Gamifying Evolution

Barrie D. Robison

stuff si junk

# Introduction

Biological evolution is a critically important process that frames much of our understanding of the natural world. Unfortunately, public misconceptions about evolution are extremely common. In fact, the mere acceptance of evolution as a real process is uncommon in many demographic groups, and is particularly low in the United States Miller et al. (2021) , and educators often face direct opposition to teaching evolution in their classrooms. The many opponents of evolution also intentionally spread demonstrably false information and have powerful lobbying capabilities. Evolution is a concept that can be difficult to present in a formal learning setting but is very understandable when directly observed. Engaging games and simulations could therefore be particularly beneficial in helping to reach students and in correcting pervasive public misconceptions about the discipline.

The “gamification” of biological evolution also has potential implications for game designers. Encoding evolution as a game mechanic creates a system that adapts to player strategy, produces unexpected results and emergent properties, and inherently scales difficulty in response to player skill. These features have the potential to increase replayability and decrease development effort on game balancing.

Game designers have long recognized the potential of evolution to SELL games. Unfortunately, the most successful commercial games that emphasize evolution (for example, Spore™, Evolve™, and Pokemon™ Evolution) aren’t evolutionary at all. Most of these games insert player choice into the “evolutionary” process, allowing for example, the player’s character to choose whether to “evolve” a beak, or a lightning attack. This approach inadvertently represents intelligent design, a pseudoscientific reskinning of young earth creationism. These games, which use evolution as a marketing angle, potentially exacerbate the public’s confusion about evolution and reinforce existing misconceptions.

There have been a few games that avoid this trend and present reasonably accurate models of evolution. *Intelligent Design: An Evolutionary Sandbox* (available on Steam) is an evolutionary game in which players create species that are placed in an evolving ecosystem - once released, the creatures continue to evolve on their own. *Niche* (available on Steam) is a turn-based game in which the player breeds members of a species to avoid extinction. Although they have not been released commercially, the game *NeuroEvolving Robotic Operatives* (NERO) and a spin-off, *EvoCommander*, are also examples of games that incorporate evolution.

## Polymorphic Games

Polymorphic Games is an interdisciplinary game studio at the University of Idaho that focuses on the gamification of evolution. Polymorphic Games employs undergraduates from a wide range of disciplines (including Computer Science, Virtual Technology and Design, Biology, English, Theater, Education, Business, and Music) to create video games that incorporate evolution as a core game mechanic. The studio is motivated by the idea that evolution can make games more compelling and more fun because the game adapts to the player. We argue that this approach represents a positive combination of STEM engagement and informal learning. We seek to make a compelling game ini which players are inherently motivated to learn the underlying evolutionary concepts in order to succeed at the game objectives.

Polymorphic Games’ first project was Darwin’s Demons, a “space shooter” game inspired by Space Invaders. In Darwin’s Demons, the player battles an evolving population of aliens (Figure 1). The enemies’ traits are encoded by a digital genome of real numbers, and the fittest enemies reproduce to create the next generation causing the population to adapt to the player. The name of the game was inspired by the Darwinian Demon metaphor, which describes an organism that can evolve without constraint. In Darwin’s Demons, each trait can evolve independently (there are no evolutionary tradeoffs), meaning that eventually the player will lose. The objective (like in Space Invaders) is to beat the high score. Players can choose from a wide range of ships and secondary equipment, which increases player interest and serves to drive evolution along different trajectories [28]. Players can upgrade their equipment between generations of enemies, which are the evolutionary equivalent of waves in a typical video game. The up-grade screen includes data on the evolutionary trajectory of the game (Figure 2). The game saves the genome of every creature as a .csv file, allowing experimentation and in-depth analysis of the creatures’ evolution in response to player strategy. Development on Darwin’s Demons began in May 2016, and the game was released for sale on Valve’s Steam platform on February 13, 2017 (the Monday after Darwin Day). To our knowledge, Darwin’s Demons was the first commercially released game to fully feature evolving enemies.

Polymorphic Games’ second commercial release was Project Hastur, an evolutionary tower defense game. Project Hastur features a 3D game environment and enemies are procedurally generated from a digital genome, producing a wide variety of morphologies, behaviors, capabilities, and other game traits. In Project Hastur, the player can play through a campaign that features 14 unique maps that unlock a series of defensive capabilities. Project Hastur also features experiment mode, in which the evolutionary parameters can be altered. This feature has numerous benefits – it allows exploration of the system on which the game is built and allows the player to autonomously test “cause and effect” as they adjust parameters.

*Development:* Project Hastur is a tower defense video game developed by students in the Polymorphic Games studio at the University of Idaho. The game features a campaign of 16 original game maps, as well as an experiment mode (described below) that allows the game to serve as a simulation model. Project Hastur was released on the Steam game site on Feb 12th, 2019. The unique feature of Project Hastur is that the enemies in the game model a finite population defined by digital genomes of quantitative traits.

*Gameplay:* In Project Hastur, the player must defend their base against waves of enemies called the Protean Swarm. The player defends their base by placing defensive towers in strategic locations, and the towers each have different capabilities, strengths, and weaknesses. In campaign mode, the player unlocks new game maps and defensive capabilities as they proceed through the story. Each map is defined by victory conditions that involve defeating a set number of enemy generations or building their defenses to specific criteria. Enemies appear in “waves”, a classic trope of tower defense games in which the developers script the number and type of enemies that appear over the course of the level. In Project Hastur, however, the enemy waves are part of a generational evolutionary model - each new enemy wave is comprised of the offspring of the most successful parents in the previous wave. In this way, we programmed Project Hastur to become an evolutionary tower defense game in which the enemy population adapts as the game proceeds.

The defensive capabilities of the player include four different categories of tower with qualitative differences in how they interact with the physics engine of the game. These include kinetic, ice, flame, and acid based towers, with each category including 4 different versions of upgrades. In addition, the player can access and upgrade two different abilities that are activated on demand with a cooldown - robotic minions and airstrikes.

The “gamification” of evolution can potentially help expose players to real evolutionary models in an engaging way. However, there are several tradeoffs between fun and realism that should be addressed when developing an evolutionary video game. In this paper, we discuss the lessons learned from developing and releasing games that feature evolution as a core mechanic, and the design decisions inherent to balancing fun gameplay with scientific realism. We will primarily use gameplay data from Project Hastur to demonstrate how the balance between realism and fun has in-game consequences.

# Evolution as a Game Mechanic

Here, we explain our approach to the gamification of evolution using four commonly taught components of the evolutionary process: Variation, Inheritance, Selection, and Time. When these components are present, the evolutionary process leads to Adaptation.

## Variation

Pretty much every video game features some kind of variation. Even the first arcade games, such as Space Invaders, Asteroids, Centipede, and Tempest, relied on variation among enemies or obstacles to add interest and challenge. In Space Invaders, for example, there three types of aliens in each wave, along with the occasional bonus flying saucer. Centipede and Tempest featured categorically different enemy types that varied in how they looked and in their capabilities. Variation is a central component of game development and is a big part of what makes video games exciting. The difficulty in video games is often increased over time by introducing new variations of enemies, changing the enemies’ capabilities (like speed or fire rate) over time, or increasing the number of enemies. One potential challenge, though, is that these variations are introduced as part of a script that is programmed and balanced by the game developers. Beating a game often means memorizing the script.

