

class Meteor : public GameObject
{
public:
	// Creates our Meteor.
	Meteor(sf::Vector2f pos);

	// Functions overridden from GameObject:
	void draw();
	void update(sf::Time& elapsed);
	sf::FloatRect getCollisionRect();
	void handleCollision(GameObject& otherGameObject);
private:
	sf::Sprite sprite_;
};

typedef  std::shared_ptr<Meteor> MeteorPtr;
```

Meteor.cpp

```
#include "Meteor.h"

// Omitted code ...

sf::FloatRect Meteor::getCollisionRect()
{
	return sprite_.getGlobalBounds();
}

void Meteor::handleCollision(GameObject& otherGameObject)
{
	if (otherGameObject.hasTag("laser"))
	{
		otherGameObject.makeDead();
	}

	makeDead();
}
```

Just like `Laser`, `Meteor` uses its sprite's `getGlobalBounds` function for its collision rectangle. In `Meteor`'s `handleCollision` function, we check to see if the game object we're colliding with is a laser (by looking for the tag `"laser"`), and if it is, we remove it from the scene via `makeDead`. Regardless of what the meteor hits, it also removes itself from the scene.

Challenge: Better hitbox

Right now our meteor's collision rectangle uses the global bounds of the meteor sprite. Usually, games don't use the full bounds of a sprite for collision detection. The main reason for this is because usually the things that are colliding aren't perfectly square. The way our game works now, you could shoot the very corner of the meteor sprite and it would vanish. This corner is within the collision rectangle but there aren't any pixels there. Gamers don't generally like it when collisions don't involve actual pixels. How do you think you could fix this?

Part 7: Animation!

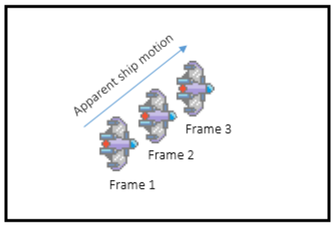
On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage

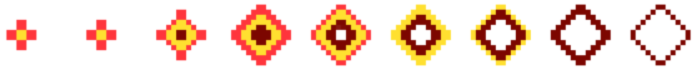
Let's get animated

Now we're able to shoot our meteors and make them disappear, but it would be even cooler if we could have them explode instead of just vanish. For that, we're going to create a new game object called `Explosion`. But before we do, let's talk a little about animation.

Recall how way back in Part 1 we talked about how movement over time creates the illusion of motion:



Well, our explosion is going to use this same technique, except instead of moving over time, the explosion is going to change its appearance over time. For this, we'll require several frames of animation. Each frame will show the explosion in a different state. Here are the frames, from first (on the left) to last (on the right):



The first frame of the explosion is a small + shape, and over each successive frame it grows larger and eventually fades out to nothing.

So, to animate our explosion, we'll need to display each of these frames on the screen, one at a time, allowing each frame to stay on the screen for a moment before it is replaced by the next. Doing this is actually quite easy, thanks to the `AnimatedSprite` class.

The Animated Sprite

All the classes we've created so far (`Ship`, `Laser`, `Meteor` and `MeteorSpawner`) have been `GameObject`s. That is, they are subclasses of GameObject (or you could say they *derive from* `GameObject`).

Here's `Ship`, for example:

```
class Ship : public GameObject
```

For our `Explosion` class, we're going to derive from `AnimatedSprite`. It's a fairly large class, so instead of just reading over the code let's look at it bit by bit. (If you want to look at the source, see AnimatedSprite.h.)

First, note that `AnimatedSprite` is itself a `GameObject`:

```
class AnimatedSprite : public GameObject
```

Everything in our `Scene` is a `GameObject`, and `AnimatedSprite`s are no exception.

Constructor

Here's the constructor for **AnimatedSprite**:

```
AnimatedSprite(sf::Vector2f position, int msPerFrame = DEFAULT_MS_PER_FRAME);
```

Looks pretty similar to what we've seen before. You can set the position of the sprite on the screen. However, we also have an argument called `msPerFrame`, which has a default value of `DEFAULT_MS_PER_FRAME`, defined as follows:

```
const int DEFAULT_MS_PER_FRAME = 20;
```

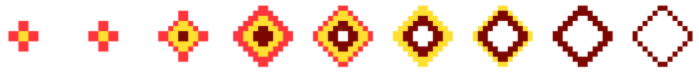
Because the constructor has a default value, you don't have to provide one yourself. If you don't, you'll get 20. But what does this do? Well, this value controls how many milliseconds each frame of animation is displayed on the screen. So, by default, each frame will be on screen for 20 milliseconds. If you provide a value to override the default, you could make it lower (say, 10 milliseconds) or higher (say, 40 milliseconds) and each frame would be on the screen for that long. If you make the value lower, the animation will play faster. If we go from 20 to 10 milliseconds per frame, the animation will be twice as fast because each frame is on screen for only half the time. If we go from 20 to 40 milliseconds per frame, the animation will be twice as slow because each frame is on screen twice as long.

Texture

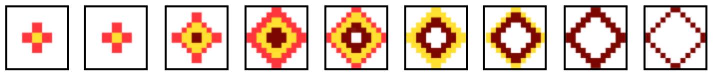
Next we have **setTexture**:

```
void setTexture(sf::Texture &texture);
```

You've used textures before. The **Ship**, **Laser** and **Meteor** all have **Texture**s that they use to display their pixels on the screen. **AnimatedSprite** also requires a texture, but it's a bit different from the textures we used for those other classes. Whereas those classes used "static" (not animated) textures (such as ship.png), for an **AnimatedSprite** we require a *spritesheet* texture. A spritesheet is an image, like any other texture, but it's special because it contains many sprites, which in this case means each frame of our animation. You've actually seen the spritesheet for the explosion already. Here it is again:



This spritesheet contains 9 sprites, one for each frame of the explosion animation. If we take a look at the spritesheet again with some borders drawn around each frame, it becomes more obvious:



We'll look at how you can create a spritesheet from a series of images shortly.

Animations

Once you have a spritesheet, and have called **setTexture** to assign it to your **AnimatedSprite**, you are ready to create animations. To do so, you need to tell the **AnimatedSprite** about the set of frames in the spritesheet that make up your animation, and associate those frames with a name. You'll use the name later when you want to play the animation.

The function we use to create animations is **addAnimation**:

```
void addAnimation(std::string name, std::vector<sf::IntRect> frames);
```

This function takes a name for the animation and a vector of `IntRect`s. Each rectangle specifies where in the spritesheet the pixels for that frame are located (basically, the locations of the black boxes we drew over our spritesheet above).

To play an animation, you use the `playAnimation` function:

```
void playAnimation(std::string name, AnimationMode mode);
```

This function takes the name of the animation and an AnimationMode. Here are the possible modes, enumerated in AnimatedSprite.h:

```
enum AnimationMode
{
  LoopForwards,
  LoopBackwards,
  OnceForwards,
  OnceBackwards,
  FirstFrameOnly
};
```

`LoopForwards` will play the animation frames in the order you specified when you called `addAnimation`, looping back around to the beginning once the end is reached. `LoopBackwards` does the same, only in reverse. `OnceForwards` will stop once the last frame is reached, and `OnceBackwards` will stop once the first frame is reached. `FirstFrameOnly` is useful if you just want to use a single static frame of animation for now, but maybe want to play a different animation later.

The `isPlaying` function can tell you if the `AnimatedSprite` is currently playing an animation or not:

```
bool isPlaying();
```

Position and origin mode

Next we have `setPosition`, which allows you to change the sprite's position on the screen:

```
void setPosition(sf::Vector2f position);
```

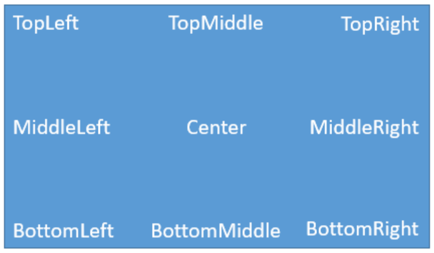
`setPosition` works in concert with another function, `setOriginMode`:

