Final Report: Genre Busters

Co-Creators: Devin Ober (BACS) and Keith Petitt (BSCS, BA Political Science)

Advisor: Laurie Murphy

Spring 2020

**Abstract:**

“Genre Busters” is a project made to satisfy a niche in the deck building genre. As a mixture of card based strategy and bullet hell action, our game caters to players who find the average deck builder too slow. Developed in Unity and coded in C#, our project utilizes the Unity ECS framework to maximize the number of simultaneous projectile and player interactions. “Genre-busting” is when an artist or author combines established styles into a unique and revolutionary experience for consumers. In “Genre Busters”, players construct decks with up to two different “genres”, allowing them to find unique and expressive ways to defeat their opponents in 1-on-1 combat.

**Table of Contents**

[**1. Introduction**](#_nlx6hlheqbe8) **4**

[1.1 Functional Objectives](#_fthumcit0rbj) 4

[1.2 Educational Objectives](#_udoervpbq4ju) 4

[**2. Requirements**](#_oqihhigbaalp) **4**

[2.1 Deck Construction](#_leqlpcf5u0d4) 4

[2.2 Competitive Matches and Cards](#_6izwx7du6qgh) 5

[2.3 Card Testing](#_7rjx7c3k67bp) 5

[2.4 Theme](#_eo4lcvfuwaps) 5

[**3. Design**](#_wb8fshiqo1ga) **5**

[3.1 Entity-Component-System](#_h1ieterp2i5m) 5

[Entities](#_g2utoetu4mby) 5

[Components](#_gdjv0ekjiucm) 6

[Systems](#_fo870h8slncf) 6

[ECS in “Genre Busters”](#_kenjzlrhmskg) 7

[3.2 Events and the Event Manager](#_5vbqeye5z6z5) 8

[IGenericEvent](#_8qdczk6q2k1z) 8

[IGenericEventListener](#_fzrbrqt1jjr) 8

[Event Manager](#_lrr0fl4su80g) 8

[How It Works](#_q46pyeygxlyv) 8

[Limitations](#_2ozbyvk9ssok) 9

[3.3 Collision Detection Optimizations](#_3cyqilzww6v) 9

[Collision Byte Masks](#_bkls0u39n174) 9

[Byte Mask Examples](#_rb6kkfro43o9) 10

[Quadtrees](#_u6jhilzic40a) 11

[Nodes](#_aiwi7vafoyr1) 12

[Implementation](#_moxtnz13mmea) 12

[**4. Implementation**](#_2ee448eg0nak) **14**

[4.1 Tools Used](#_xi77wmmtao43) 14

[4.2 Gameplay](#_xlt65zm5icpf) 15

[4.3 Testing](#_343bqkg7lcz8) 15

[4.4 Theme and Freelance Management](#_jk7h1h32s52t) 16

[**5. Future Work**](#_h2c5it1luq11) **16**

[5.1 Options](#_e3vtdk65hlsk) 17

[5.2 Network Play with Rollback Netcode](#_bsptpvo59bm2) 17

[5.3 Visual Update](#_acberk9j7seq) 17

[5.4 AI Opponents](#_1cjim86ua0t3) 17

[5.5 More Cards and Genres](#_kfavl34j5m8) 18

[**6. References**](#_5zlbfw947luf) **19**

[6.1 Special Thanks](#_yrktxnum5cws) 19

[**7. Glossary**](#_miks9q4nffsb) **20**

**Figures**

[**1.** Entity-Component-System Example **7**](#_kspi05vtn0ps)

[**2.** Collision Byte Mask Table **10**](#_dl6drla6hg6b)

[**3.** Screenshot of Quadtree **13**](#_u90jiwraqtsj)

[**4.** Character Art and Card Examples **16**](#_g6fosx8tk0o0)

# 1. Introduction

Starting as a spontaneous summer idea, Genre Busters has grown to begin filling a specific niche in the gaming industry. A marriage of bullet hell action and card based strategy that eliminates some of the downsides of each of the genres. Bullet hells as a genre are built to contain ordered chaos with patterns of projectiles that need to be avoided and woven between. Good examples of this genre are games such as “Touhou” or “Undertale”. Card games on the other hand normally utilize turn based strategy as you attempt to execute your plan for victory. Building a deck with a certain win condition in mind is at the heart of the deck building genre. Genre Busters is focused on bringing these to genres together utilizing fast paced gameplay, with an emphasis on strategic planning that can be played competitively with others. All with an expansive theme which is able to be expanded further at the drop of a hat.

