GAME PROGRAMMING

Contents

MODULE 1 1
MODULE 2 19
MODULE 7 33

MODULE 1

1. The Evolution of Gaming

Early History

- **First Video Game:** The very first video games were entirely hardware-based, meaning they relied on physical components rather than software. This limited the complexity and variety of games that could be created.

Technological Advancements

- **Microprocessor Revolution:** Rapid advancements in microprocessors have transformed the gaming landscape. These advancements have allowed for:
    - More complex game mechanics.
    - Enhanced graphics and audio.
    - The ability to create immersive worlds that players can explore.

Modern Gaming

- **Current Platforms:** Today, video games are played on versatile PCs and specialized consoles. These platforms use sophisticated software to provide a wide variety of gaming experiences.
- **50 Years of Progress:** It has been 50 years since the introduction of the first primitive games, marking significant progress in the industry.

2. Game Engines

Core Components

- **Game Engine Architecture:** Game engines vary widely in their architecture and implementation. However, they generally contain a familiar set of core components:

- **Rendering Engine:** Responsible for generating the visual output.
- **Collision and Physics Engine:** Manages interactions between objects and simulates realistic physics.
- **Animation System:** Controls the movement and behavior of characters and objects.
- **Audio System:** Manages sound effects and music.
- **Game World Object Model:** Represents all objects within the game environment.
- **Artificial Intelligence System:** Governs the behavior of non-player characters (NPCs) and their interactions with the player.

3. What is a Game?

Definition

- **Game as an Activity:** A game is a structured form of play, typically undertaken for entertainment, enjoyment, or educational purposes. It involves:
  - **Rules:** Guidelines that dictate how the game is played.
  - **Objectives:** Goals that players aim to achieve.
  - **Interaction:** Players interact with the game's environment or other players.

Types of Games

- **Physical vs. Digital:** Games can take many forms, including:
  - **Physical Games:** Such as sports (e.g., soccer).
  - **Digital Games:** Such as video games (e.g., *Warcraft*).

4. Game Programming

Definition and Process

- **Game Programming:** The process of designing and creating video games by writing code that controls gameplay, mechanics, and interactions within a game world.

Key Skills Required

- **Mathematics:** Essential for calculations related to physics, graphics, and game mechanics.
- **Physics:** Important for simulating realistic interactions and movements.
- **Graphics:** Knowledge of rendering techniques and visual design.

- **Artificial Intelligence (AI):** Techniques to create responsive and adaptive gameplay experiences.

5. What is a Computer Game?

Key Differences

- **Software Program:** Unlike board games or physical sports, a computer game is a software program.
    - **Example Comparison:** Consider the differences between chess (a board game), soccer (a physical sport), and *Warcraft* (a digital game).

Player Experience

- **Losses and Gains:**
    - **What You Lose:**
        - Physical pieces (e.g., game tokens, boards).
        - Social interaction in a physical space.
    - **What You Gain:**
        - Real-time gameplay.
        - Increased immersion and engagement.
        - Greater complexity in gameplay mechanics.

Audience Consideration

- **Player-Centric Design:** When designing a game, it’s crucial to consider the audience. The game is not just for the developer but for the players.
    - **Example:** A complicated flight simulator may be enjoyable for a flying enthusiast but could overwhelm a beginner.

6. Gameplay Dynamics

Decision Making

- **Making Choices:** Playing a game involves making decisions that impact gameplay (e.g., which weapon to use or what resources to gather).
    - **Frustration Factor:** If players feel their decisions do not matter, it can lead to frustration. Good gameplay ensures that choices have meaningful consequences.

Player Control

- **Impacting Outcomes:** Players want to feel that they can influence the game's outcome.
    - **Uncontrolled Sequences:** While unexpected events can happen, they should be infrequent and logical.

    - **Example:** In *Valorant*, agents have unique abilities that can control the battlefield, allowing players to strategically influence the game.

Game Goals

- **Need for Objectives:** Every game requires goals for players to strive towards.
    - **Example:** In *Zelda*, players must defeat Ganondorf, but there may also be sub-goals, such as recovering the Triforce or obtaining specific items.

7. What a Game is Not

Common Misconceptions

- **Not Just Features:** A game is not merely a collection of cool features. While features are necessary, they are insufficient on their own. Overemphasis on features can detract from the overall gaming experience.

Graphics vs. Gameplay

- **Fancy Graphics:** While graphics are important, they cannot compensate for weak gameplay. A game should be enjoyable even with simple visuals.
    - **Quote:** "When a designer is asked how his game is going to make a difference, I hope he talks about gameplay, fun, and creativity – as opposed to an answer that simply focuses on how good it looks." – Sid Meier

Puzzles and Stories

- **Not Just Puzzles:** While all games may include puzzles, they are not the essence of gameplay. Puzzles are specific challenges, whereas games involve broader systems and mechanics.
- **Story Importance:** A compelling story can enhance immersion, but it is not sufficient on its own. For example, in *Baldur's Gate*, a linear story may lead to death if players stray off the intended path, but it lacks interactivity if all actions lead to the same outcome.

Core Components of a Game Engine

Game engines are complex systems that provide the necessary tools and architecture for creating video games. Here are the core components:

1. Core Components

- **Rendering Engine**:
    - Responsible for generating the visual output of the game, including graphics and animations.

- **Collision and Physics Engine**:
    - Manages interactions between objects and simulates realistic physical behaviors (e.g., gravity, collision detection).
- **Animation System**:
    - Controls the movement and behavior of characters and objects, allowing for dynamic animations.
- **Audio System**:
    - Handles sound effects, background music, and voiceovers to enhance the gaming experience.
- **Game World Object Model**:
    - Represents all objects in the game environment, defining their properties and behaviors.
- **Artificial Intelligence System**:
    - Governs NPC behavior, decision-making, and interactions with players.

Design Alternatives

Within these components, there are semi-standard design alternatives that have emerged, allowing for flexibility and customization in game development.

Structure of a Typical Game Team

A game development team is composed of various roles, each contributing to the creation of a video game. The structure resembles a mini-Hollywood production.

1. Team Roles

Engineers

- **Role**: Design and implement the software that powers the game.
- **Categories**:
    - **Runtime Programmers**: Work on the engine and the game itself.
    - **Tools Programmers**: Develop offline tools for the development team.
- **Specializations**:
    - Some engineers focus on specific systems (e.g., rendering, AI, audio).
    - Others may work on gameplay programming or at a systems level.

        - **Generalists**: Engineers who can tackle various problems during development.
    - **Leadership**:
        - **Lead Engineers**: Design and write code, manage schedules, and make technical decisions.
        - **Technical Directors (TD)**: Oversee projects at a high level.
        - **Chief Technical Officer (CTO)**: Highest engineering position in a studio.

Artists

- **Role**: Produce all visual and audio content, crucial for the game's success.
- **Types of Artists**:
    - **Concept Artists**: Create sketches and paintings for the game's vision.
    - **3D Modelers**: Construct the 3D geometry of the game world, divided into foreground and background modelers.
    - **Texture Artists**: Create textures applied to 3D models for detail and realism.
    - **Lighting Artists**: Set up light sources, managing color, intensity, and direction.
    - **Animators**: Bring characters and objects to life through motion. They often use motion capture data.
    - **Sound Designers**: Collaborate with engineers to produce and mix audio.
    - **Voice Actors**: Provide character voices.
    - **Composers**: Create original music scores for the game.
- **Art Directors**: Senior artists who ensure consistency across the game's visual elements.

Game Designers

- **Role**: Design the interactive aspects of gameplay.
- **Levels of Detail**:
    - **Senior Game Designers**: Work on the overarching story arc, goals, and objectives.
    - **Level Designers**: Focus on specific areas, enemy placements, and puzzle designs.
- **Writers**: Collaborate with designers to construct the story and write dialogue.

- **Game Directors**: Oversee all design aspects, manage schedules, and ensure consistency.

Producers

- **Role**: Varies by studio but generally involves managing schedules and resources.
- **Responsibilities**:
    - May serve as human resources managers or senior game designers.
    - Act as liaisons between development and business units (finance, legal, marketing).

Support Staff

- **Role**: Crucial support team for the development process.
- **Includes**:
    - Executive management.
    - Marketing department.
    - Administrative staff.
    - IT department (provides hardware/software support).

Publishers and Studios

1. Role of Publishers

- **Function**: Handle marketing, manufacturing, and distribution of game titles.
- **Examples**: Large corporations like Electronic Arts, THQ, Vivendi, Sony, and Nintendo.

