WYDZIAŁ MATEMATYKI
i INFORMATYKI
Uniwersytet Łódzki

Paweł Kumorowski

Nr albumu: 378452

Projekt i wykonanie gry wideo "Treasure Hunter" w silniku Unity

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra Marka Badury
w Katedrze Geometrii

Łódź 2023

Spis treści

Wstęp 4
Cel i założenia 4
Terminologia 4
Zawartość rozdziałów 4
Rozdział I. Wprowadzenie 5
Wykorzystane narzędzia 5
Szersze omówienie projektu 5
Przykłady podobnych gier 8
Rozdział II. Dokumentacja programistyczna 9
Struktura projektu 9
Zawartość folderów 9
Spis skryptów 10
Omówienie wybranych rozwiązań programistycznych 11
Generowanie labiryntu pomieszczeń 11
Losowanie szans na pojawienie się konkretnego przeciwnika 14
Sztuczna inteligencja przeciwników 15
Zmiana pomieszczeń 18
Ruch kamery 18
Zapis stanu gry 19
Rozwiązania graficzne 20
Oświetlenie 20
Modele 22
Efekty 23
Rozdział III. Dokumentacja użytkownika 24
Instalacja i pierwsze uruchomienie 24
Interfejs i sterowanie 24

Rozgrywka 26
Walka 28
Przeciwnicy 30
Zakończenie 31
Bibliografia 32

Wstęp

Cel i założenia

Celem pracy było zaprojektowanie oraz wykonanie gry wideo z gatunku roguelite. Głównymi aspektami, na które postawiono nacisk były proceduralne generowanie scenariuszy rozgrywki oraz zróżnicowanie przebiegu kolejnych podejść.

Projekt miał powstać z użyciem darmowych narzędzi oraz assetów dostępnych w Internecie. Miał on zawierać wszystkie funkcjonalności finalnej wersji gry w stopniu co najmniej podstawowym, wraz z gotowym fundamentem pod dalszy rozwój dostępnej zawartości.

W dzisiejszych czasach gatunek gier roguelite jest bardzo popularny. Powstaje wiele tytułów o zróżnicowanej tematyce, stale wnoszących świeże rozwiązania pod kątem rozgrywki. Gry tworzone są często z wykorzystaniem już gotowych silników, np. Unity lub Unreal Engine. Są one bardzo rozbudowane oraz stale rozwijane, zawierają więc najnowsze rozwiązania techniczne oraz w dużym stopniu upraszczają oraz przyspieszają proces twórczy.

Terminologia

- Roguelite – gatunek gier wideo charakteryzujący się m.in. proceduralnym generowaniem świata oraz wysoką regrywalnością.
- Regrywalność – pojęcie określające jak bardzo atrakcyjne dla gracza jest kolejne przejście gry.
- Assety – gotowe elementy udostępnione do wykorzystania w projekcie.
- Prefaby – przygotowane wcześniej obiekty składające się z wielu komponentów, gotowe do wielokrotnego użycia.

Zawartość rozdziałów

1. Wprowadzenie – dokładne omówienie celów i założeń pracy, wykorzystanych narzędzi oraz przykłady podobnych gier wideo.
2. Dokumentacja programistyczna – opis skryptów i rozwiązań, które są najistotniejsze z perspektywy tematu pracy, ogólna struktura pracy.
3. Dokumentacja użytkownika – instalacja, uruchomienie, poradnik oraz opis rozgrywki.

Rozdział I. Wprowadzenie

Wykorzystane narzędzia

Gra „Treasure Hunter” została wykonana w Unity. Silnik ten jest darmowy do celów niekomercyjnych, posiada obszerną dokumentację oraz bazę poradników tworzonych przez użytkowników. Istotnym jego elementem jest również integracja z Unity Asset Store, czyli bazą assetów gotowych do natychmiastowego wykorzystania.

Skrypty w Unity pisane są w języku C#. Do tego celu wykorzystano środowisko Visual Studio, które zawiera szereg ułatwień w ich tworzeniu, m.in. IntelliSense, czyli pomoc w uzupełnianiu kodu w postaci podpowiedzi, debugowanie, czy też możliwość uruchomienia go z poziomu Unity.

Do przechowywania historii oraz kopii zapasowej skorzystano z systemu kontroli wersji GIT oraz programu GIT Bash. Ułatwiło to również synchronizację pracy na różnych urządzeniach bez konieczności każdorazowego przenoszenia całego projektu.

Niektóre gotowe modele wymagały drobnych poprawek, do tego celu wykorzystano darmowy program Blender.

Szersze omówienie projektu

Gatunek roguelite zawiera kilka istotnych elementów w kontekście regrywalności. Duża ilość elementów losowych, tj. układ świata, przeciwnicy, przeszkody, czy chociażby niespodziewane wydarzenia, razem tworzą ogromną ilość możliwych kombinacji, które gracz może napotkać w każdej rozgrywce. Innym ważnym aspektem jest stopniowe odblokowywanie bądź ulepszanie ekwipunku, statystyk oraz umiejętności. Pozwala to na docieranie z każdym podejściem coraz dalej oraz daje dodatkowy cel w postaci poczucia rozwoju bohatera. W grze „Treasure Hunter” starano się oddać charakter wyżej wymienionych cech implementując owe systemy realizujące założenia gatunku roguelite.

