Systems and feedback

In the previous chapter (TODO REF CH3) we talked about mechanics, and listed some types of mechanics with their examples. In this section, we discuss how mechanics work together, and how they can be analyzed in terms of systems which encapsulate a variety of mechanics and their interactions.

We will look at mechanics as building blocks of dynamic systems, which together bring out gameplay. First, we will look at how mechanics chain together, into a sequence of relationships (such as producers and consumers), and how to analyze them. Second, we look at what happens once chains themselves start interlocking and feeding into themselves and each other. Finally, we will look at how these kinds of dynamic systems can be designed when making a new game. But first, an example.

Motivating example: Diablo

Suppose we are playing a "dungeon crawler" game like Diablo. With a motley crew of adventurers under our control, we leave camp and head into a dungeon filled with monsters and hidden treasure. As the team goes through and clears out each area, they pick up items dropped by monsters and loot treasure from hidden chests. Eventually their backpacks get full, so everybody gathers their things and heads back to town, to sell the loot and recuperate, regain some health, maybe craft some potions or fix weapons, or train up on new spells. Then it's time to venture out again.

This simplified example shows a number of mechanics that interlock together. We can talk about them as if composed of several high-level systems, such as:

- Inventory
    - We gain items from raids (weapons, treasure, crafting resources, etc.)
    - Weapons can be equipped on characters on the team
    - Other items can be put in inventory or left behind
    - Inventory is finite, and initially small, but it can be upgraded
    - Back in camp, items from inventory can be sold, or inventory can be emptied to storage
- Economy
    - We gain items from raids (weapons, treasure, crafting resources, etc.)
    - Back in camp, we can sell items to gain gold
    - We can then spend gold to buy different items (weapons, potions, spells, etc.)
        - Or spend gold to buff stats or items (e.g., upgrading inventory size)
        - Or spend gold on services (learn a new spell, get a weapon fixed, etc.)
- Crafting
    - We gain items from raids (weapons, treasure, crafting resources, etc.)
    - Use crafting resources at a crafting station to make new kinds of items
    - Some items may be also "taken apart" back into crafting resources
    - Crafting requires a recipe, listing how resources combine, and their proportions
    - Crafting recipes are themselves items that be bought, sold, found, etc.

    … and so on.

This is just a quick illustration of three different game systems: inventory, economy, and crafting. At this point we omit combat, exploration, and several other systems, and will concentrate on just those three to make the discussion simpler. But there are more systems than those in a game like Diablo that we could talk about, and new interesting systems get invented by game designers all the time.

These kinds of systems are commonly found in games, and they are designed to *interlock*: they share various mechanics (such as items or currencies) but use them for very different purposes. Interlocking systems become more interesting as we discuss below.

Game systems

We can consider a system as a *collection of mechanics set up to work together* in specific ways. The combination of which elements are used, and how they are arranged together, defines the system's performance.

Systems are a very useful abstraction. When we try to come up with the design for a new game, we can start from mechanics – but sometimes it is more fruitful to start the conversation on the level of entire game systems instead, because that lets us "paint with a larger brush" as it were, to start with, before we come back to fill in the details.

Three examples of systems were mentioned above:

- Inventory: item equipping, ownership, and capacity limits
- Economy: the exchange of resources / items and currency
- Crafting: the transformation of some resources / items into others

Systems are made up of mechanics. To make those systems interact with each other, they should *share mechanics,* but for different purposes. The reason for this is to create interactions and trade-offs between systems, and force the player to make interesting decisions. In our example, inventory interacts with economy via items (they can be carried in the inventory, but also bought and sold), and both also interact with crafting (if crafting materials or outcomes can also be carried, bought, sold). When faced with the opportunity to pick up a new item, the player will need to evaluate the various systems and their trade-offs (for example: "Is this item useful for crafting? Can I sell it later? Is there enough capacity in the inventory? Do I have enough strength to carry it all?" and so on). A broad mechanic like items or currencies can often interact with *many* systems, including ones we have not talked about such as combat.

When designing a new game, the choice of "setting and systems" is often a good starting point: what is the setting where the game takes place, and what are the systems that the player will interact with? Answering these questions first will push the design in a specific direction, and simplify the process. For example, we could decide to focus on "combat, exploration, weapons, and upgrades, in a fantasy setting", and that starts us in the direction of Diablo. Or perhaps we focus on "crafting, building, economy, and management, in a historical setting", which pushes the design along a strategy direction like Civilization. We do not need specific details just yet – but these commitments are already enough to get us started, so that they can be fleshed out with specific mechanics.

(As a side note: the word *system* is also often used to describe stand-alone programming modules, such as the physics system or the graphics system. These kind of *code systems* are not collection of mechanics, although mechanics such as "gravity gun" can usually be built on top of them. But it is usually clear from context when people talk about code system, or gameplay systems.)

Thinking in systems

Game systems are a kind of an abstraction that elides the details of the individual mechanics when they are not needed, and allows us to focus on how the whole game is put together from larger pieces.

Systems are common in all areas of science and engineering: for example, in automotive design it's common to talk about the transmission or the engine as if they're standalone entities, and how they interact with each other – and then only worry about the implementation details once necessary. For example, in biology and medicine, we talk about systems of different kinds of organs and tissues working in concert, only worrying about the lower-level workings when needed.

In game design, there are three specific benefits:

a. *Isolation:* it is easier to discuss how some mechanics work together, if we focus on just them in separation from the outside (as we will when we discuss feedback loops below).
b. *Reuse:* existing systems are solutions to particular design problems, and we can sometimes apply those solutions to our problems, or draw inspiration from them for our own games.
c. *Scaling:* it's easier to discuss large-scale gameplay elements, like pacing or progression, when our building blocks are large (we get to these later in this chapter).

A working designer needs to have a breadth of experience with a variety of existing systems, and a multitude of prior examples to draw on. When faced with a design problem, they will be lean on the experience with other games with different types of systems, different implementations and pros and cons, to find a good solution.

Layering

A more developed game will typically contain a larger multitude of systems that interlock in interesting ways, like gears in a watch. For an extreme example, here is an incomplete list of systems that can be found in a strategy game like *Civilization 5:*

- Geography: the map is random and full of resources that feed other systems
- Territory: tiles are owned by players and only produce resources for their owner
- Cities: immovable units that produce other units, gather resources from surrounding tiles
- Buildings: generate resources or currency, convert resources
- Armies and combat: movable units that explore, attack, improve resources on tiles, etc
- Neighbor civilizations: AIs that compete with the player
- Trading and Diplomacy: exchange agreements or resources with neighbors
- Economy: using currency to buy or upgrade units, tiles, etc
- Science: tech tree that unlocks better units and buildings, reveals new tiles
- Culture and Golden Ages: systems for unlocking large -scale buffs

- Religion: passive resource generation that requires a large install base
- Victory conditions: multiple scoring rules let the player pursue different paths towards victory
  ... and some number of additional ones

The point of this list is not to be exhaustive, but to give some examples of how different systems can work in one game.

This list also does not mention how the systems interlock, but they commonly do. For example, natural resources such as wheat or sheep are a shared mechanic that provides interactions: map tiles contains random natural resources (geography), and if we can gain control over those tiles (territory) we could extract them or convert to other resources (via buildings). Those other resources, in turn, we can use to grow the population (in cities) and build an army so we can attack our enemies (combat).

Finding the right layering of systems and interlocks is a difficult, game-specific design problem, and it is often done by starting small and getting some base systems right, then growing the design from there. Sometimes this can be done descriptively, "on paper", but it is beneficial to implement simplified *system prototypes* as part of the design process, to know how they behave in the real world when actual players start using them.

