



# HOW WE WON THE GAME OFF

THE STORY OF OUR DECISION TO STUDY AND PROCRASTINATE—GIVING US ONLY SIX DAYS TO DEVELOP ONE OF GITHUB'S FAVORITE GAMES.

ELI BRADLEY - DECEMBER 15, 2016



---

## PREFACE

Hello, and thank you for reading my book! Whether you are a family member, reviewing friend, or a prospective member of Westlake High School's Accessible Programming Club, with this book you can learn a lot about the strategies we used in developing *Immolation Organization*. In it, I attempt to describe the **entire** development process, save a few bugs we weeded out and unimplemented feature requests, with the occasional help of my fellow developer and friend Joseph Jin. Be sure to read it with a copy of the final Xcode project and source code in hand, available at <https://github.com/WestlakeAPC/game-off-2016>, and the final product for when you feel bored and need a break.

## MODIFICATIONS

Copyright © 2016 Eli Bradley, some rights reserved. This book is licensed under Creative Commons Attribution 4.0 International, meaning you have the right to share and adapt the contents of this book, so long as you give appropriate credit, provide a link to the license, and indicate if changes were made to the book. You may not add additional restrictions or apply legal terms or technological measures that legally restrict others from doing anything the license permits, as defined in Article 11 of the WIPO Copyright Treaty.

**TL;DR** Share and improve, but please link back to the original.

## ATTRIBUTIONS

In making this book, I relied on the assistance of others almost as much as our team did making *Immolation Organization*. Here is a list of everyone who made this book better:

- Joseph Jin, for reviewing my drafts and describing the enemy spawner for me.
- Sam Hollenbeck, for designing amazing pixel art for the game and this book.
- Kenney Studios, for supplying an *awesome* font library with [Kenney Game Assets 2](#).

## DEDICATION

For Larry Li, Nathaniel Li, and everyone who can learn from it.

- *Eli Bradley*

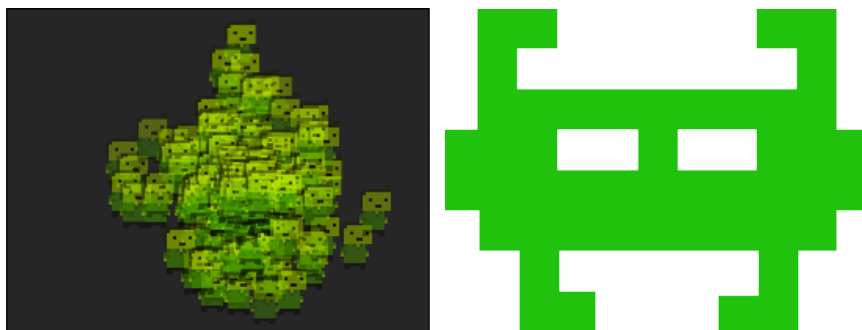
---

## PREPARATION

I'll start with a bit of clarification—there's no way to prepare *specifically* for a competition like the Game Off. I, personally, had been preparing to submit something to the competition since I discovered it years earlier (not that it helped any). You can always study previous “themes” of the Game Off—Git flow, change, hacking, modding, and augmentation—and try to guess what's next, I learned, or get to know their usual platform restrictions (they used to limit you to web frameworks,) but I'd say the only true preparation method is to brace yourself for a long month. When you can't anticipate the topic of the challenge beforehand, be ready to learn more about it when the competition starts. You have a whole month, after all.

We took this methodology to heart, and spent about half the month setting up the repository and dependencies. This included writing a `Podfile`, a `.gitignore`, and simply creating an Xcode project from Apple's SpriteKit template. We had big dreams about how we would build the game at first, and considered hooking up everything from Alamofire for networking to Fiber2D for graphics, but ended up using **none** of them. While it's good to evaluate what third party libraries you may need, it's important to determine whether not having to use the standard library is worth the hassle of dependencies.

