POZNAN UNIVERSITY OF TECHNOLOGY

Tymoteusz Bleja
Paweł Husak
Patryk Imosa
Magalena Łątkowska

Internetowa gra edukacyjna ucząca podstaw pracy z programem git

Praca inżynierska

Promotor: dr hab. inż. Marek Andrzej Wojciechowski

Poznań, 2017

Spis treści

Rozdział 1

Wstęp

1.1 Zasady odnośnie pisania pracy

- Piszemy w formie bezosobowej. Można też niby w 1.os liczby mnogiej czyli Źrobiliśmy...", ale niektórzy akceptują tylko bezosobową, czyli Źrobiono...".
- słów niepolskich - „Nie można więc bezpośrednio w tekście używać słów angielskich. Jeżeli już — powinny być wyróżnione kursywą" - niektórzy podobno bardzo hejcą za angielskie słowa niestety.
- Jak chodzi o bibliografię, to w wolnej chwili dołączcie linki do stron z jakich korzystaliście, ja to potem ogarnę i zapiszę w takiej formie jak trzeba. Ale spokojnie, bo robienie bibliografii raczej zostawię na koniec.
- Ogólnie nie przejmujcie się strukturą, formatowaniem czy innymi formalnymi bzdetami, ja potem będę to ogarniać żeby było wg zasad więc nie traćcie czasu na ogarnianie takich rzeczy.

Między innymi: skąd wgl pomysł - bo Git jest super i konieczny a ciężko się go samemu nauczyć, nauka z wielu źródeł jest chujowa, większość ma tylko blade pojęcie a potem idzie do pracy i dupa - co z tego że nauczyli się na studiach programować jak nie potrafią korzystać z Gita i współpracować z zespołem

Cel i zakres pracy

cel - nauka fajna łatwa i przyjemna, oraz praktyczna, obycie z typowymi scenariuszami jakie mogą być potrzebne w pracy

Coś o tym dlaczego akurat przeglądarkowa gra, czemu z grafiką 3D itp.

Cytat z karty pracy : Zapoznanie się z systemem Git. Opracowanie koncepcji interaktywnego samouczka do nauki podstaw korzystania z systemu Git. Opracowanie architektury systemu. Implementacja i testowanie systemu. Przygotowanie dokumentacji technicznej i użytkowej.

Rozdział 2

Podstawy teoretyczne

2.1 Sysytem kontroli wersji Git

Może rozdział powinien się nazywać „Wstęp teoretyczny", albo od razu „System kontroli wersji Git" lub samo „Git", i wtedy bez tej podsekcji - i tak w rozdziale nie może być tylko jedna

Krótki opis systemów kontroli wersji, do czego służą i jakie dają korzyści. Wyjaśnienie dlaczego Git się wyróżnia, co ma szczególnego i bardziej obszerny jego opis.

W opisie Gita o tym czym jest repozytorium, przestrzeń robocza, indeks, indeksowanie zmian, zatwierdzanie zmian, rewizja, gałąź, HEAD, łączenie gałęzi, cofanie zmian, rebase, repozytorium zdalne, push, pull, gałąź zdalna, śledzenie gałęzi zdalnej. W nawiasach do wszytskie angielskie pojęcia.

Jakaś subsekcja o tym dlaczego programiści powinny go znać i potrafić używać jako narzędzia w pracy, do czego im się przyda i że Git często nie jest możliwością, lecz koniecznością.

Ogólnie myślę że tak z 2-3 strony?

2.2 Gitflow

Praca z wykorzystaniem systemu Git może przebiegać w zgodzie ze ściśle określonym cyklem. Ustalenie reguł, których będą przestrzegać wszyscy programiści pracujący nad danym projektem, może znacząco ułatwić synchronizację pracy i pomóc w sprawnym zarządzaniu i wydawaniu nowych wersji oprogramowania. Jedną z bardziej popularnych metodyk jest Gitflow. Określa ona cykl pracy, oparty na korzystaniu z różnych gałęzi, z których każda ma dokładnie określoną rolę.

2.2.1 Podstawowe założenia

Zamiast korzystać jedynie z domyślnej gałęzi *master*, Gitflow wykorzystuje dwie główne gałęzie do rejestrowania historii projektu. Jedną z nich jest gałąź produkcyjna *master*, na której przechowywane są tylko oficjalne wydania (ang. *release*). Druga służy jako gałąź deweloperska i jest nazywana *develop*. Przeznaczona jest do integracji bieżących prac programistycznych i nowych funkcji.

Obie wspomniane gałęzie w idealnym przypadku powinny składać się jedynie z rewizji powstałych poprzez scalenie gałęzi (ang. *merge commits*). Nie powinno się pracować i zatwierdzać zmian bezpośrednio na nich. Wszystkie modyfikacje kodu należy przeprowadzać na osobnych, dedykowanych gałęziach, tworzonych tymczasowo, w dokładnie określonym celu. Koncepcja Gitflow wyróżnia trzy rodzaje takich gałęzi:

- wprowadzające nowe funkcje (ang. *feature branch*),
- przygotowujące do opublikowania nowego wydania (ang. *release branch*),
- zawierające niezbędne i szybkie poprawki (ang. *hotfix branch*).

Z racji braku polskich odpowiedników nazw wymienionych powyżej gałęzi, w przypadku odniesienia do nich, w dalszej części pracy wykorzystywane będzie oryginalne nazewnictwo angielskie.

Z technicznego punktu widzenia typy gałęzi wykorzystywanych w Gitflow niczym się nie różnią, są to zwykłe gałęzie systemu Git. Są one szczególne jedynie pod względem konkretnych celów w jakich są używane i pewnych ograniczeń dotyczących procesu tworzenia ich i łączenia. Każdy rodzaj może powstać jedynie przez rozgałęzienie z określonej gałęzi (produkcyjnej lub deweloperskiej), a na koniec musi zostać połączony ze ściśle ustaloną gałęzią lub gałęziami.

