SWENG 837 SOFTWARE SYSTEM DESIGN

JOHN MCCOUBRIE

BUSINESS REQUIREMENTS

Problem definition:

Traditional cooperative multiplayer games often fail to offer a highly interactive and physics-based environment that forces true collaboration between players. In “*Buddies in a Ball”* the core challenges lies in designing a system that synchronizes physics-based interactions for multiple players while maintaining a fun, responsive, and collaborative gameplay experience in a peer-to-peer (P2P) network. This includes addressing the technical challenges of multiplayer synchronization, implementing responsive controls for multiple users in a shared physics-based system, and creating an engaging experience that scales to different skill levels.

Functionalities:

- Physics-Based Multiplayer Collaboration
  - Players control avatars inside a shared rolling sphere, where movement relies on their combined input.
  - Implement real-time physics simulation that accounts for player interactions and the environment.
- Networked Multiplayer Gameplay
  - Peer-to-peer (P2P) networking to allow 2–4 players to connect, interact, and collaborate.
  - Synchronization of game state and physics calculations across all players to minimize latency and desynchronization.
- Dynamic Levels with Increasing Challenges
  - Predefined levels with varying and increasing difficulty catered to gameplay pacing
  - Interactive obstacles that require teamwork to navigate effectively
- Player Progression and Save System
  - Players should be able to save their progress and rejoin matches, continuing from their last checkpoint.
- Engaging User Interface
  - Intuitive menus for creating and joining games
  - HUD to display ball’s velocity, player direction, and level timer.

Target users:

- Cooperative Gaming Enthusiasts
  - Value team-based challenges and games that emphasize collaboration
  - Want gameplay that rewards communication and coordination
- Casual Gamers
  - Need fun and lighthearted gameplay that can be enjoyed with friends in short sessions

    - Seek intuitive controls and clear objectives to reduce friction
- Streamers and Content Creators
    - Need engaging gameplay that offers humorous and unexpected moments to captivate audiences
    - Desire features like spectator mode or replay system to enhance content creation

Business Goals:

- Attract a Broad Audience:
    - Design gameplay mechanics that appeal to a diverse group of players, including families, friends, and gaming communities.
- Encourage Social Interaction:
    - Foster community building through features like matchmaking, cross-platform play, and leaderboards.
- Monetization Opportunities:
    - Provide opportunities for revenue through DLCs, cosmetic upgrades for characters, and additional levels or challenges.
- Scalable Technology:
    - Ensure the P2P system can scale efficiently for global players, minimizing costs while maintaining a seamless experience.
- Retention and Replayability:
    - Introduce progression systems, unlockable content, and dynamic level design to encourage long-term engagement.

NON-FUNCTIONAL REQUIREMENTS

Performance Requirements

- Scalability:
    - The system must support 2–4 players in a peer-to-peer (P2P) environment without noticeable degradation in performance.
    - It should efficiently handle increased physics computations for larger levels or complex obstacles while maintaining synchronization.
- Response Time:
    - Game input from any player should be reflected in the physics simulation and rendered on all connected devices within 100 milliseconds to ensure responsive gameplay.

  - The matchmaking system must create or join sessions within 5 seconds.
- Throughput:
  - The system must handle simultaneous physics updates and network messages for up to 100 game objects (e.g., sphere, players, obstacles) in real time without dropping packets or frames.
  - Data packet loss should not exceed 1%, and recovery mechanisms should minimize its impact on gameplay.

Security Requirements

- Authentication:
  - Players must authenticate using a secure method, such as OAuth or a unique game ID system.
  - The system should support guest accounts for casual players and persistent accounts for returning users.
- Authorization:
  - Only authenticated players can join a session, and the host should have the authority to accept or remove players.
  - In-game purchases (if implemented) must validate users' credentials to prevent unauthorized transactions.
- Data Encryption:
  - All communication between players must use encryption protocols (e.g., AES-256 for game data and TLS for matchmaking) to secure sensitive information.
  - Save data, including progress and player information, should be encrypted locally and when transmitted over the network.

Maintainability Requirements

- Code Modularity:
  - The system must use a modular architecture where key components (e.g., physics engine, networking module, UI) are independently designed and replaceable.
  - Modular Blueprints or C++ classes in Unreal Engine should be implemented to isolate functionality.