## GETTING TO WORK! DAY 10



The first thing we did to get set up and started was attain graphics: It's extremely hard to test a game using visible bounding frames (shapes that represent the sprite's location) to model characters. So, we made the protagonist a particle emitter called “BadGuysMob” using the Xcode SpriteKit editor (above), and initialized it in the scene's `setUpScene` function:

```
self.badGuys = SKEmitterNode(fileName: "BadGuysMob")
if let badGuys = self.badGuys {
    badGuys.position = CGPoint(x: 0,
                               y: 0)

    badGuys.setScale(3)
    badGuys.particleBirthRate = 0.5
    self.addChild(badGuys)
}
```

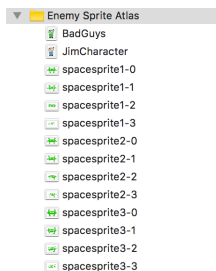
---

To make the character controllable, we added a `moveBadGuys(_ : CGPoint)` function to be called in all of the touch/mouse event functions. It consists of just one statement, which moves the player to the location specified in its unnamed parameter over a time period specified by the Pythagorean theorem, to maintain constant speed.

```
// MARK: Move "hackers" to player taps at constant speed.
func moveBadGuys(_ pos: CGPoint) {
    guard !playerDied else { return }
    badGuys?.removeAllActions()
    badGuys?.run(SKAction.move(to: pos, duration: TimeInterval.init(
        sqrt(pow(pos.x-badGuys!.frame.origin.x,2)+pow(pos.y-
badGuys!.frame.origin.y,2))/100.0)))
}
```

## NOW FOR THE ALIENS

After the protagonist, we began setting up the enemies. Our designer and good friend Sam Hollenbeck had designed three alien enemy sprites of different shapes for the game, and three stages of deterioration for each of them, because our theme was deterioration and being “hacked away at” to tie into GitHub’s theme announcement. We laid out these textures in a SpriteKit texture atlas, naming them according to the pattern `spacesprite enemy - stage`, and added them to a two-dimensional Array of `SKTextures`:



```
for enemy in 1...3 {
    for stage in 0...3 {
        textureMatrix[enemy-1][stage] =
            textureAtlas.textureNamed("spacesprite\
(enemy)-\ (stage)")
    }
}
```

## SPAWNING ALIEN SPACESHIPS

Now that we had textures for the enemies, we needed to put them on the scene! We delegated enemy spawning to its own function, which would be called ten times after screen resizing in the `setUpScene` function to populate the scene. Essentially, creating the enemies is a three-step process: First, we initialized an `SKSpriteNode` object with a random enemy texture from the `textureMatrix` using `arc4random_uniform`.

```
let enemyNumber = Int(arc4random_uniform(3))
let enemy = SKSpriteNode(texture: textureMatrix[enemyNumber][0])
```

---

Then, we need to set the enemy's position above the screen, and generate an SKAction that would take it from its position down through the scene, with random horizontal positions:

```
// Move from top to bottom
enemy.position = CGPoint(x: self.size.width/2 -
CGFloat(arc4random_uniform(UInt32(self.size.width))),
                        y: self.size.height * 0.55 +
CGFloat(arc4random_uniform(UInt32(self.size.height/9))))
moveEnemy = SKAction.moveBy(x: CGFloat((self.size.width/2) -
CGFloat(arc4random_uniform(UInt32(self.size.width)))) -
enemy.position.x,
                           y: (self.size.height * -3/5) -
enemy.position.y,
                           duration: 15.0 +
Double(arc4random_uniform(10)))
```

The final step is to put the enemy into action: Throughout its life, it calls `moveEnemy` to move across the screen, waits a random amount of time using the `waitRandom`, removes itself from the screen with and spawns another enemy by calling `spawnEnemies`.

```
// Spawn more enemies
enemy.run(SKAction.sequence([moveEnemy, waitRandom, SKAction.
removeFromParent()])), completion: self.spawnEnemies)
```

