Creating a Game from Scratch

A journal of my experience

Table of Contents

1 The Idea

After having followed several Udemy Unity and Unreal videos to make various games, I decided to try "flying solo" to see how it goes. I've made games before—in high school, during the 1980s, I programmed Commodore Vic20 Basic and made a bunch of, well, lousy games—no, I don't have them anymore. The cassette tapes they were on are long lost.

This time, I have better tools (such as Unity3d), 10 years post-secondary education, 20 years relevant (research and programming, not games) work

experience, and for the first time in a long time, *time*. Not to mention a few Udemy courses under my belt.

So, I'm back to making games. It occurred to me to document my progress so others can learn from my successes and mistakes. The first issue was, what game should I make?

Several ideas came to mind: a Tetris clone, Lunar Lander, Pac Man, Super Mario World clone, etc. The last one I rejected after doing a little reading on 2D side scroller platformer games and having realized how big a project that would be, especially the very-long-world level design. I'd rather start smaller. Big can come later. The others were more feasible but I decided, I should probably go simpler still.

So I went to this website:

https://en.wikipedia.org/wiki/Timeline_of_arcade_video_game_history

and looked at the earlier games, on the assumption that earlier is simpler. I've already done a (not quite true-to-form) knockoff of Block Buster and Space Invaders, and decided, Asteroids looks good. I even have the Space Shooter packs from Laser Defender (the not-quite-space-invaders) which would give me the right assets:

http://opengameart.org/content/space-shooter-redux

So, the game is Space Rocks. How hard can it be? What could go wrong?

2 Show Stopper?

When I was working, management often used the term "show stopper" to mean, anything that was not forseen during the planning of a project that makes it impossible to move forward. Like the story I once heard of the new government building that would save money on washing dishes by having all styrofoam and plastic dishes and utensils. When the building was near completion, the rumor was it was delayed a month to add dishwashing facilities because somebody realized that you can't cook in styrofoam pots and pans. That is what's meant by "showstopper", whether or not the story has any truth to it. Supposedly it happened in the Virginia suburbs of DC and a friend of a friend of a friend's dad worked there and experienced that, or something like that.

For the Space Rocks game, it only took a little bit of thinking to realize there was one aspect that, for someone programming from scratch in assembly language, is not really any harder than programming Space Invaders from scratch, but I'm using Unity, not assembly-from-the-ground-up. Unity has a good framework that made Space Invaders simple, but it lacks a feature found in Asteroids: the screen wrapping.

Essentially, the flat screen in Asteroids is mapped to a torus. If the ship or a rock or a missile goes off the screen at the top, it reappears at the bottom (same X coordinate), and similarly with left and right.

The "preferred" way to do this, if building from scratch, is to write shader code that automatically does this: all x and y coordinates are taken to be "modulo" the screen width and height, respectively. That is too big a project for me, so to make this work in Unity, I need a different (possibly kludg-y) way. I needed to get around this so that the show could go on.

A Google search turned up:

https://gamedevelopment.tutsplus.com/articles/create-an-asteroids-like-screen-wrapping-effect-with-unity--gamedev-15055

which suggests two ways to do it. The first way I had already thought of: when an object is completely off the screen, just teleport it to the other side. This is not quite true to Asteroids, but it's workable.

The second way is to have 9 copies of every sprite in a 3x3 array. Most of the time, only the center sprite is visible on the screen. The horizontal distance between neighboring sprites in the array is the same as the screen width and the vertical distance is the same as the screen height. Therefore, as the central sprite leaves the screen, one other sprite enters in exactly the right place. Once the central sprite is completely invisible, the coordinates are "reset" to make the central sprite the one that is visible again.

This also is workable, though it "feels" bad: William of Occam would not like it. He is the one famous for Occam's Razor, which essentially says, given more than one explanation, choose the one that doesn't multiply entities. Showing one ship for the price of 9 seems inefficient, though it might well be the best way.

A third alternative is to forget being true to Asteroids and just have sprites bounce off the sides, kind of like in Omega Race. I decided to go with this, since

that is the simplest to implement in Unity. It only needs the physics engine Unity already has.

