Beginning Game Programming with C#
Optional Project

PROJECT OBJECTIVE

This project will give you some additional experience developing a simple game using the concepts covered in the Beginning Game Programming with C# course. The project is broken up into a (large) number of (small) increments, each of which will have you apply the concepts we've covered in the course. You won't be able to build the complete game until the end of the course, but that's OK – it will be a great review of all the course concepts!

PROJECT DESCRIPTION

This project isn't worth any points, doesn't require any turn-ins, and is completely optional. Do it if you want to and don't do it if you don't want to!

The game you'll be developing is a simple 2D shooter in which you try to feed french fries to teddy bears until they explode. Unfortunately, the teddy bears explode after eating one serving of french fries! The teddy bears fight back, of course. Yes, it is totally weird. You can download an installation file for the final version of the game from the Optional Project course page.

INPUT DEVICE

The project increments below assume you're using a 360 controller as your input device for the game. If you'd rather use the keyboard, feel free to do so.

PROJECT INCREMENTS

Your work on this project is broken into a bunch of small increments. This is a good approach for a number of reasons. First, it lets you reinforce ideas you've learned in the course in a fun way. Second, it lets you experience the slow growth of an idea into a full game. Finally, this is actually how we do professional game development, so you get practice with that as well.

The only caution is that you need to do the increments in order. You can't skip increments and expect to successfully build the game.

Before starting on the project, you should download the template Visual Studio solution provided on the Optional Project course page. This template includes the classes (and even some methods) you'll need to complete the project as well a number of useful graphical and sound assets, so go download it now!

Increment 1: Add burger

After: Chapter 5

For this increment, you're adding the burger to the game. You won't be able to control it yet, but at least you'll be able to see it.

1. Add code to the `Game1` `LoadContent` method to assign the `burger` field (already declared in the `Game1` class) to a newly-constructed `Burger` object
2. Add code to the `Burger` `Draw` method to have the burger draw itself
3. Uncomment the code in the `Game1` `Draw` method that calls the `burger` `Draw` method

When you run your game, you should see a burger in the window.

Increment 2: Add bear

After: Chapter 5

For this increment, you're adding a teddy bear to the game. It won't be moving yet, but you'll see it in the window

1. Add code to the `Game1` `LoadContent` method to spawn a single teddy bear. Do this with the following code:

   ```
   SpawnBear();
   ```

2. Add code to the `Game1` `SpawnBear` method to generate random x and y locations for the bear using the `SPAWN_BORDER_SIZE` constant and the `Game1` `GetRandomLocation` method
3. Add code to the `Game1` `SpawnBear` method to generate a random speed using `GameConstants.MIN_BEAR_SPEED`, `GameConstants.BEAR_SPEED_RANGE`, and the `RandomNumberGenerator` `NextFloat` method
4. Add code to the `Game1` `SpawnBear` method to generate a random angle using `Math.PI` and the `RandomNumberGenerator` `NextDouble` method
5. Add code to the `Game1` `SpawnBear` method to create a new `Vector2` object using the random speed and angles you generated and the appropriate trigonometry
6. Add code to the `Game1` `SpawnBear` method to create a new teddy bear (name the variable that holds the new object `newBear`)
7. Add the new bear to the list of bears included in the game. Do this with the following code:

   ```
   bears.Add(newBear);
   ```

8. Add code to the `TeddyBear` `Draw` method to have the teddy bear draw itself

When you run your game, you should see a burger and a teddy bear in the window.

Increment 3: Move bear

After: Chapter 5

For this increment, you're making the teddy bear move in the game.

1. Add code to the `TeddyBear` `Update` method to move the teddy bear based on its velocity and the elapsed game time in milliseconds. Check out the `Update` code just after Figure 16.17 in the book if you need to see the solution

When you run your game, the teddy bear should be moving around.

Increment 4: Move burger

After: Chapter 8

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

1. Add code to the `Burger` `Update` method to move the burger based on the left thumbstick deflection. This should only happen if the burger has health > 0
2. Add code to the `Game1` `Update` method telling the burger to update itself using the current gamepad state. Pass `soundBank` as the third argument for the method call; you should be able to figure out what the first 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 left thumbstick.

Increment 5: Have burger shoot

After: Chapter 8

For this increment, you're making the burger shoot based on user input. There's a lot of work in this increment 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 right trigger is pulled (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 burger `drawRectangle` and the `GameConstants` `FRENCH_FRIES_PROJECTILE_OFFSET` constant to calculate the location of the projectile, and using the `GameConstants` `FRENCH_FRIES_PROJECTILE_SPEED` constant to calculate the speed of the projectile
4. Right after creating the new projectile, call the `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 right trigger. You may need to move the burger around while you shoot to test this.