## GETTING PHYSICAL! DAY 23

Now that we had sprites on the scene and navigating around, we needed to come to a decision for how they interact! We chose to make it so that your character's mob size, represented by the particle effect's `particleBirthRate`, exponentially deteriorates, and does so even faster when it touches the aliens. The per-frame decay rate we settled on was 0.999. This process should be handled in realtime, so we made a method `playerDeter`, to be called in the `update` function, which is run before every frame by `SpriteKit`.

```
// MARK: Player deterioration
func playerDeter() {
    badGuys?.particleBirthRate *= 0.999
    for enemy in enemyArray {
        if enemy!.intersects(badGuys!) {
            badGuys?.particleBirthRate *= 0.999
        }
    }
}
```

---

## MAKING IT EASIER FOR THE PLAYERS

Still, there were two major gameplay problems: the enemies couldn't weaken, and the players couldn't stop weakening! We decided to solve both of these in a single event, where the enemy would lose health and the player would gain some. In order to change the alien spaceship sprite across the deterioration stages, we created a file private custom class extending `SKSpriteNode` specifically for the enemies, called `SKEnemyNode`. It's first member is an `enum` type representing deterioration stage we called `Deterioration`:

```
// MARK: Deterioration stages
enum Deterioration {
    case perfectShape
    case goodShape
    case badShape
    case finishHim
}
```

After we define deterioration state, we have some instance variables and the `deteriorate` function, which decreases the enemy's health by a factor of 0.01 and subsequently determines whether its health passes over an integer value. If so, and the enemy is in decent shape, it switches textures and demotes its internal `Deterioration` variable called `deteriorationShape`. If the enemy isn't in decent shape when it passes over an integer value, it removes itself from the scene, and spawns a new enemy.

```
// MARK: Make enemy deteriorate.
func deteriorate() {
    // If health will pass 3.0, 2.0, or 1.0
    if (Int(health) > Int(health*deteriorationRate)) {
        switch (deteriorationStage) {
            case .perfectShape:
                deteriorationStage = .goodShape
                self.texture = textureArray?[1]
            case .goodShape:
                deteriorationStage = .badShape
                self.texture = textureArray?[2]
            case .badShape:
                deteriorationStage = .finishHim
                self.texture = textureArray?[3]
            case .finishHim:
                self.isHidden = false
                _ = gameScene?.enemyArray.remove(at:
(gameScene?.enemyArray.index(where: { $0 == self })))!)
                gameScene?.badGuys?.particleBirthRate += 0.25
                gameScene?.spawnEnemies()
        }
    }
}
```

```

        self.removeFromParent()
    }
}
health *= deteriorationRate
}

```

Now, we just needed to change the enemy spawner function, `spawnEnemies`, to make enemies instances of our special `SKEnemyNode` subclass and give them references to their texture array of deterioration stages and the scene they belong to.

```
let enemy = SKEnemyNode(texture: textureMatrix[enemyNumber][0])
```

```

enemy.textureArray = textureMatrix[enemyNumber]
enemy.gameScene = self

```

We then altered the `playerDeter` function to call this function when an enemy intersects the player, and only damage the player when five times smaller than the enemy:

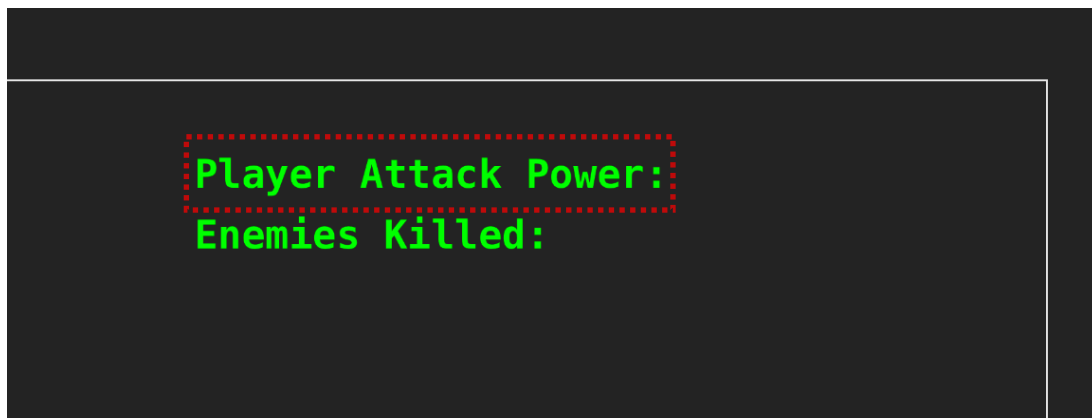
```

for enemy in enemyArray {
    if enemy!.intersects(badGuys!) {
        enemy?.deteriorate()
        if badGuys!.particleBirthRate * 5 < enemy!.health {
            badGuys?.particleBirthRate *= 0.999
        }
    }
}
}

```

## LABELING PROGRESS! DAY 33

Now that the basic framework was a done deal, it was time for a scoreboard. So, we added an `SKLabelNode` called `mobSizeLabel` to `GameScene.sks`, as highlighted below:



---

Then, we added references to the label in `setUpScene`, and positioned it in the corner of the scene at runtime, once window and scene size was known:

```
self.mobSizeLabel = self.childNode(withName: "mobSizeLabel") as?
SKLabelNode
mobSizeLabel?.position = CGPoint(x: self.size.width * 0.1, y:
self.size.height * 2/5)
```

We also added a function called `scoreUpdate` to update the label, called in the simple update method. It first changes the text of the label to "Player Attack Power: " appended to the mob size times 100. It then alters the label color to represent the severity of your current score. If the result is less than 5, it is a cause for concern, so the label is red. If you have more than 5 but less than what you started with, you're doing okay, so the label is white. Otherwise, you're better off than you started, so the label turns green.

```
// MARK: Score update
func scoreUpdate() {
    self.mobSizeLabel?.text = "Player Attack Power: \
(Int(badGuys!.particleBirthRate*100))"

    if (Int(badGuys!.particleBirthRate * 100) <= 5) {
        self.mobSizeLabel?.fontColor = SKColor.red
    } else if (Int(badGuys!.particleBirthRate * 100) <= 50) {
        self.mobSizeLabel?.fontColor = SKColor.white
    } else {
        self.mobSizeLabel?.fontColor = SKColor.green
    }
}
```

## MAKING ENEMIES A LITTLE LESS PREDICTABLE

We almost immediately gave up the idea of adding artificial intelligence to the enemies other than the straight line paths they travel on today, but that didn't stop us from making them harder on the players. Eventually, we realized that the game made more sense agnostic of direction, and that the spaceships should come from everywhere. We implemented this by wrapping the assignment of `enemy.position` and `moveEnemy` in a switch statement operating on a random number, which represents the direction the enemy would come from.

```
// Add enemy to scene
switch arc4random_uniform(4) {
    // Move from top to bottom
    case 0: // ...
    // Move from bottom to top
    case 1: // ...
```



```

// Move from left to right
case 2:      // ...
// Move from right to left
case 3:      // ...
default:
    print("arc4random_uniform(u_int32_t upper_bound); failure")
}