3 Start of Game Project

The next thing I "should" do is a Game Design Document, but I decided to put that off a bit and just get started setting up a viable game project. So, I fired up Unity (Version 5.4.2f2 to be exact, though mid-project I upgrade, always a risk but one that had little or no impact this time around), and opened a new project in 2D mode, calling it "Space Rocks".

Next I got my trusty Flexible Music Manager, which you can download for free on the Unity Asset Store at:

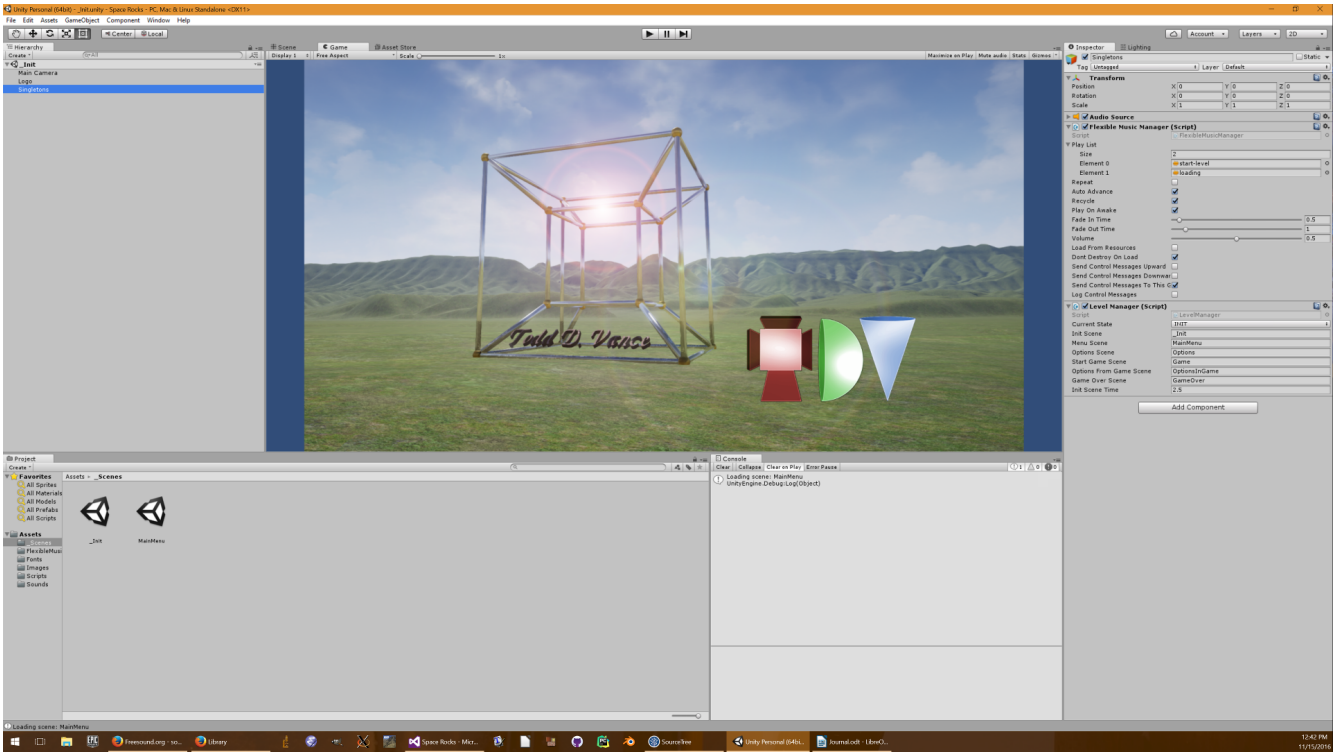
https://www.assetstore.unity3d.com/en/#!/content/73866

and a Level Manager script from the Laser Defender game. I decided to hack the Music Manager a little to be able to set tracks in code (that will eventually go in an update to the asset store asset) and to hack the Level Manager script quite a bit (and I might not even be done) to make something simpler than the mess I had made of it making Laser Defender. In fact, the newly-rewritten LevelManager.cs script is now small enough I'll just give you a link to a Github Gist of it (as it stands now—later I hack it some more):

https://gist.github.com/tdvance/340640751beb0f86eac9a88df8501268

It is made so that all you have to do is change the public "currentState" field in the inspector and it will automatically switch to the correct scene.

