Abstract

This project aimed to develop a movement-based 3D first-person shooter with multiplayer functionality and investigate how game and level design can promote competitive gameplay through well-developed movement mechanics. The game was developed in Unity 2022 LTS with Photon PUN 2 for networking functionality. Implemented systems include a responsive movement system with jumping, crouching, sprinting, and sliding, a hitscan-based weapon system with procedural animations, and a hierarchical animation system using Blend Trees and IK constraints. The project successfully created a functional multiplayer experience where players can express skill through both movement and precision. Technical challenges were primarily identified around wallrunning animations in multiplayer mode, where synchronization between clients proved more complex than expected. The game achieves over 400 FPS and testing with experienced FPS players confirmed that the movement system feels cohesive and competitive. The project demonstranters how the combination of fluid movement, responsive controls, and tactical level design can create engaging competitive gameplay.

Innehållsförteckning

Sammanfattning

Detta projekt syftade till att utveckla ett rörelsebaserat 3D skjutarspel med flerspelarläge och undersöka hur spel- och nivådesign kan främja kompetitivt spelande genom välutvecklad rörelsemekanik. Spelet utvecklades i Unity 2022 LTS med Photon PUN 2 för nätverksfunktionalitet. Implementerade system inkluderar ett responsivt rörelsesystem med hopp, duckning, sprint och glidning, ett hitscan-baserat vapensystem med procedurella animationer, samt ett hierarkiskt animationssystem med Blend Trees och IK constraints. Projektet lyckades skapa en fungerande multiplayer-upplevelse där spelaren kan uttrycka skicklighet genom både rörelse och precision. Tekniska utmaningar identifierades främst kring wallrunning-animationer i flerspelarläget, där synkronisering mellan klienter visade sig mer komplex än förväntat. Testning med erfarna FPS-spelare bekräftade att rörelsesystemet känns sammanhållet och kompetitivt. Projektet demonstrerar hur kombination av flytande rörelse, responsiv kontroll och taktisk nivådesign kan skapa engagerande kompetitivt spelande.

Inledning

Bakgrund

Rörelsebaserade skjutarspel utgör en genre där spelarens skicklighet i att navigera spelmiljön är lika viktig som förmågan att träffa mål. Två spel som exemplifierar detta är Titanfall och RIVALS.

Titanfall, utvecklat av Respawn Entertainment, kännetecknas av ett välkalibrerat rörelsesystem där hastigheter är precisionsanpassade för att överensstämma med spelets ekosystem, inklusive banornas storlek och spelets tempo [1]. Möjligheten att springa på väggar (wallrunning) skapar ett tredimensionellt spelrum som öppnar upp för kreativa lösningar på hur spelaren tar sig fram. Kamerarörelserna och den fysiska feedbacken förstärker känslan av hastighet och kontroll [2].

RIVALS, ett Roblox-spel, tar ett annorlunda approach till vertikal rörelse genom att låta spelaren omvandla horisontell kraft till vertikal kraft via interaktion med omgivningen. Spelaren måste kontinuerligt analysera sin omgivning och använda den strategiskt. Karaktärernas hand- och vapenrörelser samt kamerarörelserna är flytande och förmedlar spelarens hastighet genom visuell feedback [3].

Gemensamt för dessa spel är att de skapar ett sammanhängande ekosystem av rörelsemekaniker som kompletterar varandra. I rörelsebaserade spel är det viktigt att utvecklaren förutser att spelare kommer att kombinera alla möjliga rörelser på kreativa sätt. Detta ger spelaren friheten att uttrycka sig genom sitt spelande och anpassa sig till omgivningen genom sin egen tolkning.

Rörelsebaserade Spel och Kompetitivt Spelande

Skicklighetsbaserade tävlingsspel fokuserar typiskt inte på berättelse eller progression, utan på tävling mellan spelare [4]. Att behärska de verktyg som spelet erbjuder är vad som driver engagemanget. För rörelsebaserade FPS-spel (First Person Shooter) [5] innebär detta att rörelsesystemet måste vara responsivt, precist och tillåta kreativ frihet.

Ett välutvecklat rörelsebaserat spel karakteriseras av flera faktorer. Responsivitet innebär att spelarens input omedelbart översätts till handling utan märkbar fördröjning. Precision kräver att rörelsemekaniken är förutsägbar och konsekvent, vilket möjliggör att spelaren kan utveckla muskelminne och förbättra sina färdigheter över tid. Kreativ frihet uppstår när rörelsesystemet tillåter spelaren att kombinera olika mekaniker på

oväntade sätt, vilket skapar en hög skill ceiling där avancerade spelare kan uttrycka sin skicklighet [6].

Dessa principer är särskilt viktiga i kompetitiva miljöer där balans mellan tillgänglighet för nybörjare och djup för erfarna spelare avgör spelets långsiktiga engagemang.

Syfte

Syftet med detta arbete är att skapa ett rörelsebaserat 3D skjutarspel med flerspelarläge och undersöka hur spel- och nivådesign kan främja kompetitivt spelande genom välutvecklad rörelsemekanik.

Metod

Teknisk Terminologi

För att underlätta läsningen definieras här de viktigaste tekniska begreppen som används i rapporten:

Unity-relaterade begrepp:

Unity: En spelmotor för att skapa 2D- och 3D-spel, utvecklad av Unity Technologies. Programmeringen sker i C# [7].
Package: Tillägg i Unity som utökar funktionaliteten med nya verktyg och komponenter [8].
Prefab: En återanvändbar mall för spelobjekt i Unity [9].
Scene (Scen): En container för spelets miljö och objekt. Basscenen är den primära scenen där grundläggande utveckling sker [10].
Rigidbody: En komponent i Unity som ger ett objekt fysiska egenskaper som massa, gravitation och kraft [11].
Collider: En komponent som definierar ett objekts fysiska form för kollisionsdetektering [12].
Capsule Collider: En kapselformad collider, ofta använd för karaktärer [13].
Raycast: En virtuell stråle som skickas genom spelvärlden för att detektera kollisioner [14].

Programmeringstermer:

Metod: En namngiven kodblock som utför en specifik uppgift. I Unity används metoder för att hantera spellogik [15].

Update(): En metod i Unity som anropas varje bildruta (frame) [16].
LateUpdate(): En metod som anropas efter Update(), ofta använd för kameralogik [17].
OnCollisionStay(): En metod som anropas kontinuerligt medan två objekt kolliderar [18].
Coroutine: En metod som kan pausa sin exekvering och återuppta vid en senare tidpunkt utan att blockera programmet [19].
Mathf.Lerp: En matematisk funktion för linjär interpolation mellan två värden [20].
Mathf.Cos: En trigonometrisk funktion som beräknar cosinus av en vinkel [21].
Quaternion.Slerp: En funktion för sfärisk interpolation mellan två rotationer [22].
State Machine (Tillståndsmaskin): Ett system som hanterar olika tillstånd och övergångar mellan dem [23].
Blend Tree: Ett system i Unity för att blanda animationer baserat på parametrar [24].
Inverse Kinematics (IK): En teknik där slutpositionen av en kedja (t.ex. en arm) beräknas bakåt för att bestämma ledens vinklar [25].
Singleton-mönster: Ett designmönster som säkerställer att endast en instans av en klass existerar [26].