```
void setOriginMode(OriginMode origin);
```

Together, `setPosition` and `setOriginMode` control where your sprite appears on the screen. To understand this, let's first look at the set of possible origin modes, enumerated in AnimatedSprite.h:

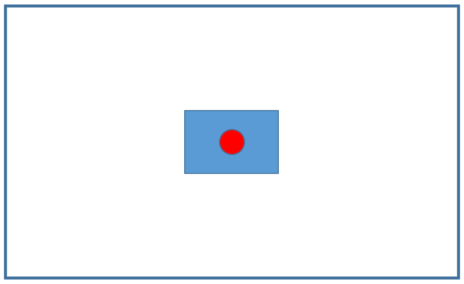
```
enum OriginMode
{
  TopLeft,
  TopMiddle,
  TopRight,
  MiddleLeft,
  Center,
  MiddleRight,
  BottomLeft,
  BottomMiddle,
  BottomRight
};
```

Each value for `OriginMode` refers to a different part of a rectangle:

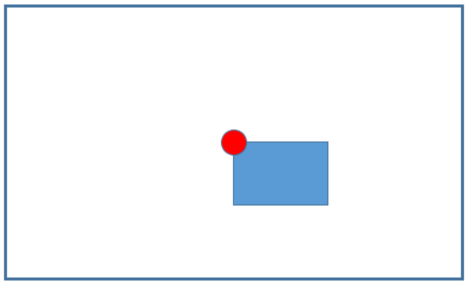


`OriginMode` basically says which part of your sprite its position refers to. In other words, if your x, y position is 100, 100, and your `OriginMode` is `Center`, the pixel in the center of your sprite will be drawn at position 100, 100 on the screen. But if your `OriginMode` is `TopLeft`, it means that the pixel at the top left corner of your sprite will be drawn at position 100, 100.

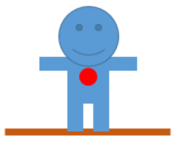
Let's see how that would look visually. Imagine the large rectangle below is the screen, and the red dot in the center represents x, y position 100, 100. We also have a sprite, represented by the blue rectangle. It has an origin mode of `Center`, and a position of 100, 100. Therefore, when we draw it to the screen, it's drawn right in the middle:



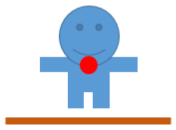
But if we give that same sprite an origin mode of TopLeft, while keeping its position the same at 100, 100, it's now drawn to the right and below center:



In many cases, the default `OriginMode` of `Center` is fine. However, there are some cases where `Center` is not very useful. Consider a character who can stand or crouch. If we used `Center` to position such a character on the screen, and their position (indicated by the red dot) did not move, their feet would come off the ground when crouching:

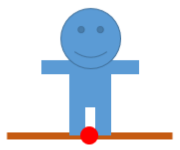


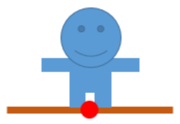
Character when standing (OriginMode Center)



Character when crouching (OriginMode Center)

(By the way, feel free to steal and re-use that amazing character artwork in your own game. You're welcome.) Anyway, how does it look if we use an `OriginMode` of `BottomMiddle` instead?


Character when standing (OriginMode BottomMiddle)


Character when crouching (OriginMode BottomMiddle)

Now when the character crouches his feet stay on the ground. As you can see, specifying the position of a character relative to its feet can be quite useful, particularly if that character can change its height.

Making a spritesheet

As we've seen, we need a special texture called a spritesheet to make an animated sprite. The spritesheet contains many frames of animation. Our explosion animation was originally drawn one frame at a time, with each frame in a different file:

explosion01.png:		explosion04.png:		explosion07.png:	
explosion02.png:		explosion05.png:		explosion08.png:	
explosion03.png:		explosion06.png:		explosion09.png:	

To make a spritesheet, all we have to do is put the images for each frame together into one image file. We'll need to keep track of the location of each frame in the final spritesheet (that is, its x and y coordinates and its width and height). This information will let us specify the `IntRect`s of our animation frames.

There are many tools you could use to make spritesheets. The tool used to create the spritesheet we're using is called Sprite Sheet Maker, available at http://www.spritesheetmaker.org/.

To use Sprite Sheet Maker, simply click **Browse** to add some images, then click **Generate sheet** and **Get final stuff**. You'll see your spritesheet (right click and save it), and some frame data. You'll need this later when using `addAnimation` to specify your frame `IntRect`s:

[{"spritename":"explosion01.png","x":0,"y":0,"height":64,"width":64},
{"spritename":"explosion02.png","x":64,"y":0,"height":64,"width":64},
{"spritename":"explosion03.png","x":128,"y":0,"height":64,"width":64},
{"spritename":"explosion04.png","x":192,"y":0,"height":64,"width":64},
{"spritename":"explosion05.png","x":256,"y":0,"height":64,"width":64},
{"spritename":"explosion06.png","x":320,"y":0,"height":64,"width":64},
{"spritename":"explosion07.png","x":384,"y":0,"height":64,"width":64},
{"spritename":"explosion08.png","x":448,"y":0,"height":64,"width":64},
{"spritename":"explosion09.png","x":512,"y":0,"height":64,"width":64}]

Our Explosion class

OK, we've spent some time looking at `AnimatedSprite` and we've seen how to create a spritesheet. Now let's make our own animated sprite: `Explosion`!

Before trying to run this code, make sure you copy the file **explosion-spritesheet.png** from the same place you got the ship, laser and meteor images and put it in your project's **Resources** folder. Then add it to your project.

Explosion.h

```
#pragma once

#include "Engine/GameEngine.h"

class Explosion : public AnimatedSprite
{
public:
	// Creates our Explosion.
	Explosion(sf::Vector2f pos);

	// Functions overridden from GameObject:
	void update(sf::Time& elapsed);

private:
	void SetUpExplosionAnimation();
};

typedef  std::shared_ptr<Explosion> ExplosionPtr;
```

Pretty straightforward. The interesting stuff is in **Explosion.cpp**:

Explosion.cpp

```
#include "Explosion.h"

Explosion::Explosion(sf::Vector2f pos)
	: AnimatedSprite(pos)
{
	AnimatedSprite::setTexture(GAME.getTexture("Resources/explosion-spritesheet.png"));
	SetUpExplosionAnimation();
	playAnimation("explosion", AnimationMode::OnceForwards);
}

void Explosion::SetUpExplosionAnimation()
{
	std::vector<sf::IntRect> frames;
	frames.push_back(sf::IntRect(  0, 0, 64, 64)); // frame 1
	frames.push_back(sf::IntRect( 64, 0, 64, 64)); // frame 2
	frames.push_back(sf::IntRect(128, 0, 64, 64)); // frame 3
	frames.push_back(sf::IntRect(192, 0, 64, 64)); // frame 4
	frames.push_back(sf::IntRect(256, 0, 64, 64)); // frame 5
	frames.push_back(sf::IntRect(320, 0, 64, 64)); // frame 6
	frames.push_back(sf::IntRect(384, 0, 64, 64)); // frame 7
	frames.push_back(sf::IntRect(448, 0, 64, 64)); // frame 8
	frames.push_back(sf::IntRect(512, 0, 64, 64)); // frame 9

	addAnimation("explosion", frames);
}

void Explosion::update(sf::Time& elapsed)
```

```
{
        AnimatedSprite::update(elapsed);

        if (!isPlaying())
        {
                makeDead();
        }
}
```

So, this class does some neat stuff. Let's take a look. First, in the constructor, we call the private function `SetUpExplosionAnimation` to create an animation named "explosion", and then we play it:

```
        SetUpExplosionAnimation();
        playAnimation("explosion", AnimationMode::OnceForwards);
```

We use `AnimationMode::OnceForwards` because we want the animation to play from the first frame to the last and stop once it reaches the end.

Let's take a look at `SetUpExplosionAnimation`. First it creates a `vector` to store the frame data:

```
        std::vector<sf::IntRect> frames;
```

Then it adds the data, one frame at a time:

```
          frames.push_back(sf::IntRect(  0, 0, 64, 64)); // frame 1
```

Remember that frame data we got from Sprite Sheet Maker? We need that data to specify the x, y, width and height values of our `IntRect` frames. Notice how we pulled the values for frame 1 from the frame data:

{"spritename":"explosion01.png","x":0,"y":0,"height":64,"width":64}

We do this for each frame, from 1 to 9. The order we add the frames is important. If we play the animation forwards, this is the order that will be used.

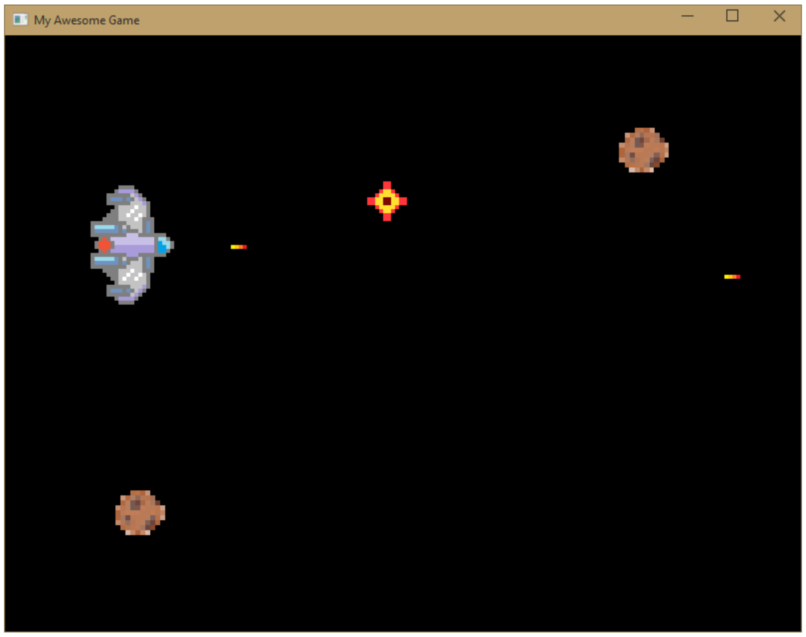
Our update function is interesting, too. We don't want our explosion to be in the scene after the animation is done playing, and update takes care of that:

```
void Explosion::update(sf::Time& elapsed)
{
        AnimatedSprite::update(elapsed);

        if (!isPlaying())
        {
                makeDead();
        }
}
```

First we make sure that the `AnimatedSprite`'s `update` function is called (otherwise our sprite wouldn't animate or be drawn to the screen). We then check to see if we're still playing, and if we're not, we remove ourselves from the scene. `isPlaying` will return `true` while the animation is playing. And since we're using an `AnimationMode` of `OnceForwards`, the animation will stop once it reaches the end. At that point, `isPlaying` will return false.

BOOM!

Now our meteors explode with a satisfying burst of light.



Challenge: Better explosion animation

You've got an animation for your explosions. It's ok. Well, if we're being completely honest with ourselves, it's kinda "meh". You could probably make a much better one. ☺

Part 8: Sound Effects

In programming, as in everything else, to be in error is to be reborn.
Alan J. Perlis

Let's pretend you can hear sounds in the vacuum of space!

OK, so space might be silent, but our game will definitely be more interesting when we add sound. In SFML, to play a sound we need an instance of the `Sound` class and a `SoundBuffer`. The relationship between `Sound` and `SoundBuffer` is kind of like the relationship between `Sprite` and `Texture`. A `Sprite` represents an image on the screen, and a `Texture` provides the data (the pixels) for it. Similarly, a `Sound` represents a waveform that is played by the computer's sound card, and a `SoundBuffer` represents the data (a series of digital "sound samples") to play.

Modifying the Explosion class to play a sound

We're going to modify our `Explosion` class so that in addition to playing an animation, it also plays a sound. The sound we're going to use is called **boom.wav** and you can find it in the same place you found **explosion-spritesheet.png**. Make sure to copy it into your **Resources** folder and add it to your project.

Explosion.h

```
#pragma once

#include "Engine/GameEngine.h"

class Explosion : public AnimatedSprite
{
public:
	// Creates our Explosion.
	Explosion(sf::Vector2f pos);

	// Functions overridden from GameObject:
	void update(sf::Time& elapsed);

private:
	void SetUpExplosionAnimation();

	sf::Sound boom_;
};

typedef std::shared_ptr<Explosion> ExplosionPtr;
```

All we did to **Explosion.h** was add a private variable called `boom_`, of type `Sound`.

Explosion.cpp

```
#include "Explosion.h"

Explosion::Explosion(sf::Vector2f pos)
	: AnimatedSprite(pos)
{
	AnimatedSprite::setTexture(GAME.getTexture("Resources/explosion-spritesheet.png"));
	SetUpExplosionAnimation();
	playAnimation("explosion", AnimationMode::OnceForwards);
```

```
        boom_.setBuffer(GAME.getSoundBuffer("Resources/boom.wav"));
        boom_.play();
}

