A Shooter Game while Learning
Creating a Basic Bullet Class using Object-Oriented Programming

By Abraham Morales

Contents

- Introduction.
- Preparing our Project.
- The Bullet Class.
- Trying it out!
- Wrapping Up!
- Conclusion.

Introduction

Why is there so much hype for **Object-Oriented Programming**? And why is it so important? People have been programming functionally for so long, and I love my thousand lines of code, why change?

Object-Oriented Programming, or **OOP** if you prefer seems like a complicated concept to grasp, especially if you are just beginning programming, however it is pretty easy, anyone can learn it and it will make your programs easier to understand, you'll do less coding, debugging won't be such a chore and you'll have more time to play games.

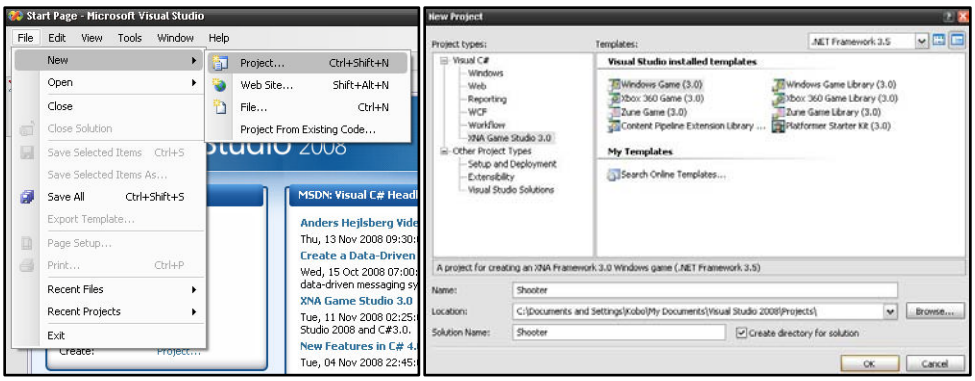
This tutorial is aimed at novice programmers, but intermediate and even professional programmers might find it useful. It's aimed at those who work with C# and the XNA Framework but some of these concepts are used on almost all Object-Oriented Languages.

By the end of this article you will create the bare bones of what could be a great shooter game.

Before coding a class, instantiating objects, encapsulating variables, and overloading constructors, (relax, you'll understand these concepts by the end of this article), let's begin by creating a new game project in **XNA Game Studio 3.0**, be that in **Visual C# Express 2008** or **Visual Studio 2008**:

1. Select from the menu, **File** > **New Project**.
2. In the **New Project** dialog window, select **XNA Game Studio 3.0** as the project type and **Windows Game (3.0)** as your template.
3. In the **Name** text box type *Shooter*, then click **Ok**.

Abraham Morales
http://kobochan.org
kurokasai@msn.com



Preparing our Project

Once it's loaded, let's insert the following lines of code immediately after `GraphicsDeviceManager` and `SpriteBatch`:

```
Texture2D shipTexture;
Vector2 shipPosition;
```

For the `shipTexture` variable, I'll use an image from **Squirrelsquid**'s gallery which most of its content is royalty free, visit his gallery at http://pixeljoint.com/p/1889.htm.

Our `Initialize()` method will look like this:

```
protected override void Initialize()
{
      graphics.PreferredBackBufferWidth = 450;
      graphics.PreferredBackBufferHeight = 480;
      Window.Title = "Easy OOP - Shooter";
      graphics.ApplyChanges();
      base.Initialize();
}
```

This will create a small window with *text* in the **Title Bar**.

Our `LoadContent()` method will be this:

```
protected override void LoadContent()
{
      // Create a new SpriteBatch, which can be used to draw textures.
      spriteBatch = new SpriteBatch(GraphicsDevice);
      shipTexture = Content.Load<Texture2D>("ship");
      shipPosition = new Vector2(Window.ClientBounds.Width / 2 -
shipTexture.Width / 2, Window.ClientBounds.Height / 2 - shipTexture.Height /
2);
}
```

We load our ship texture into `shipTexture` and we initialize `shipPosition` in the center of the screen using `shipTexture`'s dimensions.