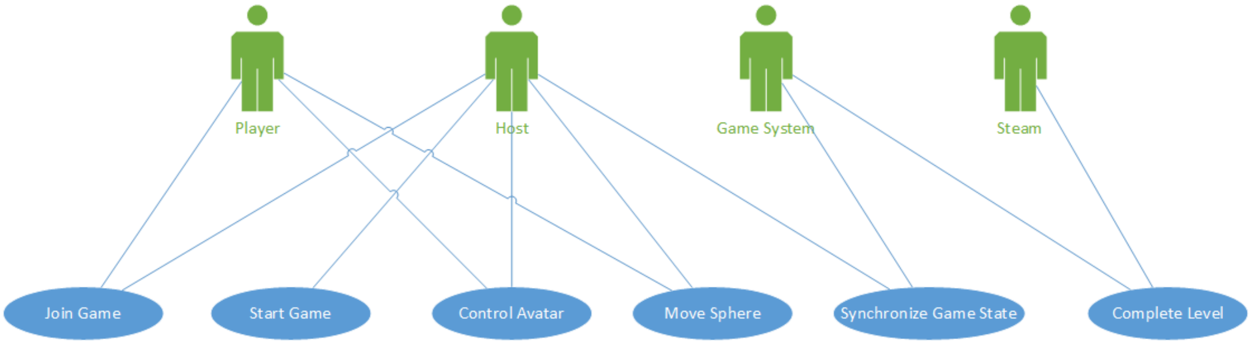
- Documentation:
    - Comprehensive documentation should be maintained for:
        - System architecture.
        - Key modules and their interfaces.
        - Installation, deployment, and troubleshooting procedures.
- Testing Strategies:
    - Automated unit tests should cover 80% of the core logic, including physics calculations, networking, and save/load functionality.
    - Functional tests should validate collaborative gameplay, including edge cases (e.g., disconnects or extreme latency).
    - Load testing should simulate a full player count and stress-test physics-intensive levels to ensure stability.

Other Non-Functional Requirements

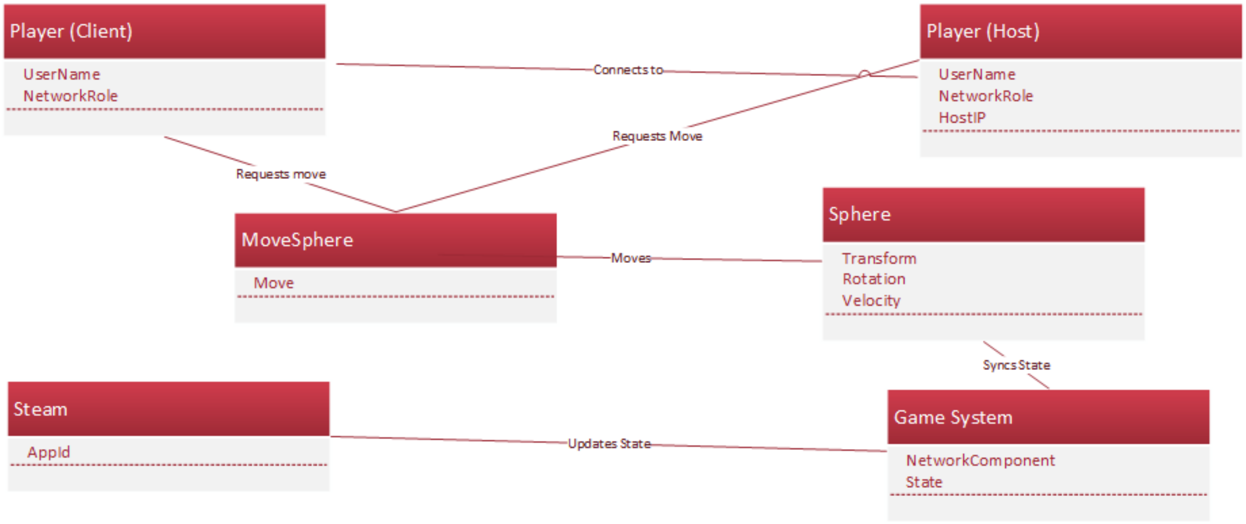
- User Experience (UX):
    - The game must run smoothly on devices with mid-tier hardware, achieving a consistent 60 FPS.
    - The interface should be simple, intuitive, and free of unnecessary complexity to cater to casual and experienced gamers.
- Cross-Platform Compatibility:
    - The system should work seamlessly across Windows, macOS, and major gaming consoles, with minimal additional development for platform-specific features.
- Reliability:
    - The system must achieve 99.9% uptime during sessions, with mechanisms for players to reconnect without losing progress in case of a crash or disconnect.
- Localization:
    - The game must support at least 5 languages to appeal to a global audience, with dynamic language switching.
- Accessibility:
    - Implement features such as adjustable text size, colorblind-friendly UI options, and configurable controls to ensure inclusivity.