2. Game Studios

- Many game studios operate independently from publishers, selling their games to various publishers as needed.

This structured approach highlights the complexity of game development, showcasing the diverse roles and components that contribute to creating engaging and immersive gaming experiences.

Game Engine Overview

1. Definition and Architecture

Data-Driven Architecture

- **Differentiation from Simple Games**: A game engine is defined by its data-driven architecture, allowing it to serve as a flexible foundation for multiple games.
- **Hard-Coded Logic**: When a game relies on hard-coded logic or special-case code, it limits reusability.
- **Extensibility**: The term "game engine" should be reserved for software that is extensible and adaptable for various game types without major modifications.

Historical Context

- **Origins**: The term "game engine" emerged in the mid-1990s, particularly associated with first-person shooter (FPS) games like *Doom* by id Software.
    - **Gameplay**: Players assume the role of Doomguy, battling through levels from Mars to hell, showcasing early 3D graphics with 2D sprites (2.5D graphics).

---

2. Licensing and Modding

Architectural Separation

- **Core Components**: *Doom* featured a clear separation between its core software components (graphics, collision detection, audio) and the game's art assets and rules.
- **Mod Community**: This separation allowed for the licensing of games, leading to a mod community where gamers could create new experiences by modifying existing games, aided by toolkits from developers (e.g., *Quake III Arena*).

---

3. Engine Differences Across Genres

- **Genre Specificity**: Game engines are often tailored to specific genres. For example:
    - **Fighting Games**: Engines for two-player boxing games differ significantly from those used for MMOGs or FPS games.

---

4. Game Engine History

Key Milestones

- **Pac-Man**: Released in 1980, it remains one of the highest-grossing games, generating over $14 billion.
- **Hydro Thunder**: An arcade racing game released in 1999, known for its high-tech speedboats and diverse environments.

Notable Engines

- **Quake III Engine**: Developed for *Quake III Arena*, this engine focused on multiplayer FPS gameplay.
- **Unreal Engine**: Known for its rich toolset and community support, it has evolved to UE4.
- **Unity**: A versatile engine with support for C# scripting, widely used for cross-platform development.

---

5. Genre-Specific Engine Features

First-Person Shooter (FPS)

- **Examples**: *Unreal*, *Half-Life*, *Call of Duty*.
- **Focus**: Efficient rendering, responsive camera control, and high-fidelity animations.

Third-Person Games

- **Examples**: *Ratchet and Clank*, *Gears of War*.
- **Focus**: Puzzle elements, moving environmental objects, and complex camera systems.

Fighting Games

- **Examples**: *Tekken*, *Soul Calibur*.
- **Focus**: Fighting animations, hit detection, and character animations.

Racing Games

- **Examples**: *Gran Turismo*, *Mario Kart*.
- **Focus**: Background object techniques, camera collision, and track design.

Real-Time Strategy (RTS)

- **Examples**: *Warcraft*, *Starcraft*.
- **Focus**: Low-resolution characters, complex goal trees, and user interaction.

Massively Multiplayer Online Games (MMOG)

- **Examples**: *World of Warcraft*, *EverQuest*.
- **Focus**: Server-side management, client-side rendering, and network consistency.

---

6. Player-Authored Content

- **Definition**: Allows players to create content within the game, distinct from traditional mods.
- **Examples**:
    - *Little Big Planet*: Emphasizes user-generated content with the tagline "Play, Create, Share."
    - *Minecraft*: A sandbox game that thrives on player creativity.

---

7. Current Game Engines

Popular Engines

- **Quake Family**: Extends to modern titles; source code for earlier engines is freely available.
- **Unreal Engine**: Rich toolset and strong developer network.
- **Unity**: Feature-rich with extensive community support.
- **Source Engine**: Known for titles like *Half-Life 2*.
- **CryEngine**: Initially developed for *Far Cry*.
- **Sony PhyreEngine**: Powerful graphics capabilities.
- **Frostbite**: Used for *Battlefield* series, includes asset creation tools.
- **Microsoft XNA and MonoGame**: Based on C#, designed for Xbox and PC but no longer supported.

2D Engines

- **Purpose**: Designed for non-programmers to create apps for mobile platforms.
- **Examples**: Multimedia Fusion 2, Game Salad Creator, Scratch.

Game Engine Architecture

1. Runtime Engine Architecture

- **Components**:
    - Comprises a tools suite and runtime components.
    - Spans from hardware to high-level applications.
    - Designed in layers to avoid circular dependencies, maximizing reuse and testability.

2. Runtime Engine Components

Low-Level Components

- **Hardware**: The system on which the game runs.

- **Device Drivers**: Shield the OS and upper layers from low-level device communication details.
- **Operating System**: Manages execution and interruption of multiple programs, typically thin on consoles.

Third-Party SDKs

- **Data Structures and Algorithms**:
    - **STL**: C++ Standard Template Library for data structures and strings.
    - **Boost**: Offers powerful data structures and algorithms, useful for handling large numbers.
- **Graphics**:
    - **OpenGL** and **DirectX** for rendering.
- **Collisions and Physics**: Libraries like **Havok**, **PhysX**, **ODE**, and **Bullet**.
- **Character Animation**: Tools for animating characters.
- **Artificial Intelligence**: Tools like Kynapse for AI data generation and optimizations for multicore architectures.

Platform Independence Layer

- **Purpose**: Allows engine development without concern for the underlying platform.
- **Features**: Provides wrappers for common operations like primitive types, networking, and file systems.

Core Systems

- **Assertions**: Error checking code.
- **Memory Management**: Efficient handling of memory resources.
- **Math Library**: Vector and matrix math, numeric integrators.
- **Custom Data Structures**: Tailored structures for game needs.

Resource Manager

- **Function**: Provides a unified interface for accessing assets.
- **Complexity**: Varies based on needs; can involve direct resource loading or complex asset manipulation.

Rendering Engine

- **Components**:
    - Scene graph management.
    - Visual effects.

    - Front-end rendering.
- **Rendering Process**: Creation of shaded images from 3D models, with animation techniques to create motion.

Graphics Objects

- **Game Objects**: Entities to which components are attached.
- **Canvas**: UI layer displaying game scores.
- **Layers**: Organize game objects, with visibility control.
- **Sprites**: 2D objects with graphical textures.
- **Renderer**: Makes objects visible on screen.

Low-Level Renderer

- **Focus**: Renders primitives quickly and richly without visibility considerations.
- **Components**:
    - Graphics Device Interface (GDI) for device access.
    - Representation of geometric primitives.
    - Camera interface abstraction.
    - Material and dynamic lighting systems.

Scene Graph

- **Purpose**: Limits the number of primitives submitted for rendering.
- **Techniques**: Uses frustum culling and spatial subdivision methods (BSP, quadtree, octree).

Visual Effects

- **Includes**:
    - Particle systems.
    - Decal systems.
    - Light mapping.
    - Dynamic shadows.
    - Full-screen post effects.

Front End

- **Components**:
    - HUD, menus, GUI for character manipulation, and full-motion video for cut scenes.

Profiling/Debugging

- **Tools**:
    - Code timing, performance stats display, memory usage tracking, event recording, and print statement control.

Collisions and Physics

- **Dynamics**: Primarily focuses on rigid body dynamics.
- **Physics Engine**: Often uses libraries like Havok, PhysX, ODE, and Bullet.

Animation

- **Types**:
    - Sprite/texture animation.
    - Rigid body hierarchy animation.
    - Skeletal animation (most popular).
    - Vertex animation and morphing.

Human Interface Devices

- **Support for**:
    - Keyboard, mouse, joypads, and specialized controllers.
- **Function**: Transforms raw data into useful information.

Audio

- **Importance**: Often overlooked until late in development, varies in sophistication.
- **Tools**: Many games utilize existing audio tools like XACT and Scream.

Multiplayer/Networking

- **Types**:
    - Single screen, split-screen, networked multiplayer, and MMOGs.
- **Challenges**: Converting single-player to multiplayer is difficult; the reverse is easier.

Gameplay Foundation System

- **Components**:
    - World loading, game object model, static world elements, and real-time agent simulations.

Event System

- **Purpose**: Facilitates communication between objects through a common messaging system.

Scripting System

- **Function**: Integrates a scripting language for rapid game-specific development without recompiling the engine.

Artificial Intelligence Foundations

- **Components**:
    - Provides AI building blocks like path planning and navmesh generation.

Game-Specific Subsystems

- **Definition**: Contains all specific elements required for a game, considered outside the core game engine.

