

09 - Player Psychology, Game Mechanics, and Systems

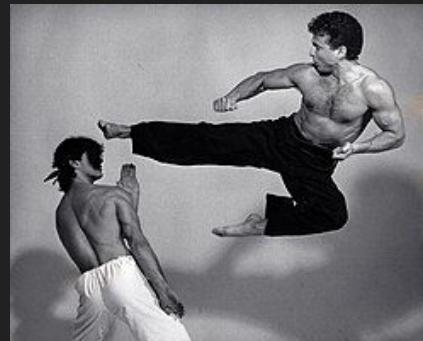
CS 3160 - Game Programming
Max Gilson

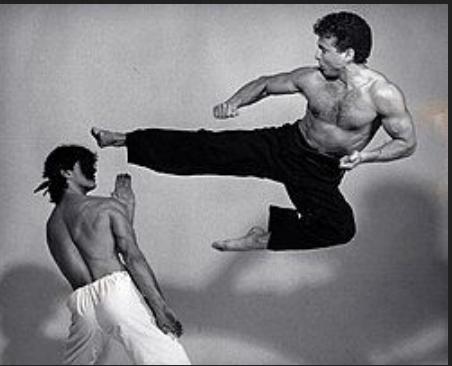
Player Psychology

- We play, enjoy, and dislike games for many different reasons
 - These reasons are completely different for different people
- One of these reasons might be because they allow us to experience or become things that we cannot experience or become in real life
 - This is player fantasy

Player Fantasy

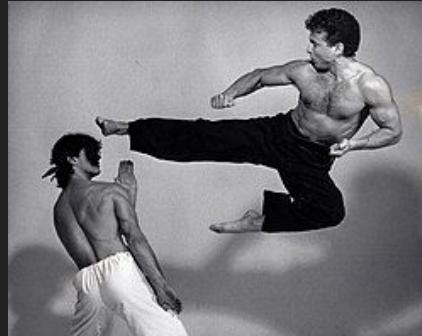
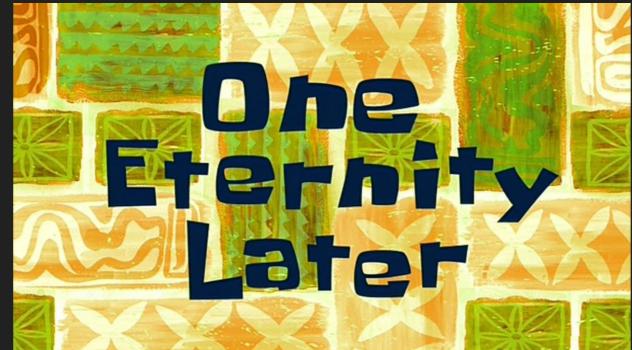
- Games let us live out a “player fantasy”
 - Happy farm community member
 - Kung fu warrior
 - Mad scientist/engineer
- The player fantasy is what players act as when they play your game
- What encourages these fantasies is subconscious





Player Mythology

- Player mythology is who your player becomes through playing your game
- Player fantasy - who you play as
 - Example: in Sifu you play as a martial artist fighting groups of enemies
 - Hitting combos and clearing out rooms feels like playing a kung fu movie
- Player mythology - who you become by playing
 - Example: to play Sifu you have to train combo attacks, fail often, and study the attack patterns of NPCs
 - This mythology IS becoming a martial artist for Sifu
 - <https://youtu.be/msJWFGntJds?si=OCFAUVSCKIMplgma&t=109>



Player Mythology (cont.)

- Player Fantasy - being a leader
 - Example: Civilization lets you play as a leader of a civilization game
- Player Mythology - being a leader
 - Example: World of Warcraft lets you be in charge (leader) of guilds filled with real people



Player Motivations

- Player motivations are the reasons why players will play your game
- Any of these motivations can be satisfied by the same player fantasy
- These motivations are subconscious
- Example: Power + Story motivation for Marvel's Spiderman
- Example: Competition + Strategy for Marvel Rivals

					
Action "Boom!"	Social "Let's Play Together"	Mastery "Let Me Think"	Achievement "I Want More"	Immersion "Once Upon a Time"	Creativity "What If?"
Destruction Guns. Explosives. Chaos. Mayhem.	Competition Duels. Matches. High on Ranking.	Challenge Practice. High Difficulty. Challenges.	Completion Get All Collectibles. Complete All Missions.	Fantasy Being someone else, somewhere else.	Design Expression. Customization.
Excitement Fast-Paced. Action. Surprises. Thrills.	Community Being on Team. Chatting. Interacting.	Strategy Thinking Ahead. Making Decisions.	Power Powerful Character. Powerful Equipment.	Story Elaborate plots. Interesting characters.	Discovery Explore. Tinker. Experiment.



