



دانشگاه ملی مهارت

دانشکده ملی مهارت دختران دکتر شریعتی

پایان نامه کار دانی / کارشناسی ناپیوسته / پیوسته

رشته مهندسی کامپیوتر گرایش نرم افزار

روایای نفرین شده : بازی و حشت

نگارش:

شقایق رحمانیه

استاد راهنمای:

دکتر سانا ز افشاری

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

تشکر و قدردانی:

با کمال افتخار، مراتب قدردانی و سپاس خود را از سرکار خانم دکتر ساناز افشاری، استاد راهنمای گران قدرم، ابراز می‌دارم که با راهنمایی‌های ارزشمند و حمایت‌های بی‌وقفه‌شان، این پروژه به سرانجام رسید. همچنین از تمامی دوستان و عزیزانی که در این مسیر با حمایت و همراهی خود یاری ام کردند، صمیمانه تشکر می‌کنم. در پایان، از خانواده‌ام که با عشق و پشتیبانی بی‌حدشان همواره پشتوانه من بودند، قدردانی ویژه‌ای دارم.

چکیده

The Haunting Dream هدف از این پروژه، طراحی و پیاده‌سازی بازی ترسناک روانشناختی با نام است که به عنوان پروژه‌ای برای مقطع کارشناسی در رشته مهندسی نرم‌افزار انجام می‌شود. این بازی به منظور ایجاد تجربه‌ای ترسناک و غوطه‌ورکننده برای بازیکن طراحی شده است که در آن بازیکن باید از میان دنیای تاریک و پر از رمز و راز عبور کند، معماها را حل کرده و با موجودات وحشتناک و چالش‌های ذهنی و فیزیکی روبرو شود. دامنه این پژوهش شامل استفاده از موتور بازی‌سازی Unity برای توسعه بازی و زبان برنامه‌نویسی C# برای طراحی مکانیک‌ها است. همچنین، برای مدل‌سازی سه‌بعدی و انیمیشن‌ها از Blender استفاده شده است.

روش‌شناسی این پروژه شامل طراحی محیط‌های سه‌بعدی و تعاملی با استفاده از ابزارهای مختلف گرافیکی و استفاده از تکنیک‌های پیشرفت‌های در هوش مصنوعی برای طراحی دشمنان است. دشمنان بازی به رفتار بازیکن واکنش نشان می‌دهند و باید به‌طور پویا و هوشمندانه او را تعقیب کنند. علاوه بر این، از سیستم تایم‌لاین‌ها و انیمیشن‌های تعاملی در یونیتی برای ایجاد واقعیت سینماتیک و تقویت جو ترسناک استفاده می‌شود.

یافته‌های این پروژه نشان می‌دهند که ترکیب گرافیک‌های سه‌بعدی، صداگذاری و هوش مصنوعی می‌تواند تجربه‌ای جذاب و دلهره‌آور برای بازیکن فراهم کند. با وجود این، چالش‌هایی در زمینه بهینه‌سازی عملکرد بازی، هماهنگی انیمیشن‌ها و صدایها، و مدیریت منابع گرافیکی با حجم بالا وجود داشت که با تکنیک‌های بهینه‌سازی و استفاده از ابزارهای مختلف حل شدند.

نتیجه‌گیری این پروژه نشان می‌دهد که طراحی بازی‌های روانشناختی و ترسناک با استفاده از تکنیک‌های پیشرفت‌های و پیاده‌سازی دقیق می‌تواند تجربه‌های جذاب و تعاملی به بازیکن ارائه دهد. این پروژه همچنین به عنوان مبنای توسعه ویژگی‌های جدید انتخاب‌های بازیکن و گسترش هوش مصنوعی و محیط‌ها در آینده قابل گسترش است.

کلیدواژه: بازی ترسناک روانشناختی, Unity, C#, Blender, هوش مصنوعی, تایم‌لاین, بهینه‌سازی عملکرد

فهرست نوشتار

۱	فصل ۱: مقدمه
۲	۱-۱- انگلیزه
۳	۱-۲- مروری بر پیشینه و کارهای مشابه
۴	۱-۳- هدف
۵	۱-۴- رئوس مطالب سایر فصل‌ها
۶	فصل ۲: تجزیه و تحلیل نیازمندی‌ها
۷	۲-۱- مقدمه
۸	۲-۱-۱- چرا Unity ؟
۹	۲-۱-۲- کامپوننت‌های مورد نیاز برای پیاده‌سازی
۱۰	۲-۱-۳- روش‌های بهینه‌سازی
۱۱	۲-۱-۴- سناریوی بازی
۱۶	۲-۲- نتیجه‌گیری
۱۸	فصل ۳: ساختار داده‌ها و بانک اطلاعات
۱۹	۳-۱- مقدمه
۱۹	۳-۲- نتیجه‌گیری
۲۰	فصل ۴: پیاده‌سازی
۲۱	۴-۱- مقدمه
۲۱	۴-۲- پیاده‌سازی کامل بازی
۲۱	۴-۲-۱- ایجاد پروژه
۲۲	۴-۲-۲- طراحی منوی اصلی و مقدمه بازی
۲۵	۴-۲-۳- طراحی مدل سه بعدی
۴۴	۴-۲-۴- ایجاد Player اول شخص
۴۸	۴-۲-۵- مدیریت مراحل توسط Game Manager
۵۳	۴-۲-۶- پیاده‌سازی Stage اول
۵۹	۴-۲-۷- پیاده‌سازی Stage دوم
۷۴	۴-۲-۸- پیاده‌سازی Stage سوم
۸۱	۴-۲-۹- پیاده‌سازی Stage چهارم
۸۶	۴-۲-۱۰- پیاده‌سازی Stage پنجم
۱۰۷	۴-۲-۱۱- پیاده‌سازی Stage ششم
۱۲۰	۴-۲-۱۲- پیاده‌سازی Stage هفتم
۱۲۲	۴-۲-۱۳- خروجی گرفتن از بازی

۱۲۴	۴-۳- اجرای نرمافزار
۱۲۸	۴-۳-۱- دانلود پروژه و خروجی بازی
۱۲۸	۴-۴- نتیجه‌گیری
۱۳۰	فصل ۵: جمع‌بندی و پیشنهادها
۱۳۱	۵-۱- نتیجه گیری
۱۳۳	۵-۲- پیشنهادهایی برای کارهای آتی
۱۳۵	منابع

فصل ١:

مقدمة

۱-۱- انگیزه

اهمیت بازی‌های ترسناک روان‌شناختی، مانند The Haunting Dream، به دلیل توانایی در ایجاد تجربه‌های عاطفی و غوطه‌ورکننده، جایگاه ویژه‌ای در صنعت بازی‌سازی دارند. بر اساس گزارش Newzoo^۱، بازار جهانی بازی‌های ویدئویی در سال ۲۰۲۴ به ارزش ۱۸۹ میلیارد دلار رسید که ژانر وحشت به دلیل جذابیت روان‌شناختی، سهمی رو به رشد دارد. این ژانر با تحریک احساسات عمیق مانند ترس و اضطراب، بازیکنان را به چالش می‌کشد و فرصتی برای کاوش در مسائل روانی فراهم می‌کند. ضرورت این پروژه در بررسی تأثیرات روان‌شناختی بازی‌های ترسناک و ارائه تجربه‌ای متمرکز بر اضطراب نهفته است که می‌تواند به تحقیقات آکادمیک در حوزه گیم‌دیزاین کمک کند.

بازی‌هایی مانند Layers of Fear^۲ و Amnesia: The Dark Descent^۳ "نمونه‌های موفقی در ژانر وحشت روان‌شناختی هستند که از اتمسفر تاریک و داستان محوری استفاده می‌کنند. با این حال، مشکلات رایج در این بازی‌ها شامل تکرار مکانیک‌های گیم‌پلی و چالش‌های بهینه‌سازی برای پلتفرم‌های مختلف است. برای مثال، در برخی بازی‌ها، نورپردازی پویا باعث افت عملکرد می‌شود که در Unity^۴ به دلیل مدیریت نادرست منابع رخ می‌دهد. The Haunting Dream با تمرکز بر پازل‌های محیطی متنوع و استفاده از PlayerPrefs^۵ برای ذخیره‌سازی ساده، سعی در رفع این مشکلات دارد.

مسئله اصلی این پروژه، طراحی یک بازی ترسناک روان‌شناختی است که با استفاده از Unity 6 و زبان C#^۶، اتمسفری ترسناک و داستان محور خلق کند. چالش‌ها شامل ایجاد تعادل بین گیم‌پلی تعاملی، مدیریت منابع برای نورپردازی پویا، و طراحی هوش مصنوعی برای موجودات کابوس است که باید رفتارهای پویا و ترسناک داشته باشند. همچنین، ارائه دیالوگ‌های انگلیسی برای افراش حس تعلق، یکی دیگر از جنبه‌های کلیدی این پروژه است.

¹ Newzoo

² Amnesia: The Dark Descent

³ Layers of Fear

⁴ Unity

⁵ PlayerPrefs

⁶ C#

۱-۲- مروری بر پیشینه و کارهای مشابه

بازی‌های ترسناک روان‌شناختی مانند Dead Space^۱، Outlast^۲ و Soma^۳ از مکانیک‌های کاوش، پازل، و مدیریت منابع برای ایجاد ترس استفاده می‌کنند. Outlast بر مکانیک فرار و مخفی‌کاری تمرکز دارد، اما داستان‌سرایی آن گاهی گنگ است. Soma با روایت عمیق روان‌شناختی، نقاط قوت بسیاری دارد، اما پیچیدگی کنترل‌ها برخی بازیکنان چالش‌برانگیز است. The Haunting Dream با ترکیب پازل‌های محیطی ساده (مانند رمز عددی ۳۸۲۵)، مکانیک چراغ‌قوه، و داستان متمرکز بر اضطراب، سعی در ارائه تجربه‌ای متعادل دارد. نقاط ضعف این پروژه شامل محدودیت‌های PlayerPrefs برای ذخیره‌سازی پیچیده و چالش‌های بهینه‌سازی مدل‌های سه‌بعدی طراحی‌شده با Blender^۴ و ProBuilder^۵ است.

۱-۳- هدف

هدف این پایان‌نامه، طراحی و پیاده‌سازی بازی ترسناک روان‌شناختی The Haunting Dream با استفاده از Unity^۶ و زبان C# برای پلتفرم PC است. این پروژه به دنبال خلق تجربه‌ای داستان‌محور با تمرکز بر اضطراب و کابوس‌های شخصیت اصلی است که از طریق مکانیک‌های کاوش، حل پازل (مانند پازل رمز عددی و مکانیک چراغ‌قوه) و نورپردازی پویا اجرا می‌شود. اهداف خاص شامل:

۱. پیاده‌سازی اتمسفر ترسناک با استفاده از نورپردازی متغیر و طراحی صوتی در Unity.
۲. توسعه پازل‌های محیطی تعاملی برای افزایش غوطه‌وری بازیکن.
۳. ایجاد هوش مصنوعی برای موجودات سایه‌مانند با رفتار پویا.
۴. ارائه دیالوگ‌های دو زبانه (فارسی و انگلیسی) برای دسترسی‌پذیری بیشتر.
۵. بهینه‌سازی عملکرد بازی با مدیریت منابع در Unity و استفاده از Blender و ProBuilder برای طراحی مدل‌های سه‌بعدی.

این پروژه به عنوان مطالعه‌ای در گیم‌دیزاین و تأثیرات روان‌شناختی بازی‌های ترسناک، به تحقیقات

¹ Dead Space

² Outlast

³ Soma

⁴ Blender

⁵ ProBuilder

آکادمیک در این حوزه کمک خواهد کرد.

۱-۴- رئوس مطالب سایر فصل‌ها

ساختار پایان‌نامه شامل پنج فصل است که در هر فصل جزئیات مختلفی از پژوهش ارائه می‌شود. در ادامه، رئوس مطالب هر فصل به‌طور مختصر معرفی می‌شوند:

فصل ۱: مقدمه

در فصل اول، انگیزه و هدف اصلی پژوهش بیان شده است. در این بخش به دلایل انجام پژوهش، اهمیت بازی‌های روانشناختی و ترسناک و اهداف کلی پژوهش پرداخته می‌شود. همچنین در این فصل به کارهای پیشین و پژوهش‌های مرتبط با این حوزه اشاره شده است تا زمینه‌سازی برای پژوهش حاضر صورت گیرد.

فصل ۲: تجزیه و تحلیل نیازمندی‌ها

این فصل به تجزیه و تحلیل نیازمندی‌های پژوهش اختصاص دارد. در آن، دلیل انتخاب Unity به عنوان موتور بازی‌سازی و ابزارهای مورد استفاده در Unity^۱، مانند کامپوننت^۱‌ها توضیح داده شده است. همچنین روش‌های بهینه‌سازی عملکرد بازی برای اطمینان از اجرای روان، بهویژه در مواجهه با حجم بالای مدل‌ها و محیط‌های پیچیده، بررسی می‌شود. در این فصل، سناریوی کلی بازی و ویژگی‌های اصلی آن نیز تشریح می‌شود.

فصل ۳: ساختار داده‌ها و بانک اطلاعات

این فصل به ساختار داده‌ها و استفاده از بانک اطلاعاتی در بازی اختصاص دارد. در اینجا، بیشتر به مدیریت داده‌ها با استفاده از PlayerPrefs پرداخته می‌شود. نحوه ذخیره‌سازی وضعیت کلید در این بخش مورد بحث قرار می‌گیرد.

فصل ۴: پیاده‌سازی

در این فصل، مراحل پیاده‌سازی بازی به تفصیل شرح داده شده است. از ابتدا، فرآیند ایجاد پژوهش در یونیتی، طراحی مدل‌ها و محیط‌ها، کدنویسی مکانیک‌های گیم‌پلی، پیاده‌سازی انیمیشن‌ها و تایم‌لاین‌ها تا پیاده‌سازی سیستم‌های هوش مصنوعی توضیح داده می‌شود. روند تکمیل بازی، بهینه‌سازی عملکرد، و خروجی گرفتن برای پلتفرم‌های مختلف مانند PC در این فصل به طور کامل

¹ Component

بررسی می‌شود.

فصل ۵: جمع‌بندی و پیشنهادات

در این فصل، جمع‌بندی کلی پروژه آورده شده و چالش‌ها و مشکلاتی که در طول توسعه بازی پیش آمد، تحلیل می‌شود. همچنین پیشنهاداتی برای بهبود بازی، توسعه ویژگی‌های جدید و گسترش پروژه در آینده ارائه می‌شود. این پیشنهادات شامل افزودن مراحل جدید، بهبود هوش مصنوعی و بهینه‌سازی عملکرد بازی برای پلتفرم‌های مختلف است.

فصل ۲:

تجزیه و تحلیل نیازمندی‌ها

۱-۲- مقدمه

در این بخش، دلایل انتخاب موتور بازی Unity، کامپوننت‌های مورد نیاز برای پیاده‌سازی، روش‌های بهینه سازی و همچنین سناریوی بازی به تفصیل مورد بررسی قرار خواهد گرفت.

۱-۱- چرا Unity؟

به عنوان موتور بازی‌سازی منتخب برای پروژه The Haunting Dream به دلیل ویژگی‌های قدرتمند، انعطاف‌پذیری، و سازگاری با نیازهای پروژه انتخاب شده است. این موتور به‌طور گسترده در صنعت بازی‌سازی برای توسعه بازی‌های دو بعدی و سه بعدی استفاده می‌شود و به ویژه برای پروژه‌های مستقل و آکادمیک مانند این پایان‌نامه کارشناسی مناسب است. دلایل اصلی انتخاب Unity عبارت‌اند از:

- **پشتیبانی از گرافیک‌های پیشرفته:** Unity از سیستم رندر Universal Render Pipeline (URP) پشتیبانی می‌کند که امکان ایجاد نورپردازی پویا و گرافیک‌های بهینه را برای بازی‌های ترسناک روان‌شناختی فراهم می‌کند. این ویژگی برای خلق اتمسفر تاریک و ترسناک پروژه (مانند تغییر نور به قرمز در صحنه‌های خطر) حیاتی است.
- **زبان برنامه‌نویسی C#:** Unity از C# پشتیبانی می‌کند که زبانی شیء‌گرا، قدرتمند، و مناسب برای پیاده‌سازی مکانیک‌های پیچیده بازی مانند پازل‌ها، هوش مصنوعی موجودات کابوس، و مدیریت مراحل است.
- **ابزارهای داخلی و انعطاف‌پذیر:** ابزارهایی مانند ProBuilder برای مدل‌سازی سه‌بعدی مستقیم در Unity و سیستم Timeline برای مدیریت صحنه‌های نمایشی و دیالوگ‌ها، فرآیند توسعه را سریع تر و ساده‌تر می‌کنند.
- **جامعه و منابع گسترده:** Unity دارای جامعه‌ای بزرگ و منابعی مانند Unity Asset Store است که دارایی‌های آماده (مانند مدل‌های سه‌بعدی و افکت‌های صوتی) را ارائه می‌دهد. این ویژگی برای پروژه‌های با زمان محدود مانند پایان‌نامه بسیار مفید است.

- **بهینه‌سازی عملکرد:** Unity ابزارهایی برای بهینه‌سازی عملکرد (مانند مدیریت GameObjects و Prefabs) ارائه می‌دهد که برای اجرای روان بازی با نورپردازی پویا و مدل‌های سه‌بعدی ضروری است.

۲-۱-۲ - کامپوننت های مورد نیاز برای پیاده سازی

شامل مجموعه‌ای از کامپوننت‌ها^۱ و ابزارها برای توسعه بازی‌ها و تجربیات تعاملی می‌باشد.

جدول ۲-۱. جدول Component‌های مهم Unity

ردیف	کامپوننت‌ها	توضیح
۱	Transform	برای تعیین موقعیت، چرخش و مقیاس اشیاء در فضای سه‌بعدی.
۲	Mesh Renderer	برای رندر کردن اشیاء سه‌بعدی با استفاده از مش‌ها.
۳	Collider	برای تعیین نواحی برخورد اشیاء (مانند Sphere Collider، Box Collider و Capsule Collider).
۴	Rigidbody	برای اعمال فیزیک بر روی اشیاء و ایجاد حرکات طبیعی.
۵	Animator	برای کنترل انیمیشن‌های کاراکترها و اشیاء.
۶	Audio Source	برای پخش صداها و موسیقی در بازی.
۷	Camera	برای مشاهده صحنه و تعیین زاویه دید.
۸	Light	برای روشنایی صحنه (مانند Point Light و Directional Light).
۹	Canvas	برای طراحی رابط کاربری (UI) بازی.
۱۰	Event System	برای مدیریت تعاملات کاربر با رابط کاربری.
۱۱	NavMesh Agent	برای ایجاد مسیر یابی هوشمند برای شخصیت‌ها.
۱۲	NavMesh Obstacle	برای مشخص کردن موانع در NavMesh Agent که باید توسط NavMesh‌ها در نظر گرفته شود.
۱۳	NavMesh Surface	ایجاد و مدیریت سطوح ناویگی ^۲ استفاده می‌شود تا کاراکترها بتوانند به طور هوشمندانه در محیط حرکت کنند.
۱۴	Playable Director	این کامپوننت برای کنترل و پخش Timeline استفاده می‌شود. شما می‌توانید آن را به GameObject اضافه کنید و یک Timeline را به آن متصل کنید.
۱۵	Script	برای اضافه کردن منطق بازی و رفتارهای سفارشی به اشیاء.

¹ component

² NavMesh

۲-۱-۳ - روش‌های بهینه‌سازی

□ بهینه‌سازی اجرا

برای افزایش عملکرد بازی و کاهش بار پردازشی بهویژه در دستگاه‌های ضعیفتر، مجموعه‌ای از اقدامات بهینه‌سازی در پروژه انجام شد:

در ابتدای پروژه، موتور گرافیکی به Universal Render Pipeline (URP) تغییر داده شد. URP یک رندر سبک و بهینه‌شده است که مخصوص دستگاه‌های مختلف طراحی شده و از قابلیت‌هایی مثل Scriptable Rendererer، تنظیمات انعطاف‌پذیر نورپردازی و Shader‌های ساده‌تر استفاده می‌کند. این کار باعث کاهش مصرف منابع GPU و CPU و افزایش فریم‌ریت بازی شد.

از Occlusion Culling برای جلوگیری از رندر شدن آبجکت‌هایی که توسط اجسام دیگر مسدود شده‌اند استفاده شد. این تکنیک به موتور Unity کمک می‌کند تا فقط آن دسته از اشیاء را که در میدان دید دوربین قرار دارند، پردازش کند. این روش مخصوصاً در محیط‌های بسته و صحنه‌هایی با آبجکت‌های زیاد بسیار مؤثر است.

برای کاهش مصرف حافظه گرافیکی و سرعت بارگذاری، رزولوشن تکسچرها به صورت بهینه کاهش داده شد. در عین حفظ کیفیت بصری، با استفاده از نسخه‌های پایین‌تر تکسچرها مصرف منابع به شدت کاهش یافت.

□ بهینه‌سازی نورپردازی

برای کاهش محاسبات بلادرنگ، تمامی منابع نور در محیط به حالت Baked تغییر یافتند. در این حالت، اطلاعات نور و سایه به صورت ایستاده در Lightmap‌ها ذخیره شده و نیازی به پردازش زمان اجرا نیست. این باعث افزایش سرعت و کاهش مصرف منابع می‌شود.

همچنین برای بهینه‌تر کردن بازتاب نورها از Reflection Probe استفاده شد. این پروب‌ها نور محیط را ذخیره می‌کنند و برای اشیاء براق بدون نیاز به بازتاب زنده جلوه‌ی نوری مناسبی ایجاد می‌کنند. پروب‌ها به صورت Baked پیکربندی شدند تا بر عملکرد تاثیر منفی نگذارند.

۴-۱-۲ - سناریوی بازی

▣ مقدمه بازی

بازی ترسناک سه‌بعدی طراحی شده در موتور یونیتی، تجربه‌ای روان‌شناختی و داستان محور ارائه می‌دهد که بر اضطراب و کابوس‌های شخصیت اصلی تمرکز دارد. این بازی با بهره‌گیری از اتمسفر تاریک و مکانیک‌های تعاملی، بازیکن را در فضایی پرتنش غرق می‌کند. داستان با یک مقدمه^۱ آغاز می‌شود که شخصیت اصلی، که از اضطراب شدید رنج می‌برد، به دنبال یک خواب آرام است.

دیالوگ مقدمه (فارسی و انگلیسی):

فارسی: «مدتی است که خواب خوبی نداشته‌ام... اضطرابم شدیدتر شده... شاید یک خواب خوب بتواند به من کمک کند.»

انگلیسی:

"I haven't been sleeping well lately... My anxiety has worsened... Perhaps one good night's sleep could help".

توضیح: این دیالوگ، زمینه روان‌شناختی شخصیت را معرفی می‌کند و بازیکن را برای تجربه‌ای متمرکز بر اضطراب و کابوس آماده می‌سازد. مقدمه با نمایش یادداشتی از مادر شخصیت اصلی ادامه می‌یابد که روی میز قرار دارد.

دیالوگ یادداشت مادر (فارسی و انگلیسی):

فارسی: «غذایت را بخور و زود بخواب، امشب شیفت شب دارم و به خانه نمی‌آیم.»

انگلیسی:

"Eat your dinner and go to bed early. I'm working the night shift tonight and won't be home".

توضیح: یادداشت مادر، حس تنها‌یی شخصیت را تقویت می‌کند و زمینه‌ساز انتقال بازیکن به مرحله خواب است. پس از رفتن به تخت، شخصیت احساس عجیبی را تجربه می‌کند که مرز بین واقعیت و خواب را محو می‌کند.

دیالوگ شخصیت در خواب (فارسی و انگلیسی):

فارسی: «چرا احساس می‌کنم خوابم نبرده؟ یا شاید خوابیده‌ام؟ این حس... خیلی عجیب است.»

انگلیسی:

¹ Intro

"Why does it feel like I'm not falling asleep? Or... am I already asleep? This feels... strange. Really strange".

توضیح: این بخش، اتمسفر مرموز بازی را تقویت می‌کند و بازیکن را به دنیای کابوس‌مانند شخصیت هدایت می‌کند.

□ گیمپلی و مکانیک‌های اولیه

بازیکن از تخت بیدار می‌شود و متوجه می‌شود که در اتاق قفل شده است. با جستجو در اتاق، کلیدی در کشوی میز کنار تخت پیدا می‌کند و با استفاده از آن، در را باز کرده و وارد راهروی خانه می‌شود. در این مرحله، صدایی از در خروجی خانه شنیده می‌شود که بازیکن را به سمت آن می‌کشاند. با نزدیک شدن به در، صدای زنی (مادر شخصیت) شنیده می‌شود که هشداری اضطراری می‌دهد. دیالوگ هشدار مادر (فارسی و انگلیسی):

فارسی: «فرار کن... عزیزم، اینجا خطرناک است! لطفاً، سریع‌تر این خانه را ترک کن!»
انگلیسی:

"Run... run, baby... it's so dangerous here! Please, just get out of this house... now"!

توضیح: این دیالوگ، حس فوریت و خطر را به بازیکن منتقل می‌کند و او را به ادامه کاوش در خانه ترغیب می‌کند. بازیکن متوجه می‌شود که در خروجی قفل است و باید خانه را برای یافتن راه فرار جستجو کند.

□ پازل رمز عددی

در ادامه، بازیکن با دری قفل شده مواجه می‌شود که نیاز به رمز عددی دارد. سرنخ‌های رمز در محیط خانه پراکنده شده‌اند. تابلویی در انتهای راهرو با چهار گل به رنگ‌های زرد، سبز، آبی، و قرمز دیده می‌شود. با کاوش در خانه، بازیکن اشیایی را پیدا می‌کند که تعداد آن‌ها به ترتیب زیر است:

۳ قوطی نوشابه زرد روی میز پذیرایی

۸ برگ سبز در گلدان

۲ مسوак آبی در دستشویی

۵ سیب قرمز در آشپزخانه

رمز عددی ۳۸۲۵ از ترکیب تعداد این اشیا به دست می‌آید.



معماه رمز عددی

توضیح: این پازل، بازیکن را به کاوش دقیق در محیط و توجه به جزئیات تشویق می‌کند. پس از وارد کردن رمز، نورهای پذیرایی به رنگ قرمز تغییر می‌کند و یک شخصیت ترسناک (زن) ظاهر می‌شود. با نزدیک شدن بازیکن، صدایی ناگهانی پخش شده، زن غیبیش می‌زند و نورها به حالت عادی بازمی‌گردند.

▣ مواجهه با عروسک تدی

بازیکن وارد اتاقی می‌شود که دری قفل شده دیگر دارد. روی دیوار نوشته‌ای به چشم می‌خورد: نوشته روی دیوار (فارسی و انگلیسی):
فارسی: «تدى می‌خواهد با تو بازی کند.»
انگلیسی:

" Teddy wants to play with you".

با باز کردن کمد، بازیکن عروسک خرسی ترسناکی پیدا می‌کند. با برداشتن عروسک، صفحه سیاه شده و دیالوگی پخش می‌شود:
دیالوگ عروسک تدی (فارسی و انگلیسی):

فارسی: «سلام... می‌خواهی با من بازی کنی؟ قوانین خیلی ساده است. بیا، بهت نشان می‌دهم. نترس.»

انگلیسی:

"Hello... Do you want to play with me? The rules are very simple. Come over here, and I'll show you. Don't be afraid".

توضیح: این بخش، بازیکن را به مرحله‌ای جدید و چالش‌برانگیز هدایت می‌کند که بر ترس و تعلیق متمرکز است.

▣ مرحله چراغ‌قوه و موجودات کابوس

بازیکن در اتاقی کوچک بیدار می‌شود که روی دیوار نوشته‌ای دیده می‌شود: نوشته روی دیوار (فارسی و انگلیسی):

فارسی: «او از نور می‌ترسد... فقط بدرخش.»

انگلیسی:

" It fears the light... just shine".

چراغ‌قوه‌ای روی زمین قرار دارد که بازیکن با برداشتن آن، در را باز کرده و وارد فضایی تاریک پر از موجود کابوس می‌شود. موجود به بازیکن نزدیک می‌شود و اگر نور چراغ‌قوه به آن برخورد کند، متوقف می‌شود؛ در غیر این صورت، به بازیکن حمله کرده و مرحله از ابتدا تکرار می‌شود.

توضیح: این مکانیک، استفاده استراتژیک از نور را به عنوان ابزاری برای بقا معرفی می‌کند و حس ترس و فشار زمانی را تقویت می‌کند.

▣ بازگشت به اتاق و انباری

پس از موفقیت در مرحله چراغ‌قوه، بازیکن به اتاقی کوچک می‌رسد و عروسک تدی را دوباره می‌بیند. با برداشتن عروسک، به اتاق خواب اولیه بازمی‌گردد و در قفل شده باز می‌شود. بازیکن وارد انباری باریکی می‌شود که میز، صندلی، و چند کارتون در آن قرار دارد. روی میز، کاغذی با این نوشته دیده می‌شود:

نوشته روی کاغذ (فارسی و انگلیسی):

فارسی: «متأسفم، اما اینجا کلیدی نیست. هیچ راه فراری نداری.»

انگلیسی:

"Sorry, but there is no key here. You have no way to escape".

توضیح: این پیام، حس نالمیدی و محبوس بودن را به بازیکن القا می‌کند و او را به ادامه داستان هدایت می‌کند.

□ انتقال به اتاق روان‌درمانگر

با باز کردن در انباری، بازیکن ناگهان خود را در اتاق روان‌درمانگر می‌یابد و با خود می‌گوید: دیالوگ شخصیت (فارسی و انگلیسی):

فارسی: «اتاق درمان؟ اینجا چه کار می‌کنم؟ آه... مدتی است که به روان‌درمانگر سر نزده‌ام.»
انگلیسی:

" The therapy room? What am I doing here? Oh... I haven't been to my therapist lately".

با کاوش در اتاق، بازیکن پرونده‌ای پزشکی پیدا می‌کند:
متن پرونده (فارسی و انگلیسی):

فارسی:

پرونده بیمار

نام: ...

شکایت اصلی: اضطراب و کابوس
شدت اضطراب: شدید

ماهیت کابوس‌ها: تعقیب شدن، مرگ، خفگی
انگلیسی:

PATIENT FILE

NAME... :

CHIEF COMPLAINT: Anxiety and nightmares

Severity of Anxiety: Severe

Nature of Nightmares: Being chased, death, suffocation

توضیح: این پرونده، عمق مشکلات روانی شخصیت را آشکار می‌کند و به بازیکن سرنخ‌هایی درباره ارتباط داستان با اضطراب می‌دهد.

□ راهروهای پایانی و پایان‌بندی

بازیکن وارد راهرویی با نور قرمز و تابلوهای ترسناک می‌شود که نمادهای اضطراب را به صورت

وحشتناک به تصویر می‌کشند. صدای ترسناکی اتمسفر را پر می‌کند. با عبور از راهرو، بازیکن وارد راهرویی با نور عادی و خالی می‌شود و دوباره صدای مادر را می‌شنود: دیالوگ مادر (فارسی و انگلیسی):

فارسی: «بیا اینجا، عزیزم. من اینجایم. نگران نباش، همیشه کنار توام.»
انگلیسی:

"Come here, darling. I'm right here. Don't worry, I'm here for you. I'll never leave you alone".

با نزدیک شدن به زنی که در نور قرمز ایستاده، او ناگهان جیغ کشیده و به سمت بازیکن می‌آید.
صفحه سیاه شده و صدای درونی شخصیت پخش می‌شود: دیالوگ درونی (فارسی و انگلیسی):

فارسی: «این واقعی نیست... فقط یک رویاست... لطفاً... بیدار شو... بیدار شو!»
انگلیسی:

"This is not real... It's just a dream... Please... Wake up... Wake up"!

شخصیت از خواب بیدار می‌شود و خود را در تخت می‌یابد. صدای مادر از پشت در شنیده می‌شود:
دیالوگ مادر (فارسی و انگلیسی):
فارسی: «عزیزم، من آمدم! وقت بیدار شدن است. امروز جلسه درمانی داری.»
انگلیسی:

" Honey, I'm home! It's time to wake up! You have your therapy session today".

توضیح: این بخش، حس آرامش کاذب را به بازیکن القا می‌کند. اما ناگهان، صدایی ترسناک پخش شده، اتفاق تاریک می‌شود و همان زن ترسناک در نور قرمز جلوی تخت ظاهر می‌شود. بازی در این نقطه به پایان می‌رسد و بازیکن را با سؤالاتی درباره واقعیت و کابوس رها می‌کند.

۲-۲ - نتیجه‌گیری

این بازی با استفاده از مکانیک‌های کاوش، حل پازل، و مدیریت منابع (چراغ‌قوه)، تجربه‌ای تعاملی و ترسناک ایجاد می‌کند. طراحی اتمسفر با نورپردازی متغیر (قرمز برای خطر، تاریکی برای ترس) و استفاده از صدای محیطی، حس اضطراب را تقویت می‌کند. تم اصلی بازی، کاوش در اضطراب و

کابوس‌های روان‌شناختی است که از طریق داستان و گیم‌پلی منتقل می‌شود. پیاده‌سازی در یونیتی با استفاده از مدل‌های سه‌بعدی، سیستم نورپردازی پویا، و اسکریپتنویسی برای تعاملات، امکان خلق این تجربه را فراهم کرده است. همچنین اجرای بازی به صورت بھینه بوده.

فصل ۳:

ساختار داده‌ها و بانک اطلاعات

۱-۳- مقدمه

در پروژه The Haunting Dream، به دلیل ماهیت تکنفره و حجم محدود داده‌ها، از Unity برای ذخیره‌سازی داده‌های بازی مانند پیشرفت مراحل مانند برداشتن کلید و باز کردن در استفاده شد. PlayerPrefs با ذخیره داده‌ها به صورت جفت‌های Key-Value Pairs در فایل‌های محلی، سادگی و سرعت پیاده‌سازی را فراهم کرد و نیاز به بانک اطلاعاتی سنتی مانند MySQL یا SQLite را حذف نمود، زیرا بازی نیازی به مدیریت داده‌های چندکاربره یا پیچیده نداشت.

در اسکریپت‌های C#، از ساختارهای داده‌ای مانند List برای مدیریت یکسری Object‌ها مانند نورها، مدیریت Stage‌ها برای مراحل بازی استفاده شد. این رویکرد، ضمن حفظ سادگی برای یک پروژه آکادمیک، امکان تمرکز بر گیم‌پلی روان‌شناختی و اتمسفر ترسناک را فراهم کرد.