Mechanic chains and loops

We often find that mechanics end up forming *chains of interactions,* and often those chains form a *loop,* such as conversion loops or feedback loops. Chaining and looping are common, fundamental design patterns, and they can exist within a system or span multiple systems.

We now take a deeper look at how that might work. Continuing with the dungeon crawler example from above, we will look specifically at the *economy* and *inventory* systems. We will see how resource acquisition and conversion mechanics end up forming a loop, and we will discuss some ways to analyze the behavior or such loops.

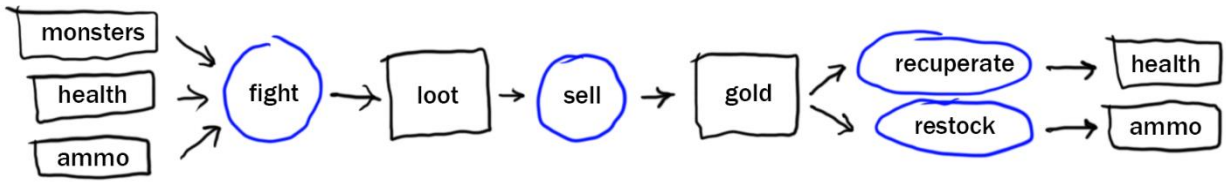
Conversion chains

Consider our dungeon crawler example from above. Let's say that we find an area overrun by trolls, and trolls carry loot. So, we send our team to fight them, and let's say it plays out like this:

1. The team fights as many trolls as possible. After fighting 10 trolls, their inventory is full, so they go back to camp. Also, in total they lost 60 health points and used up 100 arrows, so they need to heal and restock.
2. They sell 10 troll loot items for 10 gold each, for a total of +100 gold.
3. They restock ammo for -10 gold, and find a healer to restore the lost health points for -60 gold, so they spend total of -70 gold to get back on their feet, so they can go out again.

   Note that the net result from the expedition is +30 gold, so they are making a profit.

We can diagram this sequence as follows:



This is a kind of *resource diagram*, which shows how resources are produced or consumed by various actions available to the player. The diagram is very coarse, since it omits the details of how much can be converted, under what circumstances, and so on. In the diagram, we use rectangles to denote what resources are being consumed and produced, and then round nodes to denote actions that consume and produce them.

The economy mechanics (selling, healing, restocking) and combat mechanics (fighting to get loot) are set up in a *conversion chain* of producers and consumers: each step provides something that is an input to the next step. In this example: fighting costs some health and ammo, and it removes one monster from the game board, but it generates loot. This loot can then be sold for gold, and gold used to restore the health and ammo that we spent on the fight.

In other words, this diagram illustrates the *resource conversion* aspect of the troll hunt:

- *Fighting* converts trolls, health, and ammo, into loot
- *Selling* converts loot into gold
- *Healing* converts gold into health
- *Restocking* converts gold into ammo

In this diagram we didn't specify *tuning values*, that is, how much of each resource gets consumed or produced. But we mentioned them in the description, and noticed that this chain is net profitable: it takes effort to go through each step, but in the process, we gain some income (+100 gold, ammo, health, loot), incur some costs (-70 gold, ammo, health, loot), and end up with the net profit of +30 gold.

Calculating exchange rates

We can also turn these tuning values around, and consider our resources in terms of their *exchange rates*, that is, how much one resource is worth in terms of other resources:

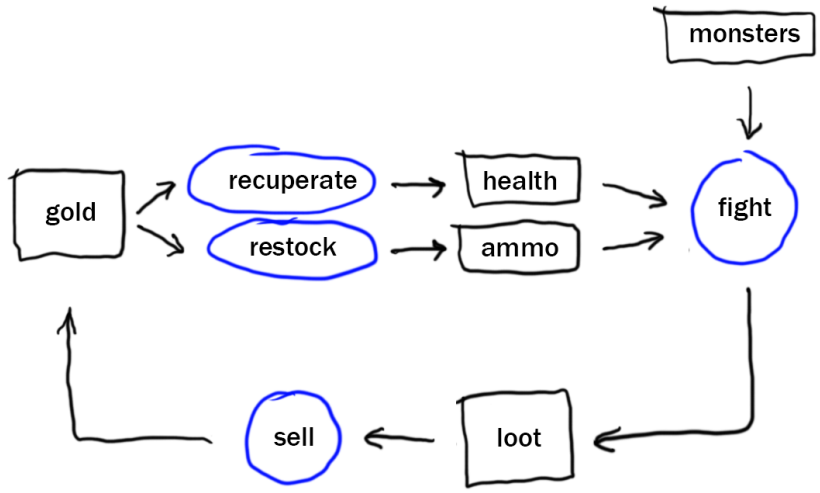
- *Selling:*
  1 loot item = 10 gold
- *Restocking:*
  100 arrows = 10 gold
  Therefore 1 loot item = 100 arrows
- *Healing:*
  60 health = 60 gold
  Therefore 1 loot item = 10 health
- *Fighting:*
  on average, getting 1 loot means attacking 1 monster, and losing 6 health and 10 arrows
  Therefore, 1 loot = (1 monster + 6 gold + 1 gold)

=> 1 loot = 1 monster + 7 gold
=> 10 gold = 1 monster + 7 gold
=> 1 monster = 3 gold

The last line of this resource conversion analysis shows something interesting: monsters are basically free money. If we find a monster and we're willing to spend some gold (and labor!) to fight it and deal with the consequences, we can generate even more gold. By doing this over and over again, each monster encounter will net us +3 gold pieces not counting labor or accidental costs.

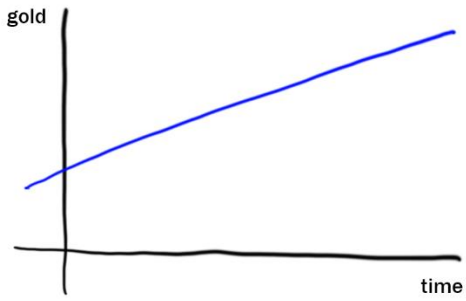
Conversion loops

Now imagine doing this repeatedly. If we ignore the actual hard work of combat and travel, and just focus on the resources being gained and lost, we can represent repeating the chain as a loop that looks like this:



The chain repeats itself to form a *loop,* because the outputs of a step eventually loop around, and produce inputs to the same step in the future. The loop as we described it always produces the same amount, but that doesn't have to be the case, as we see in a bit.

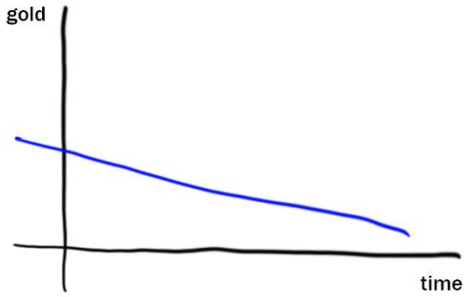
We mentioned that this loop is *profitable*: each time through it we lose -70 gold but gain +100 gold, so our income is higher than costs. Therefore, we could keep making more money by doing it more than once. It is easy to imagine what would happen if we went hunting over and over: our gold inventory over time would look something like this:



This shows a nice linear trend, as each iteration generates more in income than we pay in expenses. Over time we could amass quite a bit of gold.

Assuming that trolls were respawned without end, we would have an *infinite source* of gold. But this is bad: once there is an easy source of infinite income, it can generally be exploited by players, and will destroy the carefully tuned challenge and progression on which many games rely. For example, with infinite gold, it could become very easy to gain access to expensive and overpowered weapons and armor, which will make the game unchallenging and not fun (as we discuss in the next chapter (TODO REF CH5)).