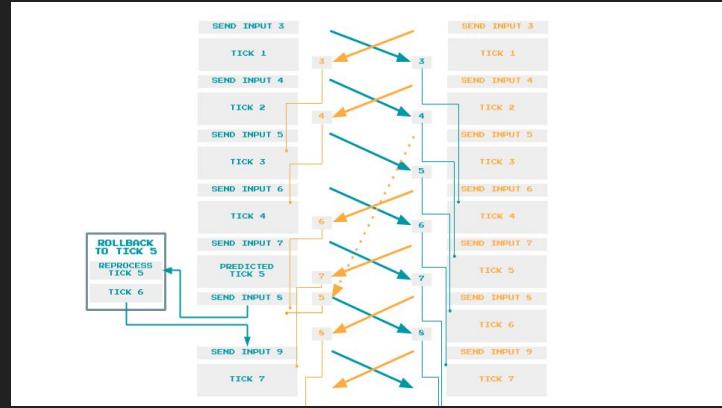
Game Design or Game Programming?

- Player psychology is mostly for game designers to worry about (not game programmers)
- There are things that programmers must consider when implementing features that affect player psychology
 - Example: karmic dice in Baldur's Gate 3 makes players feel like the game is more fair when it's objectively unfair
 - How does your random number generation (RNG) work?
 - Example: poor framerate or bad performance in a competitive game is unacceptable to most players
 - How is your performance?
 - How is your networking?



Player Psychology in Netcode Fighting Game Example

- The designer will design what moves, combos, and attacks a fighting game will have
- The programmer will develop the networking code
- Both elements could determine whether or not a fighting game is a hit with players



Player Psychology in Health Bar Example

- The player's health bar being low creates tension
- If you lie to the player about their health you can manipulate this tension
- You can give players the satisfaction of "barely" making it out of an encounter without ending their game
 - Reality: they had more health than what was actually displayed and were actually far away from losing
- <https://youtu.be/e6tJWDsZv6o?si=-LaYvb6eKWW5FKUO&t=259>



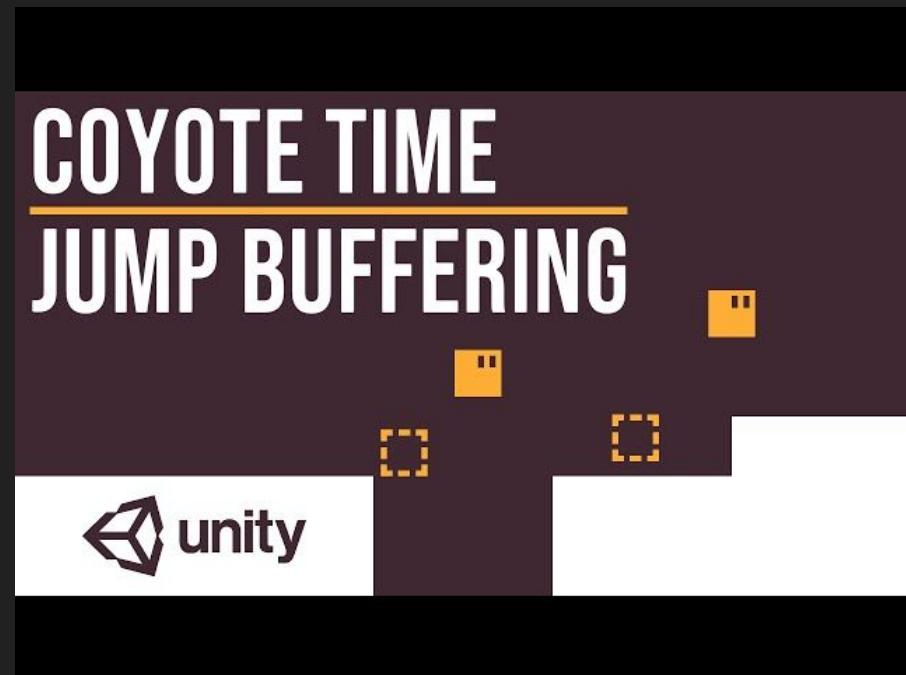
Player Psychology in Platformers Example

