Programowanie Komputerów 4

Sprawozdanie z projektu

Projekt zaliczeniowy: gra reactor idle

Prowadzący laboratorium: dr Piotr Pecka

INF AEiI Grupa 2
Michał Urbańczyk

Temat projektu

Jako projekt zaliczeniowy wybrałem stworzenie gry typu incremental. Celem gry jest jedynie zdobycie największej ilości pieniędzy w jak najkrótszym czasie, jej zakończenie nie istnieje, chociaż można za nie uznać przepełnienie zmiennej przechowywującej wartości pieniędzy.

Analiza tematu

Tworzona przeze mnie gra miała mieć za zadanie przede wszystkim być łatwa do dalszego rozwoju. Z tego powodu postanowiłem zaimplementować pobieranie danych na 3 sposoby – hard-coded poprzez tablicę char, soft-coded poprzez funkcję dodającą nowe rzeczy, oraz poprzez ładowanie pliku json.

Biblioteki użyte w projekcie to:

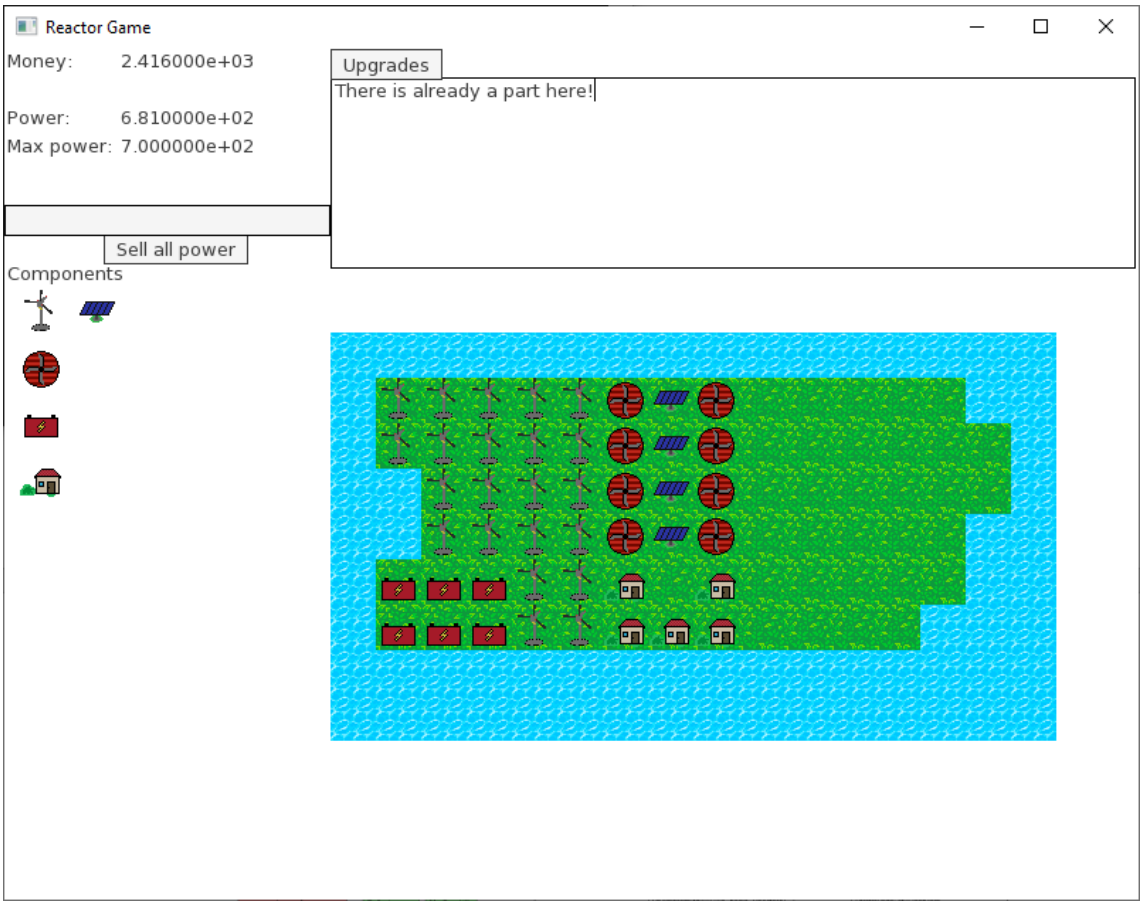
- **SFML** – podstawowa biblioteka odpowiadająca za wyświetlanie zasobów na ekran, jak i tworzenie okien.
- **TGUI** – biblioteka korzystająca z SFML, która pozwala na szybkie i relatywnie proste tworzenie interfejsów graficznych, oraz pozwala na stworzenie GUI za pomocą zewnętrznego edytora.
- **Nlohmann/json** – biblioteka pozwalająca na łatwe wczytywanie, zapis oraz operacje na plikach typu JSON.

