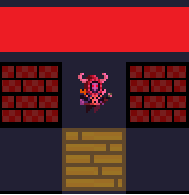
**Algorytmy związane z aktywnością bota**

Idea działania bota: Bot porusza się w sposób ciągły, nie zatrzymuje się kiedy nie jest to konieczne. Stanięcie w miejscu bota jest możliwe tylko w przypadku gdy będzie on otoczony ze wszystkich stron przez bloki zniszczalne/niezniszczalne. Ten przypadek jest bardzo mało prawdopodobny, choć możliwy (czerwone pole oznacza, że mapa w tym miejscu zmniejsza się):



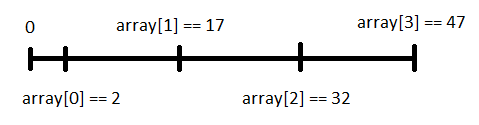
Bot w takim przypadku stanie w miejscu, z każdym wywołaniem funkcji \_physics\_process będzie miał 50% szans na podłożenie bomby (ponieważ znajduje się obok niego blok zniszczalny) i po zniszczeniu tego bloku (przez siebie lub innego gracza) kontynuuje poruszanie się (jeśli nie zginie od bomby).  
  
Stawianie bomby przez bota realizowane jest na 2 sposoby (oczywiście bot musi mieć dostępną jakąś bombę do podłożenia):

* Bot po wykryciu, że podłożona przez niego bomba spowodowałaby zniszczenie przynajmniej 1 bloku ma 50% szans na postawienie bomby na aktualnej pozycji. Sprawdzane jest to przez pętlę wykonywaną 4 razy (w każdym z kierunków: prawo/lewo/góra/dół). Iteracyjnie sprawdzamy czy w odpowiednim kierunku znajduje się jakiś zniszczalny blok. Jeśli tak, zwracamy true (postawienie bomby ma sens). Jeśli nie, sprawdzamy w innym kierunku czy postawienie bomby ma sens. Jeśli w którymś z kierunków trafimy na blok niezniszczalny to zwracamy false (wybuch bomby zatrzymuje sie na blokach niezniszczalnych, nie ma sensu sprawdzać czy za nim jest jakiś blok zniszczalny). Bot musi stać na środku 64x64px pola planszy by postawić bombę (inaczej: może podłożyć bombę po wykonaniu pełnego ruchu w danym kierunku).  
  Za sprawdzenie, czy postawienie bomby ma sens odpowiada funkcja possible\_plant(position)
* Bot ma 50% szans na podłożenie bomby w sposób losowy. Wprowadziliśmy to rozwiązanie, aby bot nie stawiał tylko bomb do zniszczania bloków zniszczalnych. W funkcji \_ready() wywoływana jest funkcja random\_planting() odpowiedzialna za losowe stawianie bomb przez bota. W tejże funkcji ustawiany jest licznik (Timer) na 1.5 sekundy. Po tym czasie bot ma 50% szans na postawienie bomby. Licznik tworzony jest na nowo, cały proces trwa aż do śmierci bota. Bot nie musi stać na środku 64x64px pola planszy by postawić bombę.

Poruszanie się bota realizowane jest w następujący sposób:

W momencie gdy bot dotarł do celu (ruch w prawo/lewo/górę/dół o 64px) wyznaczany jest kolejny kierunek poruszania się bota. Jeśli bot podłożył w tym momencie bombę (w sposób opisany w 1. powyżej) to zmienia kierunek ruchu na przeciwny (symuluje to uciekanie bota od miejsca podłożenia bomby). Jeśli bot nie postawił bomby (w sposób opisany w 1. powyżej) to ustalany jest nowy kierunek poruszania się bota według następującego algorytmu:

W funkcji new\_direction tworzona jest tablica z wartościami, które pomogą nam ustalić, w którym kierunku ma poruszyć się teraz bot.  
Tablica array symuluje dyskretną formę odcinka na osi x o początku w x=0 i końcu w x>=0 , x należy do liczb naturalnych. z liczbami typu int. Array[0] wskazuje na koniec 1 części odcinka od lewej strony. Array[0] jest więc liczbą naturalną >= 0. Analogicznie array[1] jest liczbą naturalną >= array[0].  
Idea przedstawiona na rysunku:



Ten przypadek oznacza, że bot stoi w miejscu z którego może iść w dowolym kierunku, a dotarł do tego miejsca poprzez ruch w lewo od poprzedniej pozycji (przed wykonaniem ruchu).