// Omitted code...
```

When our `Explosion` instance is constructed, we set up our sound and then play it.

First we call the `setBuffer` function on our `Sound`. This function supplies our `Sound` with the data it needs to play. To call `setBuffer`, we need a `SoundBuffer`, and we can get one from `GAME` via `getSoundBuffer`, which is invoked with the name of the sound resource.

Once the `Sound` has its `SoundBuffer`, you can play it at any time with the `play` function, as we do here. Now, when our `Explosion` appears on the screen, not only will it animate but it will also go like BOOSH.

Challenge: Laser sounds

Our meteors explode with a satisfying sound, but the lasers are way too quiet. Can you fix that?

Part 9: Keeping Score

I am rarely happier than when spending entire day programming my computer to perform automatically a task that it would otherwise take me a good ten seconds to do by hand.

Douglas Adams

Let's turn this thing into a real game

We've got most of a game now. We have a ship which the player can control with the keyboard. It shoots lasers. Meteors fly out at the ship and the lasers can destroy them. When a meteor is destroyed it explodes with a burst of light and sound.

The only real thing we're missing from our game is the… well, the game itself. What is the goal? Let's add a simple mechanism to keep score.

Game state

OK, so now we want to keep track of the player's score. Let's say we'll give them 1 point for each meteor they destroy. This score is a piece of *game state*. It's information about the game that can change over time, and will need to be accessed by more than one game object. In our case, we'll modify our `Meteor` class so that each time a meteor is destroyed, the score is updated. We'll also create a new game object called `Score` that will display the current score on the screen.

So the question is, how do we make the game state accessible to both `Meteor` and `Score`? Well, there are many, many ways to do this. If you keep going with game programming and study its patterns and practices, you'll come to realize that managing game state is one of the primary considerations when making games, particularly games that are large in size or scope. It's a fascinating problem and different game engines handle it in different ways.

Thankfully, we already have a place to put our game state. It's also a pretty decent option, all things considered. That place is our `GameScene`. All the game objects that care about score will be used in the `GameScene`, and can therefore use it as a means of sharing information.

Now, this does mean that our `Meteor` and `Score` game objects will assume that the scene they are part of is a `GameScene`. And once they start making this assumption, they can't be used in just any old `Scene`. In our game, that's probably OK. We're not going to use `Meteor` or `Score` in any other scenes. But if you had game state that you wanted to persist across different scenes you would have to find a different solution to this problem.

Adding score to GameScene

OK, so let's modify our GameScene class to track score. First, the header:

GameScene.h

```
#pragma once

#include "Engine/GameEngine.h"

class GameScene : public Scene
{
public:
	// Creates our Scene.
	GameScene();
	// Get the current score
	int getScore();

	// Increase the score
	void increaseScore();

private:
	int score_ = 0;

};

typedef  std::shared_ptr<GameScene> GameScenePtr;
```

Here we have added a private integer variable to track the score, and two functions to get and increase it. You might wonder why we do this instead of just using a simple public variable. The answer is encapsulation which, as you'll see in Part 10, comes in quite handy. OK, let's look at the implementation:

GameScene.cpp

```
#include "GameScene.h"

#include "Ship.h"
#include "MeteorSpawner.h"

// Omitted code...

int GameScene::getScore()
{
	return score_;
}

void GameScene::increaseScore()
{
	++score_;
}
```

Pretty simple. The `getScore` function simply returns `score_`, and `increaseScore` simply increments it. Easy peasy.

Create the Score game object

OK, now we need a way to display the score on screen, and for that we will add a new `GameObject` called `Score`.

Add the font resource

But, before we do that, we need to add a font resource to our project. This will allow us to actually draw text on the screen. The font we're going to use is called **Courneuf-Regular.ttf** (don't ask how to pronounce that). You know the drill: Copy it into your **Resources** folder and add it to your project.

Create the Score class

First, add a new header file to your project, **Score.h**:

Score.h

```
#pragma once
```

```
#include "Engine/GameEngine.h"

class Score : public GameObject
{
public:
	// Creates the Score instance.
	Score(sf::Vector2f pos);

	// Functions overriden from GameObject:
	void draw();
	void update(sf::Time& elapsed);

private:
	sf::Text text_;
};

typedef  std::shared_ptr<Score> ScorePtr;
```

`Score` is, not surprisingly, a `GameObject`. Its constructor takes a `Vector2f` so we can position it where we'd like on the screen. Like most `GameObject`s it has `update` and `draw` functions. What's new here is `Text`. This is a class from SFML that will let us draw text on the screen, much in the way that `Sprite` lets us draw images to the screen. Let's see how it works in the implementation. Add a new source file, Score.cpp:

Score.cpp

```
#include "Score.h"
#include "GameScene.h"
#include <sstream>

Score::Score(sf::Vector2f pos)
{
	text_.setFont(GAME.getFont("Resources/Courneuf-Regular.ttf"));
	text_.setPosition(pos);
	text_.setCharacterSize(24);
	text_.setColor(sf::Color::White);
	assignTag("score");
}

void Score::draw()
{
	GAME.getRenderWindow().draw(text_);
}

void Score::update(sf::Time& elapsed) {
	GameScene& scene = (GameScene&)GAME.getCurrentScene();

	std::stringstream stream;
	stream << "Score: " << scene.getScore();

	text_.setString(stream.str());
}
```

So, there are a couple new things going on here. Let's take a look at the constructor first. Here we are calling the `setFont` method on our `Text` instance. You may notice a pattern here… It's very similar to how we call `setBuffer` on our `Sound` instances and `setTexture` on our `Sprite` instances. In this case, the `Font` is the data that the `Text` instance needs in order to draw text on the screen. And just like those other resources, you can get a Font instance by asking GAME for it, via getFont. Once we've given text_ the Font it needs, we then set its position, size and color.

In our draw function, we put the text on the screen by using the draw function of the RenderWindow, exactly as we would with a Sprite.

Let's check out the update function. First, we're taking the reference to a Scene that is returned by GAME's getCurrentScene function and casting it as a reference to a GameScene:

```
GameScene& scene = (GameScene&)GAME.getCurrentScene();
```

We're able to do this because we know that the scene is actually a GameScene. If it was just a Scene, or some other subclass of Scene, this would fail. But there is a trade-off here. If we try to use our Score class in a scene that is not a GameScene, it will break. That's OK for our game, but it does limit the ways in which we can use Score.

Next, we update the text that should be displayed on the screen:

```
std::stringstream stream;
stream << "Score: " << scene.getScore();
```

The first line creates a stringstream that we can use to combine different bits of text. There are many different ways to combine bits of text, but stringstream is a pretty easy one so we're using it here. It's provided to us by the sstream header, which we included near the top of the file. Once we create our stream, we simply put stuff into it, in this case the literal text "Score: " and the current score as provided to us by the scene. Here you can see why we had to cast the scene to GameScene&. Our GameScene class has a getScore function, but the Scene class does not. If we didn't cast, the compiler would look for a getScore function in Scene, and when it didn't find one compilation would fail.

Finally we update the string that our Text should display on the screen, via setString:

```
text_.setString(stream.str());
```

The setString function requires a string, which is handily returned by the stringstream's str function.

Put Score into the scene

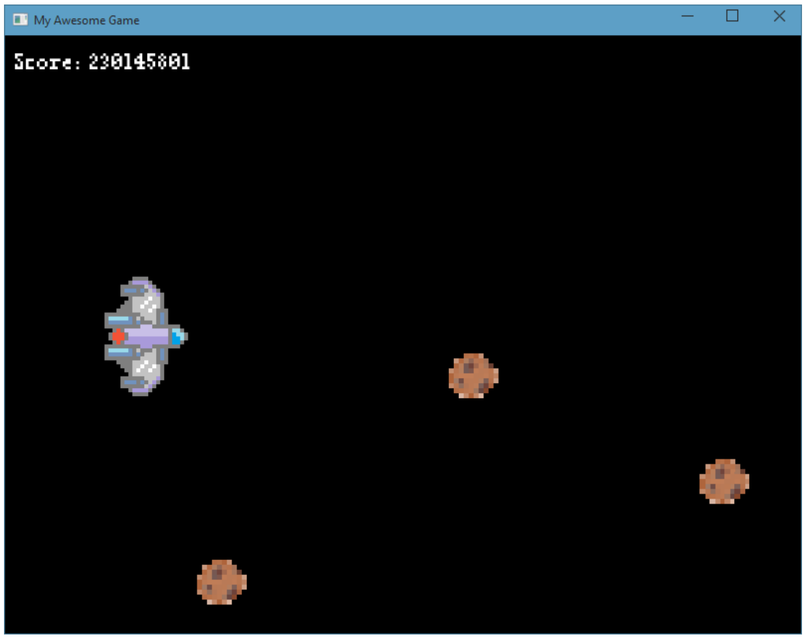
Now that we've got our Score class, we need to add it to GameScene.

GameScene.cpp

```
#include "GameScene.h"

#include "Ship.h"
#include "MeteorSpawner.h"
#include "Score.h"

GameScene::GameScene()
{
	ShipPtr ship = std::make_shared<Ship>();
	addGameObject(ship);

	MeteorSpawnerPtr meteorSpawner = std::make_shared<MeteorSpawner>();
	addGameObject(meteorSpawner);

	ScorePtr score = std::make_shared<Score>(sf::Vector2f(10.0f, 10.0f));
	addGameObject(score);
}

// Code omitted...
```

High score!

Now we can see how good we are, and brag to all our friends! Or just curl up in shame if we're not so great.