We could remedy this in several ways. One, we could change the *tuning values,* that is, the coefficients of how much different things cost. For example, what if we doubled the cost of health potions and arrows, from -7 gold per troll to -14 gold? Then the player's wealth over time would look more like this:



This loop now turned *unprofitable*, as the player would lose wealth each time they went through it. If they kept repeating it, the reward for all their hard work would be going bankrupt. This is not desirable: players would see right through this, and regard it as a broken loop.

We need to look at other options. The player should not be able to get resources endlessly, or lose them endlessly. If it is desirable for the player to gain resources in a production loop, there needs to be something that stops them from iterating through the loop forever.

We could *force* the player to switch away to a different set of activities. This is a valid approach: for example, add a wear-and-tear mechanic which will make the crossbow break down after some use, and force the player to work on crafting a new one instead. But for this example, let's just stay within the economy system itself for a while longer.

Some other ideas:

1. Cap the resources that can be harvested (such as limiting the number of trolls). This is a hard stop on the loop after n iterations, forcing the player to switch away.

2. Make the outcome less predictable (so player may have to go back to camp empty-handed). This adds uncertainty, potentially increasing the cost in randomized ways. But an experienced player will be able to figure out the expected reward by averaging the results over many attempts.

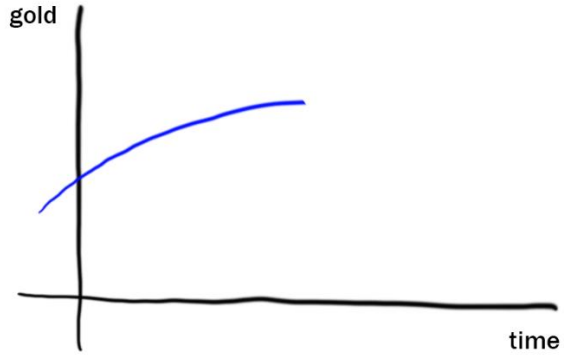
3. Increase labor cost of combat or travel (make it not fun to do this too much). This basically adds a cost of going through the conversion loop, but the cost is not in resources – it is something

external like time spent on the activity.

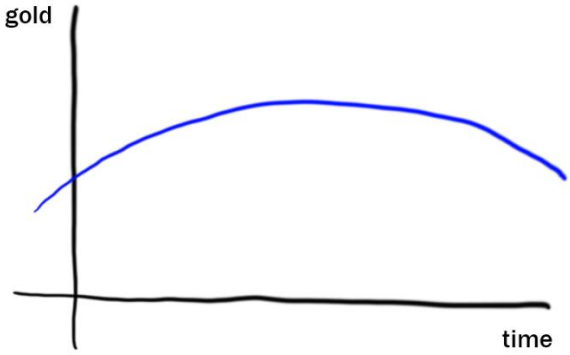
Spending a lot of time on rote actions to iterate through a resource loop is commonly called grinding, and players often have a negative opinion of it (but will keep doing it, even while complaining, if the loop is profitable *enough*).

4. Scale the profit curve over time, so that the loop changes from profitable to unprofitable as the player continues to go through it.

This last example is particularly interesting: it's an example of *dynamic tuning*. For example, we can adjust the cost dynamically by raising the prices of ammo and health potions over time. Let's say ammo and health start out costing us -70 gold per trip, but then increased by additional -10 gold with each trip, while the gold we get from selling loot doesn't change:



If our trips start out with a profit velocity of +30/trip, this tweak adds deceleration: now velocity changes over time, and for the worse. After some time, the loop turns from profitable to unprofitable, and if players keep iterating through it, they will start losing wealth:



This discourages grinding: players will be much better off if they cut their losses before the expenses get too large, and move on to another area in the game, with different challenges and loops.

In practice, dynamic tuning can work well, if the player can control when to engage and disengage with the loop in question. It is more interesting for the player if they can make the decision of how long to try

doing one thing, and when to cut their losses and move on to something else that's perhaps more profitable or interesting.

To summarize:

1. We looked at a resource *conversion chain,* where items and resources can be produced when playing the game, and then exchanged for gold.
2. This particular chain formed a *conversion loop,* which is one where outputs of some element loop back in, to produce future inputs for the same element.
3. An economic loop can be *profitable* or *unprofitable*, depending on whether the income (in terms of items, resources, currency, time investment, and so on) exceeds costs.
4. Profitable loops may be exploitable if they're *unbounded*. We can *bound* a loop through a variety of techniques, but ideally through something that keeps the player in control of how close they want to get to the bound, and when to stop.

Feedback loops

So far, we talked about loops that produce a stable amount of resources over time, as long as the player contributes labor to crank through the loop. But many loops are not so static.

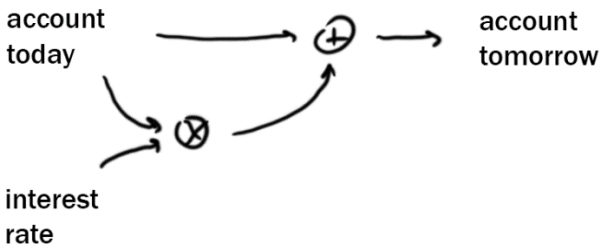
In many real-life situations, we encounter systems that perform actions proportional to the *current state* and reference some kind of a reference *set point.* These are commonly called *feedback loops,* because the system affects the state, and this state affects the system's behavior at a future time, which affects the state again – so the system's output *feeds back* to itself as an input.

Feedback loops are exceedingly common in natural and manmade systems, from animal populations to manmade control systems as we discuss below. They are very common in games as well, as they are often used to amplify or attenuate differences over time.

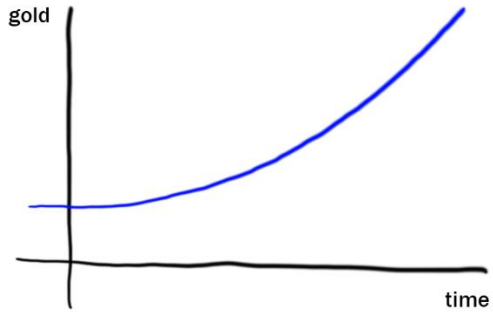
Positive feedback

A *positive feedback loop* is one where differences between the current state and the set point are processed and added back to the state. For a real-life example, consider *compound interest.* Suppose we have a bank account with some money in it, and each month an interest payment is paid out. That interest is then deposited back into the same account, so the next month's interest is based on both the principal and the interest accrued from the first month, then the next interest payment is based on the principal amount and two months' worth of interest, and so on.

Diagrammed, the feedback might look something like this:



If we plotted the account over time, assuming the interest rate is constant and positive, it might look something like this:



This a profitable loop for sure, but it's also one that gets *more and more profitable* the more times you go through it!

This is a positive feedback loop, because on each iteration, the difference from the set point of zero gets multiplied by the interest rate, and added back to the account, which will *increase* the difference next time around. Sometimes this is also called a *divergent loop*, because each iteration causes the value to diverge further and further from the set point.

This kind of feedback produces powerful build-up and amplification of initial state. The multiplication step leads to runaway exponential growth.

Positive feedback loops are very common in games, for example:

- Monopoly is every designer's favorite example of a long term positive feedback loop. The player with more money can buy or upgrade more properties, which means they collect more rent, which increases their chances of getting even more money. On the other hand, the player with less money has fewer chances to collect rent, and fewer chances to acquire more land.