W projekcie skupiono się w pierwszej kolejności na wpływie losowości na przebieg rozgrywki, zaimplementowano więc kilka systemów na niej opartych:

- Generowanie labiryntu pomieszczeń przy użyciu algorytmu DFS
- Wybór wnętrza dla każdego z pomieszczeń oraz jego losowa rotacja
- Pojawianie się opcjonalnych przeszkód
- Wybór scenariusza rozgrywki dla każdego pomieszczenia

- Przypisanie pomieszczeniom typów przeciwników i szans na ich pojawienie się

Aby rozgrywka była atrakcyjna z perspektywy gracza, musiała zostać dodana odpowiednia ilość zawartości, na którą składają się:

- 6 różnych pomieszczeń
- 5 typów przeciwników oraz łącznie 11 dostępnych wariantów
- 7 rodzajów broni oraz pancerz
- 4 dostępne scenariusze rozgrywki

Projekt posiada również gotową bazę pozwalającą na dodawanie nowej zawartości bez ingerencji w silnik gry.

Rysunek 1. Fragment rozgrywki

Rozgrywka polega na ciągłym pokonywaniu hord potworów, z tego powodu również element walki wymagał odpowiedniej uwagi. Aby jej przebieg zależał w większym stopniu od umiejętności gracza, poza standardowymi statystykami decydującymi o przewadze dodano możliwość uniku. Dzierżona w danej chwili broń wpływa na poruszanie się, efektywność uników oraz styl gry. Gra w jasny sposób sygnalizuje odnoszone obrażenia poprzez migotanie postaci a celne trafienia przeciwników odrzucają ich z siłą zależną od siły ataku gracza i ich masy. Przeciwnicy kontrolowani są przez sztuczną inteligencję a do wyznaczania ich ścieżek poruszania skorzystano z systemu NavMesh.

Rysunek 2. Przeciwnicy

Gra toczy się w pomieszczeniach, gdzie po wejściu wszystkie drzwi zostają zamykane. Aby przejść dalej i ponownie je otworzyć należy spełnić odpowiednie dla danego pomieszczenia warunki. Obecnie zaimplementowane scenariusze obejmują:

- Fale – po pokonaniu wszystkich przeciwników pojawiają się kolejni, do momentu wyczerpania wszystkich fal.
- Przetrwanie – przez określony czas co chwilę pojawiają się małe grupy przeciwników zmierzające w kierunku gracza.
- Pochodnie – należy odnaleźć i zapalić wszystkie pochodnie, do tego momentu będą ciągle pojawiali się przeciwnicy.
- Boss – walka z potężnym finałowym przeciwnikiem. Co 25% jego poziomu zdrowia pojawiają się fale dodatkowych przeciwników a każda kolejna jest silniejsza od poprzedniej.

Podczas rozgrywki gracz gromadzi monety, które pozwalają mu na ulepszanie oraz odblokowywanie arsenału broni. Wymagało to stworzenia prostego systemu zapisu pozwalającego na zachowywanie osiągnięć podczas kolejnych sesji gry.

Rysunek 3. Dostępna broń

Do stworzenia wszelkich elementów dźwiękowych oraz graficznych, tj. modele, animacje, tekstury, zostały wykorzystane darmowe rozwiązania. Jest to jednak element poboczny w tym projekcie, pozwalający jedynie na zobrazowanie działania wyżej opisanych systemów gry.

Przykłady podobnych gier

- „Hades” – Supergiant Games
- „Cult of the Lamb” – Massive Monster
- „The Binding of Isaac” – Edmund McMillen

Rozdział II. Dokumentacja programistyczna

Struktura projektu

Do stworzenia „Treasure Hunter” została użyta jedna scena, na której od razu rozpoczyna się rozgrywka. Na początku poprzez skrypt generowane są elementy świata oraz wczytywane dane z zapisu poprzedniej rozgrywki. Każdy koniec gry, niezależnie czy zwycięstwo, czy porażka dokonuje zapisu oraz restartuje scenę.

Unity pozwala na przypisywanie wartości zmiennym bezpośrednio w edytorze więc wszystkie niezbędne ustawienia pozwalające na dostosowywanie parametrów rozgrywki zostały tam udostępnione.

Elementy świata wykorzystywane w grze zostały przygotowane w formie prefabów. Część z nich będąca otoczeniem została użyta do utworzenia wnętrz pomieszczeń, natomiast pozostałe tworzone są skryptami w toku gry i należą do nich m.in.: gracz, przeciwnicy, bronie, pociski.

Skrypty, które wykorzystywane są w wielu miejscach zostały zaimplementowane w postaci singletonów. Upraszcza to późniejsze odwoływanie się do nich w kodzie, eliminując konieczność ustawiania ich referencji w edytorze. Ogranicza to również ilość błędów wynikających ze zwykłych przeoczeń.

Zawartość folderów

- Animations – humanoidalne animacje używane przez różne jednostki
- Images – graficzne elementy interfejsu
- Imported – pobrane assety
- Materials – wybrane materiały i tekstury
- Prefabs – przygotowane prefaby
- Scenes – scena gry
- Scripts – skrypty
- Settings – ustawienia Universal Render Pipeline
- Sounds – dźwięki
- TextMeshPro – importowaną paczkę obsługującą tekst interfejsu

Spis skryptów

- BulletController – kontroluje zachowanie pocisków.
- CameraMovement – ustawia miękkie śledzenie gracza kamerą.
- ChestController – dotyczy zachowania finalnej skrzyni z nagrodą oraz restartu gry po jej otworzeniu.
- DestroyableScript – pozwala na niszczenie niektórych elementów otoczenia.
- Enemies – pomocniczy skrypt losujący szanse pojawienia się konkretnych przeciwników.
- EnemyController – definiuje zachowanie przeciwników.
- LevelGenerator – przygotowuje świat gry tworząc labirynt pomieszczeń.
- MeleeHitbox – obsługuje kolizje w walce wręcz z perspektywy gracza.
- PlayerCombat – zawiera statystyki gracza oraz zachowania związane z walką.
- PlayerController – odpowiada za obsługę przycisków kontrolujących gracza, poruszanie, uniki, możliwość interakcji z otoczeniem.
- PlayerInventory – system ekwipunku gracza, zawiera ilość posiadanych monet, broń, pancerz.
- PlayerSpawn – tworzy postać gracza w pomieszczeniu startowym.
- RoomChange – pomocniczy skrypt przechowujący informację o obecnym pomieszczeniu.
- RoomController – kontroluje zachowanie pomieszczeń oraz scenariuszy rozgrywki.
- RoomGenerator – umożliwia tworzenie pomieszczeń i ustawianie przejść między nimi.
- Save – zapisuje oraz odczytuje dane z pliku zapisu.
- SoundController – przechowuje oraz odtwarza dźwięki.
- TorchController – obsługuje pochodnie będące jednym ze scenariuszy rozgrywki.
- UIController – obsługuje interfejs.