**A central theme here is that game difficulty is often controlled through variation of enemies.**

When implementing evolution as a game mechanic, we encode variation of enemy traits using a digital representation of a genome. This variation can be classified into two categories, Standing Genetic Variation and Mutational Variation. Standing genetic variation is the genetic differences among individuals at the start of the game. This can be adjusted as design principles dictate, but a general rule of thumb is that more variation in the initial population corresponds to a faster rate of adaptation in the early phases of the game. Were the game to proceed solely using the standing genetic variation present at the beginning, the combination of selection and random genetic drift would result in a genetically and phenotypically homogeneous population in rather short order. The resultant lack of variation in the population would quickly engender boredom in the player.

Mutational processes can be used each generation to introduce new genetic variation upon which selection can act. This is akin to adding more “fuel” to the process of adaptation, sustaining the evolutionary game play much longer than one would observe with only standing genetic variation. We typically use a mutational step between generations, mutating the digital gametes passed on by the parents to their descendents in the next generation. It is here that we must acknowledge a tension between designing a game for fun and creating an accurate evolutionary simulation. Mutation rates in biological populations are well studied, and rates range between 1 in 10^6 and 1 in 10^9 per locus. These rates are much too low to make for very compelling game play. In our games, we typically use mutation rates that are much higher than those found in nature. This accelerates the infusion of mutational variance into the population.

It is important to note that the amount of genetic variation in the population tends to scale with game difficulty. In our previous games, we have used the rate and effect size of new mutations as one of the key differentiators between modes of difficulty.

Example 2: Using mutational processes to scale difficulty (Darwin’s Demons)

The first design decision regarding variation is how much variation should there be in the initial population. This is known as standing genetic variation. That is should the initial population of enemies be identical, with variation only occurring after the first generation or should there be variation in the initial population. And if there is variation in the initial population should it be relatively limited or extreme.

No standing genetic variation means that the evolutionary process starts slowly, it takes multiple generations to build up enough variation for selection to work on. This can be beneficial in a long-term game, especially one where the player takes a while to get situated, e.g. collecting resources to build a base. Low standing genetic variation also means that evolved enemies will be more obvious when they appear, they player will have gotten use to seeing the same enemies and the appearance of new, tougher enemies will be a bigger shock.

Increased standing genetic variation will drive the initial evolution process much faster an is more appropriate to a faster paced, e.g. arcade style game. If the standing genetic variation is very high the way the player observes evolution can even be reversed, first they see a wide range of enemies with many different traits and appearances, but selection will quickly focus on the most successful enemies, reducing the amount of variation and making the game much harder quickly.

Beyond the initial standing variation the mutational step adds variation in every generation. The rate of mutation is defined by two factors: the number of ‘genes’ that are mutated and the size of the mutations. The number of genes mutated is usually somewhat stochastic and defined in terms of the total number of genes, i.e. genes are mutated with a probability defined as n/(total number of genes) with n being 1-3. The size of the mutations is usually defined as a Gaussian with a mean of 0 and a variance that depends on the scale of the meaningful genetic values. That is if a gene’s values range from -1 to 1 the variance of mutation should be much smaller then if gene values range from -100 to 100.

In general, smaller variation is biologically much more realistic, but higher variation will drive evolution faster be introducing more variation for selection work with. It is possible to have too much variation, if the amount of mutation swamps inheritance, i.e. if there’s so much mutation that offspring don’t meaningfully resemble their parents, then evolution stops working. There’s no benefit to selecting more fit parents if mutation is so rampant it destroys the parents’ evolutionary advantages.

*Controlling Variation in Project Hastur:* Each individual in the enemy population is defined by a digital genome of real numbers. There are 80 genes in the genome. Each Protean is diploid, such that its final trait value is calculated as the sum of the genetic values for the given locus on each chromosome. While all of these genes are used by the game engine to render a wide variety of visually distinct game enemies (see Figure x), some of the genes also affect Protean traits that are relevant to game play. These are the traits upon which the analyses in this paper were performed, and are defined in Table x.

*Variation:* When the game begins, Protean trait values are instantiated using the starting values in table x with some variation drawn from a Gaussian distribution. This creates the standing genetic variation in the population upon which selection will act. When offspring are created (described below), additional mutational variance is introduced. In campaign mode, the per locus mutation rate and effect size vary depending on the map’s degree of difficulty - more mutational variance causes the game to become much more difficult. In experiment mode, mutation rate and effect size are determined by the user. MUTATION RATE (u) is defined as the per locus probability that the genetic value of the locus will be changed by a number drawn from a Gaussian distribution with mean zero and standard deviation defined by the variable MUTATION EFFECT SIZE (e).

## Inheritance

**The addition of an inheritance model is the primary differentiator in our games that creates adaptation.**

In most games, there is no relationship between enemies within a wave or between waves. They are instantiated (spawned) with developer defined traits at a specified rate, location, and time (or level or wave). This is the key difference in the games that our lab creates. In our games, the enemies traits are specified by digital genomes that they pass on to their offspring through a form of reproduction. If the enemy has a speed value of 4, that means that when it reproduces, its offspring inherit that speed value of 4. Most of the games we work on specify about 40 to 80 “genes” in the enemy genome, each of which can influence one or more game traits such as speed, fire rate, morphology, behavior, health, armor, resistances, etc.

Inheritance is that passing of genes from parents to offspring. This can be done asexually, one parent is sufficient to create one or more offspring, mimicking the division process of bacteria. The entire parent genome is copied and then mutated to create the offspring’s genome. Alternatively, inheritance can be done sexually with two parents contributing to the genome of the offspring. There are a number of ways of combining the genomes of two parents to create an offspring. The most common approaches are one-point crossover in which both genomes are ‘cut’ at the same point and the ends combined; two-point crossover in which a section of one parent’s genome is spliced into the other parent’s genome; and uniform crossover in which genes are stochastically drawn from either parent creating a uniform ‘mixing’ of the genes. In sexual reproduction it is common for each ‘mating’ to produce two offspring. With techniques like two-point crossover this means that there is balance between the parents - each offspring is majority from one parent, with a portion of the other parent’s genome spliced in.

Generally, with sexual reproduction the individuals are treated as hermaphrodites, so any two individuals can serve as parents. Gonochoric (gendered) populations have potential advantages with respect to the evolutionary process, and introduces the option for a wide range of model variation in mate selection, but add an additional level of complexity to the model.

Sexual vs asexual. Hermaphrodites? Encoding the digital genome (quantitative trait model vs discrete alleles)

*Protean Traits:* Each individual in the enemy population is defined by a digital genome of real numbers. There are 80 genes in the genome. Each Protean is diploid, such that its final trait value is calculated as the sum of the genetic values for the given locus on each chromosome. While all of these genes are used by the game engine to render a wide variety of visually distinct game enemies (see Figure x), some of the genes also affect Protean traits that are relevant to game play. These are the traits upon which the analyses in this paper were performed, and are defined in Table x.

