the gamedesigninitiative at cornell university

Lecture 3

Game Components

Starting Prompt

- What exactly is a **game engine**?
  - What libraries does it have to provide?
  - What tools need to come with it?
- What **skills** should an engine require?
  - Extensive programming experience (3110+)?
  - Minimal programming experience (1110)?
  - No programming experience?
  - Artistic ability (vs. paying for assets)?

So You Want to Make a Game?

- Will assume you have a *design document*
  - Focus of next week and a half…
  - Building off the ideas of previous lecture
- But now you want to start building it
  - Need to assign tasks to the team members
  - Helps to break game into *components*
  - Each component being a logical unit of work.

Traditional Way to Break Up a Game

- **Game Engine**
  - Software, created primarily by programmers
- **Rules and Mechanics**
  - Created by the designers, with programmer input
- **User Interface**
  - Coordinated with programmer/artist/HCI specialist
- **Content and Challenges**
  - Created primarily by designers

Features of Game Engines

- Power the graphics and sound
  - 3D rendering or 2D sprites
- Power the character and strategic AI
  - Typically custom designed for the game
- Power the physics interactions
  - Must support collisions at a bare minimum
- Describe the systems
  - Space of possibilities in game world

Commercial Game Engines

- Libraries that take care of technical tasks
  - But *systems* always need some specialized code
  - Game studios buy *source code licenses*
- Is LibGDX a game engine?
  - It has libraries for graphics, physics, and AI
  - But you still have to provide code for *systems*
- Bare bones engine: **graphics, physics, audio**

Game Engines: Graphics

- Minimum requirements:
  - API to import artistic assets
  - Routines for manipulating images

- Three standard 3D graphics APIs
  - **OpenGL**: Unix, Linux, Macintosh
  - **Direct3D**: Windows
  - **Vulkan**: The common future

- For this class, our graphics engine is LibGDX
  - Supports OpenGL, but will only use 2D

Game Engines: Physics

- Defines physical attributes of the world
  - There is a gravitational force
  - Objects may have friction
  - Ways in which light can reflect

- Does **not** define precise values or effects
  - The *direction* or *value* of gravity
  - Friction *constants* for each object
  - Specific *lighting* for each material

Game Engines: Systems

- Physics is an example of a game **system**
  - Specifies the *space of possibilities* for a game
  - But not the *specific parameters* of elements
- Extra code that you add to the engine
  - Write functions for the possibilities
  - But do not code values or when called
- Programmer vs. *gameplay designer*
  - Programmer creates the system
  - Gameplay designer fills in parameters

Systems: *Super Mario Bros.*

- **Levels**
  - Fixed height scrolling maps
  - Populated by blocks and enemies
- **Enemies**
  - Affected by stomping or bumping
  - Different movement/AI schemes
  - Spawn projectiles or other enemies
- **Blocks**
  - Can be stepped on safely
  - Can be bumped from below

- Mario (and Luigi) can be small, big, or fiery

Characteristics of an Engine

- Broad, adaptable, and extensible
  - **Encodes** all *non-mutable* design decisions
  - **Parameters** for all *mutable* design decisions
- Outlines gameplay **possibilities**
  - Cannot be built independent of design
  - But only needs highest level information
  - **Gameplay specification** is sufficient

Data-Driven Design

- No code outside engine; all else is data
  - Purpose of separating system from parameters
  - Create game content with **level editors**
- **Examples**:
  - Art, music in industry-standard file formats
  - Object data in JSON or other data file formats
  - Character behavior specified through scripts
- Major focus for alpha release

Popular Indie Engines

- All use data-driven design
- Most game code is *scripted*
  - Uses easy-to-learn language
  - Keeps coding at 1110-level
  - **Ex:** GDScript, UScript
- Systems coded separately
  - Uses a "real" language
  - **Godot:** C, **Unreal:** C++
- But Unity uses C# for both!
  - Was core to its appeal

Engine Tradeoffs

- This design has trade-offs
  - Most systems are built-in
  - Changing can be a **fight**
  - Or extremely inefficient
  - Designer has less control
- Why AAAs still **in-house**
- **Example:** Ubisoft Anvil
  - Large, open-world games
  - Ability to climb anywhere
  - This is all **core** gameplay
  - Cannot just add this on