2.2.2 Rozszerzanie funkcjonalności

Jednym z podstawowych założeń Gitflow jest implementowanie każdej nowej funkcji oprogramowania na osobnej, dedykowanej gałęzi typu *feature branch*, której nazwa powinna zaczynać się od „feature/”. Taka gałąź może powstać jedynie poprzez rozgałęzienie z głównej gałęzi *develop*. Z założenia ma składać się wyłącznie z rewizji zawierających zmiany dotyczące danej nowej funkcji i istnieć tak długo, jak długo trwać będzie proces implementacji.

Kiedy cel zostanie zrealizowany, gałąź typu *feature branch* powinna być połączona z gałęzią *develop*. Istotne jest aby operacja scalania nie została wykonana poprzez przewinięcie do przodu (ang. *fast forward*), czyli zwykłe przesunięcie wskaźnika HEAD. Chodzi o to by główna gałąź deweloperska nie zawierała wszystkich rewizji pochodzących z gałęzi dołączanej, a jedynie jedną powstałą jako łącznik dwóch gałęzi (ang. *merge commit*). W przeciwnym wypadku określenie, które rewizje dotyczą wprowadzenia konkretnej funkcji, wymagałoby dokładnego przejrzenia zawieranych przez nie zmian. A w rezultacie znacznie trudniej byłoby usunąć pojedynczą funkcję z głównej gałęzi.

Po włączeniu zmian z gałęzi przeznaczonej do implementacji nowej funkcji do głównej gałęzi *develop*, należy usunąć tymczasową gałąź.

2.2.3 Przygotowywanie nowego wydania

Kiedy stan kodu aplikacji jest stabilny i działa zgodnie z oczekiwaniami, a wszystkie funkcje które powinny się znaleźć w nowym wydaniu oprogramowania są już zaimplementowane i włączone do głównej gałęzi deweloperskiej, należy rozpocząć proces publikacji nowej wersji. Zgodnie z koncepcją Gitflow, wszystkie niezbędne ostatnie poprawki i drobne zmiany

dotyczące przygotowania nowego wydania, powinny zostać przeprowadzone na osobnej, specjalnie utworzonej w tym celu gałęzi, nazywanej *release branch*.

Tworzenie takiej gałęzi odbywa się poprzez rozgałęzienie z gałęzi *develop*. W tym momencie ustala się też numer wydania, czyli numer wersji publikowanego oprogramowania. Nazwa gałęzi powinna być formatu „release/numer-wydania". Od tego momentu wszystkie zmiany zachodzące na głównej gałęzi *develop* będą dotyczyć następnej publikacji, nie zostaną uwzględnione w aktualnym wydaniu. Dzięki utworzeniu gałęzi typu *release*, prace mogą iść dwutorowo, poprzez przygotowywanie publikacji nowej wersji, oraz równolegle rozwijanie i rozszerzanie funkcjonalności oprogramowania. Jako przygotowanie do wydania rozumie się poprawę drobnych błędów takich jak literówki i inne niewielkie niedociągnięcia.

W momencie w którym oprogramowanie będzie już przygotowane do opublikowania, należy scalić gałąź *release* z główną gałęzią produkcyjną *master*. Analogicznie jak w przypadku scalania gałęzi typu *feature branch*, należy zwrócić uwagę, aby operacja połączenia nie polegała na przewinięciu do przodu. Konieczne jest aby w wyniku jej wykonania utworzona została nowa rewizja. Należy jej nadać etykietę (ang. *tag*) z numerem wydania, aby ułatwić wyszukiwanie konkretnych wersji oprogramowania na gałęzi produkcyjnej. Następnie należy również scalić gałąź dotyczącą najnowszego opublikowanego właśnie wydania z gałęzią *develop*. Gałęzie typu *relese* są tymczasowe, więc po włączeniu jej do obu głównych gałęzi należy ją usunąć.

2.2.4 Naprawa błędów wymagających szybkiego rozwiązania

Kolejnym typem gałęzi wykorzystywanych w koncepcji Gitflow są gałęzie nazywane *hotfix branches*, przeznaczone do naprawy niecierpiących zwłoki błędów, odkrytych w opublikowanym oprogramowaniu, najczęściej zgłoszonych przez użytkowników końcowych. Dotyczy to takich wad i problemów, które są zbyt poważne, aby mogły zostać naprawione dopiero w następnym wydaniu. W przypadku mniej istotnych niedociągnięć wystarczy poprawić dane aspekty w normalnym trybie pracy i włączyć je do gałęzi *develop*, żeby zostały uwzględnione przy kolejnej publikacji przez gałąź typu *release*.

Gałąź typu *hotfix* jest jedynym rodzajem gałęzi, który powstaje poprzez rozgałęzienie z głównej gałęzi produkcyjnej *master*. Jej nazwa powinna zaczynać się od wyrażenia „hotfix/". Taka gałąź zawiera tylko zmiany dotyczące naprawy wykrytego błędu, które powinny być wykonane w możliwie najkrótszym czasie. Kiedy problem zostanie rozwiązany, przeznaczoną mu gałąź należy połączyć z powrotem z gałęzią *master*. W tym przypadku również należy dokonać łączenia nie poprzez przewijanie do przodu. W rezultacie na głównej gałęzi produkcyjnej powstaje nowa rewizja, której należy nadać etykietę z nowym numerem wersji. Skutkiem jest publikacja nowego wydania oprogramowania, która, w przeciwieństwie do procesu z wykorzystaniem gałęzi typu *release* nie była planowana.

