Spectrochet

Senior Project: Final Report

Category: Game

Date: 5/20/16

Author: Lucas Gnos

Table of Contents:

Topic Page #

Introduction

General Concept: A Videogame

The goal for this project is to create a simple arcade-esque video game for the Windows platform. The game is a top-down shooter with a focus on color-based gameplay in order to create a unique and dynamic environment for the players to combat each other (see full explanation in the Description section below) .The game was built within the free version of the Unity game engine, a cross-platform engine developed by Unity Technologies. While the Unity engine provides some very useful and powerful tools, the scope of the work in creating a game ended up being quite broad. The scope can be broken down into several broad categories, design, assets, and implementation. Design exists entirely outside of Unity and involves a desired vision for the game's final state, a rational and structured plan for implementation, and a way of recognizing potential user issues. As with most projects of any type my design began with a vision for an end product and a rough outline for how to get there. The design was quite fluid as well, as I was constantly making adjustments to reflect what I learned about the system, or to change components that did fit well. Assets are any external components, such as sprites or sounds, that are imported into Unity and used as necessary. I am by no means a good artist, so most of my assets were simple images or shapes that were easy to make and work with. Lastly, implementation includes combining the former two elements into Unity, and any programming necessary to achieve the game functioning as desired. Unity itself brings several large components to the table, other than being an interface for design and implementation. First of all it provides a robust physics system that will manage how the players move and interact with the game environment. Secondly, it provides a very diverse build system, that allows me to create builds of the game on a variety of platforms at the click of a button. All other aspects of the game were components that I implemented myself, or imported into Unity.

Background

Game Design

Design is of course a critical component to making a quality game, and is something that is difficult to create a definitive solution for. There is often conflict between the vision that the creator has for their design, and in how the user will use or experience it, and it is important to balance the two. I believe that the way to achieve this balance is twofold. The first is experience, in both creating and playing games, as you can learn both what to expect a player will do, or want to do, and in setting up the game's design to support it. The other is an iterative design process, that goes through playtesting and feedback, which can easily catch many mistakes or assumptions the designer makes. Different games experience different scopes of design, Tetris likely required less design to be successful than a modern strategy game. For my project, the design was mostly the vision I had for the final product, and the various problems I found or thought of during development. The one piece of design that stood out to me while reading from different sources was a GamaSutra article of an excerpt from Richard Rouse's *Game Design: Theory and Practice* (Game Design, Rouse III). In the article he describes the difference between an anticipatory approach, where the designer tries to account for all the ways they think the player will act, and a complex system approach, where you design the game to react in way the player would expect to their actions.

Game Engine: Unity

Understanding how to use the Unity game engine was the largest aspect in the technical background of this project, and as such large portion of this section is dedicated to it. Unity's sells itself on two major points, one is that it has phenomenal cross platform support, both in its API and its build options. Second it has an easy to

learn interface and tries to promote independent game development and widespread use. The second point is the reason that I chose to my game in Unity, because I wanted to minimize the time I spent learning the interface and API and jump right into working on the project.

Unity supports the use of both the C# and JavaScript programming languages. Object oriented languages are both very useful and common in developing games. Most games benefit tremendously from a hierarchical system, since many things may act in unison (such as moving all the components of a player's character), or have subtle changes from each other but still interact with the environment the same way. Inheriting certain abilities, such as how to move around, or how to collide with other objects is extremely useful, and no programmer likes repeating code. While the games industry has a large variety of languages and development choices (Quora, Gavrilliuc), C# is commonly used for small console or Windows games. Since my project was smaller in scope, and I did not need the lower level access C++ and Unreal Engine (my other option) provided, I decided to go with Unity. (It is also worth noting that any code attached to a particular object is called a Script within Unity and is what I am referencing if I use script anywhere below in the paper.)