*Inheritance:* Each “wave” of Proteans in Project Hastur is actually a discrete generation of enemies created from the previous generation using a tournament selection algorithm. Proteans that are selected as parents (see Tournament Selection, below for the algorithm) each contribute one of their chromosomes, determined randomly, for each locus. Project Hastur uses a free recombination model (each locus acts as its own chromosome), but the game can specify a customized matrix of recombination frequencies among loci if the user desires. The chromosomes passed to the offspring are each passed to the mutation algorithm, and each locus has a chance (specified by u) to receive a mutation. Once the new offspring’s genome is instantiated, it is passed to the population for the next generation. Project Hastur records the unique ID numbers for each parent of every Protean. Proteans reproduce sexually, but are encoded as hermaphrodites.

On game maps that contain civilians, Proteans can also reproduce using an asexual model. In this case, if a Protean kills a civilian the Protean clones itself. This cloning process produces an offspring that is genetically identical to its parent during the gameplay phase itself (rather than waiting until the end of the wave). The number of clones produced by this process depends on the parent’s size (COLLIDER SURFACE AREA). Smaller Proteans produce more clones than larger Proteans.

The model of reproduction and inheritance can have effects on gameplay.

## Selection

Selection occurs when there is a correlation between a trait (say… hit points or movement behavior) and **Fitness**. In biological populations, Fitness is usually defined as the number of offspring produced by a given individual. We often measure proxies for fitness in empirical biology that are typically predictive of (or at least correlated with ) Fitness, such as seed set in plants, survivorship, or number of eggs in a nest. In our games, we can make explicit linkages between performance and fitness using Fitness Functions.

For example, in Darwin’s Demons the enemies accrue Aggression by moving downward on the screen. The closer they get to the bottom, the higher their Aggression score. if they move past the player and off the bottom of the screen, their Aggression score gets an even larger boost. After each wave, the enemies with the highest Aggression score also have the highest probability of mating and producing offspring. Thus, we have created an explicit link in which Aggression is correlated with Fitness (number of offspring). Our games often have more than one of these Fitness Functions. For example, the Protean Swarm in Darwin’s Demons can also increase their mating chance with Accuracy (which measures how often they shoot at the player and how close they came to hitting the player) and Lifetime (which measures how long they last on the screen) .

Project Hastur specifies two fitness functions that determine the probability that Proteans will be selected to reproduce at the end of each generation. The first Fitness Function, called BASE FITNESS specifies the closest distance to the player base that was achieved by a given Protean. Should the Protean get right next to the base (a distance of zero), then BASE FITNESS includes the DAMAGE the Protean does to the base. Lower BASE FITNESS values increase the probability of reproduction when the wave ends. TOWER DAMAGE is the other fitness function used to calculate whether an individual will reproduce. In this case, the game sums the total DAMAGE done by the Protean to any defensive structure. TOTAL FITNESS is calculated by subtracting TOWER DAMAGE from BASE FITNESS.

Selection itself should include a stochastic component, favoring more fit individuals, but leaving opportunities for less fit individuals to be selected to maintain diversity within the population. Two common approaches are roulette wheel selection and tournament selection. Roulette wheel selection is ‘fitness proportional’ the probability of being selected is proportional to an individual’s relative fitness. It is modeled as a roulette wheel in which each individual gets a roulette wheel ‘slice’ whose size is proportional to their fitness. Tournament selection is ‘rank based’, to pick a parent a small (typically 3-5) subset of the population is picked at random and the highest fitness individual in that subset wins the ‘tournament’ and the opportunity to reproduce. Individuals can be picked for multiple tournaments, so more fit individuals typically have several opportunities to reproduce and less fit individuals are unlike to win many, if any, tournaments. Rank based approaches, like tournament selection, have the advantage that if the population consists of individuals with very similar fitnesses, rank still produces a gradient to select on.

*Tournament Selection:* At the end of each generation, we use tournament selection to identify individuals that will serve as parents. To do this, the game randomly selects a sample of Proteans from the previous generation. The number of individuals in this sample is a controllable parameter in experiment mode called TOURNAMENT SIZE. The individual with the best (lowest) value for TOTAL FITNESS is selected as a parent. A new tournament is then conducted to select the mate. Reproduction then occurs as described above and a single offspring is passed to the population for the next generation. This process is repeated until the population in the next generation has a number of individuals in it equal to POPULATION SIZE.

Example: On vs off (adaptation vs drift)If we turn off the fitness functions, then after each wave the enemies undergo random mating. Adaptive evolution does not occur, but the enemy traits might still evolve randomly because of Random Genetic Drift.

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

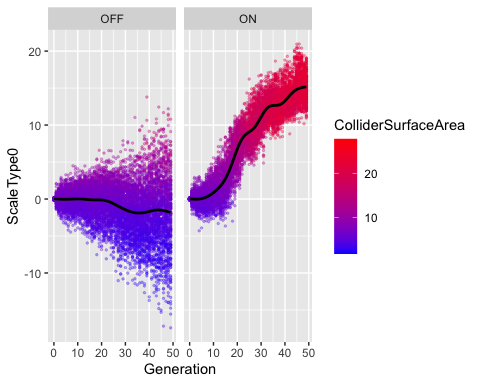


Figure 1. Values of the *ScaleType0* gene in two replicates of Project Hastur over 50 generations of game play using a standardized defense of only Autocannon Towers. Each point represents an individual game enemy, and is colored according to Collider Surface Area, which measures body size. *ScaleType0* controls the size of the main body of the individual. LEFT: Fitness functions are turned OFF, resulting in the accumulation of mutational variance over time, but no change in mean size of the population. RIGHT: Fitness functions are turned ON, resulting in directional selection for increased body size.

In Figure 1, we present an example of how the combination of inherited variation and selection via fitness functions can produce directional selection, in this case for larger enemies (which tend to have more Health. Extending this example, we can show that the patterns of trait adaptation in Project Hastur are in response to game conditions and strategies institute by the player. The data presented in Figure 1 used a standardized arrangement of Autocannon turrets on the same game map (“Crater Mountain”). The Autocannon turret relies on a raycast for targeting, firing numerous projectiles that essentially do not miss. We ran a total of 9 replicates each under these conditions (with and without fitness functions) using the Autocannon. We then ran additional replicates using the identical arrangement of towers using the Chip Shredder, a turret that fires a single, slower projectile that relies on collision detection. Figure 2 shows the pattern of trait evolution we observed under these conditions.

`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'

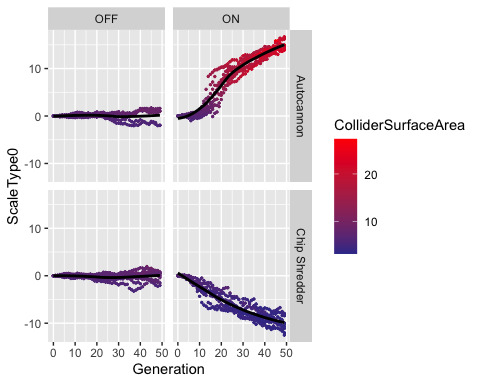


Figure 2. Mean values of the *ScaleType0* gene in 9 replicates of Project Hastur under four gameplay conditions (36 total replicates). Each point represents the mean value of a replicate at each generation, and is colored according to Collider Surface Area, which measures body size. When Fitness Functions are inactive (LEFT column), mean trait values change randomly because of genetic drift. When Fitness Functions are active (RIGHT column), the Autocannon towers select for larger individuals (which have more Health), while the Chip Shredder towers select for smaller individuals (which are faster and can evade projectiles).