Next will be our `Update(GameTime gameTime)` method:

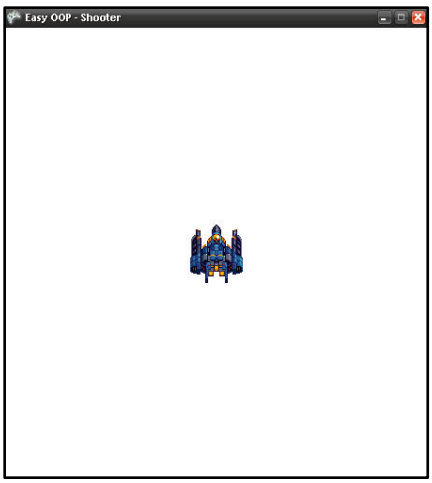
```
protected override void Update(GameTime gameTime)
{
	KeyboardState keyboard = Keyboard.GetState();
	if (Keyboard.GetState().IsKeyDown(Keys.Escape))
		this.Exit();
	if (keyboard.IsKeyDown(Keys.Left))
		shipPosition.X -= 5;
	if (keyboard.IsKeyDown(Keys.Right))
		shipPosition.X += 5;
	if (keyboard.IsKeyDown(Keys.Up))
		shipPosition.Y -= 5;
	if (keyboard.IsKeyDown(Keys.Down))
		shipPosition.Y += 5;
	shipPosition.X = MathHelper.Clamp(shipPosition.X, 0,
Window.ClientBounds.Width - shipTexture.Width);
	shipPosition.Y = MathHelper.Clamp(shipPosition.Y, 0,
Window.ClientBounds.Height - shipTexture.Height);
	base.Update(gameTime);
}
```

We restrict the ship's movement to be within the window's borders using the `MathHelper.Clamp`. By the way, for simplicity's sake, we are only using the **Keyboard** for **input**.

And finally, our `Draw(GameTime gameTime)` method will look like this:

```
protected override void Draw(GameTime gameTime)
{
	GraphicsDevice.Clear(Color.White);
	spriteBatch.Begin();
	spriteBatch.Draw(shipTexture, shipPosition, Color.White);
	spriteBatch.End();
	base.Draw(gameTime);
}
```

Press **F5** to run your project, the output window will look like this:



The Bullet Class

Now our shooter obviously isn't very much fun without shooting. I've seen literally dozens of ways to program bullets and by far the **Object-Oriented** method is the easiest and if appropriately handled the most efficient.

Before doing so, let's look into some basic OOP concepts by ordering a pizza. Bear with the analogy for now. There are a lot of ways for you to order a pizza, you can call the pizzeria by phone, go in person, or even use the interwebs, regardless of how you order it you'll get a pizza as long as you pay for it or you aren't black listed. Basically you don't care about the pizzeria's oven temperature, nor at how many degrees do they preheat it or even the process for making the dough, you just want your pizza because you are hungry, you pretty much **abstract** away the details on the pizzeria's ways of doing things, not to mention that not all pizzerias follow the same process for making a pizza, and there are plenty of pizza franchises to choose from.

Abstraction and the latter concept, defined as **classification**, are key concepts in OOP. Once you call the pizzeria, you can specify the size, the ingredients or you simply choose one from a menu, after 30 minutes (sometimes more) they'll deliver the pizza right to your home and you can do whatever it is you do with pizza solving your hunger problem.

Let's translate this into OOP lingo. One of the neat things of OOP, amongst others, is that you recycle code, hence you are programming for the programmer (which might be you). You solve a problem in the most general way possible so you can use that solution again and again. Our problem here is that we are hungry for pizza, so from our class pizzeria, we create an order; in OOP we call that **instantiating** an **object**.

An **object** is pretty much anything you can **classify**, specifically in this case, we'll call it *myPizza*, because well, it's mine, other pizzas made in the pizzeria will have different names, yet they are all different *objects* or **instances** from the same **class**, however, they all share the same **attributes** that make them, well, pizzas.

Going back to our Bullet Class, we begin by adding a new class to our project:

1. Go to your **Solution Explorer**.

2. **Right click** the *Shooter* project.
3. **Add** > **Class**.
4. In the **Add New Item** dialog, type *Bullet.cs* in the **Name** text box.
5. Click the **Add** button.