- Many platformer games implement “Coyote Time”
- This is a short window of time *after* the player has stopped touching the floor where they are still allowed to jump
- This subconsciously lets the player feel like they are better than they actually are



Player Psychology in Platformers Example

- Coyote time in Unity can be implemented by recording the time the player last touched the floor
- If the time is less than some threshold, allow jumping
- If the time is greater than some threshold, disallow jumping



How to Tweak the Psychology of Your Game

- The psychological feel of your game depends on so many factors
- It's impossible to list every consideration to make
- Even large AAA game companies make mistakes in this realm
- The only way to really know is to test your game with your audience and take notes
 - Ask your testers to play the game, take notes on where they get stuck, frustrated, happy, scared, etc.
 - Work on improving the feel of the game based on this feedback

Playtesting

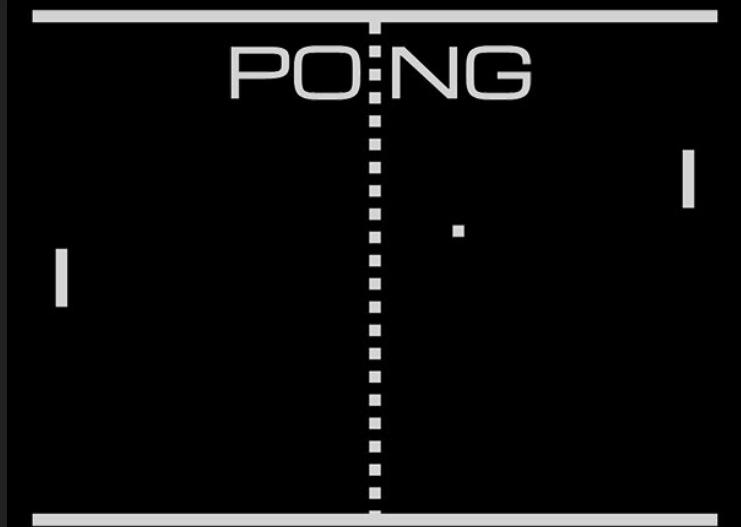
- Set a goal
 - “Teach the player with a gameplay based tutorial”
- Design a system or mechanic
 - i.e. The Gameplay Based Tutorial
- Implement the system or mechanic
 - Understand the purpose of the goal
 - Write the code
 - Create the assets
- Playtest to verify
 - Did the mechanic or system achieve the goal?
 - Test with your actual audience (RPG vs grandma OR DnD enthusiast)
- Iterate
 - If the playtest went poorly, refine the approach and retest
- Hint: you are unlikely to perfect the system or mechanic on the first try

Mechanics and Systems

- Mechanic is a rule or system that controls how a game operates
 - The rules of the game
 - Jumping (Platformer)
 - Turn-based combat (RPG)
 - Resource management (Survival)
- System is a collection of mechanics that interact to create a more complex experience
 - An aggregate of rules and their interactions
 - Movement system (Platformer)
 - Combat system (RPG)
 - Crafting system (Survival)