۲-۳- نتیجه‌گیری

در The Haunting Dream، به دلیل حجم کم داده‌ها و ماهیت تکنفره، از PlayerPrefs در Unity برای ذخیره ساده پیشرفت مراحل (مانند باز کردن درها) و از List در C# برای مدیریت اشیا و مراحل استفاده شد، که نیاز به بانک اطلاعاتی پیچیده را حذف کرد و گیم‌پلی روان‌شناختی را پشتیبانی نمود.

^۱ کلید-مقدار

فصل ۴:

پیاده‌سازی

۴-۱- مقدمه

فصل پیاده‌سازی پروژه The Haunting Dream جزئیات فی شامل ابزارها، نرم‌افزارها، چارچوب‌ها، زبان‌ها و تنظیمات مورد استفاده را شرح می‌دهد. از موتور بازی‌سازی Unity^۱ برای توسعه بازی، زبان C# برای برنامه‌نویسی^۲، Visual Studio Code برای کدنویسی^۳، و Blender برای Universal Render Pipeline (URP) مدل‌سازی سه‌بعدی استفاده شد. چارچوب Universal Render Pipeline (URP) برای طراحی محیط سه‌بعدی، و PlayerPrefs برای ذخیره‌سازی داده‌ها به کار رفتند. تنظیمات شامل Occlusion Culling، نورهای Baked و Reflection Probe برای بهبود عملکرد و اتمسفر ترسناک بازی بود.

۴-۲- پیاده‌سازی کامل بازی

پیاده‌سازی کامل بازی شامل طراحی، کدنویسی و .. می‌باشد که در ادامه توضیح خواهد داد.

۴-۲-۱- ایجاد پروژه

در ابتدای روند توسعه بازی تصمیم گرفتم از Universal Render Pipeline (URP) به عنوان سیستم رندرینگ استفاده کنم. دلیل اصلی این تصمیم، نیاز به بهینه‌سازی گرافیکی بازی برای اجرا روی سیستم‌های میان‌رده و همچنین بهره‌مندی از قابلیت‌های نوری بهتر، مانند نورپردازی واقع‌گرایانه، کنترل بیشتر بر متریال‌ها و بهبود عملکرد در محیط‌های تاریک بود؛ ویژگی‌هایی که برای یک بازی ترسناک روان‌شناختی بسیار مهم هستند.

چرایی استفاده از URP :

- افزایش عملکرد (Performance) برای سخت‌افزارهای ضعیف‌تر

¹ <https://unity.com>

² <https://code.visualstudio.com>

³ <https://www.blender.org>

- پشتیبانی بهتر از افکت‌های نوری و سایه‌های دینامیک
- امکان کنترل بهتر روی متریال‌ها، نورها
- سازگاری بالا با پلتفرم‌های مختلف

مراحل تبدیل پروژه D3 به URP :

۱. از طریق منوی بالا وارد مسیر زیر شدم:

Window > Package Manager

۲. در پنجره Package Manager، گزینه‌ی Unity Registry را انتخاب کردم تا لیست کامل پکیج‌ها نمایش داده شود.

۳. از میان پکیج‌ها، Universal RP را جستجو کرده و سپس آن را نصب کردم.

۴. پس از نصب، وارد مسیر زیر شدم:

Assets > Create > Rendering > URP Asset (Forward Renderer)

و یک فایل جدید URP Asset ایجاد کردم.

۵. سپس وارد مسیر زیر در پنجره Project Settings شدم:

Edit > Project Settings > Graphics

و فایل URP Asset ایجاد شده را به عنوان Render Pipeline Asset انتخاب کردم.

۶. برای اطمینان از سازگاری متریال‌ها با URP، از ابزار داخلی Unity استفاده کردم:

Edit > Render Pipeline > Universal Render Pipeline > Upgrade Project Materials to URP

این مرحله باعث شد متریال‌ها با سیستم رندرینگ URP هماهنگ شوند و دیگر نیازی به بازسازی دستی آن‌ها نباشد.

۴-۲-۴- طراحی منوی اصلی و مقدمه بازی

□ ساخت منوی بازی

در پروژه بازی، برای ساخت منوی اصلی، مراحل مختلفی را طی کردم که شامل ایجاد Canvas، افزودن دکمه‌ها، تنظیمات آن‌ها و اضافه کردن اسکریپت‌های C# برای کنترل منو می‌شود. در این گزارش، مراحل انجام‌شده به صورت گام‌به‌گام توضیح داده می‌شود.

اولین قدم برای ایجاد منوی اصلی، ایجاد یک Canvas در پنل Hierarchy است. این کار از منوی Canvas GameObject > UI > Canvas انجام شد. به عنوان محیطی برای قرار دادن تمامی المنتهای UI (مانند دکمه‌ها و متن‌ها) عمل می‌کند. این Canvas محلی است که تمام اجزای منو در آن قرار می‌گیرند و برای طراحی منوی اصلی به کار می‌رود.

در گام بعدی، دکمه‌های مختلف منو را اضافه کردم. برای این کار از منوی GameObject > UI > Button استفاده شد. این دکمه‌ها در Canvas قرار گرفتند و مکان آن‌ها به دلخواه تنظیم شد تا منوی ساده و قابل فهم ایجاد گردد. برای هر دکمه، متن آن‌ها به‌طور سفارشی تغییر داده شد تا هر دکمه وظیفه خود را به وضوح نشان دهد.

برای اینکه دکمه‌ها عملکردهای خاص خود را داشته باشند، یک اسکریپت C# به نام MainMenu ایجاد شد. این اسکریپت شامل دو متده است که به دکمه‌ها مربوط می‌شود:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;

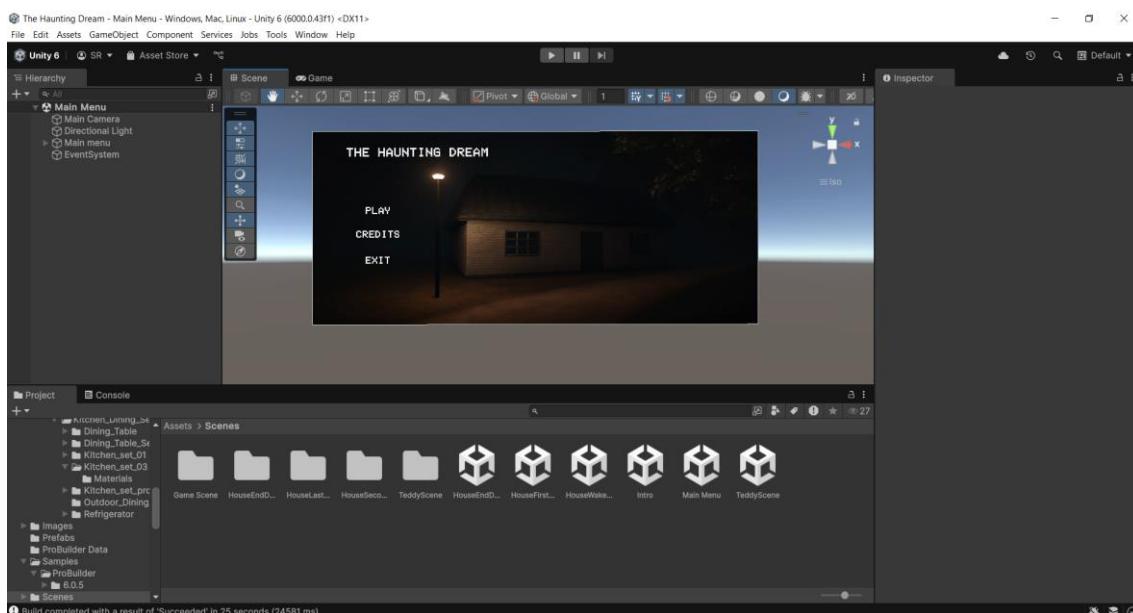
public class MainMenu : MonoBehaviour
{
    void Start()
    {
        Cursor.lockState = CursorLockMode.None; // Free cursor
        Cursor.visible = true; // show cursor
    }

    public void Play()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex+1);
    }

    public void Quit()
    {
        Application.Quit();
        Debug.Log("Player Has Quit The Game");
    }
}
```

کد بالا یک اسکریپت ساده برای مدیریت منوی اصلی در یونیتی است که در کلاس MainMenu پیاده‌سازی شده و از کتابخانه‌های اصلی یونیتی مانند UnityEngine و UnityEngine.SceneManagement استفاده می‌کند. در متده Start، اشاره‌گر ماوس از حالت قفل خارج شده (Cursor.lockState = CursorLockMode.None) و قابل دیدن می‌شود

(Cursor.visible = true) تا بازیکن بتولند آزادلنه در منو حرکت کند. متدهای Play و SceneManager.LoadScene را بازی را بارگذاری صحنه‌ی بعدی در لیست بیلد (با استفاده از Quit) شروع کند. متدهای Application.Quit و Application.Quit (برنامه می‌شود)، اما در محیط ادیتور تنها یک پیام در کنسول نمایش داده می‌شود (Debug.Log). این اسکریپت باید به یکی از آبجکت‌های موجود در صحنه‌ی منوی اصلی نسبت داده شود و دکمه‌های "شروع بازی" و "خروج" از طریق سیستم UI یونیتی به توابع Play و Quit متصل شوند.



تصویر منوی اصلی بازی

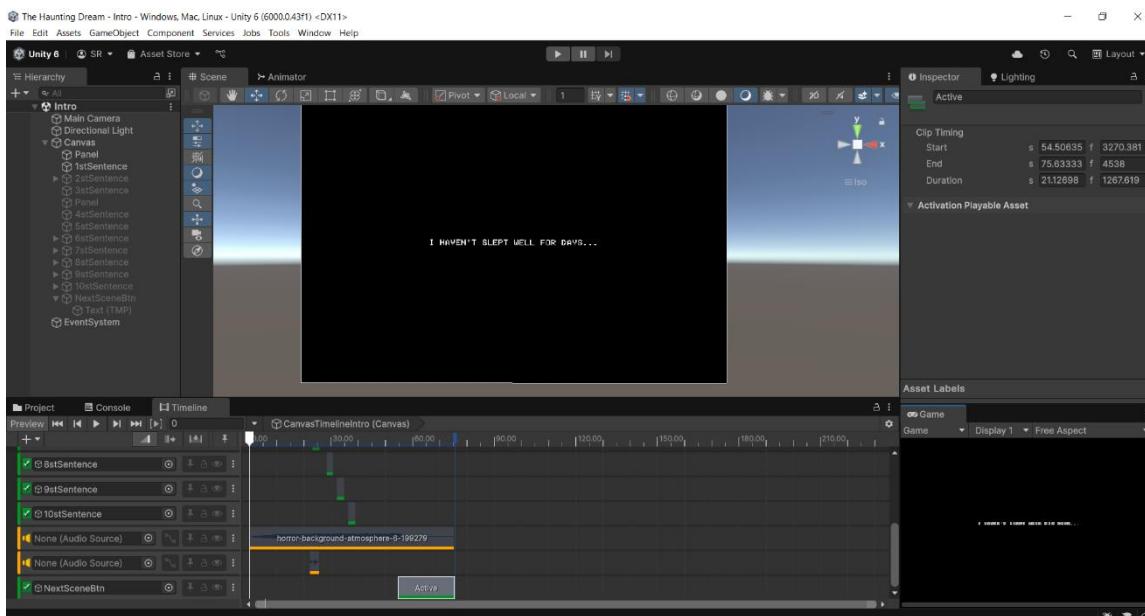
برای زیباتر شدن منوی اصلی، تنظیمات مختلفی برای Canvas اعمال شد. همچنین تصویر زمینه از نمای بیرونی مدل خانه الهام گرفته شده است.



□ ساخت مقدمه یا Intro

یک Scene جدید به نام Intro برای بازی ساختم که پس از فشردن دکمه Play در منوی اصلی اجرا می‌شود. این مقدمه شامل دیالوگ، تصاویر و صدای پس‌زمینه است و فضای ترسناک بازی را معرفی می‌کند.

یک GameObject با عنوان Playable Director می‌ساختم و Timeline جدید از CanvasTimelineIntro مربوط به مقدمه TimeLine است. `TimeLine > Timeline > CanvasTimelineIntro`



استفاده از Timeline برای ساخت Intro

وقتی از Timeline در Activation Track استفاده می‌کنید، این امکان را دارید که به‌طور دقیق کنترل کنید که چه زمانی یک GameObject فعال یا غیرفعال شود. Activation Track به شما اجازه می‌دهد تا با استفاده از Keyframe‌ها، وضعیت فعال بودن یا غیرفعال بودن یک شیء را در طول زمان تغییر دهید.

۴-۲-۳- طراحی مدل سه بعدی

□ طراحی مدل سه بعدی خانه با استفاده از ProBuilder

برای طراحی محیط اصلی بازی، تصمیم گرفتم از ابزار ProBuilder استفاده کنم. این ابزار یکی از پکیج‌های رسمی Unity است که امکان ساخت و طراحی مدل‌های سه بعدی ساده و قابل ویرایش را

مستقیم در محیط Unity فراهم می‌کند. با توجه به هدفم که ساخت یک خانه کابوس‌وار با ساختاری ساده اما کاربردی بود، ProBuilder انتخاب مناسبی بود.

ایجاد خانه با استفاده از ابزار Geometry در ProBuilder

پس از نصب ProBuilder، وارد محیط کاری آن شدم. برای طراحی فضای خانه، بیشتر از منوی استفاده کردم که ابزارهای اصلی ساخت مدل‌های پایه را ارائه می‌دهد.

ابزارهای کلیدی در منوی Geometry و کاربرد آن‌ها

ردیف	نام ابزار	کاربرد اصلی	توضیحات تکمیلی
۱	New Shape	ساخت اشکال پایه مانند Cylinder، Plane، Cube و ...	طراحی سریع دیوار، سقف، کف و درها با اشکال پایه
۲	Vertex/Edge/Face Selection	انتخاب و ویرایش رأس‌ها، لبه‌ها و سطوح مدل	کنترل دقیق بر شکل مدل، ایجاد جزئیات، برش یا تغییر شکل
۳	Extrude Face / Edge	بیرون کشیدن سطح یا لبه برای ساخت دیوار یا پرسنگی	ایجاد حجم از یک سطح یا لبه برای افزودن ساختار به مدل
۴	Inset Face	ایجاد یک ناحیه فرو رفته درون یک سطح	شبیه‌سازی قاب پنجره، در یا جزئیات سطح
۵	Bridge Edges	ایجاد یک سطح بین دو لبه انتخاب شده	اتصال دو لبه برای ساخت درگاه، تونل یا فضاهای پیوسته
۶	Delete Faces	حذف سطوح انتخاب شده	برای ایجاد سوراخ، فضاهای باز یا پاک‌سازی سطوح نامطلوب
۷	Fill Hole	پر کردن حفره‌های باز در حالت ویرایش رأس یا لبه	ساخت سریع سطوح بسته در نواحی باز مدل
۸	Insert Edge Loop	اضافه کردن یک حلقه لبه جدید از لبه‌های انتخاب شده	افزایش کنترل و جزئیات روی سطوح مدل
۹	Merge Faces	ترکیب چند سطح انتخابی به یک سطح یکپارچه	کاهش تعداد سطوح و ساده‌سازی مدل

استفاده از نقشه دو بعدی خانه برای ساخت مدل سه بعدی در Unity با ProBuilder



در این مرحله ابتدا نقشه خانه را که به صورت دو بعدی طراحی شده بود، وارد پروژه Unity کردم. برای استفاده از این نقشه به عنوان مرجع ساخت خانه در فضای سه بعدی، ابتدا یک متریال جدید ایجاد کردم. سپس تنظیمات زیر را روی متریال اعمال نمودم:

- Shader متریال را به حالت Unlit/Texture تغییر دادم. این کار باعث می‌شود تصویر بدون تأثیر نورپردازی در صحنه نمایش داده شود و با وضوح کامل دیده شود.
- تصویر پلان را به عنوان Texture اصلی این متریال تنظیم کردم.

سپس یک Plane ساده در صحنه ایجاد کردم و این متریال را روی آن انداختم. نقش سطح مرجع برای ساخت خانه را ایفا می‌کرد. اندازه Plane را به گونه‌ای تنظیم کردم که نقشه به صورت نسبتاً دقیق (بر اساس مقیاس‌های متر) در آن جای بگیرد.

ساخت خانه با استفاده از ProBuilder و تبدیل به Prefab

مرحله ۱: مدل‌سازی اجزای خانه با ProBuilder
در این مرحله با استفاده از ابزار ProBuilder اقدام به ساخت اجزای اصلی خانه از جمله موارد زیر کردم:

- دیوارها (Walls)
- درها (Doors)
- پنجره‌ها (Windows)

- دیوارهای گوشه‌ای (Corner Walls)

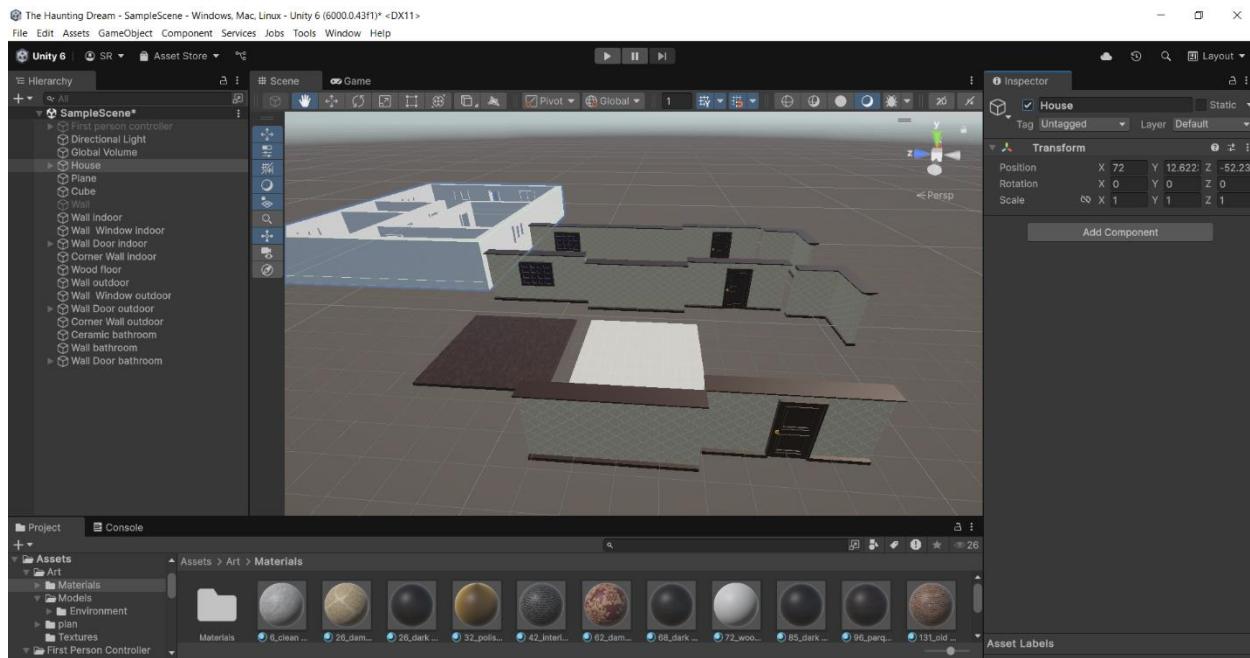
- کف زمین (Floors)

برای شروع، از گزینه‌ی ProBuilder > New Shape استفاده کردم و با انتخاب اشکال مکعبی (Cube) و تنظیم مقادیر مورد نیاز، دیوارها و کف را ایجاد نمودم. سپس با استفاده از ابزار Extrude Face برای افزایش ارتفاع دیوارها و تغییر شکل آن‌ها اقدام کردم.

مرحله ۲: طراحی و شکل‌دهی دقیق با ابزارهای ویرایشی ProBuilder پس از ساخت اولیه اجزاء، با استفاده از ابزارهای ویرایشی ProBuilder اقدام به طراحی دقیق‌تر و جزئی‌سازی مدل‌ها کردم.

مرحله ۳: اعمال متریال و بافت‌گذاری (UV Mapping)

برای واقعی‌تر شدن مدل، ابتدا یک یا چند Material Shader جدید با نوع Unlit/Texture با استفاده از بخش UV Editor در ProBuilder اقدام به تنظیم نقشه‌ی UV برای هر قطعه نمودم تا بافت‌ها به درستی و بدون کشیدگی نمایش داده شوند. همچنین با توجه به کاربرد هر بخش مثلاً دیوار یا کف، متریال‌های مجزا با رنگ یا بافت مناسب اختصاص داده شد.



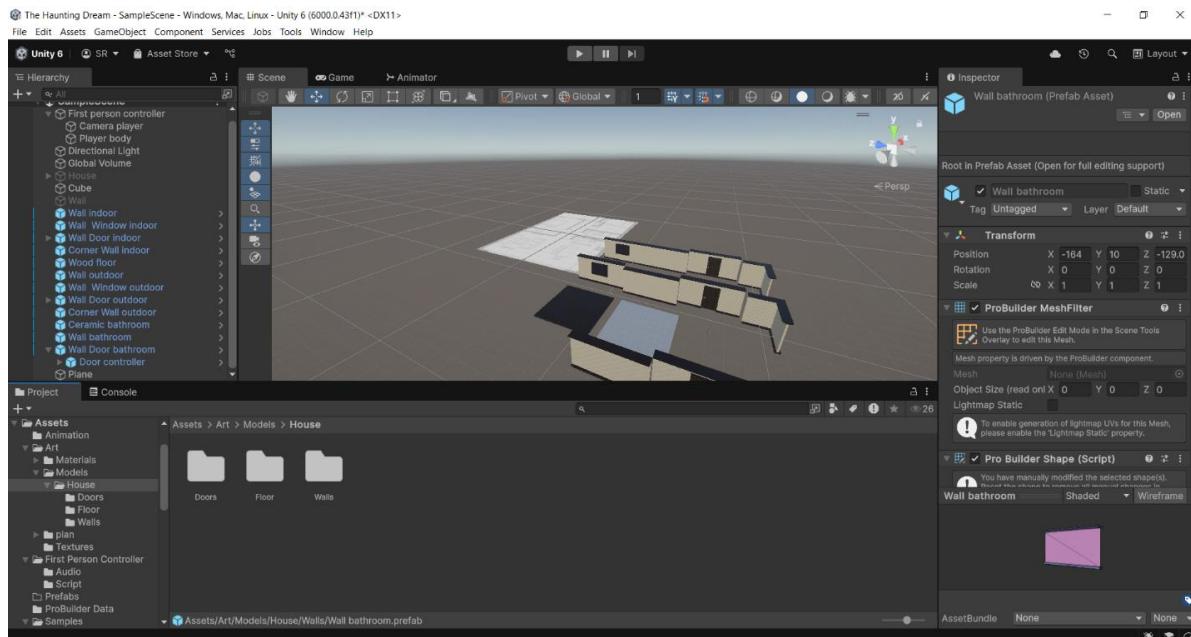
ساخت اجزای مدل خانه

مرحله ۴: تبدیل به Prefab

برای افزایش بهره‌وری در ساخت خانه و استفاده مجدد از اجزا، هر کدام از بخش‌های ساخته شده (مثلاً یک قطعه دیوار یا پنجره آماده) را به یک Prefab تبدیل کردم:

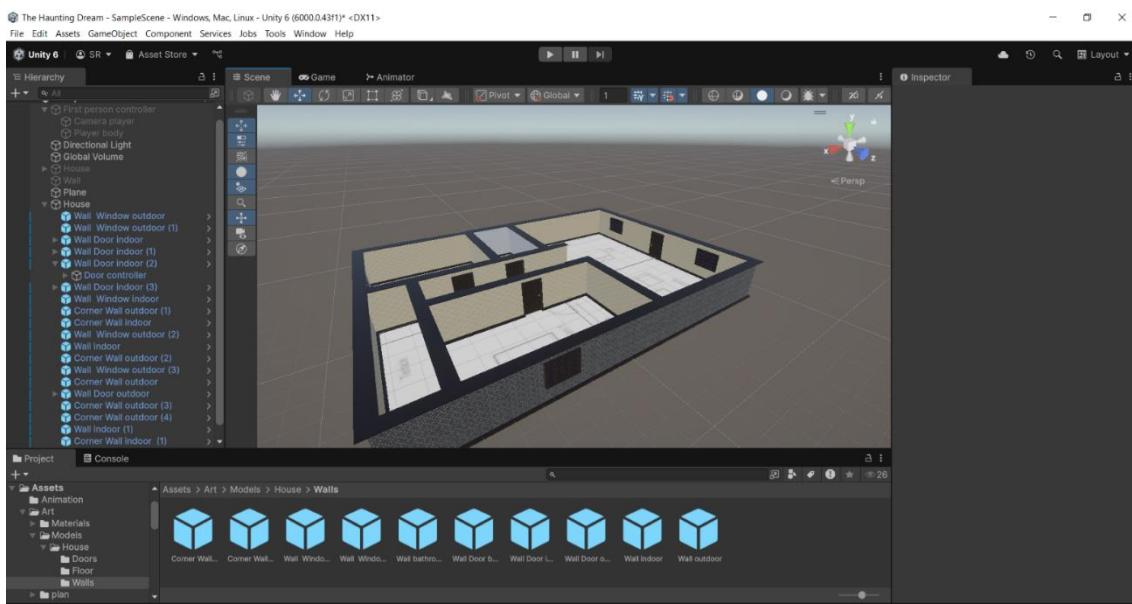
۱. کشیدن آبجکت از Hierarchy در Prefabs به پوشه‌ی

۲. اختصاص نام مناسب به هر Prefab

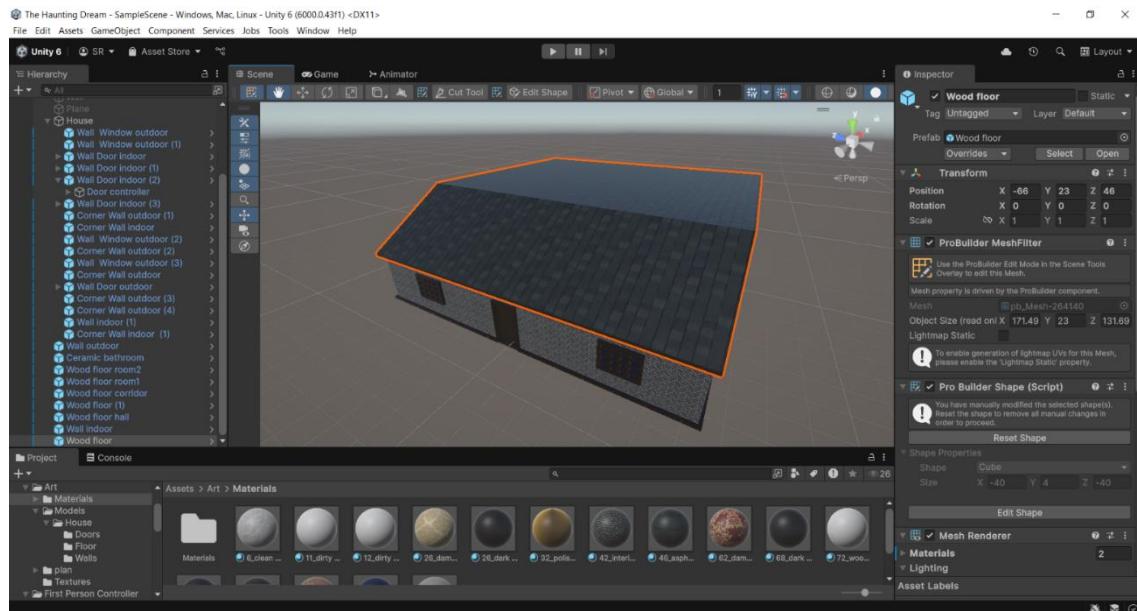


تبديل مدل اجزای خانه به prefabs

مرحله ۵: ساخت خانه نهایی با استفاده از Prefab ها
در مرحله نهایی، با استفاده از نقشه‌ی خانه‌ای که قبلاً به عنوان مرجع تصویری وارد کرد بودم، شروع به چیدن Prefab ها در صحنه کردم. با کمک ابزارهای ProBuilder خانه‌ی اصلی را ساختاردهی کردم. هر Prefab در محل مناسب خود قرار داده شد و با استفاده از تنظیمات دقیق، خانه به صورت سه‌بعدی طبق نقشه طراحی شد.



ساخت مدل خانه با prefabs



مدل کلی خانه

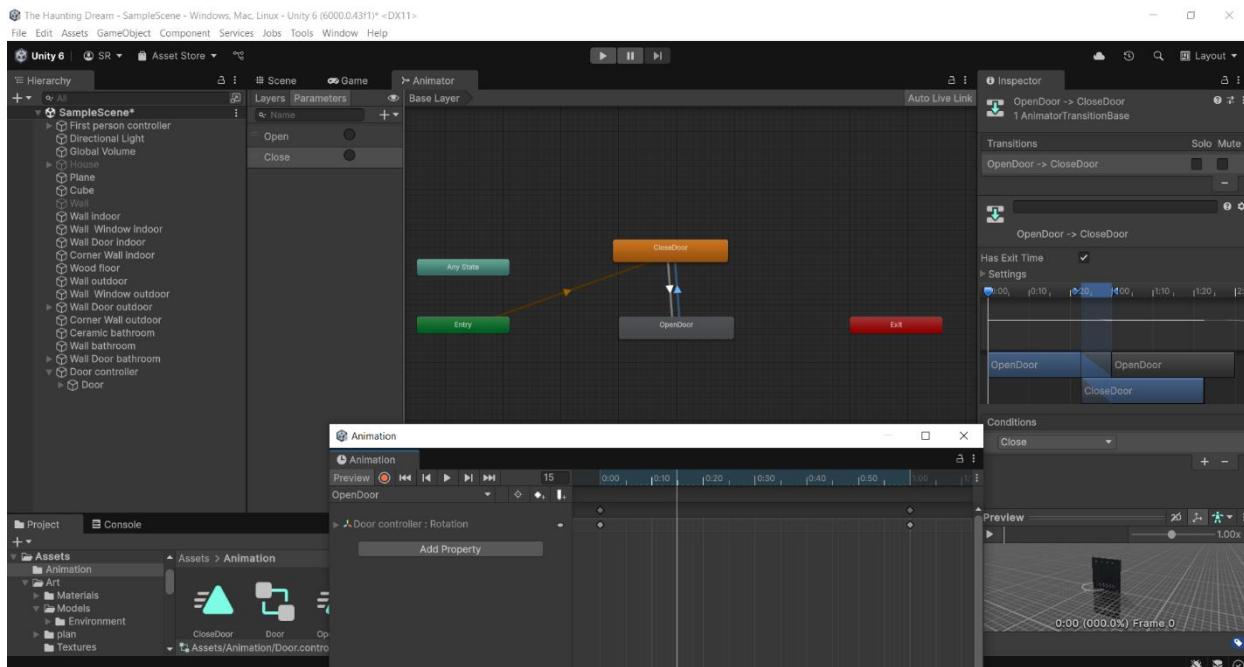
□ ساخت سیستم باز و بسته شدن در به همراه انیمیشن و صدا

برای پیاده‌سازی مکانیزم باز و بسته شدن درها در بازی، از Trigger Animation و پخش صدا استفاده کردم. مراحل انجام این کار به شرح زیر است:

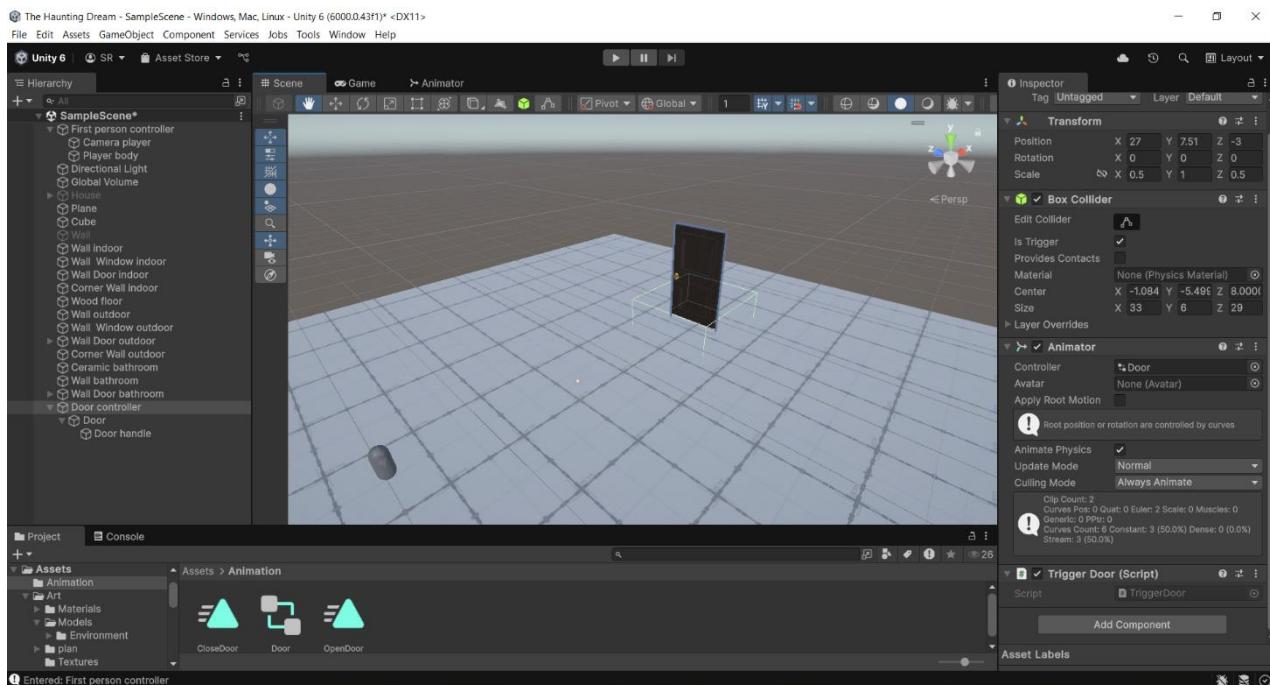
ابتدا با استفاده از پنجره Animation یک انیمیشن برای باز شدن و یکی برای بسته شدن در ساختم.

سپس یک Animator Controller ساختم و این دو انیمیشن را به آن اضافه کردم

در کنترل پخش انیمیشن‌ها از یک Trigger Animator Controller استفاده کردم



تنظیمات Animator برای انیمیشن در



ایجاد تریگر برای شناسایی باز شدن در

به در یک Box Collider اضافه کردم و تیک گزینه Is Trigger را فعال کردم. به پلیر یک Box Collider و همچنین Rigidbody (با تیک isKinematic فعال) اضافه کردم. سپس با استفاده از یک اسکریپت، در هنگام وارد شدن پلیر به محدوده Trigger، شرط بررسی شده و در صورت ورود، Trigger می‌شود.