Omówienie wybranych rozwiązań programistycznych

Generowanie labiryntu pomieszczeń

Rysunek 4. Przykład wygenerowanego labiryntu

Generowanie labiryntu pomieszczeń odbywa się z wykorzystaniem algorytmu DFS. W pierwszej kolejności tworzona jest siatka, przechowywana w jednowymiarowej liście. Składa się ona z komórek, które mają potencjalną szansę na bycie pomieszczeniem.

```
board = new List<Cell>();

for (int y = 0; y < size.y; y++)
    {
    for (int x = 0; x < size.x; x++)
        {
            board.Add(new Cell());
        }
    }
```

Komórkę reprezentuje klasa Cell posiadająca klasę wyliczeniową Direction, wartość logiczną visited określającą czy została już odwiedzona oraz tablicę status informującą o stronach dostępnych przejść.

```
public class Cell
{
    public enum Direction { Up, Down, Left, Right }
    public bool visited;
    public bool[] status = new bool[4]; //up, down, left, right
}
```

Następnie jako pierwszą przeglądaną komórkę ustawia się wcześniej wybraną, będącą startem labiryntu. Ścieżka będzie przechowywana w strukturze stosu.

```
int currentCell = mazeStart;

Stack<int> path = new Stack<int>();
```

Warunkami końca są dotarcie do pomieszczenia wybranego jako finalne oraz odwiedzenie maksymalnej liczby pomieszczeń. Każda przeglądana komórka dodawana jest do ścieżki, lecz jeżeli nie ma żadnych wolnych sąsiadów a nie jest jednocześnie końcem labiryntu, nie może być jej częścią i zostaje zdjęta ze stosu. Kolejną rozpatrywaną w pętli komórką jest losowo wybierana z dostępnych sąsiadów.

```
while (true)
{
    board[currentCell].visited = true;

    if (currentCell == mazeEnd || path.Count == mazeMaxLength)
    {
        mazeEnd = currentCell;
        break;
    }

    List<int> neighbors = CheckNeighbors(currentCell);

    if (neighbors.Count == 0)
    {
        if (path.Count == 0)
        {
            break;
        }
        else
        {
            currentCell = path.Pop();
        }
    }
    else
    {
        path.Push(currentCell);
        int newCell = neighbors[Random.Range(0, neighbors.Count)];
        AddPath(currentCell, newCell);
        currentCell = newCell;
    }
}
```

Funkcja CheckNeighbors zwraca wszystkich sąsiadów danej komórki.

```
List<int> CheckNeighbors(int cell)
{
    List<int> neighbors = new List<int>();

    //up
    if (cell + size.x < board.Count)
    {
        neighbors.Add(cell + size.x);
    }
    //down
    if (cell - size.x >= 0)
    {
        neighbors.Add(cell - size.x);
    }
    //left
    if (cell % size.x != 0)
    {
        neighbors.Add(cell - 1);
    }
    //right
    if ((cell + 1) % size.x != 0)
    {
        neighbors.Add(cell + 1);
    }

    return neighbors;
}
```

Funkcja AddPath dodaje połączenia między komórkami (pokojami). Sprawia, że odpowiednie drzwi w pokoju zostają otwarte umożliwiając przejście do jego sąsiadów. Na koniec w miejscu odwiedzonych komórek tworzone są konkretne pomieszczenia. Jeżeli dana komórka jest komórką startową, tworzone jest pomieszczenie startowe, jeżeli natomiast ostatnią, tworzone jest finalne pomieszczenie (pomieszczenie typu boss). Wszystkie pozostałe pomieszczenia są pomieszczeniami default, czyli tymi ze standardowymi zasadami rozgrywki.

```
GameObject startingRoom = null;
for (int y = 0; y < size.y; y++)
{
    for (int x = 0; x < size.x; x++)
    {
        int cellID = x + y * size.x;
        if (board[cellID].visited)
        {
            if (cellID == mazeStart)
            {
                startingRoom = roomGenerator.GenerateRoom(
                   RoomGenerator.RoomType.Start,
                   new Vector3(x * offset.x, 0, y * offset.y),
                   board[cellID].status);
            }
            else if(cellID == mazeEnd)
            {
                roomGenerator.GenerateRoom(
                   RoomGenerator.RoomType.Boss,
                   new Vector3(x * offset.x, 0, y * offset.y),
                   board[cellID].status);
            }
            else
            {
                roomGenerator.GenerateRoom(
                   RoomGenerator.RoomType.Default,
                   new Vector3(x * offset.x, 0, y * offset.y),
                   board[cellID].status);
            }
        }
    }
}
```

Losowanie szans na pojawienie się konkretnego przeciwnika

Przy tworzeniu każdego pomieszczenia przypisywane są mu dwa możliwe typy przeciwników, których można w nim napotkać.

```
enemiesByType.Clear();
enemiesByType.Add(skeletons);
enemiesByType.Add(goblins);
enemiesByType.Add(spiders);
enemiesByType.Add(golems);

string probabilities = "";

GameObject[] primaryType = enemiesByType[UnityEngine.Random.Range(0,
enemiesByType.Count)];
enemiesByType.Remove(primaryType);
GameObject[] secondaryType = enemiesByType[UnityEngine.Random.Range(0,
enemiesByType.Count)];
```

Na początku inicjujemy listę enemiesByType wszystkimi możliwymi typami przeciwników, co pozwala następnie wylosować dwa z nich, podstawowy i poboczny.

