POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Gra turowa

1. Cel projektu

Celem projektu jest wykazanie, że student nie tylko opanował podstawowe elementy języka C++, ale również ogólnie pojęte umiejętności tworzenia mniejszych projektów programistycznych, na co składają się między innymi umiejętności odpowiedniego doboru narzędzi programistycznych oraz sporządzanie dokumentacji zbudowanych modułów pomocniczych.

2. Opis projektu

Projekt polega na stworzeniu aplikacji konsolowej symulującej grę walki turowej z implementacją oddziaływań między przeciwnikami (przykład: system walki w grze Pokemon).

Gracz posiada zestaw 6 stworzeń którymi kieruje. Celem gry jest pokonanie wszystkich przeciwników stojących na drodze, z jak najmniejszym kosztem dla swojej drużyny.

Aplikacja winna być uruchamiana z poziomu linii poleceń (ang. *command line*) i, bazując na komendach czytanych z klawiatury, wykonywać odpowiednie zagrania dla postaci.

Dokładna specyfikacja programu znajduje się w dalszych częściach dokumentu.

3. Opis rozgrywki

Rozgrywka winna dzielić się na co najmniej 4 rundy, przy czym każda runda winna posiadać przynajmniej 4 przeciwników. Wszyscy przeciwnicy są losowani z puli 15 stworzeń, z których gracz może wybrać swój zespół.

Na początku rozgrywki gracz może wybrać sobie zespół składający się z sześciu stworzeń z predefiniowanej puli.

Rozgrywka dzielona jest na tury gracz–przeciwnik, gdzie w jednej turze można wykonać następujące akcje:

- Użycie umiejętności specjalnej/ataku, lub

- Wymiana stworzenia, lub
- Ewolucja stworzenia.

Po zmniejszeniu statystyki Punkty Życia stworzenia, mdleje ono, i nie można go ponownie wykorzystać do końca walki. Po pokonaniu przeciwnika przez gracza, winien pojawić się komunikat dający możliwość zapisania i wyjścia z gry albo kontynuowania rozgrywki.

W momencie pokonania wszystkich przeciwników gra się kończy zwycięstwem gracza.

4. Wymagania niefunkcjonalne

Projekt powinien być zaimplementowany zgodnie z zasadami produkcji czystego kodu wysokiej jakości oraz zgodny z dobrymi praktykami programistycznymi (zarówno uniwersalnymi jak i stricte dotyczącymi języka C++). Mając na uwadze fakt, że część z tych kryteriów może być kwestią dyskusyjną, student winien upewnić się, jego zrozumienie danych pojęć pokrywa się z elementami przedstawionymi na ćwiczeniach, na przykład poprzez konsultacje z prowadzącym.

Implementacja winna być przemyślana. Oznacza to, że należy dobrać możliwie jak najlepsze narzędzia programistyczne i zastosować je wraz z dobrymi praktykami ich używania. Wynika z tego fakt, że rozwiązanie, które "*po prostu działa*", może nie zdobyć wymaganej do zaliczenia liczby punktów.

Niekompilujące się programy będą automatycznie otrzymywały zero punktów. Należy odpowiednio skorzystać z podzielenia projektu na wiele plików.

Projekt winien być dokumentowany. Wymaganym jest, aby każda nietrywialna funkcja, metoda oraz klasa (ale już nie atrybuty klas) miały dotyczący ich komentarz zgodny ze standardem Doxygen. Głównymi aspektami, na których należy się skupić, są parametry (`@param`), wartości zwracane (`@return`) oraz krótki opis działania (przeznaczenia) danej metody / klasy / funkcji.

W tym opisie nie należy zagłębiać się w szczegóły jak dana idea jest implementowana. Proszę się skoncentrować na opisywaniu tego **co** jest robione, a nie **jak** coś jest robione.

Dobrym przykładem jest chociażby dokumentacja klasy String z Javy, której opis metod można znaleźć tutaj.

W narzędziach typu CLion czy Visual Studio, po zaimplementowaniu funkcji, klasy czy metody, jesteśmy w stanie wygenerować szablon takiej dokumentacji za pomocą wprowadzania następującej sekwencji znaków tuż nad elementem, który chcemy udokumentować: `/**`, a następnie klikając przycisk `Enter`.

5. Wymagania funkcjonalne

Każde stworzenie musi posiadać atrybuty definiujące jej zachowania i wpływ na rozgrywkę, w atrybuty wlicza się:

- Siła – definiuje, ile obrażeń może zadać stworzenie jednym atakiem;
- Zręczność – definiuje, jaka jest szansa na uniknięcie ataku przeciwnika;
- Punkty Życia – definiuje, ile obrażeń może przyjąć stworzenie zanim zemdleje;
- Moc Specjalna – określa charakterystyki mocy specjalnej dla danego stworzenia oraz maksymalną liczbę jej użyć w jednej walce.
- Punkty EXP – ile punktów EXP jest przyznawane za pokonanie przeciwnika.

Przy czym Moc Specjalna musi być adekwatna do typu stworzenia (np. stworzenie typu wodnego, nie może użyć tornada czy ognistego ataku jako mocy specjalnej). Moc specjalna również dzieli się na moce ofensywne i defensywne, które mogą wpłynąć tymczasowo (ilość tur) na statystyki gracza lub przeciwnika.