Kolejnym krokiem jest scalenie gałezi *hotfix* z główną gałęzią *develop*. Jeżeli istnieje w danym momencie gałąź typu *release*, to do niej również należy włączyć zmiany dotyczące naprawy błędu. Na koniec należy usunąć tymczasową gałąź.

Rozdział 3

Projekt

3.1 Założenia

Głównym celem samouczka GITar-Hero jest zapoznanie użytkownika z podstawowymi poleceniami systemu kontroli wersji Git, w sposób przyjemny i zrozumiały. Gra, przez połączenie nauki i rozrywki, ma za zadanie zachęcić i ułatwić proces uczenia się. Kolorowa i ruchoma grafika 3D uatrakcyjnia tę naukę, a możliwość zdobywania punktów i naturalna chęć osiągnięcia jak najlepszego wyniku dodatkowo mobilizuje użytkownika.

Z założenia, ważniejsze od dogłębnego zrozumienia strony teoretycznej systemu Git, było nauczenie właściwego korzystania z poleceń. Gra ma służyć jako samouczek, uczący praktycznego wykorzystania systemu wersji i pokazujący typowy scenariusz, jaki najczęściej występuje podczas wytwarzania oprogramowania. Po zagraniu w grę użytkownik powinien już swobodnie wykonywać komendy systemu Git, zarówno pracując indywidualnie, jak i potrafić właściwie współpracować z zespołem.

Celem było pokazanie, że wbrew panującej powszechnie opinii, korzystanie z systemu kontroli wersji Git nie musi przysparzać problemów ani trudności. Ponadto gra ma przyzwyczaić użytkownika do korzystania z wiersza poleceń. Jeżeli ma się opanowane komendy, jakie należy wprowadzać w konsoli, z reguły będzie się potrafiło skorzystać z dowolnego programu z graficznym interfejsem do obsługi systemu Git. W drugą stronę taka zależność nie występuje. W związku z tym, tylko umiejętność korzystania z systemu Git w wierszu poleceń pozwala swobodnie korzystać z tego systemu kontroli wersji, niezależnie od środowiska i zainstalowanych programów.

3.2 Przebieg gry

Gra zaczyna się od krótkiego wprowadzenia, informującego użytkownika, na czym będzie polegała rozgrywka i do czego służą poszczególne elementy interfejsu. Po zapoznaniu się z krótką instrukcją rozpoczyna się gra. U góry, po prawej stronie, wyświetla się aktualne zadanie i pierwszy krok, który należy wykonać. Z założenia użytkownik nie zna poleceń systemu Git, dlatego automatycznie otwiera się pomoc z zakładką informującą czym jest repozytorium sytemu kontroli wersji i jak je zainicjować. Pomoc zawiera wszystko, co gracz musi wiedzieć, aby poprawnie wykonać dany krok. Po wprowadzeniu przez niego właściwego polecenia i zatwierdzeniu go poprzez przycisk Enter, aktualny krok zostaje

zaliczony i następuje przejście do kolejnego. Dodatkowo w katalogu projektu, po lewej stronie, pojawiają się aktualne pliki, jakie znajdują się na tym etapie w folderze z repozytorium. Poza tym akcje wykonywane na repozytorium są odwzorowywane przez grafikę 3D, która reaguje odpowiednio na wpisane komendy. Obrazuje to, jak wywołane polecenie działa na stan repozytorium i pozwala użytkownikowi lepiej zrozumieć skutki wykonywanych komend.

Po poprawnym wykonaniu przez użytkownika wszystkich kroków zadania, otrzymuje on punkty. Ich liczba jest zależna od czasu, jaki pozostał do końca zadania. Im szybciej gracz ukończy, tym więcej punktów dostanie. Możliwe jest także nie otrzymanie żadnych punktów za wykonanie zadania, jeżeli przekroczony zostanie przydzielony do niego czas.

Kolejne zadania stopniowo wprowadzają nowe komendy. Ich poziom trudności rośnie, zawierają one coraz więcej kroków. Jeżeli użytkownik nie będzie potrafił wykonać aktualnego kroku, może w dowolnej chwili wpisać w konsoli 'help'. Otworzy się wówczas pomoc i gracz będzie mógł poszukać potrzebnych mu informacji.

Po przejściu całego scenariusza wyświetli się podsumowanie, zawierające liczbę zdobytych przez gracza punktów oraz podstawowe statystyki, zawierające informację o liczbie popełnionych błędów i komendach, z którymi miał najwięcej problemów.

3.3 Zadania

Scenariusz rozgrywki składa się z zadań zawierających niezbędne komendy do typowego wykorzystania systemu kontroli wersji Git. Użytkownik uczy się najpierw o tym, czym jest repozytorium. Poznaje dwie podstawowe metody pozyskania takiego repozytorium — poprzez sklonowanie istniejącego lub przez założenie go w wybranym folderze z projektem. W kolejnym etapie zapoznaje się z pojęciami takimi jak przestrzeń robocza, indeks, indeksowanie plików, zatwierdzanie zmian i rewizja. Potrafi rozróżnić co oznacza plik aktualny, zmodyfikowany, nieśledzony lub niezaindeksowany. Przed wprowadzeniem kolejnych poleceń i terminów zadania koncentrują się na tej tematyce, aby użytkownik zdążył je dobrze opanować.

Po wykonaniu kilku zadań dotyczących wspomnianych wyżej zagadnień, użytkownik uczy się o gałęziach, poznaje sposoby tworzenia ich i przełączania się między nimi. Wprowadzane są też pojęcia takie jak gałąź tematyczna, poświęcona konkretnej dodatkowej funkcji aplikacji (ang. *feature branch*). Przy okazji nadal utrwalane są komendy przedstawione w pierwszych zadaniach, dotyczące indeksowania plików i zatwierdzania zmian.