## 1.1 Functional Objectives

1. Game screen where players can compete.
2. The ability to build, save, and edit decks.
3. The ability to test cards within the Deck Builder.
4. Create a workable theme to immerse players.

## 1.2 Educational Objectives

1. Learn the Unity Entity Component System
2. Learn how to manage freelance artists and voice actors
3. Learn how to develop and organize a brand new game from scratch

# 2. Requirements

We developed a set of both User and Developer stories which we had to achieve to consider our game to be finished. The most important step in the process was pursuing these stories without taking shortcuts, straying away from hardcoding any features and keeping every part of the project modular.

## 2.1 Deck Construction

As a player, I want to be able to create, save, and edit a deck of cards which can be used in a competitive match. Being able to save these decks under specific names is especially important for organization of different strategies.

## 2.2 Competitive Matches and Cards

As a player, I want to be able to compete against other players locally with a deck of my choice. The cards should lend themselves to being competitively balanced to create a balanced meta, allowing multiple viable strategies.

## 2.3 Card Testing

As a player and a developer, I want to be able to test cards within the deckbuilder with the press of a button. The deckbuilder game should function exactly the same as competitive matches allowing strategies to be created. This doubles as a quick testing area for developers to test cards and test for bugs.

## 2.4 Theme

As a player I want an interesting theme to play within, complete with sound effects, voice acting, and fitting art. The visuals and audio should help players by providing sound cues and easy to understand visuals so as not to get in the way of gameplay.

# 3. Design

## 3.1 Entity-Component-System

The fundamental architecture that “Genre Busters” is built upon is the “Entity-Component-System” (ECS) architecture implemented in Unity. This framework functions in three primary parts, beginning with entities. Entities are generic objects whose whole purpose is to hold Components that are attached to them by the EntityManager. Components, in Unity, are public structs that serve to store game-related variable data that is manipulated by systems over time. Systems update at the rate at which Unity itself updates and are tasked with performing continuous calculations on component data passed from the EntityManager, updating these values to overwrite the value stored in entities.

### Entities

In “Genre Busters” entities are created using a static constructor based on the type of entity we want it to be. These static constructors give us as developers the ability to easily define the required components an entity must have in order to be considered an entity of that type. For example, as seen in Figure 1, we see both a PlayerEntity and a ProjectileEntity. While both of these entities require Translation and Movement components, they also have Player and Projectile components respectively. These unique components are added to each and every entity of their respective types.

### Components

As previously mentioned, components are key to determining what type of entity you are currently working with. By having entity constructors that closely regulate what types of components are attached to entities, you can guarantee what type of entity you are working with based on the types of components you are working with. With this in mind, whenever a function is passed in a generic “Entity” to its function, you can make a “HasComponent<type>”, with type being a component unique to the type of entity you are working with. If our function receives an “Entity” that has a PlayerComponent then we are sure to be working with a Player Entity.

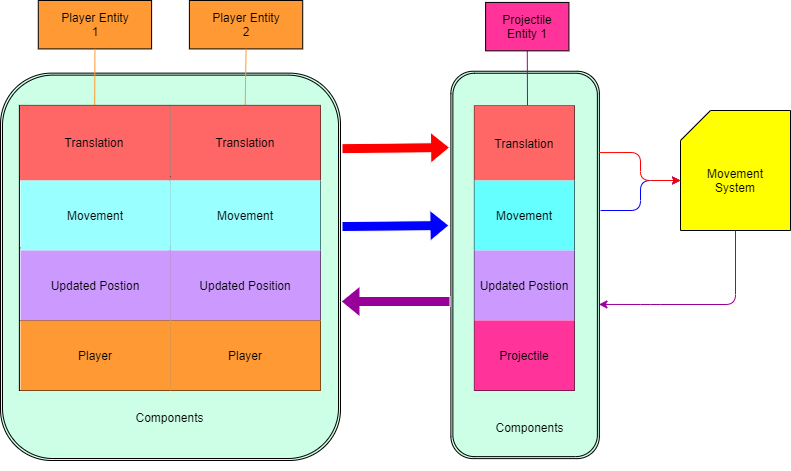
With this in mind, components can also be used as flags that can grant additional properties if your code is structured correctly. In “Genre Busters” we have an “IsBoundary” component that allows us to quickly determine if the entity we are working with is one of our two different types of boundary entity types we have defined in this project.

### Systems

Systems in Unity are powerful as they can quickly search through all entities present in the current World, with the option of filtering out entities that require certain types of components. These systems can either require “All”, “One”, or even “None” of the component types defined in the statement. This allows us as developers to easily regulate which type of entities we are working with in a system without doing extensive HasComponent checks.

