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

Let's briefly go over the structure of this document. First, the introduction is going to familiarize you with the base project code. After this, the document is broken into parts, each of which is a small unit where we'll build out some new piece of the game.

The best way to learn from this tutorial is to read through each part and add the new code to your game yourself. If you get stuck, there is a zip file containing the code for each part, giving you a baseline reference to compare your work to. These zip files can be found in the **reference-code** folder.

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. The reference section expands upon what is discussed in the main tutorial and provides links to resources where you can learn even more.

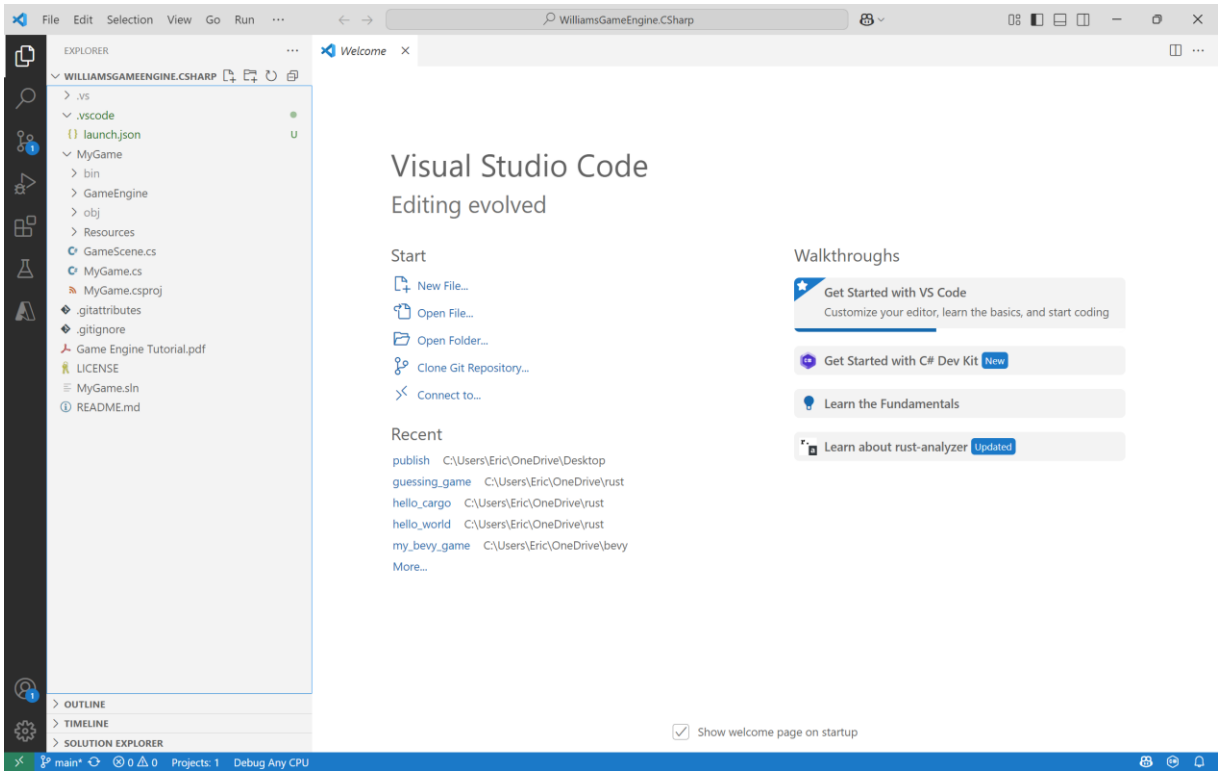
Finally, there may be stuff in this tutorial that you already know, and stuff that you may find utterly confusing. Don't worry if you don't understand everything as you go along. Just keep working at it, and by the end you will have a complete game and enough knowledge to build more games. You can always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

Getting the base project

First things first. You'll need a copy of the base game project. You can find it on GitHub here: https://github.com/MichaelTMiyoshi/WilliamsGameEngine.CSharp

A quick tour of the base project

OK, now that you have gotten the base project onto your computer, let's open it up and take a look. Open the project folder in Visual Studio Code. You should see the project loaded:



Let's see what's inside:

- **MyGame.cs** is the main source file of the game. We'll be looking at it shortly.
- **GameScene.cs** is the main "scene" of our game. We'll be talking a lot about scenes in just a little while.
- **GameEngine** 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. Some resources are already in there if you want to check them out.
- The **.vscode** folder contains the **launch.json** file which tells Visual Studio Code how to launch your game when you hit F5.
- The **.gitattributes** and **.gitignore** files configure Git's behavior.
- **README.md** is a Markdown document that describes the project.
- **LICENSE** is the project's license file which lets others know how they can use the project legally.
- **bin** contains the compiled binaries for your code and **obj** contains intermediate build files used by the compiler. These folders will only appear if you've built the project.
- **reference-code** is a folder containing one zip file for every part of this document, containing the complete project up to that part. If you get stuck, or if you want to compare your code against the reference code, simply unzip the file for that part and check it out. You can even open and run it in Visual Studio Code.

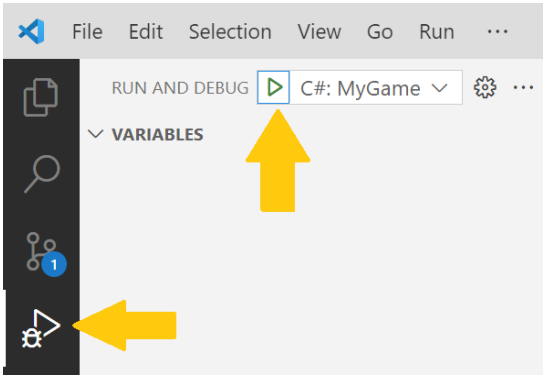
MyGame.cs

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

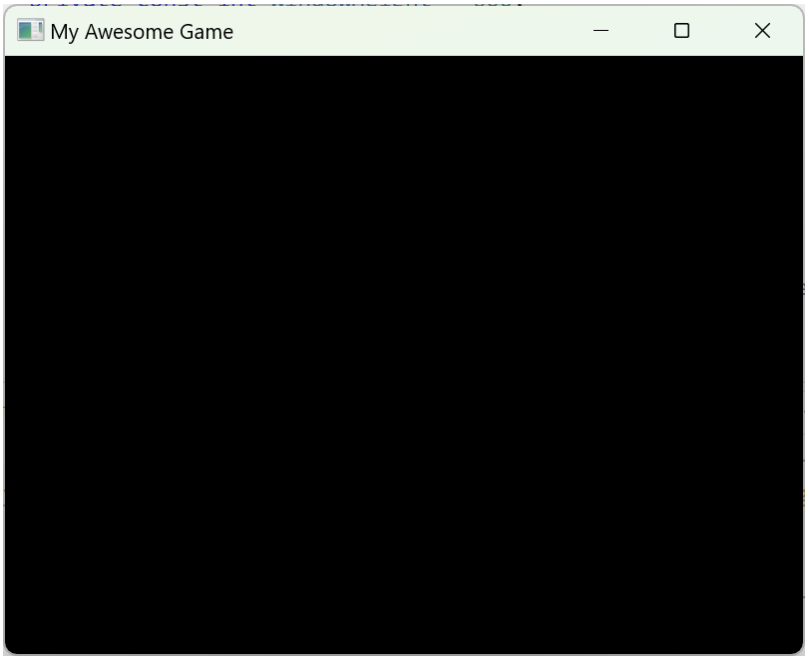
MyGame.cs

```
using GameEngine;

namespace MyGame
{
    public static class MyGame
    {
        private const int WindowWidth = 800;
        private const int WindowHeight = 600;

        private const string WindowTitle = "My Awesome Game";

        public static void Main(string[] args)
        {
            // Initialize the game.
            Game.Initialize(WindowWidth, WindowHeight, WindowTitle);

            // Create our scene.
            GameScene scene = new GameScene();
            Game.SetScene(scene);

            // Run the game loop.
            Game.Run();
        }
    }
}
```

We'll talk about each line of code in just a minute. First, let's run this and see what happens. You can either hit **F5** or click **Run and Debug (Ctrl + Shift + D)** in the left sidebar menu and then the **Start Debugging (F5)** green triangle.



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 (the **X** in the top right corner) to return to the code.

Using the game engine namespace

The first line in **MyGame.cs** is a using directive:

```
using GameEngine;
```

`GameEngine` is a namespace. Namespaces are a way to keep code organized. The `GameEngine` namespace contains all the game engine's types that you will be learning about and using as we go along, such as `Game` and `Scene`. By including this using directive in our code, we make it easier to refer to those types with their simple names. Without the using directive, if you wanted to use `Game` or `Scene` you would have to use their fully qualified names, `GameEngine.Game` and `GameEngine.Scene`, each time you used them. With the using directive, you can just say `Game` or `Scene`, and this keeps code shorter and cleaner.

Defining the game class

Next comes the declaration of the game class:

```
namespace MyGame
{
    public static class MyGame
    {
```

Here we have defined a class called `MyGame` which is inside the `MyGame` namespace. Notice how your game code is in its own namespace (`MyGame`), which is different than the namespace used by the game engine code (`GameEngine`). This keeps the code organized. For example, if you created a class called `Game` in your `MyGame` namespace (which would have the fully qualified name `MyGame.Game`), this would be distinct from the `Game` class inside the `GameEngine` namespace (which would have the fully qualified name `GameEngine.Game`).

Classes are a fundamental building block of object-oriented programming. They allow us to define some data and behavior together and then make a bunch of copies of it. For example, we might have a class that represents an enemy in our game that contains data, like hit points, and behavior, like attacking. We can create a class for our enemy and then *instantiate* many copies of it to have waves of enemies in our game. Each instance will have its own data (also called "state"), so each enemy will have its own hit points, but all instances will share the same behavior for attacking. Creating an instance of a class is done using the **new** operator, and we'll see more about that later.

Sometimes a class is not meant to have multiple copies or be instantiated at all. That is the case with the **MyGame** class. Therefore, we've declared this class as **static**. Static classes do not get instantiated. There aren't multiple copies of them in memory. This is useful for utility classes that have methods but don't need any instance state, or with classes like **MyGame** that are used to start an application. We'll see how this works in a moment when we talk about the **main** method. In C# applications, a start up class like **MyGame** is always static.