Idea: jeśli bot poruszył się w prawo, to cofnięcie się w lewo będzie następowało z mniejszym prawdopodobieństwem. Unikamy w ten sposób nienaturalnego sposobu poruszania się bota (np. częste ruchy prawo->lewo->prawo->lewo itp.).   
Po ustaleniu tablicy losujemy liczbę z zakresu [ 0, array[3] ) o typie integer. Wpadnięcie wylosowanej liczby do opowiedniej części odcinka powoduje określenie w którym kierunku ma poruszyć się teraz bot. Idąc od lewej strony odcinka mamy go podzielonego na części odpowiedzialne za ruch w prawo, lewo, górę oraz dół.

Jeśli array[3] == 0 tzn, że nie jest możliwy ruch w żadnym z kierunków. Wtedy zmienna stop = true. Dopóki bot nie wydostanie się z zablokowanego miejsca, nie może określić kolejnego kierunku poruszania się.

W momencie gdy bot wyliczył kierunek swojego następnego ruchu, aktualizowany jest wektor kierunku poruszania się bota i ustalana jest pozycja, którą musi osiągnąć (o 64 jednostki w prawo/lewo/górę/dół w zależności od wyznaczenia kierunku).

Dopóki bot nie osiągnie docelowego miejsca nie może postawić bomby w sposób opisany w punkcie 1.

Algorytm sprawdzania poprawności nicków

Mamy pętle, która wykonje się 4 razy. W każdym wykonaniu najpierw zapisujemy imię pierwszego, drugiego itp. gracza do zmiennej name. Następnie sprawdzamy, czy nick jest poprawny ( używam do tego wyrażenie regularnego w następującej postaci - ([a-zA-Z0-9]){4,20} - 4-20 znaków, cyfry, małe i duże litery). Jeśli nie jest, zwracamy false, jeśli jest, przechodzimy do następnego kroku.

W następnym kroku sprawdzamy, czy nick znajduje się już w tablicy. Jeśli tak, zwracamy false. Jeśli nie, dopisujemy go do tablicy.

Jeśli całej pętli udało się przejść bez zwracania false, oznacza to, że wszystkie nicki są poprawne i zaden się nie powtarza, czyli zwracamy true.

Wszystko odbywa się w funkcji check\_all\_names(), sprawdzenie nicku regexem - name\_correct(name)

Algorytm wybuchu bomby

Za wybuch bomby odpowiedzialna jest funkcja explodev. Na początku ustawia światło w centralnej pozycji wybuchu, oraz pierwsze efekty cząsteczkowe. Następnie mamy pętle, która odpowiada za rozważenie każdego z 4 kierunków wybuchu bomby (lewo, prawo, góra, dół).

Każdy kierunek opisany jest znowu pętlą. Trwa ona tyle, ile wynosi zasięg bomby, lub do jej przerwania. Za każdą iteracją rozważamy pojedyńczy klocek, z każdą iteracją odsuwając się od centrum bomby. Jeśli klocek jest niezniszczalny, przerywamy pętle. Jeśli jest zniczalny, to zmieniamy go na klocek, po którym można chodzić, odajemy punkt graczowi, do którego należy bombba, i też przerywamy pętlę. Jeśli to normalny klocek, po ktorym można chodzić, to dopisujemy jego współrzędne do tablicy danger\_list.

Po zakończeniu działania pętli, wysyłany jest sygnał do kazdego z graczy, świadczący o tym, że właśnie wybuchła bomba. Razem z sygnałem wysyłana jest tablica danger\_list, która pozwala sprawdzić graczowi, czy odniósł obrażenia, oraz nick gracza, do którego należała bomba.

