

Fragment dokumentacji dotyczący modułu sztucznej inteligencji

7. Projekt produktu programowego

W projekcie dominuje component-based architecture, którego użycie wynika bezpośrednio ze stosowanego przez nas silnika gier Unity.

W kodzie zastosowano wiele wzorców projektowych, przede wszystkim w celu komunikacji między modułami.

- Wzorzec obserwator - użyty został wielokrotnie do komunikacji między elementami systemu. Przykładem użycia jest EnemySpawner i EnemySpecific, gdzie spawner obserwuje każdego przeciwnika, którego pojawił. W momencie, gdy przeciwnik umiera, wywołuje on odpowiednią metodę, która informuje spawner o swojej śmierci. Gdy każdy spawner w pokoju będzie wiedział, że wszyscy podlegli mu przeciwnicy nie żyją, gracz będzie miał możliwość otworzenia drzwi i przejścia dalej
- Wzorzec fasada - użyty został do ujednolicenia komunikacji z przeciwnikami, klasa EnemySpecific daje dostęp do korzystania z metod komponentów przeciwnika, bez konieczności samodzielnego wywoływania funkcji głęboko zaszytych. Pozwala na bezpieczne korzystanie z metod klas EnemyMagicSystem, Animator, EnemySM. HealthAndMana wywołuje metodę Die() z zawartą w klasie EnemySpecific bez konieczności wiedzy, jakim konkretnie przeciwnikiem jest dany obiekt.
- Maszyna stanów - do zarządzania zachowaniami przeciwników posłużyliśmy się maszyną stanów, której użycie zapewnia znacznie większą kontrolę i łatwość w znajdowaniu błędów. Każdy typ przeciwnika ma nie więcej niż 9 stanów, zatem praktycznie żadne problemy, związane z użyciem tego rozwiązania, nie występują. Potencjalnym problemem jest powtórzenie kodu, w pokrywających się stanach, takich jak Rest, Patrol, Chase.

Obsługa przeciwników została rozwiązana w sposób zapewniający możliwie najmniejsze powtarzanie kodu i spójność między różnymi typami przeciwników:

- Każdy przeciwnik komponent EnemySM i odpowiadający za konkretny rodzaj przeciwnika (np. MeleeSM, ShamanSM)
- EnemySM służy do przechowywania elementów niepowiązanych ściśle z typem przeciwnika, takimi jak prędkość poruszania, referencja do obiektu gry gracza, navMeshAgent
- EnemySM służy przede wszystkim do kontroli ruchu przeciwnika, udostępnia metody zmiany stanu (np. podążanie za celem (graczem), losowe przemieszczanie się po mapie, postój, obracanie się w kierunku celu)

- EnemySM udostępnia metody zwracające informacje konieczne do podejmowania decyzji przez konkretne maszyny stanów (np. odległość od celu, odległość od celu z wyłączeniem wysokości, czy przeciwnik znajduje się w podanej odległości od celu, czy droga w podanym wektorze jest wolna)
- EnemySM udostępnia jednolity mechanizm znajdowania ścieżki do patrolu, który preferuje ścieżki nieprowadzące do kolizji z innymi przeciwnikami
- Konkretne maszyny stanów korzystają z informacji udostępnianych przez EnemySM i własnych parametrów, aby odpowiednio zmieniać stany, np. dla MeleeSM:

```
3 references
public bool IsGoalInRange()
{
    if(enemySM.GetGoalDynamic() == null)
    {
        return false;
    }
    return enemySM.IsInRangeXZ(detectionRange);
}
```

```
if (mainStateMachine.IsGoalInRange())
{
    mainStateMachine.ChangeState(mainStateMachine.chase);
    return;
}
```

- Konkretne maszyny stanów najczęściej ustawiają konkretny stan EnemySM przy wejściu do nowego własnego stanu, np. dla stanu Chase:

```
public void OnEnter()
{
    stateMachine.SetStateFollow();
```

- EnemySpecific jest klasą abstrakcyjną, po której dziedziczą wszystkie konkretne maszyny stanu
- EnemySpecific zawiera zestaw zbiera referencje do komponentów używanych przez każdy rodzaj przeciwnika i zestaw metod używanych przez każdy rodzaj przeciwnika (np. Die(), ChangeState(), CastSelectedSpell())
- EnemySpecific udostępnia abstrakcyjną metodę RandomizeStateTimes(), która ma jednorazowo pozwolić konkretnym maszynom stanu zmienić wartości czasu niektórych stanów, aby przeciwnicy nie robili wszystkiego w tym samym czasie
- Instancje stanów są przechowywane jako zmienne w maszynach stanu, aby nie trzeba było tworzyć nowej instancji przy każdym wywołaniu

8.4 Sztuczna inteligencja, przeciwnicy (Łukasz Małecki)

8.4.1 Dynamiczne patrole przeciwników

8.4.1.1 - Zarys problemu

Problem: przeciwnik potrzebuje możliwości znalezienia punktu, w odpowiednim promieniu od niego, do którego może swobodnie się dostać (nie ma po drodze ściany, w danym punkcie znajduje się navmesh), preferowane jest, aby zminimalizować liczbę kolizji pomiędzy różnymi przeciwnikami

Ograniczenia: przeciwnik nie zna geometrii pomieszczenia, w którym się znajduje, nie wie również, gdzie znajdują się inni przeciwnicy

Rozwiązanie: przeciwnik wywołuje funkcję MakePath w X kierunkach wokół siebie, w tych samych kierunkach wywołuje Raycast wykrywający przeciwników, zapisujemy poprawne punkty zwrócone przez MakePath, ale preferujemy te, w których Raycast nie wykrył przeciwników, w przypadku nieznalezienia takiego punktu przeciwnik wróci do pozycji, z której ostatnio przyszedł, jeśli takiej pozycji nie ma, to będzie stał w miejscu.