The Black Box

Before coding, let's back it up a little bit and plan this properly, we do this by analyzing the attributes a bullet has in our game. *What is a bullet? Where will it be drawn? How will it move? What speed will it have? What texture will it use?*

Bullet
Attributes
Texture2D texture
Vector2 position
Vector2 speed
Methods
Update()
Draw()

As we answer the above questions we start understanding how to design our class in C#. Some programmers like to call the above table a **Black Box** (it's almost always not black). Based on the table, we begin coding.

Attributes and Properties

When we create a new class we get the basic using statements, typically, we don't need to add more, it depends on the problem you wish to approach. In this case though, since we are using the XNA Framework, we need to add the following statements:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

Since we want our bullets to be *drawn* on screen and since we're going to need `Vector2` for the bullet's position and speed, we need both statements.

So right between our *class' brackets*, let's declare our **attributes**:

```
private Texture2D texture;
private Vector2 position = Vector2.Zero;
private Vector2 speed = new Vector2(0, 450);
```

Notice how I added the word `private` before the data type? It means that the variable declared as `private` can only be handled within the class, you can omit the word and it'll still be `private` though. I also initialized the variables just in case the user forgets to input them.

In order to get that user input for our bullet class, we could just change `private` to `public`, however that might get mixed up with other variables in our Game1.cs, or maybe other classes, or I just might change them by accident. To solve this, we'll need to resort to **encapsulation**.

Encapsulation is the process of keeping data and/or methods together inside objects which can be modified indirectly and safely, we also get to keep them `private`, just like medicine capsules keep the good stuff inside those plastic-like shells.

We'll encapsulate using something called properties, right after our attributes, add these lines:

```
public Vector2 Position
{
      get { return position; }
      set { position = value; }
}

public Vector2 Speed
{
      get { return speed; }
      set { speed = value; }
}
```

A property can be obtained and can be changed, so that is why we have a `get` and `set`. Let's look at the first one, `public Vector2 Position`, `get` will `return` the value in the variable `position`, while `set` will change the variable's `value` to what the user (not the one playing the game) inputs. It is important that these properties are `public` so we can read and change them from anywhere within our `namespace`. Just so you know, you can make a property **read-only** by simply omitting the `set`.

The Constructor

Remember that a pizzeria has someone to take your order? Well, in OOP you need a constructor to create an object or better said an instance from the class. The *object* we want from our *Bullet Class* will obviously be a *bullet shot from the ship*.

Next up is constructing the constructor:

```
public Bullet()
{
}
```

That was easy. It's a must for the constructor to have the same name as the class, this way, the compiler knows what it *instantiates*, also notice that the constructor doesn't have a `return` type, this is because it is not a method, this will become clear once you see it in action. Another thing is the word `public`; we need our class to be visible from anywhere within the `namespace`, especially from our *Game1.cs* so our ship is able to *create* or better said, *shoot* bullets.

We are still missing two methods, but basically we created a **general structure** for one bullet, imagine the solution like: *Every time the player presses the spacebar, a bullet instance will be created and it will move vertically at 450px per second*. This means that we will have a lot of bullet objects shooting upwards in our screen, each of them different from the other one; we will need to manage them later on.

Our constructor creates the object, but it'll be better if we could send some *arguments* from our *Game1.cs*. We need a texture for our bullets and we'll load it from our `LoadContent()`method in

Game1.cs. When we prepared our project, we added the ship's texture and position, under those declarations let's add the following line so we don't forget to do it later.

```
Texture2D bulletTexture;
```

And in our `LoadContent()` method let's add:

```
bulletTexture = Content.Load<Texture2D>("laser");
```

Obviously we need to load our asset into our Content folder and before that I created a very simple laser texture:

Since we want this texture for every *object* from the *Bullet Class* we create, we're going to have to change the constructor:

```
public Bullet(Texture2D texture)
{
      this.texture = texture;
}
```

What we have between parentheses are the arguments we're going to send when we create an *instance*, you'll see it in action later on. Another thing, since we are using an argument which has the same name as a variable we have in our class, `texture`, we need to point to the class' variable using `this`. Now `this` is great because it *refers* to the object that was *instantiated*, in this case, what it does is that it passes the argument that we send from *Game1.cs* and saves it in the `texture` variable we have in our *Bullet Class*, and this is going to be done for every bullet object in our game. The argument we will send will be `bulletTexture`, but if you were to have different bullet types, you can just add another attribute for maybe damage and change the texture you want to use for that type of bullet, the wonders of OOP.