Engine Tradeoffs

- This design has trade-offs
  - Most systems are built-in
  - Changing can be a **fight**
  - Or extremely inefficient
  - Designer has less control
- Why AAAs still **in-house**
- **Example:** Ubisoft Anvil
  - Large, open-world games
  - Ability to climb anywhere
  - This is all **core** gameplay
  - Cannot just add this on

Traditional Way to Break Up a Game

- **Game Engine**
  - Software, created primarily by programmers
- **Rules and Mechanics**
  - Created by the designers, with programmer input
- **User Interface**
  - Coordinated with programmer/artist/HCI specialist
- **Content and Challenges**
  - Created primarily by designers

Rules & Mechanics

- Fills in the values for the system
  - Parameters (e.g. gravity, damage amounts, etc.)
  - Types of player abilities/verbs
  - Types of world interactions
  - Types of obstacles/challenges
- But does not include **specific** challenges
  - Just the list all challenges that *could* exist
  - Contents of the *palette* for level editor

Rules: Super Mario Bros.

- **Enemies**
  - Goombas die when stomped
  - Turtles become shells when stomped/bumped
  - Spinys damage Mario when stomped
  - Piranha Plants aim fireballs at Mario

- **Environment**
  - Question block yields coins, a power-up, or star
  - Mushroom makes Mario small
  - Fire flower makes Mario big and fiery

Rules: Super Mario Bros.

- **Enemies**
  - Goombas die when stomped
  - Turtles become shells when stomped/bumped
  - Spinys dama
  - Pira
- **Environment**
  - Question block yields coins, a power-up, or star
  - Mushroom makes Mario small
  - Fire flower makes Mario big and fiery

Will be the topic of next few lectures

Game AI: Where Does it Go?

- Game AI is traditionally placed in **mechanics**
  - AI needs rules to make right choices
  - Tailor AI to give characters personalities
- But it is implemented by programmer
  - Search algorithms/machine learning
  - Shouldn't these be in **game engine**?

- Holy Grail: "AI Photoshop" for designers
  - Hides all of the hard algorithms

Traditional Way to Break Up a Game

- **Game Engine**
  - Software, created primarily by programmers
- **Rules and Mechanics**
  - Created by the designers, with programmer input
- **User Interface**
  - Coordinated with programmer/artist/HCI specialist
- **Content and Challenges**
  - Created primarily by designers

Interfaces

- Interface specifies
  - How player does things (player-to-computer)
  - How player gets feedback (computer-to-player)
- More than engine+mechanics
  - Describes what the player can do
  - Do not specify how it is done
- Bad interfaces can kill a game

Interface: *Dragon Age*

Interface: *Dead Space*

Designing Visual Feedback

- Designing for **on-screen** activity
  - Details are best processed at the center
  - Peripheral vision mostly detects motion
  - Visual highlighting around special objects
- Designing for **off-screen** activity
  - Keep HUD elements out of the center
  - Flash the screen for quick events (e.g. being hit)
  - Dim the screen of major events (e.g. low health)

Interface: *Witcher 3*

Other Forms of Feedback

- **Sound**
  - Player can determine type, distance
  - In some set-ups, can determine direction
  - Best for conveying action "off-screen"
- **Tactile** (e.g. Rumble Shock)
  - Good for proximity only (near vs. far)
  - Either on or off; no type information
  - Limit to significant events (e.g. getting hit)

Traditional Way to Break Up a Game

- **Game Engine**
  - Software, created primarily by programmers
- **Rules and Mechanics**
  - Created by the designers, with programmer input
- **User Interface**
  - Coordinated with programmer/artist/HCI specialist
- **Content and Challenges**
  - Created primarily by designers

Content and Challenges

- Content is **everything else**
- **Gameplay** content defines the actual game
  - Goals and victory conditions
  - Missions and quests
  - Interactive story choices
- **Non-gameplay** content affects player experience
  - Graphics and cut scenes
  - Sound effects and background music
  - Non-interactive story

Mechanics vs. Content

- **Content** is the layout of a specific level
  - Where the exit is located
  - The number and types of enemies
- **Mechanics** describe what these do
  - What happens when player touches exit
  - How the enemies move and hinder player
- Mechanics is the content *palette*