8.4.1.2 - Opis algorytmu

1. Funkcja bool FindPatrolSpot(out Vector3 destination, float rotation=30f, float pathLength=8f, int sampleStep=8, float minimumPathPercent = 0.5f)
 - a. bool - informacja, czy udało się znaleźć cel patrolu
 - b. destination - zwraca tu cel patrolu, jeśli znaleziono, w przeciwnym wypadku wektor pozycji przeciwnika
 - c. rotation - ilość stopni o które chcemy obracać kierunek w każdej iteracji (definiuje liczbę iteracji)
 - d. pathLength - max odległość destination od obecnej pozycji przeciwnika
 - e. sampleStep - liczba kroków kontrolnych dla metody MakePath
 - f. minimumPathPercent - minimalna długość ścieżki, aby została zaakceptowana jako możliwa
2. Podaną wartość rotacji zmieniamy na wartość bezwzględną i jeżeli == 0, to zwracamy false
3. Tworzymy dwie puste listy wektorów 3D: List<Vector3> potentialSpots, List<Vector3> potentialBetterSpots
4. W pętli for(float currentRotation = 0f; currentRotation <= 360f; currentRotation += rotation)
 - a. zbieramy wektor kierunku, do którego zwrocony jest przeciwnik do tempVector

- b. tempVector obracamy o currentRotation korzystając z funkcji RotateVector z EnemySM
 - c. ustalamy wartość Vector3 tempDestination na Vector3.zero
 - d. if(stateMachine.MakePathDestination(out tempDestination, tempVector, pathLength, sampleStep, minimumPathPercent))
 - i. tempDestination - tutaj zwróci potencjalny cel patrolu
 - ii. tempVector - wektor kierunku po rotacji
 - iii. pathLength - maksymalna długość ścieżki
 - iv. sampleStep - liczba kroków kontrolnych
 - v. minimumPathPercent - minimalny procent długości ścieżki, aby ją zaakceptować
 - vi. then dodaj tempDestination do potentialSpots
 - e. else
 - i. continue;
 - f. if(!stateMachine.Raycast(tempVector, pathLength*minimumPathPercent))
 - i. raycast tutaj sprawdza, czy w podanym kierunku są przeciwnicy
 - ii. then dodaj tempDestination do potentialBetterSpots
5. if(potentialSpots.Count == 0)
- a. then return false;
6. if(potentialBetterSpots.Count != 0)
- a. destination ustalamy losowy element z listy potentialBetterSpots
 - b. return true
7. destination ustalamy losowy element z listy potentialSpots
8. return true;

8.4.1.3 - Opis słowny MakePathDestination

Funkcja zwraca informację, czy da się zrobić ścieżkę w podanym kierunku o danej długości.

Funkcja przyjmuje za argumenty destination, do którego zwróci wynik ostateczny punkt, do którego może dostać się przeciwnik bez przeszkód; direction - kierunek, w którym mamy szukać ścieżki; pathLength - pożądana długość ścieżki; sampleStepCount - liczba punktów, w

których uruchamiamy SamplePosition; minimumPathLength - minimalny procent długości ścieżki, aby uznać ją za stosowną do zwrócenia.

Normalizujemy wektor kierunku i w pętli for (int i = 1; i <= sampleStepCount; i++) przypisujemy destination wartość pozycji przeciwnika z dodanym wektorem kierunku przemnożonym przez i, pathLength oraz podzielonym przez sampleStepCount. W tejże pozycji uruchamiamy SamplePosition od navmesh, które sprawdza, czy w małej odległości od danego punktu znajduje się niewycięty navmesh. Jeżeli zwróci on true, to zapisujemy nasz destination jako dotyczasowy najlepszy, w przeciwnym wypadku jeśli wartość zmiennej i jest większa/równa od sampleStepCount*minimumPathLength, wtedy break, w przeciwnym wypadku ustawiamy destination jako obecną pozycję gracza i zwracamy false.

Po pętli dla destination przypisujemy bestDestination.

Następnie korzystamy z navmesh.CalculatePath, który oblicza kawałek ścieżki (na ogół nie całą), jeśli ścieżka okaże się invalid, to zwracamy false, w przeciwnym wypadku zwracamy true.