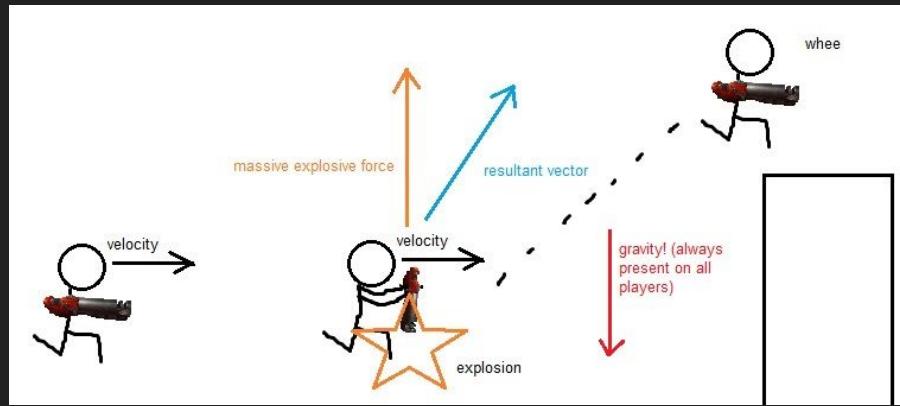
Complex Systems

- “In a complex system, the internal relationships between elements of the system become intricately compounded and extraordinarily complex”
 - “Greater than the sum of its parts”
- Complexity is a double edged sword:
 - Too much complexity and your system is confusing or overwhelming
 - Too little complexity and your system is boring and predictable



Emergence

- Emergence is the property of a complex system to generate unexpected outcomes
 - Example: rocket jumping
 - Emerged from the complex physics + movement system in Quake
 - Example: nearly all of Minecraft
- The outcome of emergence is unpredictable
- The inclusion of emergence is intentional in the design of complex systems



Systems Interconnectivity

- For a component or mechanic to be a part of a system, there must be some kind of interconnectivity
- Systems can be interconnected with each other too
 - Example: Minecraft's crafting system expands the combat system



Systems Interconnectivity (cont.)

- The interconnectivity between systems or components within systems is often not explicitly defined
 - Example: moving the player using a rigidbody interconnects the physics system with the movement system
- Other times, the interconnectivity must be explicitly defined
 - Example: Minecraft explicitly defines in its code what arrangement of items can craft other items
- Regardless, you as the programmer have to ensure that system and its interconnected components, satisfy the overarching goal
 - Ask the question: Does the system actually work like its supposed to?
 - Ask the question: Why does this system satisfy the goal?

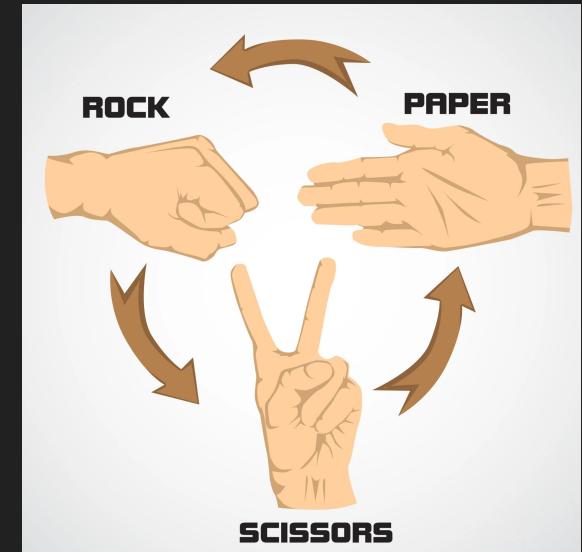
Systems Tuning

- Systems should be tuneable and allow for easy adjusting by designers
 - Easier said than done
 - Who knows what might need to be tuned and how
- This can be as simple as having some global settings that can tweak specific values
 - Example: if all characters (including the player) in a survival game have a thirst meter, a tuneable value that adjusts how fast that meter increases over time is invaluable for adjusting the game's feel

Mechanics Are Just Rules Abstracted



=

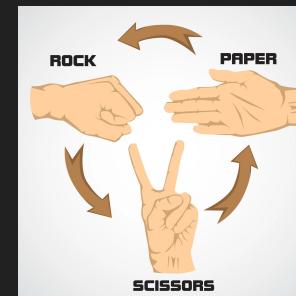


<https://youtu.be/qFRtMiMlfII?si=ICNoNrBYIB9aP30p&t=64>

Mechanics Are Just Rules Abstracted (cont.)