Nätverkstermer:

Repository: En lagringsplats för kod och projektfiler [27].
Version Control (Versionskontroll): System för att spåra ändringar i kod över tid [28].
GitHub: En webbaserad plattform för versionskontroll och samarbete [29].
Interpolation: Teknik för att beräkna mellanliggande värden mellan två datapunkter för smidigare rörelse [30].
RPC (Remote Procedure Call): Ett system som tillåter en klient att anropa metoder på andra klienters instanser över nätverket [31].
PhotonView: En komponent i Photon PUN som identifierar och synkroniserar nätverksobjekt [32].

Spelmekaniska termer:

Hitbox: Det virtuella område runt ett objekt där kollisioner kan registreras [33].
Hitscan: Vapenmekanism där träff registreras omedelbart längs en linje istället för genom en fysisk projektil [34].
FPS (First Person Shooter): Spelgenre där spelaren ser genom karaktärens ögon [35].
Wallrunning: Rörelsemekanism där spelaren kan springa längs vertikala ytor.

ADS (Aim Down Sights): Sikta genom vapnets sikte för högre precision.
Recoil: Vapnets rekyl eller bakåtrörelse när det avfyras.

UI-termer:

TextMeshPro: Ett avancerat textrenderingssystem i Unity som ger skarpare och mer flexibel text än standardkomponenten [36].

Val av Spelmotor

För projektet valdes Unity som spelmotor. Unity är en kraftfull och flexibel spelmotor som används för att utveckla både 2D- och 3D-spel. Motorn har använts för att skapa ett stort antal FPS-spel, inklusive Escape from Tarkov och Rust [37]. Utveckling i Unity sker primärt i programmeringsspråket C#, och i detta projekt användes Visual Studio som kodredigerare.

Unity valdes framför andra spelmotorer av flera anledningar. Jämfört med Godot erbjuder Unity ett mer etablerat ekosystem för FPS-utveckling samt enklare inbyggd versionskontroll genom Unity Version Control (tidigare Plastic SCM) [38]. Unity Version Control möjliggör molnbaserad lagring av repositories och tillåter samtidigt arbete från olika datorer utan att kräva extern konfiguration av GitHub. Jämfört med Unreal Engine 5 är Unity mer lättillgängligt för mindre projekt och kräver inte lika omfattande initial setup [39].

För projektet valdes Unity 2022 LTS (Long-Term Support). LTS-versioner är stabila versioner som får långvarigt stöd och uppdateringar [40]. Den äldre LTS-versionen valdes då det finns mer dokumentation, community-support och färdiga lösningar tillgängliga jämfört med nyare versioner.

Installation och Projektstruktur

Följande packages installerades för att stödja utvecklingen:

ProBuilder: Ett Unity-verktyg för att skapa 3D-geometri direkt i editorn, särskilt lämpligt för prototyping av nivåer [41].
Input System: Unities moderna system för att hantera användarinput från tangentbord, mus och kontroller [42].
Animation Rigging: Ett package för procedurella animationsjusteringar och Inverse Kinematics (IK) [43].
Photon PUN 2: Ett networking-library för realtidsflerspelarfunktionalitet [44].

Materialer och prefabs för grundläggande objekt installerades också. I basscenen skapades ett testområde med hjälp av ProBuilder där grundläggande rörelsemekanik

och gameplay kunde testas. Projektfiler organiserades i mappar som Scripts, Plugins, och Materials för att bibehålla strukturen.

Utveckling av Rörelsesystem

Grundläggande Rörelse

En capsule med capsule collider skapades som temporär representation av spelaren för att testa flerspelarläget. En grundläggande kod för rörelse implementerades baserat på etablerade metoder för FPS-utveckling [45]. Koden hanterade grundläggande funktioner som förflyttning, hopp, duckning och sprint.

Rörelsen fungerar genom att registrera spelarens input och applicera kraft på spelarens rigidbody i motsvarande riktning. Denna metod används för att skapa realistisk fysik där rörelse i luften känns annorlunda än rörelse på marken – spelaren glider mer naturligt istället för att ändra riktning abrupt.

Duckning (Crouch) implementerades genom att minska höjden på capsule collidern, vilket reducerar spelarens hitbox och gör karaktären svårare att träffa.

Hopp (Jump) implementerades genom att applicera en vertikal kraft på rigidbody:n.

Optimering av Rörelse

Den initiala implementationen resulterade i ojämn och ryckig rörelse. Detta berodde på att interpolation inte användes på capsulens rigidbody. Utan interpolation uppdaterades spelarens position synkront med fysikberäkningarna vid en fast tidsfrekvens (Fixed Timestep), vilket inte nödvändigtvis synkroniserar med bildhastigheten (FPS). Detta skapade visuellt hackig rörelse, särskilt för kameran som följer spelaren.

Interpolation innebär att Unity beräknar mellanliggande positioner mellan fysikuppdateringar baserat på aktuell bildhastighet [46]. När interpolation aktiverades på rigidbody:n blev rörelsen betydligt mjukare eftersom systemet fyller i gapen mellan fysikberäkningarna. Detta resulterar i mindre belastning på systemet och en visuellt jämnare upplevelse som anpassar sig till spelarens FPS istället för att vara låst till en fast uppdateringstakt.

Wallrunning-system

Wallrunning-systemet implementerades med hjälp av Unities fysikmotor och bygger på kollisionsdetektion för att avgöra när spelaren befinner sig vid en vertikal yta [47].

Systemet består av tre huvuddelar: detektering av väggar, aktivering av wallrun och rörelsehantering under wallrunning.

Väggdetektering

Detekteringen sker i metoden `OnCollisionStay()`, som anropas kontinuerligt medan spelarens collider nuddar ett annat objekt [48]. För varje kontaktpunkt mellan spelaren och objektet analyseras kontaktens normalvektor – en vektor som pekar vinkelrätt ut från ytan.

Om vinkeln mellan normalvektorn och `Vector3.up` (Unities representation av uppåtriktningen i världskoordinater [49]) ligger nära 90 grader (inom cirka 5 grader), indikerar detta att ytan är vertikal. Om ytan dessutom tillhör lagret "Ground" och spelaren inte redan står på marken, kan wallrunning initieras.