8.4.2 Typy przeciwników (Łukasz Małecki)

8.4.2.1 - Wstęp

W grze mamy 4 główne typy przeciwników, którym odpowiadają konkretne maszyny stanów:

- kontaktowi - MeleeSM (szkielety)
- zasięgowi - RangedSM (gobliny)
- szamani - ShamanSM (szamani)
- golemy - GolemSM (kamienny golem, rogaty golem)

Wszystkie powyższe maszyny stanów dziedziczą po abstrakcyjnej klasie EnemySpecific, która zbiera uniwersalne elementy wszystkich 4 maszyn oraz ułatwia komunikację z resztą elementów systemu.

Proces tworzenia przeciwnika na ogół zawierał następujące elementy:

- Miałem/znajdowałem model postaci 3D z użytecznymi animacjami, pasujący do naszej gry
- Biorąc pod uwagę możliwości, jakie dawał asset, tworzyłem plan maszyny stanów zaczynając od ogólnej idei
- Wymyślałem konkretne stany i potrzebne dla nich warunki przechodzenia między sobą
- Ustalałem listę potrzebnych parametrów dla działania maszyny stanów
- Tworzyłem kod maszyny stanów i każdego jej stanów

- Importowałem model do projektu
- Dostosowywałem collider i system animacji do moich potrzeb
- Łączyłem model przeciwnika i jego system animacji z maszyną stanów

Wszelkie prace związane z łączeniem maszyn stanów konkretnie z systemem magii miały miejsce w późniejszym etapie wytwarzania gry.

Później wytworzone maszyny stanów miały bardziej rozbudowany proces planowania.

8.4.2.2 - Przeciwnik kontaktowy

Plan działania przeciwnika kontaktowego opiera się przede wszystkim na opisie dostępnych dla niego stanów i przejść między nimi. Jest to najprostsza z maszyn stanów, zatem wymagała najmniej refleksji.

Fragmenty oryginalnego dokumentu:

- **Rest** – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed > restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, gdy gracz będzie w pobliżu, zmienna **reactionTime** będzie decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po tym, jak gracz wejdzie w jego zasięg widzenia
- **Patrol** – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi w stan **Rest**, przeciwnik zmieni stan na **Chase**, jeżeli gracz będzie w pobliżu
- **Chase** – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Attack**, jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znowu pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Patrol**
- **Attack** – przeciwnik zaczyna wykonywać atak, jeśli gracz jest nadal w zasięgu po wykonaniu ataku, to po cooldownie na atak wykonyuje kolejny, w przeciwnym wypadku przechodzi do stanu **Chase**

Przeciwnik ten nie jest szczególnym wyzwaniem w pojedynkę, jednak zostanie przez niego osiączonym z wielu stron bez możliwości radzenia sobie z grupami przeciwników może być zabójcze.

Atak przeciwnika został zrealizowany jako niewidzialny trigger pojawiający się przed przeciwnikiem w miejscu wykonywania animacji ataku mieczem, który szybko znika po zakończeniu animacji.

Istniała również koncepcja, aby przeciwnik nie wytraçał prędkości w momencie wyprowadzenia ataku, jednak zrezygnowaliśmy z tego, z powodu nienaturalnego wyglądu takiego zdarzenia oraz zbyt dużego poziomu trudności w unikaniu.



State times	
Rest Time	3
Patrol Time	8
Chase Time	4
Hurt Time	0.3
Reaction Time	0.5
Range	
Attack Range	2
Detection Range	19
Attack	
Attack Cooldown	0.2

```

public void UpdateState()
{
    if (!mainStateMachine.IsGoalInRange())
    {
        stateTimePassed += Time.deltaTime;
    }
    else
    {
        stateTimePassed = 0f;
    }
    if(stateTimePassed >= stateTime)
    {
        mainStateMachine.ChangeState(mainStateMachine.patrol);
        return;
    }

    if (mainStateMachine.IsGoalInAttackRange())
    {
        mainStateMachine.ChangeState(mainStateMachine.attack);
        return;
    }
}

```

Prostota MeleeSM, lewo - parametry maszyny stanów, prawo - metoda update stanu Chase

Przeciwnik pojawia się w dwóch wariantach:

- SkeletonWeak - wersja bez tarczy, słabsza, pojawia się tylko na 1-ym piętrze, dla ułatwienia, gdy gracz nie ma przedmiotów
- SkeletonBase - wersja z tarczą, standardowa, pojawia się na każdym piętrze

Oba warianty są wrażliwe na magię ognia i ziemi.

8.4.2.3 - Przeciwnik zasięgowy

Plan działania przeciwnika zasięgowego zawiera 2 dodatkowe stany w porównaniu do kontaktowego. Pomyśl na uciekanie przeciwników przed graczem wziął się z tego, że w kulturze goblinów nierzadko postrzegane są jako tchórzliwe istoty, ponadto taki dodatkowy typ

responsywności na działania gracza zapewnia lepszą immersję. W samym planie są również zawarte plany na temat sposobu implementacji ucieczki oraz sposobu wybierania celu ataku. Po dyskusji planu z resztą zespołu wybrane zostały rozwiązania zaznaczone na zielono. O decyzjach przede wszystkim zadecydowała prostota implementacji, którą PM uznał za kluczową, aby dotrzymać terminu. Ostateczna wersja działa praktycznie tak jak opisane w planie, z zastrzeżeniem do sekcji opisu ucieczki przeciwnika przed graczem, która działa w bardziej wyrafinowany sposób. Różnice te zostaną wymienione na fioletowo.