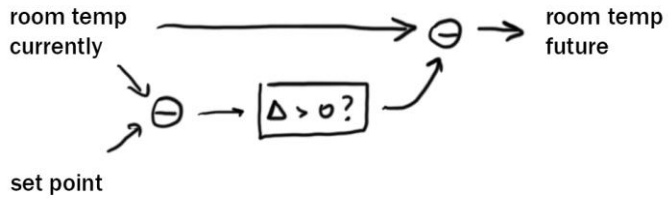
- RTS and strategy games often use divergent feedback loops as well. In StarCraft players mine minerals, which they can spend to build units. But one of those units are harvesters which will let you mine even more minerals, and faster. This positive feedback loop is fortunately kept in check by having a cap on the amount of resources that could be harvested, as well as other systems that consume those minerals.

- Similarly, in games where military victory brings wealth and resources, players with the most victorious armies will likely get the largest spoils of war, which will ensure that their armies get even larger and more powerful.

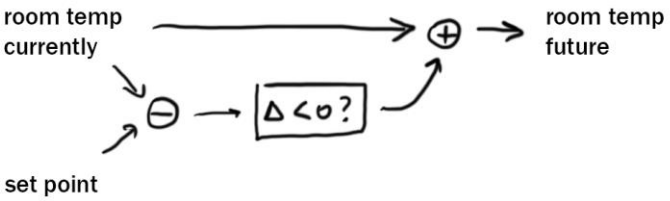
We can easily come up with many more examples of positive feedback, from tycoon games, simulations, or even something as simple as clicker games.

Negative feedback

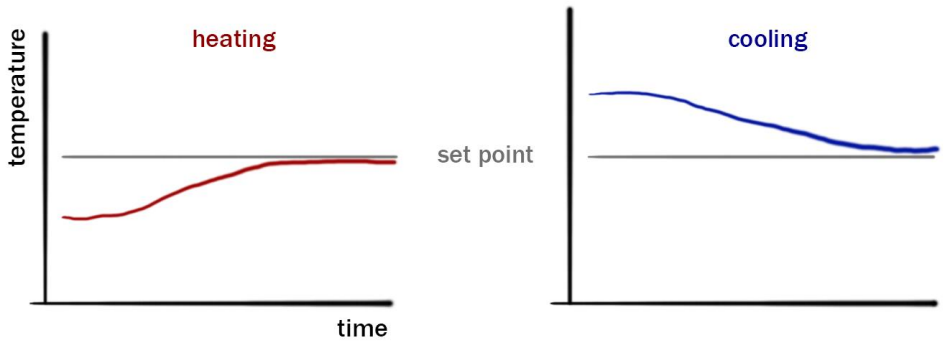
The stereotypical example of a negative feedback loop is that of a thermostat hooked up to a room heating and cooling unit. The thermostat is set to some desired temperature, and it periodically monitors room temperature. If the difference is too high, the difference from set point is positive, so a cooling element is engaged to reduce temperature and reduce the difference.



Similarly, if the room is too cold, the difference from set point is negative, so a heater starts to raise the temperature, and similarly reduce the difference from set point.



Plotted over time, the effect of the thermostat might look like this:



Negative feedback loops are called that because the difference from set point produces actions to reduce this difference in the future. Sometimes they are called *convergent loops* because feedback pushes the system to converge towards the set point.

Negative feedback is less common in games, but still present in some cases. For example, in a tycoon game, if the player has to pay a 1% annual "maintenance fee" on their account balance, absent any other revenues this will slowly drain the account, so the player has to actively counter this by having sources of revenue that exceed the losses.

A better and more common example is dynamically changing the game to make it easier or harder, depending on how well the player is doing:

- Racing games sometimes employ "rubber-banding": if the player is doing very poorly in the race, computer drivers will slow down to make sure not to get too far ahead, but if the player is leading the pack, drivers will pick up speed to make it harder for the player to win.

- In racing games where drivers get power-ups (which buff or nerf their abilities), the same effect can be achieved by modulating the quality of power-ups. For example, Super Monkey Ball makes it more likely for losing players to get speed-up buffs, compared to players who are ahead of the pack (TODO Rules of Play REF p 221).

- Similarly, some role-playing games might tune enemy stats up or down when the player enters a new area, based on the player's own level and stats. This is because the player's level might be higher or lower than anticipated when entering an area (for example, depending on whether they pursued all of the side quests), so dynamic tuning will make sure enemy encounters have the intended difficulty level, not too easy but also not too hard, no matter the player's history.

These can be considered as simple methods of dynamic difficulty adjustment. Additional methods are also possible, such as changing loot drop rates or quality, changing the distribution of enemies or materials in the level, and so on, depending on how far the player's state diverges from expected state.

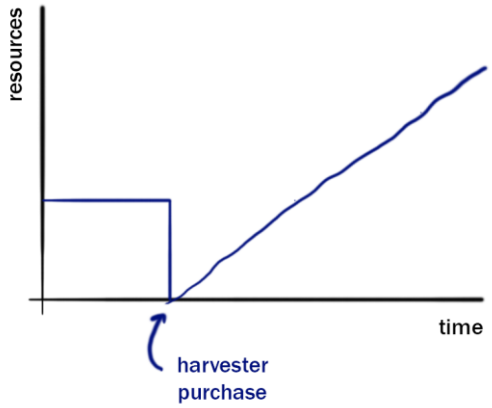
Effects of positive feedback

Positive feedback loops have some nice effects on the player experience: it feels good to see your armies or bank accounts grow. It satisfies player motivations for power and control. It also reflects many of the natural and man-made systems that we are familiar with, such as a fast-growing population overtaking an ecosystem, or the winner-takes-all tendencies of unregulated free markets. The so called *clicker games* use this fascination with positive feedback loops to great effect.

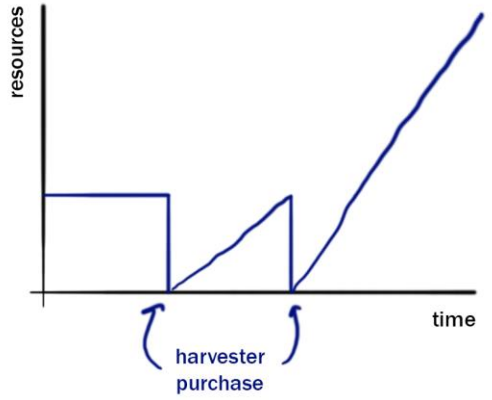
However, positive feedback loops have the effect of amplifying early advantages. In single-player games, the effect might be benign - the player who invests in early growth will be rewarded by achieving higher wealth later. To prevent this from turning into a boring runaway feedback loop, designers typically add a variety of trade-offs to make the player think twice about whether to let the loop feed itself, or spend the resources on something else.

To use an example from StarCraft, early in the game the player is faced with decisions about how to spend their crystals, which are a currency used to build various units and buildings. A variety of spending options is available, but one of the units is a harvester which digs up more crystals over time.

If the player spends their initial crystals to build one harvester (and nothing else) their crystal budget might look as follows (for the sake of this example we assume this rate is constant over time):



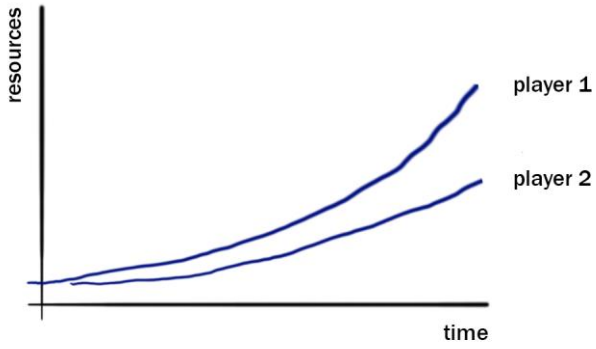
The single harvester will provide revenue at velocity X to fund future purchases. However, if the player decides to spend this money on a second harvester, they will have to wait longer to have useful revenue, but at that point they will be producing revenue at velocity 2X.