- Games are built out of smaller games
 - Mechanics are usually just smaller games
 - Think: most crafting mechanics are just puzzle games! Stealth mechanics are strategy games!
- Only 5 types of fighting games have ever existed
 - Rock paper scissors is the core foundation
 - Over time more rules are added
 - Some include movement, defense, more attacks
- <https://www.youtube.com/watch?v=zyVTxGpEO>

Fighting game	Moves	Movement
<i>Rock paper scissors</i>	3 attacks	None
<i>Karate Champ</i>	13 attacks, 4 defense	2d axis
<i>Karateka</i>	6 attacks, 2 stances, bow	Side scrolling
<i>Battle Arena Toshinden</i>	> 50 attacks, 2 stances	2 independent 2d axes
<i>Bushido Blade</i>	8 weapons, 3 stances, many moves	2d plane

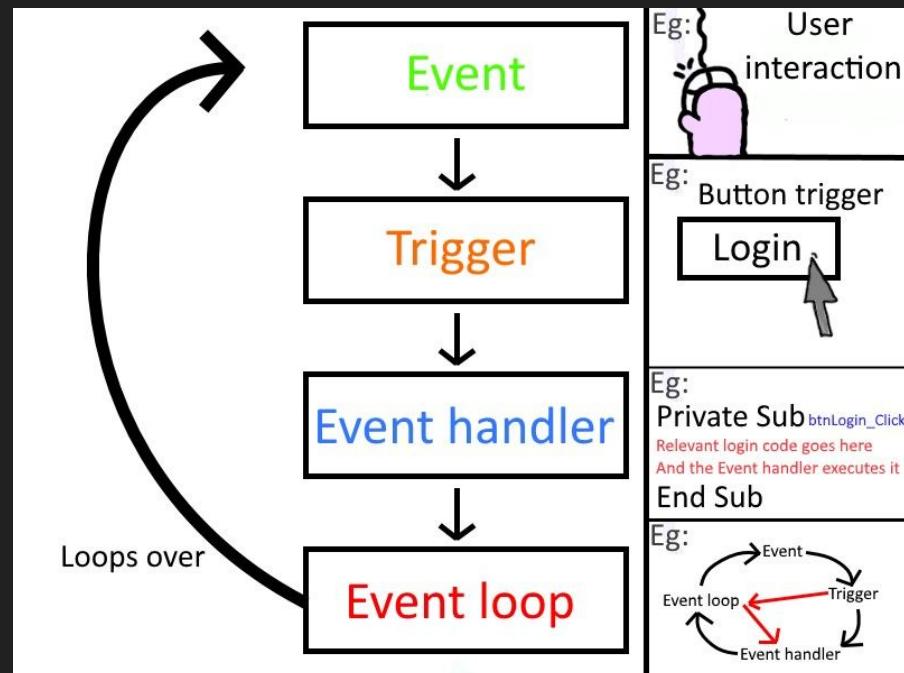


Writing Mechanics and Systems

- Writing the code for complex systems with multiple interactions and mechanics needs to be organized and maintainable
 - If not, the systems and mechanics will be error prone and difficult or impossible to modify, upgrade, or fix
- One of the ways that this could be done is the SOLID principle but there are others
- Keep all systems and mechanics as modular as possible
 - Event based systems are great for this

Event Driven Systems

- Event driven systems are great for keeping things modular:
 - Some objects publish for events
 - Some objects listen for events
- Imagine: the stealth system breaks and never publishes an event
 - The combat system will not propagate the error or invalid behavior
- Imagine: the combat system breaks and never reads an event
 - The stealth system will continue to work as expected
- Try to keep your game as decoupled as possible



Event Driven Systems (cont.)

- Some game engines are designed to be completely event driven
 - Example: OpenAGE
 - <https://github.com/SFTtech/openage>
- Every interaction of the systems and mechanics is modeled with events
 - This keeps everything nicely decoupled and modular
- <https://youtu.be/b55W87UUHr8?si=atREIClwJqCIY1x4&t=916>

Programming Patterns for Modularity

- Strictly adhering to programming patterns can enforce modular and organized code:
 - Object oriented programming (OOP)
 - Core functionality exists in classes
 - Expanding functionality should involve adding more subclasses
 - Classes should depend on abstractions (abstract classes)
 - i.e. don't force the "Player" to be dependent on a Gun class, have it be dependent on an abstract "Weapon" class
 - Entity component systems (ECS)
 - Systems are inherently written separate
 - Decoupling revolves around ensuring systems do not modify the same data or rely on each other's data
 - Actor based systems
 - Old school method