Typ podstawowy ma większą szansę na pojawienie się niż poboczny, natomiast pozostałe mają na to zerową szansę. Wszyscy możliwi przeciwnicy są przechowywani w tablicy allEnemies. Ich id posłuży do zbudowania stringa probabilities, powtarzającego daną wartość id ilość razy zależną od szansy na pojawienie się tej konkretnej jednostki. Następnie wybierając wartość pod losowym miejscem w tym stringu otrzymuje się id konkretnej jednostki zgodnie z przypisaną jej szansą.

```
foreach (GameObject e in primaryType)
{
    int arrayIndex = Array.IndexOf(allEnemies, e);
    int spawnChance = UnityEngine.Random.Range(1, 5);
    for (int i = 0; i < spawnChance * primaryToSecondaryRatio; i++)
        probabilities += arrayIndex.ToString();
}

foreach (GameObject e in secondaryType)
{
    int arrayIndex = Array.IndexOf(allEnemies, e);
    int spawnChance = UnityEngine.Random.Range(0, 5);
    for (int i = 0; i < spawnChance; i++)
        probabilities += arrayIndex.ToString();
}

return probabilities;
```

Sztuczna inteligencja przeciwników

Przeciwne jednostki kontrolowane są przez sztuczną inteligencję. Do poruszania się korzystają z systemu NavMesh. Posiadają dwa stany zaalarmowania, zależnie od tego czy gracz znajdzie się w ich zasięgu wzroku.

```
IEnumerator ScanAlert()
{
    while(true)
    {
        RaycastHit hit;
        Physics.Raycast(transform.position,
           towardsPlayer, out hit, alertRange, layerMask);

        if (hit.collider && hit.collider.CompareTag("Player"))
        {
            isAlerted = true;
            StopCoroutine(scanCoroutine);
            SoundController.Instance.PlayRandomSound(sounds,
               transform.position, 0.5f);
        }

        yield return new WaitForSeconds(1/scanRate);
    }
}
```

Po zaalarmowaniu cel ich ścieżki ustawiany jest na pozycję gracza oraz aktywnie sprawdzane jest czy gracz jest w zasięgu ataku oraz na linii strzału. Jeżeli oba warunki są spełnione rozpoczynają procedurę ataku, w przeciwnym razie starają się dotrzeć jak najbliżej postaci gracza.

```
RaycastHit hit;
float raycastRange;
if (isRanged)
    raycastRange = rangedAttackRange;
else
    raycastRange = meleeAttackRange;

Physics.Raycast(transform.position, towardsPlayer, out hit, raycastRange,
layerMask);
if (hit.collider != null && hit.collider.CompareTag("Player"))
{
    animator.SetFloat("Running", 0);
    Attack();
}
else
{
    if(isMelee)
        animator.SetFloat("Attack_Melee", 0);
    if(isRanged)
        animator.SetFloat("Attack_Ranged", 0);

    animator.SetFloat("Running", agent.velocity.magnitude);
    Move();
}
```

Po każdorazowym otrzymaniu jakiejkolwiek ilości obrażeń, sprawdzany jest stan zdrowia. Jeżeli spadnie on poniżej zera przeciwnik umiera. Odtwarzana jest wtedy animacja śmierci a zwłoki pozostają na scenie przez określoną ilość czasu.

```
IEnumerator IsAlive()
{
    if (currentHealth <= 0)
    {
        UIController.Instance.RemoveEnemy();

RoomChange.currentRoom.GetComponent<RoomController>().KillEnemy(gameObject);
        animator.SetBool("IsDead", true);
        isAlive = false;
        PlayerInventory.Instance.AddCoins(reward);

        yield return new WaitForSeconds(corpseDuration);
        Destroy(gameObject);
    }
}
```

Istnieją dwa typy ataków, atak w zwarciu i dystansowy. W zwarciu po upłynięciu określonego czasu od rozpoczęcia ataku sprawdzana jest odległość od gracza. Jeżeli jest ona większa od jego zasięgu wówczas atak spudłuje. Jest to niezbędny element pozwalający na dopasowanie momentu zadania obrażeń do animacji poprzez regulowanie opóźnienia.

```
IEnumerator AttackMelee()
{
    isAttacking = true;
    animator.SetFloat("Attack_Melee", attackSpeed);
    if (Random.Range(0, 100) < 40)
        SoundController.Instance.PlayRandomSound(
          sounds, transform.position, 0.35f);

    yield return new WaitForSeconds((1 / attackSpeed) * attackDamageDelay);

    if (towardsPlayer.magnitude <= meleeAttackRange && isAlive)
    {
        SoundController.Instance.PlayRandomSound(
            SoundController.Instance.meleeAttack, transform.position, 0.35f);
        playerCombat.DealMeleeDamage(meleeAttackDamage);
    }

    yield return new WaitForSeconds(
      (1 / attackSpeed) - (1 / attackSpeed) * attackDamageDelay);
    isAttacking = false;
    animator.SetFloat("Attack_Melee", 0);
}
```