I attached these as components of a "Singletons" GameObject and added a logo background and set it up in the inspector to play some free audio clips from the web. Here is a screenshot that illustrates what I have so far.



4 The Logo

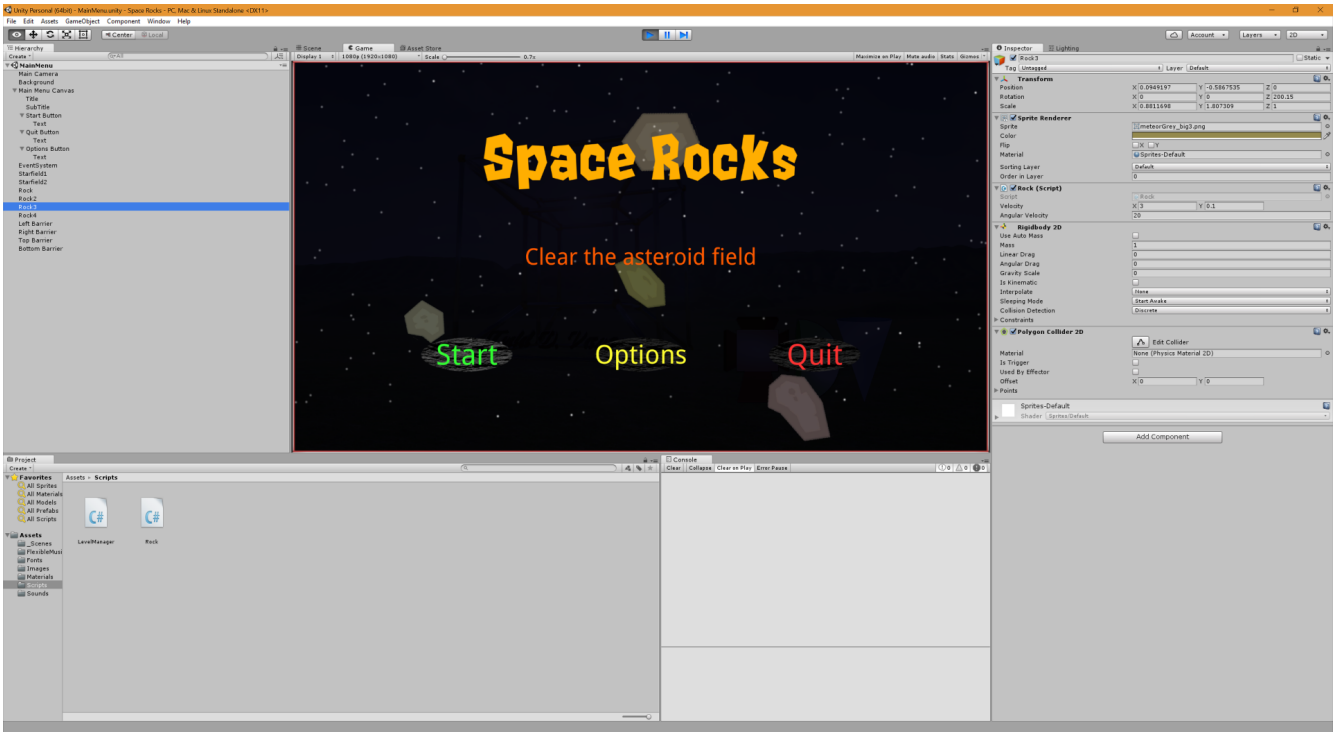
I shall take a break to explain the logo that is visible on the splash screen. The central object is a realization of a hypercube (a four-dimensional cube) in three-dimensional space, in order to give the impression of higher dimensions. The hypercube has 8 solid hyperfaces and 16 vertices. The background is in the South Branch (of the Potomac River) Valley in West Virginia: it is the view out my kitchen window, but without all the trees, houses, and so on. In reality I see lots of trees and barely see the tops of hills over them. I used topological data from

https://www.usgs.gov/products/data-and-tools/gis-data

converted (through quite a process involving downloading various tools) to a heightmap, and then imported into the Unreal Engine 4 editor as a landscape. The monogram at the bottom right is my initials rendered as 3d shapes: a tesseract (another way of showing a hypercube in three-dimensional space), a half-sphere, and a cone, to symbolize the game building in higher dimensions. The colors red, green, and blue are of course the primary colors for everything "pixel".