One of the primary aims of ECS is to have highly modular code, meaning that entities, components, and systems should have single responsibilities, lest we have unruly blocks of code to navigate in the event of bugs and errors. While it is certainly possible to have a “GameSystem”, where the one system manages all gameplay logic, it is much more logical to have systems based on their designed function. Such systems include movement, collision detection, and player input handling just to name a few. By following this design we can use these same systems in other projects that use the ECS structure, making our code much more readable, maintainable, and exportable.

### ECS in “Genre Busters”



###### Figure 1: Entity-Component-System Relationship Example (described below)

The above image is a sample illustration of how “Genre Busters” uses ECS. The Movement System is responsible for moving entities every update, using a simple equation to move entities along their currently assigned vector based on their current position. This vector is unique to each entity it is assigned to; because this is gameplay related data it is stored in the “Movement” component. As stated, the Movement System needs to update position data directly, meaning the Movement System needs positional data as well as movement data. This positional data is therefore stored in another component called a “Translation”. With access to both of these components the Movement System will calculate the new position of each entity with both Movement and Translation components. A hypothetical entity without one or both of a Movement or Translation component is therefore skipped over by the Movement System and never affected by it.

It is important to understand that systems do not filter out which type of entities they operate on, but rather what type of components each entity must have in order for it to be operated on. This is the aspect of ECS that makes it so modular and flexible an environment to work in, as our movement system is guaranteed to work exactly the same with processing Players and Projectiles. Furthermore, if we needed a third type of entity that also needed to move we would simply give it both Movement and Translation components and it would move just like our Player and Projectile entities.

## 3.2 Events and the Event Manager

The Event Manager class is a custom class that allows for the creation of event types and listeners to facilitate an indirect and decoupled relationship between the systems that create events and the MonoScripts that listen to them.

### IGenericEvent

The IGenericEvent type is an interface that is handled by the Event Manager. IGenericEvents do not require any functionality as each event is used to trigger the activation of code within a listening MonoScript.

### IGenericEventListener

The IGenericEventListener type is an interface that is used to denote a MonoScript as having the capabilities of receiving event notifications from the Event Manager, triggering the activation of the code it activates whenever it receives the correct event type. Each IGenericEventListener is required to have a public “bool Handle(IGenericEvent evt)” function that is called whenever the corresponding event type is generated.

### Event Manager

The EventManager is a Singleton variable responsible for the project’s overall event-listener design, starting with the storage of the “EventDictionary”. This dictionary is a dictionary of : <Type, List<WeakReference<IGenericEventLisener>>.

In this dictionary, the key is whatever type of IGenericEvent created, while this could have instead been limited explicitly to IGenericEvent, a key of Type allows for essentially anything to be a key in the Event Manager, making it much more flexible if we need it to be. Each list tied to the key IGenericEvent is the list of weak references to each MonoScript that implements the IGenericEventListener interface.

The second aspect of the EventManager is the storage of the Queue, a List<IGenericEvent> that the manager empties in a First-Come-First-Served fashion every update.

### How It Works

We will explore how everything comes together by analyzing how “Genre Busters” handles collision detection and collision handling. This setup will require the creation of the CollisionStartEvent, the CollisionDetectionSystem and the CollisionListener MonoScript.

When the CollisionListener enters its Start() function, it needs to call the EventManager’s RegisterListener<T>() function, where the reference to “self” is passed into the event listener List of type CollisionStartEvent. A MonoScript that never registers itself as a listener will never actually receive notifications, even if it is otherwise prepared to handle these events.

Next, the CollisionDetectionSystem needs to perform calculations that determine whether two entities are considered to be colliding with each other. Once a collision is found it then queues a new CollisionStartEvent. If we were to detect and handle collisions within this system simultaneously we would be going against our modular design principles, hence why the EventManager exists in the first place.

Once the CollisionStartEvent is queued, it will eventually be cleared by calling the Handle() function of each IGenericEventListener referenced in the current event’s Type stored in the “EventDictionary”. This means that the CollisionListener has its Handle() function called, and then calls the relevant helper functions to handle collisions. This may include any combination of deleting entities, attaching or removing components from entities, or adjusting the values of components already attached to the entities colliding.

### Limitations

While this structure is great for having decoupled code, its greatest downfall is that event handling always happens 1 update after the event gets queued. This actually presents an undesired visual phenomena whenever a player collides with a player boundary.