```
Score: 230145801
```

Challenge: GOOD LUCK !!!

It might be cool to flash "GOOD LUCK !!!" in big, colorful letters in the middle of the screen when the game starts. How would you do that?

Part 10: Endings

Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter.

Eric S. Raymond

All good things...

All good things must come to an end. That includes this tutorial, and our game. At the moment, it just goes on and on forever. Let's fix that.

GameOverMessage

Let's make a class that will show a "Game Over" message on the screen, along with the user's score. Go ahead and add two new items to your project, **GameOverMessage.h** and **GameOverMessage.cpp**. Here's the source:

GameOverMessage.h

```
#pragma once

#include "Engine/GameEngine.h"

class GameOverMessage : public GameObject
{
public:
	// Creates the GameOverMessage instance.
	GameOverMessage(int score);

	// Functions overriden from GameObject:
	void draw();
	void update(sf::Time& elapsed);

private:
	sf::Text text_;
};

typedef  std::shared_ptr<GameOverMessage> GameOverMessagePtr;
```

GameOverMessage.cpp

```
#include "GameOverMessage.h"
#include "GameScene.h"
#include <sstream>

GameOverMessage::GameOverMessage(int score)
{
	text_.setFont(GAME.getFont("Resources/Courneuf-Regular.ttf"));
	text_.setPosition(sf::Vector2f(50.0f, 50.0f));
	text_.setCharacterSize(48);
	text_.setColor(sf::Color::Red);

	std::stringstream stream;
	stream << "GAME OVER\n\nYOUR SCORE: " << score << "\n\nPRESS ENTER TO CONTINUE";
```

```
        text_.setString(stream.str());
}

void GameOverMessage::draw()
{
        GAME.getRenderWindow().draw(text_);
}

void GameOverMessage::update(sf::Time& time)
{
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Return))
        {
                GameScenePtr scene = std::make_shared<GameScene>();
                GAME.setScene(scene);
                return;
        }
}
```

Most everything here should be familiar by this point. The one new thing we're introducing here is a transition between scenes. In the `update` function of `GameOverMessage`, we call `GAME.setScene` to transition to a new instance of `GameScene`. If you think that must mean that we intend to use this `GameOverMessage` game object in some other scene besides `GameScene`... You're right. ☺

GameOverScene

As you've seen, our game engine lets us put related stuff into scenes. Throughout this tutorial we've been working with a scene called `GameScene`. We've used it to hold our ship, our meteor spawner, and so on. Now we're going to add a new screen to our game, which we'll display when the game is over. It will hold one game object, the `GameOverMessage` we created above. Let's look at the code:

GameOverScene.h

```
#pragma once

#include "Engine/GameEngine.h"

class GameOverScene : public Scene
{
public:
        // Creates our Scene.
        GameOverScene(int score);

};

typedef  std::shared_ptr<GameOverScene> GameOverScenePtr;
```

GameOverScene.cpp

```
#include "GameOverScene.h"
#include "GameOverMessage.h"

GameOverScene::GameOverScene(int score)
{
        GameOverMessagePtr gameOverMessage = std::make_shared<GameOverMessage>(score);
        addGameObject(gameOverMessage);
}
```

Again, nothing really unfamiliar here. OK, so now we have a game object to display a "Game Over" message, and we have a scene which will contain that game object. We also have a way to transition back to `GameScene` when the user presses `Enter`. Let's look at how we will end the game and transition to `GameOverScene`.

Lives

Let's create a simple mechanic for ending the game. Let's give the player some lives, and remove one life each time a meteor gets past them. To do this, we'll create a new piece of game state for the current number of lives, and we'll have our meteor reduce that value each time it removes itself from the scene (because it went off the edge, not because it was destroyed by a laser).

Modify GameScene

Let's add a bit of state for lives to `GameScene`

.GameScene.h

```
#pragma once

#include "Engine/GameEngine.h"

class GameScene : public Scene
{
public:
	// Creates our Scene.
	GameScene();

	// Get the current score
	int getScore();

	// Increase the score
	void increaseScore();

	// Get the number of lives
	int getLives();

	// Decrease the number of lives
	void decreaseLives();

private:
	int score_ = 0;
	int lives_ = 3;
};

typedef  std::shared_ptr<GameScene> GameScenePtr;
```

.GameScene.cpp

```
#include "GameScene.h"

// Omitted code...

int GameScene::getLives()
{
        return lives_;
}

void GameScene::decreaseLives()
{
        --lives_;

        if (lives_ == 0)
        {
                GameOverScenePtr gameOverScene = std::make_shared<GameOverScene>(score_);
                GAME.setScene(gameOverScene);
        }
}
```

Remember back in Part 9 where we talked about encapsulating `score_` with a `getScore` and `increaseScore` function (instead of just having a public variable anyone could modify)? Well, that was *foreshadowing*. Here you can see the real power of encapsulation. When `lives_` is changed via a call to `decreaseLives`, we are able to examine the current value and transition to the `GameOverScene` once it reaches zero. If we didn't use encapsulation for this, it would be the responsibility of classes other than `GameOverScene` to do it, and they might each do it a different way, or not do it properly, or not do it at all. Neither is a good thing.

Modify Meteor

OK, the final thing we have to do is modify Meteor so it decreases the number of lives at the appropriate time.

Meteor.cpp

```
#include "Meteor.h"

// Omitted code...

void Meteor::update(sf::Time& elapsed)
{
        int msElapsed = elapsed.asMilliseconds();
        sf::Vector2f pos = sprite_.getPosition();

        if (pos.x < sprite_.getGlobalBounds().width * -1)
        {
                GameScene& scene = (GameScene&)GAME.getCurrentScene();
                scene.decreaseLives();

                makeDead();
        }
        else
        {
                sprite_.setPosition(sf::Vector2f(pos.x - SPEED * msElapsed, pos.y));
        }
}

// Omitted code...
```

All we're doing here is adding a little extra logic in our `update` function. Now when we go off the screen, in addition to removing ourselves from the scene, we also decrease the number of lives via `decreaseLives`. And of course this means that once again we have to cast the current scene to a `GameScene` reference, because `GameScene` has the `decreaseLives` function and `Scene` does not.

Is that it?!

No! It's just the beginning. This game, and this game engine, belong to you. The game we've made while working on this tutorial is pretty simple. You could evolve it and make it a lot more interesting. You could add animated aliens, different kinds of meteors, weapon power-ups… all kinds of things. The sky's the limit!

Challenge: Keep building!

Reference

Bool

A Boolean value, represented by the type `bool` in C++, can have the values `true` or `false`. Incidentally, this is the kind of value that's returned by operators like `==` (equality), `<` (less than) and `!=` (not equal to), such as you would use in an `if` statement. See more at http://msdn.microsoft.com/en-us/library/tf4dy80a.aspx and http://en.wikipedia.org/wiki/Boolean_data_type.

Casting

Also known as typecasting or type conversion, casting allows you to change one type into another. Sometimes a cast will add or remove data. For example, casting a float to an int is a "narrowing" cast because it removes some data (an int does not have a decimal component), whereas casting an int to a float is a "widening" cast because you're now able to hold additional data in the float that the int could not represent. Casting is done by putting the name of the type you want to cast to in front of the variable, in parenthesis. So to cast a float to an int, you would do this:

```
float myFloat = 10.1f;
int myInt = (int)myFloat; // the .1 will be lost in this narrowing cast
```

See more at http://en.wikipedia.org/wiki/Type_conversion.

Classes

In object-oriented programming, a class defines the "blueprint" for an object, and is used to create "instances" of it. You can create any number of instances from a single class definition. Each class contains functions and data, some of which is private (only accessible by the class itself) and some of which is public (accessible by any other code using the class). For more on classes, see http://danielleleong.com/blog/2014/12/22/whats-an-object.html and http://en.wikipedia.org/wiki/C%2B%2B_classes.

Constants

A constant is like a variable except its value can never change. This is particularly useful for things like the name of our game which will be the same throughout the lifetime of our application. For more on const, see http://en.wikipedia.org/wiki/Const_(computer_programming).

Constructor

A constructor is a special function in a class that initializes, or "constructs", a class instance when it is created. Any time a new instance of a class is created a constructor function is called. Constructor functions can do useful things like set the initial values of instance variables. Classes can have more than one constructor, as long as each one takes different arguments ("has a different signature", as we might say). For more on constructors, see http://msdn.microsoft.com/en-us/library/s16xw1a8.aspx.

Default values

So-called "default parameters" are parameters to a function which have a default value. This means that you don't need to provide a value when you call the function. If you don't, the default value will be used.

```
// function has a default value, someInput not required when calling
void doSomething(int someInput = 10);
```

For more, see http://www.learncpp.com/cpp-tutorial/77-default-parameters/.

Encapsulation

Encapsulation, as we refer it to here, is about information hiding. It means that the data inside a class belongs to a class, and cannot be accessed directly by code outside of that class. For example, a class should never have any public variables that represent its internal state. Those variables should be private, and only changeable via public methods the class exposes.

Encapsulation is a defensive programming technique. It ensures that a class is in complete control of its own data. If a class is not defensive, and lets anyone alter its internal state by exposing its private data publically, then the correct behavior of that class is now dependent on the correct behavior of every other bit of code that uses it, and this can very easily lead to bugs.

Another important benefit of encapsulation is that it allows you to change how the class works internally with out external code being any the wiser. For example, say a class which previously got data from a text file was modified to get the data from a database. If the public functions on the class don't change their signature (the parameters they expect, their names or their return values), then no external code needs to change. Being able to change the way a class works internally without having to change code that uses the class is a huge time saver and reduces the cost of changes.

To learn more about encapsulation, see http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming).

Events

In SFML, an `Event` can represent interesting things happening, like the mouse being moved, a key being pressed, and so on. In our game engine, the function `handleEvent` allows `GameObject` instances to respond to events. When responding to events, it's important to check the `type` of the event before you use it. Here's an example:

```
void MyGameObject::handleEvent(sf::Event& event)
{
	// check for the KeyPressed event type before getting the key code
	if ((event.type == sf::Event::KeyPressed) && (event.key.code == sf::Keyboard::Space))
	{
		// code to do something interesting here
	}
}
```

For more on events, see the SFML website: http://www.sfml-dev.org/tutorials/2.0/window-events.php.

Font

In SFML, a `Font` is data that represents the letterforms in a typeface. This data is required by instances of the `Text` class to display text on the screen using the font.

You can read about `Font` at the SFML website: http://www.sfml-dev.org/tutorials/2.0/graphics-text.php.

Game loop

At the heart of most (if not all) games is the *game loop*. This is a loop that runs forever as long as the game is being played. It's responsible for making sure that everything that needs to happen in order to change game state and render the current frame actually happens. In our game engine that means all the game objects get updated and then drawn to the screen. Update, draw, update, draw, update, draw, over and over. That's the essence of a game loop. For a deep dive into the subject of game loops, see the Game Programming Patterns website: http://gameprogrammingpatterns.com/game-loop.html.

Include directive

This tells the compiler that some other file has code we would like to use. If it's code we've written in our own project, we put the name of the file in double-quotes (`#include "MyLibrary.h"`). Standard libraries don't exist in your project, but the compiler knows where they are, and you can refer to them by putting them in angle brackets (`#include <string>`).

Instance variables

Each instance of a class can have variables that are unique to that instance. These variables are declared inside the class definition and are called instance variables. For example:

```
class MyClass
{
public:
	int publicInstanceVariable;
private:
	int privateInstanceVariable;
};
```

Here, `MyClass` has declared two instance variables, one which is pubic (can be accessed by code outside the class) and one which is private (can only be accessed by the class itself). Generally, instance variables should be private because they represent the state of the instance and should not be changed without the class knowing about it. Often, public functions on the class are used to get and set the values of instance variables, which themselves are private. In object-oriented programming, this is known as "encapsulation" because the state of the object is encapsulated (or hidden) inside of it. Encapsulation leads to fewer bugs in code. Here's an example:

```
class MyClass
{
public:
	int getValue();
	void setValue(int newValue);
private:
	int encapsulatedValue;
};
```

Keyboard, mouse and joystick

SFML contains three lovely classes for dealing with keyboard, mouse and joystick input, which you may be surprised to learn are named `Keyboard`, `Mouse` and `Joystick`, respectively. You can read all about them on the SFML website: http://www.sfml-dev.org/tutorials/2.2/window-inputs.php.

Namespaces

Code in C++ can be organized into something called "namespaces". Folks who write libraries for other people to use put the library code into one or more namespaces. This is a nice thing for them to do because it keeps the names of their classes, functions, variables, etc., separate from your own. If they didn't use namespaces, there could be problems if their library code and your program code used the same name for anything. The standard library uses the namespace `std`, and SFML uses the namespace `sf`.

Pragma once directive

This lovely line of code tells the compiler that we only want to compile this header once, no matter how many source files include it. You may have seen the use of "include guards" that use **`#ifndef`**, **`#define`** and **`#endif`**. This is sort of like that, but less typing.

Random number generator

Computers are not very creative. In fact they're so uncreative that they can't even pick a number between 1 and 10 without being told exactly how to do it. Since computers can't generate real random numbers on their own, they fake it with something called a pseudorandom number generator. No need to worry about how those work just now, but you do need to know that the random number generator must be *seeded*. If you don't seed the generator, or seed it with a number that doesn't change, it will produce the exact same sequence of "random" numbers **every. single. time.** You probably don't want this (usually). That said, predictable random numbers can sometimes be really useful for finding bugs in your code. The "seed" you give the random number generator is up to you. Typically we use `time` as our seed. The `time` function returns the current system time as an integer value (the number of seconds since 1/1/1970). This makes a good seed because each "time" you run the program, `time` will be different, so no two runs will have the exact same sequence of pseudorandom numbers. You could just as easily use 1, or some other number, as your seed. A given seed value will always produce the exact same sequence of pseudorandom numbers.

Rect

SFML provides a type called `Rect` for rectangles. Each `Rect` has left, top, width and height values. `Rect` can be used with any numeric type, but for simplicity SFML provides `IntRect` and `FloatRect` for rectangles with `int` and `float` values, respectively. See the SFML website for more: http://www.sfml-dev.org/documentation/2.0/classsf_1_1Rect.php.

Reference

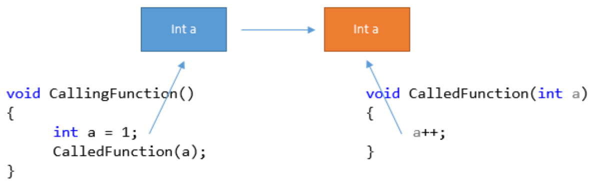
In C++ there are two primary ways that variables are passed to functions when they are called: Pass by value and pass by reference. When a variable is passed by value, a new copy of the value is made in memory. If your function changes the value of a variable which is passed by value, the calling function is none the wiser. Here's an example:

```
void CallingFunction()
{
	int a = 1;
	CalledFunction(a);
	printf("CallingFunction: The value of a is %i\n", a);
}

void CalledFunction(int a)
{
	a++;
	printf("CalledFunction: The value of a is %i\n", a);
}
```

Here's the output from this program:

```
CalledFunction: The value of a is 2
CallingFunction: The value of a is 1
```

The reason `CallingFunction` thinks `a` has a value of 1 even though `CalledFunction` thinks it's 2 is because the `a` inside `CallingFunction` and the `a` inside `CalledFunction` are two completely different variables. They are able to have the same name because their scope is limited to the functions they live in. No one outside the function can see the variable. When `CallingFunction` calls `CalledFunction`, the value of `a` inside of `CallingFunction` is copied into the `a` variable inside of `CalledFunction`. When `CalledFunction` makes changes to `a`, it's modifying its own copy of the variable. Hence, call by value.



```
void CallingFunction()
{
    int a = 1;
    CalledFunction(a);
}
```

```
void CalledFunction(int a)
{
    a++;
}
```

Each function has its own unique variable called a. When one calls the other, the value is copied. This is pass by value.

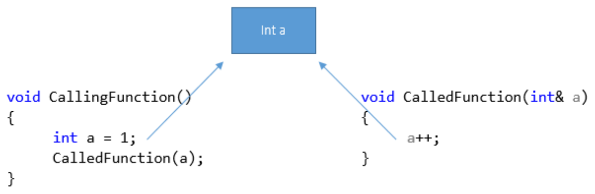
With pass by reference, the value of the variable is not copied. Instead, a reference to the place in memory where the variable lives is passed. This means that if the called function changes the value of the variable, the calling function will see this change. Both functions are effectively using the same variable. Here's an example:

```
    void CallingFunction()
    {
            int a = 1;
            CalledFunction(a);
            printf("CallingFunction: The value of a is %i\n", a);
    }

    void CalledFunction(int& a)
    {
            a++;
            printf("CalledFunction: The value of a is %i\n", a);
    }
```

This code is nearly identical to the code above, with one tiny but very important difference: Look at the signature for `CalledFunction`. Before, it had an argument of type `int`. Now it's of type `int&`, which we read as "reference to int". Now when `CallingFunction` calls into `CalledFunction`, the value of `a` will not be copied from one unique variable to another. Instead, both functions will have access to the same value. Here's the output:

```
    CalledFunction: The value of a is 2
    CallingFunction: The value of a is 2
```

Or, visually:



```
void CallingFunction()
{
    int a = 1;
    CalledFunction(a);
}
```

```
void CalledFunction(int& a)
{
    a++;
}
```

Using call by reference, both functions manipulate the same value in memory.

References are useful because they allow us to pass complicated or large objects, or objects we only want one instance of, without copying them.

RenderWindow

A class in SFML that provides a window for 2D drawing. See the SFML documentation for more: http://www.sfml-dev.org/documentation/2.2/classsf_1_1RenderWindow.php

SFML

The **S**imple and **F**ast **M**ultimedia **L**ibrary (SFML) is an open-source library for C++ (with wrappers for other languages) that provides a clean interface over sound, graphics, input and more. It's cross platform, so you don't have to worry about the particulars of Windows or OS X or Linux. It's the library that our game engine is written on top of. You can find more about SFML and read the excellent documentation at its website, http://www.sfml-dev.org/.

Smart pointers

In C++, a pointer is a variable that "points" to a place in memory where a value is held. Pointers are useful because they allow different parts of the code to read or manipulate the same values. This is handy when those values are large or time-consuming to compute. For example, you wouldn't want to have to copy all the game objects in your scene each time you draw a new frame. It's much easier to use pointers for this.

Pointers, on their own, are not dangerous. In fact, they're awesome. They let you point to instances of classes (or anything else), so you can just pass around pointers instead of copying things needlessly. The thing that's dangerous is managing memory. And to understand why, we need to talk about object lifetimes and how you create instances.

One way to create an instance is like this:

```
MyClass instance;
```

If you had declared `instance` inside of a function, the memory it needed would have been allocated, and it would have been created, with this line of code. When the function was finished, `instance` would be destroyed automatically, and the memory deallocated for you. This is great if you want your instance to have the same lifetime as a function.

This also works when you want your instance to live as long as some class it's a part of. You'll see this in the game engine a lot. For example, in GameObject.h, the `private` section has this instance variable:

```
std::set<std::string> tags_;
```

`tags_` is created when the `GameObject` is created, and memory is allocated for it. `tags_` is destroyed when the `GameObject` is destroyed, and memory is deallocated. It happens automatically and always works.

But sometimes we can't use this means of creating instances. Sometimes we need the instance to stay around after the function that creates it has finished, or after an object that used it is destroyed. In these cases, we need to control the lifetime of the instance ourselves. The tools C++ used to provide for this were the `new` and `delete` operators. The `new` operator let you create a new instance of a class (allocating its memory), and that instance would exist until you used `delete` to get rid of it (deallocating its memory).

The problem with `new` and `delete` is that using them means you're managing memory yourself, and this is *very hard* to get right. If you don't always `delete` what you `new` you will have memory leaks. If you try to use an instance after you delete it, random bad things will happen. These kinds of bugs cause programs (and computers) to crash. The difficulty of managing memory yourself is such an important problem to computer science that it's one of the driving forces behind modern languages like Java and C# where it is *completely forbidden*.

So, how to harness the power of pointers while reducing the dangers? The answer is smart pointers. These are a feature of modern C++ that manages the lifetime of the pointer, and the memory it's using, for you.

There are different kinds of smart pointers in C++: `shaerd_ptr`, `unique_ptr` and `weak_ptr`. These are provided by the standard library and can be brought into your program by using `#include <memory>`.

In the game engine, you'll notice we use `shared_ptr` quite a bit. This kind of smart pointer keeps track of how many owners it has. As long as someone is using the shared pointer, the thing it's pointing to will continue to exist. When no one is using it, the value will be deleted.

Going into depth about how to properly use smart pointers is outside the scope of this tutorial. But if you follow the patterns for creating new instances that we use in the tutorial code you should be just fine.

If you want to dig deeper, there are good resources if you search for them online, such as this video from CppCon 2014 where Herb Sutter talks about the essentials of modern C++ style: https://www.youtube.com/watch?v=xnqTKD8uD64.

Sound

In SFML, `Sound` is a class that lets your play audio via the computer's sound card. It requires a `SoundBuffer` to supply it with the data to play. Learn more at http://www.sfml-dev.org/tutorials/2.0/audio-sounds.php.

SoundBuffer

In SFML, `SoundBuffer` is a class that represents the waveform of a sound. It is the data that a `Sound` uses to play sound via the sound card. Learn more at http://www.sfml-dev.org/tutorials/2.0/audio-sounds.php.

Sprite

`Sprite` is a class in SFML that allows us to draw images onto the screen. A `Sprite` has a `Texture` (the source image from which it draws its pixels), an `IntRect` that specifies which pixels in the texture to draw, and a `Vector2f` position on the screen where it draws those pixels. There's a great explanation on the SFML website: http://www.sfml-dev.org/tutorials/2.0/graphics-sprite.php.

Standard Library

The C++ standard library is a set of functions, classes, and other useful stuff that comes with every C++ compiler. It's there so that C++ works the same just about everywhere, and so you don't have to reinvent the wheel. The stuff inside the standard library has been refined and tweaked for decades and is the way to go if you need something like a string (std::string), a list of things (std::list) or an array of things that can resize itself (std::vector). There's TONS more in there. Check it out at http://en.wikipedia.org/wiki/C%2B%2B_Standard_Library.

String

A string is a sequence of characters. There are different ways to represent strings. One ways is through arrays of characters. The `string` class in the C++ standard library is another way of representing and working with strings. Instances of this class manage an array of characters for you, and provide some additional conveniences that character arrays on their own don't provide. They're also easy to use with other types in the standard library, such as `set` and `vector`, so we use them in the game engine. You can learn more at http://en.wikipedia.org/wiki/String_(C%2B%2B).

Text

In SFML, `Text` is a class that you can use to draw text on the screen. It requires a `Font` to know how to display the text. See more at the SFML website: http://www.sfml-dev.org/tutorials/2.0/graphics-text.php

Texture

In SFML, a `Texture` represents the pixels of an image. For a `Sprite` to draw itself on the screen, it needs a texture to draw pixels from. Sometimes the entire texture is used to draw the sprite, but the real power of textures becomes apparent when you use just part of it at a time, as you would with animated sprites. By putting multiple frames of animation into one texture, you can animate your sprite simply by moving its `textureRect` from one frame to the next.

Multiple frames of animation in one texture

For more on the `Texture` class, see the SFML website: http://www.sfml-dev.org/tutorials/2.0/graphics-sprite.php.

Time

SFML's `Time` class provides access to a time value which you can ask for as microseconds, milliseconds or seconds via its `asMicroseconds`, `asMilliseconds` and `asSeconds` functions, respectively. To learn more about how time is handled in SFML, see http://www.sfml-dev.org/tutorials/2.0/system-time.php.

Typedef

The `typedef` keyword in C++ allows you to create new names for existing types. This is useful for making short names for long types, as in the following example:

```
typedef shared_ptr<GameObject> GameObjectPtr;
```

Now, instead of having to type `shared_ptr<GameObject>` everywhere, you can just use `GameObjectPtr`.

Something else typedef can do is give different names for the *same* thing. For example:

```
typedef int width;
typedef int height;
```

Now you can declare variables as having the type `width` or `height`, and you can create functions that accept `width` and `height` values as input. Even though they are both integers, you could never create a `width` variable and pass it to a function that requires `height`. The compiler would prevent you from making this mistake. But if you just used `int`, nothing would stop you. Even though this is a nifty feature of `typedef`, we don't really use it in the game engine.

You can read more about `typedef` on Wikipedia: http://en.wikipedia.org/wiki/Typedef.

Vector

The C++ standard library provides a number "sequence containers". Like arrays, these containers can hold lots of data. Unlike arrays, they have additional functionality and flexibility. The `vector` type allows random access, just like arrays (meaning you can get an element in the `vector` by its index), but unlike arrays you don't have to know how many elements you want it to hold ahead of time. It will dynamically allocate memory for you as you need it. See more at http://en.wikipedia.org/wiki/Sequence_container_(C%2B%2B).

Note: Don't confuse "`vector`" the container with "`Vector2`" the 2-dimensional vector, described below.

Vector2

SFML provides a `Vector2` class for describing 2-dimensional vectors. These are useful for specifying things like position in 2-dimensional space, or velocity along two different axes. You can create a `Vector2` from many numerical types, but there are built-in convenience types for float (`Vector2f`), int (`Vector2i`) and unsigned int (`Vector2u`).

For more, see http://www.sfml-dev.org/documentation/2.0/classsf_1_1Vector2.php.

Note: Don't confuse "Vector2" the 2-dimensional vector with "vector" the container, described above.

Virtual functions

One of the awesome things object-oriented programming provides you is *polymorphism*. This needlessly intimidating word simply means that different objects with the same interface can have different behavior. But what does *that* mean? For us, in our game engine, it means that different game objects (the player, enemies, backgrounds, parts of the environment, or whatever you have in your game), each of which have the same *interface* (the update and draw functions, for example), can have totally different code behind that interface. The way a bouncing ball updates itself is different than the way a walking robot updates itself. But to the game engine, which only cares about the common interface shared by all game objects, the ball and the robot are the same. It asks them both to update themselves, and then it asks them both to draw themselves.

In C++, one of the ways we achieve polymorphism is through the use of virtual functions. The `virtual` keyword in front of a function signature says "subclasses can provide a new implementation for this function".

In **GameObject.h** we have the following function signature:

```
virtual void update(sf::Time& elapsed) {}
```

Notice the braces at the end (`{}`). Believe it or not, this is the function body. Basically, this says that the update function *does nothing*. You can create a subclass of **GameObject** and not override the `update` function, in which case this do-nothing version would be called. However, if you do override the function in your subclass, *your* version of the function would be called. How do you override? First, you have to specify in your subclass that you are going to override:

```
class MyGameObject : public GameObject
{
public:
	void update(sf::Time& elapsed);
};
```

This says to the compiler, "Hey, `MyGameObject` is going to provide some new code for the `update` method. Call that instead of the one in the base `GameObject` class!", and the compiler will happily abide. Notice how this time the function definition does not end in empty braces. It ends with a semicolon. This means that the function has *no body* defined in the header file. When the compiler sees this, it will look for the body in the **.cpp** file:

```
void MyGameObject::update(sf::Time& elapsed)
{
	// provide my own update code here!
}
```

In this way, every subclass of `GameObject` can have its own code, and the classes that call `GameObject` functions (like `Scene`), don't need to know about those details.

Learn more about polymorphism at http://en.wikipedia.org/wiki/Polymorphism_(computer_science).

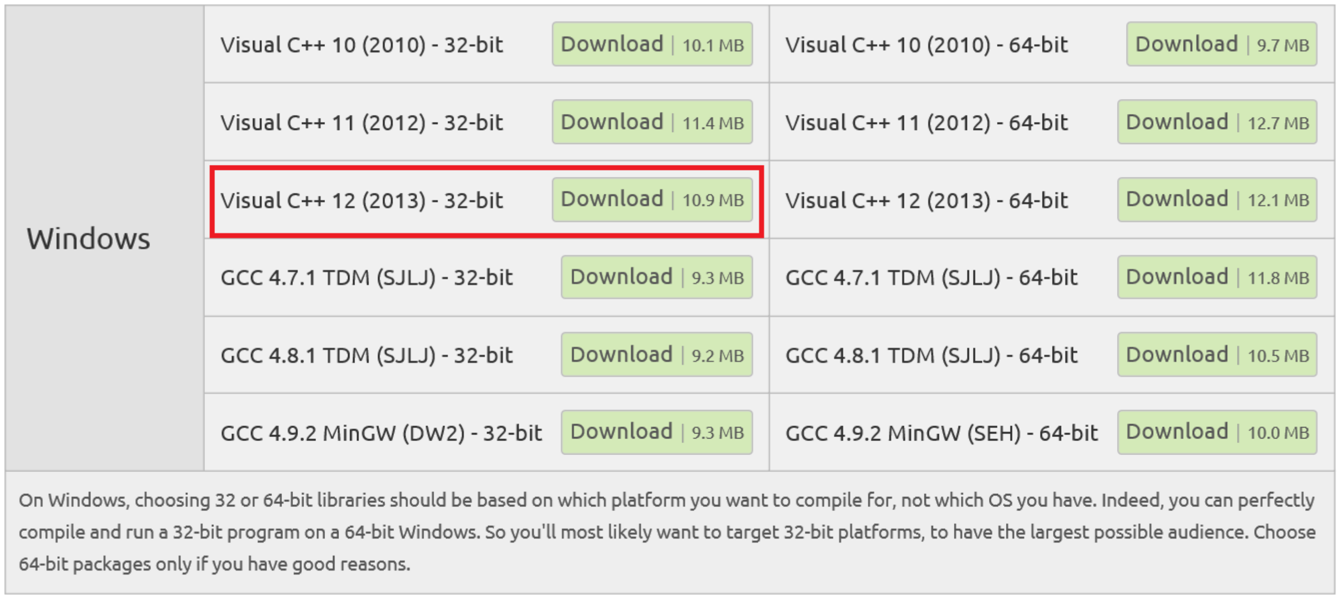
Appendix: Configuring an SFML Project

When to use this info

If you want to create a new project, or work with your code at home, you'll need to set it up to use SFML. There is complete documentation on how to do this for various types of projects and operating systems at http://www.sfml-dev.org/tutorials/. In this tutorial, we'll only cover how to set up a basic project in Visual Studio.

Downloading SFML

The first thing you'll need is a copy of SFML itself. You can download it from http://www.sfml-dev.org/download.php. Typically you will want the latest stable version. You'll need to select the correct version for your operating system and IDE. If you're using any flavor of Visual Studio 2013, this is probably the one you'll want:



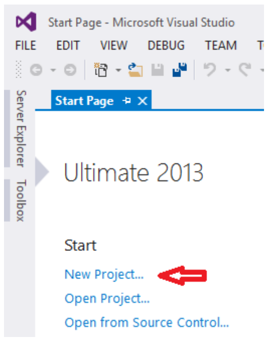
Windows	Visual C++ 10 (2010) - 32-bit	Download | 10.1 MB	Visual C++ 10 (2010) - 64-bit	Download | 9.7 MB
	Visual C++ 11 (2012) - 32-bit	Download | 11.4 MB	Visual C++ 11 (2012) - 64-bit	Download | 12.7 MB
	Visual C++ 12 (2013) - 32-bit	Download | 10.9 MB	Visual C++ 12 (2013) - 64-bit	Download | 12.1 MB
	GCC 4.7.1 TDM (SJLJ) - 32-bit	Download | 9.3 MB	GCC 4.7.1 TDM (SJLJ) - 64-bit	Download | 11.8 MB
	GCC 4.8.1 TDM (SJLJ) - 32-bit	Download | 9.2 MB	GCC 4.8.1 TDM (SJLJ) - 64-bit	Download | 10.5 MB
	GCC 4.9.2 MinGW (DW2) - 32-bit	Download | 9.3 MB	GCC 4.9.2 MinGW (SEH) - 64-bit	Download | 10.0 MB

On Windows, choosing 32 or 64-bit libraries should be based on which platform you want to compile for, not which OS you have. Indeed, you can perfectly compile and run a 32-bit program on a 64-bit Windows. So you'll most likely want to target 32-bit platforms, to have the largest possible audience. Choose 64-bit packages only if you have good reasons.

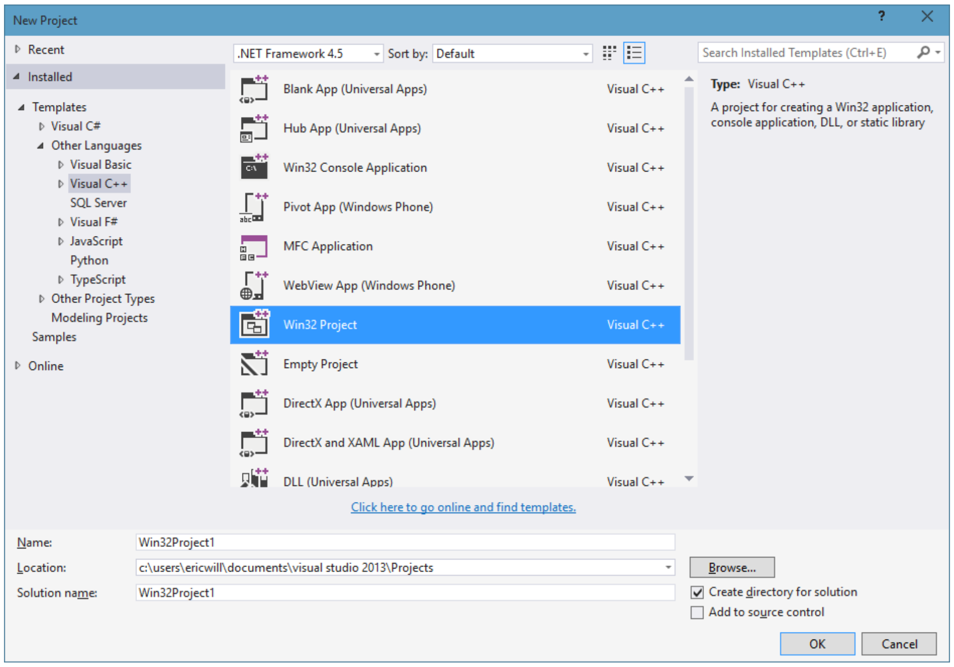
Unzip the file you download and put the SFML folder someplace you'd like it to stay, such as C:\SFML. We'll refer to this as the "SFML folder".

Creating the Visual Studio project

Open Visual Studio, and select **New Project**:



Create a Win32 Project:


New Project
Recent
Installed
Templates
Visual C#
Other Languages
Visual Basic
Visual C++
SQL Server
Visual F#
JavaScript
Python
TypeScript
Other Project Types
Modeling Projects
Samples
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank App (Universal Apps) Visual C++
Hub App (Universal Apps) Visual C++
Win32 Console Application Visual C++
Pivot App (Windows Phone) Visual C++
MFC Application Visual C++
WebView App (Windows Phone) Visual C++
Win32 Project Visual C++
Empty Project Visual C++
DirectX App (Universal Apps) Visual C++
DirectX and XAML App (Universal Apps) Visual C++
DLL (Universal Apps) Visual C++
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C++

A project for creating a Win32 application, console application, DLL, or static library

Name: Win32Project1

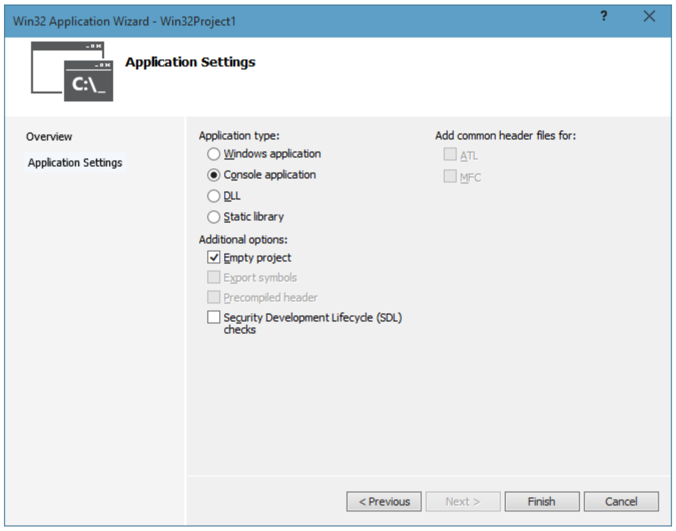
Location: c:\users\ericwill\documents\visual studio 2013\Projects

Solution name: Win32Project1

☑ Create directory for solution

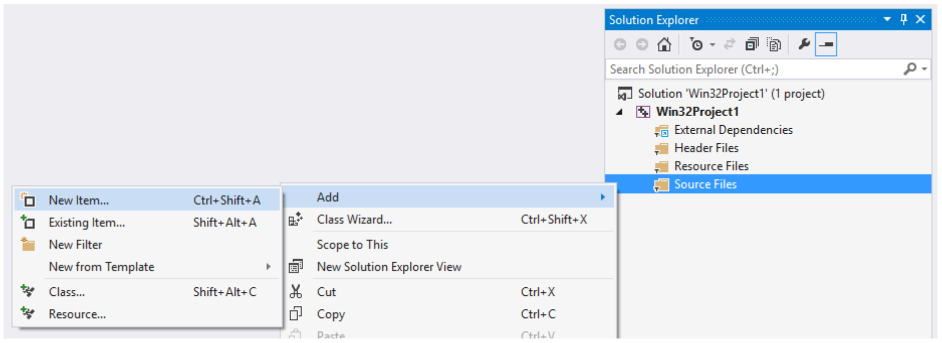
☐ Add to source control

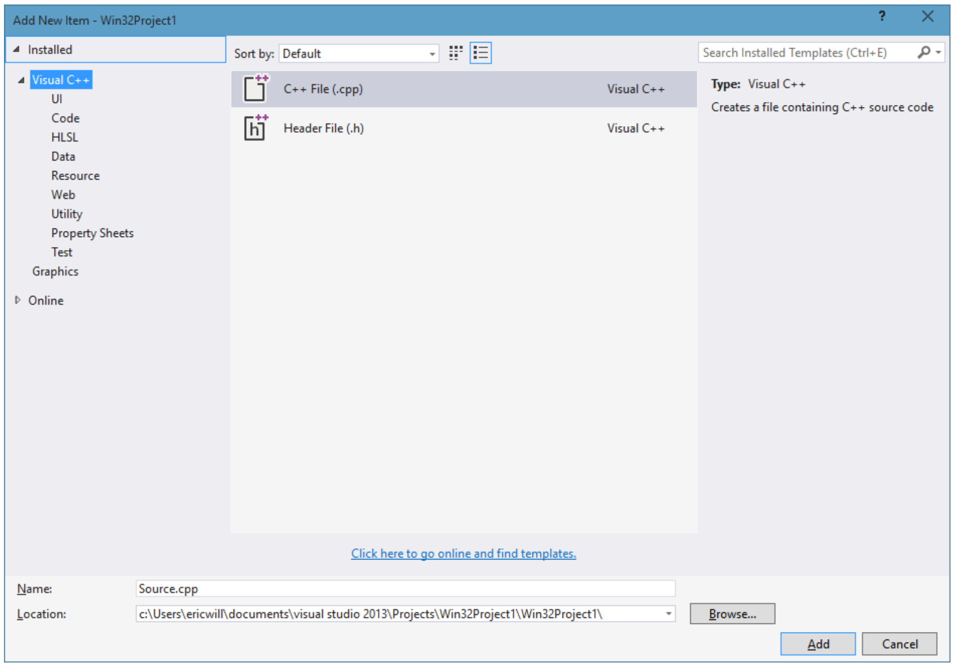
Create an empty console application, no precompiled header, no SDL checks:



Application type:
( ) Windows application
(●) Console application
( ) DLL
( ) Static library
Additional options:
[✓] Empty project
[ ] Export symbols
[ ] Precompiled header
[ ] Security Development Lifecycle (SDL) checks
Add common header files for:
[ ] ATL
[ ] MFC

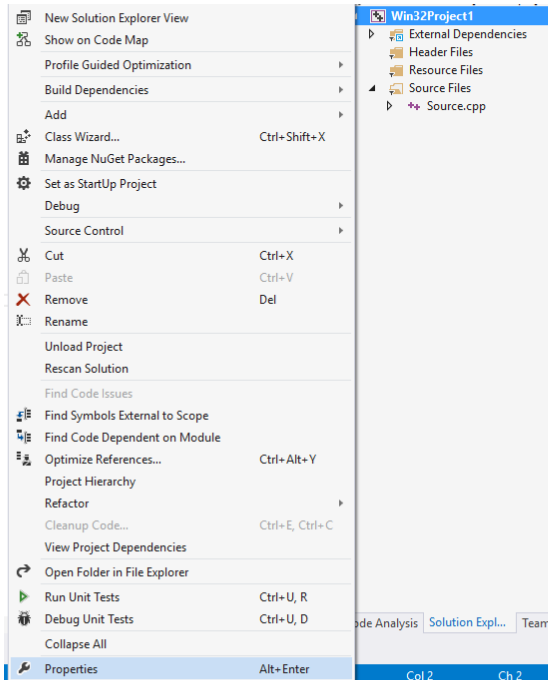
Add an empty C++ source file to the project:




Add New Item - Win32Project1
Installed
Visual C++
UI
Code
HLSL
Data
Resource
Web
Utility
Property Sheets
Test
Graphics
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
Type: Visual C++
Creates a file containing C++ source code
Click here to go online and find templates.
Name: Source.cpp
Location: c:\Users\ericwill\documents\visual studio 2013\Projects\Win32Project1\Win32Project1\

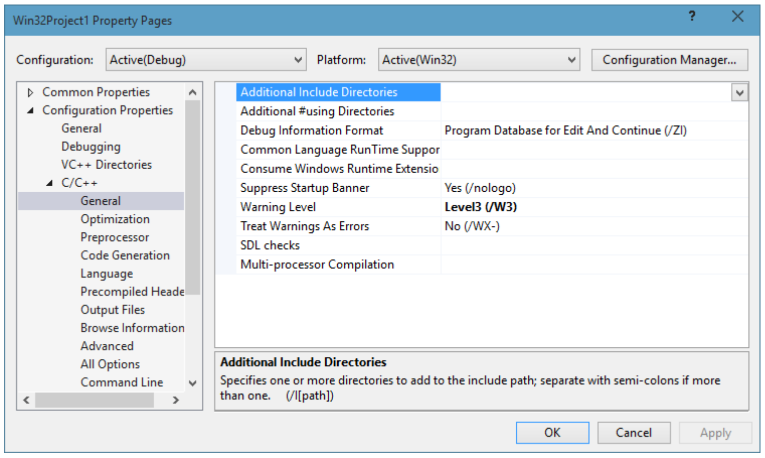
Edit the project properties

Select the project in the Solution Explorer and hit ALT+ENTER or right click and select Properties:



C/C++ properties

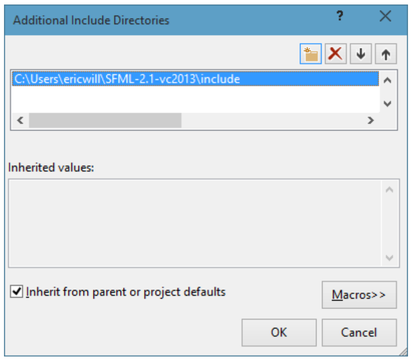
Go to the **C/C++** properties and select **Additional Include Directories**. Click on the little down arrow ( ) and select **Edit**:



Additional Include Directories	
Additional #using Directories	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Common Language RunTime Suppor	
Consume Windows Runtime Extensio	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
SDL checks	
Multi-processor Compilation	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Click on the happy little folder icon () and put in the SFML folder followed by **\include** (e.g. C:\SFML\include):

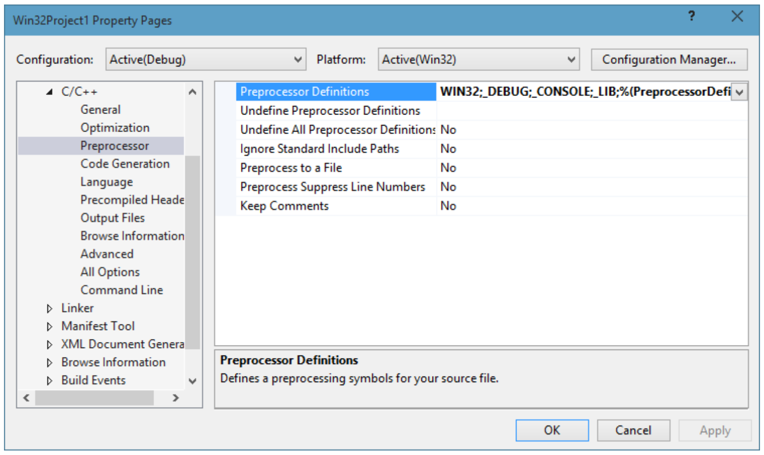


C:\Users\ericwill\SFML-2.1-vc2013\include

Inherited values:

☑ Inherit from parent or project defaults

Next select **Preprocessor** and then **Preprocessor Definitions**:

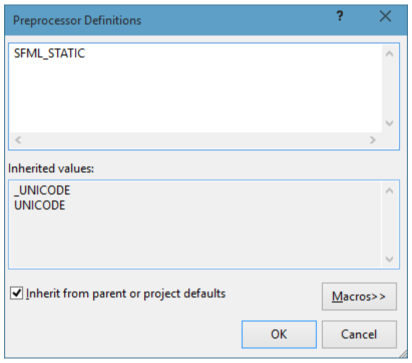


Configuration: Active(Debug) Platform: Active(Win32)

Preprocessor Definitions	WIN32;_DEBUG;_CONSOLE;_LIB;%(PreprocessorDefi
Undefine Preprocessor Definitions	
Undefine All Preprocessor Definitions	No
Ignore Standard Include Paths	No
Preprocess to a File	No
Preprocess Suppress Line Numbers	No
Keep Comments	No

Preprocessor Definitions
Defines a preprocessing symbols for your source file.

Edit this to say just SFML_STATIC:



SFML_STATIC

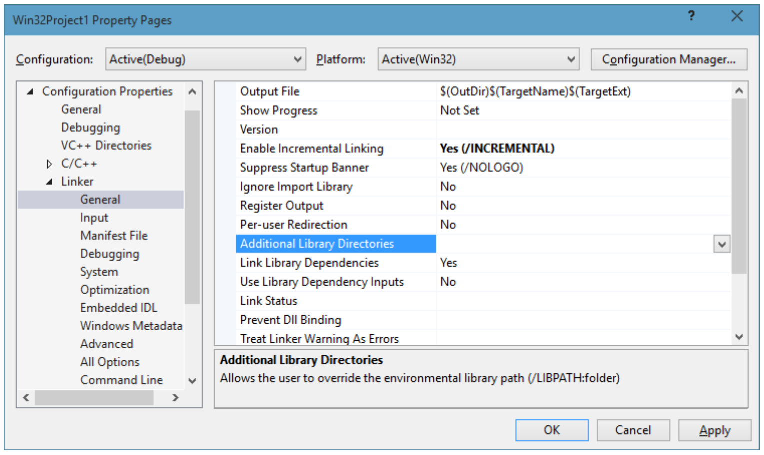
Inherited values:

_UNICODE
UNICODE

☑ Inherit from parent or project defaults

Linker Properties

Now we'll move out of the **C/C++** properties and into the **Linker** properties and **Additional Library Directories**:

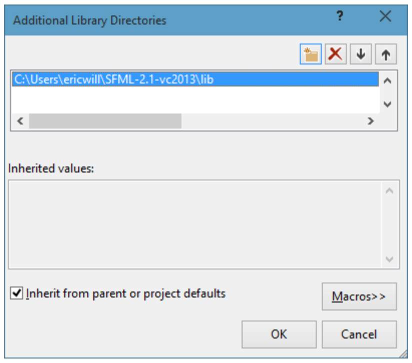


Configuration: Active(Debug) Platform: Active(Win32)

Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	

Additional Library Directories
Allows the user to override the environmental library path (/LIBPATH:folder)

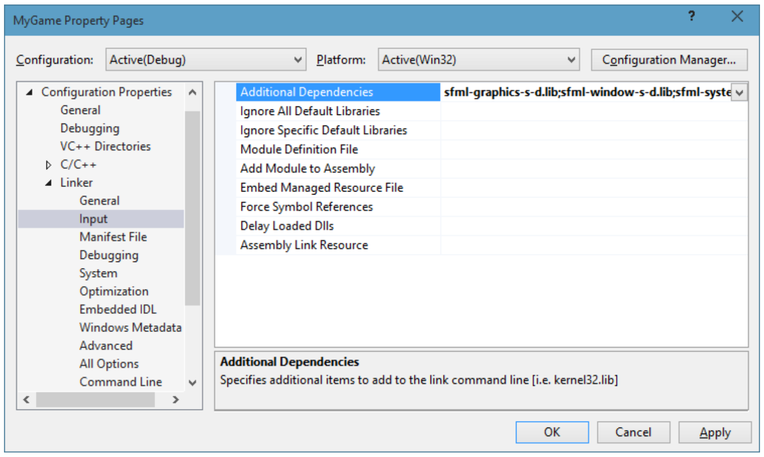
Edit this and set it to your SFML folder followed by **\lib** (e.g. C:\SFML\lib):



C:\Users\ericwill\SFML-2.1-vc2013\lib

☑ Inherit from parent or project defaults

Then go to **Input**, **Additional Dependencies**:



Configuration: Active(Debug) Platform: Active(Win32)

Additional Dependencies	**sfml-graphics-s-d.lib;sfml-window-s-d.lib;sfml-syste**
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line [i.e. kernel32.lib]

Edit this to contain the following:

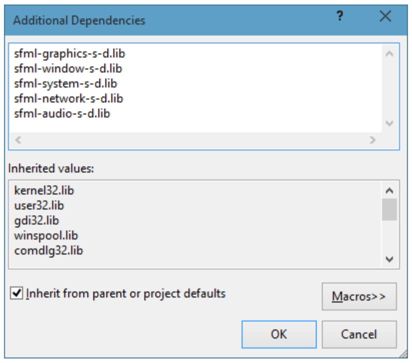
sfml-graphics-s-d.lib

sfml-window-s-d.lib

sfml-system-s-d.lib

sfml-network-s-d.lib

sfml-audio-s-d.lib