Kolejnym etapem jest nauka scalania, lub inaczej łączenia gałęzi. Rozróżnione są przy tym odmienne sposoby na wykonanie tej operacji. Następnie pojawia się podstawowy sposób wycofywania zmian i usuwania wykonanych wcześniej rewizji.

Kiedy użytkownik opanuje już komendy i sposób pracy w lokalnym repozytorium, wprowadzane jest pojęcie zdalnego repozytorium oraz sposoby komunikacji i synchronizacji z nim. W zadaniach pojawiają się komendy dotyczące ściągania i przesyłania zmian. Poruszany jest także temat zdalnych gałęzi i sposoby skonfigurowania lokalnej gałęzi śledzącej zdalną.

Ponadto podczas nauki poleceń systemu Git użytkownik zapoznaje się też z dobrymi praktykami i metodyką Gitflow, co uczy go właściwych nawyków i odpowiedniego po-

rządkowania pracy nad projektem. Zaznajamia się z koncepcją tworzenia osobnych gałęzi przeznaczonych do implementacji nowych funkcji lub naprawy błędów.

Zadania są podzielone na kroki, dzięki czemu użytkownik uczy się charakterystycznych sekwencji poleceń, często występujących razem. Ma to również na celu zautomatyzowanie zachowania użytkownika w prawdziwych przypadkach, z którymi może się spotkać w domu lub pracy. W systemie Git czasem możliwe jest użycie kilku różnych poleceń, aby osiągnąć ten sam rezultat. Zadania i kroki dopuszczają wszystkie właściwe komendy.

3.3.1 Pomoc

Wszystkie materiały dydaktyczne znajdują się w pomocy. Jest ona podzielona na zakładki, z których każda dotyczy jednej komendy lub pojęcia. Treść zawarta w niej wystarcza, aby osoba nie znająca poleceń systemu Git była w stanie poprawnie wykonać zadania ze scenariusza rozgrywki. W przypadku niektórych pojęć jest jednak bardziej obszerna i wykracza poza wymaganą do przejścia gry wiedzę. Zawiera informacje, które uznano za szczególnie istotne i przydatne do właściwego zrozumienia systemu Git.

Za każdym razem, gdy podczas rozgrywki w jednym z kroków zadania pojawia się nowe polecenie, pomoc otwiera się automatycznie na odpowiedniej zakładce. Użytkownik może poświęcić dowolnie dużo czasu na zaznajomienie się z jej treścią, a ponadto w każdej chwili ma możliwość ponownego przeczytania informacji dotyczących wcześniejszych poleceń.

3.3.2 Punkty za rozwiązanie

W projekcie wprowadzono elementy gamifikacji, takie jak punkty za rozwiązanie zadań. Mają one za zadanie zwiększyć zaangażowanie użytkownika. Z każdym rozegraniem, gracz będzie starać się poprawić swój dotychczasowy wynik. W ten sposób co raz szybciej i pewniej będzie korzystał z poleceń systemu Git. Ponadto punkty wprowadzają element rywalizacji pomiędzy użytkownikami, którzy będą dążyć do tego aby zająć jak najwyższe miejsce w rankingu.

Punkty obliczane są na podstawie czasu, w jakim użytkownik rozwiązał zadanie.

Został dla nich stworzony komponent, który reaguje na zakończenie zadania. Przy wybieraniu zadania jako kolejne ustawiana jest wartość nagrody za poprawne wykonanie całego zadania. Wartość ta obliczana jest na podstawie minimalnego czasu przeznaczonego na dane zadanie pomnożonego przez ilość wykonanych zadań oraz liczbę 10. Co daje nam możliwość premiowania zadań, które są są wykonywane głębiej naszego drzewa zadań oraz mają być wykonane szybciej niż inne zdania. Gdy zadanie zostaje wykonane pobierany jest czas wykonania zadania. Jeżeli zadanie zostaje wykonane po przekroczeniu określonego na zadanie czasu użytkownik nie otrzymuje punktów. Natomiast jeżeli użytkownik zmieścił się w czasie, obliczona wcześniej nagroda zostaje pomnożona przez stosunek czasu, który został do czasu przeznaczonego na zadanie. Całość jest zaokrąglana w górę do liczby całkowitej. Obliczona wartość jest dodawana do całkowitej ilości punktów jaką dotychczas zdobył gracz.

Rozdział 4

Implementacja

4.1 Wykorzystane technologie

4.1.1 Języki

4.1.1.1 JavaScript

4.1.1.2 GLSL

4.1.2 Biblioteki

4.1.2.1 Redux

Wymagania dotyczące aplikacji przeglądarkowych stały się na tyle skomplikowane, że interfejs użytkownika jest bardzo złożony i może składać się z wielu elementów. Zarządzanie stanem takich aplikacji jest trudne, ponieważ występuje wiele zależności między komponentami. Może to doprowadzić do sytuacji, w której nie jest jasne co tak naprawdę się dzieje, a znalezienie błędów czy rozszerzenie funkcjonalności staje się zadaniem bardzo czasochłonnym i karkołomnym.

Jednym z rozwiązań jest właśnie skorzystanie z biblioteki Redux dla aplikacji pisanych w języku JavaScript. Głównym jej założeniem jest przejrzysty stan aplikacji, który może zmieniać się tylko w określonych momentach i zawsze w przewidywalny sposób.