When a player collides with a boundary, the point at which collision is registered must be when the two are overlapping. Collision handling in this case must forcefully reposition the player to the closest position outside of the boundary, lest it gets stuck within the boundary entirely. The 1 update delay means that players that continuously move into a boundary visibly jitter; this will have to be addressed in future development.

The solution, however, is not to re-design the Event Manager entirely as it works perfectly in all other regards. Instead we will have to combine the MovementSystem and CollisionDetectionSystem to detect both simultaneously. This would give us a PhysicsSystem that moves entities as far as possible by detecting the position of objects it could collide with, then generating a collision event whenever a collision does occur.

## 3.3 Collision Detection Optimizations

As we began implementing an increasing number of cards, projectiles, and ultimately the number of collision checks being made every update we began to experience serious performance issues. After testing CPU update times we determined that issues centered around the CollisionDetectionSystem, necessitating a solution to reduce the number of intense calculations made. At this stage of the project we would have one entity check collisions with every other entity on screen, regardless of its distance away or if the two entity types actually were allowed to collide in the first place. Ultimately we had O(n2) collision checks every update which is an extremely costly and inefficient process.

### Collision Byte Masks

Our first task was finding an efficient way to regulate what types of entities can collide with one another. For collision detection purposes each entity that can collide with other entities was assigned a “CollisionComponent” that stores a byte “mask” along with a collision radius and width.

Masks are useful when we apply the & operation between masks corresponding to two potentially colliding entities. By devising a masking strategy that is consistent we can quickly determine when two entities collide. If the & operation returns 0 then the two entities do not collide, thus skipping expensive calculations. Any other non-zero value signifies a collision. This is not the end of its usefulness, however, as byte masks can also be utilized to determine what type of collision occurs based on the value returned from the collision.

We will be analyzing how the four types of entities Player Bounds, Projectile Bounds, Players and Projectiles have their collision masks defined in the following paragraphs.

A bad example of byte mask utilization would be to assign player entities with a mask 0x01, projectiles of 0x01, and player boundaries 0x01, and projectile boundaries 0x02. This would give us the desired functionality for collision between players, as they can collide with players, projectiles, and player boundaries as each returns a non-zero value, whereas projectile boundaries return a 0. You can quickly see, however, that this byte mask is not very useful to us, as a collision including a player would always return the same value despite having different collision handling code based on the entity the player collides with.

Here it is important to understand that collision masks are not based on the type of entity they are attached to, but rather what each bit in the byte mask represents.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hex | Bits | Description | | |
| 0x01 | xxxx xxx1 | Collisions between Players and Projectiles | | |
| 0x02 | xxxx xx1x | Collision between Projectiles and Projectile Bounds | | |
| 0x04 | xxxx x1xx | Collision between Players and Player Bounds | | |
| 0x08 | xxxx 1xxx | Collision between Projectiles and Player Bounds | | |

###### Figure 2: Byte Mask Table and Explanations

#### Byte Mask Examples

Before moving on to examples it is important to note that in the current state of code two entities of the same type cannot collide, meaning that we also have conditional code that skips collision checks for entities of the same type.

Based on Figure 2, we can see the simplest collision mask belongs to the projectile boundary, as its sole responsibility is to check for collisions with projectiles. Its collision mask is therefore 0x02.

Projectile entities, on the other hand, have a mask of 0x03, which is essentially 0x02 + 0x01. A collision check (&) between 0x02 and 0x03 returns 0x02, and since two entities of the same type cannot collide we can guarantee we have a collision between a projectile and a projectile boundary.

Player entities, therefore, must have both 0x04 and 0x01, so that they can collide with player boundaries and projectile entities respectively. This of course gives player entities a collision mask of 0x05.

The collision mask 0x08 is the most difficult to grasp as this is a recent addition to the project. There came a need for projectiles to have the ability to collide with player boundaries to give them unique functionality. One option was to give these projectiles the 0x04 mask, but this would give these unique projectiles the same masks as players. We also wanted the ability for projectiles to collide with player boundaries to be unique to specific projectiles and not a default property of all projectiles.

The solution, then, was to utilize one of our remaining 5 bits for these types of collisions, thus guaranteeing that a collision check (&) resulting in 0x04 would signify a player colliding with a player boundary. Projectiles with the 0x08 collision mask thus collide with player boundaries exclusively. Projectiles that collide with player boundaries and players are therefore given a 0x09 collision mask.