Z kolei zasięgowy atak, zamiast sprawdzania odległości tworzy pocisk zmierzający w kierunku gracza. Obrażenia zostaną zadane dopiero w momencie kolizji.

```
IEnumerator AttackRanged()
{
    isAttacking = true;
    animator.SetFloat("Attack_Ranged", attackSpeed);
    if(Random.Range(0,100) < 40)
        SoundController.Instance.PlayRandomSound(
            sounds, transform.position, 0.35f);

    yield return new WaitForSeconds(1 / attackSpeed * attackDamageDelay);

    if (isAlive)
    {
        SoundController.Instance.PlaySound(
            rangedAttack, transform.position, 0.35f);
        GameObject bullet = Instantiate(
           bulletPrefab, bulletOrigin.position,
           Quaternion.LookRotation(towardsPlayer, Vector3.up));
        bullet.GetComponent<BulletController>().SetBullet(
           towardsPlayer.normalized * bulletSpeed,
           rangedAttackDamage, bulletLifetime);
    }

    yield return new WaitForSeconds(
       (1 / attackSpeed) - (1 / attackSpeed) * attackDamageDelay);
    isAttacking = false;
    animator.SetFloat("Attack_Ranged", 0);
}
```

Zmiana pomieszczeń

Gracz widzi jedynie pomieszczenie, w którym obecnie się znajduje. Wszystkie pozostałe są stale ukrywane. Efekt ten osiągnięto odpowiednio włączając i wyłączając komponenty MeshRenderer wszystkich widocznych elementów znajdujących się w danym pomieszczeniu.

```
void SetMeshRenderersState(MeshRenderer[] renderers, bool state)
{
    foreach (MeshRenderer r in renderers)
        r.enabled = state;
}
```

Pomieszczenia posiadają swoje triggery sprawdzające obecność gracza. Po aktywowaniu, jeżeli pomieszczenie nie zostało jeszcze oczyszczone zamykane są wszystkie drzwi do momentu spełnienia wymagań, które zawsze jednak kończą się pokonaniem wszystkich przeciwników.

```
public void EnterRoom()
{
    boxCollider.enabled = false;
    SetMeshRenderersState(meshRenderers, true);

    if (!roomClear)
    {
        foreach (GameObject d in doors)
            d.SetActive(true);

        roomEnterTime = Time.time;
        isGoalActive = true;
        if (roomType == RoomGenerator.RoomType.Default
           || roomType == RoomGenerator.RoomType.Boss)
            UIController.Instance.SetGoal(roomGoal);
        SoundController.Instance.PlaySound(
           SoundController.Instance.doors, transform.position, 1.5f);
    }
}
```

Ruch kamery

Kamera przytwierdzona jest do postaci gracza i przesunięta o wartość offset. Aby ruch gracza nie wydawał się sztywny zaimplementowano miękkie śledzenie.

```
void FixedUpdate()
{
    if (player != null)
    {
        nextPostion = player.position + offset;
        transform.position = Vector3.Lerp(transform.position, nextPostion,
Time.deltaTime * moveSpeed);
    }
}
```

Ustawianie pozycji kamery odbywa się na podstawie pozycji gracza. Dokonywanie tego w FixedUpdate jest koniecznością, by uniknąć jej drgań. FixedUpdate jest momentem aktualizacji fizyki gry, gdzie poruszana jest również postać gracza. Funkcja Lerp pozwala na stopniową zmianę pozycji wektora z prędkością podaną w argumencie. Miękkość śledzenia oznacza tyle, że kamera nie znajduje się zawsze bezpośrednio w ustalonej pozycji nad graczem a jedynie dąży do tego punktu z pewnym opóźnieniem.

Zapis stanu gry

Plik zapisu tworzony jest po pierwszej rozgrywce. Ma on format JSON i zawiera jedynie kluczowe informacje pozwalające na przywrócenie stanu gry. Tymi informacjami są: ilość posiadanych monet oraz status odblokowania przedmiotów.

```
{"coins":33726,"weaponOwnership":[0,5,5,2,2,0,5],"armorOwnership":6}
```

Zapis i odczyt polegają na skorzystaniu z odpowiednich funkcji z klasy JsonUtility oraz stworzeniu pomocniczej klasy GameData.

```
public class GameData
{
    public int coins;
    public int[] weaponOwnership;
    public int armorOwnership;
}
```

Przy uruchomieniu poziomu sprawdzane jest, czy plik zapisu istnieje. Jeżeli tak to wartości zostają pobrane z pliku, a jeżeli nie, tworzone są dane początkowe.

```
if (File.Exists(saveFileName))
{
    string saveFile = File.ReadAllText(saveFileName);
    gameData = JsonUtility.FromJson<GameData>(saveFile);
}
else
{
    gameData = new GameData
    {
        coins = 0,
        weaponOwnership = new int[7],
        armorOwnership = 0
    };
}
```

Rozwiązania graficzne

Oświetlenie

Temat gry obejmuje eksplorację starych, opuszczonych podziemi, dlatego zdecydowano się na ograniczenie oświetlenia do minimum. Pełni ono również istotną funkcję w rozgrywce poprzez ograniczanie pola widzenia gracza do najbliższego otoczenia oraz eliminuje możliwość widzenia przez ściany, co byłoby efektem zastosowanego tu widoku z góry.