- Legal and Compliance:
  - Ensure compliance with data privacy laws such as GDPR for player information, especially if the game includes accounts or user data storage.

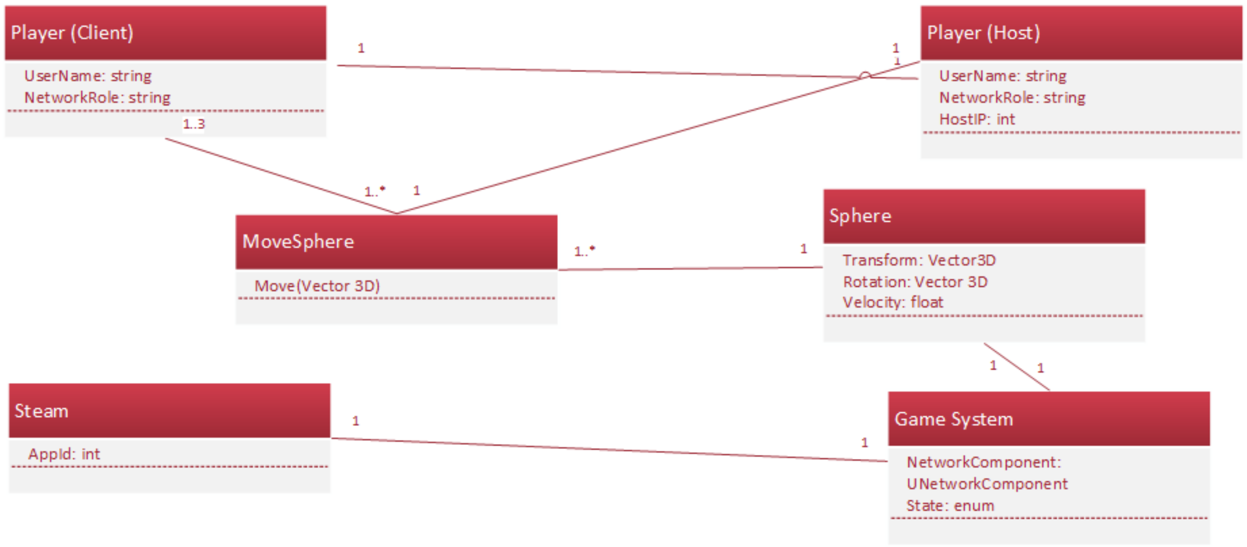
USE CASE DIAGRAM



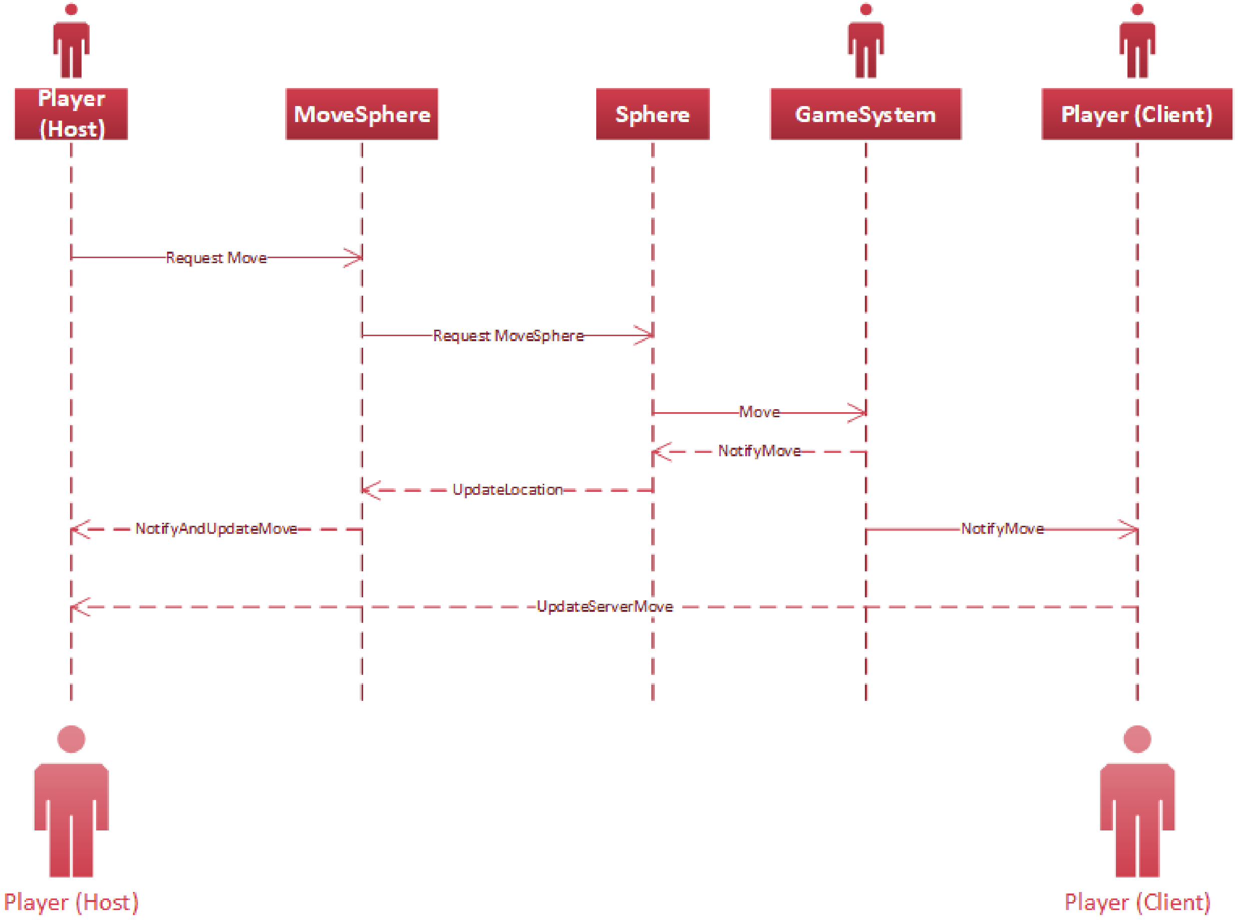
DOMAIN MODEL



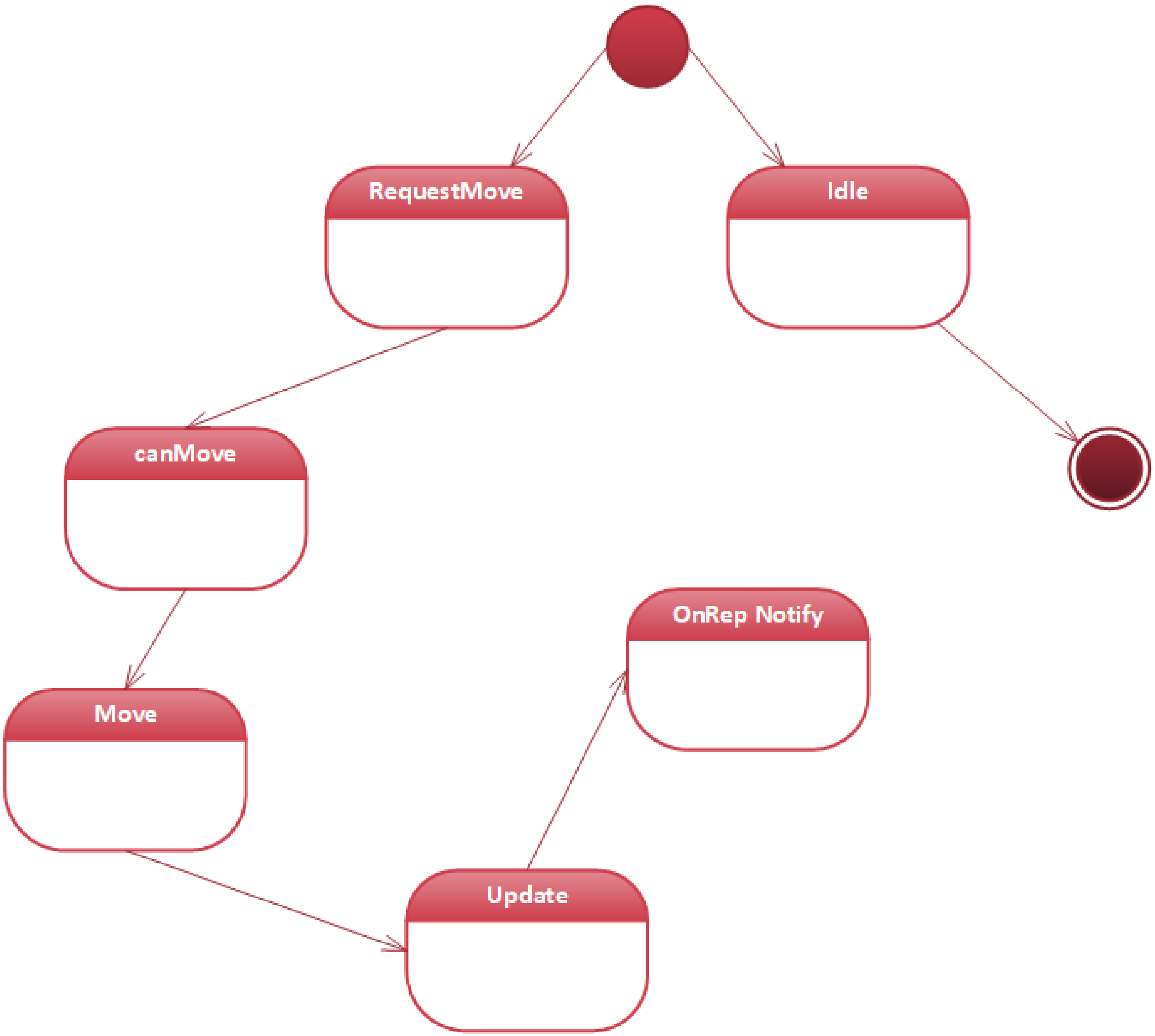
CLASS DIAGRAM



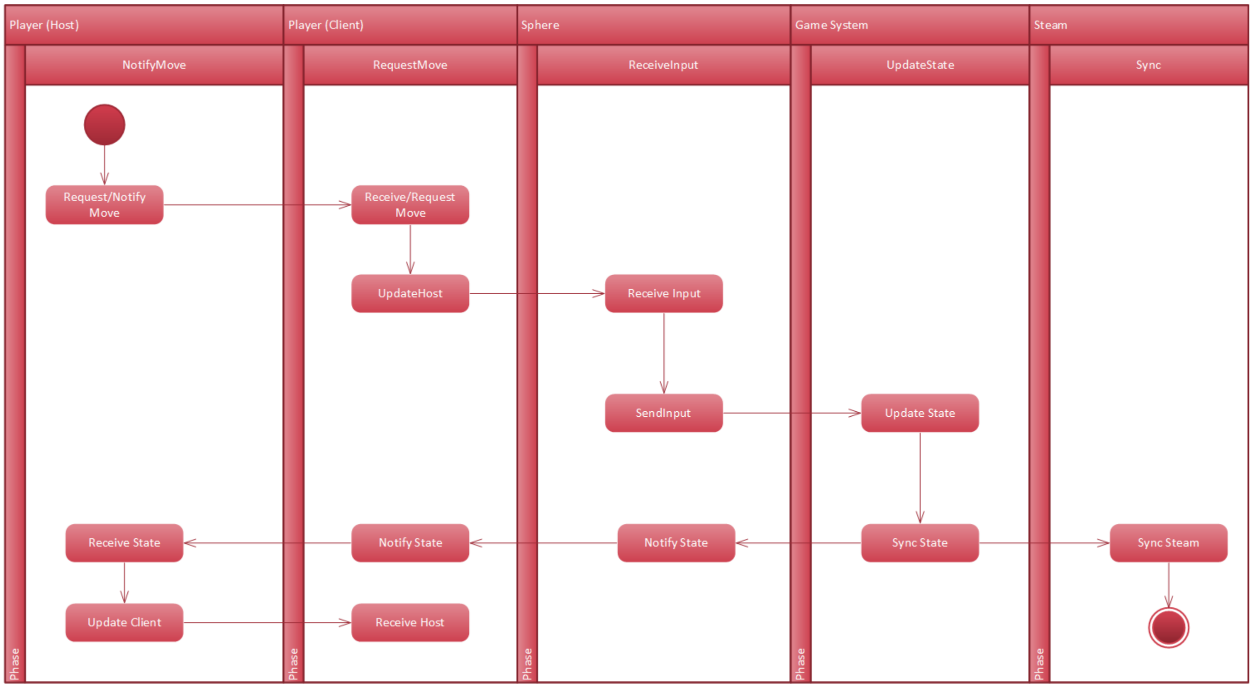
SEQUENCE DIAGRAM



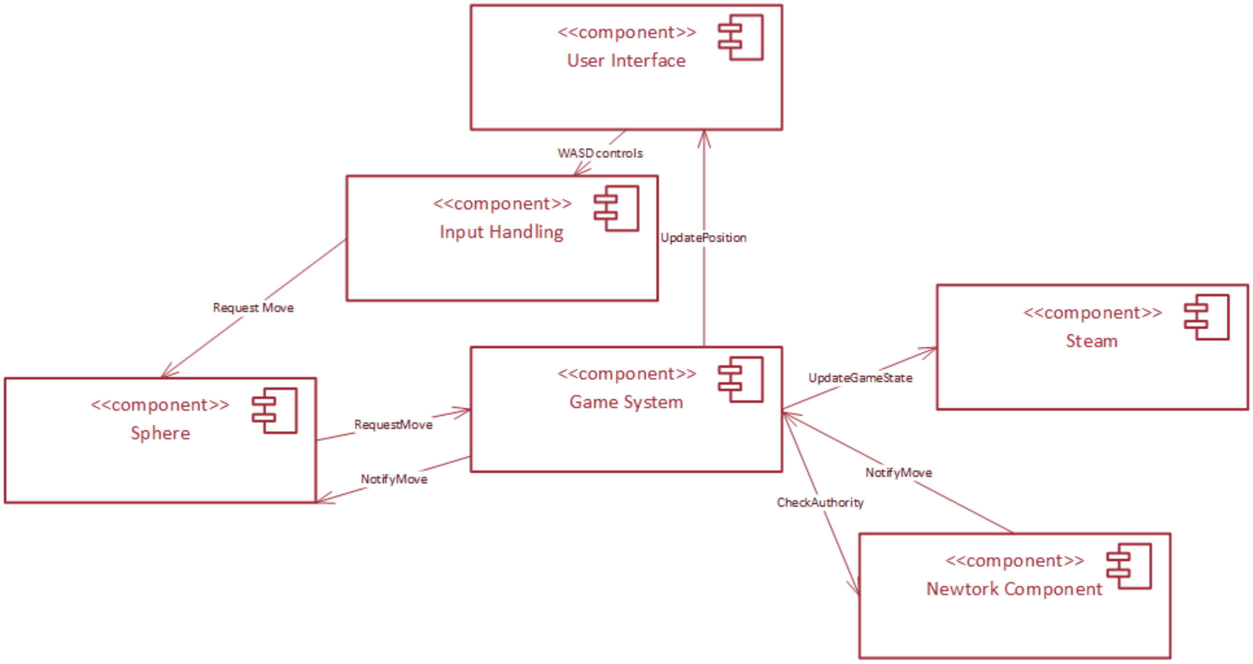
STATE DIAGRAM



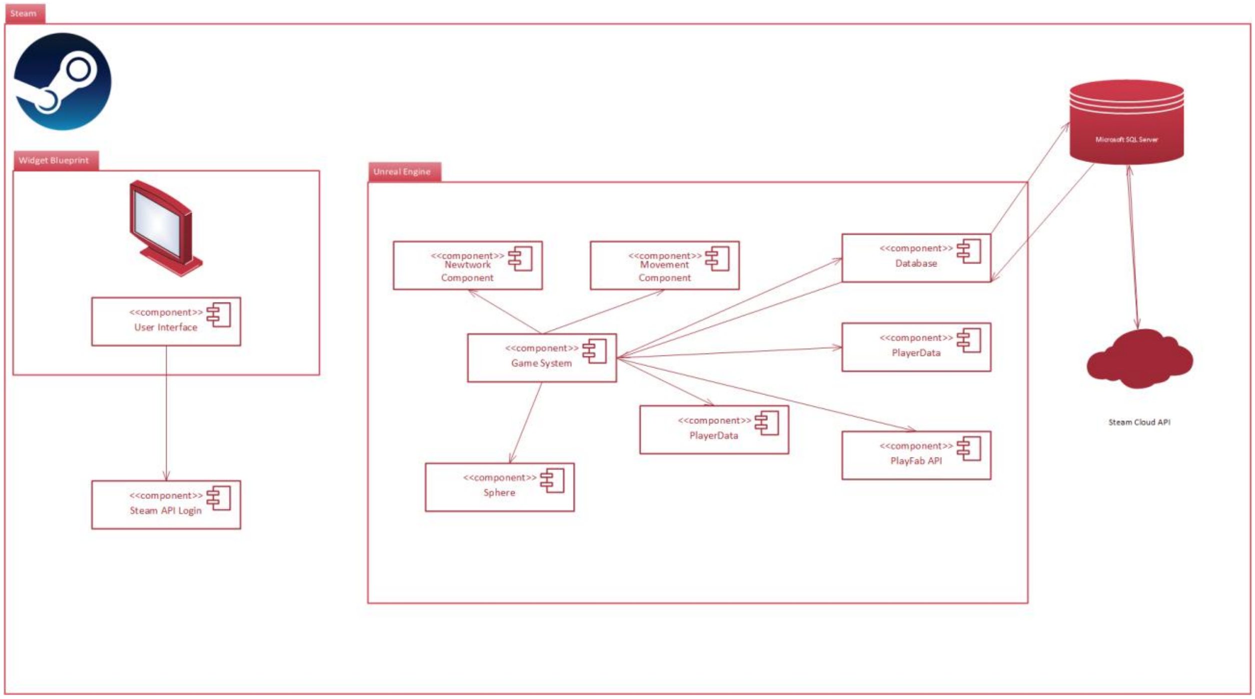
ACTIVITY DIAGRAM



COMPONENT DIAGRAM



DEPLOYMENT DIAGRAM



SKELETON CLASSES AND TABLES DEFINITION

1. Player (Host)
   a. Attributes:
      i. Username: Steam username
      ii. NetworkRole: UNetwork Component Role to distinguish authority.