We can test whether the observed changes in game traits are caused by selection by calculating the selection gradients within each generation. Selection gradients are standardized measures of the strength of selection, calculated by measuring the slope (Beta) of the relationship between a particular trait and fitness. In this case, we measure fitness by directly calculating the number of offspring for each individual. An estimate of Beta=0 indicates no selection. Positive values of Beta indicate directional selection for increased trait values, and negative values the opposite. Figure 3 shows individual estimates of Beta for each replicate in each generation. Note that the population size of 200 individuals limits our power to detect signficantly different values from 0, but combining replicates reveals a general pattern of positive directional selection on the ScaleType0 gene by the Autocannons, and negative directional selection on the ScaleType0 gene by the Chip Shredders.

`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'

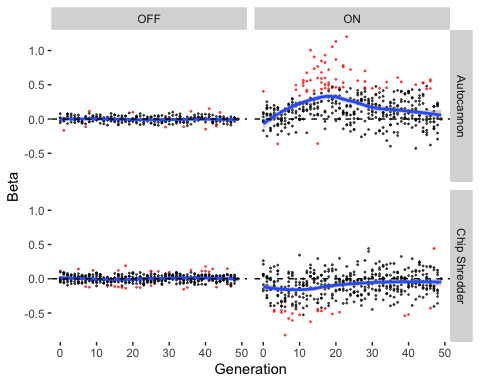


Figure 3. Estimates of the selection gradient, Beta, for the ScaleType0 gene in Project Hastur. Red points indicate estimates of Beta that are significantly different from 0 in individual tests, while the blue line indicates the Loess estimate of Beta across the entire experimental condition.

As Figures 2 and 3 show, the type of tower used in the defensive strategy has a dramatic effect on the evolutionary trajectory of the game enemies. This example is a simplified experiment intended to isolate a single trait’s response to manipulation of a single variable. The enemies in Project Hastur have more than 20 traits, and the defensive combinations available to the player are inumerable. This creates a great deal of potential variation in gameplay.

In project Hastur, we often observe variation in the evolutionary outcomes even when gameplay conditions are identical. In Figure 3, we show a heatmap of the standardized trait means across 9 replicates of the exact same game conditions. The first two rows of the heatmap (replicates 2 and 3) are a distinct cluster representing an outcome dominated by small and fast enemies with a large sight radius. The remainder of the replicates produced some variation of the large, slow “tanks” that evolve to absorb most of the hits from the Autocannons.

[1] "./JJSP1.csv"  
[1] "./JJSP2.csv"  
[1] "./JJSP3.csv"  
[1] "./JJSP4.csv"  
[1] "./JJSP5.csv"  
[1] "./JJSP6.csv"  
[1] "./JJSP7.csv"  
[1] "./JJSP8.csv"  
[1] "./JJSP9.csv"

Adding missing grouping variables: `Tower`  
`summarise()` has grouped output by 'Generation', 'Tower', 'replicate', 'Gun',  
'Evolution'. You can override using the `.groups` argument.

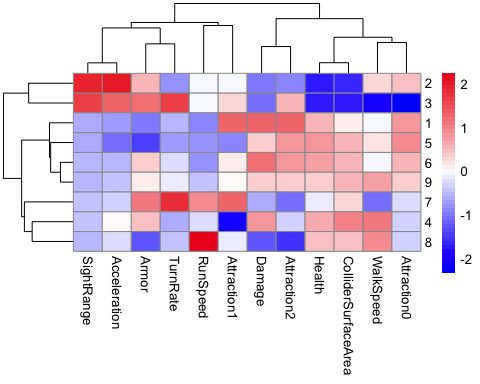
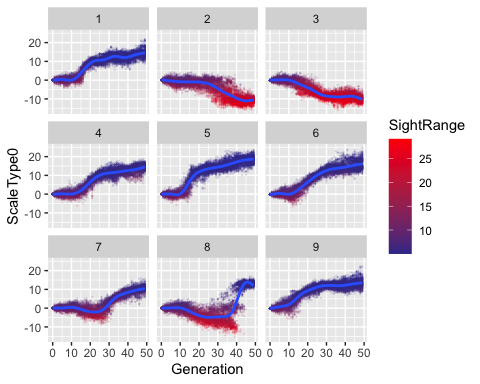


Figure 3. Heatmap of traits (columns) for 9 replicates (rows) of Project Hastur using the Autocannon conditions from Figures1 and 2. Each cell represents the Z-score of that trait, with each unit change representing one standard deviation from the overall mean of zero.

This variation in evolutionary outcomes is inherent to the stochastic nature of the evolutionary model. New mutations occur randomly, and the rather small population sizes cause random genetic drift. Even within a single game replicate, we can sometimes observe subdivision of the population as evolution drives individuals toward two separate peaks in the fitness landscape.

ggplot(AutoCiv, aes(x=Generation, y= ScaleType0))+  
 geom\_point(aes(x=Generation, y= ScaleType0, color = SightRange),  
 size=0.1, alpha = 0.1)+  
 geom\_smooth()+  
 scale\_color\_gradient(low = muted("blue"),  
  
 high = "red",  
  
 space = "Lab",  
 na.value = "grey50",  
 guide = "colourbar",  
 aesthetics = "colour")+  
 facet\_wrap(~replicate)

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'



For example, when we plot the individual values of the ScaleType0 gene and color by the Sight Range trait, we observe that replicates 2 and 3 are clearly evolving towards smaller individuals. Further, the individual patterns of evolution among the replicates differ in their temporal dynamics. The most extreme example is replicate 8, in which two sub-populations are clearly present between generation 25 and 40, after which point the larger individuals out-compete the smaller and the mean size of the enemies changes dramatically.

## Time

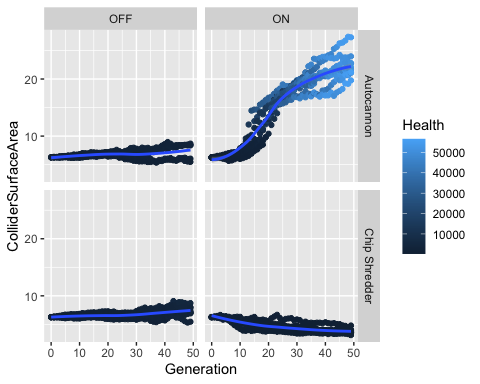
Generational model = waves

Continuous model

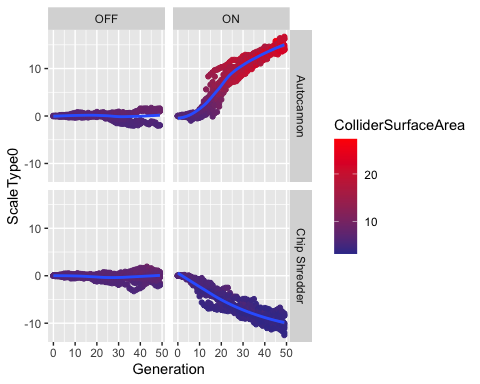
The concept of time in video games is often defined in terms of waves or levels. Both of these terms imply a progression toward the game’s goal, along with a corresponding increase in difficulty. In evolutionary games, time is specified in terms of generations. A wave of enemies begin the game, the player defeats them, and then the next wave is created with Inheritance using a mating function specified as described above. As the generations (waves) proceed, the enemies with traits that are best able to optimize the fitness functions have more offspring, and the population adapts to the game play conditions. Most of these conditions are created by player choices and playstyle, and thus the enemies adapt to the player. Difficulty increases organically and repeated gameplay often creates novel adaptive solutions to the same play style.