5 The Main Menu

After the splash screen shows for 2.5 seconds, the main menu is loaded. I used the same logo background, but darkened it to a subtle effect. I added Starfield prefabs from Laser Defender. I added new buttons using a rocky texture I found among Google images. I then added some rocks from the "Space Shooter Redux" game texture pack from Laser Defender. I scaled, rotated, and moved them around, tinted them, and added a Rock.cs script to simply give them velocity and angular velocity. I set the default Physics2D material to a perfect bounce (friction 0 and bounciness 1). I used stretched laser bolts from the asset pack to make the barriers on each side of the screen. Every sprite got a Collider2D (polygon collider 2D for rocks, box collider 2D for barriers) for bouncing, and the rocks got RigidBody2D components for physics. Here is how I set the main menu up. You can't see it but the rocks are bouncing around, and the stars are drifting.



I noted for later (right here in fact, no need to note it elsewhere):

> TODO: Make the barriers brighten when something bounces off of them, and play a force-fieldy sound.

The buttons are currently non-functional, but they highlight orange when hovered over, and dark blue when pressed.

So for the near future: make buttons work, have a game screen with player ship, missiles, rocks that fragment when hit by laser, sound effects, and a Game Design Document. The rocks and barriers on the main menu can be reused in the Game scene. A "rock spawner" could be used to automatically generate new levels.

6 Github Repository

Before doing anything else, I decided to begin source control on this project. It has saved me before, and no doubt will save me again in the future.

I use both SourceTree and the GitHub application (on Windows). Each has their advantages and disadvantages, and one covers holes left by the other. For example, SourceTree on windows won't publish a new repository to GitHub, but the GitHub app will.

So, first, in SourceTree (oops—it just crashed. I had GitHub app open too, maybe they cannot be open at the same time. So again…) in SourceTree, I clicked the Clone/new button, then selected the Create New Repository tab, then made sure the repository is Git, selected the path to the project, selected "Bookmark this repository", and clicked Create.

Now, I copied a .gitignore file from my Laser Defender project and staged it and hit F5 to refresh the window. This caused lots of cruft to be ignored.

When I clicked "Stage All", I got a warning that some files are large, so I ignored them with the right-click "ignore" feature (I ignored everything under Assets/Sounds, as those .wav files are large) and tried again. I typed the Commit message: "Initial Commit", and clicked Commit.

The repo is committed, but not published to GitHub. So, I closed SourceTree and opened the GitHub app. I clicked the down arrow next to the plus sign on the top left of the screen (GitHub app has the flaw that it uses mobile-device-like icons for everything and it's not clear where to find what you need just by looking at the icons) and selected the Add option, then selected the repo path, and clicked the Add Repository checkmark. Then I clicked the Publish button at the top right of the screen, typed a description, clicked the Publish checkmark waited for it to stop spinning, and clicked the Publish button at the top right of the screen again. (I don't know why this is necessary…). I exited the GitHub app and went to GitHub to confirm my project is there:

https://github.com/tdvance/Space-Rocks

I could see from the warning message on the website I forgot to add a README. Perhaps now is a good time to make the GDD, and do it in MarkDown so that it can be the README as well. I therefore decided to do that in the website, knowing that next time I open SourceTree, I'll have to pull first, then push as a result. I shall use my trusty GDD template, which looks something like this, though I won't fill in every section.

7 Game Design Document

The Game Design Document is updated here:

https://github.com/tdvance/Space-Rocks/blob/master/README.md

The initial version follows:

7.1 Space-Rocks

Shoot the space rocks in this 2D shooter

7.1.1 Game Design Document

Game Overview

Space Rocks is a two-dimensional game with a player's spaceship and a lot of rocks. The ship can accelerate, decelerate, rotate, or fire a missile. The rocks just bounce around. When a missile strikes a rock, the rock fragments into smaller pieces. When a missile strikes a small-enough piece, it vaporizes and is no longer a threat. Until then, a rock or fragment that hits the ship kills it. Rocks, missiles, and the ship bounce off the sides of the gamespace. The ship can be killed by its own missiles. The player gets three lives per play. Points are scored for shooting rocks and for clearing the play field, which results in a new set of rocks to destroy.