Unity's top level of organization is its scene system. Basically a single scene is a self-enclosed area that holds all the components the creator may wish to place in it. An entire game may be created in one scene if the developer wished to do so, but it would likely sacrifice organization if the game was not very simple. To avoid this, multiple scenes can be created separately, each holding all the information that it needs to function correctly and nothing else. Then multiple scenes can then be linked together in code by loading another scene at a desired time (for example, clicking a level button in a menu and then loading a scene that is that level). If specified, game objects can also be made to persist through multiple scenes (otherwise they are destroyed on a load),

which is very useful if you wanted to maintain a game controller, or certain UI components.

Every object placed in a scene in Unity is an instance of the GameObject class, which is a container for any number of components that makes that object unique. The GameObject is guaranteed to have at least one component, which is the Transform component which details the object's location, rotation and scale in 3D or 2D space. All other components are ones that the designer adds to make the objects serve a certain purpose. GameObjects also can have children objects, whose transforms are relative to the parent object rather than the world, which is useful if you want objects to spawn or move together. Lastly, GameObjects can also be tagged, which is basically a string attached to that object, which in code can be searched for, getting all objects that share the searched tag.

The first type of component that is often used is a script, or a C# class that is created attached to the GameObject. What code goes inside a script is obviously up to designer, and the purpose that the object is trying to achieve, but there are certain important virtual functions that Unity provides. The first is the Awake function, which is called only once when the object is created, and is used to do any necessary initialization. A second is the the Start function, which is similar to Awake in that it is called when the object is created, but is also run any time a new scene is loaded. Start is then used more to make a necessary changes that correspond to the current scene, or to initialize on every load. The next two are the two update functions, FixedUpdate and Update. FixedUpdate does not necessarily get called every frame, but instead has a fixed time interval between each call. Unity calculates if the average of these calls falls behind (if the game is lagging behind) and may call the FixedUpdate function several times in one frame. FixedUpdate is used mostly for physics calculations or changes, as it is guaranteed to have a more consistent rate of change. Update, on the other hand, is tied to the frame rate and is called once every frame, so it can have a variable interval

between calls. Update is used for anything else that needs to change over time, such as timers, or checking for user input.

Unity has a complex and robust physics system that can simulate real situations efficiently. It functions by placing a RigidBody component on the gameObjects in question, which will allow it to interact with any other objects with rigidbody's attached. Then in a script, you would add a force in some direction onto the rigidbody, which would move the object in the expected manner. However, since I wanted to achieve a more arcade style movement in my game I cheat the physics system by setting its velocity component directly. This allows me to force a constant speed in a given direction and avoid my objects experiencing any acceleration, making objects much easier to control. Also from a realistic standpoint my objects have infinite mass, as they do not move when hit by other objects, which was useful in keeping blocks in their initial positions. Additionally, the physics system also handles collisions, making sure two rigid bodies never overlap. A collider component can also be placed on an game object as a trigger which fires anytime a collision is detected. Then in the script the OnTriggerEnter virtual function can be implemented to do logic on certain objects colliding (such as a player losing health when struck by a shot).

For 2D games such as mine, Unity also allows for the user to import sprites created elsewhere and turn them into game objects. This essentially means taking an image (I created mine using the Paint.net software) and making it a 2D object in the scene. Once imported, you can apply a collider component to the object, which will make a mesh out of the object, and allow for a rigidbody to be attached. The mesh collider also matches the shape of the non-zero parts of the picture, so for example if you make an octogon, it will interact physically with all 8 sides, rather than applying a box collider which will encase the image in a tightest-fit four sides. This is how I created my player objects and allows the hit detection on them to be more precise, as it matches their shape exactly.

For 2D games, Unity also has a mechanism for layering different objects in a scene together. The layering system essentially allows one to assign a layer to each game object, the higher levels will cover those below it, and only interact with those on the same level. My game only uses this system simply, with a background grid underneath the game area, UI elements on the highest layer, and all others in between.

UI elements in Unity are unique for other objects in that they all must be a child object of a Canvas object. The Canvas is an invisible rectangle that maps to the current size of the screen, allowing UI elements to stay in a location relative to the screen, rather than some global coordinate. For example I have the player's health depicted as hearts on either side of the screen and they will also remain there even if I change the size of the window, while other game objects will not move. Additionally the objects are given a certain scale to the canvas, which means that if the window size changes, the UI elements will scale proportionally, so they maintain relative position and size.