Wszystkie powyższe biblioteki są na licencjach zlib lub MIT.

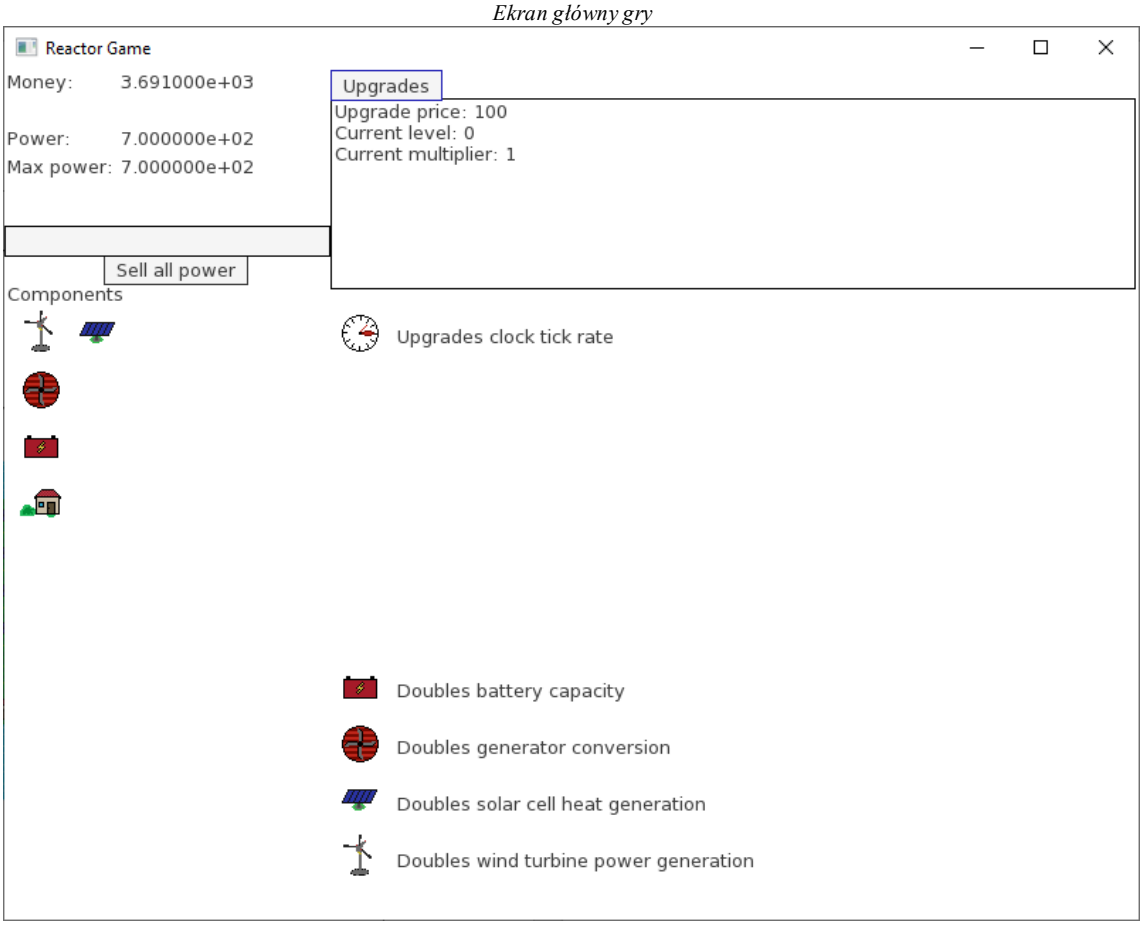
Specyfikacja zewnętrzna

Dokumentacja: projekt napisany jest w zgodności z Doxygen, więc istnieje prosty sposób na generację dokumentacji. Dostępny jest również plik Reactor2.uxf stworzony za pomocą programu UMLet pozwalający na wstępne zapoznanie się z strukturą klas.