Increment 6: Have projectiles move

After: Chapter 8

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

1. Add code to the `Projectile` `Update` method to move the projectile bear based on its velocity and the elapsed game time in milliseconds. Check out the `Update` code just after Figure 16.17 in the book if you need to see the solution

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

Increment 7: Control burger firing rate

After: Chapter 8

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 right trigger is pulled (which is 60 projectiles per second at a fixed time step). For this increment, you're controlling the burger firing rate.

1. Change the if statement in the `Burger` `Update` method to create a new projectile if the right trigger is pulled 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 right trigger. It's nicer to let the player release the right trigger to immediately set `canShoot` to `true`, yielding a different player tactic where the player repeatedly pulls and releases the trigger 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 right trigger isn't currently pulled.

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

Increment 8: Deactivate projectiles that have left the screen

After: Chapter 8

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 increment, 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 bottom of the window.

When you run your game, it should work just like it did after the previous increment. 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).

Increment 9: Have teddy bears shoot

After: Chapter 8

Teddy bears should actually be able to fight back instead of just getting shot. For this increment, 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 teddy bear `drawRectangle` and the `GameConstants` `TEDDY_BEAR_PROJECTILE_OFFSET` constant to calculate the location of the projectile, and using the `GetProjectileYVelocity` method to calculate the speed 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!

Increment 10: Have teddy bears collide with projectiles

After: Chapter 10

French fries should of course kill teddy bears. For this increment, you're making the teddy bears disappear when they collide with a projectile.

1. Add code to the `Game1` `Update` method to check for a collision between each teddy bear and each projectile in the game. A reasonable approach would be to have a foreach loop over the `bears` list with a nested foreach loop over the `projectiles` list inside it. This will make sure you check every teddy bear/projectile pairing. In the inner loop, check to see if the current teddy bear and the current projectile collide by checking if their collision rectangles intersect. If they do, set the `IsActive` property for the bear to `false` and the `IsActive` property for the projectile to `false`.
2. Add code to the `Game1` `Update` method to remove all the inactive teddy bears from the `bears` list. Look at the code just after the test cases after Figure 10.1 in the book to see a good way to do this.
3. Add code to the `Game1` `Update` method to remove all the inactive projectiles from the `projectiles` list. This will remove the projectiles from Step 1 of this increment and will also remove the projectiles you handled in Increment 8 when they left the game window.

When you run your game, the teddy bear should disappear when hit by french fries and the french fries involved in that collision should also disappear.

Increment 11: Have teddy bears blow up

After: Chapter 10

We'd like teddy bears to blow up when they're destroyed by french fries rather than just disappearing. For this increment, you're making the teddy bears blow up when they collide with a projectile.

1. Add code to the `Game1` `LoadContent` method to load the texture for the `explosionSpriteStrip`
2. Add code to the `Game1` `Update` method just after the code that deactivates a teddy bear and a projectile involved in a collision. Your new code should create a new `Explosion` object using the `explosionSpriteStrip` and the bear's `Location` property and add the new object to the `explosions` list.

When you run your game, the teddy bear should explode when hit by french fries.

Increment 12: Remove finished explosions

After: Chapter 10

At this point all the explosions we ever create in the game stay in the explosions list. For this increment, you're removing explosions that have finished playing from that list.

1. Add code to the `Game1` `Update` method to remove explosions that have finished playing from the `explosions` list.

When you run your game, it should work just like it did after the previous increment. Even though you can't see a difference, though, this is going to be important for efficiency.

Increment 13: Include multiple teddy bears

After: Chapter 10

It will be way more interesting if we have multiple teddy bears in the game. For this increment, you're adding more teddy bears to the game.

1. Add code to the `Game1` `LoadContent` method to add multiple teddy bears to the game. Move your call to the `SpawnBear` method inside a for loop that uses the `GameConstants` `MAX_BEARS` constant to spawn the appropriate number of bears.

When you run your game, you should now have multiple (precisely `MAX_BEARS`) teddy bears in the game.

Increment 14: Bounce teddy bears off each other

After: Chapter 10

You probably noticed that the teddy bears pass right through each other, but we'd rather have them bounce off each other instead. For this increment, you're making the teddy bears bounce off each other.

CAUTION: This is harder to do than it probably seems. I've put a lot of collision detection and resolution code inside the `CollisionUtils` `CheckCollision` method to protect you from that complexity. The method returns a `CollisionResolutionInfo` object you need to use, though, and you need to use that object properly for everything to work correctly.