Constants

Next, we have a couple of constants:

```
private const int WindowWidth = 800;
private const int WindowHeight = 600;

private const string WindowTitle = "My Awesome Game";
```

WindowWidth and **WindowHeight** specify the size of the window you saw when you ran the game. **WindowTitle** 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!

In C#, variables which are part of a class are called fields. We make a field constant by using **const**. The value of a constant can never change. If you try to assign a new value to a constant, it will generate a compiler error. Making things that can't change constants is good practice, because it helps prevent bugs. You don't want to accidentally change something that shouldn't change.

We also mark these fields as **private**. In C#, private is an access modifier that specifies how other parts of the code can access something. By marking these fields as **private**, we're saying that they can only be used within this class. You could also mark the fields as **public**, in which case any part of the code could access it, but it's a good idea to always restrict access as much as possible. Reducing the number of places in code that can access or modify data helps to reduce bugs, because less access means there are fewer places in the code where a bug could be introduced.

The Main method

Next up, the most important method of them all, the Main method! It's where your program begins and where your program will end. C# always looks for a **Main** method to start your program. This is called the "entry point" into your program, and it is always inside of a static class.

```
public static void Main(string[] args)
```

Our **Main** method is **static**. A rule about static classes is that all the members in them (all their fields and methods) must also be static.

Notice that the return type of this method is **void**. This just means that our method doesn't return any value at all. **Main** methods can return a value if you want, but it's not really useful here.

Finally, a `Main` method can accept a string array of arguments, which we refer to here as `args`. This allows anyone who runs our program to pass in some configuration that can control how our program behaves. You could, for example, pass in the window height and width to override the default values. We won't be exploring that in this tutorial, but it's useful to know it exists.

Initializing the game

Next, we call the `Initialize` method of `Game`:

```
Game.Initialize(WindowWidth, WindowHeight, WindowTitle);
```

This is something we only need to do once. Calling this method sets up the window into which our game will render. It takes as input the window width, height and title. `Game` is a special class in our game engine. Its primary responsibility is to run our game loop. The game loop is a sequence of steps that are run every frame to keep the game running, like updating enemies, drawing things onto the screen, responding to keyboard input, and so on.

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.

Our game project defines a class called `GameScene`. We create an instance of this class and call it `scene`. We will use this to hold all the stuff in our main game scene. In Part 2 we'll begin adding stuff to it and seeing it on the screen.

```
GameScene scene = new GameScene();
```

Here we see the `new` operator mentioned previously. `GameScene` is not a static class. This means we can create multiple copies of it if we want to, and we *instantiate* each one using the `new` operator. Notice that to use the `new` operator we follow it by the name of the class that we want to make a *new* instance of. In this case, we are making a new `GameScene` instance. Notice also that the name of the class we are instantiating is followed by parenthesis. As you've seen before, parenthesis are used when we pass arguments into a method. This is no different. Creating an instance of a class is done by calling a special method on that class called a constructor. A class can have multiple constructors, each of which may take different arguments. In the case of `GameScene`, it has a single constructor which takes no arguments, so our parenthesis are empty.

Now that we have an instance of `GameScene`, we tell `Game` to use it as the current scene:

```
Game.SetScene(scene);
```

The game engine can switch from one scene to another. When you call `SetScene`, the transition will happen. You must call `SetScene` before you can run the game at all, because without a scene the game engine has nothing to do.

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:

```
Game.Run();
```

The game loop will keep on running until the window is closed. Once that happens, our `Main` method 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

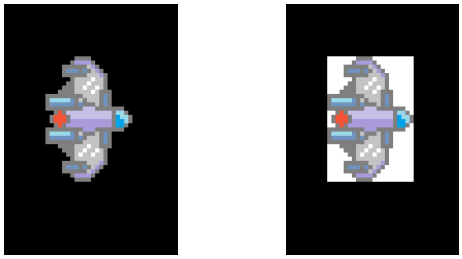
Talk is cheap. Show me the code.
Linus Torvalds

Creating a sprite resource

Let's get our player sprite on the screen. For this, you can create your own sprite image in a program like Paint or Photoshop or Paint.NET, or you can use one of the images already created for you. All of the game's resources are located inside of the **Resources** folder, such as this one named **ship.png:**

If you make your own resources, they should also go into the **Resources** 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 the background in your game, which might not be what you want. 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.

The GameObject class

OK folks, it's time to get **classy**. Our game engine is object oriented, so everything we create in our game (monsters, spaceships, 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.cs** 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.cs

```
using System.Collections.Generic;
using SFML.Graphics;
using SFML.System;
```

```
namespace GameEngine
{
    // This class represents every object in your game, such as the player, enemies, and so on.
    public abstract class GameObject
    {
        private bool _isCollisionCheckEnabled;

        private bool _isDead;

        // Using a set prevents duplicates.
        private readonly HashSet<string> _tags = new HashSet<string>();

        // Tags let you annotate your objects so you can identify them later
        // (such as "player").
        public void AssignTag(string tag)
        {
            _tags.Add(tag);
        }

        public bool HasTag(string tag)
        {
            return _tags.Contains(tag);
        }

        // "Dead" game objects will be removed from the scene.
        public bool IsDead()
        {
            return _isDead;
        }

        public void MakeDead()
        {
            _isDead = true;
        }

        // Update is called every frame. Use this to prepare to draw (move, perform AI, etc.).
        public abstract void Update(Time elapsed);

        // Draw is called once per frame. Use this to draw your object to the screen.
        public 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.
        public bool IsCollisionCheckEnabled()
        {
            return _isCollisionCheckEnabled;
        }

        public void SetCollisionCheckEnabled(bool isCollisionCheckEnabled)
        {
            _isCollisionCheckEnabled = isCollisionCheckEnabled;
        }

        // This function lets you specify a rectangle for collision checks.
        public virtual FloatRect GetCollisionRect()
        {
            return new FloatRect();
        }

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

Using directives

At the top of the file we have our using directives, letting the compiler know we want to use things from these namespaces using their simple names:

```
using System.Collections.Generic;
using SFML.Graphics;
using SFML.System;
```

Class declaration

Just as we saw with MyGame.cs, we now have a class declaration, again inside a namespace:

```
namespace GameEngine
{
    public abstract class GameObject
    {
```

This says that we have a `GameObject` class inside the `GameEngine` namespace. Everything else declared in this file will be inside of the `GameObject` class, and therefore one of its members.

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 methods for dealing with tags:

```
public void AssignTag(string tag)
public bool HasTag(string tag)
```

The `AssignTag` method sticks a `string` tag on a `GameObject`, and the `HasTag` method returns a `bool` that tells you if a tag has been stuck to the object.

The data that these functions use is stored in an instance field called `_tags`:

```
private readonly HashSet<string> _tags = new HashSet<string>();
```

There is a lot going on in this declaration! Let's break it down.

Remember when we saw `WindowWidth`, `WindowHeight` and `WindowTitle` in the `MyGame` class? Like those, `_tags` is a field. That is, it's a variable that is defined at the class level and is part of the class which contains it. Also like those fields, `_tags` is marked as private, meaning it can only be used within the class itself. No other part of the code can access it.

Unlike the fields we saw in MyGame, the `_tags` field is not marked `const`. This means it is not a constant. But it is marked as `readonly`. When something is marked as readonly in C#, it means its value can only be set once, either when it's declared or in a constructor. This is different than `const` fields, which can only be set at declaration time.

Why use `const` and `readonly`? They are part of a best practice in software engineering called *immutability*. In our program, we only want data to change when it needs to. Unexpected changes to data can cause bugs. When we know something only needs to be set once when the program is running and should never be set again, we can make sure this is enforced by marking it `readonly`. And if we know that something has a constant value that can be declared in the code and doesn't need to be set or changed when the program is running, we can make it `const`.

Notice that `_tags` is not marked `static`. Each instance of `GameObject` will have its own `_tags` data. This isn't something you can do with `static`. With `static`, the data is shared across all instances.

Also notice that we have named this field `_tags`, with a preceding underscore. This isn't strictly necessary, but it is a common practice when using object-oriented languages like C# or Java to name private fields with an underscore. This helps to keep their names obviously separate from the names of variables you declare inside of methods, helping to reduce bugs.

The `_tags` field is of type `HashSet<string>`. A `HashSet` is a type in C# that allows you to have a "set" of things where each thing is unique. For example, you could have a set that is **["apple", "banana", "pear"]** or a set that is **[10, 15, 42]**. You can add things to a `HashSet`, and if the thing you're adding already exists it won't be duplicated. This is useful for tags on a game object because we can just add tags and not worry about whether they've been added already, and we can know that a tag only exists once on a given game object.

What's up with the `<string>` part of `HashSet<string>`? This is using a feature of C# called generics. `HashSet` is a "container" type. This means it can store stuff. But what *kind* of stuff does it store? Generics allows us to specify this. In the examples given above, the set **["apple", "banana", "pear"]** would require a `HashSet<string>` because the stuff in the set is all strings, and the set **[10, 15, 42]** might use a `HashSet<int>` because the stuff in the set is all integers. A `HashSet<int>` could *never* contain **["apple", 15]** because "apple" is not an `int`. Generics enforce which types can be used and keep code cleaner and more bug-free. Most generic container types in C# have a non-generic equivalent that would let you store items of different types, but unless you have a good reason to mix and match types it's best to use generics to constrain them.

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.