This is an interesting strategic decision point for the player: whether to “eat the seed corn” and reduce the later harvest, or go hungry longer to produce a bountiful harvest later – and how to evaluate this vis-à-vis other needs that must be funded, such as building armed units to attack or defend oneself. Making decisions about which feedback loops to feed, and which to pause, leads to interesting strategic dilemmas.

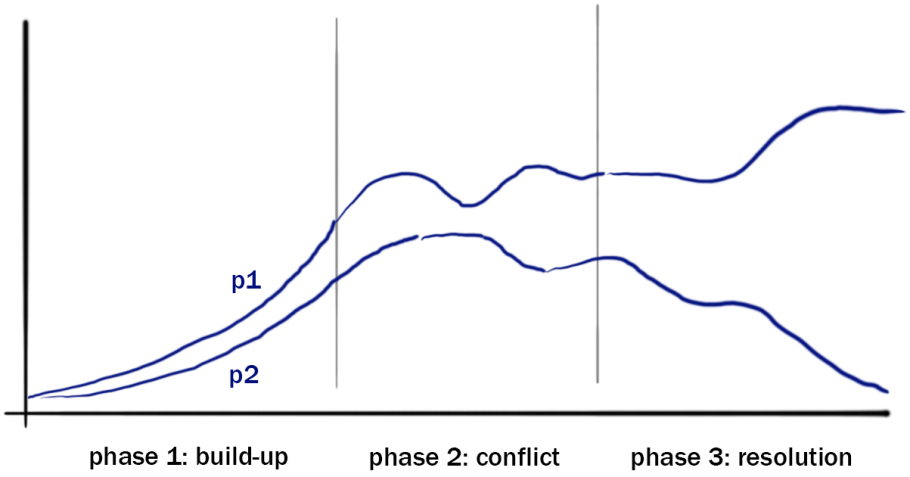
Additionally, in multiplayer games (or games with AI players), positive feedback might lead to an undesirable situation: if players engage in loops that feed upon themselves, the player who got an early advantage, even if small, may eventually outpace the other player. Then if the difference between players’ resources grows too large, to a point where the losing players cannot catch up and bridge this gap, it will have a destructive effect on the overall feel of fairness and competition.

For example, we consider the early stages of a war game where the player's army conquers lands, which lets them build and feed a larger army, which lets them conquer more lands, and so on. Let's say there are two players, and Player 1 got a bit of a head start in the beginning. Since army size and land conquest form a positive feedback loop, the player's army size curves might look like this:



Both are growing in a divergent way, but since Player 1 had a better beginning, the *gap* between them is increasing rapidly as well. At some point they may drift so far apart that Player 2 will never be able to catch up to Player 1, and if Player 2 realizes that their position is unwinnable, that will likely make the game much less enjoyable from that point on.

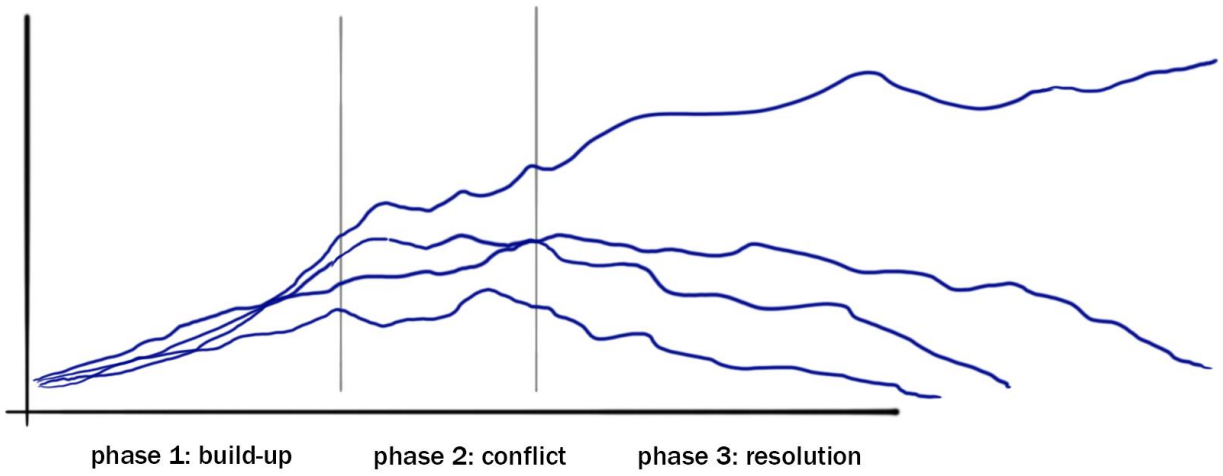
Here is an example of how it might play out:



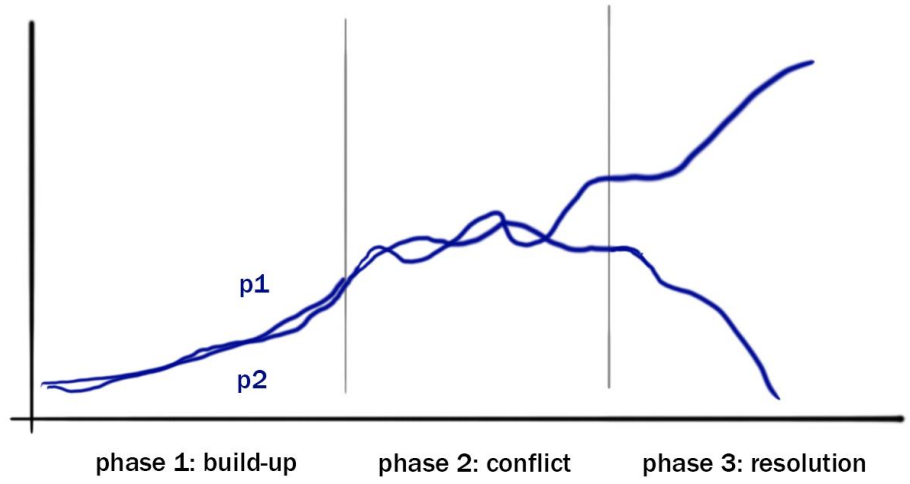
If the difference between the players is too large, once Player 1 realizes their advantage, they will force Player 2 into a conflict (combat, arms race, etc.) where Player 2 will have a hard time keeping up. Then at some point it will become clear who the victor is going to be, and the game enters a *resolution phase* where players escalate conflict and spend down their resources to settle the winner. This resolution will be far less enjoyable for players who know they don't have a fighting chance.

Monopoly is especially famous for amplifying early wins, and having a very long and tedious resolution phase, as the winning player slowly erodes the bank accounts of the other players, forcing them into bankruptcy. That is a slow *war of attrition*, where two sides chip away at each other's resources (so the side with most resources is likely to win).

The war of attrition is a common source of complaints among more advanced players. Since the outcome becomes obvious not too long into the game, and the rest of it is a long resolution, an example graph for Monopoly might look like this:



In comparison, games feel better if they have more interesting build-up and conflict phases, with more time spent in competition as players try to position themselves for victory, and a shorter resolution phase:



Effects of negative feedback

Negative feedback loops can be used to adjust the game to the player's abilities, or perhaps to some desired type of play, but this too can have a negative impact.