Stan jest zdefiniowany jako zwykły obiekt o strukturze drzewa, zawierający wszystkie możliwe informacje, jakie są potrzebne aby jednoznacznie określić i móc odtworzyć identyczną sytuację w aplikacji. Nie może on być modyfikowany, jest tylko do odczytu. Jedynym sposobem na jego zmianę jest wyemitowanie akcji, będącej po prostu zwykłym obiektem zawierającym obowiązkowo pole typ i dowolne inne potrzebne atrybuty. Zadaniem akcji jest przejrzysty opis tego, co się wydarzyło w aplikacji, dzięki czemu dokładnie wiadomo co spowodowało zmianę.

Kluczowym elementem Reduxa są specyficzne funkcje, nazywane w języku angielskim *reducers*, które definiują jak konkretna akcja wpływa na stan. Każda funkcja *reducer* musi spełniać określone wymagania. Jako parametry przyjmuje zawsze tylko i wyłącznie obecny stan aplikacji i wyemitowaną akcję, a zwraca nowy obiekt stanu, w jakim znajduje się aplikacja na skutek wykonanej akcji. Ważne jest także aby *reducer* był przewidywalny i deterministyczny. Oznacza to, że określony stan aplikacji i określona akcja spowodują

powstanie zawsze takiego samego stanu. Dodatkowo taka funkcja nie może mieć żadnych skutków ubocznych. W dużych projektach wskazane jest napisanie kilku takich funkcji, z których każda wpływa tylko na określoną część stanu. Ułatwia to utrzymanie zrozumiałego kodu, który można łatwo rozwijać i modyfikować.

Podsumowując, Redux opiera się na trzech fundamentalnych zasadach:

- cały stan aplikacji jest opisany przez pojedynczy obiekt o strukturze drzewa,
- jedynym sposobem aby zmienić stan aplikacji jest wyemitowanie akcji,
- wpływ danej akcji na sposób przekształcenia stanu określają funkcje zwane *reducers*.

Popularność biblioteki Redux zasłużenie rośnie, ze względu na prostotę i korzyści, jakie daje przestrzeganie opisanych wyżej trzech podstawowych reguł. Zdecydowano się skorzystać z tej biblioteki ponieważ jest łatwa w użyciu i pozwala w wygodny sposób zarządzać stanem aplikacji.

4.1.2.2 React.js

Jako bibliotekę do budowania interfejsu użytkownika wybrano bibliotekę React. Została ona stworzona w 2013 roku przez zespół programistów Facebook'a, aby rozwiązać problem tworzenia dużych aplikacji, w których nieustannie zmienia się to, co należy wyświetlać. Początkowo nie była ona ogólnodostępna, ale aktualnie jest udostępniona na zasadzie otwartego źródła (ang. *open-source*) w serwisie github.com. React pozwala określić jak aplikacja powinna wyglądać w różnych momentach, a samo odświeżanie interfejsu dzieje się automatycznie, gdy zmienią się dane dotyczące komponentów. Dodatkowo ogromną zaletą tej biblioteki jest fakt, że potrafi ona określić co dokładnie uległo zmianie, a w rezultacie ponownie wyrenderowć tylko fragmenty, które rzeczywiście tego wymagają.

React opiera się na koncepcji niezależnych komponentów, nadających się do wielokrotnego wykorzystania, z których można komponować skomplikowane widoki. Są to obiekty JavaScript, reprezentujące elementy HTML, czyli w istocie fragmenty interfejsu użytkownika, mające określoną strukturę i funkcjonalność.

Wykorzystywany jest też wirtualny obiektowy model elementu (ang. *virtual DOM*), czyli obiekt o strukturze drzewa, zbudowany z wcześniej zdefiniowanych komponentów. React obserwuje, czy nie nastąpiły żadne zmiany w nim, a jeżeli tak, to automatycznie modyfikuje rzeczywisty i widoczny dla użytkownika DOM, tak aby odzwierciedlał on stan wirtualnego DOM. Istotne jest, że odświeżeniu podlegają tylko te fragmenty, które uległy zmianie.

Kluczową rolę pełni funkcja *render*, przyjmująca dwa parametry — wirtualny element oraz węzeł DOM, w którym umieszczony zostanie dany element. Dopiero jej wywołanie powoduje, że komponent jest widoczny w przeglądarce. Każdy z takich elementów może mieć właściwości (ang.*props*) oraz przechowywać swój własny wewnętrzny stan (ang. *state*). Różnica między nimi jest taka, że właściwości powinny być stałe, a stan może się zmieniać w czasie. Jego zmiana wywołuje metodę *render()*, odświeżającą komponent.

Zdecydowano się na wykorzystanie biblioteki React.js, ponieważ aplikacje napisane z jej użyciem wymagają mniejszego nakładu pracy aby działać responsywnie i szybko. Dodatkowo biblioteka ta doskonale współpracuje z Reduxem, w którym stan całej aplikacji przechowywany jest w jednym obiekcie nazywanym stanem. Wystarczy odpowiednie jego

fragmenty podpiąć do konkretnych komponentów, aby interfejs automatycznie reagował na jego zmiany.

4.1.2.3 Babylon.js

Babylon.js jest otwartą biblioteką WebGL napisaną w TypeScript i wykorzystywaną przede wszystkim do tworzenia gier wideo w przeglądarkach. Pierwsza odsłona została wydana w 2013 roku. Głównymi twórcami są David Cotuhe oraz David Rousset. Jako, że Babylon.js jest silnikiem 3D, posiada wspaniałe narzędzia do tworzenia, wyświetlania i teksturowania szkieletów w przestrzeni. Przez to, że kierowana jest głównie do twórców gier, posiada wiele dodatkowych funkcji takich jak generowanie krawędzi czy tworzenie obiektu na podstawie mapy wysokości. Ponadto zapewnia natywną detekcję kolizji, grawitację sceny oraz wbudowane kamery, takie jak kamera śledząca, automatycznie podążająca za obiektem.