```
using UnityEngine;
using TMPro;

public class TriggerDoor : MonoBehaviour
{
    private Animator _doorAnimator;
    private AudioSource _audioSource;
    public GameManager gameManager;
    public MonoBehaviour stageManager; //Reference to stage1

    public AudioClip openSound;
    public AudioClip closeSound;
    public AudioClip lockedSound;
    public TMP_Text interactionText;

    private bool isPlayerInTrigger = false;
    private bool isDoorOpen = false;
    private bool isLocked = false;
    private bool isPermanentlyUnlocked = false;

    void Start()
    {
        _doorAnimator = GetComponent<Animator>();
        _audioSource = GetComponent<AudioSource>();

        if (gameManager == null)
        {
            Debug.LogWarning("GameManager not assigned for door: " +
gameObject.name + ". Please assign in Inspector.");
        }
        if (stageManager == null)
        {
            Debug.LogWarning("StageManager not assigned for door: " +
gameObject.name + ". Please assign in Inspector.");
        }
        else
        {
            isLocked = true;
        }

        if (interactionText != null)
        {
            interactionText.enabled = false;
        }
        if (_audioSource == null)
        {
            gameObject.AddComponent<AudioSource>();
            _audioSource = GetComponent<AudioSource>();
        }
    }

    void Update()
    {
        if (isPlayerInTrigger && !isDoorOpen && !isLocked)
        {
            _doorAnimator.Play("Open");
            _audioSource.PlayOneShot(openSound);
            isDoorOpen = true;
        }
        if (!isPlayerInTrigger && isDoorOpen)
        {
            _doorAnimator.Play("Close");
            _audioSource.PlayOneShot(closeSound);
            isDoorOpen = false;
        }
        if (isLocked)
        {
            _doorAnimator.Play("Locked");
            _audioSource.PlayOneShot(lockedSound);
        }
    }

    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInTrigger = true;
        }
    }

    void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInTrigger = false;
        }
    }
}
```

```
        }

    }

    void Update()
    {
        if (isPlayerInTrigger)
        {
            if (!isDoorOpen && interactionText != null)
            {
                if (isPermanentlyUnlocked)
                {
                    interactionText.text = "Press [E] to open";
                }
                else
                {
                    interactionText.text = isLocked ?
(GetKeyCountFromStage() > 0 ? "Press [E] to open (1 key)" : "Door is
locked!") : "Press [E] to open";
                }
                interactionText.enabled = true;
            }

            if (Input.GetKeyDown(KeyCode.E))
            {
                if (isDoorOpen)
                {
                    _doorAnimator.SetTrigger("Close");
                    if (closeSound != null)
(audioSource.PlayOneShot(closeSound);
                    isDoorOpen = false;
                }
                else if (!isLocked)
                {
                    _doorAnimator.SetTrigger("Open");
                    if (openSound != null)
(audioSource.PlayOneShot(openSound);
                    isDoorOpen = true;
                    if (gameManager != null)
{
                        gameManager.StageCompleted();
}
                }
                else if (stageManager != null && stageManager is Stage1
stage1 && stage1.CheckKeyAndUnlock())
                {
                    _doorAnimator.SetTrigger("Open");
                    if (openSound != null)
(audioSource.PlayOneShot(openSound);
                    isDoorOpen = true;
```

```
        isLocked = false;
        isPermanentlyUnlocked = true;
    }
    else if (isLocked && lockedSound != null)
    {
        _audioSource.PlayOneShot(lockedSound);
    }

    if (interactionText != null)
        interactionText.enabled = false;
}
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = true;
    }
}

private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = false;
        if (interactionText != null)
        {
            interactionText.enabled = false;
        }
        if (isDoorOpen)
        {
            _doorAnimator.SetTrigger("Close");
            if (closeSound != null)
                _audioSource.PlayOneShot(closeSound);
            isDoorOpen = false;
        }
    }
}

public void LockDoor()
{
    if (!isPermanentlyUnlocked)
    {
        isLocked = true;
        isDoorOpen = false;
        Debug.Log("Door " + gameObject.name + " locked.");
    }
}
```

```
}

public void UnlockDoor()
{
    isLocked = false;
    Debug.Log("Door " + gameObject.name + " unlocked.");
}

public bool IsLocked()
{
    return isLocked;
}

public bool IsPlayerInTrigger()
{
    return isPlayerInTrigger;
}

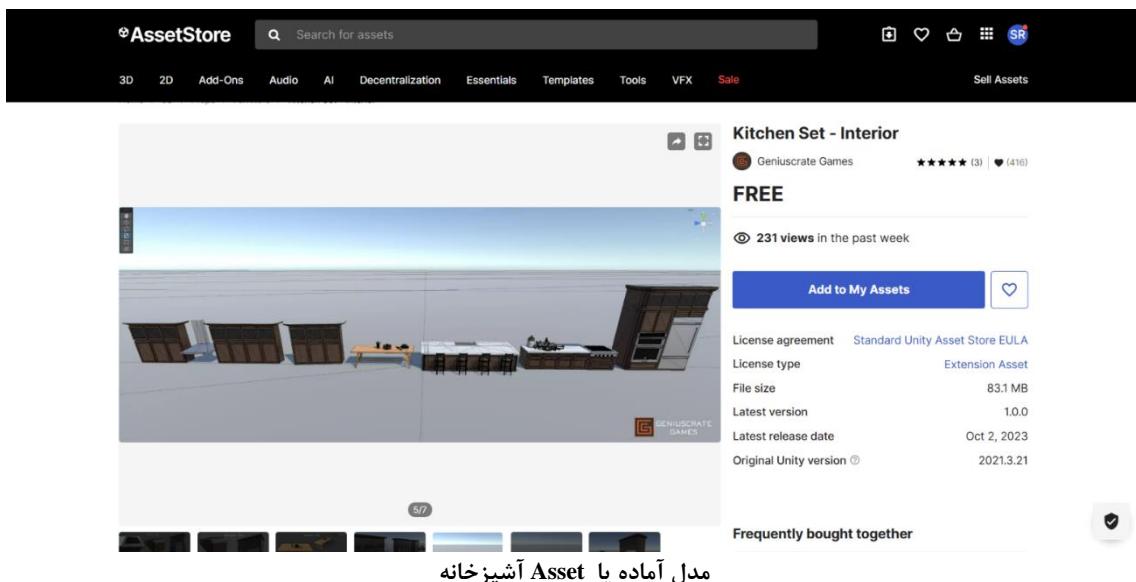
private int GetKeyCountFromStage()
{
    if (stageManager != null && stageManager is Stage1 stage1)
    {
        return stage1.GetKeyCount();
    }
    return 0;
}
}
```

کد TriggerDoor یک اسکریپت برای درهای تعاملی در محیط بازی Unity است که به بازیکن امکان می‌دهد با نزدیک شدن به در، پیام راهنمای بینند و با فشردن کلید E، در را باز یا بسته کند. در ابتدای اسکریپت (متد Start) اجزای موردنیاز مانند Animator (برای پخش اینیمیشن باز و بسته شدن)، AudioSource (برای پخش صدای مربوطه)، GameManager و StageManager (برای پخش صدای مرتبه)، interactionText (متند مقداردهی می‌شوند. اگر در قفل باشد و بازیکن کلید کافی داشته باشد، یا مرحله خاصی را پشت سر گذاشته باشد stage1.CheckKeyAndUnlock در باز می‌شود و برای دفعات بعدی نیز قفل نمی‌شود (isPermanentlyUnlocked = true). درون متند Update، اگر بازیکن وارد محدوده‌ی تریگر شده باشد (isPlayerInTrigger = true) و کلید E را بزند، در بسته یا باز می‌شود، یا در صورت قفل بودن و نداشتن کلید، صدای قفل شدن پخش می‌شود. همچنین با ورود و خروج از محدوده‌ی تریگر، نمایش متن راهنمای کنترل می‌شود. توابع UnlockDoor و LockDoor برای قفل یا باز کردن در از بیرون قابل استفاده هستند و با توابع IsPlayerInTrigger و IsLocked می‌توان وضعیت فعلی در را بررسی کرد. این کد به شکلی نوشته شده که به سادگی با سیستم مدیریت مراحل و کلیدها در بازی هماهنگ می‌شود و تعامل طبیعی و دینامیکی را برای بازیکن فراهم می‌سازد.

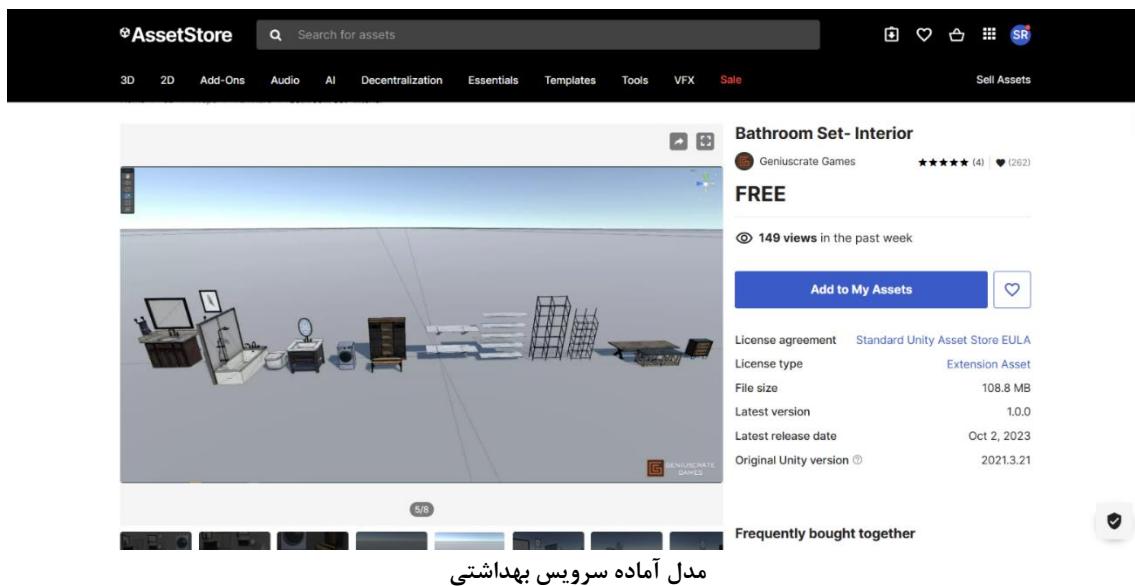
□ استفاده از مدل‌های آماده برای طراحی لوازم خانه

در پروژه، برای طراحی محیط‌های مختلف بازی، از مدل‌های آماده دانلود شده از Unity Asset Store استفاده کردم. این روش بهویژه در طراحی اجزای داخلی مانند آشپزخانه و دستشویی بسیار کارآمد بود و باعث سرعت‌بخشی به روند توسعه پروژه شد.

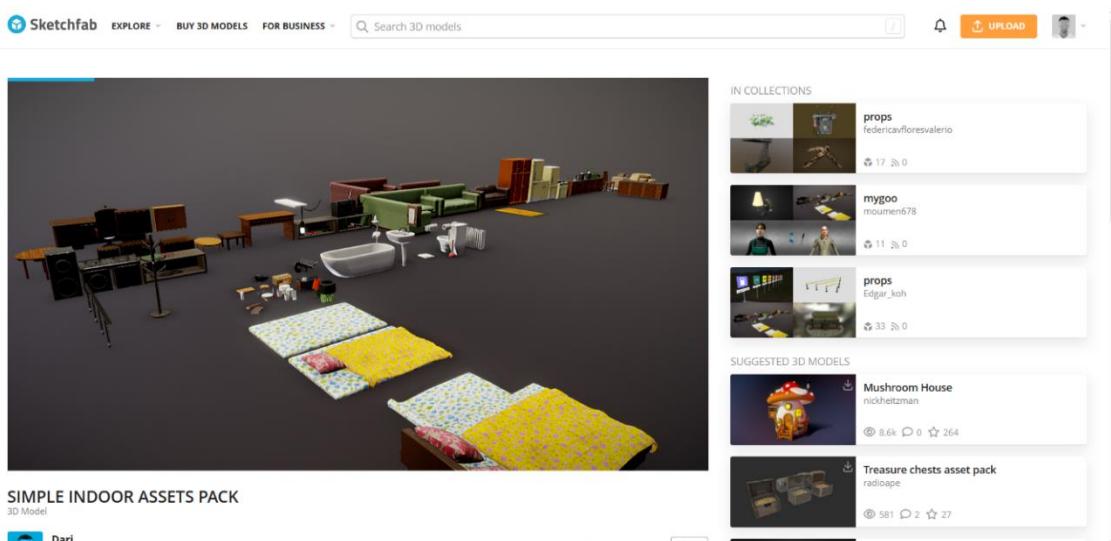
آشپزخانه: برای طراحی آشپزخانه، از مدل‌های آماده‌ای که شامل اجزای مختلف مانند یخچال، سینک ظرفشویی، اجاق گاز و کابینت‌ها بودند، استفاده کردم. این مدل‌ها به‌طور کامل با بافت‌ها و متریال‌های از پیش طراحی شده همراه بودند، که باعث کاهش زمان طراحی و بهینه‌سازی محیط شد. با وارد کردن این مدل‌ها به پروژه، تنها با تنظیم موقعیت و مقیاس آن‌ها در صحنه، توانستم محیط آشپزخانه را تکمیل کنم.



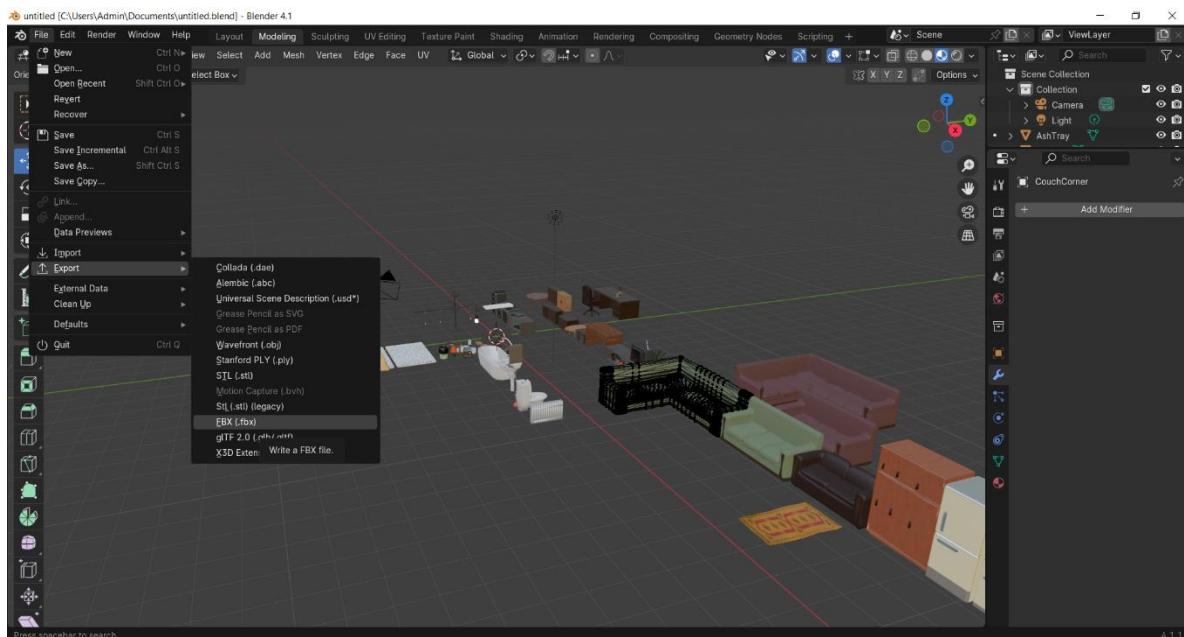
سرویس بهداشتی: در طراحی دستشویی، از مدل‌های آماده‌ای استفاده کردم که شامل اجزای اصلی مانند توالت، روشنویی، وان و شیرآلات بود. این مدل‌ها به‌راحتی در محیط بازی قرار گرفتند و برای شخصی‌سازی بیشتر، تنظیمات متریال‌ها و بافت‌ها بر اساس سبک گرافیکی بازی انجام شد.



لوازم خانه: برای طراحی وسایل داخلی خانه، از مدل‌های آماده که از Sketchfab دانلود کرده بودم، استفاده کردم. این مدل‌ها به صورت glb. بودند که برای استفاده در Unity نیاز به تبدیل به فرمت قابل پشتیبانی مانند fbx. داشتند. در ادامه، مراحل انجام این کار توضیح داده شده است. ابتدا مدل‌های مورد نیاز مانند مبلمان، لوازم آشپزخانه، و دیگر اجزای داخلی خانه را از Sketchfab دانلود کردم. این مدل‌ها به فرمت glb. بودند که یک فرمت گرافیکی رایج برای مدل‌های سه‌بعدی است و قابلیت انتقال به موتورهای بازی مختلف را دارا است.



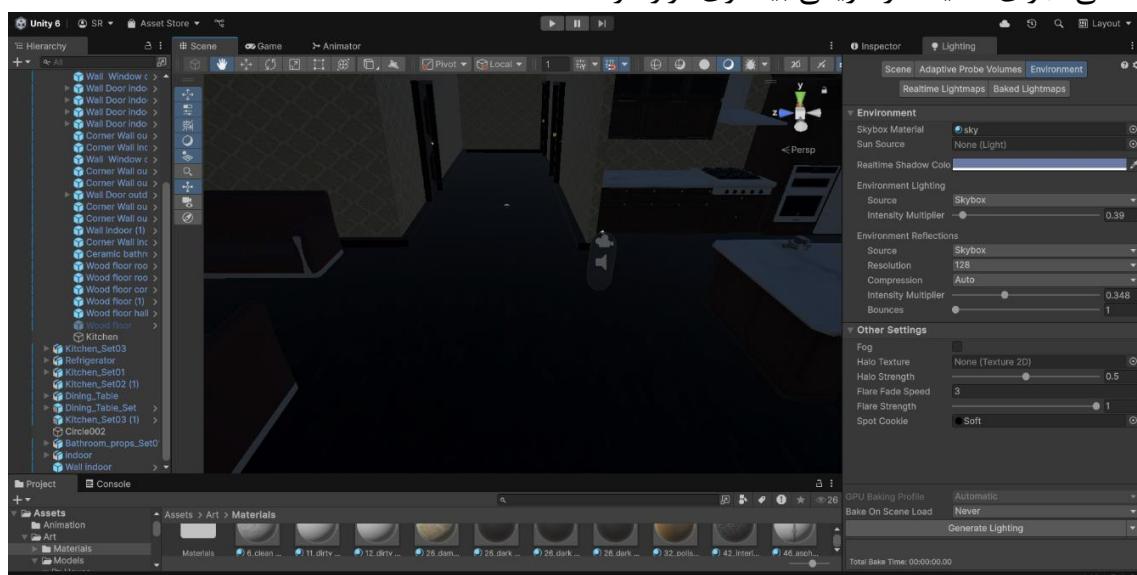
از آنجا که Unity به‌طور مستقیم از فرمت glb پشتیبانی نمی‌کند، برای استفاده از این مدل‌ها در Unity، آن‌ها را ابتدا با Blender وارد کرده و سپس به فرمت fbx تبدیل کردم. پس از تبدیل مدل به فرمت fbx، آن را به پروژه Unity وارد کردم. برای این کار، فایل .fbx را در پنجره Project در Unity کشیده و رها کردم تا مدل‌ها در پروژه ظاهر شوند و بتوان آن‌ها را در صحنه‌های بازی استفاده کرد.



تبديل فرمت در Blender

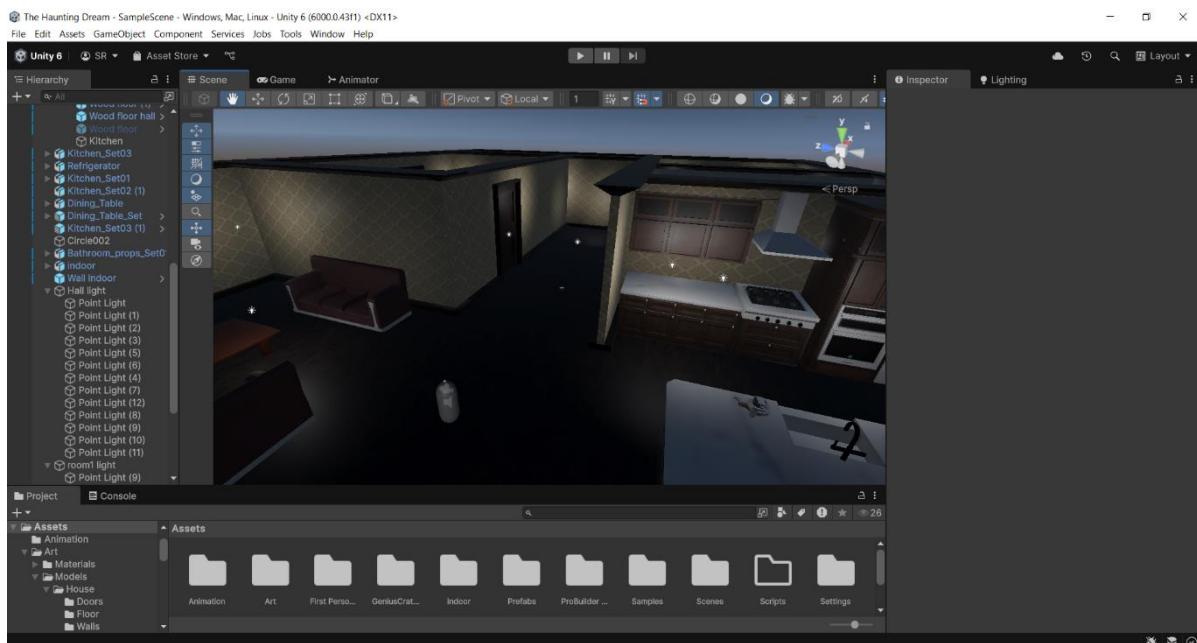
□ تنظیم نورپردازی و تاریک کردن محیط با Lighting و استفاده از Point Light

برای ایجاد جو ترسناک در محیط‌های داخلی مانند آشپزخانه و دستشویی، از تنظیمات Lighting و منابع نوری مختلف استفاده کرد. تنظیمات Lighting برای تاریک کردن محیط: در بخش Lighting Settings، شدت Ambient Light را کاهش دادم تا فضای بازی تاریک‌تر و مرموزتر شود. این کار باعث شد که نور عمومی محیط کم شده و فضای وحشت‌انگیز و دلهره‌آوری ایجاد گردد. با کاهش Intensity تمامی اجزای محیط در تاریکی بیشتری قرار گرفتند.



تاریک کردن محیط

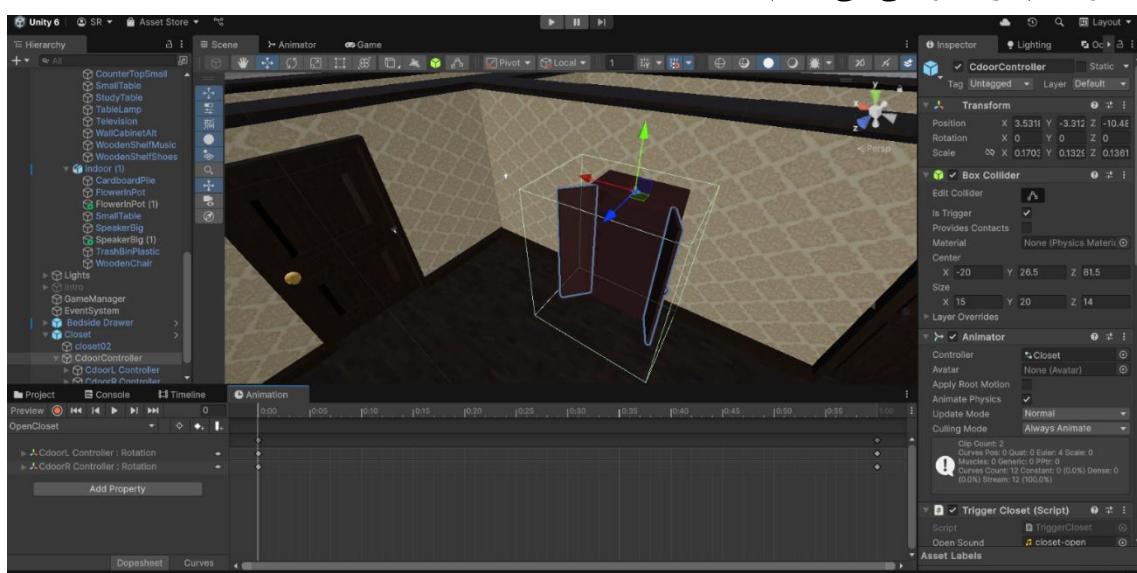
برای روشنایی موضعی و تاکید بر نواحی خاص، از Point Light ها استفاده کردم.



استفاده از Point light برای روشنایی

□ اینیمیشن باز شدن کمد و کشوی میز

ابتدا اینیمیشن کمد را ایجاد و طراحی کرده و سپس همانند مراحل ساخت در برای باز کردن آن توسط پلیر کدنویسی می کنیم.



انیمیشن و تریگر برای باز کردن در کمد

اسکریپت TriggerCloset برای باز کردن کمد:

```
using UnityEngine;
using TMPro;

public class TriggerCloset : MonoBehaviour
{
    private Animator _closetAnimator;
    private AudioSource _audioSource;

    public AudioClip openSound;
    public AudioClip closeSound;

    public TMP_Text interactionText;

    private bool isPlayerInTrigger = false;
    private bool isClosetOpen = false;

    void Start()
    {
        _closetAnimator = GetComponent<Animator>();
        _audioSource = GetComponent<AudioSource>();

        if (interactionText != null)
        {
            interactionText.enabled = false;
        }
    }

    void Update()
    {
        if (isPlayerInTrigger && !isClosetOpen &&
Input.GetKeyDown(KeyCode.E))
        {
            _closetAnimator.SetTrigger("Open");
            _audioSource.PlayOneShot(openSound);
            isClosetOpen = true;

            if (interactionText != null)
                interactionText.enabled = false;
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInTrigger = true;
        }
    }
}
```

```

        if (!isClosetOpen && interactionText != null)
            interactionText.enabled = true;
    }

private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = false;

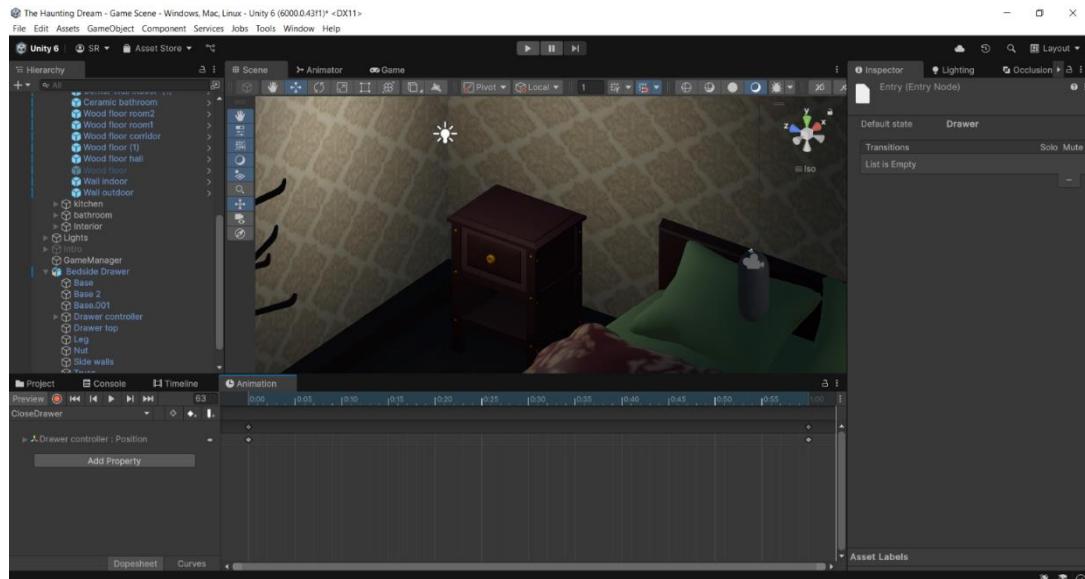
        if (interactionText != null)
            interactionText.enabled = false;

        if (isClosetOpen)
        {
            _closetAnimator.SetTrigger("Close");
            _audioSource.PlayOneShot(closeSound);
            isClosetOpen = false;
        }
    }
}
}

```

اسکریپت TriggerCloset برای کنترل یک کمد در بازی Unity طراحی شده که بازیکن می‌تواند با نزدیک شدن و زدن کلید E، آن را باز کند. در ابتدای بازی (متد Start)، کامپوننت‌های مورد نیاز یعنی AudioSource (برای پخش انیمیشن باز و بسته شدن) و Animator (برای پخش صدای باز و بسته شدن) از آبجکت گرفته می‌شوند و متن راهنمای تعامل (اگر تنظیم شده باشد) غیرفعال می‌شود. در هر فریم (متد Update)، اگر بازیکن در محدوده تریگر باشد و کمد هنوز باز نشده باشد، با فشردن کلید E، انیمیشن باز شدن کمد اجرا شده و صدای آن پخش می‌شود و متن راهنما مخفی می‌شود. در وقتی بازیکن وارد محدوده کمد می‌شود، اگر کمد بسته باشد، متن راهنمای تعامل نمایش داده می‌شود. در OnTriggerExit با خروج بازیکن، متن پنهان می‌شود و اگر کمد باز بود، با اجرای انیمیشن و صدای مناسب، دوباره بسته می‌شود. این اسکریپت تجربه‌ای تعاملی و سینمایی را در هنگام نزدیک شدن به کمد برای بازیکن فراهم می‌کند.

برای باز کردن کشوی میز هم همین روال را طی می‌کنیم.



انیمیشن و تریگر برای باز کردن کشوی میز

```

using UnityEngine;
using TMPro;

public class TriggerDrawer : MonoBehaviour
{
    private Animator _drawerAnimator;
    private AudioSource _audioSource;

    public AudioClip openSound;
    public AudioClip closeSound;

    public TMP_Text interactionText;

    private bool isPlayerInTrigger = false;
    private bool isDrawerOpen = false;

    void Start()
    {
        _drawerAnimator = GetComponent<Animator>();
        _audioSource = GetComponent<AudioSource>();

        if (interactionText != null)
        {
            interactionText.enabled = false;
        }
    }

    void Update()
    {
        if (isPlayerInTrigger && !isDrawerOpen &&
Input.GetKeyDown(KeyCode.E))
    }
}

```

```

    {
        _drawerAnimator.SetTrigger("Open");
        _audioSource.PlayOneShot(openSound);
        isDrawerOpen = true;

        if (interactionText != null)
            interactionText.enabled = false;
    }
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = true;

        if (!isDrawerOpen && interactionText != null)
            interactionText.enabled = true;
    }
}

private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = false;

        if (interactionText != null)
            interactionText.enabled = false;

        if (isDrawerOpen)
        {
            _drawerAnimator.SetTrigger("Close");
            _audioSource.PlayOneShot(closeSound);
            isDrawerOpen = false;
        }
    }
}
}

```

اسکریپت TriggerDrawer عملکرد یک کشو تعاملی را در بازی Unity پیاده‌سازی می‌کند که به بازیکن امکان می‌دهد با نزدیک شدن و فشردن کلید E، کشو را باز کند. در مت Start، کامپوننت‌های Animator (برای کنترل انیمیشن باز و بسته شدن کشو) و AudioSource (برای پخش صداهای باز و بسته شدن) از روی همان آبجکت گرفته می‌شوند و متن راهنمای تعامل (در صورت موجود بودن) غیرفعال می‌شود. در هر فریم (مت Update)، اگر بازیکن در محدوده تریگر باشد و کشو هنوز باز نشده

باشد، با فشردن کلید E، اینیمیشن باز شدن پخش می‌شود، صدای باز شدن اجرا شده و متن راهنما مخفی می‌شود. در OnTriggerEnter، اگر بازیکن وارد محدوده شود و کشو هنوز بسته باشد، متن تعامل فعال می‌شود تا بازیکن متوجه شود می‌تواند کشو را باز کند. در OnTriggerExit، با خروج بازیکن از محدوده، اگر کشو باز بود، اینیمیشن بسته شدن و صدای آن اجرا می‌شود و متن تعامل غیرفعال می‌شود. این اسکریپت یک تعامل ساده و واقع گرایانه با کشو را برای بازی‌های ترسناک یا داستان محور فراهم می‌سازد.

۴-۲-۴- ایجاد Player اول شخص

مرحله ۱: ایجاد ساختار اولیه پلیر

ابتدا یک GameObject جدید در صحنه ایجاد کردم و آن را با نام FirstPersonCamera ذخیره کردم. این آبجکت به عنوان والد اصلی کنترل کننده‌ی کاراکتر بازیکن عمل می‌کند. سپس دو جزء اصلی را به صورت زیر به این آبجکت اضافه کردم:

- Camera : برای دید بازیکن از دید اول شخص، به عنوان فرزند (child) Capsule (Body) : به عنوان جسم فیزیکی کاراکتر، برای برخورد با محیط و دریافت ورودی‌های حرکتی، زیرمجموعه‌ی همان آبجکت اصلی قرار گرفت.

مرحله ۲: افزودن Character Controller

برای کنترل فیزیک حرکت، به آبجکت اصلی FirstPersonCamera یک کامپوننت Character Controller اضافه کردم. این کامپوننت امکان برخورد صحیح با دیوارها، زمین و دیگر اجسام صحنه را فراهم می‌کند بدون نیاز به Physics Rigidbody یا دستی.

مرحله ۳: اسکریپتنویسی کنترل حرکت و نگاه

یک اسکریپت C# به آبجکت پلیر اضافه کردم که وظایف زیر را انجام می‌دهد:

اسکریپت MouseLook برای چرخش نگاه پلیر:

```
using UnityEngine;

public class MouseLook : MonoBehaviour
{
    public Transform cam;
    public Transform playerRoot;
```

```

public float sensitivity = 2f;

float rotX;
float rotY;

private void Start()
{
    Cursor.lockState = CursorLockMode.Locked;
}

private void Update()
{
    float mouseX = Input.GetAxis("Mouse X") * sensitivity;
    float mouseY = Input.GetAxis("Mouse Y") * sensitivity;

    rotX -= mouseY;
    rotY += mouseX;

    rotX = Mathf.Clamp(rotX, -90f, 90f);

    playerRoot.rotation = Quaternion.Euler(0f, rotY, 0f);
    cam.rotation = Quaternion.Euler(rotX, rotY, 0f);
}
}
}

```

اسکریپت MouseLook برای پیاده‌سازی کنترل دوربین اول شخص در بازی Unity طراحی شده و امکان چرخش آزادله دوربین با حرکت ماوس را فراهم می‌کند. در متod Start، اشاره‌گر ماوس قفل می‌شود (Cursor.lockState = CursorLockMode.Locked) تا از حرکت تصادفی آن جلوگیری شود. در هر فریم (متod Update)، ورودی‌های ماوس در محور X و Y دریافت شده و در مقدار حساسیت (sensitivity) ضرب می‌شوند. مقدار عمودی (rotX) از مقدار mouseY کم می‌شود تا حرکت بالا و پایین طبیعی‌تر باشد، و مقدار افقی (rotY) با mouseX جمع می‌شود. سپس مقدار rotX با تابع Mathf.Clamp محدود می‌شود تا از چرخش بیش از حد دوربین در جهت عمودی (نگاه کاملاً بالا یا پایین) جلوگیری شود. در نهایت، چرخش بدنه بازیکن (playerRoot) فقط حول محور Y (افقی) اعمال می‌شود، در حالی که چرخش دوربین (cam) همزمان در دو محور X و Y تنظیم می‌شود تا کنترل نگاه بازیکن را در محیط سه‌بعدی فراهم سازد.

اسکریپت PlayMovement برای حرکت پلیر:

```
using UnityEngine;
```

```

public class PlayMovement : MonoBehaviour
{
    public CharacterController controller;
    public Transform playerRoot;
    public float speed = 5f;

    void Update()
    {
        float x = Input.GetAxis("Horizontal");
        float z = Input.GetAxis("Vertical");

        Vector3 move = playerRoot.right * x + playerRoot.forward * z;
        controller.Move(move * speed * Time.deltaTime);
    }
}

```

اسکریپت PlayMovement برای حرکت بازیکن در فضای سه بعدی Unity با استفاده از کامپوننت CharacterController طراحی شده است. در متod Update، در هر فریم ورودی‌های صفحه کلید برای محورهای افقی (Horizontal) - کلیدهای A و D یا چپ و راست) و عمودی (Vertical) کلیدهای W و S یا بالا و پایین) دریافت می‌شوند. سپس با استفاده از جهت راست (right) و جلو (forward) مربوط به آبجکت playerRoot، بردار حرکت سه بعدی ساخته می‌شود که جهت حرکت بازیکن نسبت به چرخش فعلی او را مشخص می‌کند. این بردار در سرعت تعریف شده (speed) و از Time.deltaTime (برای مستقل بودن حرکت از نرخ فریم) ضرب می‌شود و با تابع Move از CharacterController اعمال می‌شود تا بدون نیاز به فیزیک Rigidbody بازیکن به صورت نرم و کنترل شده حرکت کند. این اسکریپت برای بازی‌هایی با دید اول شخص یا سوم شخص پایه‌ای مناسب است.

در آخر اسکریپت FootStepSound برای صدای قدم‌های پلیر:

```

using UnityEngine;

public class FootStepSound : MonoBehaviour
{
    public AudioSource footstepsAudio;

    void Update()
    {
        if (footstepsAudio == null)
        {

```

```

        Debug.LogWarning("FootstepsAudio is not initialized in the
Inspector!");
        return;
    }

    if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.A) ||
        Input.GetKeyDown(KeyCode.S) || Input.GetKeyDown(KeyCode.D) ||
        Input.GetKeyDown(KeyCode.UpArrow) ||
        Input.GetKeyDown(KeyCode.DownArrow) ||
        Input.GetKeyDown(KeyCode.LeftArrow) ||
        Input.GetKeyDown(KeyCode.RightArrow))
    {
        if (!footstepsAudio.isPlaying)
        {
            footstepsAudio.Play();
        }
        else
        {
            if (footstepsAudio.isPlaying)
            {
                footstepsAudio.Stop();
            }
        }
    }
}
}

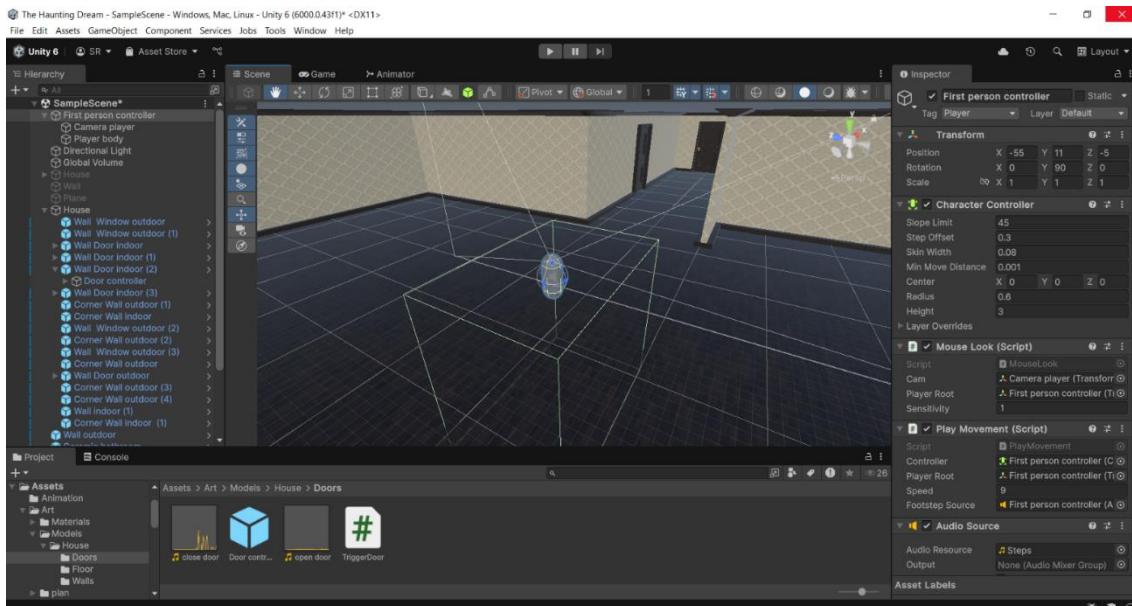
```

اسکریپت FootStepSound برای پخش صدای قدمزدن بازیکن در هنگام حرکت طراحی شده است. ابتدا در متد Update برسی می‌شود که آیا AudioSource مربوط به صدای قدمها مقداردهی شده یا خیر؛ در صورت خالی بودن، پیام هشدار در Inspector (footstepsAudio) نمایش داده شود. سپس برسی می‌شود که آیا یکی از کلیدهای حرکتی (W, A, S, D) یا کلیدهای جهتی کیبورد فشرده شده است یا نه. اگر بازیکن در حال حرکت باشد و صدای قدمها هنوز پخش نمی‌شود، صدای قدمها پخش می‌شود. در غیر این صورت، اگر بازیکن حرکت نکند و صدا هنوز در حال پخش باشد، صدا متوقف می‌شود. توجه داشته باش که در خط شرط برسی کلیدها یک خطای نحوی وجود دارد؛ باید بین Input.GetKeyDown و عملگر منطقی || (یا) گذاشته شود، و گرنه کد اجرا نخواهد شد. به طور کلی، این اسکریپت صدای قدمها را با حرکات بازیکن هماهنگ می‌کند تا تجربه‌ای طبیعی‌تر و واقع‌گرایانه‌تر در محیط بازی ایجاد شود.

مرحله ۴: تنظیم ارتفاع، سرعت و حساسیت

پارامترهایی مانند موارد زیر در اسکریپت برای تنظیم بهتر تجربه کاربری استفاده شد:

- سرعت حرکت (moveSpeed)
- حساسیت موس (mouseSensitivity)
- ارتفاع پلیر و موقعیت دوربین نسبت به کپسول (camera offset)



ایجاد first person controller

۴-۲-۵- مدیریت مراحل توسط Game Manager

بازی به طور کلی شامل شش صحنه^۱ است که هر کدام ممکن است شامل یک یا چند مرحله^۲ باشند. مدیریت Stage‌ها بر عهده اسکریپت GameManager است. هر Stage نیز می‌تواند یک یا چند رویداد را در بر بگیرد. به طور کلی صحنه‌ها و مرحله‌ها به این صورت تعریف شدند که مراحل بازی را توضیح می‌دهند.

¹ Scene

² Stage

رویداد	Satge	Scene	ردیف
منوی اصلی	...	Main Menu	۱
مقدمه بازی	...	Intro	۲
انیمیشن بیدار شدن و ایستادن پلیر	Stand Up		
باز کردن در اتاق با کلید	Stage1		
شنیدن صدای در و دیالوگ مادر و رمز عددی	Stage2	HouseFirstScene	۳
تغییر صحنه و دیدن زن ترسناک	Stage3		
صحنه و دیالوگ تدی و انتقال به مکانی دیگر	Stage4		
انیمیشن ترسیده بازیکن	Afraid	TeddyScene	۴
چالش تعقیب کابوس و چراغ قوه	Stage5		
تغییر مکان به اتاق مشاور و برخورد با کابوس	Stage6	HouseEndDream	۵
صحنه آخر بیدار شدن، دیالوگ مادر	Stage7	HouseWakeUp	۶

مراحل کلی بازی و مدیریت آن با Scene و Stage ها

▣ اسکریپت Game Manager در Scene اول برای مدیریت stage ها

```
using UnityEngine;

public class GameManager : MonoBehaviour
{
    public GameObject[] stages;
    private int currentStageIndex = 0;
    public TriggerDoor lockedDoor;
    private bool isStageInitialized = false;
    private bool isGameCompleted = false;

    void Start()
    {
        PlayerPrefs.DeleteAll();
        Debug.Log("PlayerPrefs reset for testing.");

        currentStageIndex = PlayerPrefs.GetInt("CurrentStage", 0);
    }
}
```

```
        Debug.Log($"Loaded currentStageIndex from PlayerPrefs:  
{currentStageIndex}");  
  
        if (stages == null || stages.Length == 0)  
        {  
            Debug.LogError("Stages array is null or empty! Please assign  
stages in Inspector.");  
            return;  
        }  
  
        if (currentStageIndex >= stages.Length || currentStageIndex < 0)  
        {  
            Debug.LogWarning($"Invalid currentStageIndex:  
{currentStageIndex}. Resetting to 0.");  
            currentStageIndex = 0;  
            PlayerPrefs.SetInt("CurrentStage", currentStageIndex);  
            PlayerPrefs.Save();  
        }  
  
        Debug.Log($"Stages array count: {stages.Length}");  
        for (int i = 0; i < stages.Length; i++)  
        {  
            Debug.Log($"Stage {i}: {(stages[i] != null ? stages[i].name :  
"null")});  
        }  
  
        PlayerPrefs.Save();  
        UpdateStage();  
        if (lockedDoor != null)  
        {  
            lockedDoor.LockDoor();  
            Debug.Log("Door " + lockedDoor.gameObject.name + " is  
locked.");  
        }  
        isStageInitialized = true;  
        Debug.Log($"Initial stage activation check: Starting with Stage  
{stages[currentStageIndex].name});  
    }  
  
    void UpdateStage()  
    {  
  
        Debug.Log($"Updating stage, currentStageIndex:  
{currentStageIndex}, stages length: {stages.Length}");  
        if (stages == null || stages.Length == 0)  
        {  
            Debug.LogError("Stages array is null or empty during  
UpdateStage!");  
        }  
    }  
}
```

```
        return;
    }

    foreach (GameObject stage in stages)
    {
        if (stage != null && stage != stages[currentStageIndex])
        {
            stage.SetActive(false);
            Debug.Log($"Deactivated stage: {stage.name}");
        }
    }

    if (currentStageIndex >= 0 && currentStageIndex < stages.Length
&& stages[currentStageIndex] != null)
    {
        stages[currentStageIndex].SetActive(true);
        Stage3 stage3 =
stages[currentStageIndex].GetComponent<Stage3>();
        if (stage3 != null)
        {
            stage3.ActivateStage();
            Debug.Log($"Activated Stage3 and triggered light color
change.");
        }
        else
        {
            Debug.LogWarning("Stage3 component not found on " +
stages[currentStageIndex].name);
        }
    }
    else
    {
        Debug.LogError("Failed to activate stage!");
        Debug.LogWarning("No valid stage to activate! Resetting to
Stage 0.");
        currentStageIndex = 0;
        PlayerPrefs.SetInt("CurrentStage", currentStageIndex);
        PlayerPrefs.Save();
        if (stages.Length > 0 && stages[0] != null)
        {
            stages[0].SetActive(true);
            Debug.Log("Reset to Stage 0: " + stages[0].name);
        }
    }
}

public void StageCompleted()
{
    if (stages == null || stages.Length == 0)
```

```
        {
            Debug.LogError("Cannot complete stage: Stages array is null
or empty!");
            return;
        }

        if (!isStageInitialized)
        {
            Debug.LogWarning("StageCompleted called before
initialization! Ignoring.");
            return;
        }

        if (!isGameCompleted)
        {
            currentStageIndex++;
            Debug.Log($"Stage completed, advancing to index:
{currentStageIndex}");
            if (currentStageIndex >= stages.Length)
            {
                Debug.Log("All stages completed! Game finished.");
                isGameCompleted = true;
                return;
            }
            PlayerPrefs.SetInt("CurrentStage", currentStageIndex);
            PlayerPrefs.Save();
            UpdateStage();
        }
    }
}
```

اسکریپت GameManager مسئول مدیریت مراحل Stages بازی است و از طریق سیستم PlayerPrefs پیشرفت بازیکن را ذخیره و بازیابی می‌کند. در ابتدای بازی (متد Start) ابتدا تمام داده‌های PlayerPrefs برای تست حذف می‌شوند، سپس اندیس مرحله جاری (currentStageIndex) از حافظه بارگذاری می‌شود. اگر آرایه‌ی stages در Inspector تنظیم نشده یا خالی باشد، پیام خطأ چاپ شده و ادامه متوقف می‌شود. اگر اندیس مرحله معتبر نباشد، به بازنشانی می‌شود و در نهایت متد UpdateStage اجرا می‌گردد تا تنها مرحله‌ی فعال، مرحله جاری باشد. اگر دری به عنوان lockedDoor تعریف شده باشد، در نیز قفل می‌شود. در متدهای UpdateStage همه مراحل غیرفعال می‌شوند به جز مرحله‌ی جاری که فعال شده و اگر کامپوننت StageCompleted را آن موجود باشد، متد ActivateStage اجرا می‌شود. متد StageCompleted در صورتی فراخوانی می‌شود که بازیکن یک مرحله را به پایان رسانده باشد؛ این متد ابتدا بررسی می‌کند که مراحل تعریف شده‌اند و بازی قبلًاً مقداردهی اولیه شده، سپس اندیس مرحله را افزایش می‌دهد و

اگر به پایان مراحل برسد، بازی را تمام شده اعلام می‌کند. در غیر این صورت، مرحله جدید ذخیره شده و متده `UpdateStage` برای به روز رسانی محیط بازی فراخوانی می‌شود. این اسکریپت قلب اصلی پیشرفت مرحله‌ای بازی است و هماهنگی میان صحنه‌ها، ذخیره‌سازی و تعاملات مربوط به درها و نورپردازی را مدیریت می‌کند.

۴-۲-۶ - پیاده سازی Stage اول

در مرحله‌ی اول بازی، که بازیکن تازه از خواب بیدار شده (`Stand Up`)، سه عنصر کلیدی مدنظر است: یک کشو، یک کلید، و یک در قفل شده. روند این بخش ساده است؛ بازیکن باید کلید را از داخل کشو برداشته و با استفاده از آن، در قفل شده را باز کند.

اسکریپت `StandUp` برای آنیمیشن بیدار شدن :

```
using UnityEngine;

public class StandUp : MonoBehaviour
{
    public GameObject standUp;
    public GameObject firstPersonController;
    private GameManager gameManager;

    void Start()
    {
        gameManager = FindObjectOfType<GameManager>();
        if (gameManager == null)
        {
            Debug.LogError("GameManager not found in the scene!");
            return;
        }

        if (standUp == null)
        {
            Debug.LogError("StandUp reference is not assigned!");
            return;
        }

        standUp.SetActive(true);
        Animator animator = standUp.GetComponent<Animator>();
        if (animator == null)
        {
            Debug.LogError("Animator component not found on StandUp!");
            return;
        }
    }
}
```

```
        }

        AnimatorStateInfo stateInfo =
animator.GetCurrentAnimatorStateInfo(0);
        float animationLength = stateInfo.length > 0 ? stateInfo.length :
2f;
        Invoke("OnStageCompleted", animationLength);

        if (firstPersonController != null)
{
    firstPersonController.SetActive(false);
}
else
{
    Debug.LogError("FirstPersonController reference is not
assigned!");
}
}

void OnStageCompleted()
{
    if (standUp != null) standUp.SetActive(false);
    if (firstPersonController != null)
firstPersonController.SetActive(true);
    if (gameManager != null) gameManager.StageCompleted();
}
}
```

اسکریپت StandUp برای اجرای یک انیمیشن آغازین (مانند بیدار شدن از خواب یا برخاستن) طراحی شده که پس از پایان آن، کنترل بازی به بازیکن داده می‌شود و مرحله فعلی به صورت خودکار کامل می‌گردد. در متده استارتا Start ابتدا اسکریپت GameManager در صحنه جستجو و ذخیره می‌شود. سپس بررسی می‌شود که آیا ارجاعات standUp (آبجکت انیمیشنی برای برخاستن) و کنترل کننده بازیکن (firstPersonController) به درستی در Inspector تعیین شده‌اند یا نه. اگر standUp معتبر باشد، فعال می‌شود و انیماتور آن بررسی می‌شود؛ اگر انیماتور وجود داشته باشد، زمان اجرای انیمیشن جاری از طریق AnimatorStateInfo گرفته می‌شود (یا مقدار پیش‌فرض ۲ ثانیه در نظر گرفته می‌شود) و با تابع Invoke متد OnStageCompleted پس از اتمام انیمیشن فراخوانی می‌گردد. در این متده، انیمیشن standUp غیرفعال می‌شود، کنترل بازیکن GameManager از StageCompleted فعال شده و در نهایت متده firstPersonController برای پیشروی به مرحله بعدی اجرا می‌شود. این اسکریپت انتقالی سینمایی و روان بین آغاز مرحله و شروع گیم‌پلی را فراهم می‌کند.

اسکریپت KeyPickup برای برداشتن کلید:

```
using UnityEngine;
using TMPro;

public class KeyPickup : MonoBehaviour
{
    private Stage1 stage1; // Reference GameManager
    public TMP_Text pickupText;
    public AudioClip pickupSound;
    private AudioSource audioSource;

    void Start()
    {
        stage1 = FindObjectOfType<Stage1>(); // found Stage1 in Scene
        audioSource = gameObject.GetComponent<AudioSource>(); // Check
        AudioSource
        if (audioSource == null)
        {
            audioSource = gameObject.AddComponent<AudioSource>();
        }
        if (pickupText != null)
        {
            pickupText.enabled = false; // Disabled text in start
        }

        if (audioSource != null)
        {
            audioSource.volume = 1.0f;
            audioSource.playOnAwake = false;
        }
    }

    void OnTriggerStay(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            if (pickupText != null)
            {
                pickupText.text = "Press [F] To Pick Up";
                pickupText.enabled = true;
            }

            if (Input.GetKeyDown(KeyCode.F))
            {
                if (stage1 != null)
                {
                    stage1.AddKey();
                }
            }
        }
    }
}
```

```

        if (pickupSound != null && audioSource != null)
    {
        // Add a GameObject for playing sound
        GameObject soundObject = new GameObject("TempAudio");
        AudioSource tempAudio =
soundObject.AddComponent<AudioSource>();
        tempAudio.volume = 1.0f;
        tempAudio.PlayOneShot(pickupSound);
        // Destroy the object after finished sound
        Object.Destroy(soundObject, pickupSound.length);
    }
    else
    {
        Debug.LogWarning("Failed to play sound");
    }
    gameObject.SetActive(false); // Disabled key after pick
up
    Debug.Log("Key picked up!");
    if (pickupText != null)
    {
        pickupText.enabled = false;
    }
}
}

void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player") && pickupText != null)
    {
        pickupText.enabled = false;
    }
}
}

این اسکریپت مسئول مدیریت برداشتن کلید توسط بازیکن است. وقتی بازیکن وارد ناحیه‌ای می‌شود که کلید در آن قرار دارد (با استفاده از Trigger)، یک متن راهنمایی نمایش داده می‌شود. در صورتی که بازیکن کلید F را فشار دهد، صدای مربوط به برداشتن کلید پخش شده، شیء کلید از صحنه حذف می‌شود، و متدهای در اسکریپت دیگری (Stage1) فراخوانی می‌شود تا مشخص شود کلید جمع‌آوری شده است. این روند ساده و مستقیم، یکی از رایج‌ترین الگوهای برای پیاده‌سازی مکانیک جمع‌آوری آیتم‌ها در بازی‌هاست.

و در نهایت اسکریپت Stage1 :
```

using UnityEngine;

```
public class Stage1 : MonoBehaviour
{
    public TriggerDoor lockedDoor;
    private bool isStageCompleted = false;
    private int keyCount = 0;

    void Start()
    {

        keyCount = PlayerPrefs.GetInt("Stage1KeyCount", 0);
        if (lockedDoor != null)
        {
            lockedDoor.LockDoor();
        }
        else
        {
            Debug.LogWarning("No locked door assigned to this stage!");
        }
    }

    // collect keys
    public void AddKey()
    {
        keyCount++;
        PlayerPrefs.SetInt("Stage1KeyCount", keyCount);
        PlayerPrefs.Save();
        Debug.Log("Key collected! Total keys: " + keyCount);
    }

    public bool UseKey()
    {
        if (keyCount > 0)
        {
            keyCount--;
            PlayerPrefs.SetInt("Stage1KeyCount", keyCount);
            PlayerPrefs.Save();
            Debug.Log("Key used! Remaining keys: " + keyCount);
            return true;
        }
        Debug.LogWarning("No keys available!");
        return false;
    }

    public int GetKeyCount()
    {
        return keyCount;
    }
}
```

```

public bool CheckKeyAndUnlock()
{
    if (lockedDoor != null && lockedDoor.IsLocked()) // one key for
openning door
    {
        if (UseKey())
        {
            lockedDoor.UnlockDoor();
            Debug.Log($"Door {lockedDoor.gameObject.name} unlocked
with a key. Remaining keys: " + keyCount);
            CompleteStage();
            return true; // sucess
        }
    }
    else if (lockedDoor != null && lockedDoor.IsLocked())
    {
        Debug.LogWarning("Need at least 1 key to unlock the door!
Current keys: " + keyCount);
    }
    return false; // failed
}

public void CompleteStage()
{
    if (!isStageCompleted)
    {
        isStageCompleted = true;
        if (lockedDoor != null)
        {
            lockedDoor.UnlockDoor(); // unlocked the door
        }
        GameObject.FindObjectOfType<GameManager>().StageCompleted();
// To GameManager
        Debug.Log("Stage completed!");
    }
}
}

```

این اسکریپت Stage 1 در یونیتی برای مدیریت وضعیت مرحله اول بازی طراحی شده است. وظیفه‌ی آن پیگیری تعداد کلیدهای جمع‌آوری شده توسط بازیکن، باز کردن در قفل شده در صورت داشتن کلید، و علامت‌گذاری پایان مرحله است. در شروع بازی، تعداد کلیدها از طریق PlayerPrefs بارگذاری می‌شود و در قفل می‌شود. وقتی بازیکن کلیدی جمع می‌کند، تابع AddKey تعداد کلیدها را افزایش داده و در ذخیره می‌کند. برای باز کردن در، ابتدا بررسی می‌شود که آیا در

قفل است و بازیکن کلید دارد؛ در این صورت، با استفاده از متدهای UseKey و CompleteStage به پایان می‌رسد و متدهای از کلاس فراخوانی می‌شود. اگر موفق باشد، مرحله با تابع GameManager را ثبت کند.



برداشتن کلید

۴-۲-۷ - پیاده سازی Stage دوم

در Stage دوم، دو Game object به نام‌های knockTrigger و EscapeTrigger وجود دارد. در حالت Box Collider یک knockTrigger قرار دارد که محدوده‌ای در راه را مشخص می‌کند. وقتی بازیکن وارد این محدوده شود، صدای در زدن پخش می‌شود و پیامی نمایش داده می‌شود مبنی بر اینکه این صدا از سمت در خروجی می‌آید. این اتفاق باعث می‌شود بازیکن به سمت در خروجی حرکت کند.

زمانی که بازیکن به نزدیکی در خروجی برسد، وارد محدوده‌ی EscapeTrigger می‌شود. در این لحظه صدای در زدن متوقف شده و صدای مادری که متعلق به شخصیت بازیکن است پخش می‌شود. دیالوگ‌ها و صدای مادر با استفاده از Timeline کنترل شده‌اند و تنها یک بار اجرا می‌شوند.

پس از پایان این بخش، بازی وارد مرحله‌ی بعدی می‌شود که Keypad Controller نام دارد. این مرحله شامل رمز دیجیتالی یکی از اتفاق‌های است که بازیکن باید معماهی را که در خانه قرار دارد (از جمله

قابل عکس، تعداد مسواک‌ها، تعداد سیب‌ها، تعداد قوطی‌های نوشابه و تعداد برگ‌های گلدان) حل کند و با استفاده از آن‌ها رمز در را پیدا کند.

وقتی بازیکن به صفحه کلید دیجیتال نزدیک می‌شود، می‌تواند با فشار دادن کلید F، پنجره‌ی EnterKeypadCanvas را فعال یا غیرفعال کند. پس از وارد کردن رمز صحیح، این پنجره به طور کامل بسته شده و در قفل شده برای همیشه باز می‌ماند.

در ابتدا اسکریپت TriggerKnock برای صدای در:

```
using UnityEngine;
using TMPro;

public class TriggerKnock : MonoBehaviour
{
    public AudioSource knockSound;
    public TMP_Text interactionText;

    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player") && knockSound != null &&
!knockSound.isPlaying)
        {
            knockSound.Play();
            Debug.Log("Knock sound started.");
            interactionText.text = "The sound is coming from the exit
door!";
        }
    }
}
```

در این کد، یک اسکریپت برای بازی یونیتی نوشته شده است که به‌طور خاص به صدای کوییدن در و نمایش متن تعامل در واکنش به برخورد بازیکن با یک منطقه خاص (Trigger) پرداخته است. در ابتدا، دو متغیر عمومی تعریف شده‌اند: knockSound از نوع AudioSource که برای پخش صدای کوییدن در استفاده می‌شود و interactionText از نوع TMP_Text که برای نمایش پیامی در رابط کاربری (UI) بازی به کار می‌رود.

در عنوان متدهای OnTriggerEnter که زمانی اجرا می‌شود که یک شی با تریگر برخورد می‌کند، چک می‌شود که آیا شی برخورد کرده با تگ "Player" است یا نه. اگر این‌طور باشد و همچنین صدای کوییدن (knockSound) وجود داشته باشد و در حال حاضر در حال پخش نباشد، صدای کوییدن پخش می‌شود و پیامی در کنسول نمایش داده می‌شود که نشان می‌دهد صدای کوییدن آغاز شده است. سپس، متن داخل interactionText تغییر می‌کند و پیامی به نمایش درمی‌آید که بیان

می‌کند "صدای کوبیدن از درب خروجی می‌آید!".
 این کد به طور ساده باعث می‌شود تا وقتی که بازیکن به یک ناحیه خاص وارد شود، صدای کوبیدن پخش شود و پیامی در صفحه به بازیکن نمایش داده شود.



به صدا درآمدن در

حالا اسکریپت TriggerEscape رو داریم که Game Object مربوط به این اسکریپت دارای TimeLine هست:

```
using UnityEngine;
using UnityEngine.Playables;

public class TriggerEscape : MonoBehaviour
{
    public AudioSource escapeVoice;
    public GameObject canvasWithTimeline; // Reference to canvas with timeline
    public GameObject KnockDoor;
    private bool hasPlayed = false; // Not repeat sound
    private PlayableDirector playableDirector; // Reference to Playable Director

    void Start()
    {
        if (canvasWithTimeline != null)
        {
            playableDirector =
            canvasWithTimeline.GetComponent<PlayableDirector>();
            if (playableDirector != null)
```

```
        {
            playableDirector.stopped += OnTimelineStopped; // Sign
        event in Start
            Debug.Log("PlayableDirector event registered.");
        }
        else
        {
            Debug.LogError("PlayableDirector not found on Canvas!");
        }
    }

void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player") && !hasPlayed)
    {
        Debug.Log("Player entered TriggerEscape.");
        KnockDoor.SetActive(false);

        if (canvasWithTimeline != null)
        {
            canvasWithTimeline.SetActive(true);
            Debug.Log("Canvas activated.");

            if (playableDirector != null &&
!playableDirector.state.Equals(PlayState.Playing))
            {
                playableDirector.Play();
                Debug.Log("Timeline started.");
                hasPlayed = true;
            }
            else
            {
                Debug.LogWarning("Timeline is already playing or
PlayableDirector is null.");
            }
        }
    }
}

void OnTimelineStopped(PlayableDirector director)
{
    Debug.Log("Timeline stopped, deactivating escapeTrigger.");
    if (gameObject != null)
    {
        gameObject.SetActive(false);
        Debug.Log("escapeTrigger deactivated.");
    }
    if (canvasWithTimeline != null)
```

```

    {
        canvasWithTimeline.SetActive(false);
    }
    if (escapeVoice != null && escapeVoice.isPlaying)
    {
        escapeVoice.Stop();
        Debug.Log("Escape voice stopped.");
    }
}

void OnDestroy()
{
    if (playableDirector != null)
    {
        playableDirector.stopped -= OnTimelineStopped; // Delete
event in destroy
    }
}
}

```

در این کد، بهویژه هنگامی که بازیکن وارد یک ناحیه خاص می‌شود، صدای فرار مادر پخش می‌شود، یک تایم‌لاین (timeline) شروع به پخش می‌کند. در ابتدا، متغیرهایی برای نگهداری ارجاع به صدا (escapeVoice)، کانواسی که تایم‌لاین را نمایش می‌دهد (canvasWithTimeline)، و درب کوبیده شده (KnockDoor) تعریف شده‌اند. متغیر hasPlayed برای جلوگیری از پخش تکراری صدا و تایم‌لاین مورد استفاده قرار می‌گیرد.

در متد Start، ابتدا بررسی می‌شود که آیا ارجاع به canvasWithTimeline موجود است. سپس، PlayableDirector که مسئول پخش تایم‌لاین است از کانواس گرفته می‌شود. اگر OnTimelineStopped وجود داشته باشد، یک رویداد (event) به نام PlayableDirector برای متوقف شدن تایم‌لاین ثبت می‌شود. در صورت عدم وجود PlayableDirector یا کانواس، پیغام خطا در کنسول نمایش داده می‌شود.

در متد OnTriggerEnter، زمانی که بازیکن به تریگر وارد می‌شود، بررسی می‌شود که آیا هنوز تایم‌لاین پخش نشده است (با استفاده از متغیر hasPlayed). اگر تایم‌لاین قبلًا پخش نشده باشد، درب کوبیده شده غیرفعال می‌شود و سپس کانواس با تایم‌لاین فعال می‌شود. اگر تایم‌لاین قبلًا در حال پخش نباشد، تایم‌لاین شروع به پخش می‌شود و متغیر hasPlayed به true تغییر می‌کند تا از پخش دوباره جلوگیری شود.

در نهایت، متد OnTimelineStopped زمانی که تایم‌لاین متوقف می‌شود، اجرا می‌شود. در این

متد، تریگر فرار غیرفعال می‌شود، کانواس غیرفعال می‌شود و اگر صدا در حال پخش باشد، متوقف می‌شود. همچنین در متد OnDestroy PlayableDirector موجود باشد، رویداد ثبت شده از بین می‌رود تا از بروز مشکلات در هنگام نابودی شیء جلوگیری شود. این کد به طور کلی باعث می‌شود که پس از ورود بازیکن به منطقه خاص، صدای در زدن متوقف و تایم‌لاین شروع شود و پس از اتمام تایم‌لاین، شیء TriggerEscape غیرفعال شوند.



دیالوگ مادر پشت در با استفاده از TimeLine

سپس اسکریپت TriggerKeypad که مسئول رمز عددی هست:

```
using UnityEngine;
using TMPro;
using UnityEngine.EventSystems;

public class TriggerKeypad : MonoBehaviour
{
    public TMP_Text interactionText;
    public GameObject EnterKeypadCanvas; // Reference to keypad canvas
    private bool isPlayerInTrigger = false;
    private bool isCanvasActive = false;
    private bool isSuccess = false;

    public TMP_InputField txtholder;
    public GameObject button1;
    public GameObject button2;
    public GameObject button3;
    public GameObject button4;
    public GameObject button5;
```

```
public GameObject button6;
public GameObject button7;
public GameObject button8;
public GameObject button9;
public GameObject button0;
public GameObject buttonClear;
public GameObject buttonEnter;

private AudioSource audioSource;
public AudioClip buttonClickSound;
public AudioClip successSound;
public AudioClip failedSound;

void Start()
{
    interactionText.enabled = false;
    txtholder.placeholder.GetComponent<TextMeshProUGUI>().color =
Color.white;

    if (EnterKeypadCanvas != null)
    {
        EnterKeypadCanvas.SetActive(false); // Canvas must be
disabled in start
    }
    // Event system in Scene
    if (FindObjectOfType<EventSystem>() == null)
    {
        GameObject eventSystem = new GameObject("EventSystem");
        eventSystem.AddComponent<EventSystem>();
        eventSystem.AddComponent<StandaloneInputModule>;
    }

    audioSource = gameObject.GetComponent<AudioSource>();
    if (audioSource == null)
    {
        audioSource = gameObject.AddComponent<AudioSource>();
    }
}

void Update()
{
    if (isPlayerInTrigger && !isSuccess &&
Input.GetKeyDown(KeyCode.F))
    {
        isCanvasActive = !isCanvasActive;
        if (EnterKeypadCanvas != null)
        {
            EnterKeypadCanvas.SetActive(isCanvasActive);
            if (isCanvasActive)
```