```

## STARTING OVER! DAY 43

At this point, there was a spectrum of possible progressions in the game, from a power level of 0 up to the hundreds, if you could play well. However, there was still no way to lose! So, we added another function to run along with `update` and check whether the player died, conveniently called `checkDeath`. It checks whether the player attack power, as displayed on the score label we made the before, is zero. If so, the game over screen scales from nothing to its normal size, a debugging variable `playerDied` is set to true, and the music stops.

```

// MARK: Check player death
func checkDeath() {
    if (Int(badGuys!.particleBirthRate*100) == 0) || playerDied {
        badGuys!.particleBirthRate = 0
        self.overScreen?.setScale(0)
        self.overScreen?.isHidden = false
        self.overScreen?.alpha = 1
        self.badGuys?.isHidden = false

        self.overScreen?.run(SKAction.scale(to: 4.0, duration: 1.5))
        self.playerDied = true

        backgroundMusic?.run(SKAction.stop())
    }
}

```

## STARTING BACK UP AGAIN

This game over screen will disappear as soon as it is clicked or touched. The `restart` method would be called by the `touchesBegan` and `mouseDown` handlers of the iOS/tvOS and macOS platform-specific `GameScene` extensions at the bottom of the file, respectively.

```

for i in self.nodes(at: t.location(in: self)) {
    if i.name == "overScreen" && self.playerDied {
        restart()
    }
}

```

---

```
}
```

The `restart` function removes all enemies from the scene that didn't already drift away when the game over screen was up, and then from the `enemyArray` in which they reside. The character reappears and the scores are reset, and the game over screen fades back out of the scene. Finally, the music begins to play again.

```
// MARK: Restart function
func restart() {
    for enemy in enemyArray {
        enemy?.run(SKAction.removeFromParent())
    }

    enemyArray.removeAll()
    for _ in 1...10 {
        spawnEnemies()
    }

    self.playerDied = false
    badGuys?.setScale(3)
    badGuys?.particleBirthRate = 0.5
    badGuys?.position = CGPoint(x: 0, y: 0)
    self.overScreen?.run(SKAction.scale(to: 0, duration: 1.5))

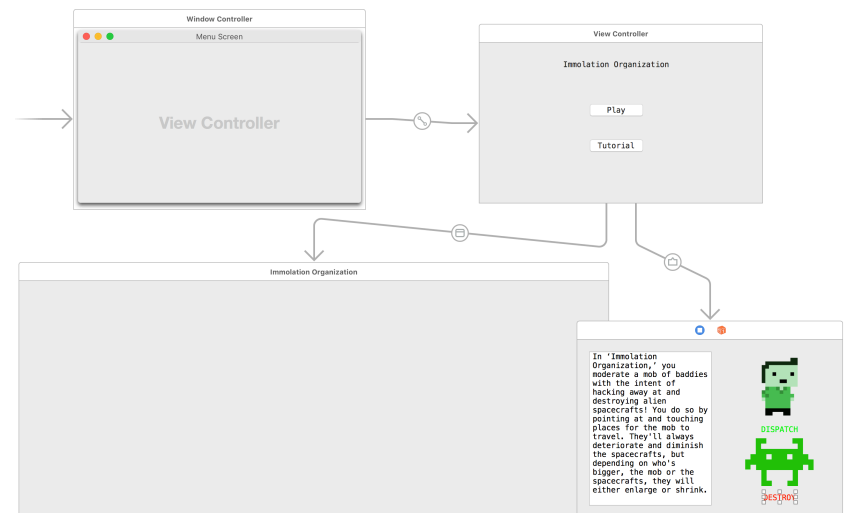
    backgroundMusic?.run(SKAction.play())
    for _ in 1...10 {
        spawnEnemies()
    }
}
```

## INTRODUCING THE PLAYERS TO THE GAME

We were in dire need of an instructions manual. While those reading this manual and rebuilding it may understand the gameplay, most would not. So, I drafted out an introduction:

In Immolation Organization, you moderate a mob of baddies with the intent of hacking away at and destroying alien spacecrafts! You do so by pointing at and touching places for the mob to travel. They'll always deteriorate and diminish the spacecrafts, but depending on who's bigger, the mob or the spacecrafts, they will either enlarge or shrink.