To make an interesting and challenging game, evolution must occur within the rather limited time frame of gameplay. In nature, biological evolution can take thousands to tens of thousands of generations to result in obvious changes. This is much too long for most games, necessitating sacrifices in realism of the model. Potential changes to model parameters include increasing the amount of genetic variance and selection pressure, both of which can speed up the evolutionary process. But there has to be a careful balance. Too much variance and advantageous traits can be lost to mutation. Too much selection and the population will converge on a single genotype, making the game less interesting because all opponents are identical and limiting variation for future evolution.

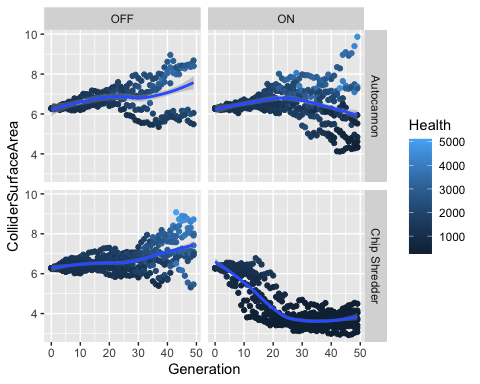
`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'



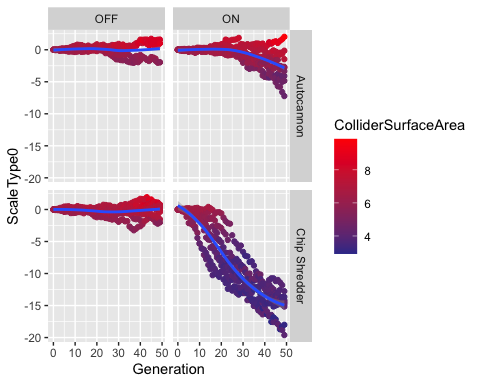
`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'



`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'

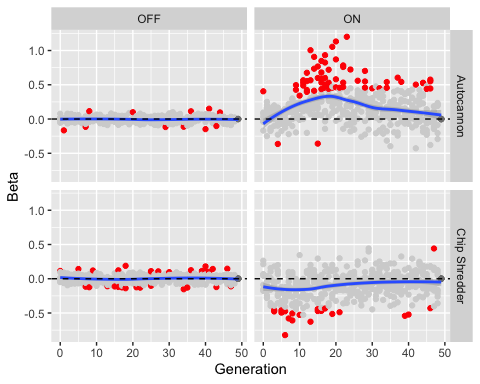


`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'



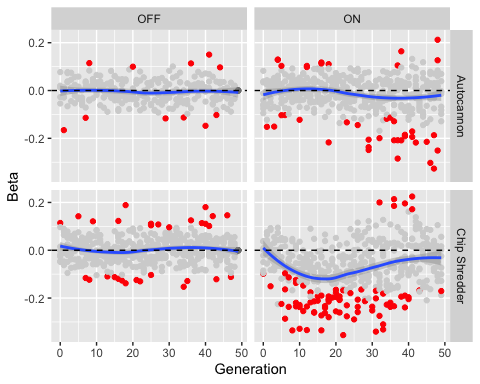
traittemp<-all\_the\_data%>%  
 select(Generation, relfit, file, Evolution, Gun, Civilians, ScaleType0)%>%  
 group\_by(Generation, file, Evolution, Gun, Civilians)%>%  
 mutate(scaleST0 = scale(ScaleType0, center = TRUE))%>%  
 mutate(scaleST02 = scaleST0\*scaleST0)  
  
Gradients <- traittemp %>%  
 group\_by(Generation, file, Evolution, Gun, Civilians) %>%  
 do({  
 model <- lm(relfit ~ scaleST0 + scaleST02, data = .)  
 data.frame(  
 Beta = coefficients(model)[2],  
 PB = summary(model)$coef[2, 4]  
 )  
 })  
  
Gradients <- Gradients %>%  
 mutate(sig = if\_else(PB < 0.05 , "Y", "N"))  
  
GunFit<- Gradients %>%  
 filter(Civilians == "OFF")  
  
CivFit<- Gradients %>%  
 filter(Evolution == "OFF")  
  
ggplot(GunFit, aes(x=Generation, y = Beta))+  
 geom\_point(aes(color = sig))+  
 geom\_smooth()+  
 scale\_color\_manual(values = c("lightgrey","red"))+  
 geom\_hline(yintercept=0, linetype="dashed", color = "black")+  
 theme(legend.position = "none")+  
 facet\_grid(Gun~Evolution)

`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'



ggplot(CivFit, aes(x=Generation, y = Beta))+  
 geom\_point(aes(color = sig))+  
 geom\_smooth()+  
 scale\_color\_manual(values = c("lightgrey","red"))+  
 geom\_hline(yintercept=0, linetype="dashed", color = "black")+  
 theme(legend.position = "none")+  
 facet\_grid(Gun~Civilians)

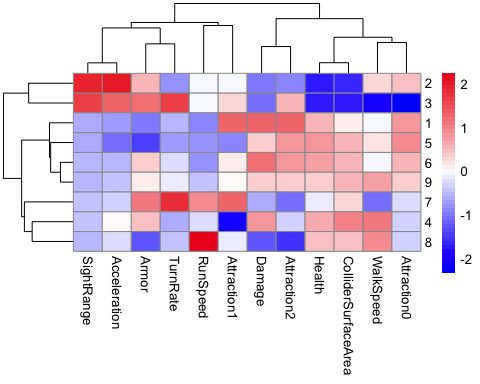
`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'



Hidden fitness functions

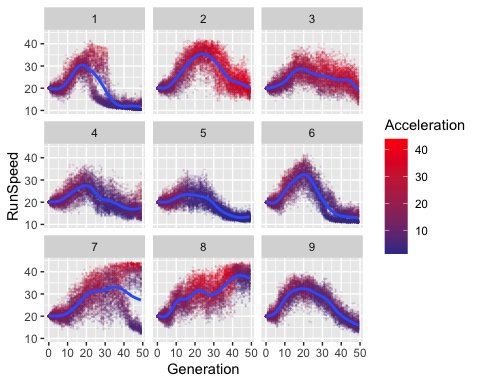
Example: Civilians and clonal reproduction.