Rozważana była jeszcze inna biblioteka 3D, a mianowicie Three.js wydana w 2009 roku, również oparta na webGL. Po zapoznaniu i przetestowaniu obu silników wybór padł na Babylon.js. Przyczyniły się do tego przede wszystkim prostota użycia oraz płynność, którą zapewniał w przeciwieństwie do Three.js. Przy renderowaniu tego samego, wybranego obiektu, Babylon.js spisywał się lepiej o kilka klatek. Poza tym biblioteka ta jest dobrze udokumentowana i posiada bogatą bazę poradników, a społeczność wykorzystująca tę bibliotekę jest bardzo liczna i stale rośnie.

4.1.2.4 Pozostałe

4.2 Scenariusz rozgrywki

4.2.1 Struktura

Scenariusz rozgrywki to skierowany graf, którego wierzchołkami są zadania, z których każde składa się z kilku kroków. Zdecydowano się na taką strukturę ze względu na to, że daje ona możliwość sekwencjonowania zadań, a zarazem pozwala na element losowości. Jeżeli z wierzchołka dotyczącego jakiegoś zadania wychodzi kilka krawędzi, to kolejny węzeł, czyli kolejne zadanie, wybierane jest w sposób losowy spośród sąsiadów aktualnego wierzchołka. Dzięki takiej reprezentacji nigdy nie zdarzy się sytuacja, w której następnym zadaniem będzie zadanie nieadekwatne do aktualnego stanu repozytorium.

4.2.1.1 Konstrukcja zadania

Pojedyncze zadanie reprezentowane jest w postaci obiektu, który składa się z następujących pól:

- identyfikator,
- zbiór identyfikatorów możliwych następników,
- tytuł,
- opis,

- minimalny czas,
- domyślny (maksymalny) czas,
- lista kroków.

Opis zadania służy ukazaniu użytkownikowi celu oraz problemu, jaki należy rozwiązać w danym zadaniu. Może to być na przykład rozszerzenie funkcjonalności lub naprawa błędu.

Aby nagradzać użytkownika punktami za szybko wykonane zadanie, zdecydowano się na określenie czasu na wykonanie zadania. Jest on obliczany w momencie dodawania nowego zadania na podstawie ilości dotychczas wykonanych zadań, a także wartości minimalnej oraz maksymalnej czasu zdefiniowanych w danym zadaniu.

Zbiór następników zawiera identyfikatory zadań, które mogą wystąpić po aktualnym. Hierarchia zadań ułożona jest w postaci grafu skierowanego, tak więc wszystkie wierzchołki, do których istnieją krawędzie wychodzące z wierzchołka reprezentującego dane zadanie, są jego następnikami.

Kluczowym elementem zadania jest uporządkowana lista kroków, które należy wykonać, aby rozwiązać zadanie i przejść do kolejnego. Wykonanie kroku polega na wpisaniu odpowiedniej komendy systemu Git. Szczegółowy opis jego konstrukcji w kolejnej sekcji.

4.2.1.2 Konstrukcja kroku

Podobnie jak zadanie, krok jest obiektem składającym się z kilku pól, a mianowicie:

- typ,
- opis,
- lista dozwolonych poleceń,
- etykiety,
- dodatkowe dane, charakterystyczne dla danego typu kroku.

Typ określa jakiej komendy systemu Git dany krok dotyczy. Obsługiwane typy to m.in. "COMMIT" czy "MERGE". Na podstawie typu poprawnie wykonanego kroku określana jest akcja, jaką należy wykonać po stronie grafiki 3D, by odwzorować aktualny stan repozytorium.

W opisie znajduje się krótka informacja, co należy zrobić, aby wykonać krok i przejść do kolejnego. Krok jest uznawany za wykonany, gdy użytkownik wpisze w konsoli jedno z poleceń z listy dozwolonych komend i zatwierdzi je.

Do każdego kroku przypisana jest lista etykiet, określających jakiego zagadnienia dotyczy dany krok. Krok posiada zawsze co najmniej jedną etykietę, wskazującą komendę, której wymaga on do poprawnego wykonania. Gdy użytkownik poprawnie zrealizuje dany krok, to zwiększany jest wskaźnik powodzenia dla wszystkich przypisanych do niego etykiet. Poza wspomnianym wskaźnikiem przechowywana jest także ilość poprawnych wywołań komendy. Informacje te umożliwiają określenie stopnia, w jakim użytkownik opanował dane zagadnienie i nad czym powinien jeszcze popracować. Brane jest to pod uwagę przy losowaniu kolejnych zadań. Dla możliwych następników wyliczana jest waga na podstawie etykiet, jakie mają w swoich krokach.

Obliczenie wagi dla zadania polega na zsumowaniu kwadratów ilorazu wartości 1 oraz wyznaczonego wskaźnika dla kolejnych etykiet w krokach zadania, będącego ilorazem poprawnie wykonanych kroków z całkowitą liczbą prób dla tej etykiety. Dzięki takiemu zabiegowi zwiększane jest prawdopodobieństwo wylosowania kolejnego zadania poruszającego zagadnienia, z którymi użytkownik nie radził sobie zbyt dobrze.

Dodatkowe dane, w zależności od typu kroku, mogą składać się z różnych elementów. Dostarczają informację np. o nazwie wykonanej rewizji czy utworzonej gałęzi. Służą też do określenia jakie pliki powinny zostać dodane lub usunięte z repozytorium. W przypadku kroku dotyczącego polecenia *git checkout* przechowują informację, czy przełączenie powinno być zrealizowane na konkretną gałąź czy rewizję.