```
public bool IsDead()
public void MakeDead()
```

The `IsDead` method tells us whether the object is dead, which will be used to let the scene know that the game object should be removed. The `MakeDead` method will set this flag to `true`. Permanently. Mua ha ha. The data for this flag is the `_isDead` field:

```
private bool _isDead;
```

You may have noticed that in the case of tags and the "is dead" flag we don't just make the fields which store this data `public` and allow code outside the `GameObject` class to modify that data directly. Instead, we encapsulate the data by requiring other parts of the code to use methods (such as `MakeDead` or `AssignTag`) to access it. This is a common practice in object-oriented software engineering. Encapsulation hides the details of how a class is implemented, so the implementation could change without impacting other parts of the code that use the class. For example, if some day we decided we wanted to store tags in a `Dictionary` instead of a `HashSet`, encapsulation would allow us to make that change without requiring anything outside of the `MyGame` class to deal with it. Encapsulation can save a ton of work and prevent a lot of bugs.

Update and Draw

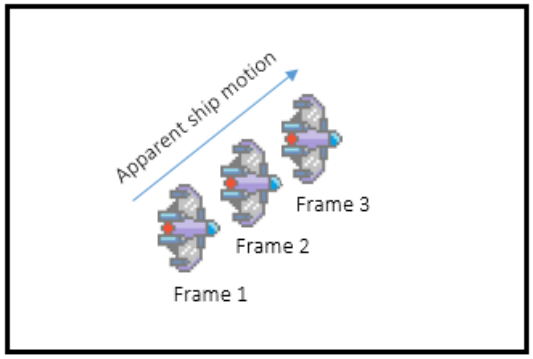
The `Update` and `Draw` methods, together, are the two most important methods in your game objects. These two methods are called once each frame, one after the other, for every game object in the scene. Together, they make up the core of your game loop. Before we talk about these method signatures, let's talk about what they do.

The `Update` method 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` method is where you put code to draw the game object to the screen. The `Update` method 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` method must draw it. You can't just draw the object to the screen once, even if it doesn't change. You must 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.



Now let's talk about the signatures of `Update` and `Draw`:

```
public abstract void Update(Time elapsed);
public virtual void Draw()
```

First, notice that `Update` is called with a reference to a `Time` instance. This tells us 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.

Next, notice that `Update` is marked as **abstract**. Abstract methods in C# are methods which aren't implemented in the class where they are declared but must instead be implemented in a subclass. This is where we can start to see some of the real power of object-oriented programming. Abstract methods allow us to have different behavior in each of the derived types of a parent type. This works out great for our `GameObject`, because our game needs different kinds of game objects with different kinds of behavior. How each game object updates itself will be distinct. Abstract methods don't have a body in the parent class where they are declared, meaning they aren't followed by brackets (`{}`). Instead, the declaration ends with a semi-colon (`;`). Later on, we'll see how we go about providing an implementation of `Update` in a `GameObject` subclass.

Finally, notice that the `Draw` method is marked as `virtual`. In C#, a virtual method is one that can be overridden in a subclass. This means that a subclass can provide a new implementation of the method, if it wants. Unless a method is marked as `virtual`, it can't be overridden like this. Whereas with `abstract` a parent class will declare a method that all subclasses *must* implement, `virtual` methods have an implementation in a parent class that subclasses *may* override, but only if they want to. In the case of our `Draw` method, the implementation in the parent class `GameObject` is just empty. It doesn't do anything. For most of our game objects, we'll want to override `Draw` so they can be rendered to the screen. We'll see this in action later on.

Collisions

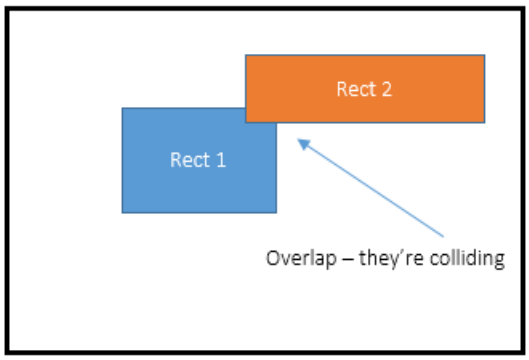
The `GameObject` class has four methods related to collisions:

```
public bool IsCollisionCheckEnabled()
```

```
public void SetCollisionCheckEnabled(bool isCollisionCheckEnabled)
public virtual FloatRect GetCollisionRect()
public virtual void HandleCollision(GameObject otherGameObject)
```

Collision handling is also part of our game loop. Every frame, the current scene will check for collisions between game objects. It will only check objects that return `true` when their `IsCollisionCheckEnabled` method 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` method allows us to say if an object should be checked or not. You can see in the code that these two methods encapsulate the `_isCollisionCheckEnabled` private instance field, which is of type `bool`.

The `GetCollisionRect` method 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.



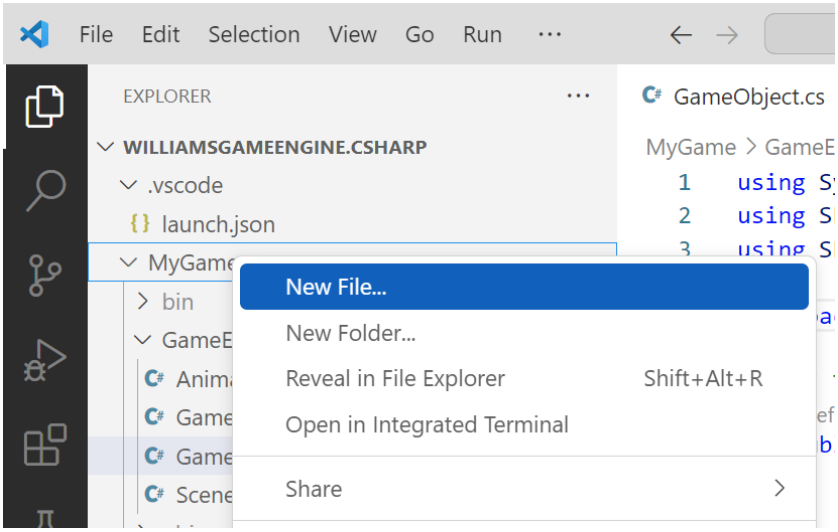
When collision rectangles overlap, a collision occurs. Both game objects in the collision have their `HandleCollision` methods called. This method is provided with a reference to the other game object in the collision. The `HandleCollision` method is where you can put any special logic or behavior that should happen upon collision, such as changing direction, losing health points, etc. We'll cover collisions in more detail in Part 6.

Creating a game object for the ship

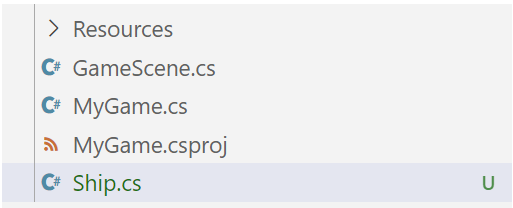
OK, now that we know a little more about the `GameObject` class, it's time to create a subclass of our own that represents our ship.

Create the class file

In Visual Studio Code's Explorer, right click on **MyGame** (the folder where we'll create our file) and select **New File**. Name this file **Ship.cs**.



You should see the new file in the Explorer:



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

Ship.cs

```
using GameEngine;
using SFML.Graphics;
using SFML.System;
using SFML.Window;

namespace MyGame
{
   public class Ship : GameObject
   {
       private readonly Sprite _sprite = new Sprite();

       // Creates our ship
       public Ship()
       {
           _sprite.Texture = Game.GetTexture("Resources/ship.png");
```

```
            _sprite.Position = new Vector2f(100, 100);
        }

        // Draws our ship
        public override void Draw()
        {
            Game.RenderWindow.Draw(_sprite);
        }

        // Updates our ship every frame
        public override void Update(Time elapsed)
        {
            // Do nothing just yet
        }
    }
}
```

Notice that the **Ship** class extends **GameObject**:

```
public class Ship : GameObject
```

This means that any instance of **Ship** is also an instance of **GameObject**. Every kind of game object we create and add to our game will be a subclass of **GameObject**.

Our ship has a private instance field for our Sprite:

```
private readonly Sprite _sprite = new Sprite();
```

Sprite is a type defined in the SFML library and is what we will use to draw graphics onto the screen.

We have a constructor, which will create a new instance of our class:

```
public Ship()
{
    _sprite.Texture = Game.GetTexture("Resources/ship.png");
    _sprite.Position = new Vector2f(100, 100);
}
```

This constructor sets the texture of our ship to the **ship.png** file in the **Resources** folder. The texture is an image file with the pixels we want to draw on the screen. If you've made your own ship texture with a different name, you would specify that here. To load the texture, we call the **GetTexture** method of **Game** and provide it with the file name of the texture we want. Next, we put the **Sprite** at x, y position 100, 100 via its **SetPosition** method, which taxes a **Vector2f**.

Next, we implement the **Draw** method:

```
public override void Draw()
{
    Game.RenderWindow.Draw(_sprite);
}
```

Our **Draw** method simply draws the sprite by calling the **Draw** method of **RenderWindow**. The **RenderWindow** represents the area of the screen where we will render (draw) our graphics, and we get it from **Game**. Our **Draw** method will do this every single frame.

Finally, we implement the `Update` method:

```
public override void Update(Time elapsed)
{
    // Do nothing just yet
}
```

Remember that `Update` is defined as `abstract` in our parent class, `GameObject`. That means we must implement it. But we don't really have anything to do yet. Our ship won't currently 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

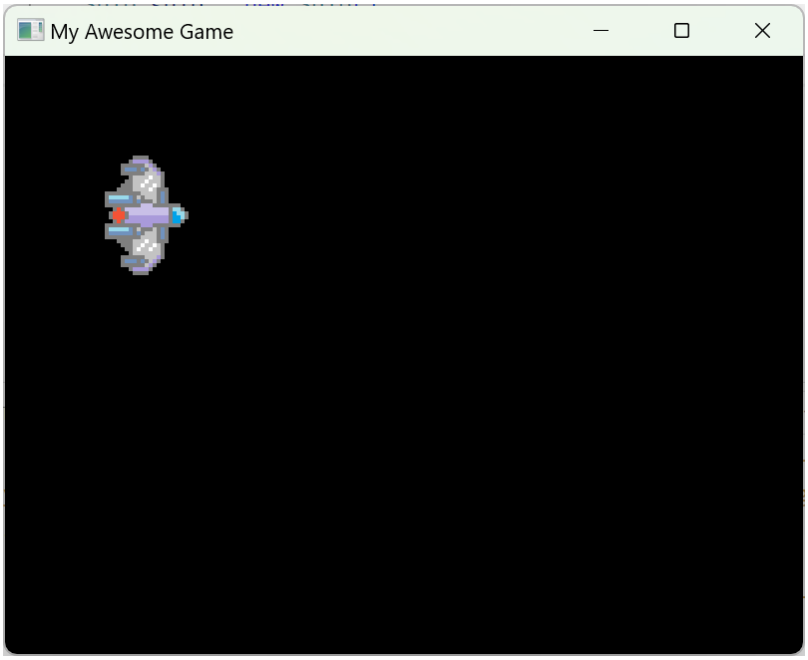
OK, the last thing we need to do before we're done with part 1 is get our ship in the scene. To do this, we need to make some changes to GameScene.cs. Note that when describing changes to an existing file, we won't always show all the source code. We'll highlight the areas you need to change. Remember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

GameScene.cs