In a single player game, such as the role-playing game we described before, we could adjust enemies automatically to match the player's current level and character stats. This will provide a consistent challenge regardless of the player's prior successes or failures. But at the same time, it will make the player's past successes *less meaningful.* The reward for getting better should be that things get easier –

and if getting better instead means the world just becomes more difficult, it makes the player wonder why they should try to get better in the first place.

In multiplayer games there is another effect. For example, in a racing game with rubber-banding, negative feedback is going to keep both players close together regardless of how well or how poorly they do. In effect, their early actions will matter much less than maneuvering into the front of the pack in the final lap. In this case, only the last moments of the game matter, and the game leading up to it will feel unrewarding if the player's early performance is inconsequential.

Games that rely heavily on randomization (like card games, children's board games, but also some computer games) often suffer from a similar effect as well, because if the random effect is too strong, it has a similar effect to rubber-banding, in the sense that it can easily obliterate the player's entire performance so far and render their early effort useless.

Ultimately, games must combine both: positive and negative feedback, rewarding players' performance in early game, making it meaningful and useful, but also preventing it from biasing the outcome too early.

Emergence and chaos

We say, "you do not see the forest for the trees" when someone focuses on the details, and fails to see things on a larger scale. A system might be a collection of smaller elements, and yet a forest has properties and behaviors which are not obvious from studying single trees specifically.

Emergent behavior

If we zoom out and analyze a system as an abstract unit, we may observe behaviors which arise from, but are not predicted by, the behavior of the constituent mechanics. We call these *emergent behaviors.* Some natural examples include:

- Schools of fish and flocks of birds can be modeled as a single group that navigates through space and avoids obstacles, even though it arises out of very simple flocking rules that each member follows, such as rules about maintaining momentum and steering towards the centroid of the group [TODO Reynolds REF].

- Ant hills and insect hives are similarly highly organized structures, formed by insects following simple, local rules, seemingly without needing to understand the overall structure.

- At the extreme end of the spectrum, all eukaryotes are examples of organized entities that emerge out of the billions of cells following simple rules, and organizing themselves into tissues, organs, and systems that have specific properties.

Most importantly, a behavior is considered emergent if it not easily predicted from the behaviors of constituent elements. Games can exhibit emergent behaviors, but designers rarely use them intentionally, because they are hard to design and hard to reason about. Given the designer's goal of creating a specific experience for the player, using emergent behaviors requires a lot of work to experiment with, and observe, and tune, and make sure the systems behave as intended.

And yet, some games are more likely to employ emergent behaviors as part of the desirable experience, in spite of their unpredictability, for example: simulation games, and massively multiplayer games.

1. In simulation games, part of the fun of the experience is being faced with a collection of complex systems with unpredictable emergent characteristics, and trying to control and manage them, or perhaps just letting them run and observing what they do.

   One anecdote from the game *Dwarf Fortress* illustrates this point. In the game, players build and manage underground dwarf settlements. The game is full of systems with emergent interactions, and one example of such interaction became well known. At one point, players discovered that cats in the fortress sometimes get drunk, which was not expected or coded intentionally, so how did it happen? It turned out to be a collection of systems working together in unexpected ways: at one point, taverns were added to the game, and dwarves who bought beer could clumsily spill it on the floor, then another system implemented liquid tracking via footprints, while the cat behavior system guided cat hygiene and self-cleaning, and yet another system guided ingestion of substances and their effects. What happened was: dwarves spilled beer, cats walked through it, got it on their paws, and when they cleaned themselves they ingested the beer, and got drunk. (TODO REF http://www.pcgamer.com/how-cats-get-drunk-in-dwarf-fortress-and-why-its-creators-havent-figured-out-time-travel-yet/)

2. Multiplayer games often exhibit emergent behavior as well, because human players bring their social behaviors, structures, and expectations into the game, such as tendencies to organize, cooperate, compete, and form larger social structures to accomplish larger goals.

   Large scale MMOs often display, and explicitly support, these kinds of social behaviors. MMO games such as *EVE Online* will host very large social structures, where individuals work together as corporations to compete with other corporations, in an ecosystem that supports emergent groups of all sizes, from very large teams to individual lone wolves. (TODO REF needed) The emergent properties of human social structures are even harder to reason about, but fortunately we have a lot of intuitions about how we behave, and a lifetime of experience, compared to our experience with analyzing other systems.

Emergent behavior arises out of systems or their elements working together, and often surprises us in unexpected ways. This can be a great asset, or an enormous liability, depending on the designer's intentions, desire for control, and intended player experience. It creates gameplay that the player did not expect from interacting with the individual elements. However, making sure the emergent behavior remains within desirable parameters is a difficult task.

Chaotic systems

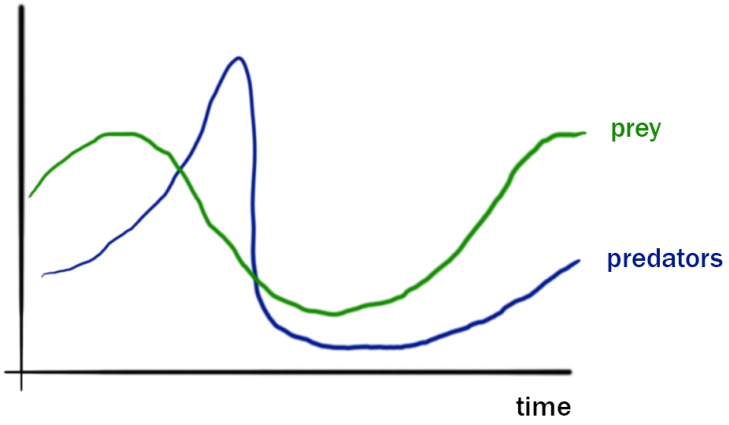
Games will often employ multiple systems with simultaneous feedback loops that feed into each other. This kind of a setup often turns *chaotic,* which is to say, it is a system where slight changes to the inputs can produce drastically different behavior over time, so it becomes difficult to predict analytically how it

will behave at a future point. This term derives from *chaos theory*, the study of dynamical system that exhibit these kinds of properties.

Animal populations in an ecosystem are popular examples of chaotic feedback loops. Each type of animals has its own growth rate: some individuals will give birth, while some will die, and if birth rate remains over replacement rate, the population will grow as long as the resources allow (e.g. food and space). But what if the food of one population is another population, such as foxes hunting rabbits?

This is the case with ecological "predator and prey" models. Two populations depend on each other in different ways: predator population grows when there is prey to capture, while prey population grows when there are fewer predators.

The result are two feedback loops that modulate each other, and the effect might look something like the following:



Simple two-element systems may sometimes be modeled analytically using differential equations, but more complex systems typically defy analytic solution, and must be simulated. The actual ecosystem of an area, such as a local forest or a river basin, is a collection of a large number of feedback loops that, just by its sheer scale, typically exhibits chaotic properties and defies analysis.

However, chaotic ecosystems do not have to be natural. Real-life economies, for example, are very complex, entirely manmade feedback systems, large networks of multiple producers and consumers all working and competing at the same time. On the micro level, the economic study of *supply chains* focuses on the real-world resource production and consumption feedback loops, but it fails to scale to large networks. They also often exhibit chaotic properties: a small change in initial conditions can be amplified over time to lead to a very different situation later, which makes their behavior difficult to predict.

The following example of games imitating life illustrates some difficulties with tuning artificial economies, and shows systems exhibiting properties that are both emergent and chaotic. The early MMO game Ultima Online was one of the few online games to try to model its ecosystem somewhat realistically, at least initially: new players had to hunt rabbits and small prey to get pelts, so they could sell them for money, to buy a sword and some armor to finally venture out in search of adventure. But

hunting animals would decrease their population in the ecosystem, which then affected how quickly the population bounced back to previous levels, as well as affected other species in the ecosystem that needed them.