4.2.1.3 Format danych

Jak opisano wcześniej, scenariusz rozgrywki to w istocie struktura drzewiasta złożona z zadań, przy czym każde z nich zawiera pewne informacje oraz uporządkowaną listę kroków. Tak złożony obiekt wymagał odpowiedniego i wygodnego formatu do przechowywania danych. Zdecydowano się na format JSON, który doskonale nadaje się do tego celu. Pliki zapisane w tym formacie można bezproblemowo odczytywać oraz modyfikować za pomocą języka JavaScript.

4.2.2 Graficzne narzędzie do definiowania scenariuszy

4.2.2.1 Motywacja

Aplikacja GITar Hero umożliwia przetwarzanie rozbudowanych scenariuszy, których ręczne tworzenie byłoby uciążliwe i czasochłonne, a ponadto podatne na błędy. Dodawanie lub modyfikacja zadań wymagałaby znaczących nakładów pracy. Z tego względu zdecydowano się na korzystanie z narzędzia z graficznym interfejsem użytkownika, za pomocą którego można by było w prosty i wygodny sposób tworzyć grafy zadań oraz zapisywać je w formacie JSON.

4.2.2.2 Poszukiwanie gotowego rozwiązania

Początkowo zamierzano skorzystać z gotowego rozwiązania. Do tego zadania wytypowano aplikację *directed-graph-creator* użytkownika cjrd [tu odnosnik do bibliografii i tam link] udostępnianą jako oprogramowanie o otwartym źródle (ang. *open source*) [1]. Umożliwia ona tworzenie grafów skierowanych oraz ich zapis do formatu JSON. Narzędzie to jest zaimplementowane w języku JavaScript i korzysta z popularnej biblioteki D3.js, która pozwala tworzyć dynamiczne i interaktywne wizualizacje danych w przeglądarkach internetowych.

Jednakże aplikacja, w formie w jakiej została udostępniona, nie wystarczała do zdefiniowania scenariusza rozgrywki. Każdy węzeł grafu mógł przechowywać tylko pojedynczy ciąg znaków. Potrzebne były zatem znaczne modyfikacje, umożliwiające zapisanie w pojedynczym węźle wszystkich niezbędnych informacji, które powinny się znaleźć w zadaniu.

[1] Oprogramowanie udostępnione jest na licencji MIT/X, dzięki czemu istnieje nieograniczone prawo do używania, kopiowania, modyfikowania i rozpowszechniania go. Jedynym wymogiem jest, by we wszystkich wersjach zachowano warunki licencyjne oraz informacje o autorze.

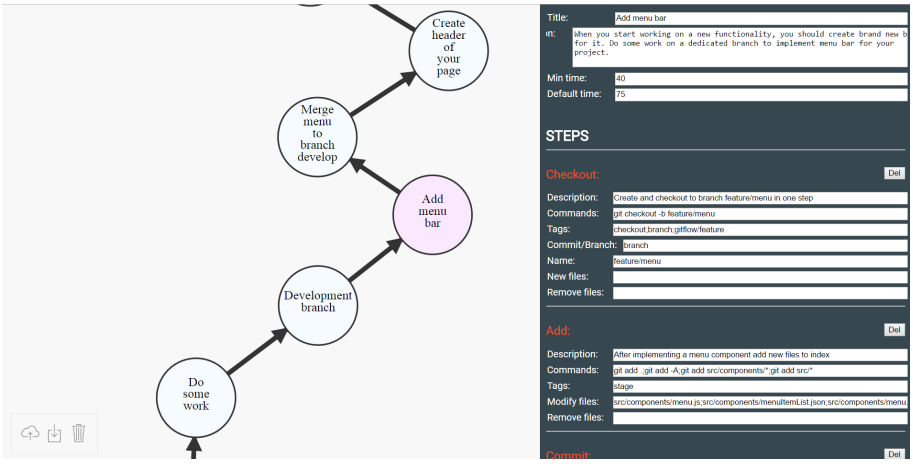
Zdecydowano się zatem stworzyć własne narzędzie do tworzenia scenariusza zadań, bazujące na ogólnodostępnym *direct-graph-creator*.

4.2.2.3 TaskCreator

Postanowiono przerobić wspomnianą wyżej aplikację. Została ona napisana tylko i wyłącznie w języku JavaScript, bez wykorzystania jakichkolwiek platform programistycznych (ang. *frameworks*). Modyfikację utrudniał także fakt, że cały kod zawarty był w jednym pliku. Postanowiono nie modyfikować struktury aplikacji i dalszą część napisać również w czystym języku JavaScript. Dodano jedynie bibliotekę jQuery, która umożliwia łatwiejsze zarządzanie elementami drzewa DOM, czyli obiektowego modelu dokumentu. W związku z tym, że jest to aplikacja internetowa, potrzebny był serwer HTTP. W tym celu użyto środowiska Node.js, wykorzystywanego do tworzenia wysoce skalowalnych aplikacji sieciowych.

Największą modyfikacją było rozszerzenie interfejsu o dodatkowy panel boczny służący do wypełniania pól zadania i definiowania listy kroków. Jest on widoczny na rysunku 4.1 po prawej stronie. Po kliknięciu na węzeł grafu, reprezentujący pojedyncze zadanie, na bocznym panelu zostają wyświetlone wszystkie informacje dotyczącego wybranego elementu. Należą do nich między innymi tytuł, opis oraz czasy wykonania. Oczywiście wszystkie pola są edytowalne. W panelu istnieje również możliwość definiowania listy kroków, które trzeba zrealizować żeby wykonać zadanie. Aby dodać jeden z nich należy wybrać jego typ i kliknąć przycisk "Dodaj", a następnie wypełnić pola opisujące dany krok. Znajdują się tam atrybuty takie jak opis, lista komend spełniająca dany krok, etykiety opisujące wykonane czynności jak również dodatkowe parametry, charakterystyczne dla poszczególnych typów kroków.