---

3. Tools and Asset Pipeline

Digital Content Creation Tools (DCC)

- **Purpose**: Create various game assets like 3D meshes, textures, sounds, and animations.
- **Examples**: Maya, 3ds Max, Photoshop, SoundForge.
- **Requirements**: Must be user-friendly and reliable.

Assets Conditioning Pipeline

- **Optimization**: DCC tools produce file formats that may not be optimized for games. Game engines typically store data in platform-specific formats for efficiency.

Conditioning Assets in Game Development

1. Types of Asset Conditioning

3D Model/Mesh Data

- **3D Models**: Must be properly tessellated (a pattern of repeated shapes).
- **Brush Geometry**: A collection of convex hulls with multiple planes, used for level design and environment creation.

Skeletal Animation Data

- **Compression and Conversion**: Skeletal animation data must be compressed and converted to function correctly within the engine.

Audio Data

- **Format Standardization**: Audio assets should be converted to a single format suitable for the target system, ensuring compatibility and performance.

Particle System Data

- **Custom Editing Tools**: Particle systems usually require custom editing tools integrated into the engine for effective manipulation.

---

2. Game World Editor

- **Integration**: Typically integrated into the engine, allowing game designers to create and modify game worlds.
- **Examples**:
  - **UnrealEd**: The editor for Unreal Engine.
  - **Hammer**: The level editor for the Source engine.
  - **Radiant**: Used for various games, including Quake.

---

3. Resource Database

- **Data Management**: A system is needed to store and manage vast amounts of game data.
- **Options**:
  - **Relational Databases**: Some companies utilize databases like MySQL or Oracle.
  - **Version Control Software**: Tools like Subversion, Perforce, or GIT are used to track changes and manage assets.
  - **Custom Software**: For example, Naughty Dog uses a custom GUI called Builder for asset management.

---

4. Approaches to Tool Architecture

Stand-Alone Tools Architecture

- Tools operate independently of the game engine.

Shared Framework Tools

- Tools built on a framework shared with the game engine, allowing for better integration.

Web-Based Tools

- **Uses**:
  - Asset management, scheduling, bug management.
- **Advantages**:
  - Easier to build and update without requiring a complete reinstall.
  - Ideal for presenting tabular data and forms.

5. Version Control

- **Definition**: A version control system allows multiple users to collaborate on files, maintaining a history of changes.
- **Importance**:
    - Essential for team development, providing a central repository for source code.
    - Keeps track of changes, allows tagging of specific versions, and supports branching for demos or patches.

---

6. Engine Support Systems

- **Low-Level Support**: Every game engine needs systems to manage essential tasks, such as:
    - Starting up and shutting down the engine.
    - Configuring engine and game features.
    - Managing memory usage.
    - Handling file system access and diverse asset types (meshes, textures, audio, etc.).
    - Providing debugging tools for developers.

Subsystem Start-Up and Shut-Down

- **Initialization Order**: Subsystems must be configured and initialized in a specific order based on their interdependencies.
- **Shutdown Order**: Typically occurs in the reverse order of startup.

---

7. Design Patterns

- **Definition**: Design patterns arise when similar problems are solved in similar ways by different programmers.
- **Common Patterns**:
    - **Singleton**: Ensures a class has only one instance and provides global access to it.
    - **Iterator**: Allows efficient access to elements of a collection without exposing its implementation.
    - **Abstract Factory**: Provides an interface for creating related classes without specifying their concrete implementations.

Need for Singleton Class for Proper Startup and Shutdown

- **C++ Object Construction**: In C++, global and static objects are constructed unpredictably before the program's entry point.
- **Destruction Order**: Destructors are called after the program returns, also in an unpredictable order.
- **Solution**: Define a singleton class (often called a manager) for each major subsystem to ensure consistent startup and shutdown processes.

Memory Management in Game Development

1. Optimizing Dynamic Memory Allocation

Performance Impact

- **Dynamic Memory Allocation**: Allocating memory during runtime can significantly affect performance.
- **Memory Access Patterns**: Access patterns should be optimized to ensure small data remains in continuous memory rather than being spread across different locations.

Limitations

- **Context Switch**: Switching from user mode to kernel mode for memory requests can be costly.

Solutions

- **Custom Allocators**:
    - Use preallocated memory blocks.
    - Operate entirely in user mode, avoiding context switch costs.

---

2. Types of Allocators

Stack-Based Allocators

- **Mechanism**: Allocate a large contiguous block of memory and maintain a pointer to the top of the stack.
- **Efficiency**: Fast allocation and deallocation by adjusting the top pointer.

Double-Ended Stack Allocators

- **Structure**: Contains two stack allocators within a single memory block.
    - **Example**: In **Hydro Thunder**, one stack is for loading/unloading levels, and another for temporary memory blocks allocated and freed every frame.
- **Benefit**: Prevents memory fragmentation effectively.

Pool Allocators

- **Usage**: Ideal for applications requiring many small memory blocks.
- **Mechanism**: Preallocate a large block of memory, divided into fixed-size slots.
- **Example**: For 256 bullets of 32 bytes each:
    - Total allocation: 256×32=8192256×32=8192 bytes.

---

3. Single-Frame and Double-Buffered Memory Allocators

Single-Frame Allocators

- **Implementation**: A stack-based allocator that resets at the beginning of each frame.
- **Limitation**: Allocated memory is only valid for the current frame.

Double-Buffered Allocators

- **Functionality**: Allows memory allocated in frame i*i* to be used in frame i+1*i*+1.
- **Example**: Memory from frame n*n* is cleared at the end of frame n+1*n*+1.
- **Use Case**: Useful for passing information like velocity calculations between frames.

---

4. Memory Fragmentation

Problem

- **Fragmentation**: Occurs when memory becomes divided into small, unusable holes, leading to allocation failures despite sufficient total free memory.
- **Virtual Memory**: While it can help, many game engines do not utilize it.

Solutions

- **Stack and Pool Allocators**:
    - Stack allocators avoid fragmentation by ensuring contiguous allocations.
    - Pool allocators prevent fragmentation issues as all blocks are the same size, ensuring allocation requests do not fail due to lack of contiguous space.

---

5. Containers in Game Development

Benefits of Containers

- **Portability**: Move applications and dependencies across different environments seamlessly.
- **Standardization**: Ensures consistent deployment procedures using tools like Docker.
- **Faster Deployment**: Reduces code changes needed for different environments.
- **Flexibility**: Supports deployment in various hosting environments (on-premise, public cloud, private cloud).

Example

- **GameObjects in Unity**: Serve as containers for components that implement functionality.

---

6. Strings in Game Development

Challenges with Strings

- **Storage and Management**: Strings in C/C++ are implemented as arrays of characters, leading to various design issues.
- **Localization**: Adapting software for different languages, requiring translation and handling of different text orientations.

Performance Considerations

- **String Operations**: Comparing strings can be costly, requiring O(n) scans.

Unicode

- **Limitations of ANSI**: ANSI strings are insufficient for languages with complex alphabets.
- **Unicode**: Provides a comprehensive character set to accommodate various languages.

---

7. Example of Localization Issues

- **Use Case**: For displaying scores and win messages:
    - Store strings like "Player 1 Score:" and "Player 2 Wins" in a localization database with unique IDs (e.g., "p1score," "p2wins") for easy access and translation.

MODULE 2

2D and 3D Graphics Overview

Introduction

- **Types of Graphics**: Games can be categorized into 2D, 3D, or a combination of both.
- **Core Knowledge**: Mathematics and computer architecture are fundamental to understanding graphics, including colors, virtual objects, and interactions.

---

1. Bits and Bytes

Memory Units

- **Bit**: The smallest unit of memory, representing a binary value (0 or 1).
- **Byte**: Consists of 8 bits (0-255), used for storing data such as ASCII characters.

Common Data Types

Data Type	Size
Byte	8 bits (1 byte)
Integer	32 bits (4 bytes)
Short	16 bits (2 bytes)
Float	32 bits (4 bytes)

Understanding Bits

- **Binary Representation**: Each additional bit doubles the number of values that can be represented. For example:
  - 1 bit: 2 values (0-1)
  - 2 bits: 4 values (0-3)
  - 8 bits: 256 values (0-255)

---

2. Memory Bytes

Byte Representation

- **Character Encoding**: Each character in a string is represented by a byte.
- **Endianess**: Refers to byte ordering in memory:
  - **Little-Endian**: Most significant byte at the lowest address.

- **Big-Endian**: Most significant byte at the highest address.