Last but not least we have the player boundary itself, which requires the ability to collide with both projectiles and players, meaning it has a collision mask of 0x0C. This means that a collision check (&) that returns 0x08 guarantees we have a collision between a projectile and a player boundary, while a collision mask of 0x04 is a collision between a player and a player boundary. This information is invaluable as we no longer have to comb through the potential components of the entity to determine what type it is. We save a significant amount of time by immediately executing the correct functions for collision detection and handling when using byte masks in this way. “Genre Busters” uses a switch statement based on the byte masks returned from these byte mask comparisons.

Even with the great benefits observed from collision masks we were still left with O(n2) collision checks, as byte masks only serve to reduce a linear amount of checks. We therefore needed another optimization technique, leading to the discovery and implementation of Quadtrees.

### Quadtrees

A quadtree is a tree-like data structure where every node has a maximum of four children. Broadly speaking, the purpose of a quadtree is to split the playing field into progressively smaller regions of space as the density of entities increases. In other words, large spaces with few entities may be represented by only one quadtree node, while large spaces with many entities will instead be split into many smaller quadtree nodes.

#### Nodes

A node serves the purpose of managing references to entities that exist in its node, as well as any subnodes it has reference to. Each node represents a fixed area of the playfield, which is stored as a CenteredRectangle custom class.

CenteredRectangles are passed in the Vector3 center, as well as the height and width of the rectangle, both of which are floats. In the constructor, the CenteredRectangle class also calculates the Vector3 “topLeft” and “bottomRight” representing the positions of the rectangle their names suggest.

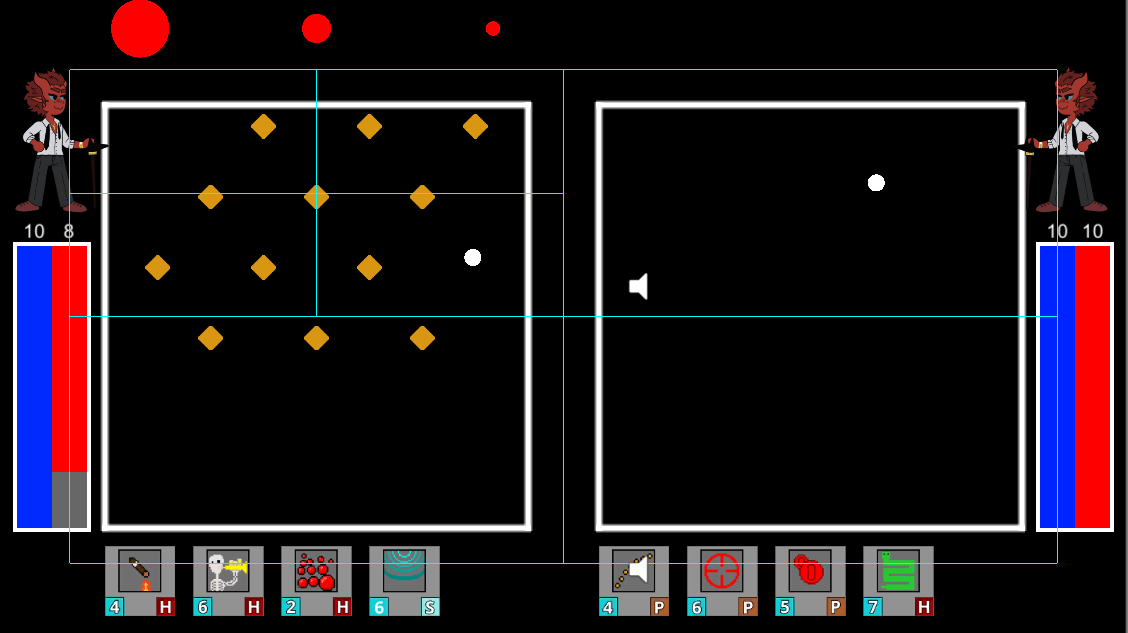
The CenteredRectangle also has two functions named Intersects() and Contains(). Intersects() checks to see if at least some portion of two rectangles overlap each other. While a useful function to have access to, this function is never actually used in the project. Contains() instead checks to see if the passed in rectangle is completely within the bounds of the rectangle calling the function. Contains() is the primary function used in our project, as this allows us to determine if an entity is fully contained within a node by adding only contained entities to the current node. If a node is to be added to a node and is not contained by any of the subnodes, it is instead added to the List<Entity> that the node stores, and is considered to be at that level of the quadtree.