      iii. HostId: Host IP for the client to connect to
   b. Methods:
      i. Move(): Moves the player avatar
      ii. Look(): Control camera movement
      iii. ServerMoveBall(): Requests ball to move on the server
2. Player (Client)
   a. Attributes:
      i. Username: Steam username
      ii. NetworkRole: UNetwork Component Role to distinguish authority.
   b. Methods:
      i. Move(): Moves the player avatar
      ii. Look(): Control camera movement
      iii. ClientMoveBall(): Requests ball to move on the server from the client
      iv. ServerMoveBall(): Requests ball to move on the server

3. Sphere
    a. Attributes:
        i. Transform: X, Y, and Z coordinates for sphere position
        ii. Rotation: X, Y, and Z coordinates for sphere rotation
        iii. Velocity: Speed of the sphere
    b. Methods:
        i. Move(): Move the ball
        ii. NotifyMove(): Notify network component of ball state
        iii. RequestUpdate(): Request update if latent
        iv. Update(): Update location to network component
4. NewtorkComponent
    a. Attributes:
        i. NetworkId: Network Id for host to connect to
        ii. GameState: Game state enum to sync states between host and client
    b. Methods:
        i. CheckState(): Check state of host and client
        ii. SyncState(): Sync state of the host and client
        iii. UpdateState(): Notify host and client of updated state

DESIGN PATTERNS

1. GRASP Patterns
    a. **Creator**:
        i. The game system (Unreal Engine) is responsible for creating game objects like the sphere, player avatars, and obstacles.
        ii. **Justification**: Unreal Engine naturally handles object creation through its lifecycle management, ensuring game objects are instantiated efficiently and only when needed.
    b. **Controller**:
        i. The **Player (Host)** acts as the controller for managing the game session, including initiating gameplay, saving progress, and handling session events.
        ii. **Justification**: This centralizes session-related responsibilities, making it easier to maintain and extend multiplayer functionality.
    c. **Information Expert**:
        i. The physics engine component handles calculations related to the movement of the sphere and collisions with obstacles.
        ii. **Justification**: Assigning physics-related responsibilities to the physics engine ensures correctness and reduces coupling with gameplay logic.
2. SOLID Principles
    a. Single Responsibility Principle (SRP):
        i. Each system component has a single purpose:
            1. **Physics Engine**: Handles all sphere movement and collision responses.
            2. **Networking Module**: Synchronizes state across players.
            3. **UI Manager**: Manages menu interactions and in-game HUD.

        ii. **Justification**: This makes components easier to test and reduces the risk of unintended changes affecting unrelated parts of the system.