```
sfml-graphics-s-d.lib
sfml-window-s-d.lib
sfml-system-s-d.lib
sfml-network-s-d.lib
sfml-audio-s-d.lib
```

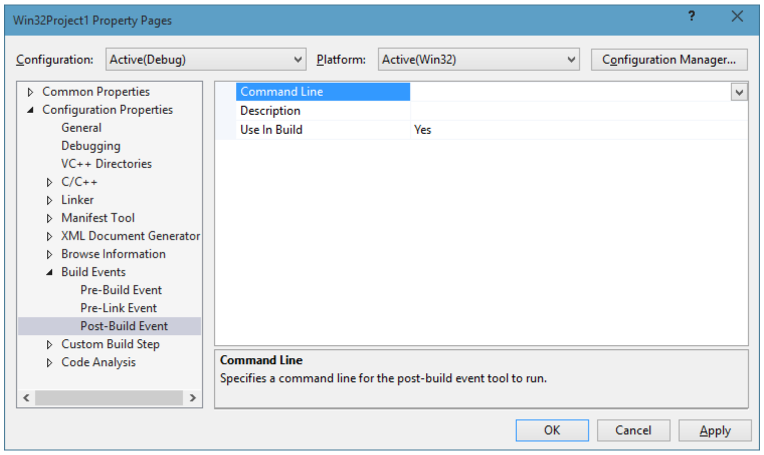
Inherited values:

```
kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
```

☑ Inherit from parent or project defaults

Build Events

Next let's move from **Linker** to **Build Events**, **Post-Build Event**, **Command Line**:



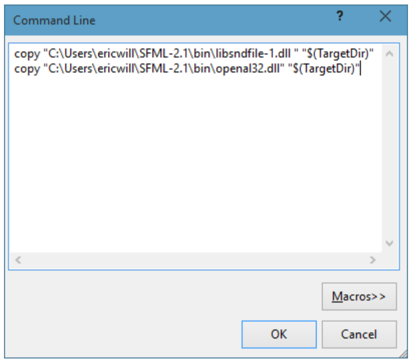
Configuration: Active(Debug) Platform: Active(Win32)

Command Line	
Description	
Use In Build	Yes

Command Line
Specifies a command line for the post-build event tool to run.

Edit this to contain the following bit of script (make sure to put in your own SFML folder name):

```
copy "C:\SFML\bin\libsndfile-1.dll" "$(TargetDir)"
copy "C:\SFML\bin\openal32.dll" "$(TargetDir)"
```



```
copy "C:\Users\ericwill\SFML-2.1\bin\libsndfile-1.dll " "$(TargetDir)"
copy "C:\Users\ericwill\SFML-2.1\bin\openal32.dll" "$(TargetDir)"
```

Test it

Now your project should be set up and ready to go. You can test it out by pasting this little bit of code in your C++ source file and trying to build/run:

```
#include <SFML/Graphics.hpp>

int main()
{
  sf::RenderWindow window(sf::VideoMode(800, 600), "My window");

  sf::CircleShape shape(50);
  shape.setFillColor(sf::Color(255, 0, 0));

  while (window.isOpen())
  {
           sf::Event event;
           while (window.pollEvent(event))
           {
                    if (event.type == sf::Event::Closed) window.close();
           }

           window.clear(sf::Color::Black);
           window.draw(shape);
           window.display();
  }

  return 0;
}
```