Importance of Endianess

- Different systems (e.g., Intel vs. PowerPC) may require byte swapping for data compatibility.

---

3. Hexadecimal Values

Base-16 System

- **Hexadecimal**: Represents values using 0-9 and A-F (0-15).
- **Example**: The hex value "BA" equals 186 in decimal.

Hex Triplet in Colors

- Commonly used in web design (e.g., HTML) to specify colors:
  - **Format**: #RRGGBB
  - Example: #68A3F8 translates to RGB values.

---

4. Color Ranges

Human Perception

- The human eye can perceive a wide range of colors and luminance values, making realistic color representation crucial in graphics.

Low-Dynamic-Range (LDR) Colors

- **LDR Rendering**: Uses a maximum of 8 bits per color component, limiting the range and precision. This can lead to artifacts and loss of detail.

High-Dynamic-Range (HDR) Colors

- **HDR Rendering**: Uses more bits per color component, allowing for higher precision and a broader range of values. Tone mapping is applied to adapt HDR data for display.

---

5. 2D Graphics

Evolution of 2D Graphics

- 2D graphics were predominant in early gaming but have seen a resurgence with the rise of indie games and casual gaming.

Sprites

- **Definition**: 2D rectangular images used in games.
- **Types**:

- **Static Sprites**: Non-animating images (e.g., characters).
- **Dynamic Sprites**: Collections of images that form animations, similar to flip-books.

Conclusion

Understanding the fundamentals of bits, bytes, hexadecimal values, and color ranges is essential for mastering 2D and 3D graphics in game development. Mastery of these concepts will be built upon in subsequent chapters, focusing on rendering techniques and the mathematics behind graphics.

3D Graphics Overview

Introduction

3D graphics have become the cornerstone of modern game development, prevalent across various platforms including PCs, consoles, and mobile devices. The focus often lies in achieving realistic real-time graphics, although stylized graphics, like those in *The Legend of Zelda: The Wind Waker*, are also common.

---

1. The Z Buffer

Depth Information

- **Z Buffer**: A critical component in rendering 3D graphics that stores depth information for each pixel.
- **Purpose**: Helps determine which objects are in front of others during rendering, a process known as depth testing.

Depth Testing

- In 2D games, sprites can be drawn in a specified order. In contrast, 3D surfaces can have varying depths, making depth testing essential for accurate rendering.

---

2. Shading Surfaces

Dynamic Surface Information

- 3D games dynamically calculate surface information, allowing for realistic shading and lighting effects.
- **Real-time Lighting**: Enhances the appearance of surfaces by calculating light interactions dynamically.

Importance of Shading

- The shading of surfaces is fundamental to creating visually appealing 3D graphics, with various algorithms employed throughout the rendering process.

---

3. Geometry and Primitives

Geometric Entities

- **Primitives**: Basic geometric shapes that represent objects in a 3D scene, processed by graphics hardware through the rendering pipeline.

Common Primitives

1. **Lines**: Defined by two points, lines can represent simple connections in 2D or 3D space.

cpp

Copy

```
struct Point {
    int x;
    int y;
};
struct Line {
    Point start;
    Point end;
};
```

2. **Polygons**: Composed of three or more points, polygons enclose an area and are used extensively in 3D graphics.

cpp

Copy

```
struct Polygon {
    int total_points;
    Point points[];
};
```

3. **Triangles**: The most commonly used primitive in 3D graphics, triangles are efficient for rendering due to hardware optimization.
    - **Triangle Lists**: Individual triangles specified in an array.
    - **Triangle Strips**: A series of connected triangles that reduce data overhead.

- **Triangle Fans**: Triangles that share a common vertex.

Convex vs. Concave Polygons

- **Convex**: No internal angles greater than 180 degrees; easier for calculations and collision detection.
- **Concave**: Has internal angles greater than 180 degrees; more complex and less efficient for calculations.

---

4. Spheres and Boxes

Bounding Volumes

- **Spheres**: Defined by a position and radius, used for quick collision detection.

cpp

Copy

```
struct Sphere {
    int radius;
    Point position;
};
```

- **Boxes**: More accurate bounding volumes than spheres, defined by width, height, and depth.

cpp

Copy

```
struct Box {
    int width;
    int height;
    int depth;
    Point position;
};
```

Collision Detection

- Simplified shapes like spheres and boxes are used for fast collision tests, improving performance in complex scenes.

---

5. Additional Geometric Objects

Mathematical Objects

- Various geometric objects can be represented mathematically, including:
    - Cones
    - Pyramids
    - Cylinders
    - Toruses
    - Bezier curves

Loading Geometry

- Geometry is typically loaded from external files during runtime, such as static models, animations, and terrains. The Wavefront OBJ file format is commonly used for this purpose.

---

6. Mathematics in Computer Graphics

Vectors and Vertices

- **Vectors**: Fundamental mathematical objects representing direction and magnitude in space.
    - **Types**: 2D, 3D, and 4D vectors, with operations like addition, subtraction, and normalization.

cpp

Copy

```
struct Vector3D {
    float x, y, z;
};
```

Common Operations

1. **Magnitude**: Length of a vector, calculated using the Pythagorean theorem.

plaintext

Copy

```
Magnitude = √(x² + y² + z²)
```

2. **Dot Product**: Measures the angle between two vectors, useful for lighting calculations.

plaintext

Copy

```
d = (V1.x * V2.x + V1.y * V2.y + V1.z * V2.z)
```

3. **Cross Product**: Produces a vector perpendicular to two input vectors, often used to determine surface normals.

plaintext

Copy

```
normal = normalize(cross_product(e1, e2))
```

Normals

- Normals are unit-length vectors perpendicular to surfaces, essential for lighting calculations.

Transformations in 3D Graphics

Overview

In 3D graphics, data must be transformed from object space to world space and finally to screen space for rendering. This involves using various coordinate spaces and mathematical matrices to manage the relationships between objects in a scene.

---

1. Coordinate Spaces

Object Space

- **Definition**: The local coordinate system of a model, where vertices are defined relative to the model itself.

World Space

- **Definition**: The global coordinate system where all objects are positioned and oriented within the scene.
- **Purpose**: Allows for efficient rendering by transforming object-space models to their appropriate locations in the scene.

Screen Space

- **Definition**: The 2D representation of the 3D scene on the display, aligned with the X and Y axes of the screen.
- **Projection**: A projection matrix is applied to convert the 3D coordinates into 2D coordinates, adding perspective.

Types of Projection

1. **Orthogonal Projection**:
    - Objects maintain size regardless of distance from the camera.
2. **Perspective Projection**:
    - Objects appear smaller as they move farther from the camera, simulating depth.

---

2. Matrices

Definition

- **Matrix**: A mathematical structure used to store information about transformations, such as translation, rotation, and scaling.
- **Types**: Commonly used matrices in 3D graphics include 3x3 and 4x4 matrices.

Operations with Matrices

- **Addition and Subtraction**: Performed element-wise on matrices of the same size.
- **Multiplication**: More complex; involves multiplying rows of the first matrix by columns of the second matrix.

Identity Matrix

- An identity matrix has ones on the diagonal and zeros elsewhere. Multiplying any vector by an identity matrix leaves it unchanged.

Transformations

- **Translation**: Positioning an object in space by modifying the last row of a 4x4 matrix.
- **Scaling**: Adjusting the size of an object by changing the diagonal elements of the matrix.
- **Rotation**: Rotating an object around an axis using a rotation matrix.

---

3. Rays

Definition

- **Ray**: A mathematical object defined by an origin and a direction, extending infinitely in one direction.
- **Uses**: Commonly used for collision detection, visibility tests, and rendering techniques like ray tracing.

Ray Casting

- **Purpose**: To test line-of-sight between objects or to select objects in a scene based on user input.

Ray Structure

cpp

Copy

```
struct Ray {
    Vector3D origin;
```

```
    Vector3D direction;
};
```

---

4. Planes

Definition

- **Plane**: An infinite flat surface defined by a normal vector and a point on the plane.

Plane Equation

- The equation of a plane can be expressed as: $Ax+By+Cz+D=0$ $Ax+By+Cz+D=0$ where AA, BB, and CC are the components of the plane's normal vector.

Classifying Points

- Determine which side of a plane a point lies on using the plane equation.

Intersections

- **Line-Plane Intersection**: Check if a line intersects a plane by classifying the endpoints of the line.
- **Triangle-Plane Intersection**: Check if any vertex of a triangle is on a different side of the plane than the others.