```
using GameEngine;

namespace MyGame
{
    public class GameScene : Scene
    {
        public GameScene()
        {
            Ship ship = new Ship();
            AddGameObject(ship);
        }
    }
}
```

Here we have created a new `Ship` instance and added it to our scene via AddGameObject.

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.

So, the challenge for Part 1 is to customize the ship. Either modify the image texture provided, or you can make your own from scratch. 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 add code to our ship's `Update` method. Recall that `Ship` is a `GameObject`, and game objects have `Update` and `Draw` methods. Right now, we've implemented the `Draw` method, but if we want our object to change over time we also need to implement the `Update` method. Let's do it.

Override the Update method in Ship.cs

Open up **Ship.cs**, and let's add some code to the update method, along with a new constant:

Ship.cpp

```
// omitted code

namespace MyGame
{
    public class Ship : GameObject
    {
        private const float Speed = 0.3f;

        // omitted code

        public override void Update(Time elapsed)
        {
            Vector2f pos = _sprite.Position;
            float x = pos.X;
            float y = pos.Y;
            int msElapsed = elapsed.AsMilliseconds();

            if (Keyboard.IsKeyPressed(Keyboard.Key.Up)) { y -= Speed * msElapsed; }
            if (Keyboard.IsKeyPressed(Keyboard.Key.Down)) { y += Speed * msElapsed; }
            if (Keyboard.IsKeyPressed(Keyboard.Key.Left)) {x -= Speed * msElapsed; }
            if (Keyboard.IsKeyPressed(Keyboard.Key.Right)) {x += Speed * msElapsed; }

            _sprite.Position= new Vector2f(x, y);
        }
    }
}
```

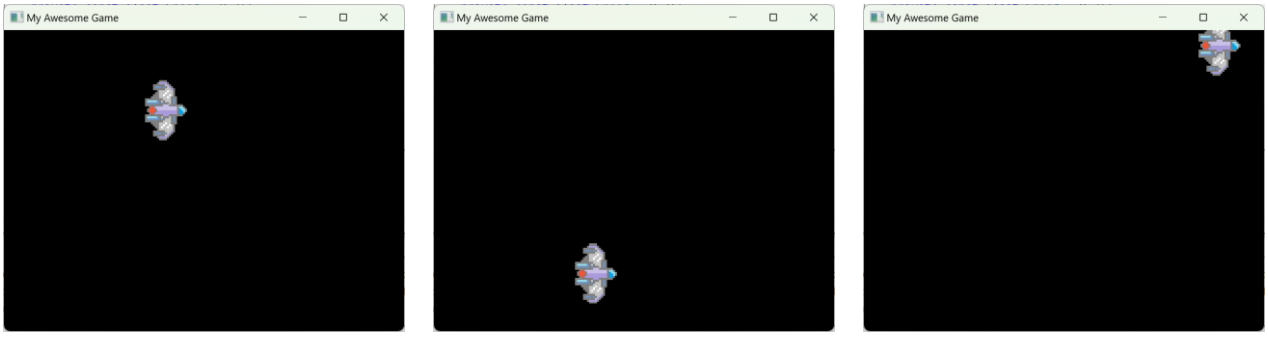
First, the `Update` method gets the current position of our sprite and its x and y values. Then it determines how many milliseconds have elapsed since the last frame. Next, it updates the values of x and y depending on which keys are pressed, which it determines via the `Keyboard` class. Finally, it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

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` method 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 maybe you could 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 spaceship without a laser cannon? Let's add one. In the resources folder you should find **laser.png**. Feel free to use this for the laser image texture or create your own.

Create Laser.cs

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

Laser.cs

```
using GameEngine;
using SFML.Graphics;
using SFML.System;

namespace MyGame
{
    class Laser : GameObject
    {
        private const float Speed = 1.2f;

        private readonly Sprite _sprite= new Sprite();

        public Laser(Vector2f pos)
        {
            _sprite.Texture = Game.GetTexture("Resources/laser.png");
            _sprite.Position = pos;

            AssignTag("laser");
        }

        public override void Draw()
        {
            Game.RenderWindow.Draw(_sprite);
        }

        public override void Update(Time elapsed)
        {
            int msElapsed = elapsed.AsMilliseconds();
            Vector2f pos = _sprite.Position;
```

```
            if(pos.X > Game.RenderWindow.Size.X)
            {
                MakeDead();
            }
            else
            {
                _sprite.Position = new Vector2f(pos.X + Speed * msElapsed, pos.Y);
            }
        }
    }
}
```

Our `Laser` class is a lot like our `Ship` class. It derives from `GameObject` and overrides the `Draw` and `Update` methods. 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.

The constructor for `Laser` loads the laser image texture and sets it on the sprite. It also sets the sprite's position to the position provided. Finally, it assigns the "laser" tag. This tag will come in handy later when we want to check for collisions.

The `Draw` method is identical to the one in Ship.cs. It just draws the sprite to the screen.

Our `Update` method 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 the `MakeDead` method 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.

Update Ship.cs

Now that we have a laser, we have to add some logic to the ship code to fire it. First let's modify Ship.cs:

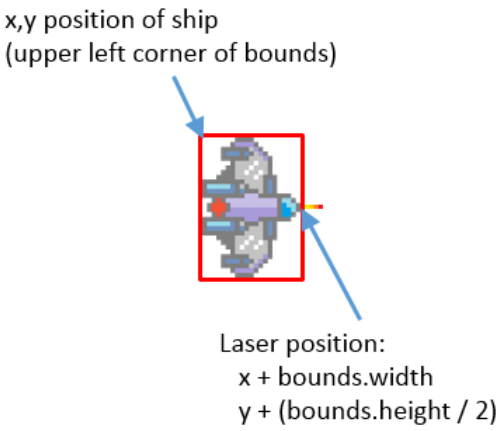
Ship.cs

```
// omitted code

namespace MyGame
{
    public class Ship : GameObject
    {
        private const float Speed = 0.3f;
        private const int FireDelay = 200;

        private int _fireTimer = 0;

        // omitted code
```

```
        public override void Update(Time elapsed)
        {
            Vector2f pos = _sprite.Position;
            float x = pos.X;
            float y = pos.Y;
            int msElapsed = elapsed.AsMilliseconds();

            if (Keyboard.IsKeyPressed(Keyboard.Key.Up)) { y -= Speed * msElapsed; }
            if (Keyboard.IsKeyPressed(Keyboard.Key.Down)) { y += Speed * msElapsed; }
            if (Keyboard.IsKeyPressed(Keyboard.Key.Left)) {x -= Speed * msElapsed; }
            if (Keyboard.IsKeyPressed(Keyboard.Key.Right)) {x += Speed * msElapsed; }

            _sprite.Position= new Vector2f(x, y);

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

            if(Keyboard.IsKeyPressed(Keyboard.Key.Space) && _fireTimer <=0)
            {
                _fireTimer = FireDelay;

                FloatRect bounds = _sprite.GetGlobalBounds();
                float laserX = x + bounds.Width;
                float laserY = y + bounds.Height / 2.0f;

                Laser laser = new Laser(new Vector2f(laserX, laserY));
                Game.CurrentScene.AddGameObject(laser);
            }
        }
    }
}
```

We've added an instance field 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. At the bottom of the ship's update method, 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.

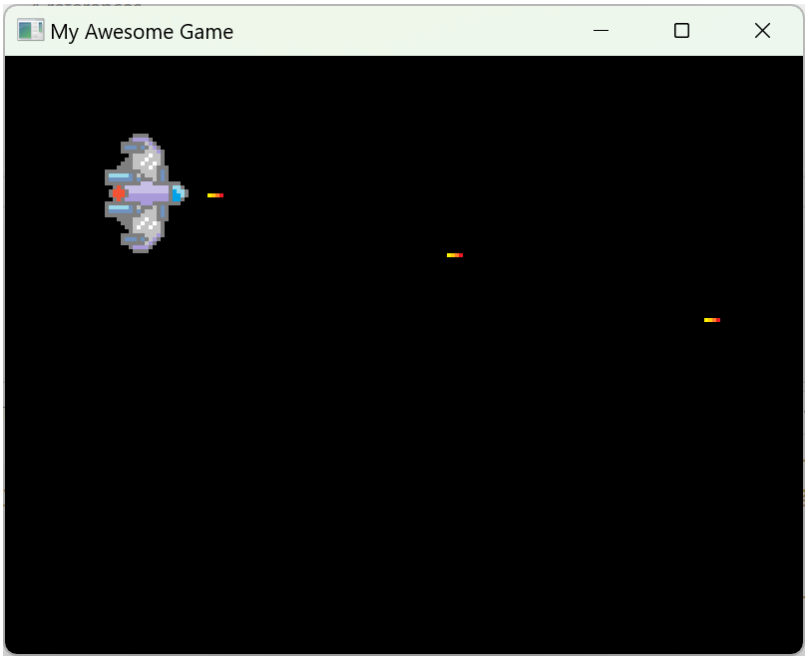
To actually fire the laser, we create a new instance of **Laser** and then add it to the scene. In the call to **Laser**'s constructor, we pass the **Vector2f** position indicating where it should appear on the screen. We want the laser to show up right in front of the ship, so we compute the position 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 halfway 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 that instead of 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.cs as a basis for Meteor.cs. If you want to test out your meteor, simply add a couple lines to the constructor in GameScene.cs to create a meteor and add it to the scene.

Turn the page to see the code!

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

Meteor.cs

```
using GameEngine;
using SFML.Graphics;
using SFML.System;

namespace MyGame
{
    class Meteor : GameObject
    {
        private const float Speed = 0.5f;

        private readonly Sprite _sprite = new Sprite();

        public Meteor(Vector2f pos)
        {
            _sprite.Texture = Game.GetTexture("Resources/meteor.png");
            _sprite.Position = pos;