Fragmenty oryginalnego dokumentu:

- **Rest** – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed** > **restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, gdy gracz będzie w pobliżu, zmienna **reactionTime** będzie decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po tym, jak gracz wejdzie w jego zasięg widzenia
- **Patrol** – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi wchodzi w stan **Rest**, przeciwnik zmieni stan na **Chase**, jeżeli gracz będzie w pobliżu
- **Chase** – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Prepare**, jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znowu pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Patrol**
- **Attack** – przeciwnik zaczyna wykonywać atak, jeśli gracz jest nadal w zasięgu po wykonaniu ataku i **attacksNumberLeft** > 0, to po cooldownie na atak wykonuje kolejny, jeśli gracz nie jest w zasięgu, to przechodzi do stanu **Chase**, jeżeli **attacksNumberLeft** <= 0, to przeciwnik przechodzi w stan **Prepare**, zmienna **attacksNumber** określa maksymalną liczbę ataków w „serii”
- **Prepare** – przeciwnik jest w odpowiednim dystansie od celu i przygotowuje się do wykonania ataku, po odpowiednim czasie przechodzi do stanu **Attack**, jeżeli przed upływaniem tego czasu cel będzie za blisko (w **dangerRange**), to jeżeli zmienna **isAttackForced** nie będzie true, to przeciwnik wchodzi w stan **Flee**, jeżeli cel wyjdzie z zasięgu ataku, to przeciwnik przechodzi w stan **Chase**
- **Flee** – przeciwnik oddala się od celu, po określonym czasie/dotarciu do odpowiedniego punktu ustawia zmienną **forcedAttack** na true, jeżeli cel jest w zasięgu, to również wchodzi w stan **Prepare**, w przeciwnym wypadku wchodzi w stan **Chase**

Kolejność od najbardziej, do najmniej prawdopodobnych w wykonaniu

1. **Flee** odbywać może się na różne sposoby, propozycje:

- a. W pewnym promieniu od przeciwnika przeszukiwana jest część punktów na okręgu (np. co kilka stopni) i wybierany jest punkt, do którego może zostać utworzona ścieżka (preferowanie jak najdalej od celu) (Zalety: elastyczność, indywidualność przeciwników; Wady: potencjalne znalezienie się punktu w innym pokoju? (rozwiązanie, branie pod uwagę

długości potencjalnej ścieżki, ustalenie maksymalnej długości ścieżki, np. $1.5 * \text{promień}$, potencjalna trudność w implementacji, potencjalne uciekanie w kierunku gracza? (rozwiązanie: nieprzeszukiwanie okręgu, ale półokręgu, potencjalny problem w przypadku, gdy przeciwnik otoczony jest ścianami z 3 stron potencjalnie rozwiązyany przez $1.5 * \text{promień}$)

- b. Na mapie będzie znajdować się kilka punktów, do których przeciwnik może uciec i wybierany punkt będzie na podstawie kryteriów: możliwie blisko przeciwnika + możliwie daleko od celu + aby cel był w zasięgu + aby cel nie był za blisko (Zalety: łatwość implementacji; Problemy: ucieczka wielu przeciwników do tego samego miejsca, silne powiązanie przeciwnika z danym pokojem)
- c. Przeciwnik ma uciekać w tym samym kierunku, w którym porusza się gracz, aż nie na trafi na ścianę, w tym momencie ma się obrócić w stronę gracza i próbować atakować, ewentualnie uderza w ścianę i się stunner (Zalety: łatwość implementacji, łatwa przewidywalność dla gracza; Problemy: potencjalne częste napotykanie bycia pod ścianą, A.I. postrzegane jako głupie) - przeciwnik próbuje uciekać w kierunku zgodnym z wektorem pozycja gracza -> pozycja przeciwnika na dystans o maksymalnej długości **fleeDistance**, jeżeli w tym kierunku nie ma możliwości utworzenia ścieżki dłuższej niż **fleeDistance***0.5, to przeciwnik ustawia **isAttackForced** i wchodzi do stanu **Prepare**, jeżeli ścieżkę udało się utworzyć to przeciwnik ucieka w kierunku celu i gdy dojdzie do niego lub minie **maxFleeDistance**, to ustawia **isAttackForced** na true i wchodzi do stanu **Prepare**

2. Cel ataków przeciwnika

- a. Przeciwnik może strzelać dokładnie tam, gdzie gracz jest w momencie strzału (Zalety: bardzo prosta implementacja, łatwa do zrozumienia mechanika; Wady: potencjalnie bardzo niska skuteczność przeciwników zasięgowych)
- b. Przeciwnik może brać pod uwagę wektor prędkości gracza i odpowiednio zmienić miejsce strzału o ustaloną liczbę stopni (Zalety: potencjalnie lepsza skuteczność przeciwnika; Wady: wymagany sposób na określenie wektora prędkości celu)
- c. Przeciwnik może brać pod uwagę wektor prędkości gracza i proporcjonalnie do niego zmienić miejsce strzału, aby trafić go jeżeli wektor prędkości zostanie zachowany (Zalety: zdecydowanie lepsza skuteczność przeciwnika; Wady: wymagany sposób na określenie wektora prędkości celu, potencjalne bycie niefair wobec gracza, gdy gracz ma bardzo wysoką prędkość (potencjalne rozwiązanie: limit prędkości lub zmiany kąta wystrzału), potencjalnie zbyt duże utrudnienie dla gracza)

Stworzenie przeciwnika zasięgowego zaowocowało utworzeniem metody **MakePathDestination** (opisanej w **8.4.1.3**), która była konieczna do zapewnienia osiągalności punktu wybranego w stanie **Flee**.