Aktivering av Wallrun

När villkoren för wallrunning är uppfyllda anropas metoden `StartWallRun()`. I denna metod sparas väggens normalvektor, och om spelaren inte redan utför wallrunning återställs den vertikala hastigheten till noll för att skapa en jämn övergång. Därefter appliceras en uppåtriktad kraft för att simulera att spelaren "trycker sig fast" mot väggen.

En boolesk flagga (`wallRunning`) sätts till true, vilket signalerar till andra delar av koden att spelaren nu befinner sig i wallrunning-läge. En cooldown-mekanism via variabeln `readyToWallrun` förhindrar att spelaren omedelbart kan starta en ny wallrun efter att ha lämnat en vägg, vilket skapar mer balanserad gameplay och minskar risken för buggar.

Rörelsehantering under Wallrunning

När flaggan `wallRunning` är aktiv körs logiken i metoden `WallRunning()` varje frame i `LateUpdate()`. En kraft riktas mot väggen (i motsatt riktning av normalvektorn) för att hålla spelaren pressad mot väggen. Samtidigt appliceras en reducerad uppåtriktad kraft som delvis motverkar gravitationen, vilket gör att spelaren glider kontrollerat längs väggen istället för att falla rakt ned. Styrkan av denna uppåtriktade kraft är lägre än normal gravitation, vilket skapar en gradvis nedåtgående rörelse längs väggen.

Avslutning av Wallrun

När spelaren lämnar väggen – antingen genom att hoppa, släppa input-tangenten, eller när ingen giltig vägg längre detekteras – anropas metoden `StopWall()`. I metoden

`StopWall()` återställs flaggorna `onWall` och `wallRunning` till false, vilket innebär att normal gravitation återgår.

För att förhindra att spelaren fastnar på väggen direkt efter avslutad wallrun används metoden `CancelWallrun()` tillsammans med en tidsfördröjning via `Invoke()`. Detta skapar en kort paus innan nästa wallrun kan initieras.

Multiplayer-implementation

Multiplayer-funktionaliteten implementerades med Photon PUN 2 (Photon Unity Networking), ett Software Development Kit för realtidsnätverkande i Unity [50]. Photon PUN 2 tillhandahåller färdiga komponenter som förenklar synkronisering av spelarpositioner och spellogik över nätverket.

Setup och Rumhantering

Efter att Photon PUN 2-paketet importerades i projektet skapades grundläggande nätverksfunktionalitet. Ett initialt skript implementerades som automatiskt placerar alla spelare i samma rum, vilket fungerade som en tillfällig lösning under tidig utveckling.

Synkronisering av Rörelser

För att synkronisera spelarnas rörelser mellan olika klienter användes komponenten Photon Transform View. Denna komponent placerades på spelarobjektet och konfigurerades för att skicka positions- och rotationsdata till servern, vilket möjliggör att förändringar uppdateras i realtid för alla anslutna spelare. En instans av komponenten placerades på spelarens huvudobjekt för att synkronisera position, och ytterligare en på orienteringskomponenten för att andra spelare ska kunna se vilken riktning spelaren är vänd åt.

Implementering av Lobby och Room Management

För att hantera nätverksanslutningar och organisera spelare i sessioner implementerades ett centralt system baserat på klassen RoomManager. Detta script använder Singleton-mönstret, vilket säkerställer att endast en instans av managern existerar genom hela spelets livscykel. Detta åstadkoms med Unity-metoden `DontDestroyOnLoad()` [51], som förhindrar att objektet förstörs när nya scener laddas.

Systemet kommunicerar med Photon-servern i flera steg. Först etableras en anslutning till Master-servern med metoden `ConnectUsingSettings()` [52], och därefter ansluts spelaren automatiskt till en nätverkslobby.

Automatiserat Värdsystem

Värdsystemet automatiserades genom metoden `JoinOrCreateRoom()` [53]. Istället för att tvinga spelaren att manuellt välja en server försöker systemet ansluta till ett fördefinierat rum med namnet "test". Om rummet inte existerar skapas det automatiskt med specifika RoomOptions, såsom ett tak för antal spelare (maxPlayers). Detta skapar en sömlös användarupplevelse där övergången från programmets start till en aktiv spelsession sker utan manuell konfiguration.

Dynamisk Spelarhantering

En central del av RoomManager är hanteringen av spelarinstanser baserat på rummets status. Systemet skiljer på `singlePlayerSpawn` och `multiplayerSpawn`. När en spelare ansluter till ett rum kontrolleras antalet deltagare. Om spelaren är ensam placeras denne i ett väntrum eller en träningsyta. Så fort en andra spelare ansluter aktiveras en mer avancerad nätverkslogik via ett RPC (Remote Procedure Call) kallat `RPC_RespawnAtMultiplayerSpawn` [54].

Master-klienten (den första spelaren som skapade rummet) instruerar här samtliga anslutna klienter att förstöra sina nuvarande lokala spelarobjekt och omedelbart instansiera nya på multiplayer-positionerna. Denna process hanteras via en Coroutine (`RespawnRoutine`) för att säkerställa att Photon-serverns tillstånd hunnit stabiliseras mellan destruktion och ny instansiering, vilket minimerar risken för synkroniseringsfel.

Vapensystem

Hitscan-implementation

Vapensystemet implementerades som hitscan-vapen [55]. Hitscan-vapen innebär att en osynlig raycast skickas från kamerans centrum rakt framåt tills den träffar ett objekt. Om raycasten träffar en spelare vid samma tidpunkt som vapnet avfyras registreras detta som en träff. Till skillnad från projektilbaserade vapen skapas ingen fysisk projektil som färdas genom spelmiljön.

Hitscan valdes för att minimera nätverksfördröjning (lag) och maximera responsiviteten i spelupplevelsen. Hitscan-vapnen kodades så att en raycast skickas från spelarens kamera. Om raycasten träffar ett objekt som innehåller ett Health-script registreras en träff och skada appliceras på objektet [56].

Vapenlogik och Visuella Effekter

Det centrala scriptet för spelets stridsmekanik, Gun.cs, hanterar interaktionen mellan spelaren och omvärlden genom en kombination av ammunitionshantering, raycast och procedurell visuell feedback.

Ammunitionshantering och Omladdning

Ammunitionshanteringen bygger på variablerna `magazineSize` och `ammoInMag`. Logiken för omladdning implementerades som en Coroutine, vilket tillåter systemet att pausa exekveringen under en definierad `reloadTime` utan att blockera programmets huvudtråd [57]. Under omladdningsprocessen används matematiska funktioner som Mathf.Lerp och Quaternion.Slerp [58] för att fysiskt rotera vapnets transform i en nedåtgående rörelse, vilket skapar en realistisk omladdningsanimation direkt via kod.