Rysunek 5. Oświetlenie

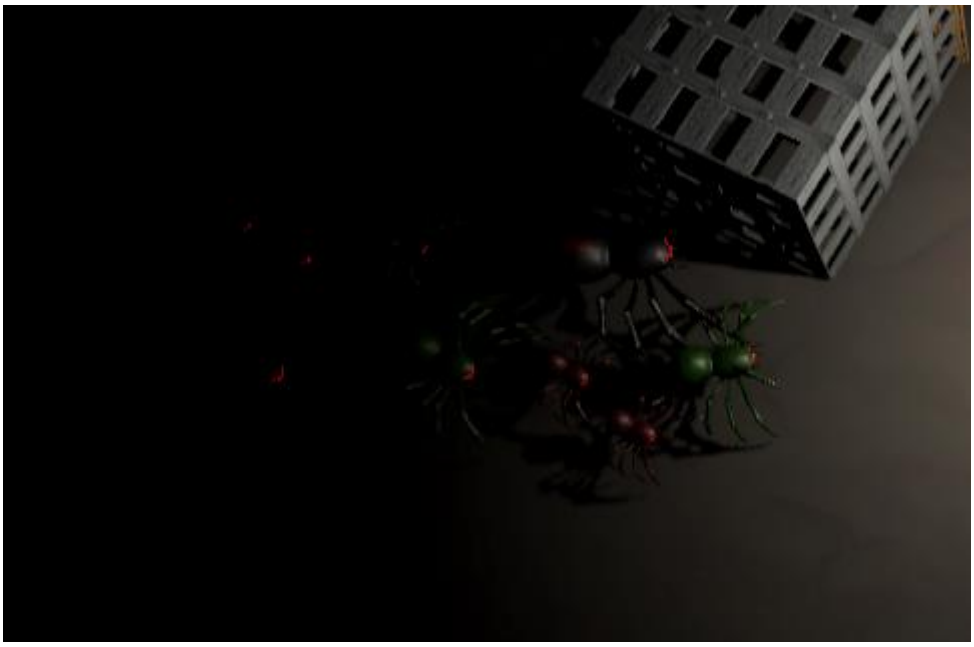
Do osiągnięcia zamierzonego efektu należało całkowicie wyłączyć w projekcie ambient light oraz umieścić źródło światła typu point nad postacią gracza. Warto zwrócić uwagę, iż sama postać gracza nie rzuca cienia, gdyż przy takim umiejscowieniu światła byłoby to nieatrakcyjne wizualnie i rozpraszające podczas rozgrywki. Dodano zatem dwa identyczne źródła różniące się jedynie maskami. Jedno oświetla tylko gracza a drugie wszystko poza nim.

Pozostałe źródła światła mają na celu skupienie uwagi gracza, jak np. podświetlenie sklepu lub w przypadku pochodni, pokazanie informacji o postępie rozgrywki.

Rysunek 6. Sklep

Rysunek 7. Zapalona pochodnia

Innym drobnym elementem związanym z oświetleniem są oczy pająków. Są one widoczne nawet w pełnej ciemności poprzez zastosowanie materiału emitującego czerwone światło.



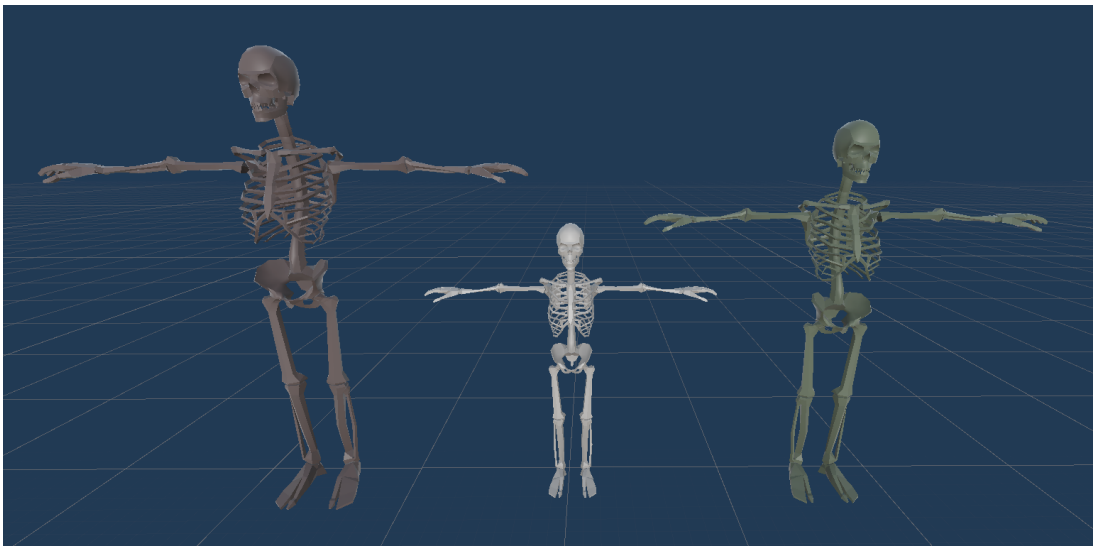
Rysunek 8. Świecące oczy pająków

Modele

Wszelkie modele oraz tekstury należało dobrać tematycznie. Przeszukano strony z assetami głównie pod kątem fraz „dungeon”, „medieval” oraz „monster”. Mimo, że ilość darmowych wersji była mocno ograniczona udało się znaleźć pasujące tematycznie modele przeciwników oraz elementy otoczenia. Skorzystano z serwisów:

- https://opengameart.org/
- https://itch.io/game-assets
- https://assetstore.unity.com/

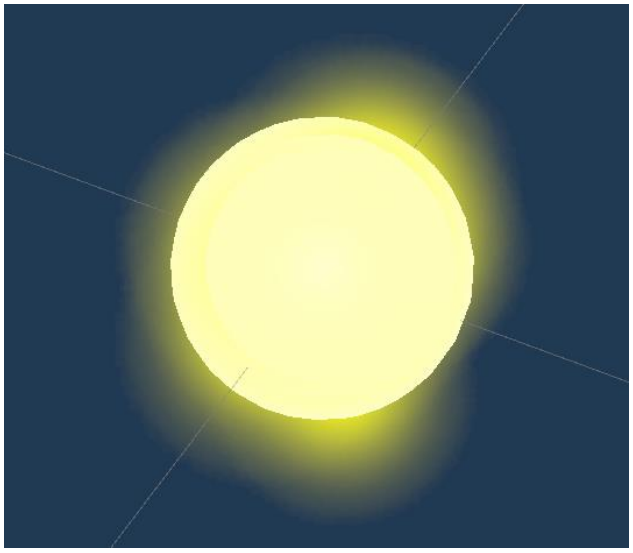
Aby zwiększyć różnorodność dodano warianty przeciwników korzystające z tego samego modelu. Zmieniona za to została ich skala, kolor, statystyki gotowej postaci oraz typ ataku. Elementy otoczenia zostały odpowiednio przeskalowane oraz nałożono na nie collidery, tak aby były gotowe do użycia na scenie.