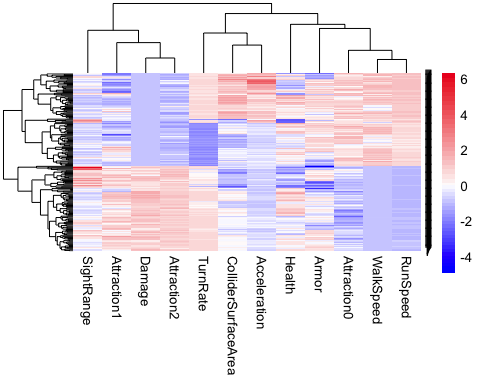
### Heatmap



ggplot(AutoCiv, aes(x=Generation, y= RunSpeed))+  
 geom\_point(aes(x=Generation, y= RunSpeed, color = Acceleration),  
 size = 0.1, alpha = 0.1)+  
 geom\_smooth()+  
 scale\_color\_gradient(low = muted("blue"),  
  
 high = "red",  
  
 space = "Lab",  
 na.value = "grey50",  
 guide = "colourbar",  
 aesthetics = "colour")+  
 facet\_wrap(~replicate)

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'





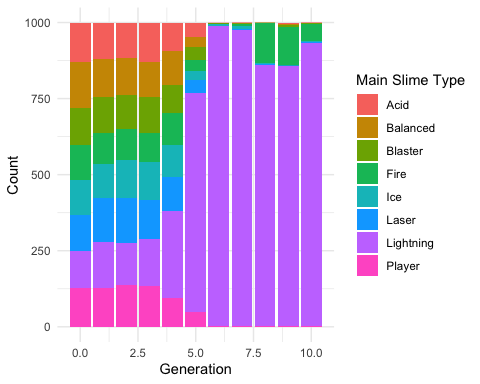
Population Size: Performance Vs. Drift

Example: Genetic Drift and Effective Population Size

Twin2<- read.csv("Twin/Gene\_Write\_File2.csv")  
Twin1<- read.csv("Twin/geneWriteFile.csv")  
Twin3<- read.csv("Twin/geneWriteFile1000.csv")  
  
  
Twin3<-Twin3%>%  
 mutate(Unique.Slime.ID = paste(Wave.Number, ".", Slime.ID))%>%  
 mutate(Unique.Parent.One = paste(Wave.Number-1, ".", Parent.One))%>%  
 mutate(Unique.Parent.Two = paste(Wave.Number-1, ".", Parent.Two))  
  
  
df\_parents <- Twin3 %>%  
 select(Unique.Parent.One, Unique.Parent.Two) %>%  
 pivot\_longer(cols = everything(), names\_to = "parent\_type", values\_to = "parent\_id")  
  
# Count the number of offspring for each parent  
offspring\_counts <- df\_parents %>%  
 group\_by(parent\_id) %>%  
 summarise(offspring\_count = n(), .groups = "drop")  
  
offspring\_counts <- offspring\_counts%>%  
 filter(parent\_id != "-1 . N/A")  
  
  
offspring\_counts<- rename(offspring\_counts, Unique.Slime.ID = parent\_id)  
  
  
  
Twin3 <- Twin3 %>%  
 left\_join(offspring\_counts, by = "Unique.Slime.ID")%>%  
 replace\_na(list(offspring\_count = 0))  
  
Twin3counts<-Twin3%>%  
 group\_by(Main.Type, Wave.Number)%>%  
 summarise(mean\_Speed = mean(Speed.Trait, na.rm = TRUE),   
 mean\_Resist = mean(Main.Resistance.Trait, na.rm = TRUE),   
 count = n())

`summarise()` has grouped output by 'Main.Type'. You can override using the  
`.groups` argument.

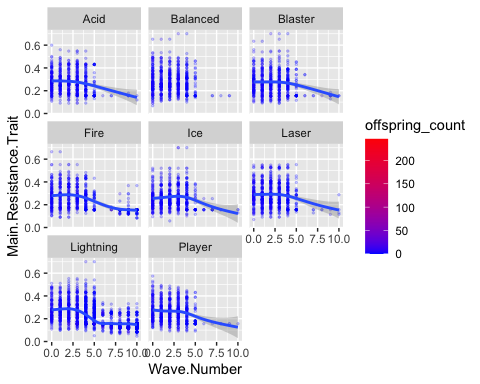
ggplot(Twin3counts, aes(x = Wave.Number, y = count, fill = as.factor(Main.Type))) +  
 geom\_col(position = "stack") +  
 labs(x = "Generation", y = "Count", fill = "Main Slime Type") +  
 theme\_minimal()



ggplot(Twin3, aes(x=Wave.Number, y= Main.Resistance.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Main.Resistance.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

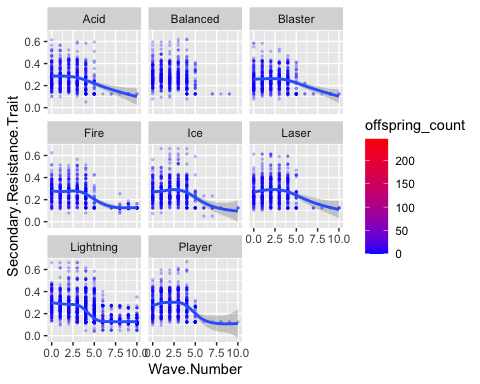
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Secondary.Resistance.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Secondary.Resistance.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

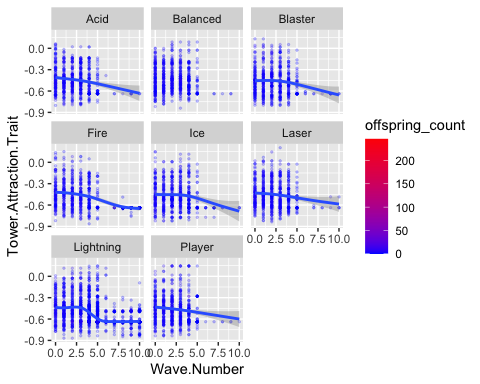
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Tower.Attraction.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Tower.Attraction.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

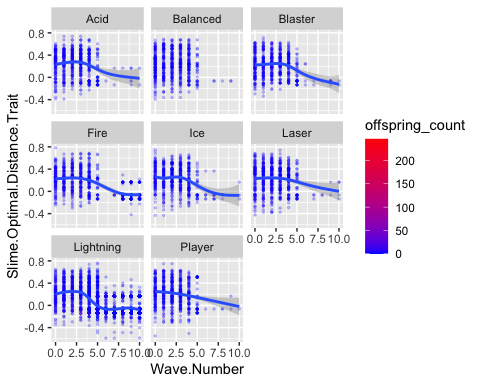
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Slime.Optimal.Distance.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Slime.Optimal.Distance.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

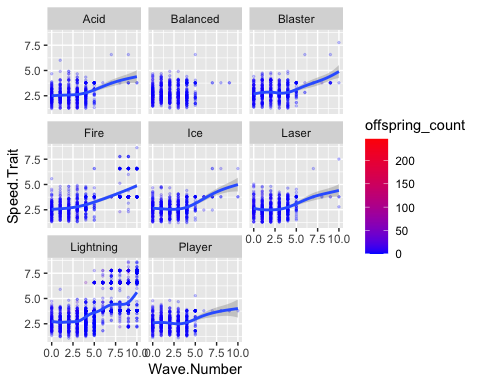
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Speed.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Speed.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

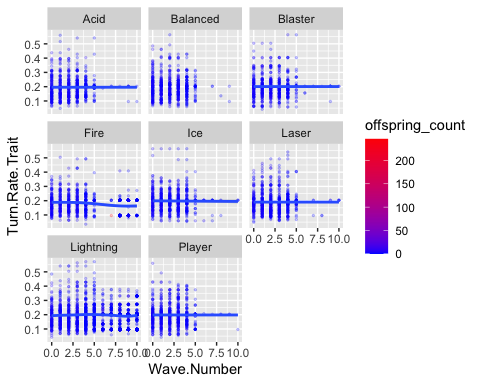
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Turn.Rate.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Turn.Rate.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