1. Add code to the `Game1` `Update` method to check and resolve collisions between the teddy bears. You should start with two for loops; check out the `CheckAndResolveBearCollisions` method in the *Write the Code* area in Section 10.7 of the book to see how to structure the for loops (don't use the bodies of the loops, just the loops).
2. In the body of the inner for loop, call the `CollisionUtils` `CheckCollision` method and save the `CollisionResolutionInfo` object it returns in a variable.
3. If the returned object isn't null, there's a collision you need to resolve. There are two possibilities for each of the two teddy bears, though, so let's look at the possibilities for the first teddy bear. If the `FirstOutOfBounds` property is `true`, the first teddy bear ended up at least partially outside the game window as a result of the collision resolution. in this case, you should set the `IsActive` property for that teddy bear to `false` to remove that teddy bear from the game. Otherwise, you should set the velocity and draw rectangle for the first teddy bear to their new values using the `FirstVelocity` and `FirstDrawRectangle` properties of the returned object. Do the same processing for the second teddy bear.

When you run your game, the teddy bears should bounce off of each other.

Increment 15: Spawn new teddy bear when one is destroyed

After: Chapter 11

At this point you can kill all the teddy bears, then there's nothing else to do (except drive the burger around and shoot french fries, of course). For this increment, you're adding more teddy bears to the game when one or more teddy bears is destroyed.

1. Add code to the `Game1` `Update` method, right after you clean out the inactive teddy bears, to spawn new teddy bears (using the `SpawnBear` method) until you have `GameConstants` `MAX_BEARS` bears in the game again. You should use a while loop for this. Why? because you might have removed more than one inactive teddy bears, so you might need to call the `SpawnBear` method multiple times. You could also do this with a for loop, but use a while loop since we haven't used one yet.

When you run your game, you should have a new bear (or new bears) spawn immediately after you kill one or more bears. Now the game is unwinnable! Go for a high score instead of trying to kill all the teddy bears.

Increment 16: Only spawn new teddy bear into a collision-free location

After: Chapter 11

You may not have seen this, but spawning a new teddy bear into a location where it's immediately in collision can make the physics really goofy (or immediately kill the new teddy bear if it spawns on top of a projectile).At this point you can kill all the teddy bears, then there's nothing else to do (except drive the burger around and shoot french fries, of course). For this increment, you're making sure that new teddy bears only spawn into collision-free locations.

1. Add code to the `Game1` `SpawnBear` method to accomplish this. First, right after you've created the new bear, get a list of the collision rectangles for the other game objects in the game by calling the `GetCollisionRectangles` method. Next, add a while loop that keeps looping while the `CollisionUtils` `IsCollisionFree` method returns `false`; you'll need to provide the new bear's draw rectangle and the list of collision rectangles as the arguments for the method call. In the body of the while loop, get new x and y locations for the new bear using the `GetRandomLocation` method (like you already do earlier in the method) and create a new `TeddyBear` object using the new x and y. Make sure you assign this new object the new bear variable you've already been using (whatever you named it); otherwise, you'll get an infinite loop for the while loop.

When you run your game, you probably won't notice any difference from the previous increment because we'd only see a problem when a new teddy bear spawns into a collision. Of course, even if you've never seen that problem, it would occur a gazillion times as soon as you shipped the game, so it's a good thing we fixed it.

Increment 17: Add burger health and burger collisions with teddy bears

After: Chapter 12

At this point, the burger never takes any damage; it's time to change that. For this increment, you're adding a health property for the burger and detecting and resolving collisions between the burger and teddy bears.

1. Add code for a `Health` property to the `Burger` class. The property should have both get and set accessors. For the set accessor, make sure that the health value never goes below 0.
2. Add code to the `Game1` `Update` method to check for a collision between the burger and each teddy bear in the game. A reasonable approach would be to have a foreach loop over the `bears` list with an if statement inside the loop that checks if the current bear and the burger collide by checking if their collision rectangles intersect. If they do, subtract health from the burger's `Health` property using the `GameConstants` `BEAR_DAMAGE` constant, set the `IsActive` property for the bear to `false`, create a new `Explosion` object using the `explosionSpriteStrip` and the bear's `Location` property, and add the new object to the `explosions` list.

When you run your game, the teddy bears should explode when they collide with the burger. How can you make sure the burger is actually taking damage? Drive the burger around running into teddy bears. If you implemented the `Burger` `Update` method properly in Increment 4, making sure the burger can only move if its health is greater than 0, at some point the burger will stop responding to the left thumbstick.

Increment 18: Add burger collisions with projectiles

After: Chapter 12

We should also handle collisions between the burger and the projectiles in the game − both the teddy bear projectiles and, if you're bad at the game, the french fries! For this increment, you're detecting and resolving collisions between the burger and projectiles.