Update and Draw

To finish our class, we'll need two methods that are called during `Update(GameTime gameTime)` and `Draw(GameTime gameTime)` in our *Game1.cs*, the former to move our bullet across the screen and the latter to draw the bullet.

Both methods have to be public, let's begin with our `Update()` and put it right under the constructor:

```
public void Update(GameTime gameTime)
{
      position -= speed * gameTime.ElapsedGameTime.TotalSeconds;
}
```

As an argument, we'll send `gameTime`. This method will update the bullet's position using the speed and making sure it runs the same on almost all frame rates (good for slow processors).

Next up is our `Draw()` method:

```
public void Draw(SpriteBatch spriteBatch)
{
      //spriteBatch.Begin();
      spriteBatch.Draw(texture, position, Color.White);
      //spriteBatch.End();
}
```

In order to draw our bullet we need to send `spriteBatch` as argument, notice that I commented the `spriteBatch.Begin()` and `spriteBatch.End()` because you might want to draw the bullet between a `Begin()` and `End()` that's already in *Game1.cs*.

Let's try it out!

Now that we finished our class, let's go back to our *Game1.cs* and see if it works.

The first step would be to declare the object, and then we'll declare a **Generic List** object so we can manage all the bullets that will be drawn on screen, a List is pretty much like an array, the difference is we can make it grow as long as we want and make it hold specific data-types. So right under our `bulletTexture` declaration, let's add:

```
Bullet bullet;
List<Bullet> bulletList = new List<Bullet>();
```

From here on, there are many ways you could experiment to get the results you might be looking for as for different bullet types or different keys to press for other projectiles, so I really encourage you to do so, we'll keep it simple here.

In our `Update(GameTime gameTime)` method, let's add a new condition for when the player presses the spacebar:

```
if (keyboard.IsKeyDown(Keys.Space))
{
      bullet = new Bullet(bulletTexture);
      bullet.Position = new Vector2(shipPosition.X + shipTexture.Width / 2 -
bulletTexture.Width / 2, shipPosition.Y - shipTexture.Height / 2);
      bulletList.Add(bullet);
}
for (int i = 0; i < bulletList.Count; i++)
      bulletList[i].Update(gameTime);
```

So every time the player presses the space bar, a new bullet will be created, using properties, we `set` the bullet's position at the top-center of the ship and then we add it to the `bulletList`, after that, the `for` statement checks how many bullets have been added to the list and updates the position of each one so they move across the screen, although this solves the problem it's not very memory efficient, the list just grows and grows and there's no end, we'll fix that later.

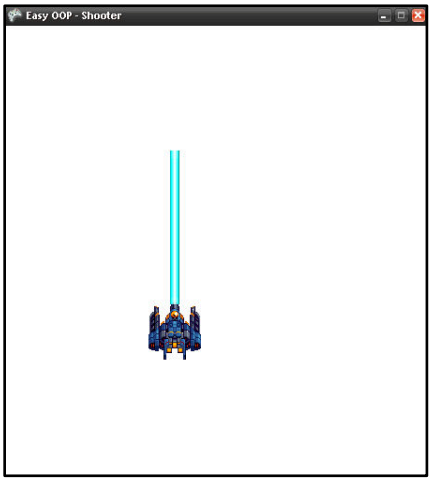
Let's change our `Draw(GameTime gameTime)` method by adding this between `spriteBatch.Begin()` and `End()`:

```
foreach (Bullet shipBullet in bulletList)
```

```
        shipBullet.Draw(spriteBatch);
```

The `foreach` statement will extract the objects in our `bulletList` and it will draw each one on screen. Did you notice the `Update` and `Draw` methods after `bulletList` and `shipBullet`? They are the same ones we coded in our *Bullet Class*!