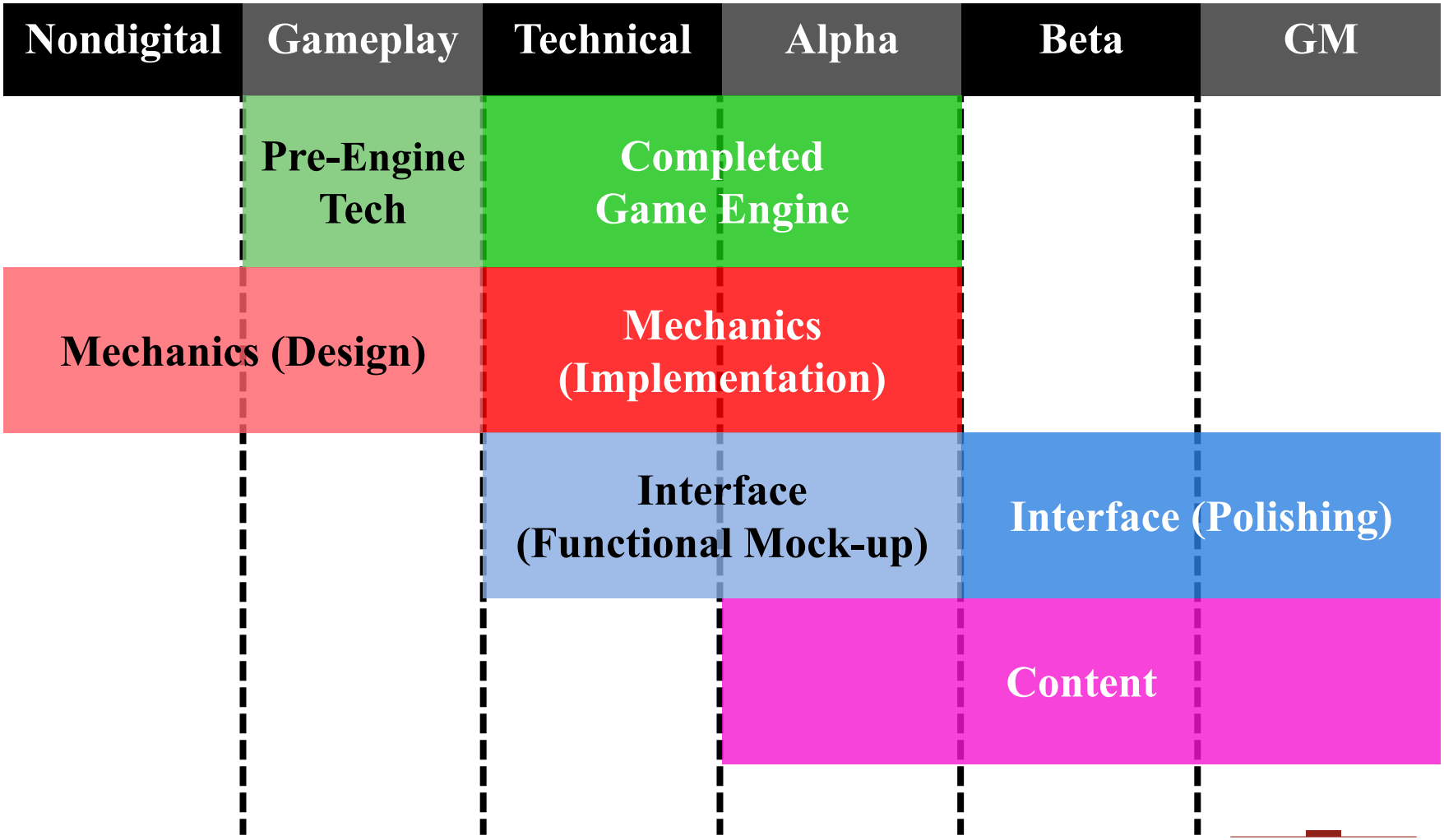
Mechanics vs. Content

Mechanics vs. Content

Why the Division?

- They are not developed sequentially
  - Content may requires changes to game engine
  - Interface is changing until the very end
- Intended to organize your design
  - **Engine**: decisions to be made early, hard-code
  - **Mechanics**: mutable design decisions
  - **Interface**: how to shape the user experience
  - **Content**: specific gameplay and level-design

Milestones Suggestions



Summary

- Game is divided into four components
  - Should keep each in mind during design
  - Key for distributing work in your group
- But they are all interconnected
  - System/engine limits your possible mechanics
  - Content is limited by the type of mechanics
- Once again: **design is iterative**