1. Add code to the `Game1` `Update` method to check for a collision between the burger and each projectile in the game. A reasonable approach would be to have a foreach loop over the `projectiles` list with an if statement inside the loop that checks if the current projectile and the burger collide by checking if their collision rectangles intersect. If they do, set the `IsActive` property for the projectile to `false`. You also need to reduce the burger's health, but the amount of damage the burger takes is dependent on the type of the projectile. Add an if statement just after the code that de-activates the projectile that uses the `Type` property for the projectile to subtract health from the burger's `Health` property using either the `GameConstants` `FRENCH_FRIES_PROJECTILE_DAMAGE` constant or the `GameConstants` `TEDDY_BEAR_PROJECTILE_DAMAGE` constant as appropriate.

When you run your game, both kinds of projectiles should disappear when they collide with the burger. How can you make sure the burger is actually taking damage? Drive the burger around running into projectiles. If you implemented the `Burger` `Update` method properly in Increment 4, making sure the burger can only move if its health is greater than 0, at some point the burger will stop responding to the left thumbstick.

Increment 19: Add burger and teddy bear shooting sound effects

After: Chapter 14

It's time to start adding sound effects to our game. For this increment, you're adding sound effects when the burger or the teddy bears shoot.

1. Add code to the `Game1` `LoadContent` method to load the audio content
2. Add code to the `Burger` `Update` method to play the BurgerShot cue when the burger shoots a projectile
3. Add code to the `TeddyBear` `Update` method to play the TeddyShot cue when the teddy bear shoots a projectile

When you run your game, you should hear sound effects when the burger and the teddy bears shoot projectiles.

Increment 20: Add teddy bear bounce sound effects

After: Chapter 14

For this increment, you're adding sound effects when the teddy bears bounce off the sides of the game window or off each other.

1. Add code to the `TeddyBear` `BounceTopBottom` method to play the TeddyBounce cue when the teddy bear bounces off the top or bottom of the game window
2. Add code to the `TeddyBear` `BounceLeftRight` method to play the TeddyBounce cue when the teddy bear bounces off the left or right of the game window
3. Add code to the `Game1` `Update` method to play the TeddyBounce cue when two teddy bears collide with each other

When you run your game, you should hear sound effects when the teddy bears bounce off the sides of the game window or off each other.

Increment 21: Add burger damage sound effects

After: Chapter 14

For this increment, you're adding sound effects whenever the burger take damage.

1. Add code to the `Game1` `Update` method to play the BurgerDamage cue when the burger takes damage from colliding with a teddy bear or a projectile

When you run your game, you should hear sound effects whenever the burger takes damage.

Increment 22: Add explosion sound effects

After: Chapter 14

Explosion should also have sound effects. For this increment, you're adding sound effects when an explosion is created.

1. Add code to the `Game1` `Update` method to play the Explosion cue whenever a new `Explosion` object is created

When you run your game, you should hear sound effects when the explosions start.

Increment 23: Add losing sound effects

After: Chapter 14

For this increment, you're adding a sound effect when the burger's health reaches 0.

1. Add code to the `Game1` `Update` method to call the `CheckBurgerKill` method whenever the burger takes damage
2. Add code to the `Game1` `CheckBurgerKill` method that checks if the burger's health is 0 (using the burger's `Health` property) and the burger isn't dead yet (using the `Game1` `burgerDead` variable). If both those conditions are true, set the `burgerDead` variable to `false` and play the BurgerDeath cue

When you run your game, you should hear a sound effect when the burger's health reaches 0.

Increment 24: Add health display

After: Chapter 15

For this increment, you're adding a text display for the burger health.

1. Add code to the `Game1` `LoadContent` method to load the Arial20 font into the `font` variable
2. Add code to the `Game1` `Update` method to set the `healthString` variable to the `GameConstants` `HEALTH_PREFIX` constant concatenated with the current burger health (using the burger's `Health` property)
3. Add code to the `Game1` `Draw` method to draw the `healthString` at the location given by the `GameConstants` `HEALTH_LOCATION` constant

When you run your game, you should see the burger's health displayed and modified as the burger takes damage.

Increment 25: Add scoring

After: Chapter 15

For this increment, you're adding scoring to the game.

1. Add code to the `Game1` `Update` method that checks if a collision between a teddy bear and a projectile was with french fries (use the projectile `Type` property). If it was, add the value of the `GameConstants` `BEAR_POINTS` constant to the score variable. Also, set the `scoreString` variable to the `GameConstants` `SCORE_PREFIX` constant concatenated with the current score
2. Add code to the `Game1` `Draw` method to draw the `scoreString` at the location given by the `GameConstants` `SCORE_LOCATION` constant

When you run your game, you should see the score displayed and modified as you hit teddy bears with french fries.

Increment 26: Celebrate!

That's it − you finished the optional project. Shoot the teddy bears to your heart's content, try to best your high score, and celebrate however you see fit!