```
        {
            Cursor.lockState = CursorLockMode.None; // Free
cursor
            Cursor.visible = true; // show cursor
        }
        else
        {
            Cursor.lockState = CursorLockMode.Locked; // locked
cursor
            Cursor.visible = false;
        }
    }
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = true;
        if (!isSuccess)
        {
            interactionText.enabled = true;
        }
        else
        {
            interactionText.enabled = false;
            Debug.Log("Success achieved, no further interaction.");
        }
    }
}

private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = false;

        if (interactionText != null)
            interactionText.enabled = false;

        if (EnterKeypadCanvas != null && isCanvasActive)
        {
            EnterKeypadCanvas.SetActive(false); // Disabled canvas
with exit
            isCanvasActive = false;
            Cursor.lockState = CursorLockMode.Locked; // locked
cursor with exit
            Cursor.visible = false;
        }
    }
}
```

```
        Debug.Log("EnterKeypadCanvas deactivated on exit.");
    }
}

public void btn1()
{
    txtholder.text = txtholder.text + "1";
    PlayButtonClickSound();
}

public void btn2()
{
    txtholder.text = txtholder.text + "2";
    PlayButtonClickSound();
}

public void btn3()
{
    txtholder.text = txtholder.text + "3";
    PlayButtonClickSound();
}

public void btn4()
{
    txtholder.text = txtholder.text + "4";
    PlayButtonClickSound();
}

public void btn5()
{
    txtholder.text = txtholder.text + "5";
    PlayButtonClickSound();
}

public void btn6()
{
    txtholder.text = txtholder.text + "6";
    PlayButtonClickSound();
}

public void btn7()
{
    txtholder.text = txtholder.text + "7";
    PlayButtonClickSound();
}

public void btn8()
{
```

```
txtholder.text = txtholder.text + "8";
PlayButtonClickSound();
}

public void btn9()
{
    txtholder.text = txtholder.text + "9";
    PlayButtonClickSound();
}

public void btn0()
{
    txtholder.text = txtholder.text + "0";
    PlayButtonClickSound();
}

public void btnClear()
{
    txtholder.text = null;
    PlayButtonClickSound();
}

public void btnEnter()
{
    if (txtholder.text == "3825")
    {
        txtholder.text = "Success";
        PlaySuccessSound();
        isSuccess = true;
        if (EnterKeypadCanvas != null && isCanvasActive)
        {
            EnterKeypadCanvas.SetActive(false); // exit canvas after
success
            isCanvasActive = false;
            Cursor.lockState = CursorLockMode.Locked; // locked
cursor
            Cursor.visible = false;
            Debug.Log("Canvas closed after success.");
        }

        Stage2 stage2 = FindObjectOfType<Stage2>();
        if (stage2 != null)
        {
            stage2.CompleteStage();
            Debug.Log("Stage2 completed.");
        }
        else
        {
            Debug.LogWarning("Stage2 script not found in Scene!");
        }
    }
}
```

```
        }
    }
    else
    {
        txtholder.text = "Failed";
        PlayFailedSound();
        Invoke("ClearFailedText", 1.0f);
    }
}

private void PlayButtonClickSound()
{
    if (audioSource != null && buttonClickSound != null)
    {
        audioSource.PlayOneShot(buttonClickSound);
        Debug.Log("Button click sound played.");
    }
    else
    {
        Debug.LogWarning(" AudioSource or buttonClickSound is
missing!");
    }
}

private void PlaySuccessSound()
{
    if (audioSource != null && successSound != null)
    {
        audioSource.PlayOneShot(successSound);
        Debug.Log("Success sound played.");
    }
    else
    {
        Debug.LogWarning(" AudioSource or successSound is missing!");
    }
}

private void PlayFailedSound()
{
    if (audioSource != null && failedSound != null)
    {
        audioSource.PlayOneShot(failedSound);
        Debug.Log("Failed sound played.");
    }
    else
    {
        Debug.LogWarning(" AudioSource or failedSound is missing!");
    }
}
```

```

private void ClearFailedText()
{
    if (txtholder.text == "Failed")
    {
        txtholder.text = "";
        txtholder.placeholder.GetComponent<TextMeshProUGUI>().color =
Color.black;
        Debug.Log("Failed text cleared, ready for new input.");
    }
}
}

```

این اسکریپت در یونیتی برای تعامل با یک کیبورد عددی طراحی شده است که به بازیکن این امکان را می‌دهد تا یک رمز عبور را وارد کند. در ابتدا، متغیرهایی برای نگهداری ارجاع به اجزای مختلف مانند متنی که برای تعامل نمایش داده می‌شود (interactionText)، کانواسی که کیبورد روی آن نمایش داده می‌شود (EnterKeypadCanvas)، و دکمه‌های مختلف کیبورد عددی (button ۱ تا ۹button و سایر دکمه‌ها) تعریف شده‌اند. این اسکریپت به‌طور کلی شامل مدیریت ورودی‌های بازیکن از کیبورد، پخش صدا در هنگام تعامل، و تغییر وضعیت‌های مختلف است.

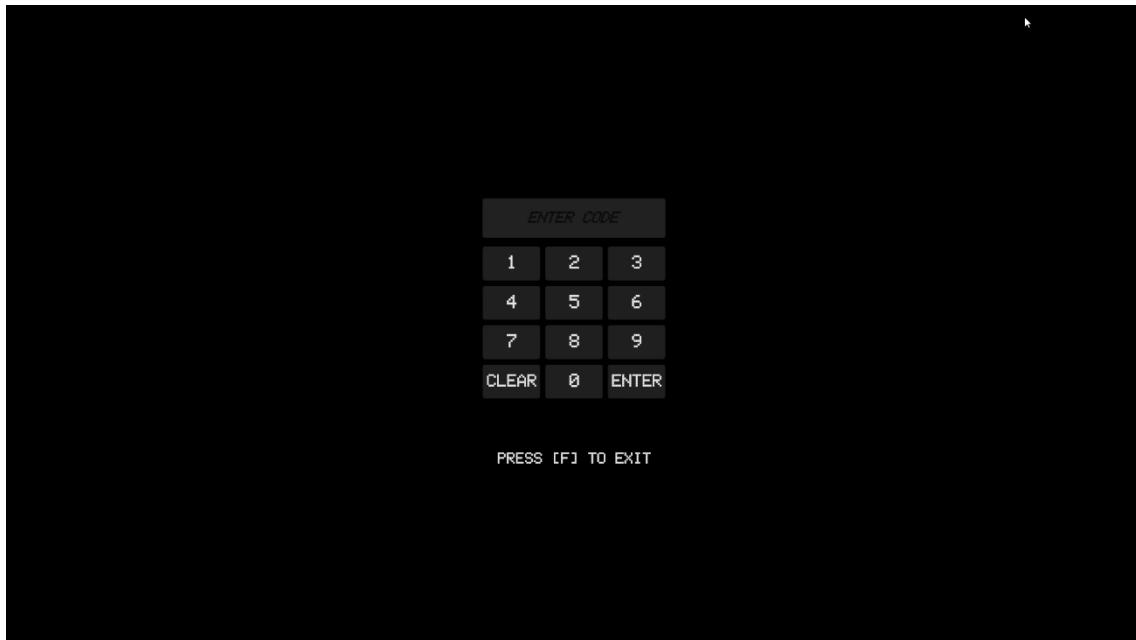
در متدهای Start و Update، ابتدا متنی که برای تعامل با بازیکن نمایش داده می‌شود غیرفعال می‌شود و رنگ پیش‌فرض برای placeholder تنظیم می‌شود. سپس، بررسی می‌شود که آیا کانواسی که کیبورد روی آن قرار دارد (EnterKeypadCanvas) وجود دارد یا نه و آن را غیرفعال می‌کند. همچنین، اگر سیستم رویداد (EventSystem) در صحنه وجود نداشته باشد، یک سیستم رویداد جدید برای مدیریت ورودی‌ها ایجاد می‌شود. در این مرحله، اطمینان حاصل می‌شود که همه چیز برای شروع تعاملات بازیکن آماده است.

در متدهای OnTriggerEnter و OnTriggerExit، زمانی که بازیکن وارد یا از تریگر خارج می‌شود، اجرا می‌شوند. در هنگام ورود به تریگر، متنی برای تعامل با بازیکن نمایش داده می‌شود و اگر موفقیت حاصل نشده باشد، بازیکن قادر به مشاهده دستورالعمل‌ها خواهد بود. در صورت موفقیت، این متن پنهان می‌شود. پس از خروج بازیکن از منطقه تریگر، تمامی اجزای مربوط به کیبورد غیرفعال می‌شوند تا از ایجاد مشکلات یا تعاملات اضافی جلوگیری شود.

در متدهای OnTriggerEnter و OnTriggerExit، زمانی که بازیکن وارد یا از تریگر خارج می‌شود، اجرا می‌شوند. در هنگام ورود به تریگر، متنی برای تعامل با بازیکن نمایش داده می‌شود و اگر موفقیت حاصل نشده باشد، بازیکن قادر به مشاهده دستورالعمل‌ها خواهد بود. در صورت موفقیت، این متن پنهان می‌شود. پس از خروج بازیکن از منطقه تریگر، تمامی اجزای مربوط به کیبورد غیرفعال می‌شوند تا از ایجاد مشکلات یا تعاملات اضافی جلوگیری شود.

دکمه‌های مختلف کیبورد عددی (از btn اول تا آخر) برای ثبت ورودی عددی بازیکن طراحی شده‌اند. زمانی که بازیکن یکی از این دکمه‌ها را فشار می‌دهد، عدد مربوطه به متن ورودی اضافه می‌شود و صدای کلیک دکمه پخش می‌شود تا بازخورد مناسبی به بازیکن ارائه شود. متدهای btnClear برای پاک کردن متن ورودی بازیکن از صفحه طراحی شده است تا بازیکن بتواند ورودی خود را اصلاح کند. همچنین، متدهای btnEnter برای بررسی صحت رمز عبور وارد شده استفاده می‌شود. اگر رمز صحیح باشد، پیامی با متن "Success" نمایش داده می‌شود و صدا موفقیت پخش می‌شود. در غیر این صورت، پیامی با متن "Failed" نمایش داده می‌شود و صدای خطا پخش می‌شود. سپس متن "Failed" پس از مدت کوتاهی پاک می‌شود تا بازیکن بتواند رمز جدیدی وارد کند.

در نهایت، متدهای PlayButtonClickSound، PlaySuccessSound و PlayFailedSound برای پخش صدای مختلف استفاده می‌شوند. این صدایها به بازیکن بازخورد مناسب را در هنگام تعامل با کیبورد می‌دهند و تجربه بازی را جذاب‌تر می‌کنند. متدهای ClearFailedText برای پاک کردن متن "Failed" پس از مدت زمان کوتاهی طراحی شده است، به‌طوری‌که رنگ placeholder به رنگ سیاه تغییر می‌کند و آماده برای ورودی جدید است. در مجموع، این اسکریپت با استفاده از ورودی‌های کیبورد، مدیریت نمایش و پنهان کردن کیبورد، و پخش صدای مختلف، تجربه‌ای تعاملی و جذاب برای بازیکن فراهم می‌آورد که به او این امکان را می‌دهد تا رمز عبور را وارد کرده و در صورت درست بودن رمز، مرحله‌ای جدید را در بازی تکمیل کند.



صفحه Canvas پد کلید



نزدیک شدن بازیکن به تریگر و باز کردن صفحه پد کلید

و در نهایت اسکریپت Stage دوم:

```
using UnityEngine;
using TMPro;

public class Stage2 : MonoBehaviour
{
    public TriggerDoor lockedDoor;
    public bool doorIsLocked = false;
    public GameObject knockTrigger;
    public AudioSource knockSound;
    private bool isStageCompleted = false;

    void Start()
    {
        if (lockedDoor != null)
        {
            lockedDoor.LockDoor();
            Debug.Log($"Door {lockedDoor.gameObject.name} locked for this
stage.");
        }
        else
        {
            Debug.LogWarning("No locked door assigned to this stage!");
        }
    }

    void OnTriggerEnter(Collider other)
```

```

{
    if (other.CompareTag("Player"))
    {
        knockSound.Play();
    }
}

public void CompleteStage()
{
    if (!isStageCompleted)
    {
        isStageCompleted = true;
        if (lockedDoor != null)
        {
            lockedDoor.UnlockDoor();
            Debug.Log($"Door {lockedDoor.gameObject.name} unlocked
after completing Stage2.");
        }
        GameObject.FindObjectOfType<GameManager>().StageCompleted();
        Debug.Log("Stage2 completed!");
    }
}
}

```

این اسکریپت برای مدیریت مرحله دوم بازی طراحی شده است. در ابتدا، متغیرهایی برای نگهداری ارجاع به در قفل شده (lockedDoor)، وضعیت قفل بودن در (doorIsLocked)، تریگر برای پخش صدای کوبیدن در (knockTrigger)، و صدای کوبیدن در (knockSound) تعریف شده‌اند.

همچنین متغیر isStageCompleted برای ردیابی وضعیت تکمیل شدن مرحله استفاده می‌شود. در متدهای Start()، ابتدا بررسی می‌شود که آیا در قفل شده به درستی به این اسکریپت اختصاص داده شده است یا نه. اگر اختصاص داده شده باشد، متدهای LockDoor برای قفل کردن در فراخوانی می‌شود و پیامی در کنسول چاپ می‌شود که نشان‌دهنده قفل شدن در برای این مرحله است. در صورتی که در قفل شده اختصاص داده نشده باشد، پیامی هشدار در کنسول نمایش داده می‌شود.

در متدهای OnTriggerEnter، زمانی که بازیکن وارد ناحیه تریگر می‌شود، صدای کوبیدن در پخش می‌شود تا بازیکن احساس کند که در حال آماده‌سازی برای باز شدن در است.

در متدهای CompleteStage، زمانی که مرحله تکمیل می‌شود (یعنی در صورتی که isStageCompleted برابر با false باشد)، وضعیت isStageCompleted را به true تغییر می‌کند و سپس در قفل شده باز می‌شود با فراخوانی متدهای UnlockDoor در شیء lockedDoor. پس از آن، متدهای StageCompleted از کلاس GameManager به نام فراخوانی می‌شود تا این مرحله به طور رسمی به عنوان تکمیل شده علامت‌گذاری شود. پیامی نیز در کنسول چاپ می‌شود که

نشان دهنده تکمیل شدن مرحله است.

این اسکریپت به طور کلی مدیریت قفل و باز شدن در مرحله دوم بازی را انجام می‌دهد و با پخش صدای کوبیدن در، حس تعليق و انتظار را برای بازيکن ايجاد می‌کند. همچنین با تکمیل شدن مرحله، در باز می‌شود و پیشرفت بازی ثبت می‌شود.

۴-۲-۸ - پیاده سازی Stage سوم

در مرحله‌ی سوم که بلافاصله پس از باز شدن در و تکمیل مرحله‌ی دوم آغاز می‌شود، بازيکن با يك صحنه‌ی ترسناک مواجه می‌شود.

در آغاز اين مرحله، مجموعه‌ای از نورهای موجود در فضای پذيرايی به صورت آرایه‌ای از طریق اسکریپت فراخوانی می‌شوند و رنگ آن‌ها به قرمز تغيير می‌يابد. همزمان، يك صدای ترسناک نيز پخش می‌شود تا حس اضطراب و تنفس را افزایش دهد.

در اين مرحله، يك شئ بازی به نام Threat وجود دارد که شامل يك کاراكتر تهدیدکننده و دو Trigger است. پس از فعال شدن اين Trigger‌ها، زمانی که بازيکن وارد محدوده آن‌ها می‌شود، کاراكتر تهدیدکننده وارد صحنه می‌شود، و بلافاصله پس از آن نورها به حالت سفید بازمی‌گردند. پس از پخش يك افکت صوتی نهايی، مرحله‌ی سوم نيز به پايان می‌رسد.

در ابتدا اسکریپت TriggerThreat که برای فعال شدن نمايش ترسناک و تمام شدن آن در محدوده تريگر‌ها است:

```
using UnityEngine;

public class TriggerThreat : MonoBehaviour
{
    public GameObject Threat;
    private Stage3 stage3; //Refrence to Stage3

    void Start()
    {
        stage3 = FindObjectOfType<Stage3>();
        if (stage3 == null)
        {
            Debug.LogWarning("Stage3 script not found in Scene!");
        }
    }
}
```

```
void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        if (Threat != null && Threat.activeSelf)
        {
            if (stage3 != null)
            {
                stage3.SetThreatTriggered(true);
            }
            else
            {
                Debug.LogWarning("Stage3 reference not found!");
            }
        }

        Threat.SetActive(false);
        Debug.Log($"Threat {Threat.name} deactivated.");
    }
    else if (Threat != null)
    {
        Debug.LogWarning("Threat is not active!");
    }
    else
    {
        Debug.LogWarning("Threat GameObject is not assigned!");
    }
}
else
{
    Debug.LogWarning($"Object {other.gameObject.name} is not
tagged as Player!");
}
```

این اسکریپت به منظور مدیریت تهدید در بازی طراحی شده است که زمانی که بازیکن وارد ناحیه خاصی می‌شود، یک تهدید فعال (مانند یک دشمن یا موجود ترسناک) غیرفعال می‌شود و همچنین وضعیت تهدید در مرحله سوم بازی تغییر می‌کند.

در این اسکریپت، متغیر Threat برای نگهداری ارجاع به شیء تهدید (که باید یک GameObject فعال باشد) و متغیر stage برای نگهداری ارجاع به اسکریپت Stage استفاده می‌شود. در ابتدا، در متدهای Start و Update اسکریپت Stage از صحنه پیدا می‌شود و در صورتی که وجود نداشته باشد، یک پیام هشدار در کنسول چاپ می‌شود.

در متدهای OnTriggerEnter، زمانی که بازیکن وارد ناحیه تریگر می‌شود، ابتدا بررسی می‌شود که آیا شیء Threat فعال است یا نه. اگر تهدید فعال باشد، وضعیت تهدید در اسکریپت Stage3 به تنظیم می‌شود (با استفاده از متدهای SetThreatTriggered(true) و SetThreatTriggered(false)). همچنین پیامی در کنسول نمایش داده می‌شود غیرفعال می‌شود (Threat.SetActive(false)). که نشان می‌دهد تهدید غیرفعال شده است.

اگر شیء تهدید فعال نباشد، یک پیام هشدار در کنسول چاپ می‌شود که نشان می‌دهد تهدید در حال حاضر فعال نیست. همچنین اگر شیء تهدید به اسکریپت اختصاص داده نشده باشد، پیامی دیگر چاپ می‌شود که نشان‌دهنده عدم اختصاص آن است.

در صورتی که شیء برخورد کرده با تریگر، تگ "Player" نداشته باشد، پیامی هشدار در کنسول نمایش داده می‌شود که نشان می‌دهد شیء برخورد کرده بازیکن نبوده است.

این اسکریپت به طور کلی به منظور ایجاد تغییر در وضعیت تهدید در بازی و تعامل آن با مرحله سوم طراحی شده است. به عنوان مثال، می‌توان از آن برای مدیریت دشمن‌ها یا موجودات ترسناک در بازی استفاده کرد که پس از ورود بازیکن به یک منطقه خاص غیرفعال می‌شوند.

سپس اسکریپت Stage3 :

```
using UnityEngine;

public class Stage3 : MonoBehaviour
{
    public Light[] lights;
    public GameObject threat;
    public AudioSource Horrorsound;
    private bool isStageCompleted = false;
    private bool isThreatTriggered = false;
    private bool soundPlayed = false;

    void Start()
    {
        if (lights == null || lights.Length == 0)
        {
            Debug.LogWarning("Lights array is null or empty! Please assign lights in Inspector.");
            return;
        }

        gameObject.SetActive(false);
    }
}
```

```
        Debug.Log("Stage3 deactivated, waiting for GameManager.");
    }

    public void ActivateStage()
    {
        if (!gameObject.activeSelf)
        {
            gameObject.SetActive(true);
            Debug.Log("Stage3 activated successfully.");
        }
        else
        {
            Debug.LogWarning("Stage3 was already active!");
        }
        ChangeLightColors(); // change color B81B1B
        ActivateThreat();
    }

    private void ChangeLightColors()
    {
        if (lights == null || lights.Length == 0)
        {
            Debug.LogError("No lights assigned to change colors in Stage3!");
            return;
        }

        Color targetColor = new Color(0.722f, 0.106f, 0.106f); //B81B1B
        foreach (Light light in lights)
        {
            if (light != null)
            {
                light.color = targetColor;
                Debug.Log($"Changed light {light.name} color to {targetColor} in Stage3");
            }
            else
            {
                Debug.LogWarning("Null light reference found in lights array!");
            }
        }
    }

    private void ActivateThreat()
    {
        if (threat != null)
        {
```

```
        threat.SetActive(true);
        Debug.Log($"Threat {threat.name} activated in Stage3.");
    }
    else
    {
        Debug.LogWarning("Threat GameObject is not assigned in
Inspector!");
    }
}

public void SetThreatTriggered(bool value)
{
    isThreatTriggered = value;
    if (isThreatTriggered && !isStageCompleted && !soundPlayed)
    {
        TurnLightsWhite();
        PlaySound();
    }
}

private void TurnLightsWhite()
{
    if (lights != null && lights.Length > 0)
    {
        foreach (Light light in lights)
        {
            if (light != null)
            {
                light.color = Color.white;
                Debug.Log($"Changed light {light.name} color to white
before sound.");
            }
        }
    }
    else
    {
        Debug.LogWarning("No lights assigned to change to white!");
    }
}

private void PlaySound()
{
    if (Horrorsound != null && Horrorsound.clip != null)
    {
        Horrorsound.Play();
        soundPlayed = true;
        Debug.Log("Horror sound started playing in Stage3.");
    }
    else
```

```

    {
        Debug.LogError("Horrorsound or its AudioClip is not assigned
in Stage3!");
    }
}

private void CheckCompletion()
{
    if (!isStageCompleted && isThreatTriggered && soundPlayed)
    {
        if (Horrorsound != null && !Horrorsound.isPlaying && threat
!= null && !threat.activeSelf)
        {
            isStageCompleted = true;
            Debug.Log("Stage3 completed! Threat deactivated.");
            GameObject.FindObjectOfType<GameManager>()?.StageComplete
d();
        }
    }
}

void Update()
{
    CheckCompletion();
}
}

```

این اسکریپت برای مدیریت مرحله سوم بازی طراحی شده است و شامل تغییرات در نورپردازی،
فعال‌سازی تهدید و پخش صدای ترسناک می‌شود. هدف اصلی این اسکریپت ایجاد یک تجربه
ترسناک و دلهره‌آور است که پس از فعال شدن تهدید، نورهای محیط تغییر کرده، صدای ترسناک
پخش می‌شود و در نهایت مرحله به عنوان تکمیل شده علامت‌گذاری می‌شود. در ابتدا، این اسکریپت
یک آرایه از نورها (lights)، شیء تهدید (threat) و صدای ترسناک (Horrorsound) را نگهداری
می‌کند و متغیرهایی برای پیگیری وضعیت‌های مختلف مانند تکمیل بودن مرحله
در متد (isStageCompleted)، فعال بودن تهدید (isThreatTriggered) و پخش شدن صدا
(soundPlayed) ایجاد می‌کند. در متد Start()، اگر آرایه نورها خالی یا نادرست باشد، یک هشدار
چاپ می‌شود. سپس، مرحله سوم غیرفعال می‌شود تا زمانی که به‌طور خاص توسط اسکریپت دیگری
فعال شود.

در متد ActivateStage، مرحله سوم فعال می‌شود، رنگ نورها تغییر می‌کند (به رنگ قرمز خاصی
با کد B1B81B)، و تهدید (که می‌تواند یک دشمن یا موجود ترسناک باشد) فعال می‌شود. متد
برای تغییر رنگ نورها به رنگ قرمز طراحی شده است که جو دلهره‌آور و ChangeLightColors

ترسناکی ایجاد می‌کند. در همین زمان، متدهای `ActivateThreat` و `SetThreatTriggered` بررسی می‌کند آن روبرو شود. سپس، زمانی که تهدید فعال می‌شود، متدهای `TurnLightsWhite` و `PlaySound` پخش صدای ترسناک به سفید تبدیل می‌شوند تا فضای بازی نورها به رنگ سفید تغییر می‌کند و صدا پخش می‌شود که آیا تهدید فعال است یا نه و بر اساس آن، نورها به رنگ سفید تغییر می‌کنند و صدا پخش می‌شود تا فضای بازی پر از تنفس و هیجان شود.

متدهای `TurnLightsWhite` و `PlaySound` مسئول تغییر رنگ نورها به سفید است و در این مرحله، نورها قبل از پخش صدای ترسناک به سفید تبدیل می‌شوند تا فضای بازی آماده برای وقوع یک اتفاق وحشتناک شود. متدهای `PlaySound` برای پخش صدای ترسناک استفاده می‌شود و زمانی که شرایط آن فراهم باشد (یعنی صدا و کلیپ آن به درستی تنظیم شده باشند)، صدای ترسناک پخش می‌شود و متغیر `soundPlayed` به `true` تغییر می‌کند.

در نهایت، متدهای `CheckCompletion` و `StageCompleted` هر فریم وضعیت تکمیل شدن مرحله را بررسی می‌کند. اگر تهدید فعال شده و صدا پخش شده باشد و تهدید غیرفعال شده باشد، مرحله به عنوان تکمیل شده علامت‌گذاری می‌شود و از طریق متدهای `GameManager` در اسکریپت `StageCompleted` این پیشرفت ثبت می‌شود. این اسکریپت به طور کلی برای ایجاد لحظات هیجان‌انگیز و ترسناک در بازی‌های ترسناک بسیار مؤثر است، زیرا با استفاده از تغییرات در نورپردازی، صدا و تهدیدات محیطی، تنفس و اضطراب را در بازیکن ایجاد می‌کند و حس ترس را به طور مؤثری تقویت می‌کند.



تهدید و نمایش ترسناک با کرکتر کابوس

۴-۲-۹ - پیاده سازی Sateg چهارم

در این بخش، زمانی که بازیکن وارد اتفاق می‌شود، با نوشتہ‌ای مواجه می‌شود با این مضمون که "تدى می‌خواهد با تو بازی کند." در ادامه، بازیکن شروع به جستجوی اتفاق می‌کند و در نهایت یک عروسک تدى ترسناک را درون کمد پیدا می‌کند. زمانی که بازیکن با فشردن کلید F، تدى را برمی‌دارد، یک Timeline اجرا می‌شود که یک سکانس سینمایی کوتاه را نمایش می‌دهد. پس از پایان این Timeline، بازیکن به صحنه‌ی بعدی (Scene) با عنوان TeddyScene منتقل می‌شود.

ابتدا اسکریپت : TeddyController

```
using UnityEngine;
using TMPro;

public class TeddyController : MonoBehaviour
{
    public TMP_Text interactionText;
    public GameObject teddytalk;
    private bool isPlayerInTrigger = false;
    private Stage4 stage4;
    private bool hasTimelineStarted = false;

    void Start()
    {

        if (interactionText != null)
        {
            interactionText.enabled = false;
            Debug.Log("interactionText initialized and disabled.");
        }
        else
        {
            Debug.LogWarning("interactionText is not assigned in Inspector!");
        }

        if (teddytalk == null)
        {
            Debug.LogError("teddytalk is not assigned in Inspector!");
        }

        // Reference to Stage4
        stage4 = FindObjectOfType<Stage4>();
        if (stage4 == null)
        {
            Debug.LogError("Stage4 script not found in the scene!");
        }
    }
}
```

```

        }

    }

    void Update()
    {
        if (isPlayerInTrigger && !hasTimelineStarted)
        {
            if (Input.GetKeyDown(KeyCode.F))
            {
                if (teddytalk != null && stage4 != null)
                {
                    teddytalk.SetActive(true);
                    stage4.StartTimeline();
                    hasTimelineStarted = true;
                }
            }
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInTrigger = true;
            if (interactionText != null)
            {
                interactionText.enabled = true;
            }
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInTrigger = false;
            if (interactionText != null)
            {
                interactionText.enabled = false;
            }
        }
    }
}

```

هدف اصلی این اسکریپت این است که بازیکن وارد ناحیه خاصی می‌شود، پیامی برای تعامل با عروسک تدی نمایش داده شود و پس از فشار دادن کلید F، یک رویداد خاص (مانند شروع تایم‌لاین) آغاز شود.

در ابتداء، متغیرهایی برای نگهداری ارجاع به متن تعامل (interactionText)، گیم‌اجکت گفتگو با تدی (teddytalk)، و همچنین ارجاع به اسکریپت ۴Stage تعريف شده‌اند. در متدهای Start و teddytalk، بررسی می‌شود که آیا interactionText به درستی تنظیم شده است یا نه و آن را غیرفعال می‌کند. همچنین، اگر teddytalk تنظیم نشده باشد، یک پیام خطا در کنسول چاپ می‌شود. سپس، ارجاع به اسکریپت ۴Stage می‌شود و اگر این اسکریپت در صحنه وجود نداشته باشد، یک پیام خطا چاپ می‌شود.

در متدهای Update و hasTimelineStarted، اگر بازیکن در ناحیه تریگر باشد و تایم‌لاین هنوز شروع نشده باشد، بازیکن می‌تواند با فشار دادن کلید F، گفتگو با تدی را فعال کند. پس از فعال شدن گفتگو، تایم‌لاین مرحله چهارم (stage4.StartTimeline) شروع می‌شود و متغیر true به تغییر می‌کند تا از تکرار رویداد جلوگیری شود.

در متدهای OnTriggerEnter و OnTriggerExit، زمانی که بازیکن وارد یا از ناحیه تریگر خارج می‌شود، پیامی برای تعامل با عروسک تدی نمایش داده می‌شود یا مخفی می‌شود. این پیام فقط زمانی نمایش داده می‌شود که بازیکن در ناحیه تریگر باشد.

به طور کلی، این اسکریپت برای ایجاد تعاملات خاص با شخصیت تدی طراحی شده است که بازیکن می‌تواند با آن وارد تعامل شود و بر اساس فشار دادن کلید F، یک رویداد جدید در بازی آغاز می‌شود.

سپس اسکریپت stag4 :

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Stage4 : MonoBehaviour
{
    private bool isStageCompleted = false;
    private bool isTimelinePlaying = false;
    private float timer = 0f;
    public Camera mainCamera; // Reference to main camera
    public TriggerDoor lockedDoor;

    void Start()
    {
        gameObject.SetActive(false);
        Debug.Log("Stage4 deactivated, waiting for GameManager.");
        if (lockedDoor != null)
    }
}
```

```
        lockedDoor.LockDoor();
        Debug.Log($"Door {lockedDoor.gameObject.name} locked for this
stage.");
    }
    else
    {
        Debug.LogWarning("No locked door assigned to this stage!");
    }
}

public void ActivateStage()
{
    if (!gameObject.activeSelf)
    {
        gameObject.SetActive(true);
        Debug.Log("Stage4 activated successfully.");
    }
    else
    {
        Debug.LogWarning("Stage4 was already active!");
    }
}

public void StartTimeline()
{
    isTimelinePlaying = true;
    timer = 0f;
    Debug.Log("Timeline started in Stage4.");
}

void Update()
{
    if (isTimelinePlaying && !isStageCompleted)
    {
        timer += Time.deltaTime;
        if (timer >= 14f)
        {
            mainCamera.enabled = false;
            Debug.Log("Main Camera disabled before scene
transition.");
            isTimelinePlaying = false;
            isStageCompleted = true;
            Debug.Log("Stage4 completed!");
            SceneManager.LoadScene("TeddyScene");
            GameObject.FindObjectOfType<GameManager>()?.StageComplete
d();
        }
    }
}
```

{
}

این اسکریپت برای مدیریت مرحله چهارم بازی طراحی شده است و به طور خاص برای فعال‌سازی یک تایم‌لاین و انجام برخی تغییرات در محیط بازی مانند غیرفعال کردن دوربین اصلی و انتقال به صحنۀ جدید پس از اتمام تایم‌لاین است. در ابتدا، مرحله چهارم غیرفعال می‌شود تا از آن تنها زمانی استفاده شود که اسکریپت‌های دیگر آن را فعال کنند. همچنین، در صورتی که درب قفل شده برای این مرحله وجود داشته باشد، درب قفل می‌شود. سپس، در متده Activestate، اگر مرحله قبل‌اً فعال نشده باشد، آن را فعال می‌کند. در متده StartTimeline، تایم‌لاین شروع می‌شود و یک تایمر برای پیگیری مدت زمان تایم‌لاین فعال می‌شود. در طول اجرای بازی، در متده Update، تایم‌لاین به‌طور پیوسته با استفاده از تایمر پیگیری می‌شود و زمانی که تایم‌لاین به مدت مشخصی می‌رسد (۱۴ ثانیه)، دوربین اصلی غیرفعال می‌شود تا فضایی برای تغییر صحنۀ فراهم شود. در نهایت، مرحله به عنوان تکمیل شده اعلام می‌شود و با استفاده از ("SceneManager.LoadScene("TeddyScene")" بازی به صحنۀ جدید منتقل می‌شود. این اسکریپت به‌طور مؤثر فرآیند تکمیل مرحله چهارم را مدیریت کرده و تجربه بازی را با استفاده از تایم‌لاین و تغییرات محیطی تقویت می‌کند.



عروسوک تدی

۴-۲-۱ - پیاده سازی Stage پنجم

در مرحله‌ی پنجم، بازیکن وارد محیطی کاملاً جدید و متفاوت می‌شود. این بخش با صحنه‌ای آغاز می‌شود که در آن، بازیکن نفس‌زنان در اتاقی کوچک به هوش می‌آید. بر روی دیوار این اتاق، جمله‌ای با مضمون "نترس، فقط بدرخش" نوشته شده و یک چراغ قوه نیز روی زمین قرار دارد. بازیکن می‌تواند با فشردن کلید F، چراغ قوه را بردارد و پس از آن وارد اتاقی بزرگ و تاریک می‌شود که در آن چندین روح ساکن و بی‌حرکت در نقاط مختلف ایستاده‌اند. در حین پیشروی، ناگهان یک شخصیت ترسناک (کابوس) شروع به تعقیب بازیکن می‌کند.