Other Unity elements I used include Lighting, Audio and Builds. Since Unity is cross platform it supports multiple rendering API's such as DirectX11 and OpenGL. However, I had no reason to change how my objects were rendered, so I only use the default lighting system. Audio in Unity is handled by attaching audio components to game objects, which can then be controlled within the script. The Build system is what makes Unity such a great tool for making games multi-platform, as with a single click I can make the files necessary for a variety of systems.

Description

Concept

Although the game's concept was iterative throughout the project, being influenced with what was feasible, I did start from the initial desire to make a 2D arcade-esque top down shooter. My initially reasoning was behind this was twofold. First, I knew my artistic capabilities were very limited in terms of creating artwork for the game, and thought that a 2D format would make it easier to create sprites or other objects. More importantly, before learning more about Unity, I thought that the most difficult challenge would be to move game objects in a physically realistic way, and to manage collisions, and so I chose an arcade style in order to simplify it. That being said, I did want to add a unique element to my game, allowing it stand out from *just* a 2D top-down shooter. My idea was to incorporate color in some core manner, giving an identity to each player, and adding a dynamic element to the game's environment. My idea was to have each player pick a color upon starting a game, their character and shots would be this color, and their shots would ricochet off objects of only their color and self destruct upon hitting another color. (which is also the inspiration for the name, spectrum of colors + ricochet = Spectrochet). The second part of the core concept involved the arena in which the players would compete in. The shape of the entire of the arena would be a rectangle, but it would be filled in by a background grid of some shape. I ultimately chose a grid of squares for simplicity's sake, although I would have liked to add options for multiple grid shapes. Each cell in this grid would be able to spawn a block over itself. The block would begin a neutral white color, however, any player could shoot a neutral block to convert it to their color (and allowing them to ricochet off it!). A few blocks would be populated at the beginning of a round, but they would also "fall" onto the map during the course of the game, potentially hitting a player, but also adding a block to the grid. Additionally, to keep the grid from getting cluttered up by fallen blocks, a player could also use a dash ability to break apart a block that

shared a color with them. The falling blocks would simultaneously change the map over the course of a game, but also add a sense of urgency for the players, who would have to watch out for their opponent and the falling blocks. These two elements combined created a dynamically changing environment for the players, with the block obstacles not only potentially changing color, but also constantly being created and destroyed. The color and grid was the concept for the gameplay itself, but I there needed to be some structure to the overall game as well. For a structural aspect, I chose to make each match be a best-of-two rounds, where in each round, each player would get three lives, and each life would take three hits to lose.

Another important part of the initial concept was that I wanted the game to have online multiplayer. Since the controls use the mouse to affect the player's direction there was no way to have a local multiplayer component. The idea was to have two players form a connection and pick their colors, then play the best two out of three rounds against each other.

<u>In-Game Components and Technical Details</u>

- Menus and Game Controller

The first component that the player sees when starting the game is the Main Menu. The menus in my game are simple in functionality and aesthetic (mostly due to the fact that I am not an artist), there is a Main Menu that allows the player to create a game or to exit. Selecting the create option will open a new menu screen from which the player can search for an opponent and then select the color they wish to play as (players cannot select the same color). Additionally there is a win screen where the player can go back to the main menu or exit the game. The menus are created by having each screen be its own Scene, when a user clicks on button, it loads the desired scene. This allowed for the objects that made up the menus to be cleanly separate from say the gameplay components. All Unity objects are destroyed when a new scene is

loaded unless explicitly stated otherwise in a script. The only object in my game that persists through multiple scenes is the GameController class. The game controller is provides critical information the other object's in the scene and is responsible for tracking the game's current state (score, lives, health etc.). The controller is created in the Create Game scene, where it records the color selected by each player. When the game scene is loaded, many objects use this by querying the controller for one of the chosen colors in order set their own color, which is how certain objects match their respective player. The most important job of the game controller while the player's are competing is ensure that the information being displayed accurately depicts the current state. This involves tracking the lives and health of each player, and when a player takes damage, a call is made to the controller, which updates the UI elements accordingly. It does this by querying all objects in the scene for one that has a tag that matches the desired tag, once it has reference to the object it then performs the desired action, or calls a relevant function attached to that object. The controller also checks the status of the match, if a round has ended it will reload the game scene, or if a player has won two rounds, it will load the game over screen. In the game over screen it allows the text there to display the rightful winner. Lastly the game controller checks if the player chooses to go back to the main menu after finishing a match and will destroy itself, avoiding the conflict of having two of the same object if the player decides to start another match.