Difference from other similar games

Unlike asteroids, the game entities bounce off the sides instead of wrapping to the other side in torroidal fashion.

Feature Set

- 2D physics
- ship accelerates by control, but there is drag. Rotation happens when the rotate key is held down
- firing is single shot, one missile per keypress. Firing may be rate-limited. Missiles travel a fixed distance before evaporating.

- Rocks, when hit by a missile, split into smaller rocks which inherit the energy from the parent rock.
- Score display and high score saving
- music and sound effects
- various visual effects
- Options menu to adjust volume for music and sound effects

Camera

2D orthogonal

Game Engine

Unity3d

8 Wiring Up the Buttons

I've done this multiple ways in the past. This time, I put the Start, Quit, and Options buttons under the same empty game object called "Buttons", and attached a "Buttons.cs" script to the "Buttons" game object. This script contains simple methods for each button, methods which leverage the LevelManager singleton. The test of this is getting an error message that the game scene, etc. cannot be found, because I haven't made them yet.

In addition, I took the opportunity to put the barriers under a Barriers game object and the rocks under a Rocks game object, and I made everything a prefab.

I also took my Score Display prefab from Laser Defender and put two instances of it on the Menu scene, one for the score of the last game played, and one for the high score. I also sped up the rocks a bit on the menu scene. Finally, I pushed a commit to Github. It failed because I forgot I had changed the README file. So, I did a pull, then another commit, then another push.

9 Making the Game Scene

Next, I used Ctrl-D to duplicate the MainMenu scene and renamed it Game. After switching to the new Game scene, I removed the title and subtitle, changed the background image to all black, removed the Start and Quit buttons (but left the Options button), removed the High Score display, and moved the Options button to the top right of the screen. I also made the Options button transparent so rocks could be seen under it.

Then I added a player ship from the sprite assets, with Rigidbody2D and Polygon collider and put a simple script on it using CrossPlatformInput (one of the standard assets) with the horizontal axis mapped to angular velocity, and the vertical axis to AddForce, so up-down arrows, or W and S keys, would accelerate and decelerate (or accelerate backward), and left-right arrows or A and D keys would rotate left and right. Because this is CrossPlatformInput, it equally works with a gamepad joystick. I used the "Fire1" input (by default, left mouse button and left Ctrl key) to spawn a missile prefab. The missile has a RigidBody2D and a Box Collider, and on instantiation, is given an initial velocity. A float parameter timeToLive is set to 1.5f, and missile self-destructs after 1.5 seconds. For now, it just bounces off everything. It is now starting to look game-ish. This I committed to GitHub with the commit message "Created Game Scene".

10 Make Rocks Fragment When Shot

This turned out to be more involved than I thought it would be. I made several false starts that I had to delete. What ultimately worked (in GitHub under commit message "Make Rocks Fragment When Shot") was this.

The spritesheet from the Space Shooter Redux has 4 large gray rocks (I use gray since I can color it at will; I ignore the brown rocks), 2 medium gray rocks, 2 small gray rocks, and 2 tiny gray rocks.

The rules for when a rock is hit by a missile are as follows:

- A large rock fragments into four rocks, randomly selected among medium, small, and tiny.
- A medium rock fragments into two rocks, randomly selected among small and tiny.
- A small rock just turns into a single tiny rock.
- A tiny rock vaporizes.
- Spawned rocks are given random velocities to make them fly apart.

Some gameplay issues led me to add more conditions:

- A rock's velocity is capped at 3 game units per second (or else it will fly right through the barriers).

- When a rock exceeds the cap, it is adjusted toward the cap (rather than immediately set to the cap value) for smooth motion changes, or else rocks look like controlled spaceships.
    - In code, this looks like:

```
    void FixedUpdate() {
        if (rb.velocity.magnitude > maxVelocity) {
            rb.velocity *= Mathf.Sqrt(maxVelocity /
rb.velocity.magnitude);
        }
    }
```

Next, I was too lazy to build ten prefabs, one for each kind of rock, so I simply had the Rock.cs script swap the sprite. That led to a problem: the polygon collider no longer matched the sprite. That caused little rocks to seemingly bounce off of nothing. Googling got me a Kludge that works:

```
//Kludge: reset the polygon collider
Destroy(GetComponent<PolygonCollider2D>());
gameObject.AddComponent<PolygonCollider2D>();
```

There is currently no simple way to reset the collider in script the way you can in the Inspector. So, destroy it and add a new one. The comments on the Googled solution mentioned that this is not good if you have to hand-edit the collider for efficiency. But it works well enough on a simple game like this.