#### Implementation

Our quadtree begins with the root node which is created and stored in the QuadTreeSystem. The QuadTreeSystem also stores a static Dictionary<int, QuadTreeNode> which is used for future reference by other parts of the code. Every update the QuadTreeSystem is recreated as it loops through each entity in the game that has both a CollisionComponent and a Translation. The QuadTree is only responsible for calling AddReference() on the root node as the rest of the logic is handled by the QuadTreeNode class itself.

Inside of the root node we begin by calling AddReference() passing in the first entity the QuadTreeSystem finds. Due to the fact that the root node is empty at this point we simply add the first entity to the List<Entity> within the node. This process repeats until the 5th entity is added to the root node, after which the root node splits into 4 sub nodes each representing 1/4th of the root node’s space; afterwards each entity that was stored inside the root node is re-added to the quadtree.

The quadtree handles adding new entities recursively, meaning that new entities need only be added to the root node by the QuadTreeSystem. AddReference() is called whenever a node Contains() that entity, but an entity is only ever stored in the quadtree once no rectangle Contains() it any more.



###### Figure 3: Quadtree Example (light blue)

In Figure 3 we see an example state of the game, with the QuadTree drawn in light blue. The outer perimeter rectangle represents the root node, as all collision logic is contained within this rectangle.

Before getting too much further into the example, it is worth highlighting that the three red projectiles on the top left of the screen are outside of this rectangle. All entities existing outside of the quad tree are simply added to the root node, meaning that these projectiles, alongside our invisible Projectile Boundaries, are located there. We made this decision because we realized that projectiles will only be outside of the quad tree temporarily and that we greatly benefitted from a smaller root node that focussed directly on the playing field.

Now, back to the example. We see a rectangle on the bottom left that is exactly 1/4th the size of the root node. This node is therefore a child of the root node, and contains 3 projectiles inside. As such it does not need to be reconstructed given it only has 3 entities inside of it instead of the required 4 in a quadtree.

The top left quadrant of the playing field, however, is packed with its 10 entities and was therefore split into 4 more subnodes. We will be focussing on the quadrant that the left player (white circle) is inside of for this comparison. First our collision detection checks for collisions with all entities that the rectangle Contains(), means that a direct collision check with only one projectile is made. We then move up to the next level as these projectiles are ones that exist on the boundaries of a quadtree node. We therefore perform 3 projectile collision checks, one for each projectile that runs along the very middle of the top left quadrant. Finally the red projectiles on the top left are checked with the player.

While a total of 7 collisions may seem inefficient we also should consider how many collision checks are made for the player in the top right quadrant. The quadtree structure has eliminated all collision checks barring the red projectiles meaning we only perform 3 collision checks for the second player.

In the brute force collision detection strategy we would have performed one collision check for every projectile in the scene, for a total of 30 collision checks between the two players. Our quadtree structure has instead reduced the number checks to only 10, a massive improvement considering our game intends to create many more projectiles on screen at a time.

# 4. Implementation

During production we used a loose agile approach for our project working in 2-3 week sprints and setting up a group of user story cards that we’d assign priorities to and work on. We had a few meetings where we exclusively brainstormed new ideas for cards. First semester our schedule was a bit inconsistent except for meeting for about 2 hours Tuesday and Thursday to work. Finally we crunched before the end of the first semester and got the framework of our project made, a skeleton that we could build upon. After that we started meeting in person 3 days a week and online on the weekends. We kept this momentum even once Covid-19 hit, meeting online every day but Monday for 2-3 hours a day. To make things easier to merge, we also only ever had one person implementing cards at a time while the other worked on implementing other mechanics.

## 4.1 Tools Used

For version control and file sharing, GitHub was an extremely important tool for us allowing us to work seamlessly online. Our development was staged in Unity, allowing us to access their ECS environment. One of the warnings we had starting our project is that too many people who wanted to make games for Capstone over relied on resources Unity gave, so we made sure to use as much of our own content as possible. Trying to only use Unity resources for code that was outside of the scope of our project like graphics rendering. Unity is paired with Visual Studio and uses C# so all of our code was written in that environment. Both of us already had some experience with C# so thankfully most of the learning process was focused on learning the intricacies of ECS.

Trello was what we utilized for organizing our priorities and tasks. At the end of every meeting we would take a look at the board and update or reorganize it depending on what our new priorities were. We utilized tags to mark each card’s priority as well as if it was on hold or in progress. It also helped us track how much work we had gotten done by dragging used cards to their respective finished category, having one for each semester and J-term. It was especially motivating to see in J-term we had gotten an equal amount of work done in the one month as the first semester, hitting our stride in full.