Przeciwnik stanowi rzeczywiste zagrożenie dla gracza (zwłaszcza, że zazwyczaj nie występuje w pojedynkę) i zmusza go do bycia w ciągłym ruchu.

Strzały goblina są w rzeczywistości zaklęciem wykonanym w ramach modułu Systemu magii. Na ogół gobliny mają szansę na wystrzelanie jednej z dwóch strzał: jedna zwykła, druga mająca efekt statusu (np. ślepota, podpalenie). Stosunek prawdopodobieństwa tych strzał wynosi na ogół 6:1, choć może się różnić między wariantami.



State times	
Rest Time	3
Patrol Time	5
Chase Time	4
Prepare Time	2.5
Hurt Time	0.3
Reaction Time	0.5
Max Flee Time	5
Range	
Attack Range	14
Detection Range	17
Danger Range	5
Attack	
Attack Cooldown	1
Attack Series Number	2
Flee	
Flee Distance	3
Flee Steps	8
Cast Point	CastPoint (Transform)

```
public void UpdateState()
{
    if (mainStateMachine.IsGoalInRange())
    {
        stateTimePassed += Time.deltaTime;
    }
    else
    {
        stateTimePassed -= Time.deltaTime;

        if (stateTimePassed < 0)
        {
            mainStateMachine.ChangeState(mainStateMachine.chase);
            return;
        }
    }
    if(mainStateMachine.IsGoalInDangerRange() && !isAttackForced)
    {
        mainStateMachine.ChangeState(mainStateMachine.flee);
        return;
    }

    if (stateTimePassed >= stateTime)
    {
        mainStateMachine.ChangeState(mainStateMachine.attack);
        return;
    }
}
```

Lewo - odpowiednia możliwość parametryzacji przeciwnika, Prawo - stan prepare ma możliwość przejścia do jednego z aż 3 stanów (return konieczne po każdym ChangeState, aby zapobiec błędowi w przypadku, gdy warunki dla więcej niż 1 stanu są spełnione)

Mamy 4 warianty goblińskiego łucznika, 2 wykorzystują zmodyfikowaną w programie GIMP pod kątem koloru wersję assetu Goblin Necro:

- GoblinBase (zielony) - bazowy goblin, odporny na magię powietrza, wrażliwy na magię ziemi i elektryczność, drugi typ ataku nakłada truciznę
- GoblinFire (czerwony/pomarańczowy) - ognisty goblin, odporny na magię ognia i ciemność, wrażliwy na magię wody, ziemi i elektryczność, drugi typ ataku nakłada podpalenie
- GoblinDark (fioletowy) - goblin ciemności, odporny na magię powietrza i ciemność, nieduża wrażliwość na ziemię i elektryczność, niższe obrażenia, wyższa obrona, drugi typ ataku nakłada oślepienie
- GoblinElite (żółty) - elitarny goblin, odporny na magię powietrza, światła, elektryczności, wrażliwość na magię wody, ziemi, ciemności, najlepsze statystyki ze wszystkich goblinów, nie ma specjalnego ataku, ale w serii jest w stanie wystrzelić 3 strzały

Wersja fioletowa i żółta zostały osiągnięte przy użyciu podmiany kolorów tekstur w programie do obróbki graficznej GIMP.

8.4.2.4 - Szaman

Plan tworzenia szamana powstał jeszcze zanim został skrytalizowany system działania zaklęć dla przeciwników oraz nie było pewności, co do możliwości istnienia zaklęć obniżających/wzmacniających czasowo statystyki entities. Ostatecznie BattleCry wykorzystywany jest do rzucania rzadziej, ale potężniejszych zaklęć, a FocusCast do bardziej typowych, ale nadal bardziej zjawiskowych. QuickCast wystrzeliwuje prosty pocisk zadający obrażenia odpowiedniego typu. Za knockbackSpell służą zaklęcia blisko zasięgowe lub defensywne. Zrealizowana została idea zwiększenia odporności szamana w trakcie trwania stanu **Focus**. Ale znaczna większość informacji jest aktualna.

Fragmenty oryginalnego dokumentu:

Ogólny pomysł

Szaman ma specjalizować się w zasięgowych atakach. Różni się tym, że jest w stanie on korzystać z zaklęć. Zaklęcie rzucane podczas grania animacji BattleCry ma być albo buffem dla przeciwników lub debuffem dla gracza Curse(?). Poza tym zaklęciem będzie posiadał QuickCast, szybki w użyciu wystrzał pocisku (animacja z laską do przodu). Ponadto będzie posiadał FocusSpell, rzucenie tego czaru ma zajmować więcej czasu, ale powinien być potężniejszy, np. duży pocisk podążający za graczem, obszarowe zaklęcie, coś na kształt lasera, etc.