Räckvidd och Skadehantering

För att reglera vapnets effektivitet används variabeln `range`, som definierar det maximala avståndet för raycasten som skickas från spelarens kamera. När en träff registreras mot ett objekt med en hälsokomponent kommunicerar scriptet detta till nätverket via Photon PUN:s RPC-system [59]. Genom att anropa `RPC_TakeDamage()` säkerställs att skadan registreras korrekt på den träffade spelarens instans, oavsett vilken klient som avfyrade skottet.

Visuella och Auditiva Effekter

Vid varje skott aktiveras flera visuella och auditiva element. En `muzzleFlash` visas vid vapnets mynning samtidigt som ett skottljud spelas upp. För att visualisera träffar i miljön instansieras en `hitEffectPrefab` vid träffpunkten, orienterad efter ytans normalvektor för att simulera gnistor eller damm.

Dessutom skapas ett skotthål genom `bulletHolePrefab`, som via kod görs till ett underobjekt till det träffade kolliderobjektet. Detta garanterar att den visuella markeringen följer med om målet rör på sig. För att synkronisera dessa effekter för alla deltagare i rummet används metoden `RPC_PlayShotEffects()`, som instruerar övriga klienter att instansiera både mynningsflammor och en `bulletTracer` som ritar upp kulans bana mellan vapnet och träffpunkten.

Användargränssnitt för Ammunition

För att spelaren ska kunna hålla reda på sina resurser implementerades ett dynamiskt användargränssnitt som visar aktuell ammunitionsnivå. Systemet använder

TextMeshPro [60] för att säkerställa hög visuell skärpa oavsett skärmupplösning. UI-elementet är kopplat till Gun.cs via variabeln `AmmoCount`.

Funktionaliteten bygger på reaktiv uppdateringslogik snarare än att belasta systemet med beräkningar varje bildruta. Genom metoden `UpdateAmmoUI()` uppdateras textsträngen endast vid specifika händelser: när ett skott avfyras och `ammoInMag` minskar, samt när omladdningssekvensen slutförts och magasinet återställts. Texten formateras med stränginterpolation [61] för att visa både nuvarande antal skott och vapnets totala magasinskapacitet (t.ex. "30 / 30").

I flerspelarläget är UI-hanteringen begränsad till den lokala spelaren genom en kontroll av `photonView.IsMine` [62]. Om scriptet körs på en instans som tillhör en annan spelare inaktiveras ammunitionsvisningen automatiskt via `SetActive(false)` [63]. Detta förhindrar att en spelares UI-element visar data från andra spelares vapen eller skapar visuella konflikter.

Procedurella Vapenrörelser

Vapenanimationerna och rörelseeffekterna implementerades inte genom traditionella animationer utan genom matematiska beräkningar i realtid [64]. Detta möjliggjorde mer dynamiska och responsiva vapenrörelser som reagerar direkt på spelarens handlingar.

Gångrytm och Vapenvagning

För att simulera naturlig rörelse när spelaren går implementerades en oscillerande upp-och-ner-rörelse på vapnet. Detta åstadkoms genom att använda en cosinus-funktion (Mathf.Cos) som genererar värden som varierar mellan 1 och -1 baserat på tiden. Dessa värden adderas till vapnets Y-koordinat, vilket skapar en vågrörelseseffekt synkroniserad med spelarens gångcykel. Vapnet vinklar sig också beroende på spelarens rörelseriktning, vilket förstärker känslan av momentum och riktningsförändring.

ADS (Aim Down Sights)

Siktsystemet implementerades genom en boolesk variabel som aktiveras vid högerklick. När spelaren siktar används en Lerp-funktion för att mjukt förflytta vapnet från höftposition till en position närmare spelarens ansikte och kamera. Lerp-funktionen interpolerar gradvis mellan start- och målposition, vilket skapar en smidig övergång istället för en abrupt förflyttning.

Recoil-system

Rekylen implementerades med två separata Lerp-funktioner – en för vertikal rörelse (Y-axel) och en för horisontell rörelse (X-axel). När vapnet avfyras läggs slumpmässiga värden till vapnets rotation och position, vilket simulerar rekylens kraft. Vapnet trycks bakåt och uppåt med varierande intensitet för att skapa en naturlig känsla.

Ett ackumulerande recoil-system implementerades där vapnet inte hinner återhämta sig helt mellan skott vid snabb eld. Varje skott adderar rekyl till den befintliga rörelseseffekten. Lerp-funktionens återhämtningshastighet är kalibrerad så att vid kontinuerlig eld når vapnet en maximal rekylhöjd där Lerp-funktionens återhämtning balanserar den tillagda rekylen från varje skott. Detta skapar en realistisk "spray pattern" där vapnet gradvis stiger i Y-led tills det når en jämvikt.

Animationssystem

Avatar och Grundläggande Animationer

En humanoid avatar importerades från Unity Asset Store [65] och ersatte den tidigare capsule-representationen. Avataren levererades med en förberedd benstruktur (rig). Animationer för gående och springande i olika riktningar laddades ner och implementerades [66].

En state machine (tillståndsmaskin) skapades med hjälp av Unities Animation Rigging-package [67]. Inverse Kinematics (IK) implementerades för att möjliggöra realistiska hand- och vapenpositioner [68][69].

Hierarkisk Blend Tree-struktur

För att åstadkomma smidiga övergångar mellan olika animationstillstånd implementerades en hierarkisk Blend Tree-struktur. Blend Trees är nödvändiga för flytande animationer – utan dem skulle animationer byta abrupt vid input-ändringar, vilket skapar en hackig visuell upplevelse.

Överordnad Blend Tree (1D)

Den överordnade Blend Tree:n är en 1D Blend Tree som fungerar som huvudkontrollsystem (se Figur 1). Denna innehåller två underordnade Blend Trees:

> **Locomotion Blend Tree** - Hanterar all stående rörelse (gående, springande, sidsteg)
> **Sliding/Crouching Blend Tree** - Hanterar duckning och glidrörelse

Strukturen med en överordnad 1D Blend Tree möjliggör mjuk interpolation mellan dessa två huvudtillstånd. Parametern som styr den överordnade Blend Tree:n är

kopplad till spelarens rörelsestatus (stående kontra duckande/glidande). Detta betyder att spelaren, oavsett vilken rörelse som utförs i locomotion-läget, alltid kan göra en smidig övergång till sliding eller crouching.

Locomotion Blend Tree (2D)

Den underordnade Locomotion Blend Tree:n är en 2D Blend Tree som hanterar all normal rörelse (se Figur 2). Här används parametrarna Move X och Move Y, som styrs av spelarens tangentbordsinput (typiskt WASD-tangenterna) via movement-scriptet.

Denna Blend Tree fungerar som ett tvådimensionellt koordinatsystem där varje punkt representerar en kombination av rörelseriktningar. Animationer för olika riktningar (framåt, bakåt, vänster, höger, samt diagonala riktningar) placeras ut som punkter i detta 2D-utrymme. Den röda punkten i Figur 2 representerar spelarens aktuella rörelseriktning.