## 4.2 Gameplay

We decided to model the Bullet Hell part of the game after Toby Fox’s game “Undertale” where the player is in a box and has to dodge bullet patterns from the enemies they encounter. We gave each of our players one of these boxes and put them next to each other. This setup allowed for some very interesting mechanics, including bullets passing from one field to the other. This field approach translated quite well to the deck builder as well, allowing players to spawn cards on their own field to practice avoiding them.

Speaking of cards, we ended up deciding on a hand size of 4, allowing players to play the cards using 4 corresponding keys. We also placed constraints on the deck to encourage players to create unique play styles as well as not overusing certain cards. There are 15 cards required in a deck, with only 3 of the same card to be used, and only 1 super allowed in the deck. In a game, you would reshuffle your deck and begin playing again once your deck is completely out of cards. In total there are 30 cards to pick and choose from spread over 4 genres.

## 4.3 Testing

In Unity, everything is compiled and usable in real time, so we were able to jump into the game any time we needed to test any bugs we needed to fairly quickly. With any hard issues we had to solve, we used a series of console outputs to track the origin of the issue and then tracked our way from there. Something that ended up being extremely useful for us was the deck builder, which sped up our testing by granting us instant access to any scenario we wanted to create. For example, we theorized that the card “Gear Box” may have been causing processing issues, so we simply pulled up the card in the deck builder and played it several times to make sure it was the issue. Then as we changed code we could just open up the deck builder and make sure that the changes we made were effective. No need to put the card into a deck and then bring it into a game only to wait to actually pull it to see it used. Since everything works the exact same way in both the deck builder and the actual game it streamlined testing other functions as well, such as sound effects, visual effects, and collision testing.

Our general workflow consisted of creating a card, extensively testing it in the deckbuilder, and then pushing it to the repository. After getting 3 or 4 cards done, we would set aside time to use the cards in a full game and stress test them, trying to identify any issues as we played. Upon finding an issue we would take note of it and continue playing. Once we felt like we were done, we would go down the list and work out any of the bugs using the deck builder. When the fixes were done, we would go back and play a few more rounds to make sure no more issues arose.

## 4.4 Theme and Freelance Management

The title and theme of “Genre Busters” spawned from our inability to decide on a singular theme for the game. Did we want to make it based in a steampunk world and use airships as the players? No, that limited our options for bullet patterns too much. What about a fantasy setting? No, cards would be hard to differentiate between different factions. We repeated this pattern until we had a line up of genres to choose from and then decided that we would just make the genres themselves the factions. Then combining a primary and secondary deck would also make sense thematically, being a combination of two genres.

###### Figure 4: Finn (Sci-Fi) and Tansy (Fantasy) Character Art Alongside Genre Specific Cards

That left us with a question however, what was the best way to represent these different factions? The decision was to have a character represent each genre. With little art experience, both of us were at a loss as to how to bring these characters to life until we started reaching out to freelance artists to help us get the job done. The artist who ended up helping was Karley Elizabeth Wise. With her help we designed a character for each faction. Granting us not only plenty of concept art, but also animated sprites for every character. From there we moved into territory neither of us had really touched before, voice acting. We put out a Google Form on Twitter and Discord which piqued the interest of some voice actors to come by and volunteer. Working closely with the voice actors was important. Sitting in a call with them playing hot or cold with the voicelines until we had several takes of every required line. Then the recording was cut up in Audacity and imported into Unity to be implemented.

# 5. Future Work

First and foremost, after seeing the potential of what this project could become, our plan is to expand the scope of the project beyond that of a Capstone project. We both plan on continuing development post graduation and have a list of additions that we think would make it a more complete product.

## 5.1 Options

The ability to customize controls, audio, and additional settings is incredibly important for a competitive game. It allows players to customize the play experience to match their tastes much more accurately. Players should never be losing games due to a failure on the game’s part to adapt controls to their needs.

## 5.2 Network Play with Rollback Netcode

At the moment, the only way to play online with your friends is by utilizing 3rd party software such as Parsec to connect remotely to another computer. We would love to make Network play available with rollback netcode. This is a technique that simulates frame inputs without delay rather than waiting for an input to be received from the other signal. This creates nearly seamless gameplay for both players as they challenge each other in real time. At this point, rollback netcode is a necessity for any modern game trying to be competitive in any way shape or form.

With netplay would also come features such as leaderboards and matchmaking ranking allowing players to climb the competitive ladder and prove their worth.

