|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **US** | **Name** | **State** |
| * 1. **GUI bis Lobby (Menüsteuerung)** |  |  |  |
| * + 1. (Nur Deklaration) GUI-Klasse, SPConfig, MPConfig, State-Enum, Color-Enum + javaFX Rahmen (primaryStage anlegen & anzeigen,…) | *(Appropriate Design)* |  |  |
| * + 1. Scenes (Anzeigbar über primaryStage.setScene(xyScene) o.ä.), bestehen aus: + jeweils eigenes .fxml Dokument + ein geteilter Controller (s. http://stackoverflow.com/a/17914784)  + Jew. eigene Methode in der GUI, idealerweise -> zu implementierende Scenes: |  |  |  |
| * + - 1. SplashScreen Scene | *SplashScreen* |  |  |
| * + - 1. MainMenu Scene | *MainMenu* |  |  |
| * + - 1. NameEntry Scene | *NameEntry* |  |  |
| * + - 1. HostAndJoin Scene          1. (Simulierte) Passwort-Check-Scene | *HostAndJoin MPJoin* |  |  |
| * + - 1. (Funktionslose Lobby) + Config Scene (Parametrisiert! SP oder MP Config anbieten.) | *MP/SPConfig* |  |  |
| * + - 1. (Funktionsloser Gamescreen) mit Sound-Toggle, Möglichkeit zu verlassen. | *GameExit* |  |  |
| * 1. **BoardState-Dummy in netzwerkloser Client-Klasse** |  |  |  |
| * + 1. Geometry-Struktur implementieren (gesamtes Model) | *BoardStateSync* |  |  |
| * + 1. Client-Klasse und BoardState-Klasse implementieren |  |  |  |
| * + - 1. Keine Netzwerkanbindung o.ä., erstmal nur Test-Boardstates generieren und regelmäßig updateGUI() aufrufen -> Threads! Hier am besten nachfragen statt selber recherchieren. | *BoardStateSync* |  |  |
| * + - 1. Die Client-Klasse enthält die main-Methode oder wird als erste gestartet, ist zu entscheiden. Das heißt, GUI-Statemachine anstarten nicht vergessen! | *Alle GUI-State-UStorys* |  |  |
| * + 1. In der Client-Klasse schonmal validateInput() als debug-methode implementieren + methode die regelmäßig (feste wiederholrate; auch in einem thread) die aktuell gedrückte richtung erstmal auf der Konsole ausgibt -> man muss also die keyUps und keyDowns auswerten; beachten dass man zB auch rechts UND links drücken kann. Parameter der Methode wäre dann ein KeyEventHandler o.ä. | *PlayerBar- Movement* |  |  |
| * 1. **GUI Logik:** |  |  |  |
| * + 1. State-Machine in der GUI-Klasse implementieren; Verlassen einer Menüebene bedeutet immer State-Wechsel. Ruft dann jeweils Methoden aus 1) auf um die Scenes zu wechseln.  -> Ist fertig, wenn man alle Menüwege Splash <-> Gamescreen ablaufen kann und im Gamescreen klar ist, in welchem Modus man ist (MPHost/MPJoin/SP + welche Konfiguration). Im Gamescreen dann erstmal als Debug-Text ausgeben. -> (Hier muss man als MPJoin erstmal nicht in die Lobby kommen, da Netzwerk wschl. noch nicht da ist.) | *LobbyExit*  *GameExit*  *Alle GUI-State-UStorys* |  |  |
| * 1. **Sound** |  |  |  |
| * + 1. Sammeln welche Sounds benötigt warden, passende Sounds suchen | *Sound & -Toggle* |  |  |
| * + 1. Sound-Files in eine geeignete Datenstruktur laden über init-Methode | *Sound & -Toggle* |  |  |
| * + 1. Verschiedene sinnvolle Methoden anbieten (Play, Pause, etc. -> schließlich müssen sich mehrere Sounds überlappen können!) | *Sound & -Toggle* |  |  |
| Ganz wichtig: Man arbeitet mit Files -> IOExceptions etc. beachten!! |  |  |  |
| * 1. **Gamescreen** |  |  |  |
| * + 1. Anbinden an updateGUI() -> Darstellen beliebiger BoardStates; erstmal rein funktional, schön kommt später. Bzw dauerhaft nebenbei als langlaufender Task | *BoardStateSync* |  |  |
| * + 1. Anbinden der KeyHandler: Keys gehen ungefiltert an Client, Sound an/aus (+ natürlich Möglichkeiten Sound zu testen), Möglichkeit ins Hauptmenü zurückzukehren (wobei natürlich der Client benachrichtigt werden muss) | *PlayerBar- Movement*  *Sound & -Toggle GameExit* |  |  |
| * + 1. Gamescreen passt sich automatisch der Fenstergröße an, indem alle Elemente gestreckt werden. | *GameScreenRes- olutionScaling* |  |  |