---

5. Frustums

Definition

- **Frustum**: A volume defined by multiple planes, used to determine visibility in a scene.

View Frustum

- Represents what the camera can see, defined by near and far planes, as well as left, right, top, and bottom planes.

Culling

- **Frustum Culling**: A technique to avoid rendering objects that are outside the view frustum, improving performance.

---

6. Occlusion Culling

Definition

- **Occluder**: An object that blocks the view of other objects.

- **Occlusion Culling**: A technique to avoid rendering objects that are blocked from view by other objects.

---

7. Quaternion Rotations

Definition

- **Quaternion**: A mathematical object used to represent rotations in 3D space, consisting of four components: w, x, y, and z.

Advantages of Quaternions

- More efficient memory usage and faster rotation calculations compared to matrices.
- Avoids gimbal lock, providing smoother interpolations.

Quaternion Operations

- **Multiplication**: Not commutative; used for combining rotations.
- **Interpolation**: Quaternions can be linearly or spherically interpolated for smooth transitions.

Rotation with Quaternions

- Rotating a quaternion around an axis can be represented mathematically, providing efficient rotation management in 3D graphics.

Module 2 Overview

Content

1. **2D Graphics**: Sprites, Tiled Images, and Backgrounds
2. **3D Graphics**:
    - 3D Graphics Pipeline
    - 3D Math
    - Coordinates and Coordinate Systems
    - Quaternion Mathematics
    - Transformations & Geometry
    - Rendering Pipeline

2D Graphics

Introduction

- **Evolution**: 2D graphics have been integral to gaming since its inception. The methods for generating and displaying 2D images have advanced significantly.

- **Historical Context**: Prior to the 3D era (Nintendo 64, PlayStation), most console games were 2D.

Sprites

- **Definition**: A sprite is a 2D image representing objects or elements in a video game.
- **Types**:
    - **Static Sprite**: A single, non-animating image.
    - **Dynamic Sprite**: A series of images that create an animation, akin to a flipbook.

Steps to Create and Use Sprites in Unity

1. **Creating Sprites**:
    - Use tools like Adobe Photoshop or GIMP.
    - Save as PNG for transparency.
2. **Importing Sprites into Unity**:
    - Drag the PNG file into Unity's Assets folder.
    - Configure settings in the Inspector.
3. **Using Sprites**:
    - Create a new scene.
    - Drag the sprite into the scene and adjust its position and scale.
4. **Animating Sprites**:
    - Create a sprite sheet and import it.
    - Slice the sprite sheet and create an animation clip.
5. **Scripting for Interaction**:
    - Write scripts to control sprite movement and animations.

Tiled Images and Backgrounds

- **Usage**: Tiled images create environments and backgrounds by repeating patterns.
- **Benefits**:
    - Efficient for large, scrolling environments.
    - Allows for flexibility in design.

Steps to Create Tiled Images in Unity

1. **Creating Seamless Textures**: Design textures that tile seamlessly.

2. **Importing Textures**: Drag PNG texture files into Unity and set them to repeat.
3. **Using Tiled Textures**: Create a Quad or Plane for the background and apply the material.
4. **Dynamic Tiling (Optional)**: Write scripts to adjust tiling based on object size.

3D Graphics

Introduction

- **Focus**: Achieving realistic real-time graphics is a primary goal in 3D game development.

The Z Buffer

- **Function**: Stores depth information for each pixel, crucial for determining which objects are in front during rendering.

Shading Surfaces

- **Advantage**: 3D games can dynamically calculate surface information, allowing for realistic lighting and shading effects.

Geometry and Primitives

- **Types**:
    - **Lines**: Defined by two points.
    - **Polygons**: Made up of three or more points.
    - **Triangles**: Most common primitive due to efficiency in processing.

Types of Triangles

1. **Triangle Lists**: Individual triangles specified in an array.
2. **Triangle Strips**: Defined by the first three points and additional points for subsequent triangles.
3. **Triangle Fans**: Connect to a common point, creating various shapes efficiently.

Convex vs. Concave Polygons

- **Convex**: No internal angles greater than 180 degrees; better for collision detection.
- **Concave**: Has internal angles greater than 180 degrees; more complex for calculations.

Spheres and Boxes

- **Use**: Commonly employed for collision detection and physics calculations.
- **Efficiency**: Testing simpler shapes (like spheres) is faster than complex models.

Additional Geometric Objects

- Includes cones, cylinders, toruses, and more.

Mathematics in Computer Graphics

- **Vectors**: Fundamental for defining directions and points in space.
- **Operations**: Include addition, subtraction, and dot/cross products.

Transformations

- **Coordinate Spaces**: Objects are transformed from object space to world space and finally to screen space.
- **Types of Transformations**:
  i. **Translation**: Moving an object.
  ii. **Rotation**: Rotating an object around an axis.
  iii. **Scaling**: Changing the size of an object.

Projections

- **Orthogonal Projection**: Maintains size, no depth perception.
- **Perspective Projection**: Objects appear smaller with distance, creating depth perception.

Rendering Pipeline

- **Objective**: Draw virtual 3D objects onto a 2D screen.
- **Components**: Include modeling transformations, viewing transformations, and clipping.

Rays

- **Definition**: A ray has an origin and direction, used for visibility tests and rendering.
- **Ray Tracing**: Involves casting rays to determine visibility and lighting.

Scene Graph

- **Structure**: A data structure for managing 3D scenes, consisting of nodes that represent objects and transformations.
- **Usage**: Facilitates rendering and can optimize collision detection.

Camera Modeling

- **Parameters**: Include field of view, depth of field, and clipping planes.
- **Projection Matrix**: Encapsulates camera parameters for rendering.

Rasterization

- **Process**: Converts 3D shapes into 2D images by breaking shapes into triangles and filling pixels.
- **Clipping**: Ensures only visible parts of objects are rendered.

Scan-Line Conversion

- **Definition**: Defines a bounding rectangle for polygons and processes each row of pixels.

Performance Issues

- **Pixel Overdraw**: Occurs when multiple primitives are rendered in the same pixel space, leading to inefficiencies.

MODULE 7

GAME DESIGN

Definition

- **Game Design**: The art of **creating a game**, encompassing **goals, rules, challenges**, and aesthetics for various purposes such as **entertainment, education, and experimentation**.

Key Elements

- **Goals**: Objectives players aim to achieve.
- **Rules**: Guidelines that govern gameplay.
- **Challenges**: Obstacles players must overcome.

Types of Games

1. **Board Games**: Involves pieces moved on a board according to rules.
2. **Casino Games**: Players gamble on random outcomes.
3. **Role-Playing Games (RPGs)**: Players assume character roles in fictional settings.

Sample Game Design: Tetris

- **Overview**: Created in 1985 by Alexey Pajitnov, Tetris is a puzzle game where players arrange falling blocks to clear lines.
- **Goals**:

    i. Move and rotate blocks to fill lines.
    ii. Clear lines to score points.
- **Rules**:
    i. Game ends if blocks reach the top.
    ii. Lines are cleared when fully filled.

Feedback Mechanism

- Players receive immediate feedback through points scored and game over conditions.

Player Engagement

- **Hook**: The desire to play Tetris over other activities.
- **Obstacles**: Game speeds up as players improve, increasing difficulty.

The Five Domains of Play

1. **Novelty**: Players seek variety and unexpected elements.
2. **Challenge**: High-challenge players enjoy difficult tasks.
3. **Stimulation**: Preference for social interaction in games.
4. **Harmony**: Cooperative games foster social connections.
5. **Threat**: Players with high neuroticism thrive in high-pressure situations.

Gender Considerations in Gaming

- **Market Demographics**: More adult women (31%) play games than teenage boys (19%).
- **Character Representation**: Increasing focus on female protagonists in games.
- **Design Considerations**:
    - Understand differing learning styles and attitudes towards risk.
    - Avoid designing games solely for one gender; create engaging content for all.

Game Types / Genres

- **Definition**: Genres classify games based on **gameplay characteristics, objectives, and storylines**.