## 5.3 Visual Update

Most of the assets were drawn by the two of us as we mostly just needed things to be functional and put most of our budget towards the character art. As we continue forward however, getting better defined art for cards, projectiles, borders, menus, and backgrounds would be fantastic. It is just as important for a game to look good as it is to feel good mechanically as visuals draw the attention of players. However, with this overhaul we’d be keeping in mind the importance of keeping the visuals somewhat simplistic so it’s easy to determine what is going on. With so many projectiles and effects on the screen it’s imperative to be able to determine what is happening with all of the objects on the screen or else it would be incredibly hard to play the game effectively.

## 5.4 AI Opponents

Programming some kind of AI opponent that can be given a deck is also a fairly important step to take.Giving players something that fights back that isn’t another player is certainly important for practice. This would let players test combos of cards against an opponent with the pressure of an actual game, but without the consequences of losing to another player. This would also be a necessary step if any single player story mode was to be added, allowing the player to go through a story and seeing a variety of decks and play styles along the way.

## 5.5 More Cards and Genres

At the moment we have a total of 30 cards split between 4 genres. In the future we’d love to expand this as much as possible. More cards and genres means more interesting deck options and more opportunities to make new playstyles, which is the core of our game. Expanding the amount of card interactions is exciting as well. One of the most interesting parts of testing the game came from testing out new cards in the card builder and seeing how they interacted with other cards.There are plenty of ways to create a win condition, but the more we can create the more diverse the game will get.

# 

# 

# 

# 

# 

# 

# 

# 6. References

Code Monkey “Unity DOTS / ECS Tutorials” *Unity DOTS / ECS Tutorials*. Accessed October 2019.

<https://www.youtube.com/playlist?list=PLzDRvYVwl53s40yP5RQXitbT--IRcHqba>

C00pala “Unity UI - Scroll Menu Pt 1-3” *Unity UI - Scroll Menu*. Accessed January 2019.

<https://www.youtube.com/watch?v=TUSvupeYZv4>

Brackeys “Introduction to AUDIO in Unity” *Introduction to AUDIO in Unity.* Accessed March 2019.

<https://www.youtube.com/watch?v=6OT43pvUyfY>\

## 6.1 Special Thanks

Karley Elizabeth Wise

Character Design and Art

<https://twitter.com/karleydraws>

Shishishena

Voice Actor for Stea

<https://twitter.com/shishishiena>

Natalie Enzmann

Voice Actor for Tansy

<https://twitter.com/AnEnzmann>

Zachary Gerken

Voice Actor for Rory

<https://www.linkedin.com/in/zachary-gerken/>

Dreamy Mist (Kairi)

Voice Actor for Finn

Philip Rader

Resident ECS expert, EventManager designer, and volunteer gameplay tester

<https://github.com/pjrader1>

<https://www.linkedin.com/in/philip-r-524635b6/>

# 7. Glossary

**Collision Byte Mask**: A specialized byte variable that is used to categorize collision detection in “Genre Busters”. Each entity with a Collision Component is assigned a collision mask. Masks are compared in the Collision Detection System, where masks that can collide will return a non-zero number if compatible. This non-zero number is also used to determine which type of collision occurs.

**Component**: A struct that contains gameplay related data. Components are very specific, meaning that an entity should have one component for each aspect of game data it stores.

**Entity**: A generic object that is solely responsible for storing Components. In “Genre Busters” entities are created by calling a static “Create()” function for each type of entity. In this constructor we attach all components that are fundamental to the functionality of that entity.

**Entity Manager**: An object provided by Unity ECS that contains references to all entities and provides methods to modify these entities directly.

**Entity-Component-System Framework**: Also referred to as ECS, this is a data oriented framework that uses entities, components, and systems for managing and updating the current game state.

**Event Manager**: A custom event system that directs event creation and storage, event listening, and event handling. “Genre Busters” uses this event manager to handle gameplay logic in a modular fashion, as calls to external code are abstracted through event listening.

**Modular Design**: A programming technique that prioritizes separating program functionality into independent, interchangeable modules. In “Genre Busters”, the ECS framework allows us to disable and enable systems at will, thus changing how the program executes without having to hard code in any adjustments to how the game functions.

**Quad Tree**: A tree structure composed of “nodes” that are used to represent the 2D playing field of “Genre Busters”. Each node has a maximum of four references before being split into sub nodes, with each node being responsible for referencing each entity that exists within its defined region.

**System**: A constantly updating loop that operates on all entities that meet its filtering requirements. Systems are responsible for updating the component data of entities it filters. Systems are very specific, meaning that a system should only affect entities for one type of continuous calculation.