Potencjalny pomysł, może dałoby się zrobić tak, aby szamanowi dałoby się equipnąć z poziomu edytora dowolne zaklęcie? Ewentualnie mógłby mieć metodę sprawdzającą jego kompatybilność z danym zaklęciem, np. czy ma odpowiednie metody, które mogłyby mu pozwolić je rzucić etc.

Stany

Rest – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed > restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, gdy gracz będzie w pobliżu, zmienna **reactionTime** będzie

decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po tym, jak gracz wejdzie w jego zasięg widzenia

Patrol – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi wchodzi w stan **Rest**, przeciwnik zmieni stan na **Chase**, jeżeli gracz będzie w pobliżu

Chase – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Focus**, jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znów pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Patrol**

Attack– przeciwnik ma do wyboru kilka zaklęć o różnym efekcie po użyciu:

- Szybki atak – wyrzuca prosty pocisk w kierunku gracza, po użyciu szaman odczeka połowę **focusTime** i może użyć kolejnego losowego zaklęcia
- FocusSpell – używa silniejszego zaklęcia, po użyciu szaman wchodzi w stan **Focus**
- BattleCry – używa buffa lub debuffa, po użyciu wchodzi w stan **Focus**
- KnockbackSpell(?) – używa zaklęcia odrzucającego gracza, po użyciu wchodzi w stan **Focus**

przeciwnik po wejściu w stan **Attack** wybiera jedno z zaklęć (poza Knockback) i go używa, jeżeli jednak zmienna **isInDanger** == true, to używa Szybkiego ataku lub KnockbackSpell(?), jeśli po Szybkim ataku gracz nie jest w zasięgu, to przechodzi do stanu **Chase**, w przeciwnym wypadku odczeka 50% **focusTime** i wybiera kolejne losowe zaklęcie, w przypadku innych zaklęć przechodzi do stanu **Focus**

Focus (animacja Block)– przeciwnik jest w odpowiednim dystansie od celu i przygotowuje się do wykonania ataku, po odpowiednim czasie przechodzi do stanu **Attack**, jeżeli przed upłynięciem tego czasu cel będzie za blisko (w **dangerRange**), to zmienna **isInDanger** zmienia się na true, jeżeli zmienna **isInDanger** == true, to przeciwnik potrzebuje tylko 50% **focusTime**, aby wejść w stan **Attack**, ale będzie mieć dostęp tylko do szybkiego ataku (lub odrzucenia gracza), jeżeli cel wyjdzie z zasięgu ataku, to przeciwnik przechodzi w stan **Chase**

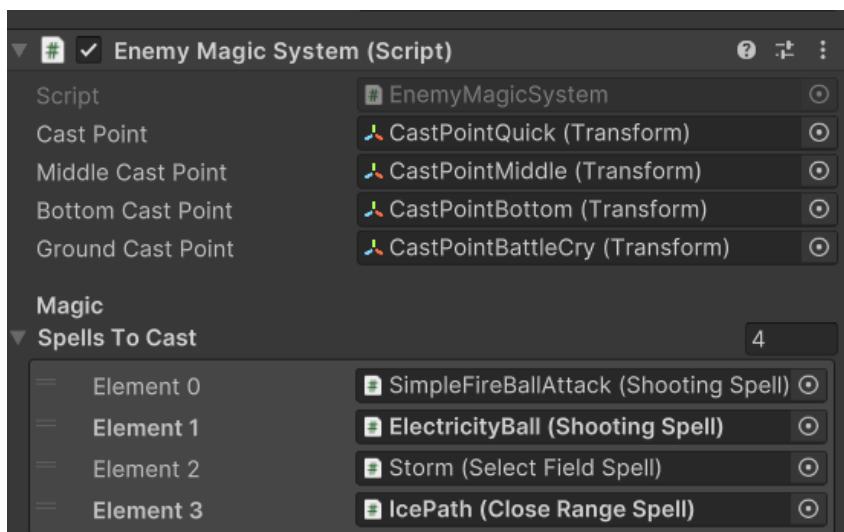
Szaman był pierwszym przeciwnikiem, który z założenia miał korzystać z systemu magii, w czasie tworzenia nie było pewności, jak ten system będzie wykorzystywany przez przeciwników, zatem przez pewien czas przeciwnik aktywował odpowiednie animacje zgodnie z planem, ale w rzeczywistości nie rzucał żadnych zaklęć.

Szaman jest przeciwnikiem, który może stanowić poważne zagrożenie dla nieuwaznego gracza, ich podstawowe ataki są dosyć słabe, ale reszta ich arsenalu potrafi poważnie uszkodzić gracza, ponadto jest w stanie nałożyć nieprzyjemne efekty statusu.

Szamani z idei mieli różnić się od goblinów byciem niewzruszonym, ich reakcją na zagrożenie miała być walka. Gdy gracz podejdzie za blisko albo zostanie przebitý silnym zaklęciem blisko zasięgowym lub szaman rozsądnie osłoni się przed natarciem.