Press **F5** to run your project, the output window will look like this:



Bullet Delay

Although it is a laser, I assume it wasn't what you were expecting, this is because there is no delay between the bullets created, and we still need to address the problem we have with our ever-growing `bulletList`.

Let's kill two birds with one shot then, in *Game1.cs* under `List<Bullet> bulletList = new List<Bullet>();` let's add:

```
int bulletDelay;
```

Let's change our *spacebar* condition in the `Update(GameTime gameTime)` method to:

```
if (keyboard.IsKeyDown(Keys.Space) &&
gameTime.TotalGameTime.TotalMilliseconds - bulletDelay > 200)
{
        bulletDelay = (int)gameTime.TotalGameTime.TotalMilliseconds;
        bullet = new Bullet(bulletTexture);
        bullet.Position = new Vector2(shipPosition.X + shipTexture.Width / 2 -
bulletTexture.Width / 2, shipPosition.Y - shipTexture.Height / 2);
        if (bulletList.Count <= 1)
                bulletList.Add(bullet);
}
```

This will separate each bullet by `200` milliseconds and resets the delay timer once a bullet was shot, another thing I was looking into is that since it's a small screen, I want only a couple of bullets shot, hence the `bulletList.Count <= 1` condition. Let's go ahead and add another condition right under the *spacebar*'s:

```
if (keyboard.IsKeyUp(Keys.Space))
      bulletDelay = 0;
```

This will reset the player's shooting ability when the *spacebar* is released, otherwise you shoot once and that's it, however, we still need to reset our `bulletList` to 0 objects so we can create new ones without wasting memory, for that, we need to change our `for` statement to this:

```
for (int i = 0; i < bulletList.Count; i++)
{
      bulletList[i].Update(gameTime);
      if (bulletList[i].Position.Y < -bulletTexture.Height)
            bulletList.RemoveAt(i);
}
```

Now you are able to shoot as long as no more than two bullets are on screen, and as soon as one bullet is off-screen you can shoot again, we make use of the `Position` property here.

Wrapping it up!

By now we have the basics of creating a shooter game, of course there is a lot more work to be involved but let's just finish up with the details for what we have.

The bullets look fine, but they surely take their time to get off-screen, we could change this attribute in the *Bullet Class*, but, let's do it from our *Game1.cs*, for this we'll make use of the *Speed* property in our *Bullet Class*.

In our spacebar condition, let's go under the `bullet` and `bullet.Position` statements and add:

```
bullet.Speed = new Vector2(0, 700);
```

And let's go ahead and change our `bulletDelay` condition from `> 200` to `> 400`.

Overloading methods and the constructor

An **overload** is simply another way you could use the same method or constructor by passing different arguments. One example would be the `spriteBatch.Draw()` method, you can give it either a `Vector2` position or a `Rectangle`.

Let's go to our *Bullet Class* and overload the `Draw(SpriteBatch spriteBatch)` method, for this, simply copy-paste the method right under it and change it like so:

```
public void Draw(SpriteBatch spriteBatch, Color color)
{
      //spriteBatch.Begin();
      spriteBatch.Draw(texture, position, color);
      //spriteBatch.End();
}
```

Now you can specify the color elsewhere, in our Game1.cs, let's change our `shipBullet.Draw` in our `foreach` within the `Draw(GameTime gameTime)` like so:

```
shipBullet.Draw(spriteBatch, Color.Red);
```

Try it out!

In our constructor, back in the *Bullet Class*, we can overload it so you send the `position` and `speed` as arguments instead of editing their respective properties, so again, copy-paste the constructor right under it and change it like so:

```
public Bullet(Texture2D texture, Vector2 position, Vector2 speed)
{
      this.texture = texture;
      this.position = position;
      this.position = speed;
}
```

Conclusion

I do believe the XNA community isn't just for professional programmers and independent developers, by creating this simple tutorial I wish more and more novice programmers and aspiring game developers get into the community and into the industry one step at a time without getting overwhelmed by complicated functions and shader languages, it takes time, which is spent in a lot of reading, practice and patience, so I do hope you found this useful. Happy coding!

Abraham Morales
http://kobochan.org
kurokasai@msn.com