Każde stworzenie posiada również swój typ (Woda, Ziemia, Powietrze, Ogień, Lód, Stal) które wpływają jak dobrze ich ataki wpływają na inne stworzenia. Tabelka oddziaływań znajduje się poniżej (Tabela 1).

Stworzenie Atakujące	Woda	Ziemia	Powietrze	Ogień	Lód	Stal
Woda						
Ziemia						
Powietrze						
Ogień						
Lód						
Stal						

Tabela 1. Interakcje typów między sobą.

Gdzie **zielony** kolor oznacza zwiększoną efektywność, natomiast **żółty** kolor oznacza zmniejszoną efektywność.

Program powinien posiadać pulę przynajmniej 15 stworzeń, z których gracz może wybrać swoją drużynę, oraz z których losowane są drużyny przeciwników.

Gra winna posiadać przynajmniej 4 przeciwników, każdy z nich powinien posiadać przynajmniej 4 stworzenia. Należy zaimplementować poziomy trudności które te wartości zmieniają.

Wybór stworzeń oraz ataków ma się odbywać przez klawiaturę. Interfejs użytkownika powinien być czytelny, oraz posiadać instrukcję obsługi po wpisaniu na konsoli komendy `-h` lub `--help`.

Każde stworzenie może po uzyskaniu określonej dla niego liczby EXP ewoluować, zwiększając swoje atrybuty. Gracz ma mieć możliwość decydować które dwa atrybuty chce powiększyć w stworzeniu.

6. Kryteria oceniania

Tabela 2 określa liczbę punktów do zdobycia za każde z wymienionych wymagań.

Wymaganie	Liczba punktów	Dodatkowy komentarz
Odpowiednie dobieranie i wykorzystywanie narzędzi programistycznych.	5	Przykładowo używanie standardowych algorytmów zgodnie z ich przeznaczeniem, raczej używanie szablonów niż makr, korzystanie z `std::function` zamiast ze wskaźników na funkcje, odpowiednie dobieranie kontenerów zgodnie z ich przeznaczeniem. Dobieranie i stosowanie narzędzi tak, aby jasno wskazywały intencje programistyczne – zawiera się w tym również czysty kod, a zatem i wybieranie deskryptywnych, dobrych nazw zmiennych.
Stosowanie się do *const-correctness*.	1	Oznacza to stosowanie `const` nie tylko wszędzie tam, gdzie można, ale też wszędzie tam, gdzie koncepcyjnie dany element nie powinien być zmieniany.
Unikanie niepotrzebnych kopii danych w programie.	1	Zgodnie z wiedzą przedstawioną na ćwiczeniach. Tyczy się to typów większych niż prymitywy – wskazanym jest kopiowanie zmiennych typu `int` czy `double` zamiast przekazywać je przez `const&` (oczywiście w przypadku niemodyfikowanych kopii oznaczenie ich przez `const` jest wciąż wymagane).
Sporządzenie dokumentacji zgodnej z uproszczonym standardem Doxygen.	4	
Klarowność komunikatów błędów przy niepoprawnym użytkowaniu aplikacji.	3	
Poprawne zaimplementowanie hierarchii polimorficznej/generacji przeciwników.	5	Przy generacji przeciwników, student winien się zastanowić nad algorytmem generacji, gdzie każde stworzenie powinno posiadać zbalansowane cechy i umiejętności. Algorytm nie powinien

		być trywialny. W razie wątpliwości proszę skonsultować się z prowadzącym.
Poprawna implementacja zapisywania i wczytywania stanu gry do pliku `.txt`.	4	
Poprawna implementacja tablic interakcji.	3	Rozwiązanie tablic interakcji powinno być jak najbliższe optymalnemu. Należy wziąć pod uwagę zamknięty zakres interakcji oraz ich potencjalnie częste wyszukiwanie.
Poprawna implementacja Atrybutów i ich wpływu na rozgrywkę.	5	
Poprawna implementacja rozgrywki.	10	W rozgrywkę wciela się: interakcja użytkownika, wymiana stworzeń, akcje turowe, wymiana przeciwników, **skalowalność trudności**, projekt interfejsu.
Poprawna implementacja ewolucji stworzeń.	5	W implementację ewolucji wlicza się obliczanie punktów EXP potrzebnych do zdobycia aby stworzenie ewoluowało oraz wybór powiększenia statystyk.
Poprawna implementacja mechaniki ataku.	3	
Poprawna implementacja funkcji `--help`.	1	

Tabela 2. Kryteria oceniania.

7. Obrona

Student będzie przepytany ze szczegółów implementacyjnych swojego rozwiązania. Brak zrozumienia i identyfikacji dowolnego fragmentu kodu może być podstawą do **niezaliczenia całego projektu**. W związku z tym sugeruje się, aby studenci, którzy ukończyli projekt semestralny długo przed terminem obrony, zadbali o to, aby wszystkie szczegóły projektu zostały przypomniane.

8. Dodatkowe uwagi

Zabrania się stosowania rozwiązań, których student nie rozumie w dobrym stopniu. Każde wykorzystane narzędzie musi być opanowane przez studenta. Zezwala się na używanie elementów, które nie były przedstawione na ćwiczeniach czy wykładzie, lecz należy

równocześnie pamiętać, że odpytywanie podczas obrony może się skoncentrować również i na tych elementach.

W przypadku dowolnych niejasności należy skonsultować treść i wymagania projektu z prowadzącym ćwiczenia.