När spelaren rör sig beräknar Blend Tree:n automatiskt vilka animationer som ligger närmast den aktuella positionen och blandar dem proportionerligt. Exempelvis, om spelaren rör sig diagonalt framåt-höger kommer animationerna för "gå framåt" och "gå höger" att blandas för att skapa en naturlig diagonal rörelseanimation.

Sliding/Crouching Blend Tree

Den andra underordnade Blend Tree:n hanterar animationer relaterade till duckning och glidrörelse. Denna är strukturerad på liknande sätt som Locomotion Blend Tree:n men innehåller animationer för lägre kroppspositioner och snabbare glidrörelse.

Mjuk Interpolation med Mathf.Lerp

Ett kritiskt problem uppstår om Move X och Move Y sätts direkt baserat på spelarens input. I sådana fall skulle den röda punkten i Blend Tree:n omedelbart "teleportera" till den nya positionen, vilket resulterar i abrupt växlande mellan animationer.

Lösningen implementerades med `Mathf.Lerp` (Linear Interpolation) [70], som skapar en gradvis övergång mellan värden. Varje frame i `Update()` [71] beräknas nya värden för Move X och Move Y som ligger någonstans mellan det nuvarande värdet och målinputvärdet. Detta skapar en mjuk acceleration och deceleration i animationerna.

Resultatet är att den röda punkten i Blend Tree:n glider mjukt mellan olika positioner istället för att hoppa abrupt. Kombinerat med den hierarkiska strukturen möjliggör detta naturliga övergångar mellan alla rörelsetillstånd – från sprint till slide, från gång till duckning, eller mellan vilka rörelser som helst.

IK Constraints för Vapenhållning

För att säkerställa att avataren håller vapnet korrekt under alla animationer implementerades flera komponenter från Animation Rigging-paketet [72]:

Two-Bone IK Constraint användes för att positionera händerna exakt på vapnet. Denna constraint beräknar armledens och handledens vinklar baserat på vapnets position, vilket säkerställer att händerna alltid griper vapnet korrekt oavsett dess position [73].

Multi-Aim Constraint applicerades på avatarens huvud och överkropp för att skapa naturliga blickriktningar och kroppspositioner. När spelaren tittar uppåt eller nedåt justeras huvudet och överkroppen automatiskt så att rörelserna ser realistiska ut från tredjepersonsperspektiv, vilket är viktigt för hur andra spelare ser avataren i flerspelarläget [74].

Rig-komponenten fungerar som huvudcontainer för alla IK-constraints och hanterar beräkningsordningen för att säkerställa korrekt hierarki mellan olika kroppsdelar [75].

Kombinationen av dessa IK-constraints gör att vapnet kan röras procedurellt under reload-animationer eller springrörelse, medan händerna automatiskt följer efter och bibehåller greppet. Detta eliminerar behovet för manuellt animerade handrörelser för varje vapenposition och skapar ett mer dynamiskt och responsivt system.

Resultat

Projektet resulterade i ett fungerande rörelsebaserat 3D skjutarspel med flerspelarläge för två spelare. Majoriteten av de planerade systemen implementerades framgångsrikt och fungerar som avsett.

Implementerade System

Rörelsesystem

Alla grundläggande rörelsemekaniker implementerades och fungerar utan märkbar input-fördröjning. Systemet inkluderar hopp, duckning, sprint och glidning. Varje rörelsetyp har implementerats med dynamisk FOV-anpassning (Field of View) som förmedlar känslan av hastighet till spelaren. Rörelserna integreras sömlöst och tillåter flytande övergångar mellan olika tillstånd.

Wallrunning-systemet implementerades för spelarens lokala upplevelse och fungerar på alla vertikala ytor. Systemet detekterar väggar korrekt och möjliggör rörelse längs

vertikala ytor enligt specifikationen. Dock uppstod tekniska begränsningar gällande animationssynkronisering i flerspelarläget, vilket beskrivs vidare i Diskussion-avsnittet.

Vapensystem

Hitscan-vapensystemet fungerar över nätverket utan märkbar fördröjning. Alla visuella effekter implementerades framgångsrikt, inklusive muzzle flash vid vapnets mynning, bullet tracers som visualiserar kulans bana, och bullet impacts som skapar skotthål på träffade ytor. Dessa effekter synkroniseras korrekt mellan klienter – bullet tracers och impacts är synliga för alla spelare i rummet.

Recoil-systemet kalibrerades för att ge en bekant känsla för spelare med FPS-erfarenhet. Det ackumulerande recoil-systemet fungerar som designat, där vapnet gradvis stiger vid kontinuerlig eld tills en jämvikt uppnås mellan rekyl och återhämtning.

Ammunition-UI:t, implementerat med TextMeshPro, fungerar korrekt för varje spelare individuellt och uppdateras reaktivt vid relevanta händelser.

Multiplayer-funktionalitet

Lobby- och room management-systemet fungerar enligt specifikationen. Systemet hanterar automatisk rumskapande och anslutning, samt dynamisk spelarhantering baserat på antalet deltagare. Övergången från single-player spawn till multiplayer spawn sker sömlöst när en andra spelare ansluter.

Synkroniseringen av spelarpositioner, rotationer och handlingar fungerar utan märkbar nätverksfördröjning. Photon Transform View-komponenterna säkerställer att alla spelarrörelser uppdateras i realtid för båda klienterna.

Animationssystem

Den hierarkiska Blend Tree-strukturen fungerar som avsett och skapar smidiga övergångar mellan alla rörelsetillstånd utom wallrunning. Systemet hanterar kombinationer av rörelser naturligt, exempelvis övergången från sprint till slide eller från gång till duckning.

IK constraints för vapenhållning fungerar korrekt. Two-Bone IK Constraint positionerar händerna exakt på vapnet under alla animationer, och Multi-Aim Constraint justerar huvudet och överkroppen baserat på spelarens blickriktning. Detta skapar realistiska animationer från tredjepersonsperspektiv för andra spelare.

Ett undantag är wallrunning-animationen, som endast fungerar från förstapersonsperspektiv men inte synkroniserar korrekt i flerspelarläget. Spelaren som

utför wallrunning upplever rörelsen normalt, men andra spelare ser inte motsvarande animation.

Procedurella Vapenrörelser

De matematiskt baserade vapenrörelserna fungerar som designat. Gångrytm, vapenvagning, ADS-övergångar och recoil implementerades alla procedurellt och reagerar direkt på spelarens handlingar. Kombinationen av dessa system skapar en dynamisk känsla inspirerad av RIVALS, där vapnet och kameran rör sig i reaktion till spelarens momentum.