- Grid and Block System

The next important aspect is the background grid and block system. The system is initialized and managed by a GridController class, which creates a specified number of cells upon loading the scene. The cells are created within the boundaries created by the four walls that make up the limits of the playable arena and stored in a 2D array accessible only by the grid controller. Then a variable number of starting blocks are populated in the grid randomly by getting a random value for each grid dimension and creating a block on that cell. The Cell class's script only does a few things, it has create

and destroy functions for the block to be created atop it, or it changing its displayed color. The blocks themselves have scripts that are trigger-based, when Unity detects a collision, the OnTriggerEnter2D function is called which I then implement to function as I wish. In this case I check the type of the collided object, if it is a player, the player is dashing and shares a color with the block, then the block object is destroyed and the space in the grid is opened up for future blocks. I can also set it so the block can not be destroyed this way, which is how I create the arena walls, and allows me to create instances of a single block template. However, if the collided object is a player's shot then I track how many times that player has shot this block, and update the color slightly in the player's color. Once a player has hit the block three times, it is locked into their color and will not change until the round ends or it is destroyed. The grid controller also manages how new blocks are introduced on to the grid during the game. It keeps track of a timer set to some limit, when it reaches the limit, an empty cell is randomly selected, and its color slowly turns red to indicate that a block is incoming. Once the warning period is over, a new block is created on the chosen cell, potentially damage the player if it "lands" on them, and resetting the cell's color back to normal.

- Player Management

The next important aspect is the player object. The player has one notable non-script difference from the other objects. While all the other objects are simple shapes (blocks are rectangles, shots are circles) that can use default colliders, the player is represented by a sprite that I created outside of Unity and imported. The player object uses a polygon collider, which essentially creates a shape out of the sprite image by adding edges along the colored portions. This allows the hit detection and physics reactions to match the shape that the user sees, instead of being say a simple invisible box around the character image. The player is also one of only two object types that move around the arena during the game (the other being the player's shots). Input is handled neatly by Unity use the getAxis function of the built-in Input class, which gives me the vector direction the player has selected. The vector itself is created by checking

the WASD key input, and setting the respective dimensions represented by the pressed buttons. Moving objects update their position in the FixedUpdate function (described in further detail in the background section), and in this case I merely get the direction the player is pressing and set full speed in that direction. The lack of acceleration creates the arcade-esque feeling I was going for conceptually, and also lends itself to more engaging gameplay, as the player doesn't have to slow down to change directions. The player also has a dash ability, which modifies the speed at which the player moves in their current direction. From a programming perspective, the dash was one of the more difficult things to implement and took a while to work as intended. It functions through switch-case statement that tracks the current dash state, which can be, ready to dash, dashing, or on cooldown. Various timer's track how long to stay in the dashing or cooldown states, and the dash itself is initiated is by checking if the player has pressed the space button since the last check. One notable element of the dash is that it multiplies the current speed of the player, so if they are not moving, or say pressed against a block, the dash will have no effect as their is no base speed to increase. The player controller script also keeps the player's sprite oriented towards the current mouse location. It does this by finding the angle between the current forward direction of the player (initially the positive x-axis) and the vector created by taking the difference between the mouse position and the player's location. It then rotates the player by this angle, which it turn updates the player's forward direction. The player controller also manages the player shooting. A shot is created and is sent straight ahead from the player (so it is always shot towards the mouse, when the mouse button is clicked). Shot objects are created with, and maintain, a constant velocity, reflecting off objects of the same color or destroying themselves if hitting and off color object. The constant velocity is done explicitly by setting the object's velocity to the current direction times the desired speed. Lastly the player controller also alerts other objects to events it detects. If the player shoots or dashes, the UI element that manages displaying the cooldown for the respective ability is alerted, allowing it to display accurately. Also if the player is hit by a

shot of not its own color, it will alert the game controller, so it can update the UI elements or game state as necessary.