Once the game went into testing with more players, these loops turned out disastrous. New players could initially only hunt small prey, so they emptied the forests of small animals to raise enough money. With the population not replacing itself fast enough, the next cohort of players who joined after them found themselves with no way to make money, and do anything meaningful in the game, which made the game *really not fun*.

In the end, the dynamic ecosystem simulation had to be scrapped and replaced with non-looping chains of stable sources and sinks: animals would just spawn at a desired rate to maintain a population, and local merchants bought it at fixed prices, so that new players would always have a source of easy money. While not realistic, at least it was predictable and tunable, and it made the game reasonable for new players. (TODO REF Garriott (per wikipedia), Starr Long in https://www.gamespot.com/articles/ultima-online-preview/1100-2559974/)

Systems design

Now that we have seen a variety of systems, in general and in detail, how do we get started on them?

The basic approach is iterative: we need to alternate coming up with building blocks, putting them together, and seeing how they work. There are two basic directions to start from:

1. Bottom up, creating and testing one system, then adding on another one and testing how they work together, and so on. In this way systems accrue and accumulate over time.
2. Top down, imagining what gameplay would be like, describing it, and then figuring out what systems are needed to make this gameplay happen, and how they interact with each other.

Both approaches are used in game development. The bottom up approach is more exploratory, and lets the designer experiment and see where the exploration takes them. The top down approach is more common when there is a specific end goal, or specific experience that we want to evoke, and we need to figure out how exactly to do it. Both approaches can also be used together, in which case we would use top down analysis to figure out some necessary systems, but also experiment and explore to build on that base in unexpected directions.

From user stories to systems

The top down approach can seem complex: we might have big ideas for gameplay, but also have difficulty translating them into specific systems. To make this process easier, we can employ an iterative technique to help us list out a variety of systems that will be needed.

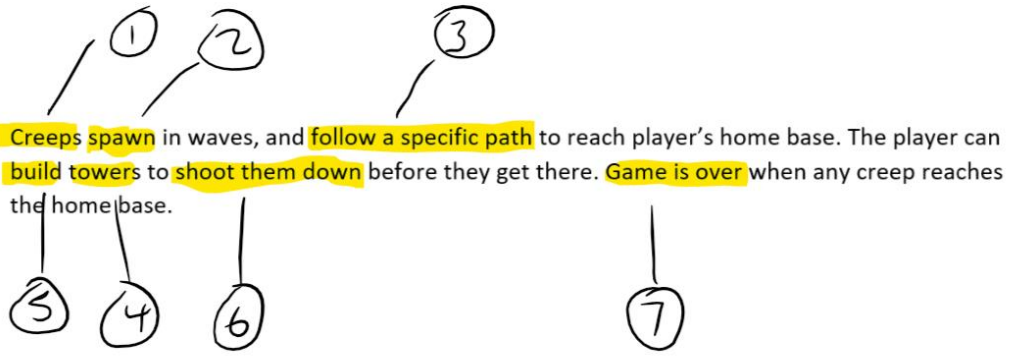
We start with a “user story”, a narrative description of a bit of gameplay, and then we analyze what kinds of systems or mechanics are entailed by that user story, then we analyze what kinds of systems or mechanics are in turn entailed by those, and so on, iteratively filling out our design with each pass.

For example, let's say we want to make a tower defense game, where lines of creeps follow a specific path to reach the player's base, while the player sets up a defense perimeter to shoot them down before they get there.

First, we write out a narrative about the desired experience:

> *Creeps spawn in waves, and follow a specific path to reach player's home base. The player can build towers to shoot them down before they get there. Game is over when any creep reaches the home base.*
>
> (There is much more that we could describe, but we will stop here to keep this example simple.)

Second, we go through this simple narrative sentence by sentence, word by word, and mark it up. What kinds of systems are implied by the narrative? What do we need to make this happen? For example:



1. *We need some kinds of enemies, or "creeps", to try to capture the player's home.*
2. *We need a system to spawn those enemies in waves.*
3. *We need pathfinding, or some pre-determined paths and hand-made levels.*
4. *We need buildings (towers) to shoot down the creeps.*
5. *We need the ability to build and place those towers.*
6. *We need a combat system to determine how shooting will work.*
7. *We need victory conditions, and loss conditions.*

Next, we go through this list, and ask ourselves: for each item, how will it work?

1. *We need some kinds of enemies, or "creeps", to try to capture the player's home.*
   What does this mean?
   a. Creeps should be simple and dumb entities that march towards the end-goal.
   b. They will get shot at by the player, so we need some defensive abilities.
   c. Let's say a few can counter-attack. They need some attack abilities as well.
   d. There need to be multiple types of creeps to provide variety.

2. *We need a system to create those enemies in waves.*
   What does this mean?

   a. We want fewer and easier creeps initially, and then ramp them up.
   b. We need a spawner to keep track of time in each level, and create the right type and number of creeps. Designer needs to be able to specify different waves per level.

3. *Creeps need to be able to find their way towards the home base.*
   What does this mean?
   a. Let's say levels are free-form so the player can place buildings anywhere.
   b. We need a pathfinding system for creeps to find their way.

4. *We need towers to shoot down the creeps.*
   What does this mean?
   a. These will be buildings that can be placed, but not moved. Placement matters.
   b. Just like creeps, towers also need to have defensive and offensive abilities.
   c. Tower attack need to be limited to some range, to make placement interesting.
   d. To add variety, we want to have different towers with stat trade-offs: more defense or more offense, better or worse firing range or speed, and so on.
   e. The player should be able to upgrade a tower to buff its stats.

5. *We need the ability to build towers.*
   What does this mean?
   a. There should be different costs to placing or upgrading towers with different stats.
   b. If there is a cost, there needs to be a currency and a way to earn it.
   c. Let's say the player earns currency by clearing out a wave of creeps.
   d. Creeps should be worth different amounts of currency based on type and stats.

6. *We need a combat system to determine how shooting will work.*
   What does this mean?
   a. We're going to use attack and defense points, and health bars like in RPGs. The difference between those determines how much health is lost during an attack.
   b. Towers (and creeps that attack) also need a firing rate, which will be different for different unit types.

7. *We need victory conditions, and loss conditions.*
   What does this mean?
   a. The game ends when a creep survives and reaches player's base.
   b. We decide the game cannot be "won", it's just an endless sequence of levels.
   c. We can make the levels increase in difficulty.
   d. To mark progression, we also add score, which increases faster on harder levels.
   e. We also add a high score table and leaderboards for competitive players.

As we can see, a simple, three-sentence description of gameplay can easily imply a lot of mechanics and systems: combat, construction, pathfinding, progression, economy, and so on. And even within those, we have already started discussing some very specific mechanics: attack and defense points, buffs and upgrades to improve those points, weapons, area of effect ranges, currencies and how they can be earned and spent.

In this top down part of the design process, we generate narrative descriptions like these for different parts of gameplay, and then try to figure out what systems and mechanics would make that kind of gameplay happen.

If we want to, we can merge this with bottom up design as well: now that we have some ideas about systems and mechanics, we can experiment and explore adding different ones. We could start prototyping and do this during the prototyping stage, or we can do it earlier and just consider different ideas purely "on paper", for example:

- What would happen if we added a spell and magic system? Maybe spells could support combat, and provide new kinds of weapons and upgrades – but instead of paying with a currency, we acquire them via casting spells.
- But how do we get spells? Maybe the player needs to collect scrolls to unlock spells, and accumulate mana points to actually "fuel" them.
- But how do we get mana? Maybe specific kinds of creeps drop mana as they walk, and scrolls are something you can research by spending currency on building a research tower?
- Does this mean we need to add research towers? How do they work?
  … and so on, and so forth.