Teknisk Prestanda

Spelet uppnår över 400 FPS på testdatorn och bör vara spelbart på både stationära datorer och bärbara datorer. Inga märkbara prestandaproblem identifierades under testning. Relevanta specifikationerna för test datorn var följade:

Grafikkort : GeFore RTX 4060 ti

CPU: Ryzen 7 5800x

32GB DDR5 5200Mhz RAM

Kända Begränsningar

En bugg identifierades där hitscan-raycasten inte fungerar vid en specifik vinkel. Detta påverkar cirka 10% av möjliga siktvinklar och misstänks bero på att delar av spelarens egen modell blockerar raycasten vid extrema vinklar. Buggen upptäcktes sent i projektet och hann inte åtgärdas.

Wallrunning-animationen synkroniserar inte korrekt i flerspelarläget, vilket innebär att andra spelare inte ser den animationen när en spelare utför wallrunning.

Användarfeedback

Testning genomfördes med flera personer med erfarenhet av FPS-spel. Feedbacken indikerade att spelet kändes sammanhållet och igenomtänkt. Rörelseekosystemet upplevdes som intuitivt och responsivt, och integration mellan olika mekaniker fungerade enligt förväntningarna.

Diskussion

Detta avsnitt analyserar projektets resultat, utmaningar, designbeslut och hur väl syftet uppnåddes.

Tekniska Utmaningar

Wallrunning-animationen i Flerspelarläge

Den mest betydande tekniska utmaningen var implementeringen av wallrunning-animationer som fungerar korrekt i flerspelarläget. Problemet härstammar från den inneboende konflikten mellan anatomi och spelmekanik. För att wallrunning ska se realistiskt ut måste benen springa längs väggen medan överkroppen behöver kunna rotera 360 grader för att spelaren ska kunna sikta fritt. Detta är anatomiskt omöjligt för en mänsklig kropp.

Många kommersiella spel löser detta genom att begränsa hur mycket spelaren kan rotera kameran under wallrunning. Dock kräver denna lösning omfattande anpassade animationer för olika väggorienteringar och kamerarotationer. Tillgång till sådana animationsresurser var begränsad, då professionella wallrunning-animationer sällan finns tillgängliga gratis och kräver avancerad kunskap i animationsrigging för att skapa från grunden.

Ett tillvägagångssätt som testades var att använda IK constraints för att fästa benen till väggen medan kroppen roterar. Detta skulle kombineras med den befintliga väggnormaldetekteringen från movement-systemet. Utmaningen blev att benen måste förbli fästa vid väggen även när kroppen rör sig, vilket skapar en komplex kedjereaktionslogik som kräver avancerad koordination mellan flera animationssystem. Tidsramen för projektet tillät inte fullständig implementation av denna lösning.

I efterhand skulle ett enklare designbeslut kunnat fattas tidigare i projektet, såsom att wallrunning automatiskt avbryts efter en kort tid eller att begränsa kamerarotationen hårdare. Detta hade kringgått animationsproblemen till priset av något reducerad spelarfrihet.

Hitscan-buggen vid Specifika Vinklar

Den identifierade hitscan-buggen där raycasten inte registrerar träffar vid vissa vinklar misstänks bero på kollisioner med spelarens egen modell. När spelaren tittar i extrema vinklar (exempelvis rakt nedåt) kan delar av avatarens geometri sticka ut framför kameran och blockera raycasten innan den når målet.

En möjlig lösning hade varit att placera raycastens ursprung något framför kameran eller att exkludera spelarens egna colliders från raycast-beräkningarna genom användning av Unity's layer-system. Buggen upptäcktes dock sent i projektet när fokus låg på att färdigställa andra system, och prioriterades därför inte för åtgärdning innan deadline.

Animations- och Multiplayer-komplexitet

Den mest överraskande lärdomen från projektet var komplexiteten i att skapa trovärdiga karaktärsanimationer som beter sig naturligt under alla omständigheter. Att uppnå smidiga övergångar mellan animationer krävde omfattande detaljarbete och fintuning av Blend Trees och interpolationsparametrar.

Ett konkret exempel är slide-mekaniken. När spelaren avbryts mitt i en slide måste systemet intelligent välja nästa animation baserat på kontext – om spelaren stannar helt övergår avataren till en knästående position, medan om spelaren fortsätter röra sig övergår animationen till gång eller sprint. Denna typ av kontextuell animationslogik krävde betydligt mer arbete än förväntat.

Multiplayer-aspekten adderade ytterligare komplexitet genom kravet att separera inputs och tillstånd mellan olika klienter. Photon PUN 2:s RPC-system och PhotonView-komponenter underlättade dock denna process avsevärt jämfört med att implementera ett eget nätverkssystem från grunden.

Designbeslut och Motiveringar

Modulärt Vapensystem

Vapensystemet designades med modularitet i åtanke. Variabler som `range`, `spread`, `magazineSize`, och `reloadTime` kan justeras per vapen utan att modifiera kodbasen. Detta möjliggör enkel skapande av nya vapen med olika egenskaper, vilket är värdefullt för framtida expansion av spelet.

Recoil-balansering

Balanseringen av recoil-systemet baserades på erfarenhet från andra FPS-spel snarare än systematisk testning med fokusgrupper. Målet var att skapa en bekant känsla för spelare med FPS-bakgrund, vilket skulle minska inlärningskurvan och göra spelet mer tillgängligt. Det ackumulerande recoil-systemet, där vapnet gradvis stiger vid kontinuerlig eld, efterliknar mekaniken i etablerade titlar som Counter-Strike och Valorant.

Separerad Klientlogik för Visuella Effekter

Vissa visuella effekter, såsom muzzle flash och vapenanimationer, körs lokalt på varje klient snarare än att synkroniseras över nätverket. Detta minskar nätverksbelastningen och förbättrar responsiviteten. Effekter som måste vara synkroniserade för spelupplevelsens skull, såsom bullet tracers och impacts, synkroniseras via RPC-anrop. Denna hybrid-approach balanserar prestanda med synkroniseringskrav.

Uppfyllande av Syfte

Projektets syfte var att "skapa ett rörelsebaserat 3D skjutarspel med flerspelarläge och undersöka hur spel- och nivådesign kan främja kompetitivt spelande genom välutvecklad rörelsemekanik."

Kompetitivt Spelande genom Rörelsemekanik

Spelet uppnår kompetitivitet genom att skapa en miljö där spelaren har multipla val och kan uttrycka skicklighet genom både rörelse och precision. Det flytande rörelsesystemet ger erfarna spelare möjlighet att utnyttja avancerade tekniker som slide-jumps och momentum-bevarande mellan rörelser, medan systemet förblir tillgängligt för nybörjare genom intuitiva grundkontroller.

Nivådesignen påverkar direkt vilka rörelsestrategier som är effektiva. Vertikala element möjliggör wallrunning (där det fungerar), öppna ytor gynnar snabb förflyttning med sprint och slide, medan trånga utrymmen kräver mer försiktig navigation. Denna variation i miljödesign tvingar spelare att anpassa sin strategi beroende på position, vilket ökar spelets taktiska djup.