- User Interface

The UI elements in the game has both static and dynamic elements. The static elements, such as text do not move or change during the game, and most of them have only a simple script to set their color to the respective player's color upon creation. The dynamic components include the health and lives of each player, which will change when a player takes damage or loses a life. Also there are two meters for each player at the bottom which track the cooldown status of a player's shot and dash abilities, and will show a bar that refills during the cooldown period. The UI bar is not 100% accurate as its own timer is set after the player's and may update sooner or later depending on how it is called in the frame. However, any delay is imperceptible to the player so the system functions fine. Another set of UI elements are two red panels, each taking up half the screen, and are transparent most of the time. When a player takes damage, the game controller will set the transparency to a perceptible level for a small fraction of second on the panel for their side of the arena. This red flash alerts the player that one of them has taken damage in somehow in order to avoid confusion. The last important UI element are buttons, of which there are several in each of the menu scenes. The buttons are static until the user hovers over or presses them, at which point they will change to specified color, on a press will call set of specified functions. In my game they are mostly used to call a function that loads the desired next scene, with the exception of the color select buttons in the create game scene. The color buttons have a script that checks the status of the other player's color buttons, not allowing both players to select the same color, or more than one color.

- Audio

Another, small component is the use of audio in the game. Whenever a player takes damage or shoots, a short audio clip is played to provide some feedback to the

player. The audio clips I used were ones I found on freesounds.org and imported into Unity, attaching them to the desired object to be used when necessary.

- Networking

The networking component ended up being the most technically difficult part of the project. In order for the game to function with multiplayer over the network, nearly all of the information in the scene had to be shared to both players. I initially thought when researching how to implement the multiplayer aspect, that I could just share the player's movements and commands, then each client could compute the game state on their own, but this was not the case for two reasons. One, is that any slight differences in computing the state on each client, would result in them diverging from each other to the point where the game no longer made any sense, and there would be no way to resync. Second, the background grid component in my game used a random number generation to create the distribution of blocks on the grid. This meant that there had to be only a single grid between both clients, and if this was the case, then every other component must be shared as well, since the grid connects to them in some way. The portion of the networking I did get working centered on Unity's NetworkManager component. This component handled the lower level actual network connection between the two clients, and provided a very basic in game HUD in order to initialize a connection. I would give this component a prefab of the player class, and when any client connected to the server, an instance of this class would be spawned. Any game object that would be shared over the network would also have to be assigned a network id component, in order for the manager to keep track of it. The aspect that caused problems is that once an instance of the player was spawned, the player script had to be changed in order to be able to differentiate what instance belonged to what client. In other words, each client wanted only to control their player instance, but the script on both instances were looking for input so either player could move each instance. To fix this the scripts for most of the game components had to be refactored to check for client ownership before accepting any input, which is currently a work in progress.

Evaluation

The following table shows my evaluation of the project's performance. The left column describes the component or aspect of the project that is being judged. The middle column is the baseline result for success that I set for myself at the end of the first quarter, and the last column is the evaluation of the project for that metric at the time of completion.