            AssignTag("meteor");
        }

        public override void Draw()
        {
            Game.RenderWindow.Draw(_sprite);
        }

        public override void Update(Time elapsed)
        {
            int msElapsed = elapsed.AsMilliseconds();
            Vector2f pos = _sprite.Position;

            if(pos.X < _sprite.GetGlobalBounds().Width * -1)
            {
                MakeDead();
            }
            else
            {
                _sprite.Position = new Vector2f(pos.X - Speed * msElapsed, pos.Y);
            }
        }
    }
}
```

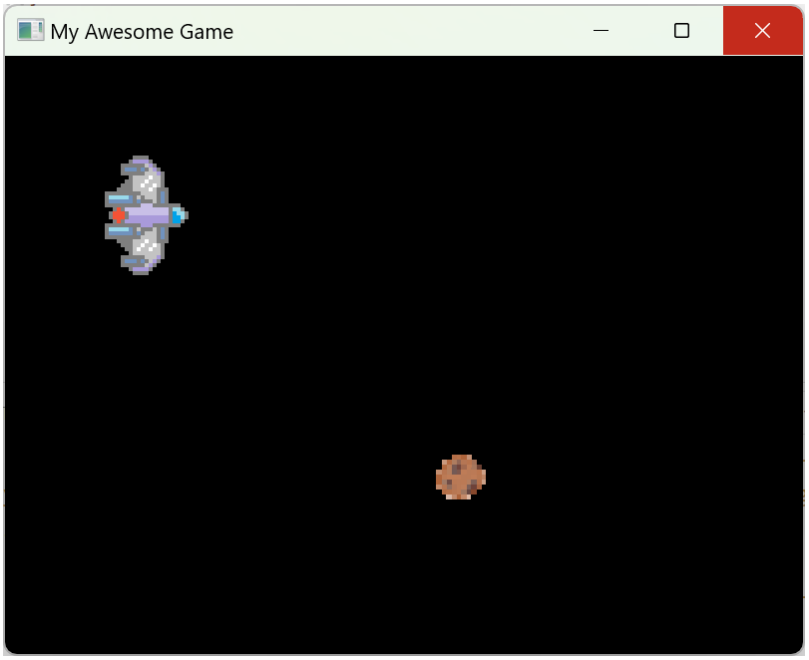
To test out your meteor, modify the constructor in GameScene.cs to create a `Meteor` and add it to the scene:

```
Meteor meteor = new Meteor(new Vector2f(800, 400));
AddGameObject(meteor);
```

Note that for this code to work, you'll need a using declaration at the top:

```
using SFML.System;
```

Once you run the project, you should see a meteor floating by:



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.cs 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.cs

```
using GameEngine;
using SFML.Graphics;
using SFML.System;

namespace MyGame
{
    class MeteorSpawner : GameObject
    {
        // The number of milliseconds between meteor spawns.
        private const int SpawnDelay = 1000;

        private int _timer;

        public override void Update(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 = SpawnDelay;
                Vector2u size = Game.RenderWindow.Size;

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

                // Spawn the meteor somewhere along the height of the window, randomly
                float meteorY = Game.Random.Next() % size.Y;

                // Create a meteor and add it to the scene
                Meteor meteor = new Meteor(new Vector2f(meteorX, meteorY));
                Game.CurrentScene.AddGameObject(meteor);
            }
        }
    }
}
```

The spawner implements the `Update` method but because it's not going to draw itself to the screen it doesn't override the `Draw` method. Note that we have a private `_timer` value, which we'll use in the implementation to control when we spawn a 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.

Let's take a closer look at how we generate the meteor's Y position:

```
float meteorY = Game.Random.Next() % size.Y;
```

`Game` provides a property, `Random`, which returns a random number generator of type `System.Random`, a type which is part of .NET itself. This random number generator allows us to… generate random numbers. Its `Next()` method returns an integer between 0 and `Int32.MaxValue` (which is the biggest possible `int`, or 2,147,483,647). We modulo (`%`) this number with the `size.Y` dimension (the height) of the screen. Doing modulo to it means that we only get the remainder, which will always be between 0 and `size.Y - 1`. In other words, it constrains the Y position of the meteor to an actual position on the screen.

Random numbers are interesting, as are random number generators. There are fabulous rabbit holes waiting to be explored here, along with connections to cryptography, physics, and phoilosphy.

Add the Meteor Spawner to our scene

Modify **GameScene.cs** to add the Meteor Spawner. Note that if you added meteor test code here in the previous section, you can remove it now.

GameScene.cs

```
using GameEngine;

namespace MyGame
{
    public class GameScene : Scene
    {
        public GameScene()
        {
            Ship ship = new Ship();
            AddGameObject(ship);

            MeteorSpawner meteorSpawner = new MeteorSpawner();
            AddGameObject(meteorSpawner);
        }
    }
}
```

Watch out! Cookies ahead!

Test your skills 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?

Extra Challenge: Better randomness

It should be pointed out that the `System.Random` instance `Game` returns via its `Random` property is seeded with a constant value (42, in fact). Our random number generator is actually a *pseudorandom* number generator. That means it produces numbers which kinda look random, but they're really not. The seed you provide to a random number generator determines the random numbers you get. With a constant seed, you get the same sequence every time. You can verify this yourself: Run the game multiple times in a row, and notice that the meteors spawn in the same places, in the same order, every time. One way to get better random numbers is to use a different seed each time you run the game. A typical seed for this is the system time. Can you figure out how to use the system time to seed the random number generator?

Part 6: Shooting the Meteors

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

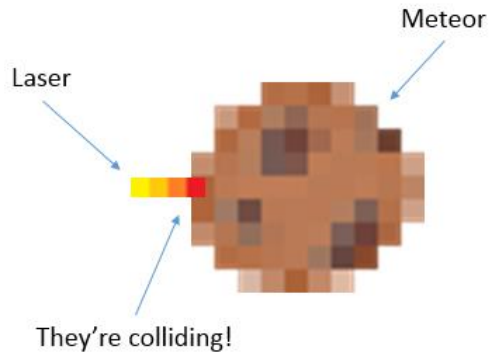
Bill Gates

Time for collisions

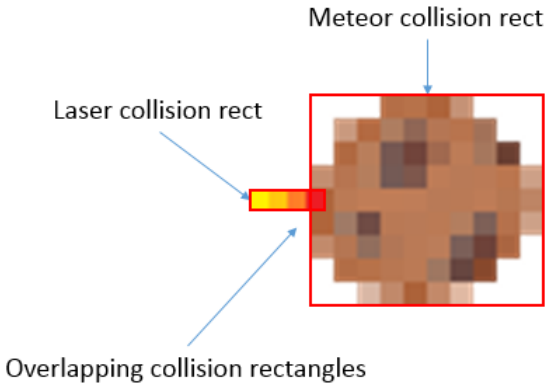
It would be fun to blast these meteors into atoms, don't you think? Having the lasers destroy the meteors is actually really easy. All we need to do is handle the collisions that happen between them, and when a collision occurs, remove both. We'll make a couple of small changes to our `Laser` and `Meteor`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

How do collisions work?

First, let's take a moment to talk about collisions. In our game engine, a collision can happen between two `GameObject` instances 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 comparing these rectangles:



The collision rectangle returned from a game object's `GetCollisionRect` method is of type `FloatRect`. This type has float values `Left`, `Top`, `Width` and `Height`. 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 as a lot of unnecessary checking would be done.

To tell the engine to check our game object for collisions, we simply call the `SetCollisionCheckEnabled` method of `GameObject` 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 game engine will check on each frame. To do so, the engine asks whether the collision rectangle of the game object being checked 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.cs.)

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`. Add this to the bottom of the `Laser` class:

Laser.h

```
// omitted code

namespace MyGame
{
    class Laser : GameObject
    {
        // omitted code
```

```
        public override FloatRect GetCollisionRect()
        {
            return _sprite.GetGlobalBounds();
        }
    }
}
```

Notice how the collision rect we're returning comes from the laser sprite's `GetGlobalBounds` method. This method 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.cs

```
// omitted code

namespace MyGame
{
    class Meteor : GameObject
    {
        // omitted code

        public Meteor(Vector2f pos)
        {
            _sprite.Texture = Game.GetTexture("Resources/meteor.png");
            _sprite.Position = pos;

            AssignTag("meteor");
            SetCollisionCheckEnabled(true);
        }

        public override FloatRect GetCollisionRect()
        {
            return _sprite.GetGlobalBounds();
        }

        public override void HandleCollision(GameObject otherGameObject)
        {
            if (otherGameObject.HasTag("laser"))
            {
                otherGameObject.MakeDead();
            }
```

```
            MakeDead();
        }

        // omitted code
    }
}
```

Just like `Laser`, `Meteor` uses its sprite's `GetGlobalBounds` function for its collision rectangle. In `Meteor`'s `HandleCollision` method, 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!

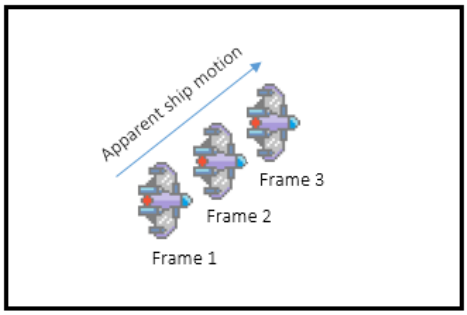
On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

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 derived from `GameObject`. That is, they are all subclasses of GameObject. Because of this, instances of `Ship`, `Laser`, `Meteor` and `MeteorSpawner` are also all instances of `GameObject`, too.

Here's `Ship`, for example:

```
public class Ship : 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.cs.)