Interfejs użytkownika: Gra składa się z dwóch głównych scen – sceny gry oraz ulepszeń. Scena gry pozwala na budową oraz sprzedaż budynków, podczas gdy scena ulepszeń pozwala na ulepszanie już istniejących budynków.



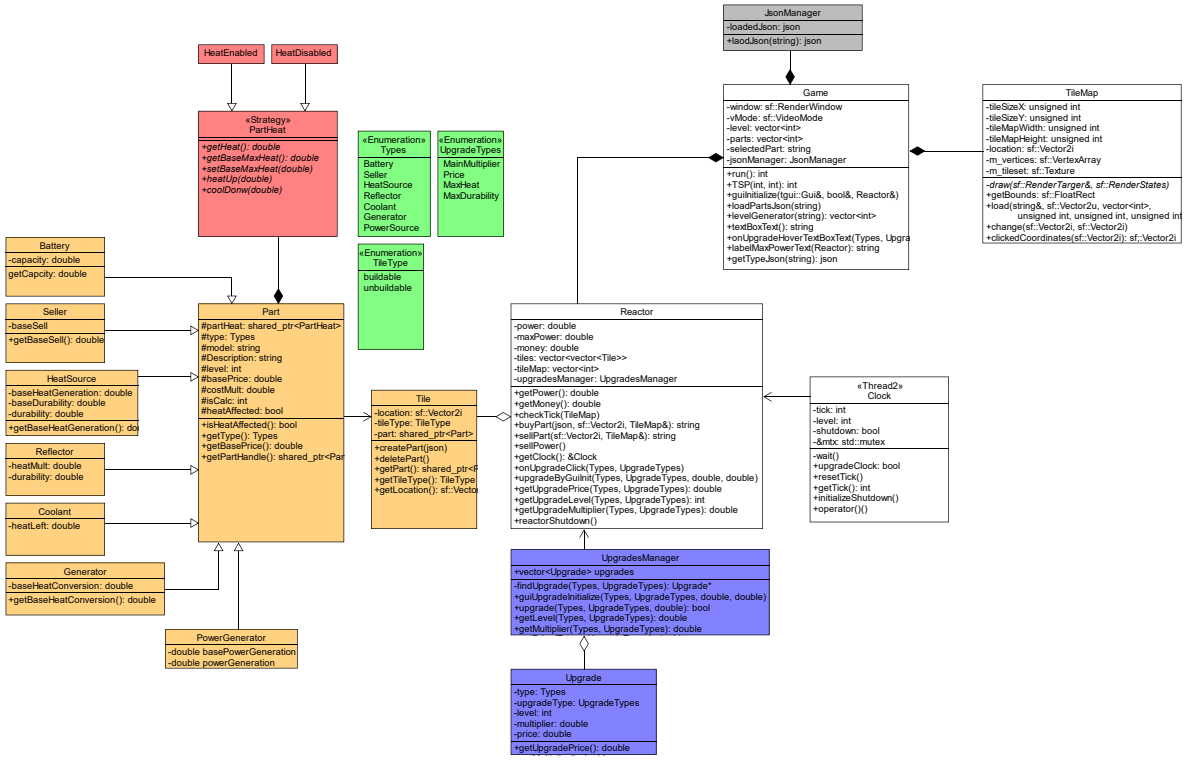
Ekran główny gry



Ekran ulepszeń

Specyfikacja wewnętrzna

Diagram klas, dostępny również na repozytorium github:



Opis głównych klas:

Game - główna klasa gry - odpowiada za malowanie wszystkich obiektów na ekran oraz obsługę interakcji z graczem

JsonManager - Klasa służąca do ładowania pliku JSON z wszystkimi częściami.