A few other design decisions: when the rock fragments, each rock's rotation is sped up by a factor of 3 to simulate (approximately) conservation of angular momentum. Also child rocks inherit the color tinting of the parent rock.

Finally, the missiles are tagged with "Missile" and the rocks with "Large", "Medium", "Small", and "Tiny" so scripts can easily keep track of what is what.

I made a new generic RockTemplate prefab (and deleted the "rocks" prefab, leaving the Main Menu rocks as they were since that was good enough) that gets spawned, and then all its properties set to what they need to be. Thus currently rocks can spawn rocks when they are destroyed. This should make spawning a new level of large rocks straightforward by copying code from the Rock.cs script.

Now, time to take inventory and decide what's in the near future:

- Shooting rocks should add to score.
- Getting hit by a rock should damage/destroy ship
- Options menu still is not available

- Music doens’t yet play during game...should be a simple fix.
- No sound effects yet.
- Tweaking: game might be too hard as-is? Harder than original Asteroids. The bouncing off walls adds so much chaotic motion not found in the original “torroidal” Asteroids.
- Check my first “todo” list several sections back: mostly complete and what isn’t is subsumed by this new “todo”
- Look at GDD and consider progress/todo/changes. I’ll do that right now in fact.

11 Game Design Document Revisited

A quick read of the GDD shows I’m following it fairly well with no major changes. So I shall leave it as is for now.

12 Music Plays on Game Level

This is an easy fix. Put a new GameObject in the Game scene called “StartGame” and attach script “StartGame.cs” to it. The script’s “Start()” monobehaviour method then sets the correct track (which is added to the playlist via the Inspector), sets “repeat” to true, and calls “Play()”, all on the FlexibleMusicManager instance.

13 Options Menu

I duplicated the MainMenu. Then I deleted the rocks, but left the barrier because it makes a good frame. I changed the title to Options and deleted the subtitle. I deleted two of the buttons, and renamed the third one to “Return”, and wired it to the Buttons.cs script’s “SubmitResumeMenu()” method.

Then I added a new empty GameObject called “Settings” and a “Settings.cs” script to handle it. Initially I added only one setting, Music Volume, a slider. Todo: add other options, such as sound effects volume, and save options between runs.

Next, I duplicated the Options scene and renamed it to OptionsInGame, for setting options while the game is paused. I added an Exit button (to exit the game in progress and return to the main menu) and wired it to the appropriate Buttons.cs method, which is still nonfunctional. I also deleted the eventsystem

and the camera, as it will be loaded additively to the Game scene which already has one of each.

Now, I go to the Buttons.cs method and fill in the TODOs to allow loading the OptionsInGame scene additively while pausing the game. There were a few sticky points: I had to add a script to the Menu scene to ensure it resumed playing the right track upon exiting the game. I also had to disable the Game canvas to keep it from being put on top of the Options canvas. I also had to replace the background sprite with a background raw image in the Options canvas or else the game's rocks and ship would render on top of it. The commit at this stage is titled "Options From Game".

Next up are sound effects and player ship damage.

14 Damage to Player Ship

This is another that seems involved because everything I do has dependencies I haven't done yet, so I work (mostly) top-down.

First, I give the ship some health:

```
public float initialHealth = 4;
private float health = 4;
```

This follows a suggestion in:

http://www.gamasutra.com/blogs/HermanTulleken/20160812/279100/50_Tips_and_Best_Practices_for_Unity_2016_Edition.php

to expose only initial health to the Inspector. To quote from the document:

> **36. Avoid making changes to inspector-tweakables in code.** A variable that is tweakable in the inspector is a configuration variable, and should be treated as a run-time constant and not double as a state-variable. Following this practice makes it easier to write methods to reset a component's state to the initial state, and makes it clearer what the variable does.