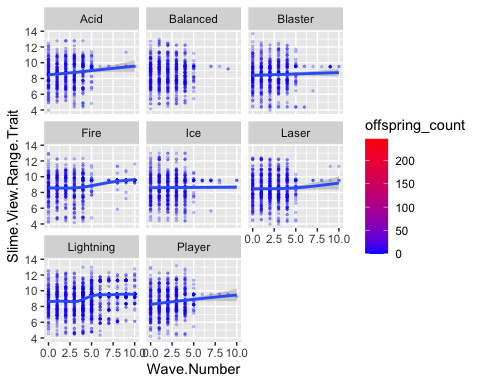
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Slime.View.Range.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Slime.View.Range.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

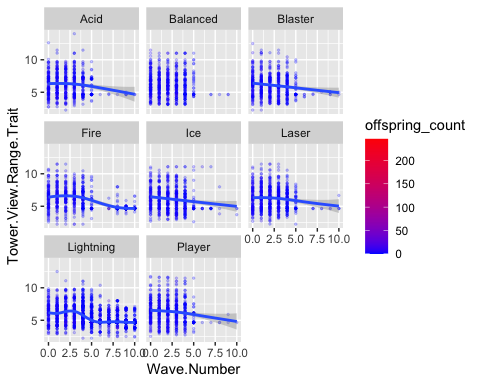
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Tower.View.Range.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Tower.View.Range.Trait, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

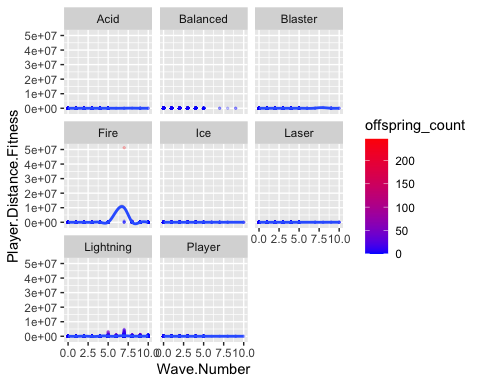
Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



ggplot(Twin3, aes(x=Wave.Number, y= Player.Distance.Fitness))+  
 geom\_point(aes(x=Wave.Number, y= Player.Distance.Fitness, color = offspring\_count),  
 size= 0.5, alpha =0.2)+  
 geom\_smooth()+  
 facet\_wrap(~Main.Type)+  
 scale\_color\_continuous(low="blue", high = "red")

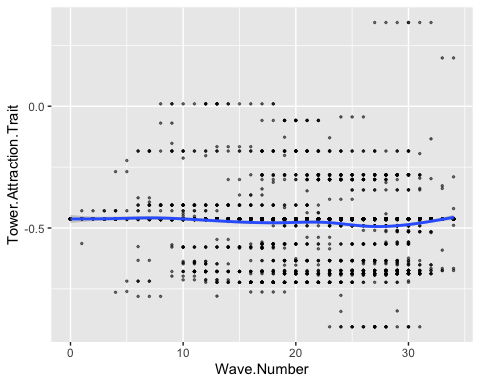
`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'

Warning: Computation failed in `stat\_smooth()`  
Caused by error in `smooth.construct.cr.smooth.spec()`:  
! x has insufficient unique values to support 10 knots: reduce k.



Twin2Lightning <- Twin2 %>%  
 filter(Main.Type == "Lightning" | Secondary.Type == "Lightning")  
  
ggplot(Twin2Lightning, aes(x=Wave.Number, y= Tower.Attraction.Trait))+  
 geom\_point(aes(x=Wave.Number, y= Tower.Attraction.Trait),  
 size=0.5, alpha = 0.5)+  
 geom\_smooth()

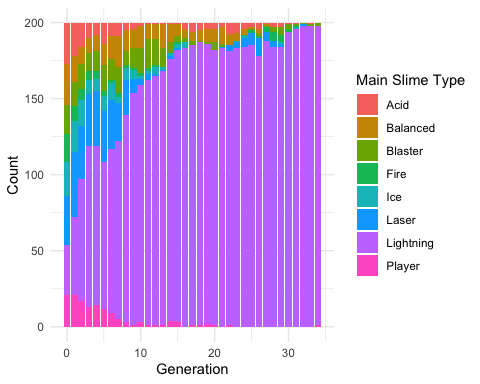
`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'



Twin2counts<-Twin2%>%  
 group\_by(Main.Type, Wave.Number)%>%  
 summarise(mean\_Speed = mean(Speed.Trait, na.rm = TRUE),   
 mean\_Resist = mean(Main.Resistance.Trait, na.rm = TRUE),   
 count = n())

`summarise()` has grouped output by 'Main.Type'. You can override using the  
`.groups` argument.

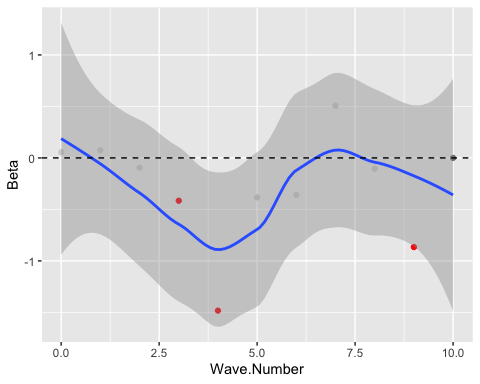
ggplot(Twin2counts, aes(x = Wave.Number, y = count, fill = as.factor(Main.Type))) +  
 geom\_col(position = "stack") +  
 labs(x = "Generation", y = "Count", fill = "Main Slime Type") +  
 theme\_minimal()



LastGen <- Twin3  
  
LastGen <- Twin3 %>%  
 select(Wave.Number, Slime.ID, Player.Distance.Fitness, Main.Resistance.Trait, Secondary.Resistance.Trait, Slime.View.Range.Trait,   
 Tower.View.Range.Trait, Player.View.Range.Trait, Wall.View.Range.Trait, Sheep.View.Range.Trait, Slime.Attraction.Trait,   
 Tower.Attraction.Trait, Player.Attraction.Trait, Wall.Attraction.Trait, Sheep.Attraction.Trait, Slime.Optimal.Distance.Trait,   
 Speed.Trait, Turn.Rate.Trait, Sprint.Duration.Trait, Sprint.Cooldown.Trait, offspring\_count) %>%  
 filter(Wave.Number == 35)  
  
  
paletteLength <- 50  
myColor <- colorRampPalette(c("blue", "white", "#ED2024"))(paletteLength)  
# length(breaks) == length(paletteLength) + 1  
# use floor and ceiling to deal with even/odd length pallettelengths  
  
  
Heatmap2 <- LastGen %>%  
 select(Player.Distance.Fitness, Main.Resistance.Trait, Secondary.Resistance.Trait, Slime.View.Range.Trait,   
 Tower.View.Range.Trait, Player.View.Range.Trait, Wall.View.Range.Trait, Sheep.View.Range.Trait, Slime.Attraction.Trait,   
 Tower.Attraction.Trait, Player.Attraction.Trait, Wall.Attraction.Trait, Sheep.Attraction.Trait, Slime.Optimal.Distance.Trait,   
 Speed.Trait, Turn.Rate.Trait, Sprint.Duration.Trait, Sprint.Cooldown.Trait)  
  