    b. **Open/Closed Principle (OCP)**:

        i. Game components like levels, obstacles, or avatar customization are designed to be extended without modifying existing code.

        ii. **Justification**: Adding new levels or cosmetic options should not require altering the core gameplay mechanics, enabling easy future updates.

    c. **Liskov Substitution Principle (LSP)**:

        i. Base classes like GameObject allow substitution with derived classes for specific entities like Obstacle or PlayerAvatar.

        ii. **Justification**: This ensures polymorphism works correctly, reducing runtime errors when new entity types are introduced.

    d. **Interface Segregation Principle (ISP)**:

        i. Separate interfaces for distinct responsibilities, e.g., INetworkable for network-related methods and IMovable for movement-related methods.

        ii. **Justification**: Prevents classes from implementing unnecessary methods, keeping the design modular.

    e. **Dependency Inversion Principle (DIP)**:

        i. High-level modules like the game logic depend on abstractions (e.g., interfaces) rather than specific implementations of the physics or networking modules.

        ii. **Justification**: This allows swapping out implementations (e.g., different networking protocols) with minimal changes.

3. GOF (Gang of Four) Patterns

    a. **Observer Pattern**:

        i. Used for synchronizing game state across the host and clients.

        ii. **Justification**: Ensures that changes in the game world (e.g., sphere position, obstacle state) are broadcasted to all players in real time.

    b. **Strategy Pattern**:

        i. Implements different movement behaviors for obstacles or dynamic elements in the game.

        ii. **Justification**: Allows easy customization of movement patterns (e.g., rotating platforms, sliding walls) without altering the core movement logic.

    c. **Command Pattern**:

        i. Used for handling player inputs (e.g., movement, jump, special abilities).

        ii. **Justification**: Decouples input handling from game mechanics, making it easier to map new actions or support input from multiple devices.

    d. **Singleton Pattern**:

        i. Ensures there is a single instance of key systems like the networking manager or save/load manager.

        ii. **Justification**: Prevents conflicts and ensures global access to shared resources.

4. Microservices and Best Practices

    a. **Modular Components**:

        i. Networking, physics, and UI systems act as modular components that communicate through well-defined APIs or interfaces.

        ii. **Justification**: Modularization allows independent testing, parallel development, and easier debugging.

b. **Data Persistence as a Service**:
   i. A separate save/load system handles player progress independently from the gameplay logic.
   ii. **Justification**: Decouples persistence from gameplay, making it easier to extend or modify save formats without affecting core mechanics.

c. **Stateless Services**:
   i. Networking follows stateless communication wherever possible (e.g., sending periodic updates of the game state rather than relying on continuous state).
   ii. **Justification**: Reduces complexity and potential errors caused by inconsistent state synchronization.

d. **Event-Driven Communication**:
   i. Use an event-based approach to trigger actions like saving progress, player respawns, or session joins.
   ii. **Justification**: Improves responsiveness and simplifies the flow of communication between components.

SCALABILITY AND SPEED

1. Peer-to-peer (P2P) network architecture
    a. By using a P2P network system the system can scale to as many players as possible. Each host creates its own sessions on the Steam network at no additional cost. The SQL Server backend would have to scale if used, but built-in Steam high scores can be used for player data.
2. Steam
    a. Various built-in Steam assets and functions that are cost free with no cap on player amount.
3. Unreal Engine Specific
    a. Levels of Detail: 1000m shows less detail than 100m, built in unreal-engine optimization
    b. Network Optimization tools: The network profiling tools allow developers to analyze network bottle necks to update code and create new strategies to provide efficient handling of network data.
4. Custom Blender Assets
    a. Blender creates low-poly shaders and optimizations with each asset.

COST OPTIMIZATION

1. Peer-to-peer (P2P) network architecture
    a. By using a P2P network system there will be no dedicated server with zero cost. Unreal engine has a 0 cost P2P system so the host will always host each client on the system. In doing so, the players host each game session.
2. Steam
    a. There is only a $100 steam app id cost to publish a game on Steam. This is the only upfront cost
3. Asset Store
    a. The Fab marketplace has various free assets if blender is not used. These are pre-built assets that are not as optimized for speed, but are cost free and save time.
4. Custom Blender Assets
    a. While requiring more time, blender is an effective 3D modelling tool that is free to all developers and artists. As a solo developer this costs nothing.