By combining top-down analysis based on a desired narrative, as well as experimenting with new directions and evaluating where they lead, the designer can perform a kind of guided exploration of the system design space. This kind of exploration gets easier as we become familiar with numerous examples of systems, in other games as well as scientific and engineering fields, to be able to draw on existing models and bring them to our own game.

System tuning

Once we have defined the systems and the interlocking between mechanics, we must figure out how *exactly* they will interact. How much will a tower cost, and what currency will be used? How many attack and defense points will each creep have? How much currency do we get for killing a creep?

This process of finding those exact numeric values, and making those low-level decisions, is called *tuning* the game. Tuning is often an iterative process, largely guided by experience and playtesting, and finding the correct balance for all the moving parts takes work. If the game is tuned too tight, such as when it is quite difficult for the player to progress and succeed, the game will be more challenging but also more frustrating for players who are not sufficiently experienced. Likewise, if a game is tuned too loose, when the rewards are plentiful and challenges are easy, this will make life easier for a new player, but will frustrate and bore an experienced player.

However, there might not even be a "middle ground" for tuning that satisfies all player types, which is why so many games attempt to add multiple difficulty levels as a stop-gap solution. Worse yet, since different parts of the game get tuned separately, we could arrive at a place where different parts of the game are tuned differently: for example, if combat is tuned too easy, so creeps die quickly, while the economy is tuned too hard so we cannot afford new towers easily, this will be bound to frustrate players across all skill levels.

A later chapter (TODO REF CH6) describes the pitfalls around difficulty levels and player motivation, but for now we focus on how it gets done, not the difficulties around it.

Approaches

There are four basic approaches to system tuning:

- Manual tuning
- Spreadsheet models
- Simulation
- Analytics

1. Manual tuning is the obvious case: we have some ideas about what to improve, so we go into the game, change some content (tuning variables, level layout, etc.), and play it to see how it works. Based on the playtest we form a hypothesis about what to change next, and repeat. Sometimes those changes will be improvements, while at other times they will cause problems and need to be undone. But in this way, we improve the game iteratively by repeating the tune-and-test steps until we reach a satisfactory state.

   This approach is slow, but thorough: because the designer keeps playing the game, they can observe how the tuning changes not only affect the system in question, but also what knock-on effects they have on other systems. Unfortunately, the amount of gameplay required for thorough testing is often prohibitively large (because games have enormous state spaces), so designers rely on compartmentalization (playtesting specific parts of the game in separation) and experience to figure out how the game will play based on just limited testing.

2. To remove the need for gameplay testing, designers sometimes build models of the game systems in spreadsheet software such as Microsoft Excel. This is particularly useful for numbers-heavy systems such as economy or combat. In this case, the designer creates a simplified model of just the systems in question, and simulates how the model would evolve given some player inputs. This is much faster than manual tuning, but has a significant drawback: because the spreadsheet is just a model of some game systems, it will inevitably be simpler and different from what is in the game itself. This means the model is limited in scope, and can lead to erroneous conclusions, depending on the how well the model corresponds to the game. [TODO REF stationary model etc]

3. This discrepancy between model and base game can be fixed in a different way: we can use the game itself, by feeding it artificially prepared player inputs and seeing how it plays out. This simulation would be a kind of automated testing, and it could be much faster than manual testing (maybe the system could play the game at much higher speed, or we parallelize testing across a farm of computers). This approach is difficult and expensive, however: the game must be instrumented for this kind of forward simulation, and we must find a way to produce a variety of fake player inputs for testing, which is actually quite hard for games of non-trivial complexity. One feasible work-around to this limitation, is to limit simulation testing to just a single system (such as economy or combat) which means we only need to produce player inputs for a specific subset of relevant actions (such as buying and selling and earning currency). This is easier than trying to have the computer play the full game, but it shares some of the drawbacks with spreadsheet modeling. [TODO REF Tozour]

4. The last approach is to release the game to players (maybe a limited beta test group, or maybe to all players), collect data on what they actually do, and tune the data accordingly. This approach based on *analytics* has several benefits: the results are based on actual player behavior, rather than assumptions or models about how they might play the game, and players in large numbers tend to be very good at finding edge cases and degenerate strategies.

   However, it is not a sufficient replacement for other forms of tuning. A game cannot be shipped to players with bad tuning, because they will not enjoy it, and this will not result in a good quantity or quality of feedback. However, analytics are great for fine-tuning systems that are already in a roughly good shape, or for finding degenerate strategies that could escape in-house testers.

   Analytics are also used often for iterative feature roll-out and tuning, that is, releasing a new feature of the game to a small number of players first, to confirm that it behaves as expected (and to retune it quickly if it does not), before releasing it to all players.

Role of tuning in the production process

Regardless of the approach, tuning usually happens many times during a game's lifetime: during the *development* of individual systems, then during *integration* of multiple systems together, as well as during a *polish* phase when the entire game comes together.

1. When developing a single system, it is usually a good idea to give it a rough tuning pass right during development – for example, making sure that character movement feels right in a platformer, before moving on to level design. This does not have to be perfect, but once we integrate multiple systems together, tuning deficiencies in one will affect the player's experience of the whole thing, so it is best to get ahead of that early on.

2. Once several systems are integrated, another tuning pass will be needed to iron out interactions between them. For example, in action games, it is common to tune the 3Cs (character, camera,

and controls) together as early as possible, because they all affect each other, and they affect the player's experience with everything else.

3. Once the game comes together in an alpha state, where all the systems are roughly in place, more holistic tuning can happen that considers the total experience, and the player's interactions with all systems and content. At this stage, the tuning changes usually will have smaller scope, but standing up the entire game will often expose a very large number of deficiencies and opportunities for improvement.

In this sense, the tuning process is akin to a visual artist's process: we start with a rough sketch of the whole piece, and then iteratively fill in the details, otherwise it will be hard to see how the whole piece composes together.

From systems to gameplay

In this part we looked at game systems, which are built from simpler individual mechanics. It is often more convenient to talk about systems as standalone entities, both for the purpose of analyzing their behavior in isolation, as well as looking at how they interact with each other as encapsulated modules.

We also looked at the behavior of systems over time. Looking specifically at resources, we developed the idea of analyzing systems in terms of *chains* of steps, which then can hook back into *loops* that feed their outputs into future inputs. Finally we found a common pattern with artificial and manmade systems, where past output causes proportional amplification or attenuation of future output, and the resulting positive and negative feedback loop. We looked at how feedback loops can be incorporated in games, and what effects they might have on the experience of the game.

Now we are ready to switch to the next topic in our sequence, and talk about *gameplay,* or how interacting with mechanics and systems develops over time into the experience of playing the game.

Further reading

Mechanics and systems are the focus of *Advanced Game Design* by Sellers (TODO REF)

"Machinations", which are detailed, functional representations of feedback loops and other mechanics, are described in *Game Mechanics: Advanced Game Design* by Adams and Dormans (TODO REF).

Outside of games, systems have been studied in many contexts, including economics and business management. Economics textbooks such as *The Economy* (TODO Core Proj REF) can be great introductions to contemporary thinking about large scale and complicated systems such as the world economy. Classics of the genre such as *Industrial Systems* and *Principles of Systems* by Forrester (TODO REF) may also be of historical interest.