Heatmap2[,1:18]<- scale(Heatmap2[,1:18])  
  
colnames(Heatmap2) <- c('Fitness', 'Main Resistance', 'Secondary Resistance', 'Slime View Range', 'Tower Viewe Range', 'Player View Range',   
 'Wall View Range', 'Sheep View Range', 'Slime Attraction', 'Tower Attraction', 'Player Attraction', 'Wall Attraction',  
 'Sheep Attraction', 'Slime Optimal Distance', 'Speed', 'Turn Rate', 'Sprint Duration', 'Sprint Cooldown')  
   
Heatmatrix2 <- as.matrix(Heatmap2)  
  
myBreaks2 <- c(seq(min(Heatmatrix2), 0, length.out=ceiling(paletteLength/2) + 1), seq(max(Heatmatrix2)/paletteLength, max(Heatmatrix2), length.out=floor(paletteLength/2)))  
  
# Generate the heatmap  
pheatmap(Heatmatrix2,   
 cluster\_rows = TRUE, # don't cluster rows  
 cluster\_cols = TRUE, # don't cluster columns  
 clustering\_distance\_cols = "euclidean",   
 clustering\_distance\_rows = "euclidean",   
 clustering\_method = "complete",  
 cellheight = 0.75,  
 fontsize\_col = 10,  
 color = myColor,  
 breaks = myBreaks2)

traittemp.Twin<-Twin3%>%  
 select(Wave.Number, offspring\_count, Slime.Optimal.Distance.Trait)%>%  
 group\_by(Wave.Number)%>%  
 mutate(Trait = scale(Slime.Optimal.Distance.Trait, center = TRUE))%>%  
 mutate(Trait2 = Trait\*Trait)  
  
Gradients <- traittemp.Twin %>%  
 group\_by(Wave.Number)%>%  
 do({  
 model <- lm(offspring\_count ~ Trait + Trait2, data = .)  
 data.frame(  
 Beta = coefficients(model)[2],  
 PB = summary(model)$coef[2, 4]  
 )  
 })  
  
Gradients <- Gradients %>%  
 mutate(sig = if\_else(PB < 0.05 , "Y", "N"))  
  
  
  
ggplot(Gradients, aes(x=Wave.Number, y = Beta))+  
 geom\_point(aes(color = sig))+  
 geom\_smooth()+  
 scale\_color\_manual(values = c("lightgrey","red"))+  
 geom\_hline(yintercept=0, linetype="dashed", color = "black")+  
 theme(legend.position = "none")

`geom\_smooth()` using method = 'loess' and formula = 'y ~ x'



# ggplot(Twin3, aes(x = Player.Distance.Fitness)) +  
# geom\_histogram(binwidth = 1000, fill = 'blue', color = 'black') +  
# labs(x = "Variable Name", y = "Frequency") +  
# theme\_minimal()+  
# facet\_wrap(~Wave.Number)

### Educational Outcomes

We reasoned that playing these types of games might have implications for STEM education. The success of the player is based on her comprehension and application of principles of evolutionary biology. The parallels to real world examples are numerous, and include the rapid evolution of antibiotic resistance in microbial pathogens, adaptation of crop pests to chemical and biological control measures, and behavioral adaptation to captivity in domesticated animals.

Adding biological evolution to video games makes the games better for the game player and facilitates player comprehension of complex concepts that are hard to teach.

The basic evolutionary model used in the game studio projects consists of four key elements: selection, inheritance, variation, and time. In each game there is a population of enemies that ‘attack’ the player. At the end of each wave/generation each enemy is assigned a fitness value based on how well it performed. Typically, this includes a measure of how much damage it did to the player or their resources or lives. However, in many cases the enemies are defeated before they inflict and damage, so fitness also measures how close they came to doing damage, either by shooting close to the player or by getting close to the player. This is necessary to make sure that there is a broad fitness gradient - i.e. that the majority of the enemies do not get assigned the same fitness.

Once the enemies have been assigned a fitness based on their performance against the player the enemies with the highest fitness are *selected* to reproduce. There are a number of difference selection techniques, but all of them combine favoring the most fit individuals to drive evolution with some level of stochasticity to maintain a diverse population. Over selection of a few best individuals can lead to a population with too little diversity to effectively evolve.

The individuals selected to reproduce pass their digital genes into the next generation in a form of *inheritance*. Commonly two selected individuals will mix their digital genes in a form of simulated sexual reproduction. This increases the *variation* in the population by creating new hybrids.

Additionally the newly generated genomes undergo a mutation step in which some of their values have random noise added. This further increase the *variation* in the population of enemies. Mutation is typically applied at rates so that 1-3 gene values vary per individual per generation. This is several orders of magnitude higher that natural, biological evolution, which is necessary to increase the rate of evolution to the point where it is observable to the character and results in change that is meaningful on the timescale of a game that only lasts tens of generations. This higher mutation rate works because there are no lethal mutations in our models, all of the mutations effect traits like speed, strength, resistance, etc. whereas in biological evolution significant mutations are often lethal in the form of failed biological processes.

Finally, this process gets repeated every wave - every evolutionary generation - in the game. This repetition of the evolutionary process over *time* is what allows the enemies to evolve to become tougher opponents.

These basic components: selection, inheritance, variation, and time can be implemented in many games to add an evolutionary component. The critical aspect of evolution over other techniques to scale difficulty is that evolution is drive purely by the enemies performance against the player. If the player implements a novel strategy, whatever enemies perform well against that strategy will reproduce. The game automatically builds in difficulty in response to the player’s choices.

Variation Inheritance Selection Time [done above?]

### Project Hastur

*Development:* Project Hastur is a tower defense video game developed by students in the Polymorphic Games studio at the University of Idaho. The game features a campaign of 16 original game maps, as well as an experiment mode (described below) that allows the game to serve as a simulation model. Project Hastur was released on the Steam game site on Feb 12th, 2019. The unique feature of Project Hastur is that the enemies in the game model a finite population defined by digital genomes of quantitative traits.

*Gameplay:* In Project Hastur, the player must defend their base against waves of enemies called the Protean Swarm. The player defends their base by placing defensive towers in strategic locations, and the towers each have different capabilities, strengths, and weaknesses. In campaign mode, the player unlocks new game maps and defensive capabilities as they proceed through the story. Each map is defined by victory conditions that involve defeating a set number of enemy generations or building their defenses to specific criteria. Enemies appear in “waves”, a classic trope of tower defense games in which the developers script the number and type of enemies that appear over the course of the level. In Project Hastur, however, the enemy waves are part of a generational evolutionary model - each new enemy wave is comprised of the offspring of the most successful parents in the previous wave. In this way, we programmed Project Hastur to become an evolutionary tower defense game in which the enemy population adapts as the game proceeds.

The defensive capabilities of the player include four different categories of tower with qualitative differences in how they interact with the physics engine of the game. These include kinetic, ice, flame, and acid based towers, with each category including 4 different versions of upgrades. In addition, the player can access and upgrade two different abilities that are activated on demand with a cooldown - robotic minions and airstrikes.

Miller, Jon D., Eugenie C. Scott, Mark S. Ackerman, Belén Laspra, Glenn Branch, Carmelo Polino, and Jordan S. Huffaker. 2021. “Public Acceptance of Evolution in the United States, 19852020.” *Public Understanding of Science* 31 (2): 223–38. <https://doi.org/10.1177/09636625211035919>.