Project Component	**Baseline Metric**	**Results**
Map - Playable area filled with a grid that creates blocks obstacles	One rectangular map that uses a square grid / block shape. One or two walls start in the player's respective colors	Fully functional playable arena and background grid.
Player - Complete control over the player via WASD and mouse input	Player movement always orients toward mouse. Player can dash in their current direction	Player moves cleanly in an arcade-esque manner, dashing functions well but not perfect.
Player Shots (Circles)	Full collision interaction with players and obstacles. Reflect off of same color objects, otherwise are destroyed on contact.	Shots work perfectly in a local setting.
Map Obstacles (Blocks)	Full collision interaction with players and shots. Players can break blocks using their dash ability. Players shoot blocks to change their color. Blocks fall from the "sky" with some animation warning the player, if it lands on them, the player loses life/takes damage. Blocks fall continuously during the game into open spots	Blocks work well in a local setting, cells gradually turn red to warn of an incoming block, which will damage the player if they are hit when it lands. Open cell spots are continuously filled during a round.
Power ups	Power ups appear on the map instead of a block.	Only a speed boost power up was implemented, but it

	Power ups have simple changes like increase speed or lower cooldowns	works well, and other one's could be inserted easily.
Audio	Audio cues for shooting shots, blocks falling, and a player getting hit by either	Short audio clips play when a player shoots or takes damage.
Networking (Multiplayer)	Player can connect with another to compete in the 2-player game	Networking not finished. Can host a match and connect to it, but meaningful gameplay once connected is nonexistent.
UI	UI in the game accurately displays lives, health, cooldowns and round wins	UI works as expected
Menus	Start menu that allows for create game and exit. Create game will just start look for an opponent then start the game when one is found	Menu system functions well, also has a win/loss screen after a game has ended.
Effects (Animations)	Some sort of effect is displayed when a player takes damage or is destroyed	A side of the screen flashes red when the corresponding player takes damage.

If we analyze the project with a black or white view of whether it was a success or failure, the lack of a functioning network component would make it a failure. I would attribute the failure here to several reasons. The first would be a time management issue on my part with this project relative to my other coursework. Since the deadlines for this project were mainly self-imposed, often the project would "take the fall" and lag behind with the pressure of other due dates. If I had given it more time, or managed myself better I don't doubt that the project could be improved, even if the networking still remained unfinished, but this issue was to be expected. Another reason would be my inexperience in the both making games and using Unity as a development tool. With 20/20 hindsight it is easy to point out what I should have or should not have done in my development process to make it smoother, but it is not exactly helpful. However, I definitely could have researched the networking component more thoroughly early in

the first quarter, and built the project to support it from the beginning. My idea for the project was to have a more modular approach, where I ensured everything worked locally and then add features, such as networking, as necessary. This approach may work better in more focused topics than with the wide breadth of components that the game required, and is something I will take away from the project.

However, if we forego the networking component I believe the project was quite successful, as met or exceeded all the goals I set for myself. Although functionality may have stayed the same, I refactored much code several times to make it run more smoothly, and to be easier to understand and work with. In the end I am satisfied with the state of the project outside (if not completely happy with it), and I think given a little more time it could work as intended.

Conclusions

I was very uncertain on what kind of project I wanted to tackle when faced picking a topic for my senior project. The idea of making a game seemed good on one hand, it would be something I would look forward to working on, but it might not be the most original choice on the other. Ultimately, I decided to go for it because it seemed a more interesting topic than any of my other ideas that fit into the scope of a two-quarter project. Looking back at the project the first thing I would say is that creating a game ended up being a much more broad of a topic than I initially expected. Many components, such as creating sprites, audio, or design are all done without requiring any programming experience. The breadth does create a lot of space for creativity and uniqueness, but if you're not experienced in the other areas, it requires designing around them. There also seems to be a thousand little areas that could see some manner of improvement or attention, which can be a little overwhelming with the pace and time constraints of being enrolled in a university. The one thing that I did find frustrating with the project was how much time I spent learning how to use Unity’s API and interface efficiently, even at the end I was still learning how to use certain features.

It was frustrating to be held back at times by ignorance of the tools rather than complexity of the problem. This time sink may have been avoided in another type of project, but at the very least I became familiar with an industry standard tool. Also I think that as a topic a game would have been more suited to a group setting, not only to cover the breadth of components, but to make easier to potential design problems. Overall I enjoyed working on the game, especially the design aspect, always looking at what things could be added or changed to make it a more enjoyable experience.

Annotated Bibliography

- Unity Manual. (n.d.). Retrieved March 03, 2016, from http://docs.unity3d.com/Manual/index.html
    - The chosen game engine for the project was the Unity Engine developed by Unity Technologies. As such the documentation pages provided on their website were used often in order to determine engine functionality.