▼ Spell Chance Weights	
= Element 0	4
= Element 1	2
= Element 2	1
+ -	
State times	
Rest Time	3
Patrol Time	7
Chase Time	4
Focus Time	3
Hurt Time	0.3
Reaction Time	0.5
Range	
Attack Range	14
Detection Range	17
Danger Range	4.5
Attack	
Attack Cooldown	0.2
Quick Cast Point	CastPointQuick (Transform)
Focus Cast Point	CastPointFocus (Transform)
Battle Cry Point	CastPointBattleCry (Transform)
Knockback Cast Point	CastPointKnockback (Transform)



SpellChanceWeights pochodzi z EnemySpecific i służy do nadawania wag prawdopodobieństwu wystąpienia zaklęcia o odpowiadającym indeksie, KnockbackSpell nie ma swojej wagi, bo szansa na niego ma być tylko, gdy gracz jest blisko. Szaman ma dodane wiele punktów wylotu zaklęć typu Shooting Spell, aby w zależności od animacji zmienić CastPoint w EnemyMagicSystem (middle, bottom i ground CastPoints zostają zawsze te same)

Mamy 2 warianty szamana:

- ShamanBase - bazowy szaman posługujący się magią ognia i elektryczności, odporny na ogień, ciemność i elektryczność, wrażliwy na wodę, ziemię i jasność
- ShamanWater - szaman wodny posługujący się magią wody i lodu, odporny na wodę, ziemię, ciemność, wrażliwy na ogień, powietrze, elektryczność, jasność, generalnie bardziej wytrzymały

8.4.2.5 - Golem

Dla golema znalezione zostały świetne 2 assety i zamiast tworzyć 2 różne maszyny stanów, postanowiono, że jedna maszyna stanów będzie obsługiwać oba typy. Spowodowało to sporo usprawnień, które nie były wspomniane w dokumencie. Powstały osobne zmienne boolowskie, których ustawienie decydowało o działaniu golema w pewnej sytuacji. Ponieważ jeden z golemów ma animację rzucania, a drugi ma świetną animację szarzy, zdefiniowano 2 różne zachowania w zależności od tego, czy golem może szarżować czy rzucać.

Fragmenty oryginalnego dokumentu:

Ogólny pomysł

Golem powinien być wytrzymały, lecz dość powolnym przeciwnikiem. Jednakże, aby nadal mógł stanowić zagrożenie będzie posiadał stan **Rage**. Kiedy w trakcie gonienia celu oddali się on na zbyt daleki dystans od golema wchodzi on w stan **Focus** (ładuje się) i po tym wchodzi w stan **Rage**. Sam stan **Rage** może być różnie rozwiązyany, jednym z pomysłów jest **Charge**, aby golem zaczął poruszać się w stronę celu z coraz to wyższą prędkością (np. na czas bycia w tym stanie zwiększymy speed agenta i zmieniamy odpowiednio przyspieszenie, można pobawić się z angular speed). Przy osiągnięciu prędkości granicznej lub

odpowiedniego dystansu od celu można ustawić, że cel golema nie będzie się już aktualizował (aby przesadnie nie skręcił na gracza lub nie zwolnił). Innym pomysłem jest po prostu ogólny buff do prędkości, odporności, może szybkości animacji ataku, etc. Innym konceptem jest również rzucenie jakiegoś silnego zaklęcia/rzucenie kamieniem (mini legion rock golem).

Stany

Rest – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed > restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, jeżeli gracz będzie w pobliżu, zmienna **reactionTime** będzie decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po wykryciu przeciwnika, jeżeli **isRestForced == true**, to niezależnie czy gracz jest blisko, golem po upłynięciu czasu wchodzi w stan **Chase**

Patrol – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi w stan **Rest**, przeciwnik zmieni stan na **Chase**, jeżeli gracz będzie w pobliżu

Chase – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Attack**, jeżeli gracz będzie przez pewien czas za daleko od Golema, to wchodzi w stan **Focus** (chyba, że jest już pod wpływem buffa ze stanu **Focus**), jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znowu pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Rest** lub **Patrol**

Attack – przeciwnik wykonuje atak wręcz, jeśli po wykonaniu ataku gracz nadal będzie w zasięgu to po odczekaniu cooldownu wykona on kolejny atak, jeżeli gracza nie będzie już w zasięgu to przejdzie do stanu **Chase**