Kombinationen av rörelse och aim skapar en skill ceiling där båda aspekterna är viktiga för framgång. En spelare med överlägsen aim men dålig rörelse kan besegras av en motståndare som utnyttjar momentum och positionering för att vara svår att träffa.

Kreativitet och Spelarbeteende

Feedbacken från testspelare med FPS-erfarenhet bekräftade att rörelsesystemet kändes sammanhållet och tillät kreativa lösningar. Spelare kunde experimentera med olika rörelsekombinationer och upptäcka effektiva strategier genom trial-and-error, vilket alignar med målet att ge spelare friheten att uttrycka sig genom sitt spelande.

Inspiration från Titanfall och RIVALS märks tydligt i slutresultatet. Likt Titanfall erbjuder spelet tredimensionell rörelse och snabb navigation, medan det procedurella vapenbeteendet från RIVALS bidrar till känslan av hastighet och momentum.

Verktygsval

Unity och Photon PUN 2

Unity visade sig vara ett lämpligt val för projektet. Motorns tillgänglighet och omfattande dokumentation underlättade utvecklingsprocessen betydligt. Unity Version Control fungerade väl för samtidigt arbete utan konflikter.

Photon PUN 2 integrerades smidigt och hanterade nätverksfunktionaliteten på ett tillförlitligt sätt. RPC-systemet och färdiga komponenter som PhotonView och PhotonTransformView reducerade komplexiteten avsevärt jämfört med att implementera ett eget nätverkssystem.

Dock identifierades begränsningar i tillgängliga resurser för Unity jämfört med Unreal Engine. Specifikt för wallrunning-problemet finns det etablerade plugins och community-resurser i Unreal Engine som hade kunnat påskynda utvecklingen. För projekt med komplexa animationskrav eller avancerade rörelseemekaniker kan Unreal Engine vara att föredra, även om det kräver högre teknisk kompetens och mer omfattande setup.

Projektomfattning och Tidshantering

Projektets omfattning bedöms som välbalanserad i förhållande till tidsramen. Majoriteten av de planerade funktionerna implementerades framgångsrikt. De enda betydande avstegen från ursprungsplanen var wallrunning-animationen i multiplayer och hitscan-buggen, vilka båda upptäcktes sent och inte prioriterades för åtgärdning.

I retrospektiv hade mer tid allokerats till research av wallrunning-implementation innan utvecklingen påbörjades. Detta hade möjliggjort ett tidigare designbeslut om huruvida fullständig wallrunning var genomförbart inom projektets ramar, eller om en förenklad variant skulle implementeras istället.

Lärdomar

Den mest betydande lärdomen är förståelsen för hur spelutveckling fungerar på komponentnivå. Kod är essentiellt komponenter som styr fysiska objekt i spelmiljön, och dessa komponenter måste designas för att fungera tillsammans utan att "interfere" med varandras funktionalitet. Detta kräver noggrann planering av systemarkitektur och tydliga gränssnitt mellan olika systems.

Animationssystem visade sig vara mer komplexa än förväntat, särskilt kraven för att skapa naturliga övergångar mellan tillstånd. Multiplayer-aspekten adderade ytterligare komplexitet genom behovet att separera och synkronisera tillstånd mellan klienter.

Positivt var att grundläggande FPS-mekanik (rörelse och skjutande) var enklare att implementera än förväntat tack vare tillgängliga tutorials och dokumentation online. Detta möjliggjorde mer tid att fokusera på polish och detaljarbete i andra områden.

Slutsats

Detta projekt har framgångsrikt utvecklat ett fungerande rörelsebaserat 3D skjutarspel med flerspelarläge som uppfyller majoriteten av projektets ursprungliga mål. Spelet demonstrerar hur välutvecklad rörelsemekanik kan främja kompetitivt spelande genom att ge spelaren multipla strategiska val och möjligheten att uttrycka skicklighet genom både rörelse och precision.

De implementerade systemen – grundläggande rörelse, hitscan-vapen, procedurella animationer, multiplayer-funktionalitet och animationssystem – fungerar sammanhållet och skapar en responsiv spelupplevelse. Det flytande rörelsesystemet med dynamisk FOV-anpassning och sömlösa övergångar mellan tillstånd uppnår målet att skapa ett snabbt och uttrycksfullt spel.

Projektet mötte tekniska utmaningar, särskilt gällande wallrunning-animationer i flerspelarläget, vilket belyser komplexiteten i att utveckla avancerade rörelseemekaniker med trovärdiga animationer. Trots dessa begränsningar uppnår spelet sitt syfte och har verifierats genom positiv feedback från testspelare med FPS-erfarenhet.

Rekommendationer för Framtida Utveckling

För vidareutveckling rekommenderas följande prioriteringar:

Wallrunning-animationen: Implementera en förenklad variant av wallrunning som begränsar kamerarotation men möjliggör korrekt animationssynkronisering, alternativt utforska användning av Unreal Engine för tillgång till etablerade wallrunning-plugins.
Hitscan-buggen: Implementera layer-baserad raycast-filtrering för att exkludera spelarens egen modell från hitscan-beräkningar.

Content-expansion: Det modulära vapensystemet möjliggör enkel addition av nya vapen med varierande egenskaper. Flera kartor med olika layouts skulle ytterligare testa spelarnas rörelseförmåga och taktiska anpassning.

Ytterligare rörelsetyper rekommenderas dock inte, då det finns en gräns för hur komplext movement kan vara innan det blir oöverskådligt för den genomsnittliga spelaren.

Avslutande Reflektion

Unity i kombination med Photon PUN 2 rekommenderas för liknande FPS-projekt där snabb prototyping och robust multiplayer-funktionalitet är prioritet. För projekt med höga krav på komplexa animationssystem eller avancerade rörelseemekaniker kan Unreal Engine vara att föredra, förutsatt att utvecklarna har nödvändig erfarenhet för att hantera dess högre komplexitet.

Projektet har gett värdefull insikt i spelmekanisk design, animationssystem, multiplayer-arkitektur och vikten av tidigt designbeslut för tekniskt komplexa features. Slutresultatet är ett spelbart och sammanhållet FPS-spel som demonstrerar principerna för rörelsebaserat kompetitivt spelande.

Referenser

[1] Respawn Entertainment (2014). *Titanfall*. Electronic Arts.

[2] GamesRadar+ (2014). "Titanfall review". Tillgänglig: https://www.gamesradar.com/titanfall-review/

[3] Roblox Corporation. *RIVALS*. Tillgänglig: https://www.roblox.com/games/17625359962/RIVALS

[4] Ryan, R. M., Rigby, C. S., & Przybylski, A. (2006). "The motivational pull of video games: A self-determination theory approach". Motivation and Emotion, 30(4), 344-360. https://doi.org/10.1007/s11031-006-9051-8

[5] Wikipedia. "First-person shooter". Tillgänglig: https://en.wikipedia.org/wiki/First-person_shooter

[6] Sirlin, D. (2006). "Playing to Win: Becoming the Champion". *Accessible at sirlin.net*.