Rysunek 9. Warianty modelu przeciwnika

Efekty

Wygląd pocisków zależy od jednostki, która je wystrzeliła. Różnią się kształtem, kolorem i wielkością oraz emitują światło, aby łatwo było je zlokalizować i rozpoznać zagrożenie. Aby wzbogacić wizualnie ich wygląd zastosowano system cząsteczek dodając im komponent ParticleSystem. Z ich powierzchni emitowana jest duża ilość cząsteczek o krótkiej długości życia, co imituje efekt magii.



Rysunek 10. Efekt cząsteczkowy pocisku

Rozdział III. Dokumentacja użytkownika

Instalacja i pierwsze uruchomienie

Pobrany projekt w formacie .zip należy wypakować w dowolne miejsce, nie wymaga on dodatkowej instalacji. Zawiera on skompilowaną wersję gry. Do uruchomienia służy plik „Treasure Hunter.exe”. Dane postępu zapisywane są w pliku „SaveFile”, który pojawi się po pierwszej rozgrywce. Aby zresetować postępy wystarczy go usunąć.

Interfejs i sterowanie

Klawiszologia:

- „w”, „s”, „a”, „d” – poruszanie gracza, odpowiednio w: górę, dół, lewo, prawo
- Strzałki – atakowanie w wybranym kierunku
- Spacja – unik
- „e” – przycisk interakcji (skrzynia, pochodnia) oraz wyekwipowania odblokowanej broni z poziomu sklepu.
- „b” – zakup

Dodatkowe opcje ułatwiające testowanie

- „k” – natychmiastowe pokonanie wszystkich obecnych przeciwników
- „m” – dodanie monet
- „r” – restart gry

Rysunek 11. Interfejs użytkownika

Interfejs użytkownika zawiera jedynie najistotniejsze informacje z punktu widzenia gracza. W lewym górnym rogu widnieje ilość posiadanego obecnie zdrowia (ikona serca) oraz monet (ikona monety), natomiast w prawym górnym rogu znaleźć można ilość znajdujących się w pomieszczeniu przeciwników i status obecnego celu. Dla poszczególnych scenariuszy jest to informacja o:

- Fale (ikona mieczy) – numer obecnej fali i ilość wszystkich fal
- Przetrwanie (ikona klepsydry) – czas pozostały do końca
- Pochodnie (ikona latarni) – zarówno ilość zapalonych jak i wszystkich pochodni
- Boss (ikona płonącej czaszki) – procent zdrowia finalnego przeciwnika

O pomyślnym przejściu danego pomieszczenia informuje ikona uciekiniera z workiem łupów znajdująca się w miejscu statusu celu. W trakcie gry napotkać można obiekty, z którymi można wejść w interakcje, takie jak skrzynie oraz pochodnie. Pojawia się wówczas odpowiednia informacja w dolnej części ekranu.

Interfejs sklepu zawiera listę broni oraz pancerz. Liczba z lewej oznacza obecny poziom danego wyposażenia, następnie znajduje się jego nazwa oraz cena wymagana, aby kupić lub ulepszyć do wyższego poziomu. Poziom 0 oznacza, że dany element nie został jeszcze kupiony i nie można go użyć. Biała ramka podświetlająca

napis informuje który z nich jest obecnie wybrany i poprzez wciśnięcie odpowiedniego klawisza można go kupić, ulepszyć bądź wyekwipować.

Rysunek 12. Menu sklepu

Rozgrywka

Gracz rozpoczyna swoją wędrówkę w pomieszczeniu startowym. Nie zawiera ono przeciwników, pozwala za to odpowiednio się przygotować. Przed wyruszeniem do walki warto odwiedzić sklep poprzez podejście do podświetlonego stoiska z ekwipunkiem i dokonanie zakupów. Kupiona broń zostaje automatycznie wyekwipowana.

Każde pomieszczenie posiada czworo drzwi, otwierane są natomiast jedynie te wzdłuż wygenerowanej ścieżki. Aby się to zadziało należy najpierw spełnić cel w obecnie odwiedzanym pomieszczeniu. W pomieszczeniach typu default wybierany jest jeden z trzech celów: fale, przetrwanie lub pochodnie.

Rysunek 13. Pomieszczenie startowe

W trakcie wędrówki można napotkać elementy otoczenia pozwalające na destrukcję, są nimi wazy których zniszczenie nagradza niewielką ilością monet.

Rysunek 14. Wazy

Po dotarciu do finalnego pomieszczenia i pokonaniu przeciwników na środku pomieszczenia pojawia się skrzynia. Po jej otworzeniu gracz otrzymuje znaczną ilość monet a rozgrywka zaczyna się na nowo.

Rysunek 15. Finałowa skrzynia

Walka

Walka opiera się na modelu zręcznościowym. Należy przede wszystkim starać się unikać ataków przeciwnika, gdyż po sprowadzeniu punktów zdrowia gracza do zera gra rozpoczyna się na nowo a główna nagroda nie zostaje zdobyta. W kwestii przeżywalności dodatkowo pomaga pancerz, który jest dostępny w sklepie. Zmniejsza on procentowo ilość obrażeń otrzymywanych w pojedynczym ataku.