**Milestone 1:**

Menüsteuerung fertig, Gamescreen und Client tauschen automatisch inputs und boardstates aus; Gamescreen stellt BoardStates dar.

|  |  |  |  |
| --- | --- | --- | --- |
| * 1. **Server** |  |  |  |
| * + 1. Klären: Eigenes Projekt oder Teil des selben? Ziel: Nur einer der Clients startet den Server, alle können mit ihm kommunizieren. | *BoardStateSync* |  |  |
| * + 1. Netzwerkkommunikation Server<->Client implementieren | *BoardStateSync* |  |  |
| * + 1. (-> g.iii) | *BoardStateSync* |  |  |
| Ziel: Möglichkeit inputs (genauer InputHandlers) vom client zu empfangen sowie BoardState-Objekte zu senden. |  |  |  |
| * 1. **Game Manager** |  |  |  |
| * + 1. Initialisierung | *BallInitialize* |  |  |
| * + 1. Kollisionserkennung + Einordnung nach Typen (s. u.) | *EPIC: Collision* |  |  |
| * + 1. (A0) Kollisions-Handling, jeweils eigene Methode: |  |  |  |
| * + - 1. (A1) Ball-Bar 1+2 (inkl. Bar-Position & Bar-Richtung) | *BallBar1 + 2* |  |  |
| * + - 1. (A2) Ball-PowerUp | *BallPowerUp* |  |  |
| * + - 1. (A3) Ball-PlayerGoalEdge | *BallPlayGoalEdg* |  |  |
| * + - 1. (A4) Ball-CornerEdge | *BallCornerEdge* |  |  |
| * + - 1. (A5) CornerEdge-Bar | *CornerEdgeBar* |  |  |
| * + 1. InputState wird bei jeder eingehenden Nachricht aktualisiert (ist das über EventHandler zu lösen oder braucht man einen Thread, der alle x ms überprüft? -> Im Server zu klären) | *BoardStateSync PlayerBarMove- ment* |  |  |
| * + 1. KI implementieren | *AI* |  |  |
| * + 1. Implementierung des Haupt-Loops: |  |  |  |
| * + - 1. (B1) InputStates auswerten -> Bar-Positionen updaten | (?) |  |  |
| * + - * 1. Auch KI! Da noch keine Lobby existiert, erstmal davon ausgehen, dass 1 Spieler gegen 3 KI spielt. | *AI* |  |  |
| * + - 1. (B2) Ball-Direction, -Speed auswerten -> Ball-Positionen updaten | *BallBar (1 or 2?)* |  |  |
| * + - 1. (B3 Kollisionsüberprüfung + evtl. Handling | *EPIC: Collision* |  |  |
| * + - 1. Zusammenpacken zu einem BoardState, senden über Server | *BoardStateSync* |  |  |
| * 1. **Client + GUI** |  |  |  |
| * + 1. Möglichkeit der Servererstellung implementieren -> Geschieht beim Betreten der Lobby (erstmal nur als SP) | *BoardStateSync* |  |  |
| * + 1. Anbinden von validateInput() an den Server statt an die Debug-Konsolenausgaben | *BoardStateSync* |  |  |

**Milestone 2:**

Kommunikation mit dem Server ist gewährleistet. Das Spiel (auch wenn momentan nur im Singleplayer mit KI) läuft komplett über den Server.

Hier kann man dann testen, ob der input lag bei einem rein servergesteuerten Spiel evtl zu viel ist; gegensteuern kann man mit höheren Update-Raten oder notfalls clientseitiger Kompensation.

|  |  |  |  |
| --- | --- | --- | --- |
| * 1. **Client** |  |  |  |
| * + 1. Lobby mit multiplen Spielern | *LobbyReadyHost LobbyReadyJoin* |  |  |
| * + 1. Zuordnung von Farben, Namen, Positionen synchron | *PlayerColor* |  |  |
| * + 1. Spieler verlassen / verlieren das Spiel |  |  |  |
| * 1. **GameManager** |  |  |  |
| * + 1. PowerUps | *PowerUpEffekte* |  |  |
| * + 1. Win&Lose | *PlayerLoss WinningScreen NextRound- Concept* |  |  |
|  |  |  |  |
|  |  |  |  |

UserStories

1. SplashScreen
2. PlayerBarMovement
3. MainMenu
4. LobbyReady- Host
5. PlayerColor
6. LobbyReady- Join
7. MPConfig
8. SPCOnfig
9. Host&Join
10. Sound&SoundToggle
11. WinningScreen
12. LobbyExit
13. PowerUpEffekte
14. BallInitialize
15. NextRoundConcept
16. BoardStateSync
17. PlayerLoss
18. MPJoin
19. NameEntry
20. GameExit
21. AppropriateDesign
22. GameScreenResolutionScaling
23. EPIC: Collision
24. Ball-Bar collision
25. BallPowerUp Collision
26. BallPlayerGoalEdge Collision
27. Ball-Bar Collision 2
28. Ball ConrnerEdge Collision
29. CornerEdgeBar Collision
30. AI