Of course, I'm far from following all 50 best practices, but this is one of the easier ones to follow. Note the older version of this list still has value too, even though intended for an earlier version of Unity:

http://devmag.org.za/2012/07/12/50-tips-for-working-with-unity-best-practices/

I also added fields for bigDamage and smallDamage, the former for large and medium rocks and missiles, and the latter for small and tiny rocks. Then, the collision code in Ship.cs becomes:

```
void OnCollisionEnter2D(Collision2D collision) {
     if (collision.gameObject.tag == "Missile"
         || collision.gameObject.tag == "Large"
         || collision.gameObject.tag == "Medium") {
         Damage(bigDamage);
     } else if (collision.gameObject.tag == "Small"
          || collision.gameObject.tag == "Tiny") {
         Damage(smallDamage);
     }
 }
```

Initially, I had "health -= bigDamage" and likewise instead of calling a Damage method, but it occurred to me it's better to refactor the damage out so I could, for example, add sprite effects without so much clutter in the if statements. Thus, Damage is:

```
void Damage(float amount) {
     health -= amount;
    //TODO show damage on sprite
    if (health < 0) {
         Die();
     }
 }
```

So, I need a Die method now (and writing that means StartGame needs a RestartLevel method, and on and on).

```
void Die() {
    StartGame game = FindObjectOfType<StartGame>();
     game.RestartLevel(1.5f);
     Destroy(gameObject);
 }
```

Initially, it was just the Destroy instruction, but I need to respawn a ship. After some fiddling, I decided the game class should handle that. I realize now the game class is not aptly named, because it does more than start the game (TODO rename the StartGame class to Game and be sure all references get fixed, and same with the game object "holder"). It now starts levels, or starts new ships in a level (which is the RestartLevel) method. So, I had to write that too:

```
public void StartLevel() {
     RestartLevel(0);
 }

public void RestartLevel(float delay = 0) {
    //TODO fixed number of lives
    Invoke("StartShip", delay);
 }

public void StartShip() {
    Ship ship = FindObjectOfType<Ship>();
    if (!ship) {
```

```
            ship = Ship.SpawnNewShip();
        }
        ship.Reset();
    }
```

I decided the Ship class should be responsible for the actually spawning and resetting things like its health. So, I began this:

```
public void Reset() {
        health = initialHealth;
    }

    public static Ship SpawnNewShip() {
        Debug.Log("Spawn New Ship");
        return null;//TODO
    }
```

which means I can test play, let the ship get destroyed, and see that a log message shows time to respawn after a short delay. I now have the scaffolding for damage to player ship. I committed the code at this stage to Github.

Next, I made the ship respawn. To do so, I needed a static GameObject field of the Ship class to hold a Ship prefab. It took some false starts to get it working correctly, for reasons I still don't understand. Sometimes destroying the ship GameObject made the prefab unusable as well. The commit titled "Ship respawns after destruction" has the working code.

Now, the ship respawns *ad infinitum*, so time to require "three lives and then game over." The place for the "lives" variable would be StartGame. This resulted in a need to change the Buttons script to call a new GameOver routine when exit is pressed in options menu during game. The GameOver is also called when there are no more lives. The commit for this change is "Three Lives and Game Over."

Remaining to do in the near future: visual and sound effects, and new level when level is cleared. Also need to update score display, high score display, and have a graphic for number of ships remaining.

15 Rename StartGame to Game

Since I'm not using MonoDevelop, but Visual Studio, renaming a class is not a one-step operation. It always makes me nervous because I've had issues doing it in the past. It's like going to pet a dog that has bitten you before. So, time to just, switching metaphors, eat that frog.

So, I have found, changing the name of the file first in Unity works slightly better than changing the code first. I can't remember what goes wrong with the latter way that doesn't also go wrong with the former way, just that the former way is better. So... done. As long as I'm in the Unity window, I might as well change the GameObject from StartGame to Game as well. Done.

Now, I go to Visual studio and change the class name from StartGame to Game. I click to put the cursor on the text "StartGame" where it says "public class StartGame", and hit Ctrl-R Ctrl-R to refactor-rename. I change the text to Game, and if all goes well, that's all to do. So, I test the game. F7 in Visual Studio to compile: success. Run the game in Unity: sometimes prefabs get broken, or wired-up buttons get broken when scripts change name. But no errors, so it worked this time. Nothing had StartGame wired to it by name.

