# Project Increment 2

***Step 1: Move burger***

For this step, you're making the burger move in the game based on user input.

Note: It's of course crazy to control the burger with the mouse -- we should use the keyboard -- but because we haven't learned how to use keyboard input yet, we'll use the mouse for now, then change it before the game is finished.

1. Add code to the `Burger` `Update` method to move the burger based on the current mouse location -- in other words, to follow the mouse. The burger should always be completely within the game window and it should only follow the mouse if it has health > 0. Note that you can access the window width and window height in the `GameConstants` class to keep the burger in the window
2. Add code to the `Game1` `Update` method telling the burger to update itself using the current mouse state. You should be able to figure out what the two arguments should be on your own at this point in the course

When you run your game, you should be able to move the burger around with the mouse.

***Step 2: Have burger shoot***

For this step, you're making the burger shoot based on user input. There's a lot of work in this step because we need to add the projectile functionality as well

1. Add code to the `Game1` `LoadContent` method to load the textures for the `teddyBearProjectileSprite` and `frenchFriesSprite` variables
2. Add code to the `Game1` `GetProjectileSprite` method to return the appropriate sprite based on the provided projectile type
3. Add code to the `Burger` `Update` method to create a new projectile if the left mouse button is pressed (this should only happen if the burger's health is > 0). You'll have to call the `Projectile` constructor to create this new object, making sure you create a french fries projectile, using the static `Game1` `GetProjectileSprite` method to get the french fries sprite, using the burger `drawRectangle` and the `GameConstants` `FRENCH_FRIES_PROJECTILE_OFFSET` constant to calculate the location of the projectile (centered horizontally, offset from the center vertically), and using the `GameConstants` `FRENCH_FRIES_PROJECTILE_SPEED` constant to set the y velocity of the projectile so it's going up
4. Right after creating the new projectile, call the static `Game1` `AddProjectile` method to add the new projectile to the game
5. Add the following code to the `Game1` `AddProjectile` method:

   ```
   projectiles.Add(projectile);
   ```

6. Add code to the `Projectile` `Draw` method to have the projectile draw itself

When you run your game, you should be able to create projectiles using the left mouse button. You may need to move the burger around while you shoot to test this.

***Step 3: Have projectiles move***

As you can see from the previous step, the burger can now drop projectiles like land mines, which isn't really the behavior we're looking for. For this step, you're making the projectiles move after they're created.

1. Add code to the `Projectile` `Update` method to move the projectile based on its y velocity and the elapsed game time in milliseconds. Refer to the Programming Assignment 3 Evaluation video if you need to see the solution

When you run your game, projectiles you fire from the burger should move up.

***Step 4: Control burger firing rate***

At this point, it seems clear that we're going to need to control the burger firing rate rather than firing a projectile on every update while the left mouse button is pressed (which is 60 projectiles per second at a fixed time step). For this step, you're controlling the burger firing rate.

1. Change the if statement in the `Burger` `Update` method to create a new projectile if health is > 0, the left mouse button is pressed, and `canShoot` is `true`. Inside the body of the if statement, just before creating the new projectile, set `canShoot` to `false`. If you run your game at this point, you'll only be able to fire a single projectile.
2. Add code to the `Burger` `Update` method that checks if `canShoot` is `false` (it would be inefficient to do what comes next if `canShoot` is `true`). If `canShoot` is `false`, add the elapsed game time in milliseconds to the `elapsedCooldownTime`. Use the `elapsedCooldownTime` and the `GameConstants` `BURGER_COOLDOWN_MILLISECONDS` constant to determine whether or not it's time to re-enable shooting (take a look at deciding whether or not to advance to the next animation frame in Chapter 7 if you need help with the idea). If it is, set `canShoot` to `true` and set `elapsedCooldownTime` to 0. If you run the game now, you should see that the firing rate is controlled.
3. One more nice gameplay thing. At this point, the cooldown timer has to expire even if the player releases the left mouse button. It's nicer to let the player release the left mouse button to immediately set `canShoot` to `true`, yielding a different player tactic where the player repeatedly presses and releases the left mouse button as quickly as they can. This is actually easy to implement. Change the Boolean expression in the if statement that determines whether or not it's time to re-enable shooting to also evaluate to true if the left mouse button isn't currently pressed.

When you run your game, you should be able to hold the left mouse button to fire projectiles at a constant rate and also repeatedly press and release the left mouse button to get a different (faster) firing rate.

***Step 5: Deactivate projectiles that have left the screen***

You may not have realized it yet, but every projectile we fire stays in the game world forever, even after it leaves the game window. This is really bad because we waste CPU time updating all those projectiles even though they're really out of the game. For this step, you're deactivating projectiles that have left the window (we'll actually remove inactive projectiles from the game in a later increment).

1. Add code to the `Projectile` `Update` method that sets the `active` field to `false` if the projectile has left the game window. You'll need to use the `GameConstants` `WINDOW_HEIGHT` constant to determine whether or not the projectile has gone past the top or bottom of the window. You need to check both the top and the bottom of the window because french fries go up and teddy bear projectiles will go down.

When you run your game, it should work just like it did after the previous step. Even though you can't see a difference, though, this is going to be important for efficiency (once we actually remove the inactive projectiles from the game later).

***Step 6: Have teddy bears shoot***

Teddy bears should actually be able to fight back instead of just getting shot. For this step, you're making the teddy bears fire projectiles periodically.

1. Add code to the `TeddyBear` `Update` method to add the elapsed game time in milliseconds to the `elapsedShotTime`. If the `elapsedShotTime` is greater than the `firingDelay`, it's time for the teddy bear to shoot. Set `elapsedShotTime` to 0 and give `firingDelay` a new value using the `GetRandomFiringDelay` method. Call the `Projectile` constructor to create a new object, making sure you create a teddy bear projectile, using the static `Game1` `GetProjectileSprite` method to get the teddy bear projectile sprite, using the teddy bear `drawRectangle` and the `GameConstants` `TEDDY_BEAR_PROJECTILE_OFFSET` constant to calculate the location of the projectile (centered horizontally, offset from center vertically), and using the `GetProjectileYVelocity` method to set the y velocity of the projectile. Right after creating the new projectile, call the `Game1` `AddProjectile` method to add the new projectile to the game

When you run your game, the teddy bear should be firing at random intervals.

Notice that we didn't just have the teddy bear fire every second (for example). Instead, we make the teddy bears seem a little more intelligent by adding some variety in the delay between each shot. Don't expect too much intelligence, of course − they are just teddy bears, after all!