Game Genres

1. **Action**:
    - **Stealth Games**: Emphasize strategy over brute force (e.g., Metal Gear).
    - **Survival Games**: Players gather resources in hostile environments.
    - **Shooter Games**: Focus on combat at a distance.
    - **Platform Games**: Navigate through obstacles in 3D environments.
2. **Adventure**:
    - **Graphic Adventures**: Narrative-driven games with puzzles.
    - **Interactive Adventures**: Based on movies or novels.
    - **Real-time 3D Adventures**: Immersive experiences in fictional worlds.
3. **Role-Playing Games (RPGs)**:
    - Players assume specific roles and progress through stories.
    - **Sub-genres**:
        - **Roguelike**: Procedurally generated levels (e.g., Nethack).
        - **Fantasy RPGs**: Explore magical worlds (e.g., Final Fantasy).
        - **Sandbox RPGs**: Open-world exploration (e.g., GTA series).
        - **MMORPGs**: Massively multiplayer online interactions.
4. **Simulation Games**:
    - Replicate real-life scenarios (e.g., farming, vehicle management).
5. **Strategy Games**:
    - **4X Games**: Explore, expand, exploit, exterminate.
    - **Real-time Strategy (RTS)**: Continuous decision-making (e.g., Age of Empires).
    - **Tower Defense**: Protect structures from waves of enemies.
    - **MOBA**: Players control single characters in team battles.
6. **Sports Games**:
    - Emulate traditional sports gameplay (e.g., FIFA, NBA).
7. **Scientific and Educational Games**:

- Designed to educate while entertaining.

GAME MODES

A **game mode** is a specific configuration that alters gameplay, affecting how various game mechanics function. Different modes can provide unique settings and change the way players interact with the game, enhancing the overall experience.

Common Examples

- **Single Player vs. Multiplayer**: The most prevalent distinction, where multiplayer can be cooperative or competitive.
- **Dynamic Mode Changes**: Switching modes during gameplay can increase difficulty or serve as a reward for player success.

Mode Types

1. **Hard Mode**
   - **Definition**: A gameplay mode designed for expert players, featuring significant changes in gameplay mechanics.
   - **Example**: In *Doom*, the "Nightmare" difficulty increases the number of enemies, their speed, and attack frequency.
2. **Easy Mode**
   - **Definition**: A mode that significantly alters gameplay to accommodate inexperienced players, often allowing them to enjoy the story.
   - **Example**: In *New Super Mario Bros. Wii*, failing multiple times unlocks the "Super Guide," which demonstrates how to complete levels.
3. **Permadeath Mode**
   - **Definition**: Introduces permadeath, where players lose their character permanently upon death.
   - **Example**: *Diablo II* features a "hardcore mode" with permadeath, while *You Have to Win the Game* has a "YOLO mode."
4. **Story Mode**
   - **Definition**: Builds a narrative around core gameplay, often including cutscenes and boss fights.
   - **Example**: *Pinball Quest* adds RPG elements to traditional pinball gameplay, including characters and dialogue.
5. **Puzzle Mode**

- **Definition**: Focuses primarily on puzzle-solving elements, often reducing action or story components.
- **Example**: *Tetris Attack* presents specific configurations that players must clear with limited moves.

6. **Multiplayer Mode**
   - **Definition**: Transforms a single-player game into a multiplayer experience, significantly changing gameplay dynamics.
   - **Example**: *Age of Empires* features various multiplayer modes like versus, co-op, and deathmatch alongside its single-player story mode.
7. **Single Player Mode**
   - **Definition**: Adapts a multiplayer game for solo play, altering mechanics to suit a single player.
   - **Example**: *Left 4 Dead* includes modes like "Last Man on Earth" that remove teammates and adjust gameplay.
8. **Endless Mode**
   - **Definition**: Allows players to continue playing indefinitely without a defined endpoint.
   - **Example**: Common in falling block games like *Tetris* and *Puyo Pop*.
9. **Speed Run Mode**
   - **Definition**: Tracks the time taken to complete the game, encouraging players to beat their records.
   - **Example**: *Axiom Verge* has a "Speedrun Mode" that tracks completion times and milestones.
10. **Time Attack Mode**
    - **Definition**: Challenges players to achieve specific goals in the least amount of time.
    - **Example**: Many *Tetris* games feature a "sprint mode" where players aim to clear 40 lines as quickly as possible.
11. **High Score Mode**
    - **Definition**: Players aim to achieve the highest score possible, often with constraints like time limits.
    - **Example**: Often referred to as "score attack mode," this mode emphasizes scoring over other gameplay aspects.
12. **Casual Mode**
    - **Definition**: Removes high-pressure elements to allow players to enjoy the game without stress.

- **Example**: *Minecraft* features a "creative mode" where players cannot die, have unlimited resources, and can fly.

Example: Valorant Game Modes

Valorant features various permanent PvP game modes, including:

- **Unrated**: Standard play without ranked implications.
- **Competitive**: Ranked matches with skill-based matchmaking.
- **Spike Rush**: Quick rounds with a set number of rounds.
- **Deathmatch**: Free-for-all combat.

Additionally, there is a rotating PvP mode that includes options like Escalation, Replication, and Snowball Fight. Players can also practice in the Range, honing their skills outside of competitive play.

GAME PLAYING PERSPECTIVES

Game playing perspectives define the player's viewpoint within a game, significantly influencing the gaming experience. Here are the primary perspectives:

1. First-Person Perspective

- **Description**: The player views the game world through the eyes of the character they are controlling.
- **Characteristics**:
  - Common in action and shooter games.
  - Provides an immersive experience, as players feel directly involved in the game.
- **Example**: *Call of Duty*, where players see the environment from the character's viewpoint.

2. Third-Person Perspective

- **Description**: The camera is positioned behind and slightly above the character, allowing players to see their character on-screen.
- **Characteristics**:
  - Often referred to as the "over-the-shoulder" view.
  - Popular in many modern 3D games, as it provides a better sense of character movement and environment interaction.
- **Example**: *The Witcher 3*, where players can see Geralt and navigate the world around him.

3. Top-Down Perspective

- **Description**: The game is viewed from above, as if the camera is hovering over the action.
- **Characteristics**:
    - Commonly used in strategy games, both turn-based and real-time.
    - Allows players to see the entire battlefield or game area at once.
- **Example**: *Pokémon* games on the GameBoy, where players navigate through the environment from a bird's-eye view.

4. Isometric Perspective

- **Description**: A slightly tilted "three-quarter" view that gives the impression of 3D while still being 2D.
- **Characteristics**:
    - Often confused with top-down views but offers a more dynamic visual representation.
    - Allows for detailed environments and character interactions.
- **Example**: *Diablo* series and *The Sims*, where players can navigate and interact with a richly detailed environment.

5. Flat, Side-View

- **Description**: A traditional two-dimensional perspective that displays the action from the side.
- **Characteristics**:
    - Popularized by side-scrolling games in the 1980s and 1990s.
    - Less common today due to the rise of 3D graphics.
- **Example**: Classic *Mario* and *Sonic the Hedgehog* games, where players move left or right across the screen.

6. Text-Based Games

- **Description**: Games that rely primarily on text for gameplay, often with minimal or no graphics.
- **Characteristics**:
    - Focus on narrative and player choices through written descriptions.
    - Often involve puzzles and exploration based on text input.
- **Example**: Classic text adventures like *Zork* and *The Hitchhiker's Guide to the Galaxy*, where players read descriptions and input commands to progress.

Conclusion

Each perspective offers unique gameplay experiences and caters to different player preferences. Understanding these perspectives can help designers create engaging and immersive game worlds that resonate with their audience.

Scripting in Unity

Scripting in Unity allows developers to define behaviors and interactions within a game. Below are some fundamental concepts and examples of how to implement various scripting techniques in Unity using C#.

1. Scripts as Behaviour Components

Example: Color Change on Click

To create a script that changes the color of a GameObject when specific keys are pressed:

1. **Create a GameObject**: Open Unity, place a Cube or Sphere in the scene.
2. **Create a Script**: Go to Project > click '+', select C# Script, and rename it to Example1.
3. **Add Component**: In the Inspector window, click Add Component and select Example1.
4. **Edit the Script**: Open the Example1 file and add the following code:

```
using UnityEngine;

public class ExampleBehaviourScript : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.R))
        {
            GetComponent<Renderer>().material.color = Color.red;
        }
        if (Input.GetKeyDown(KeyCode.G))
        {
            GetComponent<Renderer>().material.color = Color.green;
        }
        if (Input.GetKeyDown(KeyCode.B))
        {
            GetComponent<Renderer>().material.color = Color.blue;
        }
    }
}
```

5. **Play Test**: Run the scene and press R, G, or B to change the color of the object.