Then, I edited the macOS storyboard and connected a new View Controller to the entry Window Controller with two buttons: one that "modally" shows the View Controller containing the SKView with the game in it, and one that "pops up" the tutorial page, complete with graphics indicating which characters are yours and which are the enemies:

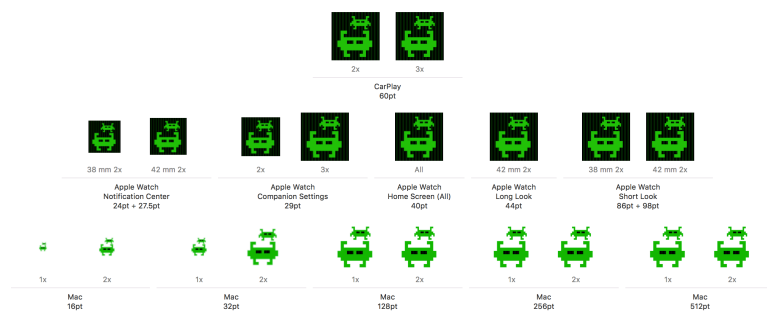


On iOS and tvOS, the process was similar, but there was no need to customize the animations between View Controllers, or to use Window Controllers at all.



## ICONS, BILLS, AND BACKPORTS! DAY 53

When designing an app icon, it's important to think about platform, regardless of whether you're using a generator (we did). For example, you might want to manually curve the corners of your icon if building for Android, or utilize transparency if building for macOS. Once we removed the background of our macOS icons generated the rest, we plopped them right into the `Assets.xcassets` image set of `Assets.xcassets`.



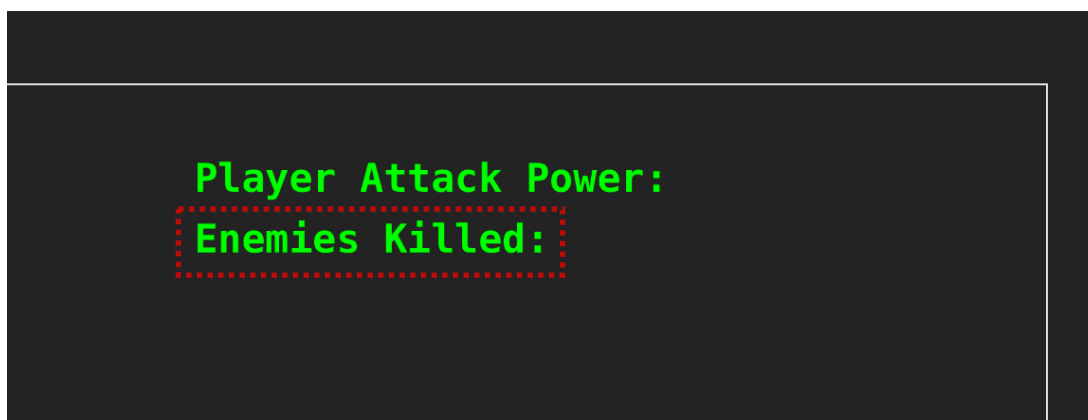
---

## KEEPING TRACK OF KILLED ENEMIES

While your attack power shows how well you are at the game, the number of enemies you've dispatched is a better measure of how long you last in it. Due to this fact, we added an internal kills counter, incremented in `SKEnemyNode.deteriorate`, when the enemy removes itself from the scene. Also, we made the `restart` function reset `kills` to 0.

```
case .finishHim:
    self.isHidden = false
    _ = gameScene?.enemyArray.remove(at:
(gameScene?.enemyArray.index(where:{ $0 == self })))!)
    gameScene?.badGuys?.particleBirthRate += 0.25
    gameScene?.kills += 1
    gameScene?.spawnEnemies()
    self.removeFromParent()
```

Also, we decided to add an `SKLabelNode` called `killsLabel` to `GameScene.sks` so we could represent this number on screen for the players.