- Tutorials. (n.d.). Retrieved March 03, 2016, from https://unity3d.com/learn/tutorials
    - The first few weeks of the project were spent learning how to use the engine and the syntax differences for using C# with Unity. The majority of this time was spent watching or following the various tutorials provided by Unity on their website.

- Unity Community. (n.d.). Retrieved March 03, 2016, from https://unity3d.com/community
    - Community forums resource provided by Unity for questions and answers regarding the engine and its uses.

- Unity2D - Develop 2D games using Unity or similar engines • /r/Unity2D. (n.d.). Retrieved March 03, 2016, from https://www.reddit.com/r/Unity2D/
    - The subreddit for Unity, 2D specifically, questions answered by other Unity users. I found it a good place to get my questions answered when I hit a bug or was confused on how to accomplish something within the engine.

- Stack Overflow. (n.d.). Retrieved March 03, 2016, from http://stackoverflow.com/
    - One of my largest difficulties was finding how to use Unity's physics engine to achieve the effect that I wanted, any most of the solutions I found were due to the answers on various Unity StackOverflow posts.

- Gavriliuc, Anatolie. "What Programming Language Is the Most Used to Make Video Games? Which Is the Best?" Quora. Quora. Web. 18 June 2011. https://www.quora.com/What-programming-language-is-the-most-used-to-make-video-games-Which-is-the-best
    - A solid article by a popular writer and game developer describing the different languages and design tools used in the games industry.

- "Game Design - Theory And Practice: The Elements of Gameplay." Gamasutra Article. Web. 03 June 2016. http://www.gamasutra.com/view/feature/131472/game_design__theory_and_practice_.php
    - Interesting article about game design that is an excerpt from Richard Rouse's Game Design - Theory and Practice: The Elements of Gameplay

Analysis of of Senior Project Design

- Summary of Functional Requirements
    - Top down 2D shooter game that runs smoothly on a Windows platform.
    - Allows for local play between the player and an AI.
    - WASD, space, and mouse for player controls.
- Primary Constraints
    - The first constraint was learning the Unity API and interface, as well as picking up the syntax for programming in C#. A lot of time was spent learning how to implement certain simple ideas given the tools I was using and my experience with using them.
    - The other major setback was attempting to create a networked multiplayer component into the game in order to allow multiple players to compete against each other. Adding the networking would require a change to nearly every component in the game to allow their information to be shared across the network. This realization was discovered too late into the development process to realistically refactor to project given other time responsibilities and time constraints.
- Economic
    - Estimated and actual cost of game was zero.
    - Predicted estimated work time 150-200 hours.
    - Actual estimated work time 180 hours
- If Manufactured on a Commercial Basis
    - If put up on numerous locations to download games, sales for a year could be anywhere from a couple hundred to a several thousand if popular.
    - No manufacturing cost for each download.
    - I would list the game as a free download, so no cost.
    - Free download, so no expected profits.
    - Cost for the user would not noticeably increase the cost of otherwise using their system as normal.
- Environmental
    - No environmental impact that would add to the already everyday impact of using computers.
- Manufacturability
    - No manufacturability problems for a software-based project. Unity provides an easy to use build system, so providing the software to different platforms should be a relatively simple process.
- Sustainability
    - Sustainability issues could include a problem in porting the game to multiple systems, or loss of developmental support if Unity becomes unavailable for some reason.
- Ethical

    - No ethical concerns for making a simple videogame.
- Health and Safety
    - Only health and safety concerns would be those commonly associated with videogames, such as increased risk of seizure or migraine for those with photosensitivity issues.
- Social and Political
    - No Social or political concerns
- Development
    - I learned a lot about game design and how to use an industry standard development tool in the Unity game engine.
    - In terms applicable outside of game design I feel like I learned a lot in how to better prepare my design plan for a project of a larger scope. In particular, I learned how important it is to take into account future components of a project when first starting out, in order to make implementation much easier when the project eventually reaches that component.