Algorytm ustawiania klawiszy (sterowania)

Sterowanie odbywa się za pomocą wbudowanych w silnik akcji. Akcje odpowiadają ruchom mozliwym do wykonania przez gracza oraz jego identyfikatorze. Przykladowo akcja P1\_ui\_up jest odpowiedzialna za ruch gracza w górę. W menu sterowania gracz przypisuje odpowiednim akcjom odpowiednie klawisze, nastepnie klasa gracza samodzielnie przechwytuje odpowiednie akcje i wykonuje potrzebne ruchy.

Algorytm działania highscore

Można go podzielić na dwie części - pobieranie danych, i zapisywanie, opisane odpowiednio funkcjami get\_list oraz try\_to \_add. Będę często wykorzystywał obiekt scorePair - ma on dwa pola, nickname, oraz score. Tablicę wypełnioną obiektami typu scorePair można sortować po polach score.

get\_list - ta funkcja zwróci tablicę obiektów scorePair. Pobierane są one z pliku high.score . Jeśli plik jest pusty, lub nie istnieje, zostanie zwrócona pusta tablica. Jeśli plik coś zawiera, to na poszątku odczytujemy pierwszą linię - liczbę rekordów, a potem wykonyujemy pętlę (razy liczba rekordów). W każdej pętli zczytujemy dwie linie - nick gracza oraz jego wynik. Tworzymy z teog obiekt scorePair, i dodajemy go do tablicy. Tablicę zwracamy na koniec.

try\_to\_add - ta funkcja próbuje dopisać do pliku obiekt scorePair. Robi to następująco - pobiera listę funkcją get\_list, następnie próbuje do niej dopisać na koniec nowy obiekt. Może to zrobić, jeśli w tablicy jest poniżej 10 obiektów, lub jeśli jest ich 10, i ostatni obiekt ma mniejsze pole score od aktualnie dodawanego obiektu. Wtedy usuwamy ostatni, i dodajemy ten nasz nowy. Następnie sortujemy listę, i nadpisujemy plik. Zapisujemy najpierw liczbę rekordów, a w następnych liniach rekordy, każdy podzielony na dwie linie - najpierw nick, potem wynik. Jesli operacja się powiodła, zwracamy true, jeśli nie udało się dopisać,. zwracamy false

Jest jeszcze funkcja reset, która po prostu nadpisuje plik pustym plikiem, lub tworzy nowy pusty plik, jeśli żaden nie istniał.

Wybór węzła Highscore, Sounds oraz ConfigurationNode jako **singletony**

Singleton jest wzorcem projektowym, który ogranicza możliwość tworzenia obiektów (u nas węzłów) danej klasy do jednej instancji. Węzły będące singletonem oferują globalny dostęp do swoich atrybutów oraz metod. Singleton jest idealnym rozwiązaniem kiedy myślimy o rzeczach typu konfiguracja. Stąd w naszym projekcie w 3 węzłach zastosowaliśmy ten wzorzec. Niektóre węzły korzystają z informacji zawartych w singletonach (np. pobierają odpowiednie dane konfiguracyjne takie jak sterowanie, głośność muzyki, nicki graczy, aktualnie wybrana mapa itd.). Singletony zastępują nam więc zmienne globalne oraz dostarczają globalny dostęp do odpowiednich metod.

ConfigurationNode:

- pozwala uzyskać nam najważniejsze dane konfiguracyjne takie jak nicki graczy, ich sterowanie, głośność muzyki, to czy postać jest wybrana jako aktywna, to czy postać jest botem.  
- pozwala na regularny zapis/odczyt danych z pliku konfiguracyjnego.  
- odpowiedzialny jest za utworzenie odpowiednich InputEventów które używane są do określania sterowania graczy.  
- korzystamy z jego zmiennych/metod m.in w scenie MapScene, SoundSettings oraz Character Selection.

Sounds:

-pozwala na zmianę głośności z menu SoundOptions  
-używany do grania dźwięków w każdej scenie projektu

Highscore:

-pozwala na zapis do pliku informacji o najwyższych wynikach  
-zapewnia trwałość danych, i brak duplikatów  
-używany w kilku miejscach, więc wygodnie było go zrobić jako singleton