The label is positioned and referenced similarly to `mobSizeLabel`, just a bit below.

```
self.killsLabel = self.childNode(withName: "killsLabel") as? SKLabelNode
killsLabel?.position = CGPoint(x: self.size.width * 0.1, y:
self.size.height * 2/5 - 1.5 * (killsLabel?.fontSize!))
```

We then added dynamic updating of the label to `scoreUpdate`. The label is red when you've killed no enemies, white after you've killed one, and green after you've killed ten.

```
self.killsLabel?.text = "Enemies Killed: \(kills)"

if (kills == 0) {
    self.killsLabel?.fontColor = SKColor.red
} else if (kills <= 10) {
    self.killsLabel?.fontColor = SKColor.white
}
```

---

```
} else if (kills > 10) {  
    self.killsLabel?.fontColor = SKColor.green  
}
```

## REMOVING OFF-SCREEN ENEMIES

While enemies were weeded out after their `SKAction` sequence ends, and when they're killed by players, this process isn't fast, and just isn't enough to keep screen constantly populated with alien spaceships. Because of this, we added a check of whether the enemy is off screen three seconds in to its life: If it's been alive for less than three seconds, it might be headed towards the screen, but after that, it can be weeded out of the game.

We performed this by adding a member `birthTime` to `SKEnemyNode`, which is initialized and set as the current time when the sprite is instantiated.

```
let birthTime: Date = Date()
```

Then, we ran a new function in called `removeOffScreenEnemies`, which goes through and checks whether any of the enemies older than three seconds don't intersect the scene, and removes them from both the `enemyArray` and the scene if so.

```
// MARK: Remove off screen enemies  
func removeOffScreenEnemies() {  
    for enemy in enemyArray {  
        if !(enemy?.intersects(self))! &&  
Date().timeIntervalSince(enemy!.birthTime) > 3.0 {  
            enemyArray.remove(at: enemyArray.index(where: { $0 ==  
enemy } )!)  
            enemy?.removeFromParent()  
            spawnEnemies()  
        }  
    }  
}
```

## SUPPORTING YEAR-OLD AND UNCOMMON APPLE PLATFORMS

Backporting to the last generation of Apple operating systems warranted several changes. First of all, we had to account for it in the `GameOff.xcodeproj/project.pbxproj` file's build targets themselves. After that, we had to change the code to use the characters' frames' `intersects` method instead of the characters' own `intersects`, so that the code would recognize their intersections on iOS 9.0.

```
- if enemy!.intersects(badGuys!) {  
+ if enemy!.frame.intersects(self.badGuys!.frame) {
```

---

Also, we had to use the resource package to store textures instead of a texture atlas.

```
- let textureAtlas = SKTextureAtlas(named: "Enemy Sprite Atlas")  
- textureMatrix[enemy-1][stage] = textureAtlas.textureNamed("spacesprite\(enemy)-\(stage)")  
+ textureMatrix[enemy-1][stage] = SKTexture(imageNamed: "spacesprite\(enemy)-\(stage).png")
```

In the process of testing the validity of our codebase on older platforms, we found that on watchOS, the game over screen and score labels are too large to show up!

```
+ #if os(watchOS)  
+     self.overScreen?.isHidden = true  
+     self.killsLabel?.isHidden = true  
+     self.mobSizeLabel?.isHidden = true  
+ #endif
```

And, finally, we were forced to mark the watchOS 2.0 version useless as WatchKit wasn't compatible with SpriteKit at the time. (Don't tell Apple we did that!)

```
+ @available(watchOSApplicationExtension 3.0, *)
```

## BUGS AND BUG FIXES! DAY 63

I've been incorporating 'clean' code as examples into the book, taken directly from the current codebase, or with minor modifications and features stripped out if they'll be shown in later examples. However, that's definitely *not* how it played out during development. Here's just a short list of bugs we delayed till the end of development (some of which never fixed!):