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

```
public class AnimatedSprite : GameObject
```

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

Constructor

Here's the constructor for `AnimatedSprite`:

```
public AnimatedSprite(Vector2f position, int msPerFrame = DefaultMsPerFrame)
```

The first argument to this constructor, `position`, looks pretty similar to what we've seen before. It lets us set the position of the sprite on the screen. However, we also have an argument called `msPerFrame`, which has a default value of `DefaultMsPerFrame`, defined as follows:

```
private const int DefaultMsPerFrame = 20;
```

Because `msPerFrame` 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 the `Texture` property:

```
public Texture Texture
{
    get { return _sprite.Texture;  }
    set { _sprite.Texture = value; }
}
```

Properties are something we haven't seen yet in this tutorial. They are a blend of a field and a method. Code which uses the property can just treat it like a field. For example, you could set a texture by assignment:

```
myAnimatedSprite.Texture = myTexture; // calls set
```

Or:

```
myTexture = myAnimatedSprite.MyTexture; // calls get
```

Properties can be used in this way, like fields, but are implemented by *accessors*, either `get` (when returning the value) or `set` (when setting the value). These accessors can do all kinds of stuff, like validate that the value is within an acceptable range, write the value to a file, and so on. In the `Texture` property above, the accessors encapsulate access to `_sprite.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. Those classes used "static" textures, that is, textures which are not animated, such as ship.png (not to be confused with `static` members, that's a different kind of "static"). But 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 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 you have assigned it to your `AnimatedSprite` via the `Texture` property, you are ready to create animations. To do so, you need to tell your `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 method we use to create animations is `AddAnimation`:

```
public void AddAnimation(string name, List<IntRect> frames)
```

This function takes a name for the animation and a `List` of `IntRect`. 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` method:

```
public void PlayAnimation(string name, AnimationMode mode)
```

This function takes the name of the animation and an AnimationMode value. `AnimationMode` is an enum defined in `AnimatedSprite`. Enums allow us to specify a set of named constants, constraining the possible values of an enum instance to that set. Variables of type `AnimationMode` could have as their value just one of the following:

```
public 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 frame of animation for now but maybe want to play a different animation later.

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

```
public bool IsPlaying()
```

Position and origin mode

`AnimatedSprite` contains the property `Position`, which allows you to change the sprite's position on the screen:

```
public Vector2f Position
```

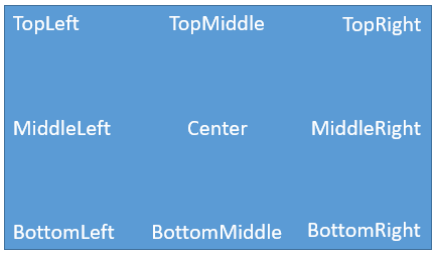
`Position` works in concert with the `SetOriginMode` method:

```
public void SetOriginMode(OriginMode originMode)
```

Together, `Position` and `SetOriginMode` control where your sprite appears on the screen. To understand this, let's first look at the set of possible origin modes. `OriginMode` is defined in `AnimatedSprite`:

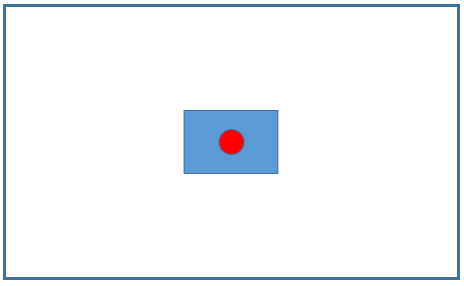
```
public enum OriginMode
{
    TopLeft,
    TopMiddle,
    TopRight,
    MiddleLeft,
    Center,
    MiddleRight,
    BottomLeft,
    BottomMiddle,
    BottomRight
}
```

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

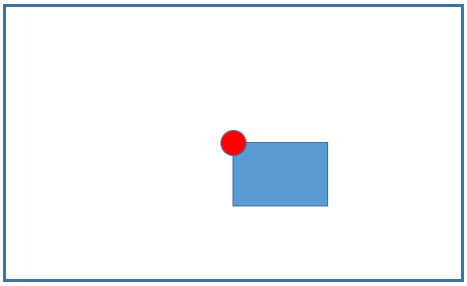


`OriginMode` basically says which part of your sprite its position refers to. In other words, if your x, y position is 100, 100, and your `OriginMode` is `Center`, the pixel in the center of your sprite will be drawn at position 100, 100 on the screen. But if your `OriginMode` is `TopLeft`, it means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

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. By the way, feel free to steal and re-use that amazing character artwork in your own game. You're welcome.

Making a spritesheet

As we've seen, we need a special texture called a spritesheet to make an animated sprite. Spritesheets contain frames of animation. Our explosion animation was originally created in an image editor by drawing it one frame at a time, with each frame saved to 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 need to do is put the images for each frame together into one image file, and keep track of the location and size of each frame (that is, its x and y coordinates within the spritesheet and its width and height). This information will let us specify the `IntRect` values for our animation frames.

There are many tools you could use to make spritesheets. Some options include Piskel (https://www.piskelapp.com/) and LibreSprite (https://libresprite.github.io/). Or you can make them by hand in your image editor.

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`!

Create a new file named Explosion.cs:

Explosion.cs

```
using GameEngine;
using SFML.Graphics;
using SFML.System;
using System.Collections.Generic;

namespace MyGame
{
    class Explosion : AnimatedSprite
    {
        public Explosion(Vector2f pos) : base(pos)
        {
            Texture = Game.GetTexture("Resources/explosion-spritesheet.png");
            SetUpExplosionAnimation();
            PlayAnimation("explosion", AnimationMode.OnceForwards);
        }

        public override void Update(Time elapsed)
        {
            base.Update(elapsed);

            if (!IsPlaying())
            {
                MakeDead();
            }
        }

        private void SetUpExplosionAnimation()
        {
            var frames = new List<IntRect>
            {
                new IntRect( 0, 0, 64, 64),     // Frame 1
```

```
                new IntRect( 64, 0, 64, 64),    // Frame 2
                new IntRect(128, 0, 64, 64),    // Frame 3
                new IntRect(192, 0, 64, 64),    // Frame 4
                new IntRect(256, 0, 64, 64),    // Frame 5
                new IntRect(320, 0, 64, 64),    // Frame 6
                new IntRect(384, 0, 64, 64),    // Frame 7
                new IntRect(448, 0, 64, 64),    // Frame 8
                new IntRect(512, 0, 64, 64)     // Frame 9
            };

            AddAnimation("explosion", frames);
        }
    }
}
```

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, we create a `List` of `IntRect` to store the frame data, adding one frame at a time:

```
var frames = new List<IntRect>
{
    new IntRect( 0, 0, 64, 64),     // Frame 1
    new IntRect( 64, 0, 64, 64),    // Frame 2
    new IntRect(128, 0, 64, 64),    // Frame 3
    new IntRect(192, 0, 64, 64),    // Frame 4
    new IntRect(256, 0, 64, 64),    // Frame 5
    new IntRect(320, 0, 64, 64),    // Frame 6
    new IntRect(384, 0, 64, 64),    // Frame 7
    new IntRect(448, 0, 64, 64),    // Frame 8
    new IntRect(512, 0, 64, 64)     // Frame 9
};
```

Each IntRect contains the x and y position of the frame within the spritesheet, as well as the frame's width and height.

After creating the frames, we add the animation for them:

```
AddAnimation("explosion", frames);
```

Now the animation is ready to play!

The `Update` method is interesting, too:

```
public override void Update(Time elapsed)
{
    base.Update(elapsed);
```

```
            if (!IsPlaying())
            {
                MakeDead();
            }
        }
```

First, we make sure that our parent class (`AnimatedSprite`) has its own `Update` method invoked. If we don't do this, our sprite won't animate or be drawn to the screen, since AnimatedSprite.Update has all the code for doing that. We then check to see if we're still playing, and if we're not, we remove ourselves from the scene. We do this because we don't want the explosion game object to hang around after it's done exploding. `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.

OK, now that we have an explosion, it's time to modify our `Meteor` class to explode when it's hit by a laser. Think you can figure out how to do this on your own? Give it a shot before checking out the code on the next page!

OK, here is how we can update our `Meteor` class to play the explosion animation. Our changes are in the `HandleCollision` method:

Meteor.cs

```
// omitted code

namespace MyGame
{
    class Meteor : GameObject
    {
        // omitted code

        public override void HandleCollision(GameObject otherGameObject)
        {
            if (otherGameObject.HasTag("laser"))
            {
                otherGameObject.MakeDead();
            }

            MakeDead();

            Vector2f pos = _sprite.Position;
            pos.X = pos.X + _sprite.GetGlobalBounds().Width / 2.0f;
            pos.Y = pos.Y + _sprite.GetGlobalBounds().Height / 2.0f;

            Explosion explosion = new Explosion(pos);
            Game.CurrentScene.AddGameObject(explosion);
        }

        // omitted code

    }
}
```

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. 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 it should be in your **Resources** folder. Remember, you can add other sounds to your **Resources** folder too.

Explosion.cs

```
using GameEngine;
using SFML.Audio;
using SFML.Graphics;
using SFML.System;
using System.Collections.Generic;

namespace MyGame
{
    class Explosion : AnimatedSprite
    {
        private readonly Sound _boom= new Sound();

        public Explosion(Vector2f pos) : base(pos)
        {
            Texture = Game.GetTexture("Resources/explosion-spritesheet.png");
            SetUpExplosionAnimation();
            PlayAnimation("explosion", AnimationMode.OnceForwards);

            _boom.SoundBuffer = Game.GetSoundBuffer("Resources/boom.wav");
            _boom.Play();
        }

        // omitted code
```

```
    }
}
```

Here we have added a private field called `_boom`, of type `Sound`. When our `Explosion` instance is constructed, we set up our sound. We load the audio data via `Game.GetSoundBuffer`, and provide this data to `_boom` via its `SoundBuffer` property.

Once the `Sound` has its `SoundBuffer`, you can play it at any time with the `Play` method, 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 have a well-deserved reputation for being something of a gadget freak, and am rarely happier than when spending an entire day programming my computer to perform automatically a task that 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:

GameScene.cpp

```
using GameEngine;

namespace MyGame
{
    public class GameScene : Scene
    {
```

```
        private int _score = 0;

        // omitted code

        // Get the current score
        public int GetScore()
        {
            return _score;
        }

        // Increase the score
        public void IncreaseScore()
        {
            ++_score;
        }
    }
}
```

Here we have added a private integer variable to track the score, and two functions to get and increase it. `GetScore` 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`. We'll display text using the font Courneuf-Regular.ttf, which can be found in your Resources folder.

Create the Score class

First, add a new file to your project, Score.cs:

Score.cs

```
using GameEngine;
using SFML.Graphics;
using SFML.System;

namespace MyGame
{
    class Score : GameObject
    {
        private readonly Text _text = new Text();

        public Score(Vector2f pos)
        {
            _text.Font = Game.GetFont("Resources/Courneuf-Regular.ttf");
            _text.Position = pos;
            _text.CharacterSize = 24;
            _text.FillColor = Color.White;
```