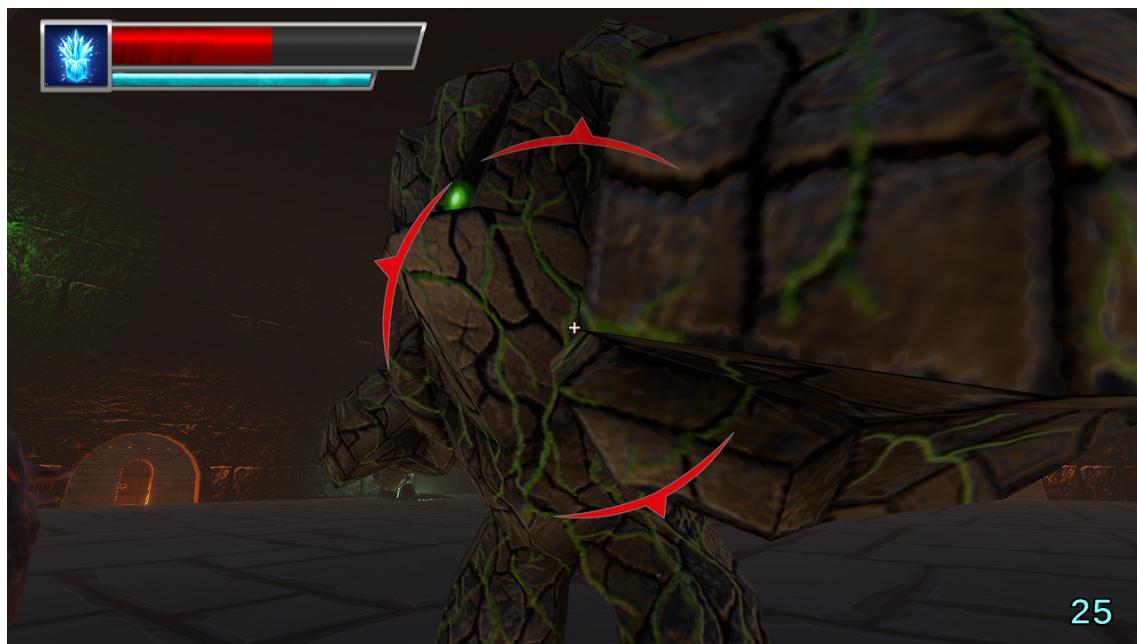
Focus (animacja Victory/Rage) – przeciwnik przygotowuje się do wykonania ataku, po odpowiednim czasie przechodzi do stanu **Rage**, jeżeli przed upłynięciem tego czasu cel będzie w zasięgu zwykłego ataku, to przejdzie do stanu **Attack**, w stanie **Focus** golem powinien mieć zmniejszone otrzymywane obrażenia, czas trwania tego stanu powinien być warunkowany zmienną **focusTime**, jeżeli wartość ta będzie wynosić ≤ 0 , to wykorzystany zostanie oryginalny czas animacji

Rage – golem wykonuje jedną z silnych akcji (przy pomocy booli możemy ustalić, czy dany golem ma do konkretnej akcji dostęp). Po wykonaniu akcji wchodzi w stan **Rest** (forced) lub **Chase**:

- **Charge** – golem zaczyna iść w stronę gracza stopniowo przyspieszając do znacznie wyżej prędkości niż podstawowa i uderza gracza, po tym wchodzi w stan **Rest** (forced)
- **Throw** – golem dokonuje rzutu kamieniem (lub zaklęciem) w gracza po czym wchodzi w stan **Chase**
- **Reinforce** – golem wzmacnia się (nie jest grana dodatkowa animacja, ale może jakiś efekt graficzny się pojawia typu poświata) na czas działania zwiększa jest prędkość, obrażenia etc. od razu wchodzi do stanu **Chase**

Plan był tworzy, gdy nadal nie było pewności, co do zakresu możliwości zaklęć, zatem istniał pomysł na zaklęcie wzmacniające, który ostatecznie został zażegnany. Pomysł na stan rage,

jak opisane w wycinku z oryginalnego dokumentu, poparty był umożliwieniem golemom bycia rzeczywistym zagrożeniem. Szarża nie jest niezawodna i bardzo często nie trafia gracza, ale nadal daje mu poczucie zagrożenia, gdy rogaty golem jest z nim w pokoju.



▼ Spell Chance Weights		4
= Element 0	0	
= Element 1	0	
= Element 2	2	
= Element 3	1	
		+ -
State times		
Rest Time	3	
Patrol Time	7	
Reaction Time	0.5	
Rage Time	3	
Chase Time	5.5	
Range		
Attack Range	4	
Detection Range	17	
Rage Range	13	
Attack		
Attack Cooldown	0.3	
Hurt Time	0.3	
Cast Point Attack	FistAttackPoint (Transform) (◎)	
Rage		
Focus Animation Count	1	
Can Throw	<input checked="" type="checkbox"/>	
Throw Point	Index_Proximal_R (Transform) (◎)	
Can Headbutt	<input type="checkbox"/>	
Rush Speed Percent	3	
Max Rush Time	12	
Focus Ongoing	<input type="checkbox"/>	

Przykładowe wypełnienie parametrów dla kamiennego golema, który jest w stanie rzucić zaklęcia. W SpellWeights pierwsze 2 miejsca są zarezerowane na podstawowy atak i potencjalny wariant tego ataku nakładający efekt. Reszta to zaklęcia, które może wyrzucić golem.

Mamy 2 warianty golema:

- RockGolem - kamienny golem, bardzo wysoka odporność na ziemię, elektryczność, fizyczne, pewna odporność na światło, wrażliwość na ogień, wodę, ciemność, wysoka obrona i siła, korzysta z magii ziemi
- HornedGolem - rogaty golem, bardzo wysoka odporność na wiatr, pewna odporność na wodę i fizyczne, wrażliwość na ogień, elektryczność, lekka wrażliwość na ziemię, wysoka obrona i siła, używa wyłącznie ataków o typie fizycznym