16 Leveling Up

First, I added a new GameObject public variable, "rocks" so I can set it to the "rocks" GameObject in the scene heirarchy via the inspector. This is the parent of all rocks. Now, the very first level already has rocks there, so I'll leave it that way. I took much care to make sure the rocks don't bear down on the ship too quickly, a good thing for a first, "easiest" level. So, I must, before spawning rocks on starting a level, make sure there are not already rocks there. That's easy to do, using the same method used in Laser Defender: "foreach(Transform rock in rocks.transform){/*count the rocks*/}".

Thus, the StartLevel method is changed to:

```
public void StartLevel() {
      int count = 0;
      int desiredCount = 3;
      if (levelNumber > 2) {
            desiredCount = 4;
          if (levelNumber > 5) {
                desiredCount = 5;
              if (levelNumber > 10) {
                    desiredCount = 6;
                }
            }
       }
      foreach(Transform rock in rocks.transform) {
            count++;
       }
      if (count < desiredCount) {
            SpawnRocks(desiredCount);
       }
       numLivesRemaining = initialNumLives;
```

```
        RestartLevel(0);
    }
```

Next, I wrote the SpawnRocks script:

```
public void SpawnRocks(int howmany) {
        //spread them out on a grid
        int gridX = (int)Mathf.Sqrt(howmany);
        int gridY = (int)Mathf.Sqrt(howmany);
        if (gridX * gridY < howmany) {
            gridX++;
        }
        if (gridX * gridY < howmany) {
            gridY++;
        }
        int count = 0;
        for(float x=-7; x<=7; x+= 14f / gridX) {
            for (float y = -4; y <= 4; y += 8f / gridY) {
                count++;
                if (count > howmany) {
                    break;
                }
                SpawnRock(x, y);
            }
        }
    }
```

Thus, I need to move the SpawnRock method from the Rock script to the Game script, another refactoring. After thought, I decided, no, this is really a different routine than what the Rock script uses. The Rock script spawns smaller rocks out of a destroyed rock, but here I need to spawn a single large rock. So, I decided to copy some of the code instead, even though there is a little bit of duplication:

```
  public void SpawnRock(float x, float y) {
        GameObject rock = Instantiate(rockTemplate);
        rock.transform.SetParent(rocks.transform);
        rock.transform.position = new Vector3(x, y, rock.transform.position.z);
        Rock r = rock.GetComponent<Rock>();
        r.velocity = new Vector2(x / 4f, y / 4f);
        r.angularVelocity = Random.Range(-180f, 180f);
        r.sizeTag = "Large";
        int index = Random.Range(0, largeRockSprites.Length);
        r.rockSprite = largeRockSprites[index];
    }
```

Of course, this means more public inspector fields to fill in with sprites and prefabs, a small amount of duplication.

So, after doing all this, I went to test it, and it didn't work. I forgot to call StartLevel when all the rocks are cleared. So, back to the scripts to do this.

This was some work. I had to make a new StartGame method that called StartLevel, or else I reset to three lives every time I leveled up. After fixing that

and a few other issues, it worked. Ultimately, I count rocks remaining in the Game class's Update method, which for this small game doesn't seem to cause any performance issues. Initially, I had the Rock script do this upon rocks being destroyed, but that caused a chicken-and-egg problem, which I guess I could have solved by testing if rocks remaining was 1, not 0. Another issue I had was Update would call the new LevelUp method every tick when the rocks were destroyed, so the level number would go up by more than just one. I fixed that by having a boolean "levelingUp" variable that was true as soon as the rock count was 0, but was reset to false when the delay was finished and the new rocks were spawned. Then I only counted rocks when levelingUp was false.

The code as of this stage is in the Level Up commit.

At this stage, it is a playable game, just without too many nicities. It now needs score/lives displays, visual and audio effects, a few extras like bonus lives, etc. This is on track with the GDD. Some additional features that would be good are a shield button and a stop moving button (add a lot of drag) separate from the accelerate backward button. Perhaps an occasional UFO, mine, or whatever could add spice to the game.

I'll stop the narrative here, as these are personal choices that would be different from a reader making the game "their own" and, well, I'm getting lazy too :) I'll release this document first, and the game later (official release date: "when it's done"--what could go wrong? Space Rocks Forever!).