```
            AssignTag("score");
        }

        public override void Draw()
        {
            Game.RenderWindow.Draw(_text);
        }

        public override void Update(Time elapsed)
        {
            GameScene scene = (GameScene)Game.CurrentScene;
            _text.DisplayedString = "Score: " + scene.GetScore();
        }
    }
}
```

`Score` is, not surprisingly, a `GameObject`. Its constructor takes a `Vector2f` so we can position it where we'd like on the screen. It's a lot like other `GameObject` classes we've seen before. 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 take a look at the constructor code:

```
_text.Font = Game.GetFont("Resources/Courneuf-Regular.ttf");
_text.Position = pos;
_text.CharacterSize = 24;
_text.FillColor = Color.White;
AssignTag("score");
```

Here we assign the `Font` property of our `Text` instance. You may notice a pattern here… It's very similar to how we set the `Buffer` of our `Sound` instances and `Texture` of 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` method, we put the text on the screen by using the `Draw` method of the `RenderWindow`, exactly as we would with a `Sprite`:

```
Game.RenderWindow.Draw(_text);
```

Let's check out the `Update` method. First, we're taking the reference to a `Scene` that is returned by `Game.GetCurrentScene` and casting it to type `GameScene`:

```
GameScene scene = (GameScene)Game.CurrentScene;
```

Casting is a way of changing one type into another type, assuming the cast is possible. For example, if you want to call a method which requires a `float` but all you have is an `int`, you could cast it: `(float)myIntValue`. This works because converting an `int` to a float is possible. You couldn't, for example, cast a `string` to an `int`. There is no defined conversion for this. You can also use casting to convert between compatible reference types, such as casting a base type into a derived type, which is exactly what we're doing here when we cast `Game.CurrentScene` to `GameScene`. The `CurrentScene` property returns type `Scene` (the base type), but we know that the scene is actually a `GameScene` (the derived type). If `CurrentScene` was just a `Scene`, or some other

subclass of **Scene**, this would fail. So 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. It expects to be in a **GameScene** so it can call score-related methods. That's OK for now, but it does limit the ways in which we can use **Score**, and it may be something we need to refactor as we expand upon our game in the future.

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

```
_text.DisplayedString = "Score: " + scene.GetScore();
```

Here you can see why we had to cast the scene to **GameScene**. Our **GameScene** class has a **GetScore** method, but the **Scene** class does not. If we didn't cast, the compiler would look for a **GetScore** method in **Scene**, and when it didn't find one compilation would fail.

The **DisplayedString** property of **_text** allows us to specify the actual text that gets drawn to the screen. Here we are *concatenating* (combining one after the other) two things: the string **"Score: "** and the integer returned by **scene.GetScore()**. The reason we can combine an **int** and a **string** is because the string concatenation operator (**+**) implicitly converts from **int** (and other types) to **string**. Like casting, this is a form of type conversion.

Put Score into the scene

Now that we've got our **Score** game object, we need to add it to **GameScene**:

GameScene.cs

```
using GameEngine;
using SFML.System;

namespace MyGame
{
    public class GameScene : Scene
    {
        private int _score = 0;

        public GameScene()
        {
            Ship ship = new Ship();
            AddGameObject(ship);

            MeteorSpawner meteorSpawner = new MeteorSpawner();
            AddGameObject(meteorSpawner);

            Score score = new Score(new Vector2f(10.0f, 10.0f));
            AddGameObject(score);
        }

        // omitted code

    }
}
```

If we run our game now, we will see a score on the screen, but it doesn't increase when we destroy a meteor. Can you think of how to wire up the meteor getting shot with the score? Try to figure this out on your own before checking out the code on the next page.

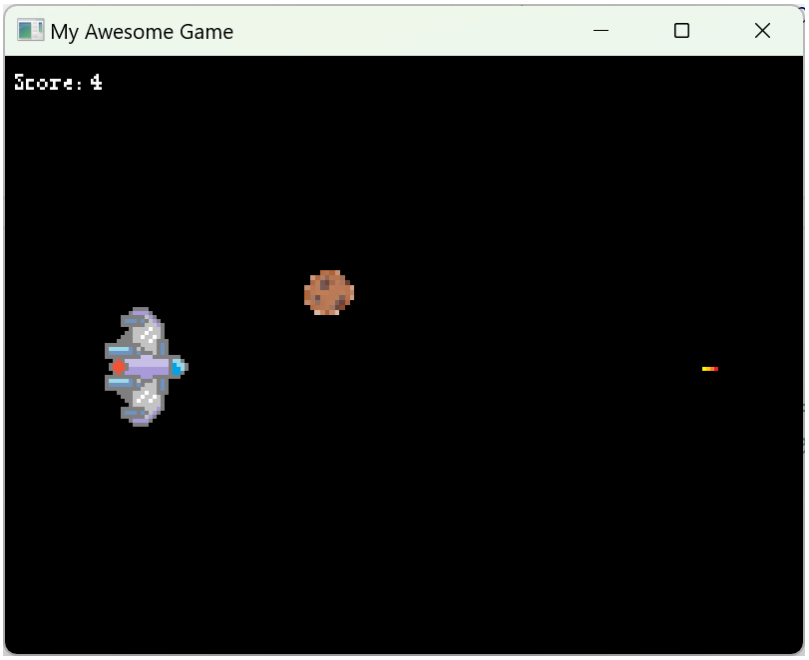
OK, here is the code you would add to your Meteor class to increase the score:

Meteor.cs

```
// omitted code

namespace MyGame
{
    class Meteor : GameObject
    {

        // omitted code

        public override void HandleCollision(GameObject otherGameObject)
        {
            if (otherGameObject.HasTag("laser"))
            {
                otherGameObject.MakeDead();
                GameScene scene = (GameScene)Game.CurrentScene;
                scene.IncreaseScore();
            }

            MakeDead();

            Vector2f pos = _sprite.Position;
            pos.X = pos.X + _sprite.GetGlobalBounds().Width / 2.0f;
            pos.Y = pos.Y + _sprite.GetGlobalBounds().Height / 2.0f;

            Explosion explosion = new Explosion(pos);
            Game.CurrentScene.AddGameObject(explosion);
        }

        // omitted code

    }
}
```

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.



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 a new class to your project named **GameOverMessage.cs**. Here's the source:

GameOverMessage.cs

```
using GameEngine;
using SFML.Graphics;
using SFML.System;
using SFML.Window;

namespace MyGame
{
    class GameOverMessage : GameObject
    {
        private readonly Text _text = new Text();

        public GameOverMessage(int score)
        {
            _text.Font=Game.GetFont("Resources/Courneuf-Regular.ttf");
            _text.Position = new Vector2f(50.0f, 50.0f);
            _text.CharacterSize = 48;
            _text.FillColor = Color.Red;
            _text.DisplayedString = "GAME OVER\n\nYOUR SCORE:" + score + "\n\nPRESS ENTER TO CONTINUE";
        }

        public override void Draw()
        {
            Game.RenderWindow.Draw(_text);
        }

        public override void Update(Time elapsed)
        {
            if(Keyboard.IsKeyPressed(Keyboard.Key.Enter))
            {
```

```
                GameScene scene = new GameScene();
                Game.SetScene(scene);
            }
        }
    }
}
```

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` method 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 know, 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 scene 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.cs

```
using GameEngine;

namespace MyGame
{
    class GameOverScene :Scene
    {
        public GameOverScene(int score)
        {
            GameOverMessage gameOverMessage= new 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.cs

```
// omitted code

namespace MyGame
{
    public class GameScene : Scene
    {
        private int _score = 0;
        private int _lives = 3;

        // omitted code

        // Get the number of lives
        public int GetLives()
        {
            return _lives;
        }

        // Decrease the number of lives
        public void DecreaseLives()
        {
            --_lives;

            if(_lives == 0)
            {
                GameOverScene gameOverScene = new GameOverScene(_score);
                Game.SetScene(gameOverScene);
            }
        }
    }
}
```

We've seen encapsulation a few times now, including back in Part 9 where we encapsulated `_score` with `GetScore` and `IncreaseScore` methods (instead of just having a public field anyone could modify). Well, here we have encapsulation again, and you can start to see some of its real power. 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, we'd have to find another, less elegant way, like checking the remaining lives during `Update`, or having the `Meteor` class check remaining lives itself.

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.cs

```
using GameEngine;
using SFML.Graphics;
```

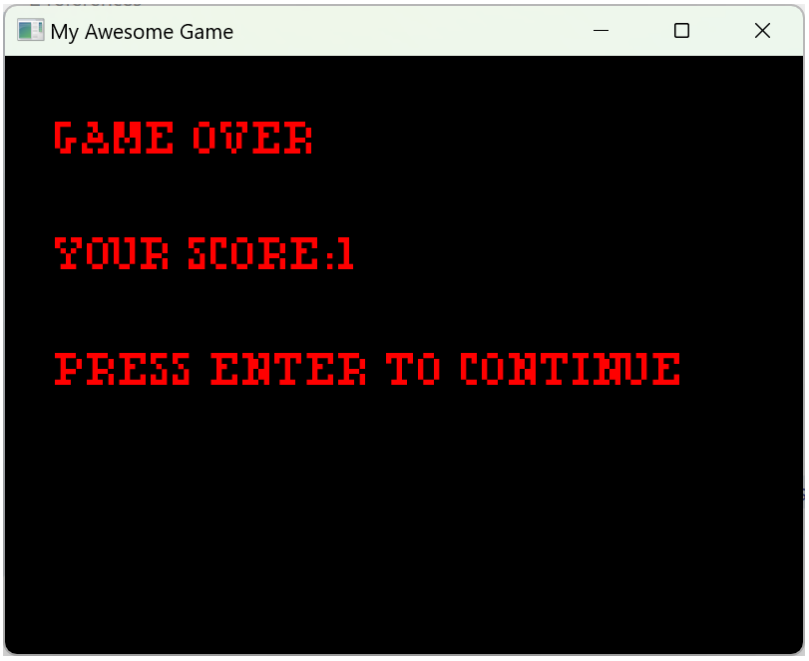
```
// omitted code

namespace MyGame
{
    class Meteor : GameObject
    {

        // omitted code

        public override void Update(Time elapsed)
        {
            int msElapsed = elapsed.AsMilliseconds();
            Vector2f pos = _sprite.Position;

            if(pos.X < _sprite.GetGlobalBounds().Width * -1)
            {
                GameScene scene = (GameScene)Game.CurrentScene;
                scene.DecreaseLives();

                MakeDead();
            }
            else
            {
                _sprite.Position = new Vector2f(pos.X - Speed * msElapsed, pos.Y);
            }
        }
    }
}
```

All we're doing here is adding a little extra logic in our `Update` method. 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!