در این حالت، تنها راه نجات بازیکن استفاده صحیح از نور چراغ قوه است؛ اگر نور چراغ قوه به سمت این کابوس گرفته شود، از نزدیک شدن آن جلوگیری می‌شود. اما اگر کابوس در تاریکی باقی بماند و نوری به آن نتابد، به تعقیب بازیکن ادامه می‌دهد. در صورتی که به بازیکن نزدیک شود، بازی‌کننده بازنده خواهد بود و باید این مرحله را از ابتدا طی کند.

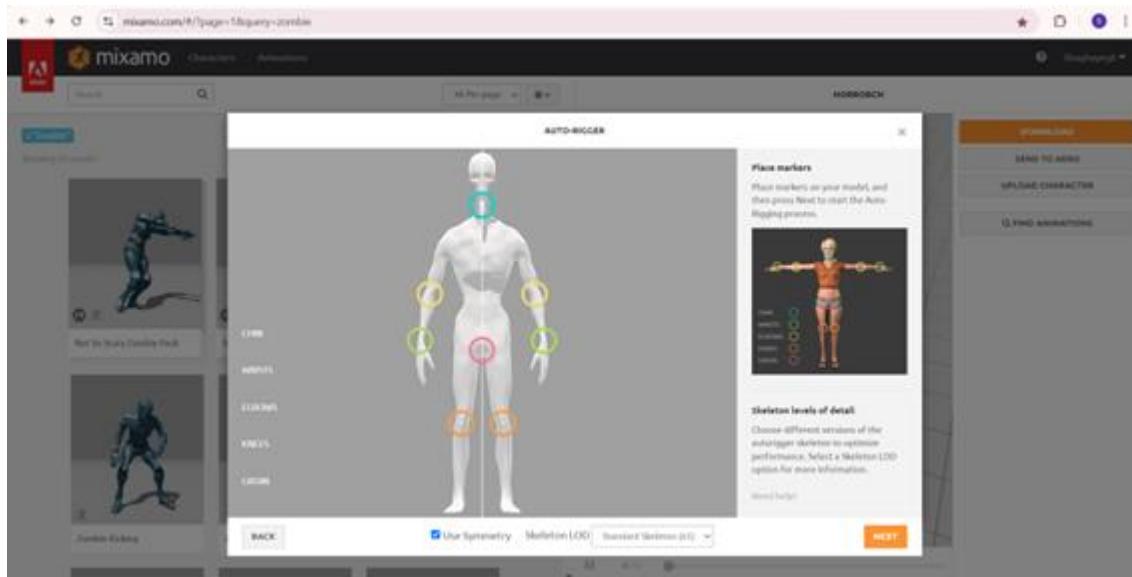
در صورتی که بازیکن موفق شود از دست کابوس فرار کرده و خود را به اتاق کوچکی که در انتهای فضای تاریک قرار دارد برساند، وارد منطقه‌ی امن می‌شود. در این اتاق، عروسک تدی مجدد روی زمین دیده می‌شود. با برداشتن تدی، بازیکن دوباره به اتاق خانه‌ی خود بازمی‌گردد و مرحله پایان می‌یابد.

■ ساخت انیمیشن کابوس

برای شخصیت کابوس در این مرحله، از انیمیشن‌های آماده‌ی سایت معتبر Mixamo استفاده شده است. سه انیمیشن مجزا برای این کاراکتر در نظر گرفته شده که هر یک متناسب با وضعیت‌های مختلف او در جریان بازی به کار می‌روند:

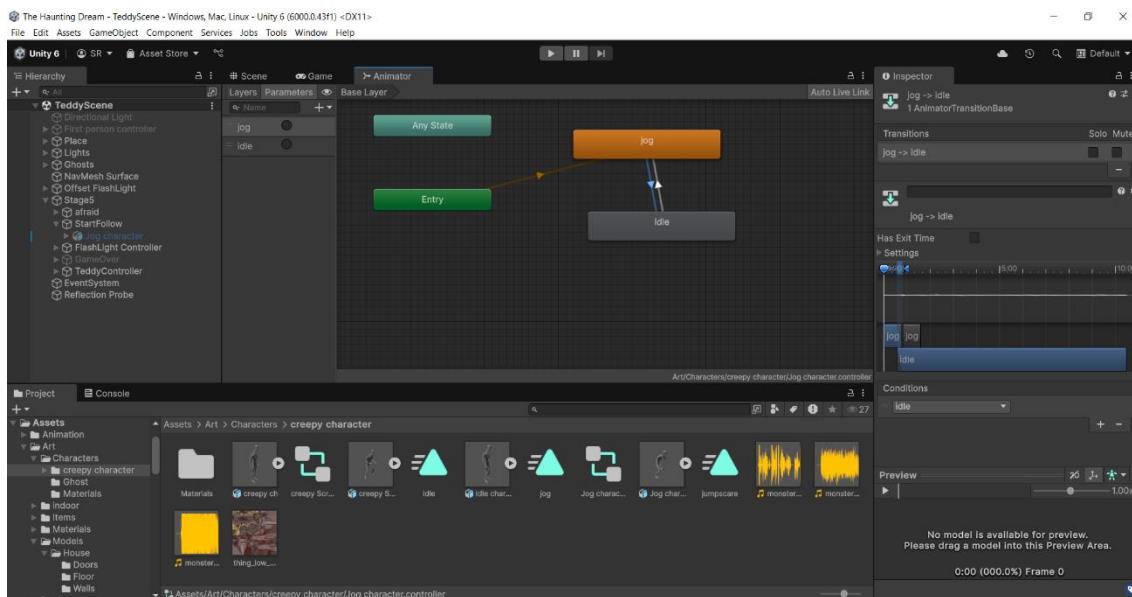
۱. انیمیشن ایستادن (Idle): زمانی که کابوس هنوز فعال نشده و در حالت ساکن قرار دارد.
۲. انیمیشن حرکت (Walking/Running): هنگام تعقیب بازیکن در محیط تاریک.
۳. انیمیشن فریاد (Screaming): در لحظه‌هایی که کابوس به بازیکن نزدیک می‌شود یا شکست بازیکن در بازی رقم می‌خورد.

این انیمیشن‌ها به صورت پیوسته و با منطق رفتاری کابوس در موتور Unity و سیستم Animator می‌شوند تا تجربه‌ای واقع‌گرایانه و تنشزا برای بازیکن فراهم گردد.



Mixamo ساخت انیمیشن کرکتر در سایت

پس از دانلود و آماده‌سازی انیمیشن‌ها، آن‌ها را وارد محیط Unity می‌کنیم. در این مرحله، انیمیشن‌های حرکتی به شخصیت کابوس اضافه می‌شوند. برای انجام این کار، ابتدا کامپوننت Animator را به کارکتر اختصاص می‌دهیم و سپس انیمیشن‌ها (ایستادن، حرکت، و فریاد) را در کنترلر Animator تعریف و مدیریت می‌کنیم.



تعریف انیمیشن برای کرکتر کابوس

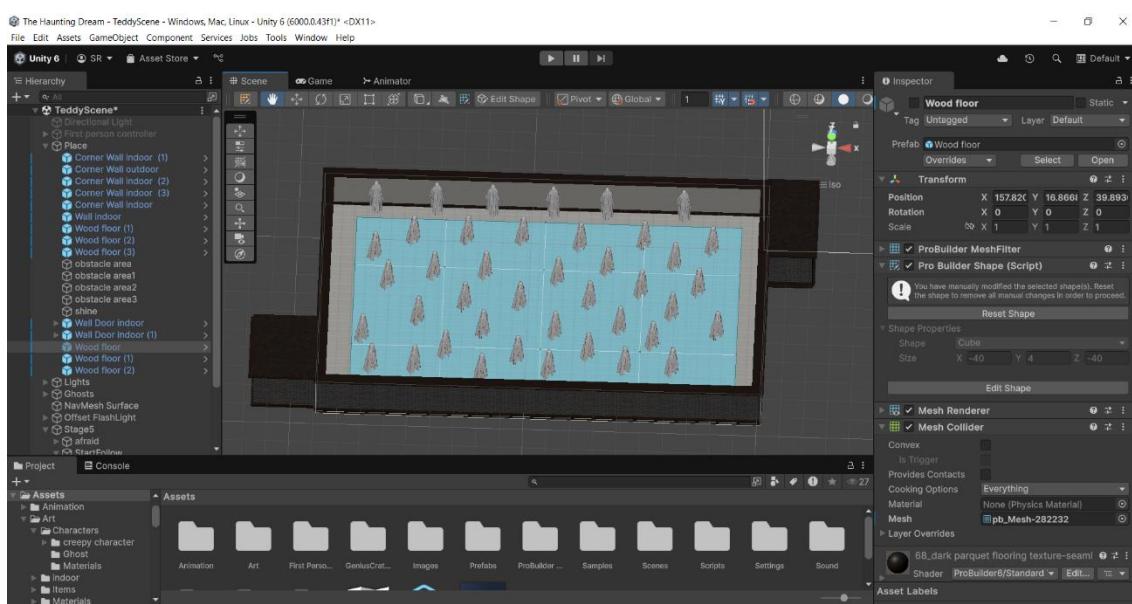
□ استفاده از هوش مصنوعی برای تعقیب

برای پیاده‌سازی رفتار هوشمندانه تعقیب بازیکن، کامپوننت NavMesh Agent از سیستم هوش مصنوعی Unity به کارکتر کابوس افزوده می‌شود. این کامپوننت امکان مسیریابی خودکار و تعقیب

هدف در محیط را فراهم می‌سازد.

همچنین به منظور مشخص کردن محدوده‌ی مجاز حرکت کابوس در محیط بازی، کامپوننت NavMesh Surface بر روی سطح مناسب از محیط اعمال می‌شود تا مسیرهای قابل پیمایش تولید گردد.

در نهایت، برای جلوگیری از عبور کابوس از موانع خاص (مانند دیوارها، روح‌های در صحنه)، از کامپوننت NavMesh Obstacle استفاده می‌شود. این کامپوننت مناطقی را تعریف می‌کند که کابوس نمی‌تواند از آن‌ها عبور کند، در نتیجه رفتارش طبیعی‌تر و محدودتر خواهد بود.



تعیین محدوده حرکت کرکتر کابوس

□ پیاده‌سازی سیستم چراغ قوه

در ابتدا، یک GameObject با نام Spot ایجاد می‌کنیم و درون آن یک Light قرار می‌دهیم. این نورافکن به گونه‌ای تنظیم می‌شود که زاویه‌ی تابش آن مشابه نور چراغ قوه باشد. سپس OffsetFlashlight را در موقعیتی نزدیک به دوربین اصلی (Main Camera) قرار می‌دهیم، به طوری که گویی چراغ قوه در دست بازیکن قرار دارد و هم‌زمان با دید او حرکت می‌کند. برای کنترل عملکرد چراغ قوه (روشن/خاموش شدن یا دیگر ویژگی‌ها)، یک اسکریپت اختصاصی نوشته می‌شود که به GameObject مربوطه متصل می‌گردد.

اسکریپت OffsetFlashLight برای چراغ قوه:

```
using UnityEngine;
```

```

public class OffsetFlashLight : MonoBehaviour
{
    private Vector3 OffsetVector3;
    public GameObject FollowCam;
    [SerializeField] private float MoveSpeed=13f;
    public Light FlashLight;
    private bool FlashLightIsOn=false;

    void Start()
    {
        OffsetVector3=transform.position-FollowCam.transform.position;
    }

    void Update()
    {
        transform.position=FollowCam.transform.position+OffsetVector3;
        transform.rotation=Quaternion.Slerp(transform.rotation,
FollowCam.transform.rotation, MoveSpeed* Time.deltaTime);

        FlashLight.enabled=true;
        FlashLightIsOn=true;
    }
}

```

این اسکریپت برای مدیریت یک فلاش‌لایت طراحی شده است که به‌طور همزمان موقعیت آن با دوربین بازی (که در اینجا FollowCam نامیده می‌شود) تنظیم می‌شود و فلاش‌لایت همیشه روشن باقی می‌ماند. در ابتدا، یک بردار فاصله (OffsetVector^۳) بین موقعیت فلاش‌لایت و دوربین محاسبه می‌شود تا فلاش‌لایت همیشه در یک موقعیت ثابت نسبت به دوربین قرار گیرد. این بردار در متدهای `Start()` و `Update()` تعیین می‌شود. سپس، در متدهای `MoveSpeed` و `FlashLightIsOn` به صورت پیوسته به موقعیت دوربین و با استفاده از `Quaternion.Slerp` تنظیم می‌شود تا فلاش‌لایت همچنان در همان فاصله نسبت به دوربین قرار گیرد.

همچنین، چرخش فلاش‌لایت با استفاده از متدهای `FlashLight.enabled` و `FlashLight.intensity` می‌تواند تغییر کند. در نهایت، فلاش‌لایت روشن می‌شود (با تنظیم `FlashLightIsOn` به `true`) و تغییر می‌کند تا مشخص شود فلاش‌لایت روشن است. به‌طور کلی، این اسکریپت برای دنبال کردن دوربین با فلاش‌لایت و اطمینان از روشن بودن آن به‌طور مداوم در بازی طراحی شده است.

□ پیاده سازی مرحله پنجم

در ابتدای مرحله‌ی پنجم، بازیکن در فضایی پرتنش و همراه با نفس‌نفس‌زدن، خود را در یک اتاق کوچک می‌بیند. این صحنه توسط یک GameObject با نام Afraid پیاده‌سازی شده است که دارای یک آنیمیشن نمایشی و کامپونت Animator است. در این بخش، یک دوربین فرعی (Secondary Camera) وجود دارد که حرکات آن از پیش در قالب آنیمیشن طراحی شده و برای ایجاد یک فضای سینمایی و غوطه‌ورکننده به کار می‌رود.

در طول اجرای این آنیمیشن، بازیکن کنترلی بر حرکت ندارد و صرفاً شاهد صحنه است. پس از پایان یافتن آنیمیشن دوربین، دوربین اصلی (Main Camera) فعال می‌شود و کنترل دوباره به بازیکن بازمی‌گردد، به‌طوری که از آن لحظه به بعد، بازیکن می‌تواند در محیط حرکت کرده و با عناصر بازی تعامل داشته باشد. اسکریپت Afraid :

```
using UnityEngine;

public class Afraid : MonoBehaviour
{
    public GameObject afraid;
    public GameObject firstPersonController;

    void Start()
    {
        if (afraid == null)
        {
            Debug.LogError("Afraid reference is not assigned!");
            return;
        }
        afraid.SetActive(true);
        Animator animator = afraid.GetComponent<Animator>();
        if (animator == null)
        {
            Debug.LogError("Animator component not found on Afraid!");
            return;
        }

        AnimatorStateInfo stateInfo =
animator.GetCurrentAnimatorStateInfo(0);
        float animationLength = stateInfo.length > 0 ? stateInfo.length :
2f;
        Invoke("OnStageCompleted", animationLength);

        if (firstPersonController != null)
        {
            firstPersonController.SetActive(false);
        }
    }

    void OnStageCompleted()
    {
        if (afraid != null)
        {
            afraid.SetActive(true);
        }
    }
}
```

```

        }
    else
    {
        Debug.LogError("FirstPersonController reference is not
assigned!");
    }
}

void OnStageCompleted()
{
    if (afraid != null) afraid.SetActive(false);
    if (firstPersonController != null)
firstPersonController.SetActive(true);
}
}

```

این اسکریپت برای مدیریت وضعیت ترس در بازی طراحی شده است. در ابتدا، بررسی می‌شود که آیا ارجاع به شیء `afraid` به درستی تنظیم شده است یا خیر. اگر این ارجاع تنظیم نشده باشد، پیامی در کنسول چاپ می‌شود و اسکریپت متوقف می‌شود. اگر ارجاع صحیح باشد، شیء `afraid` فعال می‌شود و سپس یک کامپوننت `Animator` برای مدیریت انیمیشن‌ها بررسی می‌شود. اگر کامپوننت `afraid` وجود نداشته باشد، پیامی در کنسول چاپ می‌شود و اسکریپت متوقف می‌شود.

سپس، متغیر `AnimatorStateInfo` برای دریافت اطلاعات مربوط به انیمیشن جاری استفاده می‌شود. طول انیمیشن از این اطلاعات استخراج شده و پس از آن، با استفاده از متده `Invoke`، متده `OnStageCompleted` پس از پایان انیمیشن فراخوانی می‌شود. این متده به‌طور خودکار پس از پایان انیمیشن اجرا می‌شود.

در ادامه، اگر ارجاع به `firstPersonController` تنظیم شده باشد، این شیء غیرفعال می‌شود تا بازیکن نتواند به کنترل شخصیت اول شخص دسترسی داشته باشد. اگر ارجاع به `firstPersonController` تنظیم نشده باشد، پیامی در کنسول چاپ می‌شود که نشان‌دهنده عدم تخصیص این شیء است.

در نهایت، در متده `OnStageCompleted`، زمانی که انیمیشن ترس به پایان می‌رسد، شیء `afraid` غیرفعال می‌شود و شیء `firstPersonController` دوباره فعال می‌شود تا بازیکن کنترل را به‌دست گیرد. به‌طور کلی، این اسکریپت برای ایجاد یک لحظه ترس در بازی طراحی شده است که در آن، شخصیت بازیکن از دسترسی به کنترل خود محروم می‌شود و یک انیمیشن که بازیکن ترسیده پخش می‌شود. پس از اتمام انیمیشن، کنترل به بازیکن بازمی‌گردد.



حین اجرای انیمیشن

بعد از تمام شدن انیمیشن بازیکن باید چراغ قوه را بردارد و روشن کند. اسکریپت که وقتی بازیکن نزدیک چراغ قوه شود با زدن کلید F می‌تواند آن را بردارد و روشن کند:

```
using UnityEngine;
using TMPro;

public class FlashLightController : MonoBehaviour
{
    public TMP_Text interactionText;
    public AudioClip pickupSound;
    public GameObject objectToActivate;
    private AudioSource audioSource;
    private bool isPlayerInTrigger = false;

    void Start()
    {
        audioSource = gameObject.GetComponent<
```

```
{  
    Debug.LogWarning("interactionText is not assigned in  
Inspector!");  
}  
  
if (audioSource != null)  
{  
    audioSource.volume = 1.0f;  
    audioSource.playOnAwake = false;  
}  
else  
{  
    Debug.LogWarning(" AudioSource not set for FlashlightPickup on  
" + gameObject.name);  
}  
  
if (pickupSound == null)  
{  
    Debug.LogWarning("pickupSound not set for FlashlightPickup on  
" + gameObject.name);  
}  
  
if (objectToActivate == null)  
{  
    Debug.LogWarning("objectToActivate is not assigned in  
Inspector!");  
}  
else  
{  
    objectToActivate.SetActive(false);  
}  
}  
  
void OnTriggerEnter(Collider other)  
{  
    if (other.CompareTag("Player"))  
    {  
        isPlayerInTrigger = true;  
        if (interactionText != null)  
        {  
            interactionText.text = "Press [F] to Pick Up Flashlight";  
            interactionText.enabled = true;  
        }  
  
        if (Input.GetKeyDown(KeyCode.F))  
        {  
            if (pickupSound != null && audioSource != null)  
            {  
                GameObject soundObject = new GameObject("TempAudio");  
            }  
        }  
    }  
}
```

```

        AudioSource tempAudio =
soundObject.AddComponent<AudioSource>();
        tempAudio.volume = 1.0f;
        tempAudio.PlayOneShot(pickupSound);
        Destroy(soundObject, pickupSound.length);
    }
    else
    {
        Debug.LogWarning("Failed to play sound");
    }

    if (objectToActivate != null)
    {
        objectToActivate.SetActive(true);
    }

    Destroy(gameObject);
    if (interactionText != null)
    {
        interactionText.enabled = false;
    }
}

void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player") && interactionText != null)
    {
        isPlayerInTrigger = false;
        interactionText.enabled = false;
    }
}

```

این اسکریپت برای مدیریت تعاملات مربوط به برداشتن یک فلش لایت در بازی طراحی شده است و فرآیند تعامل بازیکن با شیء فلش لایت را به طور کامل کنترل می‌کند. در ابتدای اسکریپت، متغیرهایی برای نگهداری ارجاع به اجزای مختلف مانند متن تعامل (`interactionText`)، صدای برداشتن فلش لایت (`pickupSound`)، شیءی که قرار است فعال شود (`objectToActivate`)، و همچنین منبع صدا (`audioSource`) تعریف شده‌اند. در متدهای `Start`، `Update`، `OnCollisionEnter` و `OnCollisionExit`، ابتدا بررسی می‌شود که آیا منبع صدا (`AudioSource`) برای پخش صدای مربوط به برداشتن فلش لایت وجود دارد یا نه. اگر منبع صدا وجود نداشته باشد، یک `AudioSource` جدید به شیء افزوده می‌شود. سپس، بررسی می‌شود که آیا ارجاع به `objectToActivate`، `pickupSound` و `interactionText` به درستی تنظیم شده‌اند.

شده‌اند یا خیر. اگر هر کدام از این ارجاع‌ها تنظیم نشده باشند، پیامی هشدار در کنسول چاپ می‌شود. همچنین در این متد، صدای مربوط به برداشتن فلش‌لایت به‌طور پیش‌فرض تنظیم می‌شود تا بعداً به‌راحتی پخش شود.

در متد `OnTriggerStay`, زمانی که بازیکن وارد ناحیه تریگر می‌شود و در آن باقی می‌ماند، بررسی می‌شود که آیا بازیکن در ناحیه تریگر است یا نه. اگر بازیکن در ناحیه باشد، متن تعامل به بازیکن نمایش داده می‌شود که از او می‌خواهد فلش‌لایت را با فشردن کلید [F] بردارد. این پیام به‌طور زنده به بازیکن اطلاع می‌دهد که چه کاری باید انجام دهد. سپس، در صورتی که بازیکن کلید [F] را فشار دهد، صدای برداشتن فلش‌لایت پخش می‌شود. برای این کار، یک شیء موقت برای پخش صدای خاص ساخته می‌شود تا صدای آن پس از اتمام پخش حذف شود. پس از پخش صدا، شیء فلش‌لایت که قرار است فعال شود (`objectToActivate`), به بازی اضافه می‌شود و فعال می‌شود. همچنین، شیء اصلی که این اسکریپت روی آن قرار دارد از بازی حذف می‌شود تا بازیکن به‌طور کامل فلش‌لایت را در اختیار بگیرد. در نهایت، پیام تعامل که به بازیکن نشان داده می‌شود غیرفعال می‌شود تا از نمایش غیرضروری پیام جلوگیری شود.

در متد `OnTriggerExit`, زمانی که بازیکن از ناحیه تریگر خارج می‌شود، پیام تعامل غیرفعال می‌شود تا دیگر نمایش داده نشود و بازیکن در خارج از ناحیه تریگر نباشد. این اسکریپت به‌طور کلی برای مدیریت یک تعامل ساده و جذاب در بازی طراحی شده است که به بازیکن این امکان را می‌دهد که با فشردن یک دکمه به‌راحتی فلش‌لایت را بردارد و تجربه بازی را غنی‌تر کند. با این روش، بازیکن همیشه به‌طور مستقیم با محیط و اجزای مختلف آن ارتباط برقرار می‌کند و تجربه‌ای تعاملی و روان در بازی ایجاد می‌شود.

وقتی بازیکن از اتاق کوچک خارج شود و وارد اتاق بزرگ و تاریک شود یک تریگر وجود دارد که با وارد شدن به آن کرکتر کابوس فعال و شروع به حرکت و تعقیب بازیکن می‌کند. اسکریپت `StartFollow` که مربوط به فعال سازی کرکتر کابوس است:

```
using UnityEngine;

public class StartFollow : MonoBehaviour
{
    public GameObject character;

    private void OnTriggerEnter(Collider other)
    {
```

```

    if (other.CompareTag("Player"))
    {
        if (character != null)
        {
            character.SetActive(true);
        }
        else
        {
            Debug.LogWarning("Character GameObject is not assigned in
the Inspector!");
        }
    }
}

```

این اسکریپت برای مدیریت تعاملات مربوط به فعال‌سازی یک شخصیت یا شیء خاص در بازی طراحی شده است. زمانی که بازیکن وارد یک ناحیه خاص (تریگر) می‌شود، شخصیت موردنظر فعال می‌شود. در این اسکریپت، متغیر `character` برای نگهداری ارجاع به شیء شخصیت یا شیء خاصی که قرار است فعال شود، تعریف شده است.

در متدهای `OnTriggerEnter(Collider other)`, زمانی که بازیکن وارد ناحیه تریگر می‌شود (با داشتن تگ "Player"), ابتدا بررسی می‌شود که آیا شیء `character` به درستی تنظیم شده است یا نه. اگر ارجاع به شیء `character` وجود داشته باشد، آن شیء فعال می‌شود با استفاده از `character.SetActive(true)` و در بازی ظاهر می‌کند. اگر شیء `character` تنظیم نشده باشد، یک پیام هشدار در کنسول چاپ می‌شود که نشان‌دهنده عدم تخصیص شیء در `Inspector` است.

و در نهایت کرکتر کابوس شروع به تعقیب می‌کند، حالا در این قسمت از بازی وقتی بازیکن چراغ قوه را به سمت کابوس بگیرد او می‌ایستد و اگر نور به سمت اون نباید به دنبال بازیکن می‌آید. اسکریپت برای حرکت و توقف کرکتر کابوس :

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class FollowAi : MonoBehaviour
{
    public NavMeshAgent ai;
    public Transform player;
    public Animator aiAnim;
    public Light flashLight; //Reference
    public AudioSource soundStop;
}

```

```
public AudioSource soundMove;

Vector3 dest;

void Start()
{
    if (ai == null)
    {
        ai = GetComponent<NavMeshAgent>();
        if (ai == null)
        {
            Debug.LogError("NavMeshAgent not found on " +
gameObject.name);
        }
    }

    if (flashLight == null)
    {
        Debug.LogError("FlashLight reference not set on " +
gameObject.name);
    }

    if (soundMove == null)
    {
        Debug.LogError("soundMove AudioSource is not assigned on " +
gameObject.name);
    }
    if (soundStop == null)
    {
        Debug.LogError("soundStop AudioSource is not assigned on " +
gameObject.name);
    }
}

void Update()
{
    if (ai != null && player != null && ai.isOnNavMesh)
    {
        dest = player.position;
        bool isInLight = flashLight != null && flashLight.enabled &&
IsInFlashlightRange();

        if (!isInLight)
        {
            // Move
            ai.SetDestination(dest);
            aiAnim.SetTrigger("jog");
        }
    }
}
```

```

        if (soundStop != null && soundStop.isPlaying)
        {
            soundStop.Stop();
        }
        if (soundMove != null && !soundMove.isPlaying)
        {
            soundMove.Play();
        }
    }
    else
    {
        // Stop
        ai.SetDestination(transform.position);
        aiAnim.SetTrigger("idle");

        if (soundMove != null && soundMove.isPlaying)
        {
            soundMove.Stop();
        }
        if (soundStop != null && !soundStop.isPlaying)
        {
            soundStop.Play();
        }
    }
}

bool IsInFlashlightRange()
{
    if (flashLight == null) return false;

    Vector3 directionToEnemy = transform.position -
    flashLight.transform.position;
    float angle = Vector3.Angle(directionToEnemy,
    flashLight.transform.forward);

    if (angle < flashLight.spotAngle * 0.5f &&
directionToEnemy.magnitude <= flashLight.range)
    {
        return true;
    }
    return false;
}
}

```

این اسکریپت برای ایجاد رفتار هوش مصنوعی (AI) در بازی‌های سه‌بعدی طراحی شده است که به‌طور دائمی بازیکن را دنبال می‌کند. این هوش مصنوعی با استفاده از سیستم NavMeshAgent برای حرکت در محیط بازی، و همچنین با بهره‌گیری از انیماتور و صدایها، به واسطه‌هایی خاص بر

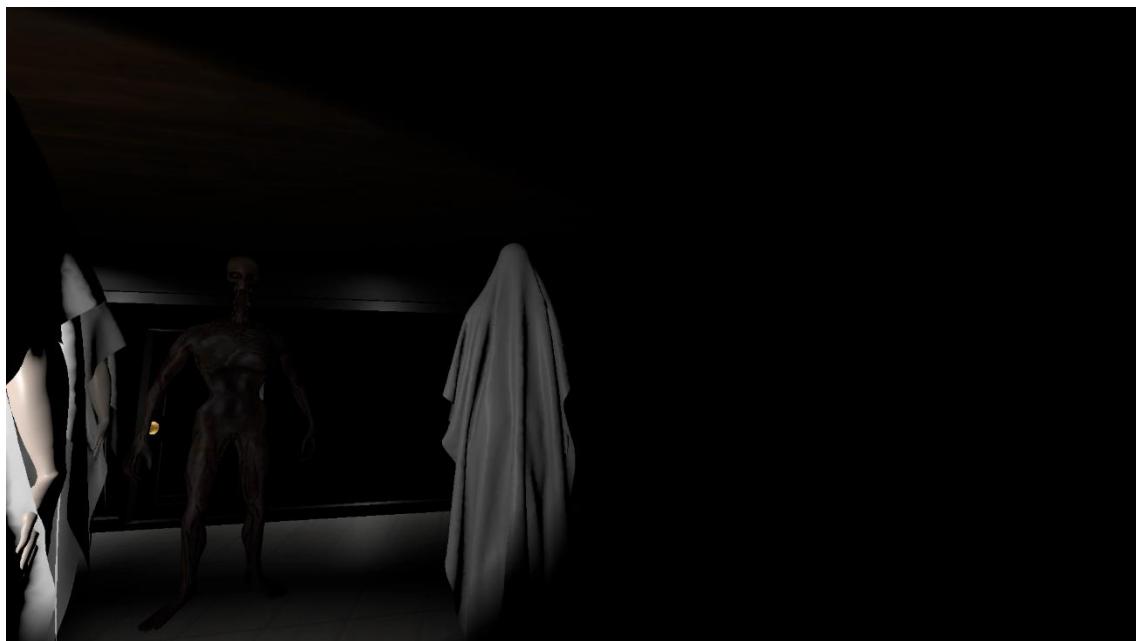
اساس موقعیت و شرایط خاص (مانند نور فلش‌لایت) پرداخته و رفتارهای متفاوتی را از خود نشان می‌دهد.

در ابتداء، در متدهای `Start`، `Update`، بررسی می‌شود که آیا ارجاع به کامپوننت `NavMeshAgent` موجود است یا نه. این کامپوننت مسئول حرکت هوش مصنوعی بر اساس ناویمش (`NavMesh`) است. اگر این کامپوننت روی شیء موجود نباشد، پیام خطا در کنسول چاپ می‌شود. سپس، ارجاع به فلش‌لایت (`flashLight`)، و همچنین صدای حرکت (`soundMove`) و توقف (`soundStop`) بررسی می‌شود. اگر یکی از این ارجاعات در `Inspector` تنظیم نشده باشد، هشدارهایی در کنسول نمایش داده می‌شود. این بررسی‌ها برای جلوگیری از بروز خطاها احتمالی و اطمینان از اینکه همه اجزاء به درستی تنظیم شده‌اند انجام می‌شود.

در متدهای `Update`، که در هر فریم از بازی اجرا می‌شود، ابتدا بررسی می‌شود که آیا هوش مصنوعی، بازیکن و سیستم `NavMeshAgent` در وضعیت مناسبی قرار دارند. اگر همه اجزاء به درستی تنظیم شده باشند، مقصد هوش مصنوعی به موقعیت بازیکن تنظیم می‌شود. این یعنی هوش مصنوعی به طور پیوسته سعی دارد بازیکن را دنبال کند. سپس بررسی می‌شود که آیا فلش‌لایت روشن است و آیا هوش مصنوعی در دامنه نور فلش‌لایت قرار دارد. اگر فلش‌لایت روشن نباشد یا هوش مصنوعی خارج از محدوده نور فلش‌لایت گیرد، هوش مصنوعی به سمت بازیکن حرکت می‌کند و انیمیشن "jog" (دویدن آرام) اجرا می‌شود. در این وضعیت، اگر صدای توقف در حال پخش باشد، متوقف می‌شود و صدای حرکت پخش می‌شود.

اما اگر هوش مصنوعی در دامنه نور فلش‌لایت قرار گیرد (این وضعیت با استفاده از متدهای `IsInFlashlightRange` بررسی می‌شود)، حرکت آن متوقف می‌شود و صدای توقف پخش می‌شود. این تغییرات در حرکت و صدا به طور دینامیک و بر اساس شرایط محیطی تنظیم می‌شود. متدهای `IsInFlashlightRange` مسئول بررسی این است که آیا هوش مصنوعی در محدوده نوار نور فلش‌لایت قرار دارد یا نه. ابتدا، فاصله بین هوش مصنوعی و فلش‌لایت محاسبه می‌شود. سپس زاویه بین جهت فلش‌لایت و جهت هوش مصنوعی محاسبه می‌شود. اگر این زاویه کمتر از نصف زاویه مخروط نور فلش‌لایت باشد و فاصله بین فلش‌لایت و هوش مصنوعی نیز در محدوده شعاع نور فلش‌لایت باشد، متدهای `IsInFlashlightRange` `true` بر می‌گرداند و نشان می‌دهد که هوش مصنوعی در دامنه نور فلش‌لایت قرار دارد. در غیر این صورت، `false` بر می‌گرداند.

در نهایت، این اسکریپت به‌طور کلی رفتار هوش مصنوعی را تحت شرایط مختلف کنترل می‌کند و باعث می‌شود که هوش مصنوعی به‌طور دائمی بازیکن را دنبال کند. با این حال، زمانی که در محدوده نور فلش‌لایت قرار می‌گیرد، متوقف می‌شود. همچنین انجمن‌ها و صدای مختلف برای ایجاد یک تجربه هیجان‌انگیز و دلهره‌آور برای بازیکن به‌طور دقیق و به موقع تغییر می‌کنند. این اسکریپت برای استفاده در بازی‌های ترسناک یا بازی‌هایی که در آن‌ها دشمنان یا موجودات بر اساس شرایط محیطی (مانند نور یا صدا) رفتار می‌کنند، بسیار کاربردی است.



ایستادن کابوس با وجود نور

در ادامه‌ی مرحله، در صورتی که شخصیت کابوس بیش از حد به بازیکن نزدیک شود، بازیکن شکست می‌خورد و وضعیت Game Over فعال می‌شود. اسکریپت TriggerEnemy به همین منظور است:

```
using UnityEngine;

public class TriggerEnemy : MonoBehaviour
{
    public GameObject gameOver;

    void Start()
    {
        if (gameOver == null)
        {
            Debug.LogError("gameOver reference not set on " +
gameObject.name);
        }
    }
}
```

```

if (gameOver != null && gameOver.activeSelf)
{
    gameOver.SetActive(false);
}
}

void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        Debug.Log("Player entered trigger!");
        if (gameOver != null && !gameOver.activeSelf)
        {
            gameOver.SetActive(true);
            Debug.Log("GameOver activated.");
        }
        gameObject.SetActive(false);
    }
}
}

```

این اسکریپت برای مدیریت وضعیت "Game Over" در بازی طراحی شده است و زمانی که بازیکن به یک تریگر برخورد می‌کند (مثلاً با یک دشمن یا شیء خاص)، پیام "Game Over" به نمایش در می‌آید. در ابتدا، بررسی می‌شود که آیا ارجاع به شیء gameOver به درستی تنظیم شده است یا نه؛ اگر تنظیم نشده باشد، یک پیام خطأ چاپ می‌شود. سپس، اگر gameOver فعال باشد، آن را غیرفعال می‌کند تا از نمایش اولیه آن جلوگیری کند. زمانی که بازیکن وارد ناحیه تریگر می‌شود، پیام "Game Over" فعال می‌شود تا نشان‌دهنده پایان بازی باشد و سپس شیء‌ای که این اسکریپت به آن متصل است، غیرفعال می‌شود تا از ادامه بازی جلوگیری کند. این اسکریپت به‌طور مؤثر برای نمایش پیام پایان بازی و جلوگیری از ادامه بازی پس از وقوع یک رویداد خاص، مانند برخورد با دشمن، طراحی شده است.

وقتی GameOver فعال شود در این حالت، یک دوربین فرعی فعال می‌شود که صحنه‌ای سینمایی از کابوس را نمایش می‌دهد؛ در این صحنه، کابوس با حالت تهدیدآمیز غرش کرده و فریاد می‌زند. نمایش این صحنه برای افزایش تنش و تأکید بر شکست بازیکن طراحی شده است. برای مدیریت این اتفاق، از اسکریپتی با عنوان GameOver استفاده می‌شود که وظیفه‌ی تشخیص باخت، فعال‌سازی دوربین فرعی و توقف گیمپلی را برعهده دارد:

```

using UnityEngine;

public class GameOver : MonoBehaviour

```

```
{  
    public GameObject character;  
    public GameObject fpController;  
    public GameObject PlayAgain;  
    public GameObject chAmimation;  
    private Animator animator;  
  
    void Start()  
    {  
        if (character != null)  
        {  
            character.SetActive(false);  
        }  
  
        if (fpController != null)  
        {  
            fpController.SetActive(false);  
        }  
  
        animator = GetComponent<Animator>();  
  
        if (PlayAgain != null)  
        {  
            PlayAgain.SetActive(false);  
        }  
  
        if (animator != null)  
        {  
            animator.SetTrigger("scream");  
        }  
  
        Invoke("ActivatePlayAgain", 2.26f);  
    }  
  
    void ActivatePlayAgain()  
    {  
        if (chAmimation != null)  
        {  
            chAmimation.SetActive(false);  
        }  
  
        if (PlayAgain != null)  
        {  
            PlayAgain.SetActive(true);  
        }  
    }  
}
```

این اسکریپت برای مدیریت وضعیت "Game Over" (پایان بازی) در یک بازی طراحی شده است. هدف اصلی این اسکریپت این است که پس از پایان بازی، برخی از اجزای بازی مانند شخصیت اصلی و کنترلر اول شخص غیرفعال شوند و سپس انیمیشن‌هایی مانند فریاد یا نمایش‌های خاص پخش شوند. همچنین پس از گذشت مدت زمانی کوتاه، به بازیکن امکان بازی مجدد داده می‌شود. در متدهای Start و fpController (که کنترلر اول شخص است) به درستی تنظیم شده است یا نه. اگر تنظیم شده باشند، این اجزاء غیرفعال می‌شوند تا از ادامه بازی جلوگیری شود. سپس یک انیماتور برای پخش انیمیشن مربوط به پایان بازی (که در اینجا با نام "scream" مشخص شده) فراخوانی می‌شود. با استفاده از متدهای Invoke، پس از ۰.۲۶ ثانیه متد ActivatePlayAgain فراخوانی می‌شود که در آن ابتدا انیمیشن یا شیء chAmimation غیرفعال می‌شود و سپس گزینه بازی مجدد (PlayAgain) فعال می‌شود تا بازیکن بتواند دوباره بازی را آغاز کند. به‌طور کلی، این اسکریپت برای مدیریت و کنترل وضعیت پلیان بازی در بازی‌های مختلف کاربرد دارد و به‌ویژه در بازی‌های ترسناک یا اکشن که پایان‌های هیجان‌انگیز نیاز دارند، به‌خوبی عمل می‌کند.

پس از فعال شدن Timeline با نام PlayAgain نیز که شامل یک GameObject است که صحنه‌ای سینمایی یا گرافیکی را نمایش می‌دهد و پیامی با مضمون دعوت به تلاش مجدد به بازیکن ارائه می‌کند. در طول اجرای این Timeline، کنترل بازیکن غیرفعال است تا تمرکز بر روی فضای احساسی و نتیجه‌ی شکست حفظ شود. پس از پایان نمایش، بازیکن می‌تواند با فشردن کلید Enter مجددًا بازی را از همان Scene فعلی آغاز کند. اسکریپت PlayAgain برای لود دوباره مرحله بازی:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class PlayAgain : MonoBehaviour
{
    private bool canPressKey = false;

    void Start()
    {
        Invoke("EnableKeyPress", 3f);
    }

    void Update()
    {
        if (canPressKey && Input.GetKeyDown(KeyCode.Return))
```

```
{  
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);  
}  
  
}  
  
private void EnableKeyPress()  
{  
    canPressKey = true;  
}  
  
public void PlayAgainBtn()  
{  
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);  
}  
}
```

این اسکریپت برای مدیریت امکان شروع دوباره بازی پس از پایان آن طراحی شده است. در ابتدا، متغیر canPressKey برای کنترل اینکه آیا بازیکن می‌تواند کلید را فشار دهد یا نه، تعریف می‌شود. در متدهای Start و Invoke برای فعال کردن فشار کلید بعد از ۳ ثانیه استفاده می‌شود، به این معنی که بازیکن تنها پس از گذشت مدت زمانی مشخص قادر خواهد بود کلید Enter را فشار دهد تا بازی دوباره شروع شود. در متدهای Update و OnGUI، بررسی می‌شود که آیا بازیکن می‌تواند کلید Enter را فشار دهد یا نه و اگر کلید فشرده شود، بازی دوباره بارگذاری می‌شود و از ابتدا شروع می‌شود. متدهای PlayAgainBtn و EnableKeyPress به طور خودکار پس از ۳ ثانیه فعال می‌شوند و متغیر canPressKey را به true تغییر می‌دهند تا اجازه دهد بازیکن کلید Enter را فشار دهد. همچنان، متدهای OnGUI و OnSceneSwitched برای دکمه "Play Again" طراحی شده است که اگر بازیکن بر روی آن کلیک کند، بازی دوباره از ابتدا شروع می‌شود. این اسکریپت به طور کلی برای ایجاد تجربه‌ای ساده و کاربرپسند برای شروع مجدد بازی پس از اتمام آن طراحی شده است و از دو روش (فشردن کلید Enter یا کلیک بر دکمه) به بازیکن این امکان را می‌دهد که بازی را دوباره آغاز کند.



صفحه تلاش دوباره

در ادامه‌ی مرحله، در صورتی که بازیکن موفق شود از دست شخصیت کابوس فرار کند و خود را به اتاق امن جدید برساند، صحنه وارد فاز پایانی می‌شود. در این اتاق، عروسک تدی روی زمین قرار دارد. زمانی که بازیکن به تدی نزدیک می‌شود، می‌تواند با فشردن کلید F آن را بردارد. پس از انجام این عمل، بازی به صورت خودکار به صحنه‌ی جدیدی با عنوان HouseEndDream منتقل می‌شود. این انتقال نشان‌دهنده‌ی پایان کابوس فعلی و بازگشت به خانه یا فضایی امن‌تر در روایت بازی است. حالا اسکریپت TeddyWin برای اینکار:

```
using UnityEngine;
using TMPro;
using UnityEngine.SceneManagement;

public class TeddyWin : MonoBehaviour
{
    public TMP_Text interactionText;
    private bool isPlayerInTrigger = false;

    void Start()
    {
        if (interactionText != null)
        {
            interactionText.enabled = false;
            Debug.Log("interactionText initialized and disabled.");
        }
        else
        {
```

```
        Debug.LogWarning("interactionText is not assigned in
Inspector!");
    }

}

void Update()
{
    if (isPlayerInTrigger)
    {
        if (Input.GetKeyDown(KeyCode.F))
        {
            SceneManager.LoadScene(SceneManager.GetActiveScene().buil
dIndex+1);
        }
    }
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = true;
        if (interactionText != null)
        {
            interactionText.enabled = true;
            Debug.Log("interactionText enabled when player entered
trigger.");
        }
    }
}

private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInTrigger = false;
        if (interactionText != null)
        {
            interactionText.enabled = false;
            Debug.Log("interactionText disabled when player exited
trigger.");
        }
    }
}
}
```

این اسکریپت برای مدیریت تعامل با یک ناحیه خاص در بازی طراحی شده است که وقتی بازیکن وارد

آن ناحیه می‌شود، یک متن برای تعامل به او نمایش داده می‌شود و پس از فشار دادن کلید [F]، بازیکن به مرحله بعدی بازی منتقل می‌شود. این اسکریپت شامل چندین بخش است که به طور دقیق فرآیند ورود به ناحیه و رفتن به مرحله بعد را مدیریت می‌کند.

در ابتدا، در متد Start، بررسی می‌شود که آیا ارجاع به interactionText (متنی که برای تعامل نمایش داده می‌شود) تنظیم شده است یا نه. اگر تنظیم شده باشد، متن غیرفعال می‌شود و پیامی در کنسول چاپ می‌شود. در صورتی که ارجاع به interactionText تنظیم نشده باشد، یک پیام هشدار در کنسول چاپ می‌شود تا نشان دهد که این ارجاع به درستی در Inspector تنظیم نشده است. در متد Update، زمانی که بازیکن وارد ناحیه تریگر می‌شود و کلید [F] فشرده می‌شود، بازی به مرحله بعدی منتقل می‌شود. این کار با استفاده از SceneManager.LoadScene() انجام می‌شود که صحنه بعدی را بارگذاری می‌کند.

در متد OnTriggerEnter، زمانی که بازیکن وارد ناحیه تریگر می‌شود، متغیر true به isPlayerInTrigger تغییر می‌کند و متن تعامل نمایش داده می‌شود. در متد OnTriggerExit، زمانی که بازیکن از ناحیه تریگر خارج می‌شود، متن تعامل غیرفعال می‌شود و بازیکن نمی‌تواند دیگر با تریگر تعامل داشته باشد.

این اسکریپت به طور کلی برای ایجاد یک نقطه تعامل در بازی استفاده می‌شود که در آن بازیکن می‌تواند با فشردن یک دکمه وارد مرحله بعدی بازی شود، و به طور مؤثر تجربه‌ی بازی را گسترش می‌دهد.

۴-۲-۱۱ - پیاده سازی Stage ششم

در مرحله ششم ابتدا شی come back فعال می‌شود که دارای انیمیشن هست و بازیکن نفس نفس زنان اطرافش را نگاه می‌کند، باز هم اینجا از یک دوربین فرعی استفاده شده. اسکریپت :ComeBack

```
using UnityEngine;

public class Comeback : MonoBehaviour
{
    public GameObject comeback;
    public GameObject firstPersonController;
    private GameManager gameManager;
```

```
void Start()
{
    gameManager = FindObjectOfType<GameManager>();
    if (gameManager == null)
    {
        Debug.LogError("GameManager not found in the scene!");
        return;
    }

    if (comeback == null)
    {
        Debug.LogError("Come Back reference is not assigned!");
        return;
    }

    comeback.SetActive(true);
    Animator animator = comeback.GetComponent<Animator>();
    if (animator == null)
    {
        Debug.LogError("Animator component not found on comeback!");
        return;
    }

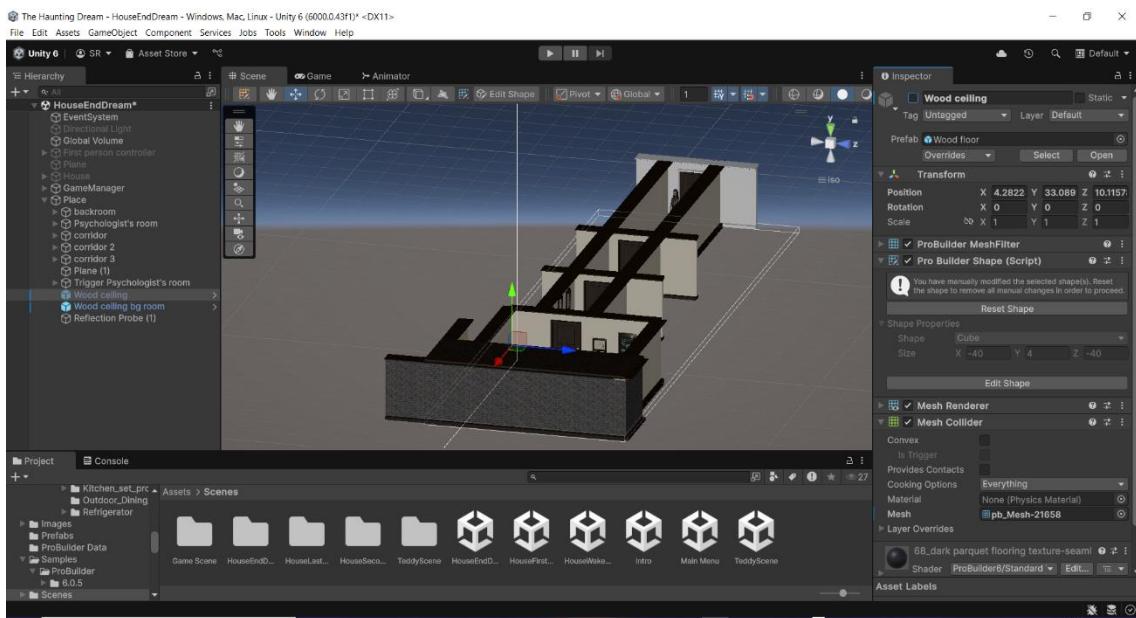
    AnimatorStateInfo stateInfo =
animator.GetCurrentAnimatorStateInfo(0);
    float animationLength = stateInfo.length > 0 ? stateInfo.length :
2f;
    Invoke("OnStageCompleted", animationLength);

    if (firstPersonController != null)
    {
        firstPersonController.SetActive(false);
    }
    else
    {
        Debug.LogError("FirstPersonController reference is not
assigned!");
    }
}

void OnStageCompleted()
{
    if (comeback != null) comeback.SetActive(false);
    if (firstPersonController != null)
firstPersonController.SetActive(true);
    if (gameManager != null) gameManager.StageCompleted();
}
}
```

این اسکریپت برای مدیریت وضعیت‌های خاص در بازی طراحی شده است که در آن بازیکن پس از یک رویداد یا اینیمیشن خاص به حالت قبلی خود بازمی‌گردد. در ابتدا، اسکریپت بررسی می‌کند که آیا firstPersonController به درستی تنظیم شده‌اند. اگر هر کدام از این ارجاع‌ها تنظیم نشده باشد، پیام خطا در کنسول چاپ می‌شود. پس از آن، شیء comeback فعال شده و انیماتور آن دریافت می‌شود. مدت زمان اینیمیشن به دست آمده و پس از پایان آن، با استفاده از متدهای OnStageCompleted() و StageCompleted() فراخوانی می‌شود. در این متدها، پس از اتمام اینیمیشن، شیء firstPersonController غیرفعال و comeback فعال می‌شود تا بازیکن بتواند دوباره بازی را کنترل کند. همچنین، متدهای GameManager از فراخوانی می‌شود تا نشان دهد که مرحله تکمیل شده است. این اسکریپت برای بازی‌های مناسب است که پس از یک رویداد خاص یا اینیمیشن نیاز به بازگشت به وضعیت قبلی دارند و به‌طور مؤثر به مدیریت تجربه بازی کمک می‌کند.

سپس بازیکن وارد در اتاق پشتی یا انباری می‌شود که در آن یک میز، صندلی و کارتون وجود دارد، وقتی بازیکن نزدیک میز می‌شود یک کاغذ روی میز می‌بیند که نوشته شده کلیدی وجودی نداره و راه فراری نداری در همین حین در واقع بازیکن وارد یک تریگر شده که شیء House که اجزای کل خانه هست غیرفعال و شیء جدید Place که یک محیط جداگانه هست فعال می‌شود ولی در اینجا بازیکن هنوز متوجه نشده، وقتی بازیکن برای کاوش بیشتر از انباری بیرون می‌رود یک محیط کاملاً جدید را می‌بیند.



محیط جدید

برای اینکار اسکریپت Transfer نیاز است:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;

public class Transfer : MonoBehaviour
{
    public GameObject placeBackroom;
    public GameObject house;
    public GameObject ghost;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            Debug.Log("Player entered trigger");
            placeBackroom.SetActive(true);
            ghost.SetActive(true);
            house.SetActive(false);
        }
    }
}
```

این اسکریپت برای مدیریت انتقال میان مکان‌ها یا صحنه‌ها در بازی طراحی شده است. زمانی که بازیکن وارد یک ناحیه خاص (تریگر) می‌شود، چندین شیء مختلف فعال یا غیرفعال می‌شوند تا شرایط محیطی بازی تغییر کند. در این اسکریپت، زمانی که بازیکن وارد تریگر می‌شود، شیء

ghost و placeBackroom فعال می‌شوند تا بازیکن بتواند با آنها تعامل کند، در حالی که شیء ghost غیرفعال می‌شود تا از نمایش آن جلوگیری شود. این اسکریپت برای بازی‌های مناسب است که نیاز به تغییر محیط‌ها یا مکان‌ها دارند و به ویژه در بازی‌های ترسناک یا ماجراجویی که انتقال به مکان‌های جدید و نمایش موجودات یا واقعی خاص اهمیت دارد، کاربردی است.

بعد از وارد شدن بازیکن به محیط جدید با اتاق مشاوره رو به رو می‌شود که برای آگاهی بهتر کاربر شی Timeline فعال می‌شود که دارای Psychologist's room هست که یک زیر نویس از طرف خود بازیکن نمایش می‌دهد که کاربر متوجه داستان شود. اسکریپت TriggerTherapyRoom :

```
using UnityEngine;
using UnityEngine.Playables;

public class TriggerTherapyRoom : MonoBehaviour
{
    public PlayableDirector timeline;
    public GameObject timelineTrigger;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            timelineTrigger.SetActive(true);
            Invoke("StopTimeLine", 9f);

        }
    }

    void StopTimeLine()
    {
        game0bject.SetActive(false);
    }
}
```

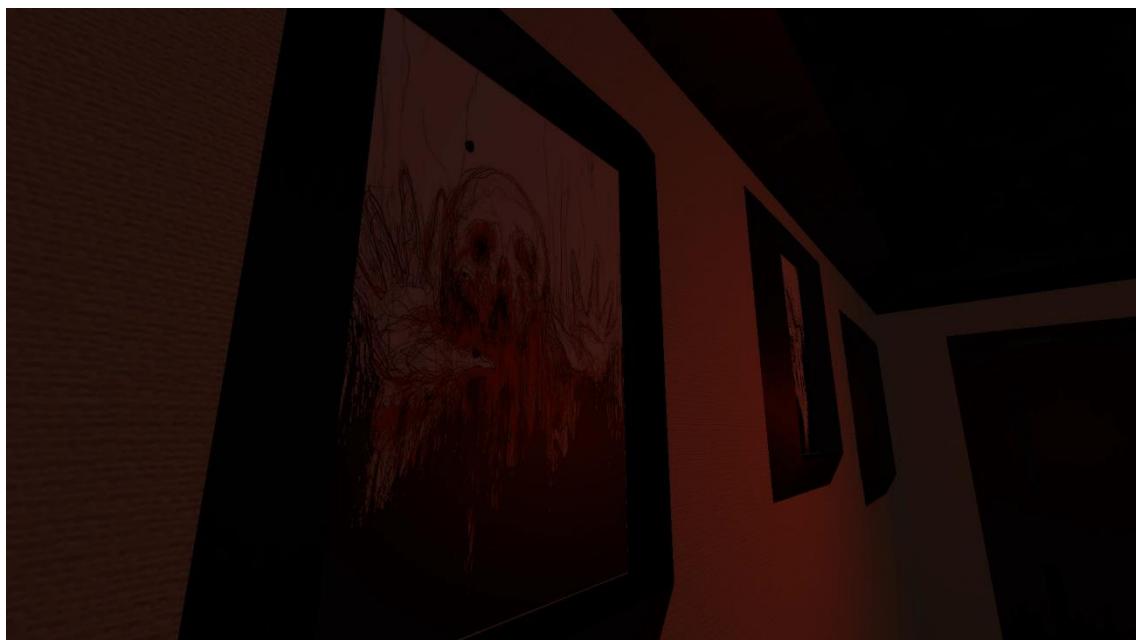
این اسکریپت برای مدیریت شروع و توقف تایم‌لاین‌ها در بازی طراحی شده است. زمانی که بازیکن وارد ناحیه خاصی (تریگر) می‌شود، تایم‌لاین فعال می‌شود و یک شیء مشخص (timelineTrigger) نیز فعال می‌شود. سپس با استفاده از متدهای StopTimeLine، Invoke پس از ۹ ثانیه اجرا می‌شود تا تایم‌لاین را متوقف کند. این اسکریپت به طور مؤثر برای بازی‌های طراحی شده که نیاز به رویدادهای زمان‌محور دارند، مانند بازی‌های داستانی یا ترسناک که در آن‌ها تایم‌لاین‌ها نقش مهمی در پیشبرد داستان و تجربه بازی دارند.

بعد از آن بازیکن می‌تواند پروندهٔ پزشکی خود را که مشکلات او را شرح می‌دهند روز میز ببیند.



پروندهٔ پزشکی کرکتر

بعد از آن وارد یک راهروی ترسناک می‌شود که نورهای آن به رنگ قرمز هست، بازیکن بلاfaciale بعد از ورود می‌تواند موسیقی ترسناک را بشنود. همچنین روی دیوار تابلوی‌هایی با نماد اضطراب ولی ژانر وحشت به نمایش گذاشته شده که کاربر می‌تواند حس ترس و اضطراب را درک کند.



تابلوهای ترسناک در راهرو

وقتی بازیکن وارد تریگر می‌شود صدای ترسناک نیز پخش می‌شود. برای این کار اسکریپت `BgSound`

```
using UnityEngine;

public class BgSound : MonoBehaviour
{
    public AudioSource BackgroundSound;

    private void OnTriggerEnter(Collider other)
    {
        BackgroundSound.Play();
    }
}
```

این اسکریپت برای پخش صدای پس‌زمینه هنگام ورود بازیکن به یک ناحیه خاص طراحی شده است. زمانی که بازیکن وارد تریگر می‌شود، صدای پس‌زمینه از طریق کامپوننت `AudioSource` پخش می‌شود. این اسکریپت به‌طور مؤثر برای بازی‌هایی کاربرد دارد که می‌خواهند صدای محیط یا موسیقی پس‌زمینه در نواحی خاص فعال شود، به‌طوری که بازیکن هنگام ورود به این مناطق تجربه شنیداری جدیدی داشته باشد. این روش می‌تواند در بازی‌های ترسناک یا ماجراجویی برای افزایش اتمسفر و جذابیت محیط استفاده شود.

در ادامه بازیکن وارد یک راهروی دیگر می‌شود که اینبار خالی هست ولی وقتی وارد راهرو می‌شود همزمان وارد تریگر شده و شی `Mom sound` فعال می‌شود که دارای یک `TimeLine` هست. این تایم لاین دیالوگ مادر را همراه با صدا پخش می‌کند که می‌گوید بیا اینجا. برای اعمال این مرحله اسکریپت `MomThreat` نیاز داریم:

```
using UnityEngine;
using UnityEngine.Playables;

public class MomThreat : MonoBehaviour
{
    public PlayableDirector timeline;
    public GameObject timelineTrigger;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            timelineTrigger.SetActive(true);
            Invoke("StopTimeLine", 7f);
        }
    }
}
```

```

    }

    void StopTimeLine()
    {
        game0bject.SetActive(false);
    }
}

```

این اسکریپت برای مدیریت پخش و توقف تایم‌لاین در بازی طراحی شده است. زمانی که بازیکن وارد ناحیه تریگر می‌شود، تایم‌لاین آغاز به پخش می‌کند و شیء مربوط به تایم‌لاین فعال می‌شود. سپس، با استفاده از متدها StopTimeLine()، Invoke()، از متدهای StopTimeLine()،Invoke() استفاده شود تا تایم‌لاین متوقف شود و شیءی که این اسکریپت به آن متصل است غیرفعال گردد. این اسکریپت به ویژه برای بازی‌هایی که نیاز به تایم‌لاین‌ها یا رویدادهای زمان محور دارند، مانند بازی‌های ترسناک یا داستانی که در آن‌ها تغییرات تدریجی در محیط یا داستان رخ می‌دهد، بسیار کاربردی است.

بعد از آن بازیکن دوباره وارد یک راهروی دیگر می‌شود که در انتهای راهرو می‌تواند زن ترسناکی در ابتدای بازی بود را ببیند. اینجا وقتی بازیکن نزدیک تر برود وارد تریگر Ghost section می‌شود زن ترسناک جیغ زنان به سمت بازیکن می‌آید، در همین حین دوربین برای حس ترس بیشتر به لرزه در می‌آید و وقتی وارد تریگر بازیکن شود شیء Please Wake up فعال می‌شود که دارای یک TimeLine هست که صدای درونی بازیکن را به نمایش می‌گذارد که می‌گوید بیدار شو. در ابتدا اسکریپت GhostSection را توضیح می‌دهم:

```

using UnityEngine;

public class GhostSection : MonoBehaviour
{
    public static bool isTriggerGhost=false;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            Debug.Log("Player entered trigger");
            isTriggerGhost=true;
        }
    }
}

```

این اسکریپت برای مدیریت وضعیت ورود بازیکن به ناحیه خاصی طراحی شده است که به‌طور خاص به یک شبح یا موجود ترسناک مربوط می‌شود. زمانی که بازیکن وارد ناحیه تریگر می‌شود، متغیر تغییر می‌کند تا نشان دهد که بازیکن در ناحیه‌ای است که با یک شبح یا رویداد خاص در بازی ارتباط دارد.

در ادامه توسعه‌ی بازی، شخصیت زن ترسناک نیز همانند کاراکتر کابوس با استفاده از سایت معتبر Mixamo انیمیشن‌گذاری شده است. انیمیشن‌های مورد استفاده شامل حالت‌های ایستاده، حرکت، و رفتارهای تهدیدآمیز هستند که با توجه به سیناریو بازی، به صورت منطقی در Animator تنظیم شده‌اند.

این شخصیت نیز جهت حرکت و تعقیب بازیکن از کامپوننت NavMesh Agent بهره‌مند است. بدین ترتیب، زن ترسناک می‌تواند به صورت هوشمندانه در محیط حرکت کند و مسیر خود را بر اساس موقعیت بازیکن و مسیرهای تعریف شده در NavMesh Surface پیدا کند برای راهرو تنظیم شده. حال برای حرکت زن اسکریپت GhostAi :

```
using UnityEngine;

public class GhostAi : MonoBehaviour
{
    public UnityEngine.AI.NavMeshAgent ai;
    public Transform player;
    public Animator aiAnim;
    public AudioSource soundMove;
    public CameraShake cameraShake;
    public GameObject TimelineEnd;
    Vector3 dest;

    void Start()
    {
        if (ai != null && !ai.isOnNavMesh)
        {
            UnityEngine.AI.NavMeshHit hit;
            if (UnityEngine.AI.NavMesh.SamplePosition(transform.position,
out hit, 5.0f, UnityEngine.AI.NavMesh.AllAreas))
            {
                ai.Warp(hit.position);
            }
        }
    }

    void Update()
    {
        if (player != null)
        {
            if (Vector3.Distance(transform.position, player.position) > 5.0f)
            {
                ai.destination = player.position;
            }
            else
            {
                ai.Warp(player.position);
            }
        }
    }
}
```

```
{  
    if (ai != null && ai.isOnNavMesh)  
    {  
        if (GhostSection.isTriggerGhost)  
        {  
            dest = player.position;  
            ai.destination = dest;  
            ai.isStopped = false;  
            if (!soundMove.isPlaying)  
            {  
                soundMove.Play();  
            }  
            if (cameraShake != null)  
            {  
                cameraShake.StartShake();  
            }  
            aiAnim.ResetTrigger("idle");  
            aiAnim.SetTrigger("jog");  
        }  
    }  
    else  
    {  
        ai.isStopped = true;  
        if (soundMove.isPlaying)  
        {  
            soundMove.Stop();  
        }  
        if (cameraShake != null)  
        {  
            cameraShake.StopShake();  
        }  
        aiAnim.ResetTrigger("jog");  
        aiAnim.SetTrigger("idle");  
    }  
}  
  
}  
  
private void OnTriggerEnter(Collider other)  
{  
    if (other.CompareTag("Player"))  
    {  
        Debug.Log("Player entered trigger");  
        gameObject.SetActive(false);  
        soundMove.Stop();  
        TimelineEnd.SetActive(true);  
    }  
}
```

}

این اسکریپت برای کنترل رفتار یک شبح هوش مصنوعی در بازی طراحی شده است که به‌طور دائم بازیکن را دنبال می‌کند و در صورت ورود بازیکن به ناحیه خاص، یک رویداد خاص را اجرا می‌کند. زمانی که شبح فعال است (بر اساس متغیر `GhostSection.isTriggerGhost`، به سمت بازیکن حرکت کرده و صدا و انیمیشن حرکت پخش می‌شود. همچنین، اگر دوربین دارای لرزش باشد، لرزش شروع می‌شود تا تنفس و ترس را در بازیکن ایجاد کند. در صورتی که بازیکن از ناحیه خارج شود، حرکت شبح متوقف شده، صدا قطع و انیمیشن به حالت بی‌حرکت تغییر می‌کند. زمانی که شبح بازیکن بروخورد می‌کند، شیء شبح غیرفعال می‌شود و یک تایملاین جدید فعال می‌شود. این اسکریپت برای بازی‌هایی که در آن‌ها موجودات یا دشمنان بر اساس شرایط خاص حرکت می‌کنند و تعاملات خاصی با بازیکن دارند، بسیار مفید است.

حالا برای لرزش دوربین اسکریپت `CameraShake` که آن را به دوربین اصلی اضافه می‌کنیم :

```
using UnityEngine;

public class CameraShake : MonoBehaviour
{
    public float shakeDuration = 0.1f;
    public float shakeMagnitude = 0.2f;
    private Vector3 originalPosition;
    private float shakeTimer = 0f;
    private bool isShaking = false;

    void Start()
    {
        originalPosition = transform.localPosition;
    }

    void Update()
    {
        if (isShaking && shakeTimer > 0)
        {

            transform.localPosition = originalPosition +
Random.insideUnitSphere * shakeMagnitude;
            shakeTimer -= Time.deltaTime;
        }
        else
        {
            transform.localPosition = originalPosition;
            isShaking = false;
        }
    }
}
```