2. Variables and Functions

Example: Using Variables and Functions

```
using UnityEngine;

public class VariablesAndFunctions : MonoBehaviour
{
    int myInt = 5;

    void Start()
    {
        myInt = MultiplyByTwo(myInt);
        Debug.Log(myInt);
    }

    int MultiplyByTwo(int number)
    {
        return number * 2;
    }
}
```

---

3. Control Flow

If Statements

```
using UnityEngine;

public class IfStatements : MonoBehaviour
{
    float coffeeTemperature = 85.0f;
    float hotLimitTemperature = 70.0f;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            TemperatureTest();
            coffeeTemperature -= Time.deltaTime * 5f;
        }
    }

    void TemperatureTest()
    {
        if (coffeeTemperature > hotLimitTemperature)
            Debug.Log("Coffee is too hot.");
    }
}
```

Loops

For Loop

```
using UnityEngine;

public class ForLoop : MonoBehaviour
{
    int numEnemies = 3;

    void Start()
    {
        for (int i = 0; i < numEnemies; i++)
        {
            Debug.Log("Creating enemy number: " + i);
        }
    }
}
```

While Loop

```
using UnityEngine;

public class WhileLoop : MonoBehaviour
{
    int cupsInTheSink = 4;

    void Start()
    {
        while (cupsInTheSink > 0)
        {
            Debug.Log("I've washed a cup!");
            cupsInTheSink--;
        }
    }
}
```

Do-While Loop

```
using UnityEngine;

public class DoWhileLoop : MonoBehaviour
{
    void Start()
    {
        bool shouldContinue = false;
        do
        {
            Debug.Log("Hello World");
        } while (shouldContinue);
```

```
    }
}
```

For Each Loop

```
using UnityEngine;

public class ForeachLoop : MonoBehaviour
{
    void Start()
    {
        string[] strings = { "First string", "Second string", "Third string"
};

        foreach (string item in strings)
        {
            Debug.Log(item);
        }
    }
}
```

4. Scope and Access Modifiers

```
using UnityEngine;

public class ScopeAndAccessModifiers : MonoBehaviour
{
    public int alpha = 5;
    private int beta = 0;

    void Start()
    {
        alpha = 29;
    }

    void Update()
    {
        Debug.Log("Alpha is set to: " + alpha);
    }
}
```

5. Awake and Start

```
using UnityEngine;

public class AwakeAndStart : MonoBehaviour
{
    void Awake()
    {
        Debug.Log("Awake called.");
```

```
    }

    void Start()
    {
        Debug.Log("Start called.");
    }
}
```

6. Update and FixedUpdate

```
using UnityEngine;

public class UpdateAndFixedUpdate : MonoBehaviour
{
    void FixedUpdate()
    {
        Debug.Log("FixedUpdate time: " + Time.deltaTime);
    }

    void Update()
    {
        Debug.Log("Update time: " + Time.deltaTime);
    }
}
```

7. Vector Maths

Example: Angle Calculation

```
using UnityEngine;

public class AngleExample : MonoBehaviour
{
    public Transform target;

    void Update()
    {
        Vector3 targetDir = target.position - transform.position;
        float angle = Vector3.Angle(targetDir, transform.forward);
        if (angle < 5.0f)
            Debug.Log("Close");
    }
}
```

8. Damage Calculation

Sample 1

```
public float health = 10f;
public float damageTick = 1f;
```

```
void OnTriggerEnter(Collider obj)
{
    if (obj.gameObject.tag == "damage")
    {
        health -= damageTick;
    }
}
```

Sample 2

```
void OnUseAbility()
{
    Character target = GetMouseTarget(); // Acquire the target
    float damageAmount = 1.5f * strength; // Damage formula
    Damage(this, damageAmount, DamageType.Magic, target);
}

void Damage(Character source, float dmgAmount, DamageType dmgType, Character
target)
{
    if (dmgType == DamageType.Physical)
        dmgAmount *= (1 - target.def);
    else
        dmgAmount *= (1 - target.mdef);

    target.health -= dmgAmount;
}
```

---

9. Camera Look At

```
using UnityEngine;

public class CameraLookAt : MonoBehaviour
{
    public Transform target;

    void Update()
    {
        transform.LookAt(target);
    }
}
```

10. Linear Interpolation

Example: Lerp Function

```
void Update()
{
    light.intensity = Mathf.Lerp(light.intensity, 8f, 0.5f * Time.deltaTime);
```

```
}
```

11. Destroying Objects

Basic Destroy

```
using UnityEngine;

public class DestroyBasic : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKey(KeyCode.Space))
        {
            Destroy(gameObject);
        }
    }
}
```

Destroy Other

```
using UnityEngine;

public class DestroyOther : MonoBehaviour
{
    public GameObject other;

    void Update()
    {
        if (Input.GetKey(KeyCode.Space))
        {
            Destroy(other);
        }
    }
}
```

AUDIO ENGINEERING

- **Audio engineering** involves creating and implementing audio elements for a game, such as:
    - Music
    - Sound effects
    - Voice acting
    - Ambient audio
- Audio engineers are responsible for integrating audio assets into the game engine. This includes:

    - Implementing sound effects, music, and dialogue into the gameplay
    - Ensuring proper synchronization with in-game events and player actions

Responsibilities of Audio Engineers

- Responsibilities can range from:
    - Sound design for in-game effects
    - Engineering dialogue audio to sound appropriate within the environment
- Music production and composition may also be part of the job, but it isn't always strictly involved.
- Audio is a core component of any video game and has a significant impact on the gaming experience. For example:
    - Critiques often mention how a certain attack isn't satisfying to use; sometimes, audio is what makes or breaks that experience.
- Additionally, since sound effects and soundtracks occur dynamically, audio engineers need to be aware of how their work impacts the game's overall performance.

Audio Engineering and Sound

- Audio engineering ensures all audio in the game is of the highest quality without affecting performance. An engineer also captures audio, such as:
    - Voice-over dialogue
    - Instrument recordings
    - Correctly replays it within the game environment
- **Sound design** is a specific discipline focusing on crafting unique sounds. This discipline is essential across:
    - Music
    - Movies
    - Video games
- A sound designer needs to have strong knowledge of working with hardware and software tools.

- **Composition** can also be considered music production, focusing primarily on creating the game's soundtrack. However, unlike typical music production, a composer faces unique challenges, such as:
    - Transitioning between tracks based on player activity
    - Working with generative AI systems to create music

Sound and Music in Level Design

- Sound and music are important components of video games and can significantly impact the player's experience.

Immersion

- Music can help players feel immersed in the game's world and reinforce the story and emotions.
- It contributes to creating an immersive atmosphere, enhancing game memory and recognition.
- By using instruments and sounds consistent with the game's world, music can reinforce feelings of realism and legitimacy.

LEVEL DESIGN

- **Level design** is the process of creating the stages, maps, and missions of a video game. It is a key part of the game development process, involving the design of the layout, challenges, and interactions within each level.
- A level is a space where a game takes place. Examples include:
    i. The Fortnite island
    ii. An obstacle course ("obby") in Roblox
    iii. A basketball court, race track, or playground
    iv. A Monopoly board, crossword puzzle, or coloring book
- All these game spaces set boundaries for players to move and interact. Different levels offer variation; for instance, while all basketball courts have similar shapes, an outdoor court and an indoor gym provide different experiences, cultures, and moods.
- Level designers focus on how different game spaces can influence player feelings and behaviors.

Functional Level Design vs. Environment Art

- Level designers concentrate on shaping player behavior. In large studios, they often:

    - Write documentation
    - Draft layouts
    - Build blockouts
    - Observe playtests
    - Balance maps and encounters
- In contrast, an **environment artist** focuses more on graphics, handling:
    - Model art passes
    - Materials
    - Set dressing
    - Lighting to refine the level's visual appearance
- While environment art is primarily decorative, good art supports experience design goals and enhances player engagement.

Room Design vs. World Design

- Level designers can spend days or even weeks designing a single room.
- For large-scale games like battle royales, open worlds, or MMOs with hundreds or thousands of rooms, focusing on a single room is impractical.
- A **world designer** considers:
    - Flow and wayfinding for neighborhoods instead of individual houses
    - Biomes instead of specific locations
    - Categories instead of instances
- This generic approach allows players and systems to breathe, but without players or systems to fill the void, the resulting world may feel empty or bland.
- To create a directed or scripted experience, obsess over every room like an architect. For player-heavy, system-heavy games with vast spaces, world design offers a lighter touch, akin to urban planning.

GAME PROJECT MANAGEMENT

- In the fast-paced and dynamic world of game development, effective project management is crucial for delivering successful games on time and within budget.

- From concept to launch, every stage of game development requires meticulous planning, coordination, and adaptability to overcome challenges and deliver exceptional gaming experiences.