W sklepie można również zaopatrzyć się w broń. Można podzielić ją na trzy główne typy zależne od jej charakterystyki:

- Jednoręczna – dość wyważona, umożliwia częsty dostęp do uników a spowolnienie przy atakach jest niskie, niestety kosztem niskich obrażeń oraz krótkiego zasięgu.
- Dwuręczna – jest zdecydowanie wolniejsza a dzierżący ją gracz jest dużo mniej mobilny. Rekompensuje to silnymi atakami o większym zasięgu.
- Dystansowa – pozwala na trzymanie się z bezpieczniejszej odległości, zazwyczaj jednak jest słabsza lub bije tylko pojedynczy cel. Wymaga również dodatkowej celności.

Rysunek 16. Broń biała

Rysunek 17. Broń dystansowa

W pomieszczeniach możemy natrafić na różnych przeciwników. Niektórych zdecydowanie łatwiej jest eliminować konkretnym typem broni. Różnorodność ta pozwala również na wybranie najlepszego dla siebie stylu gry. Warto również brać pod uwagę układ otoczenia. Niektóre pokoje posiadają wiele miejsc na schowanie się przed ostrzałem i zmuszenie przeciwników do zbliżenia się do walki w zwarciu, inne za to posiadają dużo bardziej otwarte przestrzenie, co sprzyja walce dystansowej.

Przeciwnicy

Przeciwnicy dzielą się na typy, w obrębie których można napotkać różne warianty. Każdy typ posiada swoją charakterystykę zgodnie z którą został zaprojektowany:

- Szkielety – posiadają wyważone statystyki. Należy uważać na małe jednostki, które są bardzo szybkie. Pociski magów mają wysoką prędkość oraz zasięg.
- Pająki – silne w grupach, pojedynczo łatwe do pokonania. Wielki pająk jest zdecydowanie trudniejszym przeciwnikiem, lecz pojawia się pojedynczo.
- Gobliny – jednostki zarówno zasięgowe jak i walczące w zwarciu. Sposób ataku wybierają w zależności od odległości gracza.
- Golemy – wolne z silnymi atakami i większą ilością zdrowia.
- Boss – ogromna ilość zdrowia i siła ataku, wymaga dobrych uników. Jedyna jednostka, której nie można przepchnąć co wpływa na niebezpieczeństwo bycia zablokowanym w kącie.

Warianty w obrębie jednego typu również różnią się między sobą nie tylko wyglądem. Różnice obejmują m.in.: liczebność grup w jakich się pojawiają, statystyki oraz rodzaj ataku.

Zakończenie

Stworzenie gry kompletnej pod kątem rozgrywki wymagało odpowiedniego rozplanowania pracy. Obserwacja innych gier z tego gatunku zdecydowanie pomogła ustawić zadania do wykonania hierarchicznie pod kątem ważności. Dzięki temu wszystkim poświęcono odpowiednią ilość czasu na przemyślenie i implementację. Istotną rolę pełniło również testowanie wszystkiego na bieżąco, co sprawiło, że w porę dokonywane były kluczowe zmiany rozgrywki.

Niestety samodzielne tworzenie tak rozbudowanego projektu jakim jest gra komputerowa wymaga czasem nałożenia pewnych ograniczeń. Niektóre systemy zostały przez to zaimplementowane jedynie w stopniu podstawowym. W tym przypadku, gdzie nacisk postawiony był na opracowanie i implementację rozgrywki, elementy graficzne w postaci modeli, animacji, tekstur, ikon zostały zepchnięte na dalszy plan. Wykorzystano więc dostępne za darmo assety, które wpasowywały się tematyką do tworzonego projektu.

Środowisko Unity posiada bardzo dużą ilość dostępnych publicznie poradników, co sprawiło, że pod kątem programistycznym nie napotkano większych problemów. Również język C# jest językiem wysokiego poziomu, co pozwoliło w sprawny sposób przelewać pomysły na kod. Duża baza assetów w Unity Asset Store oraz jej bardzo dobra integracja z silnikiem Unity sprawiła, że wdrażanie gotowych elementów ograniczało się jedynie do kilku kliknięć.

Gra „Treasure Hunter” od początku tworzona była z zamysłem dalszej rozbudowy i zwiększania zawartości. Kod odpowiednio dzielony był na skrypty, co uprasza wprowadzanie nowych systemów do gry. Dostępna zawartość może być bez trudu rozszerzana bez ingerencji w kod.

Bibliografia

[1] Gibson J., *Introduction to Game Design, Prototyping, and Development*, wrzesień 2022

[2] Baron D., *Game Development Patterns with Unity 2021*, lipiec 2021,

[3] Brackeys, *Everything to know about the Particle System*, luty 2018, https://youtu.be/FEA1wTMJAR0, dostęp: 15.06.2023

[4] Brackeys, *Smooth Camera Follow in Unity*, czerwiec 2017 https://youtu.be/MFQhpwc6cKE, dostęp: 15.06.2023

[5] SilverlyBee, *Procedural Dungeon Generator in Unity*, wrzesień 2021, https://youtu.be/gHU5RQWbmWE, dostęp: 15.06.2023

[6] Blackthornprod, *Random Dungeon Generator*, marzec 2018, https://youtu.be/qAf9axsyijY, dostęp: 15.06.2023

[7] iHeartGameDev, *How to Animate Characters in Unity 3D – Animator Explained*, maj 2020, https://youtu.be/vApG8aYD5aI, dostęp: 15.06.2023

[8] *Dokumentacja języka C#*, Microsoft, https://learn.microsoft.com/pl-pl/dotnet/csharp/, dostęp: 15.06.2023

[9] *Documentation*, Unity, https://docs.unity.com/, dostęp: 15.06.2023