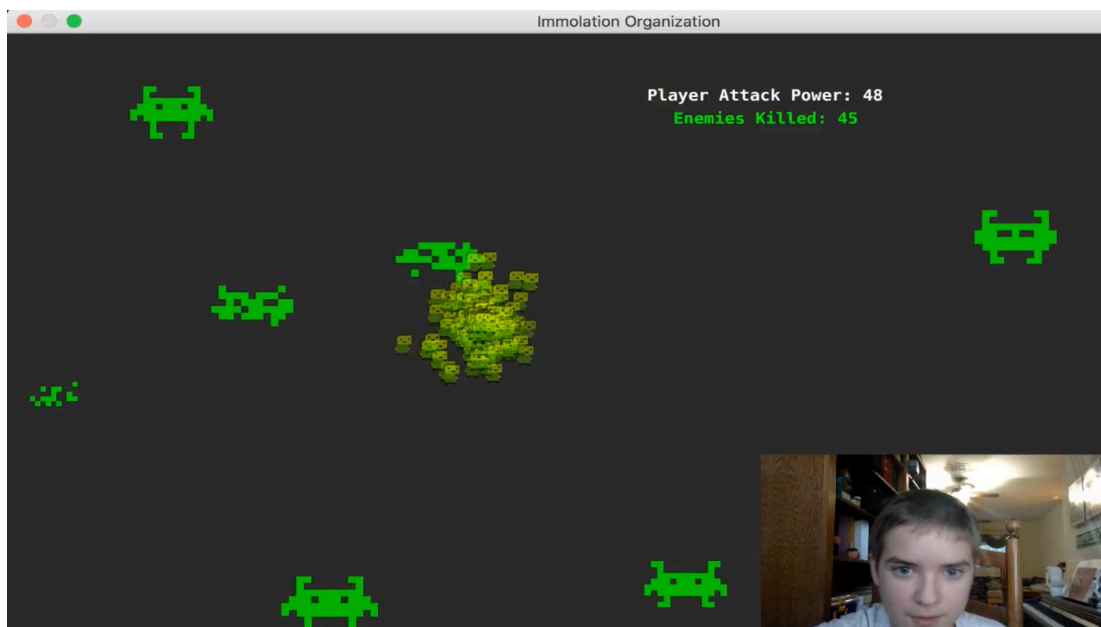
- Enemy numbers decrease, due to multiple respawn mechanisms activating at once.
- The Pythagorean theorem snippet doesn't make speed constant like it should.
- The WatchKit app extension was silently failing to load the Menlo font.
- A different difficulty was activated when you restart than when you opened the app.

---

## LET'S PLAY IO! (AFTERWARDS)

We had protagonists and spaceships coming from all directions, on the screen and navigating around, and deteriorating each other upon contact. Every feature was present from alien spaceship population balancing and score labels for power and kills, to a way for players to get back on track when disadvantaged in the game. We'd made a popup of encouragement for when the players weren't able to get back in the game, and even a training page to help them figure out how to. It was time to get the video footage GitHub required.

I assume most didn't take the route we did, but I knew this was calling for a livestream. So, that's what I did. For the first 20 minutes, I played the game, and for the rest, I explained its code. However, I didn't go into as much depth explaining it then as I am now.



We turned it in the following night, both to App Review and the Game Off, with links to the game footage we produced and a snapshot taken in it. We submitted six days of effort and testing and hoped for the best, and GitHub took the bait... We were featured on Lee Reilly's GitHub Game Off IV Highlights blog post, with the following description:

Take control of a mob that attempts to control a group of bad guys. Created by [@EtherTyper](#), [@AnimatorJoe](#), and [@SamHollenbeck](#) from [Westlake High School's Accessible Programming Club](#) using. Created with [Swift](#) and [SpriteKit](#). Available to from [Apple's App Store](#).

Not quite what the game's about, but we took it!

**THANKS FOR READING,**

**ELI BRADLEY.**