1. **Complexity and Dynamic Nature**

   Game development projects are known for their intricate and dynamic nature. Designing captivating gameplay mechanics, creating visually stunning graphics, and immersive storytelling requires orchestrating numerous interconnected elements. Managing this complexity demands careful planning, clear communication, and a robust project management framework.

2. **Time Constraints and Deadlines**

   Time is critical in game development, with projects often operating under strict deadlines. Meeting release dates and maintaining project timelines are essential for success. Effective project management practices, such as Agile methodologies, enable teams to break down development tasks into manageable iterations, ensuring consistent and timely delivery of milestones.

3. **Resource Allocation and Team Collaboration**

   Game development projects involve multidisciplinary teams comprising artists, programmers, designers, and audio engineers. Coordinating these diverse skill sets and ensuring efficient resource allocation can be challenging. Effective project management facilitates clear communication, collaboration, and resource planning, ensuring that the right people are assigned to the right tasks at the right time.

4. **Risk Management and Mitigation**

   The game development landscape is not without its risks. From technical glitches and scope creep to changing market trends, various factors can impact project success. Effective project management involves identifying potential risks, developing contingency plans, and continuously monitoring and mitigating them to minimize their impact on progress and outcomes.

AGILE PROJECT MANAGEMENT

- At its core, **Agile project management** is rooted in principles and values that prioritize collaboration, adaptability, and continuous improvement.
- These principles promote customer satisfaction, early and frequent delivery of working software, and embracing change to enhance the final product. The **Agile Manifesto**, a foundational document in Agile project management, outlines these principles and guides Agile teams in the game development industry.

- Agile methodologies, such as **Scrum**, **Kanban**, and **Lean**, offer numerous benefits that align with the needs of game development projects. Their iterative approach and regular feedback loops empower teams to quickly adapt to changing requirements, market trends, and player feedback, ensuring that the final product aligns with players' evolving needs and expectations.

Benefits of Agile Project Management

- Agile project management methodologies have proven invaluable in game development, offering a range of benefits and addressing the unique challenges of the industry.
- The flexibility, enhanced collaboration, accelerated time-to-market, and improved quality associated with Agile approaches make them highly relevant in today's fast-paced and ever-evolving game development landscape.

GAME DESIGN DOCUMENTATION

- Game documentation is usually seen as the least fun part of the game development process. Many opt to skip it entirely, thinking, "No one reads GDDs anyway," or "They become outdated the minute you finish writing them."
- While they are not entirely wrong, rigid, multi-page GDDs have no place in modern game development. However, game design documentation has merely evolved rather than become obsolete.

What is a Game Design Document (GDD)?

A game design document (GDD) is a software design document that serves as a **blueprint** from which your game is to be built. It helps you **define the scope** of your game and sets the general **direction** for the project, keeping the entire **team** on the same page.

A GDD Usually Includes:

1. **Executive Summary**: Game concept, genre, target audience, project scope, etc.
2. **Gameplay**: Objectives, game progression, in-game GUI, etc.
3. **Mechanics**: Rules, combat, physics, etc.
4. **Game Elements**: Worldbuilding, story, characters, locations, level design, etc.
5. **Assets**: Music, sound effects, 2D/3D models, etc.

How to Write a GDD

Modern game design documentation follows several best practices:

1. **Keep it Lightweight**: The original game concept doesn’t always work out. Keep initial documentation to a **minimum** so you don’t have to rewrite a multi-page document. Aim for a **single page** and let it **evolve** from there.
2. **Write Collaboratively**: Involve your **team** in the process from the start. Make your GDD the **central hub** where team members can discover, discuss, and solve issues together.
3. **Evolve with the Project**: A GDD is only useful when it’s **up-to-date.** Choose a **documentation tool** that preserves **version history** and update it daily as your game evolves.
4. **Use Visual Aids**: Don't let your readers drown in text. Convey many ideas more clearly using **graphs, flow charts,** and **concept art.** Consider turning your entire game design doc into a **mind map.**

RAPID PROTOTYPING

- Rapid prototyping is a game development technique that involves creating **playable prototypes** of a game idea **quickly and cheaply.**
- The goal is to focus on the **core gameplay, mechanics, and features,** and to get **feedback** from users early in the development process.
- Apart from **verifying ideas** and **testing gameplay mechanics,** another major goal of rapid game prototyping is to gather user opinions at an early stage of production. This feedback helps developers improve the finished product to create immersive games that appeal to players.

The Process of Rapid Game Prototyping

Rapid game prototyping is a powerful tool in game development, used to iterate on game ideas and design aspects. The procedure can be divided into several **important phases**:

1. **Ideation**: Brainstorm and come up with **ideas** for the game. Discover **key factors** that affect gameplay experience, including mechanics, themes, and features.
2. **Prototype Creation**: Use various **tools** and programming frameworks to produce simple prototypes that illustrate essential gameplay components. Prioritize **utility** over visual refinement for speedy testing and iteration.
3. **Testing and Feedback**: Test the developed prototypes before sharing them with stakeholders and potential participants. Acquire insightful input on user experience, mechanics, and enjoyment.
4. **Iteration and Refinement**: Make adjustments based on user opinions, fine-tuning and optimizing the game in this phase.

5. **Validation and Decision-making**: Evaluate the viability and compatibility of the prototypes with the planned vision. Use feedback to determine whether prototypes or concepts should be developed further.

Benefits of Rapid Game Prototyping

- Faster Concept Validation and Iteration
- Enhanced Collaboration and Communication
- Cost and Time Efficiency
- Risk Mitigation
- Iterative Improvement
- Player-Centric Design
- Innovation and Creativity

Tools and Technologies of Rapid Game Prototyping

- In rapid game prototyping, developers have access to a wide range of tools and technologies that significantly enhance the process. Notable tools include **Unity** and **Unreal Engine**.
- These tools provide **solid frameworks** and **extensive libraries** for rapid prototyping. **Visual scripting tools** also offer users an easy-to-use interface for prototyping without requiring a detailed understanding of programming.
- Dedicated prototype software, such as **Framer** or **InVision**, simplifies the process by providing **specific capabilities** and **templates**.
- These tools, with their **diverse capabilities** and **user-friendly interfaces**, play a critical role in **boosting productivity** and **creativity** throughout the rapid game prototyping phase.

GAME TESTING

- In the fast-paced and constantly evolving world of game development, testing has become a critical pillar of success.
- Recent challenges faced by major industry players due to buggy releases have underscored the vital importance of thorough and rigorous testing.

Types of Game Testing

1. **Combinatorial Testing**:
    - A software testing technique that focuses on testing **all possible combinations of input values** for a given **feature** or **function**. This approach is particularly useful for game

testing, as it can help identify bugs or issues that may only occur under specific combinations of circumstances.

- Refer: Testbytes Blog

2. **Clean Room Testing:**
    - A software development methodology that **emphasizes defect prevention** rather than defect detection. In game testing, it involves a structured process of **creating test cases** based on **formal specifications**, ensuring that the game is thoroughly tested before it reaches the player.
3. **Functionality Testing:**
    - A crucial process that ensures the **game functions** as intended and meets the player's expectations. It involves testing the game's **core features, mechanics, and gameplay** to identify and fix any bugs or issues that could hinder the player's experience.
4. **Compatibility Testing:**
    - Plays a crucial role in ensuring that the **game runs smoothly** across a wide range of **hardware configurations, software environments, and input devices**. It aims to identify and resolve any compatibility issues that could affect the player's experience.
5. **Tree Testing:**
    - A usability testing technique used to evaluate the **information architecture** of a game's **menu system** or **navigation structure.** It helps determine how easily players can find desired information or functionality within the game's user interface.
6. **Regression Testing:**
    - An essential part of game development that ensures new code changes or **updates don't introduce new bugs**. It involves **selectively retesting a system or component** to verify that modifications have not caused unintended effects on previously functioning software.
7. **Ad hoc Testing:**
    - An **informal** software testing method that involves testing the game **without a predefined plan or test cases**. It relies on the tester's experience, intuition, and creativity to identify defects. Ad hoc testing is often used in **later stages** when the game is more **stable.**
8. **Load Testing:**
    - A crucial aspect of game development that ensures the game can handle the anticipated number of concurrent users **without**

performance degradation. It involves **simulating a large number of users** interacting with the game simultaneously to assess scalability and identify potential bottlenecks.

9. **Play Testing:**

- Involves **actual players** interacting with the game in a real-world setting to provide valuable feedback and identify potential issues. It **complements other testing methods** by providing insights into the overall user experience and gameplay.