```
}

public void StartShake()
{
    isShaking = true;
    shakeTimer = shakeDuration;
}

public void StopShake()
{
    isShaking = false;
    shakeTimer = 0f;
}
```

این اسکریپت برای ایجاد و مدیریت اثر لرزش دوربین در بازی طراحی شده است و به طور خاص برای شبیه‌سازی لرزش‌های محیطی یا واکنش‌های دوربین در مواجهه با رویدادهایی مانند برخوردگاهی شدید، ترسناک یا ضربات مختلف استفاده می‌شود. هدف اصلی این اسکریپت این است که با ایجاد تغییرات تصادفی در موقعیت دوربین، حس اضطراب، ترس یا هیجان را به بازیکن منتقل کند.

در ابتدا، موقعیت اصلی دوربین (originalPosition) ذخیره می‌شود. این موقعیت برای زمانی استفاده می‌شود که لرزش متوقف می‌شود و دوربین باید به جایگاه اولیه خود بازگردد. در متد Update(), اسکریپت بررسی می‌کند که آیا لرزش فعال است یا خیر و آیا زمان باقی‌مانده برای لرزش هنوز ادامه دارد. اگر لرزش فعال باشد، دوربین به‌طور تصادفی جابه‌جا می‌شود. برای این کار ازتابع Random.insideUnitSphere استفاده می‌شود که موقعیتی تصادفی در اطراف نقطه اصلی دوربین ایجاد می‌کند، که به‌طور معمول باعث لرزش و حرکت ناگهانی دوربین در جهات مختلف می‌شود. سپس تایмер لرزش (shakeTimer) به مقدار Time.deltaTime کاهش می‌یابد. این کار باعث می‌شود که هر فریم از زمان لرزش، تایمر کاهش پیدا کند تا مدت زمان لرزش کنترل شود. پس از اتمام مدت زمان لرزش (زمانی که تایmer به صفر برسد)، لرزش متوقف شده و دوربین به موقعیت اصلی خود بازمی‌گردد.

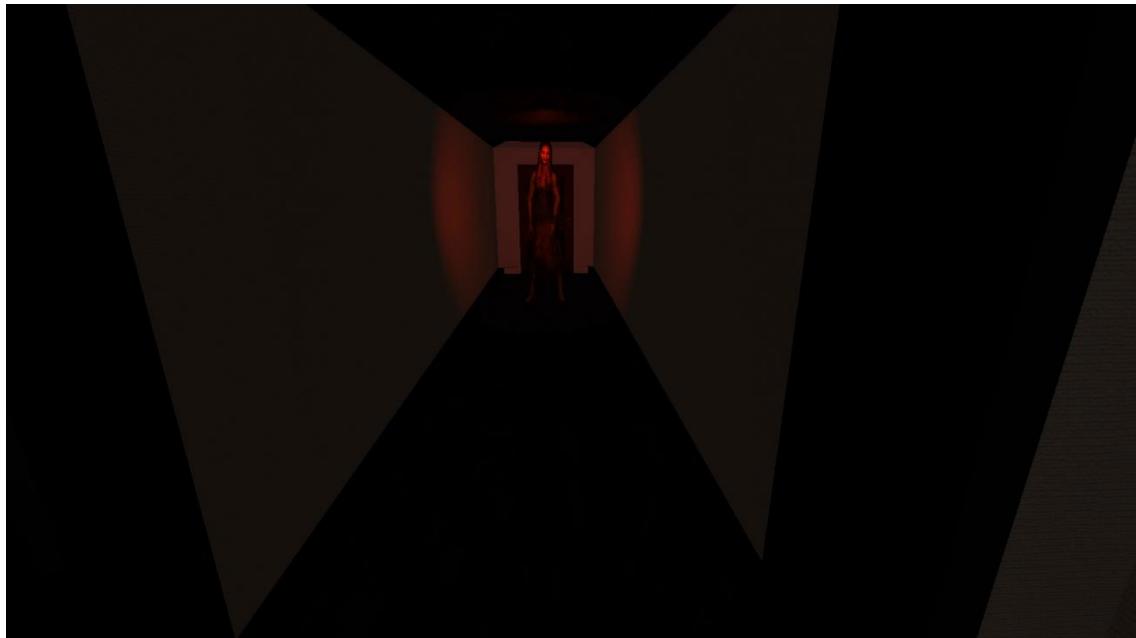
اسکریپت شامل دو متد اصلی برای کنترل لرزش است:

StartShake: این متد برای شروع لرزش فراخوانی می‌شود. زمانی که این متد فعال شود، متغیر isShaking به true تغییر می‌کند که نشان‌دهنده شروع لرزش است و تایمر لرزش به مدت زمان مشخص شده (در اینجا shakeDuration) تنظیم می‌شود.

StopShake: این متد برای توقف لرزش استفاده می‌شود. زمانی که این متد فراخوانی می‌شود، متغیر isShaking به false تغییر می‌کند و تایمر لرزش به صفر می‌رسد. این باعث می‌شود که لرزش

متوقف شده و دوربین به موقعیت اصلی خود بازگردد.

این اسکریپت به ویژه برای بازی‌هایی که در آن‌ها لحظات هیجان‌انگیز یا ترسناک وجود دارد و نیاز به شبیه‌سازی تغییرات محیطی و حرکات دوربین برای ایجاد احساس اضطراب یا تنش دارند، بسیار کاربردی است. به عنوان مثال، در بازی‌های ترسناک، زمانی که یک موجود وحشتناک ظاهر می‌شود یا یک انفجار یا برخورد شدید رخ می‌دهد، می‌توان از این اسکریپت برای لرزش دوربین و ایجاد احساس بیشتر در بازیکن استفاده کرد.



زن ترسناک در انتهای راه رو

بعد از آن شی `please wake up` که قبل از توضیح داده بودم فعال می‌شود و تایم لاین پخش می‌شود، بعد از اتمام تایم لاین هم بازیکن به `Scene` جدید و آخر یعنی `HouseWakeUp` منتقل می‌شود. اسکریپت `TimelineSceneChanger`

```
using UnityEngine;
using UnityEngine.Playables;
using UnityEngine.SceneManagement;

public class TimelineSceneChanger : MonoBehaviour
{
    public PlayableDirector director;
    public AudioSource bgAudio;

    void Start()
    {
        bgAudio.Stop();
```

```

    director.stopped += OnTimelineFinished;
}

void OnTimelineFinished(PlayableDirector obj)
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex +
1);
}
}

```

این اسکریپت برای مدیریت تغییر صحنه پس از اتمام تایم‌لاین در بازی طراحی شده است. در ابتدا، صدای پس‌زمینه متوقف می‌شود تا از پخش موسیقی یا صدای اضافی جلوگیری شود. سپس، با استفاده از رویداد stopped تایم‌لاین، متى به نام OnTimelineFinished به‌طور خودکار پس از پلیان تایم‌لاین فراخوانی می‌شود. در این متى، از SceneManager.LoadScene برای بارگذاری صحنه بعدی استفاده می‌شود و بازی به مرحله یا صحنه بعدی منتقل می‌شود. این اسکریپت به‌ویژه برای بازی‌هایی که از تایم‌لاین‌ها برای انیمیشن‌ها یا رویدادهای خاص استفاده می‌کنند، کاربرد دارد و به‌طور خودکار تغییر صحنه را پس از اتمام تایم‌لاین انجام می‌دهد.

۴.۲-۱۲ - پیاده سازی Satge هفتم

در مرحله‌ی هفتم، تنها اتاق بازیکن نمایش داده می‌شود. برای القای حس روز و آرامش ابتدایی صحنه، از یک Directional Light با تنظیمات نور روز استفاده شده است.

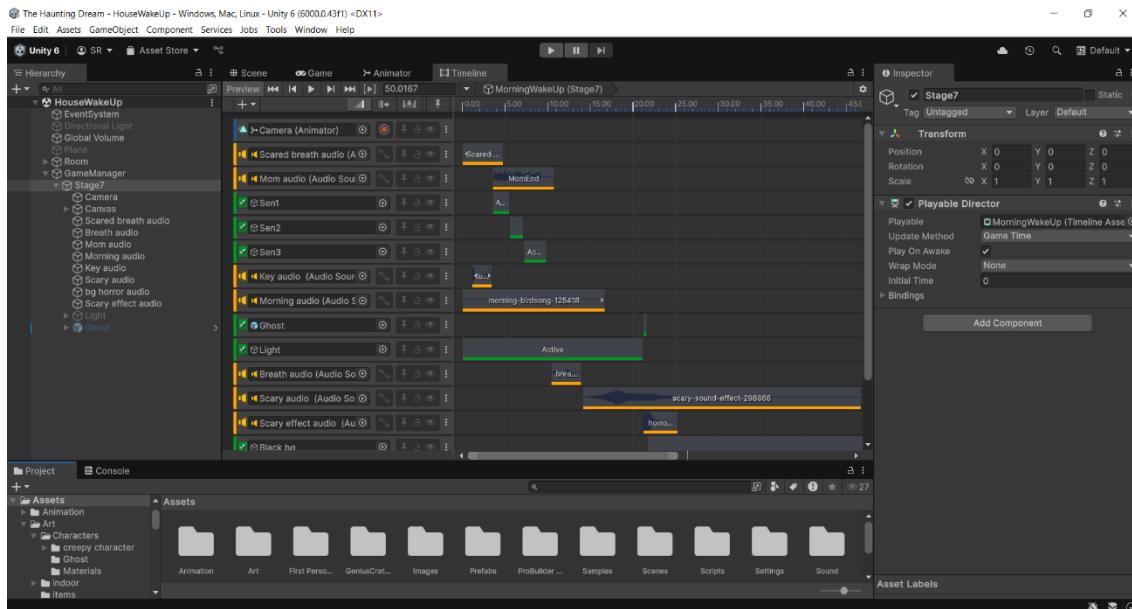
این مرحله صرفاً شامل یک Timeline سینمایی است که در آن بازیکن به‌تدریج از خواب بیدار می‌شود. در طول این Timeline، دوربین فرعی فعال شده و از زوایای مختلف، واکنش‌های بازیکن در هنگام بیدار شدن به‌نمایش گذاشته می‌شود. انیمیشن‌های بدن و سر بازیکن نیز برای نگاه‌کردن به اطراف در نظر گرفته شده‌اند.

در این لحظات، صدای مادر بازیکن شنیده می‌شود که می‌گوید:
«وقت بیدار شدن... امروز جلسه‌ی درمانی داری.»

پس از شنیدن این جمله، بازیکن نفس عمیقی می‌کشد. چند ثانیه بعد، به‌صورت ناگهانی موسیقی‌ای با فضای ترسناک پخش می‌شود و نور اتاق به‌طور کامل خاموش می‌شود.

در لحظه‌ای غافلگیرکننده، نور قرمز تنها بخشی از اتاق را روشن می‌کند و زن ترسناک، ایستاده رو به‌روی تخت بازیکن ظاهر می‌شود. این لحظه نقطه‌ی اوج تنش مرحله است و نشان‌دهنده‌ی آن است

که کابوس هنوز به پایان نرسیده است.



نمایش TimeLine

بعد از آن هم متن تشكیر برای بازی کردن و سخن آخر به نمایش در می آید که باز زدن دکمه Enter به منوی اول بازی باز می گردید. اسکریپت رفتن به منوی اصلی

:EndKey

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;

public class EndKey : MonoBehaviour
{
    private bool canPressKey = false;

    void Start()
    {
        Invoke("EnableKeyPress", 50f);
    }

    void Update()
    {
        if (canPressKey && Input.GetKeyDown(KeyCode.Return))
        {
            SceneManager.LoadScene("Main Menu");
        }
    }

    private void EnableKeyPress()
    {
        canPressKey = true;
    }
}
```

{

}

این اسکریپت برای مدیریت فعال‌سازی فشار دادن کلید [Return] پس از مدت زمان مشخص طراحی شده است. زمانی که بازی شروع می‌شود، بازیکن باید ۵۰ ثانیه صبر کند تا بتواند با فشردن کلید [Return] به صفحه اصلی بازی (Main Menu) منتقل شود. در ابتدا، متغیر EnableKeyPress غیرفعال است، اما پس از گذشت ۵۰ ثانیه از طریق متده canPressKey می‌شود. سپس، در هر فریم بازی بررسی می‌شود که آیا بازیکن قادر به فشردن کلید است یا نه، و اگر این اتفاق بیفتد، بازی به صفحه اصلی منتقل می‌شود. این اسکریپت برای بازی‌هایی که نیاز به وقفه زمانی قبل از انجام یک عمل خاص دارند و بازیکن باید پس از سپری شدن زمان مشخص به صفحه اصلی یا منوها دسترسی پیدا کند، مفید است.

۱۳-۲-۴- خروجی گرفتن از بازی

برای تنظیمات صحنه^۱ و ایجاد آیکون بازی در یونیتی و خروجی گرفتن برای پلتفرم PC، مراحل ساده‌ای وجود دارد که با دنبال کردن آن‌ها می‌توانید بازی خود را آماده ساخت کنید. ابتدا برای تنظیمات صحنه، باید یک صحنه موجود را باز کنید. سپس از منوی File گزینه Build Settings را انتخاب کرده و در این بخش، پلتفرم‌های مختلف نمایش داده می‌شود که می‌توانید پلتفرم PC را انتخاب کنید (Windows, Mac, Linux).

برای اضافه کردن آیکون بازی، ابتدا به منوی Project Settings بروید و گزینه Edit کنید. در بخش Player، آیکون‌هایی برای بازی در اندازه‌های مختلف مانند ۱۶X۱۶، ۳۲X۳۲ و ۶۴X۶۴ وجود دارد که می‌توانید برای هر اندازه، آیکون مورد نظر خود را بارگذاری کنید. برای بارگذاری آیکون، کافی است بر روی دکمه + در کنار اندازه‌های مختلف کلیک کرده و تصویر آیکون را انتخاب کنید.

در مرحله بعد، برای خروجی گرفتن بازی برای PC، دوباره به منوی File بروید و را انتخاب کنید. پس از انتخاب پلتفرم Build PC, Mac & Linux Standalone، بر روی دکمه کلیک کنید تا مسیر ذخیره‌سازی فایل اجرایی بازی را انتخاب کنید. در صورت نیاز به تنظیمات اضافی

¹ Scene

مانند نام بازی و ورژن، می‌توانید بر روی Player Settings کلیک کرده و آن‌ها را تغییر دهید. در نهایت، پس از انجام تمامی تنظیمات مورد نظر، بر روی دکمه Build کلیک کنید تا فرآیند ساخت بازی آغاز شود و پس از اتمام، فایل اجرایی بازی شما آماده خواهد بود. این مراحل به شما کمک می‌کنند تا بازی خود را برای پلتفرم PC خروجی بگیرید، آیکون دلخواه را تنظیم کرده و بازی را برای انتشار آماده کنید.



تصویر آیکون بازی

۴-۳- اجرای نرم‌افزار



فضای داخلی خانه در صحنه اول



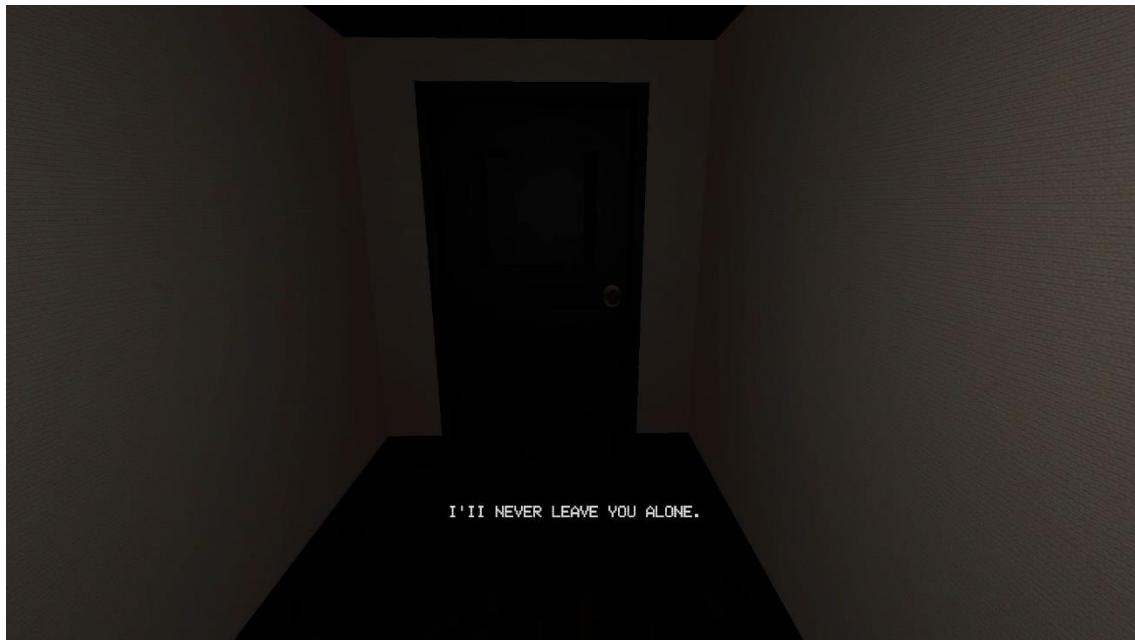
قسمتی از صحنه تدی و کرکتر کابوس



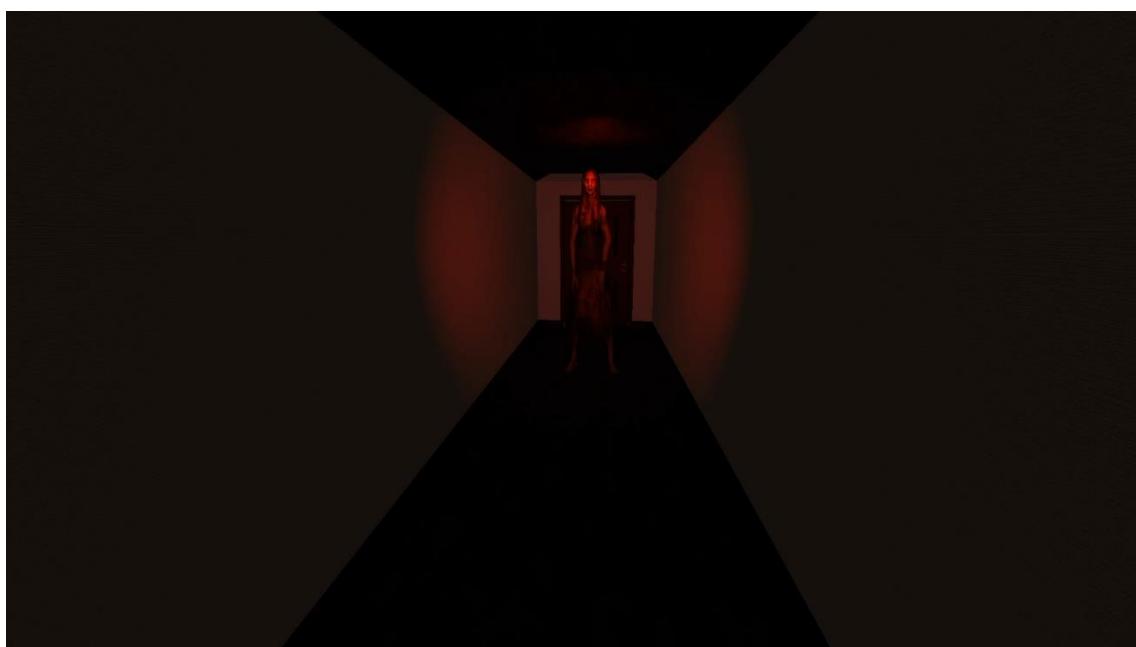
تصویری از نامه روی میز در اتاق پشتی



صحنه‌ای از اتاق مشاور



راهروی خالی و دیالوگ مادر



ایستادن زن ترسناک در انتهای راهرو

□ متن گزینه Credits در منوی اصلی

Made by one person. Just me.

No studio, no team, no budget.

Only time, stress, obsession... and a brain that refuses to shut up.

Writing, design, code, art, sound?

Yep – all me.

Shaghayegh Rahmanieh

If you liked it, cool.

If not... well, I didn't ask.

▣ متن سخن آخر پایان بازی

Thank you for playing The Haunting Dream.

This game was created with love, fear, and a deep curiosity about the dreams that haunt us long after we wake up.

It was fully developed by me, Shaghayegh, as a personal and creative project to explore interactive storytelling, psychological horror, and emotional reflection.

Every moment, every sound, every shadow you experienced was crafted to make you feel, think, and wonder.

If even one moment stayed with you... then this dream was worth dreaming.

– Written, designed, and developed by:

Shaghayegh Rahmanieh

ترجمه فارسی:

ممnon که رویای نفرین شده رو بازی کرده

این بازی با عشق ، ترس و کنجکاوی عمیق در مورد رویاهایی که مدت ها پس از بیدار شدن ما را آزار می دهند ، ایجاد شده است.

این پروژه به طور کامل توسط من ، شقايق ، به عنوان یک پروژه شخصی و خلاقانه برای کشف داستان های تعاملی ، وحشت روانی و بازتاب عاطفی توسعه داده شد.

هر لحظه ، هر صدا ، هر سایه ای که تجربه کردی برای اینکه احساس کنی ، فکر کنی و تعجب کنی ، ساخته شده بود.

اگر حتی یک لحظه با شما ماند... پس این رویا ارزش خواب دیدن را داشت.

-نوشته شده ، طراحی شده و توسعه یافته توسط:

شقايق رحمانيه

۱-۳-۴- دانلود پروژه و خروجی بازی

برای دیدن خروجی بازی می‌توانید از طریق لینک زیر اقدام کنید:

[https://youtu.be/TsbSpEVTdJI¹](https://youtu.be/TsbSpEVTdJI)

یا در صورت خرابی لینک در نرم افزار Youtube عبارت The Haunting Dream: A horror game را جست و جو کنید.

برای دانلود پروژه و دسترسی به فایل‌های پروژه و خروجی می‌توانید از Github یا Google Drive اقدام کنید:

[https://github.com/Shaghayegh-Rahmanieh/The-Haunting-Dream-A-horror-game.git²](https://github.com/Shaghayegh-Rahmanieh/The-Haunting-Dream-A-horror-game.git)

[https://drive.google.com/drive/folders/1yM8HJTccTJOtaxAJI3MIHRQHC GVvsAO9?usp=sharing³](https://drive.google.com/drive/folders/1yM8HJTccTJOtaxAJI3MIHRQHC GVvsAO9?usp=sharing)

۴-۴- نتیجه‌گیری

پروژه The Haunting Dream یک بازی روانشناختی ترسناک است که با هدف بررسی مفاهیم طراحی بازی، برنامه‌نویسی و استفاده از ابزارهای مختلف برای ایجاد تجربه‌ای غوطه‌ور و ترسناک برای بازیکن طراحی شده است. این پروژه به عنوان یک مطالعه موردنی در زمینه استفاده از Unity به عنوان موتور بازی‌سازی، #C# برای برنامه‌نویسی و ابزارهای مختلفی چون Blender برای مدل‌سازی سه‌بعدی و انیمیشن‌ها به کار گرفته شد.

در طول پیاده‌سازی این پروژه، چالش‌های متعددی مانند طراحی و پیاده‌سازی مکانیک‌های گیم‌پلی، ایجاد محیط‌های سه‌بعدی و طراحی انیمیشن‌های روان و تعاملی برای ایجاد احساس ترس و اضطراب در بازیکن به وجود آمد. همچنین، پیاده‌سازی سیستم‌های هوش مصنوعی برای تعامل با بازیکن و ایجاد یک تجربه دلهره‌آور و طبیعی از دیگر چالش‌های پروژه بود. با استفاده از PlayableDirector در

¹ Youtube

² Github

³ Google drive

Unity، تایم‌لین‌های تعاملی و انیمیشن‌های خاص در بازی به‌طور مؤثر به کار گرفته شدند تا تجربه بازی به‌طور داینامیک و سینماتیک جذاب‌تر شود.

این پروژه همچنین نشان‌دهنده توانایی استفاده از گرافیک‌های سه‌بعدی و صداگذاری حرفه‌ای برای ایجاد جو و فضای بازی‌های ترسناک بود. طراحی و پیاده‌سازی پازل‌ها و چالش‌های ذهنی و فیزیکی در طول بازی نیز از دیگر جنبه‌های مهم این پروژه بود که به بازیکن امکان حل معماها و تعامل با محیط بازی را می‌داد. علاوه بر این، بخش‌های مختلف مانند مدیریت وضعیت‌ها، تنظیمات صحنه‌ها و آیکون‌ها، و خروجی گرفتن بازی برای پلتفرم‌های مختلف مانند PC به‌طور دقیق و با استفاده از ابزارهای یونیتی و امکانات آن طراحی و پیاده‌سازی شد.

فصل ۵:

جمع‌بندی و پیشنهاد‌ها

۱-۵- نتیجه‌گیری

پروژه The Haunting Dream به عنوان یک بازی روان‌ساختی ترسناک طراحی و پیاده‌سازی شد. اهداف و کارهای اصلی انجام‌شده در این پروژه شامل موارد زیر است:

برای ایجاد یک بازی ترسناک با گرافیک‌های سه‌بعدی طراحی و توسعه بازی: استفاده از Unity و C#.

مکانیک‌های گیم‌پلی: پیاده‌سازی معماها، چالش‌های ذهنی و فیزیکی برای بازیکن.

سیستم‌های هوش مصنوعی: طراحی دشمنان با هوش مصنوعی که به رفتار بازیکن واکنش نشان می‌دهند و او را تعقیب می‌کنند.

تایم‌لاین‌ها و انیمیشن‌ها: استفاده از PlayableDirector در Unity برای انیمیشن‌های تعاملی.

صدای‌گذاری: استفاده از صدای محیطی و موسیقی برای ایجاد حس ترس و اضطراب.

▣ مشکلات و چالش‌ها

چالش‌ها و مشکلات مختلفی در طول فرآیند توسعه وجود داشت که برخی از آن‌ها مربوط به نورپردازی، مدیریت منابع و حجم بالای مدل‌ها بود. این موارد به ویژه در ایجاد یک تجربه روان و بدون نقص برای بازیکن تأثیرگذار بودند. در ادامه برخی از این چالش‌ها بررسی شده است:

• چالش در نورپردازی:

نورپردازی یکی از مؤلفه‌های کلیدی در ایجاد جو و فضای ترسناک بازی بود، اما در یونیتی، پیاده‌سازی نورپردازی پویا و بهینه برای محیط‌های پیچیده، چالش‌برانگیز بود. به ویژه، زمانی که چندین منبع نور و سایه‌های پویا در صحنه وجود داشت، مشکلاتی مانند کاهش کارایی و افت فریم به وجود می‌آمد.

نورپردازی در بازی‌های ترسناک باید به گونه‌ای باشد که فضای تاریک و مرموز نگه دارد، اما در عین حال محیط برای بازیکن قابل تشخیص باشد. بنابراین، بهینه‌سازی منابع نور و ایجاد سایه‌های نرم و مناسب برای محیط‌های مختلف یکی از چالش‌های مهم در این پروژه بود.

• مدیریت منابع و حجم بالای مدل‌ها:

یکی دیگر از مشکلات قابل توجه، حجم بالای مدل‌های سه‌بعدی و منابع مختلف استفاده شده در بازی بود. مدل‌های پیچیده و با جزئیات زیاد باعث افزایش حجم بازی و مصرف منابع بیشتری از پردازنده و

حافظه می‌شدند. این موضوع در هنگام اجرای بازی به‌ویژه در صحنه‌های شلوغ و پر از مدل‌های سه‌بعدی، باعث کاهش عملکرد و افت فریم می‌شد. علاوه بر این، به دلیل محدودیت‌های سیستم و سخت‌افزار، لازم بود تا مدل‌ها و بافت‌ها بهینه شوند تا بازی بر روی پلتفرم‌های مختلف به درستی اجرا شود.

• چالش در تکمیل مراحل و هماهنگی اجزا:

تکمیل مراحل بازی و هماهنگی بین اجزای مختلف، مانند گرافیک، انیمیشن‌ها، صدا و مکانیک‌های گیم‌پلی، یک چالش بزرگ بود. برای ایجاد تجربه‌ای روان و بدون نقص، لازم بود که همه‌ی این اجزا به درستی با هم هماهنگ شوند. این هماهنگی در مراحل مختلف بازی به‌ویژه در بخش‌هایی که تایم‌لاین‌ها و انیمیشن‌های تعاملی به کار رفته بودند، دشوار بود. ایجاد لحظاتی که در آن‌ها بازیکن به‌طور کامل در جو و فضای ترسناک بازی غوطه‌ور شود، نیازمند زمان و تلاش زیادی بود تا تمام جزئیات به‌درستی همگام‌سازی شوند.

• بهینه‌سازی عملکرد بازی:

به دلیل حجم بالای مدل‌ها و جزئیات سه‌بعدی، بازی به منابع پردازشی زیادی نیاز داشت. برای اطمینان از عملکرد روان بازی، تکنیک‌های مختلفی برای بهینه‌سازی استفاده شد. این تکنیک‌ها شامل کاهش جزئیات مدل‌ها، استفاده از سیستم‌های نورپردازی پویا به‌صورت بهینه، و مدیریت تعداد اشیاء فعال در هر فریم بود. این اقدامات به کاهش مصرف پردازنده و حافظه و جلوگیری از افت فریم کمک کردند.

• طراحی سیستم‌های پیچیده هوش مصنوعی:

دشمنان بازی باید به رفتارهای بازیکن واکنش نشان دهند و او را تعقیب کنند. طراحی هوش مصنوعی برای این دشمنان نیازمند الگوریتم‌های پیچیده بود که باید به‌طور مؤثر و بدون ایجاد افت عملکرد اجرا می‌شدند. استفاده از NavMesh برای حرکت دشمنان و پیاده‌سازی واکنش‌های طبیعی و هوشمندانه به رفتار بازیکن از جمله چالش‌هایی بود که باید به دقت پیاده‌سازی می‌شد.

• هماهنگی انیمیشن‌ها و صدایها:

برای ایجاد تجربه‌ای غوطه‌ور کننده و روان، انیمیشن‌های دشمنان، حرکات دوربین، و صدای‌گذاری‌ها باید

به طور دقیق هماهنگ می‌شدند. به‌ویژه در صحنه‌هایی که تغییرات فوری نیاز بود، هماهنگی دقیق بین ایمیشن‌ها، نورپردازی و صداها اهمیت زیادی داشت. استفاده از PlayableDirector و تایملاین‌ها برای همگام‌سازی این اجزا به‌طور تعاملی و بدون تأخیر یکی از چالش‌های مهم بود.

۲-۵- پیشنهادهایی برای کارهای آتی

در طول انجام پروژه The Haunting Dream با وجود پیشرفت‌های قلبل توجه و پیاده‌سازی ویژگی‌های اصلی بازی، برخی از کارها به دلیل محدودیت‌های زمانی یا منابع قادر به تکمیل نشدند. این موارد می‌توانند در آینده بهبود یابند یا تکمیل شوند تا بازی تجربه‌ای جامع‌تر و جذاب‌تر برای بازیکن فراهم کند. در ادامه به برخی از این کارها اشاره شده است که می‌توان در آینده به آن‌ها پرداخته و توسعه داد:

- **گسترش هوش مصنوعی دشمنان:**

در حال حاضر، دشمنان بازی به‌طور پایه‌ای به رفتار بازیکن واکنش نشان می‌دهند و او را تعقیب می‌کنند، اما می‌توان هوش مصنوعی را پیچیده‌تر کرد. به عنوان مثال، اضافه کردن رفتارهای متنوع‌تر مانند پنهان شدن دشمنان، تغییر مسیر به‌طور تصادفی یا استفاده از قابلیت‌های ویژه می‌تواند تجربه بازی را جذاب‌تر کند. همچنین، ایجاد دشمنانی با استراتژی‌های پیچیده‌تر و هوش مصنوعی که قادر به یادگیری رفتارهای بازیکن باشد، می‌تواند چالش‌های جدیدی به بازی اضافه کند.

- **اضافه کردن مراحل بیشتر و تنوع محیط‌ها:**

بازی تنها شامل تعدادی محیط و مراحل خاص است. با افزودن مراحل بیشتر، تنوع محیط‌ها و طراحی سطوح جدید، می‌توان تجربه بازی را گسترش داد. این مراحل جدید می‌توانند محیط‌های مختلف، معماهای متفاوت و تهدیدات جدید برای بازیکن ارائه دهند. طراحی مکان‌های جدید با چالش‌های خاص به‌ویژه در دنیای بازی‌های ترسناک، می‌تواند عمق بیشتری به داستان و گیم‌پلی بازی اضافه کند.

• بهبود صداگذاری و افکت‌های صوتی:

با وجود استفاده از صداگذاری و موسیقی برای ایجاد جو ترس، هنوز فرصت‌هایی برای بهبود و گسترش این بخش وجود دارد. افزودن افکت‌های صوتی بیشتر برای تعاملات بازیکن، صدای محیطی پیچیده‌تر و استفاده از تکنیک‌های پیشرفته‌تر برای صداگذاری می‌تواند تجربه شنیداری بازیکن را بهبود بخشد و تاثیر بیشتری بر ترس و اضطراب بازیکن بگذارد.

• پیاده‌سازی سیستم انتخاب‌های بازیکن و تأثیرات آن بر داستان:

یکی از ویژگی‌هایی که می‌توان در آینده به بازی اضافه کرد، سیستم انتخاب‌های بازیکن و تأثیر آن‌ها بر روند داستان است. به عنوان مثال، ایجاد انتخاب‌های مختلف برای بازیکن که بر مسیر بازی، پایان‌های مختلف یا تعاملات با شخصیت‌ها تأثیر بگذارد، می‌تواند ارزش تکرار بازی را افزایش دهد و تجربه‌ای تعاملی‌تر و شخصی‌سازی‌شده‌تر برای بازیکن فراهم کند.

• بهینه‌سازی بیشتر برای عملکرد بهتر:

با توجه به حجم بالای مدل‌ها و پیچیدگی‌های گرافیکی، بهینه‌سازی عملکرد بازی نیازمند توجه بیشتر است. استفاده از تکنیک‌های جدیدتر برای بهینه‌سازی منابع، کاهش بار پردازشی و بهبود عملکرد در محیط‌های پیچیده‌تر می‌تواند کمک کند تا بازی به طور روان‌تری اجرا شود، بهویژه در سیستم‌های با سخت‌افزار پایین‌تر.

• اضافه کردن پشتیبانی از پلتفرم‌های مختلف:

در حال حاضر، بازی فقط برای PC طراحی شده است، اما در آینده می‌توان آن را برای پلتفرم‌های دیگر مانند کنسول‌ها (PlayStation, Xbox) و موبایل نیز توسعه داد. این کار نیازمند انجام تغییرات در تنظیمات ورودی، بهینه‌سازی منابع و تطبیق با ویژگی‌های خاص هر پلتفرم است.

منابع

فهرست منابع

◆ منبع یادگیری

[1] <https://www.youtube.com/>

◆ منبع دانلود Asset

[2] <https://sketchfab.com/>

[3] <https://assetstore.unity.com/>

[4] <https://www.cgtrader.com/>

[5] <https://www.fab.com/>

◆ منبع دانلود متریال

[6] <https://www.sketchuptextureclub.com/>

◆ منبع افکت های صوتی

[7] <https://elevenlabs.io/>

[8] <https://pixabay.com/>

Abstract

The aim of this project is to design and implement a psychological horror game named The Haunting Dream, which serves as a final project for the Bachelor's degree in Software Engineering. This game is designed to provide a terrifying and immersive experience for the player, where they must navigate through a dark and mysterious world, solve puzzles, and face terrifying creatures along with mental and physical challenges. The scope of this research involves using the Unity game engine for game development and C# for designing game mechanics. Additionally, Blender is used for 3D modeling and animation.

The methodology of this project includes the design of interactive 3D environments using various graphical tools and the application of advanced artificial intelligence techniques to design enemies. The enemies in the game react to the player's behavior and are designed to chase the player in a dynamic and intelligent manner. Furthermore, Unity's timeline and interactive animations are used to create cinematic events and enhance the horror atmosphere.

The findings of this project indicate that combining 3D graphics, sound design, and artificial intelligence can provide an engaging and nerve-wracking experience for the player. However, challenges such as optimizing game performance, coordinating animations and sounds, and managing high-volume graphical resources were present, which were solved through optimization techniques and the use of various tools.

The conclusion of this project shows that designing psychological horror games using advanced techniques and precise implementation can offer players interactive and immersive experiences. This project also serves as a foundation for future development, such as expanding player choices, enhancing artificial intelligence, and further developing game environments.

Keywords: Psychological horror game, Unity, C#, Blender, Artificial Intelligence, Timeline, Performance optimization



Technical and Vocational University
Shariaty Technical College

Technical and Vocational University
Shariaty Technical College

A Dissertation Submitted in Partial Fulfillment for the Degree of Bachelor
of Science in Electrical/ICT/Software Engineering

The Haunting Dream: A horror game

By:

Shagheygh Rahmanieh

Supervisor:

Dr Sanaz Afshari

July 2025