Reactor - Klasa odpowiadająca za reaktor i wszystko z nim związane - zarządzanie częściami, obliczenia ilości pieniędzy oraz energii.C

Tile - Klasa reprezentująca pojedyncze pole na planszy. Pole może posiadać część lub nie, oraz może być zdatna lub niezdatna do budowy

Part - Klasa przedstawiająca część w grze. Przykładowe części to turbina, bateria itp.

PartHeat - Klasa stworzona jako wzorzec strategii w celu ograniczenia potrzeby przepisywania tych samych elementów do wielu obiektów

Clock - Klasa odpowiadająca za zarządzanie, czy powinna wykonać się tura (tick) gry. Jest przystosowana do pracy w wątkach.

Upgrade - Klasa odpowiedzialna za tworzenie ulepszenia i jego ulepszania.

UpgradesManager - Klasa odpowiedzialna za podawanie wartości mnożnika danego ulepszenia. Zawiera ona w sobie wektor różnych ulepszeń, wśród których sprawdza, czy dana rzecz jest ulepszona.

Wszystkie klasy dziedziczące z Part są jedynie reprezentacjami danego budynku

Testowanie i uruchamianie

Analiza pomysłów i błędów:

Podczas tworzenia projektu testowałem 3 różne sposoby na tworzenie dużej ilości elementów i związanych z nimi danych.

Pierwszy sposób to implementacja klasy TileMap – jej malowanie odbywa się poprzez iterację przez tablicę intów które odpowiadają za wybranie odpowiedniej tekstury. Kod ten ma swoje plusy – prostota implementacji(tworzymy jedynie tablicę intów). Posiada jednak szereg wag – w ten sposób możemy zakodować tylko jedną wartość oraz samo czytanie i uzupełnianie tabeli np. 16x9 potrafi być trudne. Wymagane jest również zakodowanie wymiarów tabeli.

Jako drugi sposób stworzyłem klasę JsonManager, która to za pomocą biblioteki Nlohmann/json. Plusem tego rozwiązania jest łatwość zapisywania i wczytywania danych oraz łatwe modyfikacje nie wymagające kompilacje – dodanie dodatkowej części lub jej edycja odbywa się poprzez edycję odpowiedniego pliku tekstowego z rozszerzeniem json. Wadami są natomiast problemy w konwersji i odróżnianiu jednostek (często trzeba używać get<TYP>() w celu otrzymania odpowiedniego typu). Dodatkową wadą może być to, że drobna literówka, niewykrywalna przez kompilator, może spowodować czasochłonne szukanie błędu w kodzie.

Trzecim i ostatnim sposobem było stworzenie odpowiednich klas – Upgrade do przechowywania danych o jakiejś rzeczy oraz UpgradesMangerdo łatwego dostępu i wyszukiwania tych danych. Zaletą tego rozwiązania jest największa przejrzystość: stworzenie nowego ulepszenia wymaga jedynie wywołanie odpowiedniej funkcji. Dodatkowo, obiekty mogą zostawać dodawane podczas działania programu, co jest niemożliwe w pierwszy i drugim sposobie. Minusami jest duża liczba kodu potrzebna do pełnego działania implementacji.

Finalnie doszedłem jednak do wniosku, że nie ma „najlepszego sposobu”. Każda z wyżej wymienionych implementacji powinna być wykorzystywana w sprzyjających sytuacjach: pierwsza, gdy potrzebujemy bardzo szybko coś zaimplementować. Druga, gdy chcemy zakodować coś w średnim czasie i jesteśmy w stanie dobrze udokumentować co jest czym w pliku json i zawsze wykorzystywać to w ten sposób. Trzeci, gdy potrzebujemy solidny, łatwo modyfikowalny i solidny, a jednocześnie mamy wystarczająco czasu na jego zakodowanie.

Uwagi i wnioski

Wykorzystanie zaawansowanych funkcji języka pozwoliło mi lepiej zrozumieć działanie języka i zaobserwować korzyści i wady różnych rozwiązań. Porównując mój projekt z tego, jak i zeszłego semestru pozwala na zauważenie znacznej różnicy – front jak i backend są ze sobą połączone znacznie luźniej, oraz wiele żmudnych zadań rozwiązywane jest w kilku liniach. Dodatkowo używanie takich funkcji języka jak wątki pozwala na znaczne przyspieszenie działania programu.