[7] Unity Technologies. "Unity User Manual". Tillgänglig: https://docs.unity3d.com/Manual/index.html

[8] Unity Technologies. "Packages". Tillgänglig: https://docs.unity3d.com/Manual/Packages.html

[9] Unity Technologies. "Prefabs". Tillgänglig: https://docs.unity3d.com/Manual/Prefabs.html

[10] Unity Technologies. "Scenes". Tillgänglig: https://docs.unity3d.com/Manual/CreatingScenes.html

[11] Unity Technologies. "Rigidbody". Tillgänglig: https://docs.unity3d.com/ScriptReference/Rigidbody.html

[12] Unity Technologies. "Collider". Tillgänglig: https://docs.unity3d.com/ScriptReference/Collider.html

[13] Unity Technologies. "Capsule Collider". Tillgänglig: https://docs.unity3d.com/ScriptReference/CapsuleCollider.html

[14] Unity Technologies. "Raycast". Tillgänglig: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[15] Microsoft. "Methods (C# Programming Guide)". Tillgänglig: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods

[16] Unity Technologies. "Update()". Tillgänglig: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

[17] Unity Technologies. "LateUpdate()". Tillgänglig: https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html

[18] Unity Technologies. "OnCollisionStay()". Tillgänglig: https://docs.unity3d.com/ScriptReference/Collider.OnCollisionStay.html

[19] Unity Technologies. "Mathf.Lerp". Tillgänglig: https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html

[20] Unity Technologies. "Mathf.Cos". Tillgänglig: https://docs.unity3d.com/ScriptReference/Mathf.Cos.html

[21] Unity Technologies. "State Machine Basics". Tillgänglig: https://docs.unity3d.com/Manual/StateMachineBasics.html

[22] Unity Technologies. "Blend Trees". Tillgänglig: https://docs.unity3d.com/Manual/class-BlendTree.html

[23] Unity Technologies. "Inverse Kinematics". Tillgänglig: https://docs.unity3d.com/Manual/InverseKinematics.html

[24] GitHub. "About repositories". Tillgänglig: https://docs.github.com/en/repositories

[25] Atlassian. "What is version control". Tillgänglig: https://www.atlassian.com/git/tutorials/what-is-version-control

[26] GitHub. "GitHub". Tillgänglig: https://github.com

[27] Unity Technologies. "Rigidbody interpolation". Tillgänglig: https://docs.unity3d.com/ScriptReference/Rigidbody-interpolation.html

[28] Giant Bomb. "Hitbox". Tillgänglig: https://www.giantbomb.com/hitbox/3015-2424/

[29] TV Tropes. "Hitscan". Tillgänglig: https://tvtropes.org/pmwiki/pmwiki.php/Main/Hitscan

[30] Wikipedia. "First-person shooter". Tillgänglig: https://en.wikipedia.org/wiki/First-person_shooter

[31] Unity Technologies. "Unity Showcase". Tillgänglig: https://unity.com/showcase

[32] Unity Technologies. "Unity Version Control". Tillgänglig: https://unity.com/products/version-control

[33] TechRadar (2023). "Unity vs Unreal Engine". Tillgänglig: https://www.techradar.com/news/unity-vs-unreal-engine

[34] Unity Technologies. "Unity LTS releases". Tillgänglig: https://unity.com/releases/lts

[35] Unity Technologies. "ProBuilder". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.probuilder@6.0/manual/index.html

[36] Unity Technologies. "Input System". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.17/manual/index.html

[37] Unity Technologies. "Animation Rigging". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/index.html

[38] Photon Engine. "PUN 2". Tillgänglig: https://doc.photonengine.com/pun/current/getting-started/pun-intro

[39] Dave / GameDevelopment (2021). "First Person Movement in Unity - FPS Controller". YouTube. Tillgänglig: https://www.youtube.com/watch?v=wDmXmgvzuel

[40] Unity Technologies. "Rigidbody.interpolation". Tillgänglig: https://docs.unity3d.com/ScriptReference/Rigidbody-interpolation.html

[41] Dave / GameDevelopment (2021). "Wall Running in Unity - FPS Controller". YouTube playlist. Tillgänglig: https://www.youtube.com/playlist?list=PLbuvmgQy6XRt9_F3LCwi_R5xcXPh4QaE_

[42] Unity Technologies. "Collider.OnCollisionStay". Tillgänglig: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Collider.OnCollisionStay.html

[43] Unity Technologies. "Vector3.up". Tillgänglig: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Vector3-up.html

[44] Photon Engine. "Photon Unity Networking 2". Tillgänglig: https://www.photonengine.com/pun

[45] Brackeys (2018). "Making a Multiplayer FPS in Unity" YouTube playlist. Tillgänglig: https://www.youtube.com/playlist?list=PLPV2KyIb3jR7dFbE2UQYu7QWMdUgDnlnk

[46] Brackeys (2018). "Health System - Making a Multiplayer FPS in Unity". YouTube. (Del av playlist [45])

[47] Dave / GameDevelopment (2023). "Procedural Gun Animation in Unity". YouTube. Tillgänglig: https://www.youtube.com/watch?v=DR4fTllQnXg

[48] BOXOPHOBIC. "Banana Man". Unity Asset Store. Tillgänglig: https://assetstore.unity.com/packages/3d/characters/humanoids/banana-man-196830

[49] Kevin Iglesias. "Human Basic Motions FREE". Unity Asset Store. Tillgänglig: https://assetstore.unity.com/packages/3d/animations/human-basic-motions-free-154271

[50] Unity Technologies. "Animation Rigging Package". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/index.html

[51] Dapper Dino (2020). "Unity IK - Inverse Kinematics Tutorial". YouTube. Tillgänglig: https://www.youtube.com/watch?v=fB0P0C_3sPU

[52] Dapper Dino (2021). "Weapon IK in Unity". YouTube. Tillgänglig: https://www.youtube.com/watch?v=0o7SzYSsR-o

[53] Unity Technologies. "Mathf.Lerp". Tillgänglig: https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html

[54] Unity Technologies. "MonoBehaviour.Update()". Tillgänglig: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

[55] Unity Technologies. "Animation Rigging Package". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/index.html

[56] Unity Technologies. "Two Bone IK Constraint". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/constraints/TwoBoneIKConstraint.html

[57] Unity Technologies. "Multi-Aim Constraint". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/constraints/MultiAimConstraint.html

[58] Unity Technologies. "Rig Component". Tillgänglig: https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/RigBuilder.html