There is a lot that you could add to your game! One quality-of-life feature that would be nice is some kind of indicator of how many lives the player has left. Maybe the player should lose a life when their ship collides with a meteor? It would be really cool if the game had some power-ups, maybe different weapons, or different things to shoot at. Maybe an alien that attacks you with its own lasers? Where do you want to take your game?

Reference

Access Modifier

Access modifiers in C# control how other parts of the code can access something. For example, if you have a private method in a class, it can only be used within that class itself. A protected method could be used by the class itself as well as subclasses. And a public method could be used by anyone. For more on access modifiers, see https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

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 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/bool 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, objects contain data plus related behavior, and classes define the "blueprints" for objects. Classes are used to create object "instances". In C#, this is done with the new operator:

```
GameScene myScene = new GameScene();
```

You can create any number of instances from a single class definition, unless that class is `static`. Static classes cannot be instantiated. Each class defines 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. For information on classes in C#, see https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/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. Constants are compiled into your application, and therefore can't be object instances. For object instances that don't change, you can use `readonly`. For more on C# constants, see https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constants

Constructor

A constructor is a special method 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 method is called. Constructor methods 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", we might say). For more on constructors, see https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors

Default values

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

```
// someInput is optional, not required when calling
void DoSomething(int someInput = 10);
```

For more on optional arguments in C#, see https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments

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 fields that represent its internal state. Those fields 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 publicly, 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 without 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).

Enum

Enum types declare a set of related, named constants. Variables which have a particular enum type can only have values from the set of constants defined on that enum type. For example, with the following enum, variables could be have the value `Red`, `Green` or `Blue`, but not `Orange`, because it doesn't exist on the enum:

```
enum Color
{
    Red,
    Green,
    Blue
```

```
}
```

Before enums, code might use “magic numbers” or strings, for example making 1 mean “red” or just using the string “red” itself. Enums make code less error-prone and more maintainable. The compiler can verify that you are using a constant defined by the enum.

To learn more about enums in C#, see: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/enums

Fields

In C#, a field is a variable that is declared as part of a class or struct. Along with methods, fields are *members* of their containing type. For more on fields, see https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/fields

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: https://www.sfml-dev.org/tutorials/3.0/graphics/text/

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.

Generics

Generics provide a means to have *type parameters* as part of a class or method. By using a type parameter, you can swap out types, for example by having a `List` which can only contain integers (`List<int>`) or one that contains strings (`List<string>`). The declaration of a generic list class would look like `class List<T>` where `T` is the type parameter. In the `List<T>` implementation, `T` can be used to refer to the type of whatever the list contains, which is specified when you declare your `List<int>` or `List<string>` or `List<Whatever>`. So for example the `List<T>` implementation might have an `Add(T item)` method. Without generics, you could put any kind of object into a `List`, which could cause problems if a type that you don’t expect gets in. You would also need to know what type of objects the `List` contains, meaning you would have to cast them to the correct type when you want to use them. Also, the compiler can’t verify the correctness of your code if it doesn’t know what types are being used. Generics provide a lot of safety and reduce bugs in your code. Most of the collection types in C# have a generic version which takes one or more type parameters. But you will see as you go along that generics are used in places other than collections, too.

For more, see: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics

Main method

In C#, the `Main` method is the entry point into your program. An application can only have a single `Main` method. For more on the `Main` method and how it can be specified, see https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/main-command-line

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, and other types, 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.

Property

Properties in C# provide classes with a means of exposing data that is as convenenient to use as fields but is safer, more flexible and more powerful. Here is a simple example:

```
private int _score;

public int Score
{
    get { return _score; }
    set { _score = value; }
}
```

Here we have a property Score, which is backed by a private integer field. Callers to this class could get and set the score as though it was a field of your class:

```
instance.Score = 10;
scoreCopy = instance.Score;
```

Properties provide `get` and `set` accessors which are invoked when getting or setting their value, respectively. You can see in the example above that for setters, the incoming value is provided in a special variable called `value`.

Property accessors can be used to validate data being set, something that a field can’t do. For example, here’s how we could validate that Score is between 0 and 100:

```
public int Score
{
    get { return _score2; }

    set {
        if (value < 0 || value > 100)
        {
            throw new ArgumentOutOfRangeException("Score must be between 0 and 100");
        }

        _score = value;
    }
}
```

Simple properties can also be automatically implemented, without the need to specify the backing field:

```
public int Score { get; set; }
```

Properties don’t need to be backed by a private field at all. They could do all kinds of stuff behind the scenes, like read and write the value from a database or file.

You can also make properties read-only and prevent anyone outside your class from setting them:

```
public int Score { get; private set; }
```

Properties are great because they provide encapsulation, protect access to private instance data, and can be changed over time without callers having to know. They also allow for custom logic, calculating values, or doing other work that you couldn't do if you just exposed the data as a field.

To learn more about properties in C#, see: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

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. A common way is to use the system time. You seed the random number genrator with an `int`, and you can get the system time as an int via `(int)DateTime.Now.Ticks`. `DateTime` is part of the `System` namespace. The `DateTime.Now.Ticks` method returns the number of 100-nanosecond intervals since January 1, 0001. This makes a good seed because each time you run the program, `DateTime.Now.Ticks` will be different. No two runs of your game will likely ever have the exact same sequence of pseudorandom numbers.

For more on `System.Random` and generating random numbers in C#, see https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-random

Readonly

In C#, `readonly` is typically used to indicate that a field cannot have its value set after assignment (either when it's declared or in a constructor). This helps to reduce bugs my reducing the number of places which could make changes to data. This is part of a practice called *immutability*. For more on immutability and why it produces more bug-free code, see https://medium.com/@shoebsd31/best-practices-for-achieving-immutability-in-c-e0375b11f77d. For more on the `readonly` keyword in C#, see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly

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: https://www.sfml-dev.org/documentation/3.0.0/classsf_1_1Rect.html

Reference

The two main categories of type in C# are **value types** and **reference types**. All variables in your running program, regardless of their type, exist somewhere in memory. With value types, that spot in memory contains the value itself. For example, the memory for an `int` variable might contain the value `42`. Types like `int`, `float` and `char` are value types, as are `struct` types like `Vector2`. With reference types, the spot they occupy in memory doesn't contain a value, but instead contains a reference to something else, elsewhere in memory. References essentially *point to* something somewhere in memory. Class instances and arrays are examples of reference types.

The default in C# is that arguments are passed to methods by value. Passing by value means a copy of the value is sent, and this copy is different in both the calling method and the called method. Let's consider the following code:

```
private void CallingMethod()
{
    int a = 10;
    CalledMethod(a);
    Console.Out.WriteLine(a);
}

private void CalledMethod(int a)
{
    a++;
}
```

Take a moment to think about this code. What does it print to the screen? `CallingMethod` creates an integer named `a` and then invokes `CalledMethod`, passing it `a` as a parameter. Because this parameter is passed by value, `CallingMethod` and `CalledMethod` have two separate copies of it. In other words, the `a` inside `CallingMethod` and the `a` inside `CalledMethod` are totally different and occupy two different spots in memory. When `CalledMethod` increases the value of its own `a` by 1, it has no effect on the `a` inside `CallingMethod`. Therefore, when `CallingMethod` prints the value of its own `a` to the screen, the output will be `10`.

Reference types are also passed by value, but the value in this case is the reference itself. This means that the calling method and the called method will each have their own copy of the reference. The difference however is that both copies of the reference *point to* the same thing in memory. Both methods can modify the thing being pointed to (unless something prevents modification, such as the use of readonly). Some more code should help make this clear:

```
private static void CallingMethod()
{
    int[] a = new int[1] {10};
    CalledMethod(a);
    Console.Out.WriteLine(a[0]);
}

private static void CalledMethod(int[] a)
{
    a[0]++;
}
```

Here we create array of integers containing a single element, initialized to the value `10`. When `CallingMethod` invokes `CalledMethod`, a copy of the reference to this array is made, but both references point to the same data in memory. This means that the `a` inside of `CalledMethod` is not the same as the `a` inside of `CallingMethod`, but they both reference the same array in memory. Therefore, when `CalledMethod` modifies that array, it's modifying the same data that `CalledMethod` later prints to the screen, and the output is `11`.

For more on value types, see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types

For more on reference types, see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types

For more on passing by value and passing by reference, see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters

RenderWindow

A class in SFML that provides a window for 2D drawing. See the SFML documentation for more: https://www.sfml-dev.org/documentation/3.0.0/classsf_1_1RenderWindow.html

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/.

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 here: https://www.sfml-dev.org/tutorials/3.0/audio/sounds/

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 here: https://www.sfml-dev.org/tutorials/3.0/audio/sounds/

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: https://www.sfml-dev.org/tutorials/3.0/graphics/sprite/

Static Class

In C#, a static class is just a regular class except that it cannot be instantiated. This means you can't use the **`new`** operator to create an instance of the class. Therefore, it has no instance variables. All of the members in a static class, in other words all of its fields and methods, must also be static. Static classes are typically utility classes or classes used for starting up an application, but they have other uses as well. For more, see: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

String

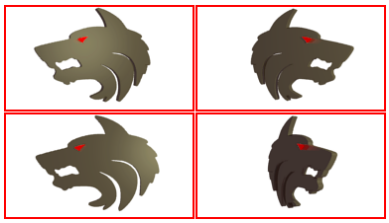
A string is a sequence of characters. String variables in C# are used to store text. Any text your game needs, such as player names, user interface text, dialog and so on can be represented by a string. For more on strings, see: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/

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: https://www.sfml-dev.org/tutorials/3.0/graphics/text/

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: https://www.sfml-dev.org/tutorials/3.0/graphics/sprite/

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: https://www.sfml-dev.org/tutorials/3.0/system/time/

Using directive

A `using` directive allows you to import types from a namespace, which lets you use those types in your code using their simple names. For example, the directive `Using System;` allows you to use the types in the `System` namespace, such as `Random`, without having to specify them fully, e.g. without having to specify `System.Random` every time you want to use the class to generate a random number. Using directives help make code cleaner and less verbose. For more, see: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive

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: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive

Void

In C#, `void` is used to indicate that a method does not return a value. For more, see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/void