Aby dodać nowy węzeł grafu należy przytrzymać klawisz Shift i kliknąć myszką w wybrane miejsce. Żeby edytować wierzchołek należy go wybrać poprzez kliknięcie. Z kolei przytrzymanie klawisza Shift oraz wciśniętego lewego przycisku myszki a następnie przeciągniecie kursora znad jednego węzła na drugi utworzy skierowaną krawędź między nimi. Aby usunąć wybrany wierzchołek grafu bądź jego krawędź, należy go zaznaczyć kliknięciem oraz nacisnąć klawisz Delete.



Rysunek 4.1: Interfejs graficzny narzędzia TaskCreator

Stworzony w ten sposób graf zadań można zapisać do formatu JSON. W tym celu wystarczy kliknąć drugi przycisk w lewym dolnym rogu ekranu. Narzędzie wygeneruje strukturę grafu zrozumiałą dla aplikacji GITar Hero, określi zadanie inicjujące rozgrywkę i zapisze dane do pliku taskGraph.json. Zapisane w ten sposób grafy można wczytać ponownie do aplikacji TaskCreator naciskając pierwszy przycisk w lewym dolnym rogu ekranu i wybierając plik do wczytania.

4.3 Komponenty/elementy(?) aplikacji

4.3.1 Lista zadań

Screen jakiś, jak działa, o implementacji

4.3.2 Konsola

Screen jakiś, jak działa, o implementacji

4.3.3 Pomoc (HelpDrawer)

Screen jakiś, jak działa, o implementacji

4.3.4 Drzewko repo

Screen jakiś, jak działa, o implementacji

4.3.5 Canvas

W projekcie skorzystano z komponentu 'canvas' z html w wersji 5. Pozwala on na wyświetlanie grafiki w przeglądarce. Służy jako kontener dla graficznego silnika 3D, pochodzącego z biblioteki babylon.

4.4 Grafika 3D

Tu będzie sporo, do tego stopnie sporo, że nie wiem jeszcze jak to zaplanować i porozdzielać, względem czego.

Czy np podsekcje takie jak: repo3d - gałęzie, commity, co to są i jak powstają na akcje, o ich teksturze, obramowaniu, w tym o solidExplode ground - co to, jak działa itp particle w tle (wg elementów aplikacji)

Czy może raczej podsekcje wg 'elementów' Babylona: Meshe, SolidParticle, Particle, Materiały, Tekstury, Shadery

4.4.1 Repozytorium 3D

Aby odwzorowywać stan repozytorium na ekranie został stworzony odpowiedni kontroler. Ma on za zadanie reagować na realizowane przez użytkownika komendy i zarządzać elementami grafiki 3D. Nasłuchuje na wykonywane akcje i w zależności od ich typu dodaje, usuwa bądź modyfikuje elementy na scenie. W kontroler ten zawiera listę gałęzi będących w stanie repozytorium.

4.4.2 Branch

Obiekt 3D reprezentujący gałąź systemu git jest tworzony przy pomocy funkcji CreateTube z biblioteki Babylon.js. Funkcja ta generuje siatkę wierzchołków, na podstawie podanej listy punktów określających ścieżkę, w kształcie tuby. Pozwala również na określenie średnicy oraz szczegółowości siatki. Parametry te zostały dobrane w taki sposób, aby wygenerować tubę, której przekrój przypomina koło przy jednoczesnym zachowaniu jak najmniejszej liczby użytych wierzchołków. Jedna gałąź może składać się z dwóch rodzajów siatki. Pierwsza z nich reprezentuje główny człon. Ścieżka użyta do jej wygenerowania zawiera tylko dwa punkty wyznaczające początek i koniec tuby. Dzięki temu można osiągnąć dowolnie długi odcinek nie zwiększając przy tym liczby wierzchołków. SPRAWDZIC I OPISAC PROBLEM Z IN FRUSTRUM. Drugi rodzaj siatki wykorzystywany jest do wygenerowania tuby będącej łącznikiem między gałęziami. Do wyznaczenia ścieżki skorzystano z funkcji pomocniczej generującą krzywą Beziera. Funkcja ta przyjmuje dwa punkty określające początek i koniec ścieżki oraz dwa inne określające jej kształt. SPRAWDZIC KRZYWE BEZIERA.

4.4.3 Commit

Zatwierdzenie zmian w repozytorium git w grafice trójwymiarowej jest przedstawione w postaci kuli umieszczonej na gałęzi, której dotyczy. Funkcja CreateSphere pochodząca z babylon'a buduje siatkę wierzchołków w kształcie sfery o określonej średnicy oraz teselacji. Przy wartości teselacji równej szesnaście powierzchnia obiektu wygląda na wygładzoną. W momencie utworzenia kuli pojawia się nad nią napis z wiadomością przekazywaną przy zatwierdzaniu zmian, którego kolor jest niewiele jaśniejszy niż kolor samego obiektu. Ponadto może również zostać dodany tekst z numerem wersji tzw. tag, znajdujący się nieco powyżej tekstu z wiadomością. Sam tekst jest wyświetlany tylko dla 'commitów' na gałęzi, na której aktualnie znajduje się użytkownik, a jego pojawienie się , czy ukrycie jest animowane.

W kwestii wyglądu wobec kuli zastosowane te same zabiegi co do gałęzi, na której jest umieszczona. Posiada ten sam kolor, materiał oraz obramowanie.

Z obiektem ukazującym zatwierdzenie zmian związane są animacje takie, jak pojawienie się czy zniknięcie z wybuchem cząsteczek. Pierwsza z nich trwa 0.4 sekundy i polega na modyfikowaniu skali z wykorzystaniem funkcji generującej krzywą Beziera, tak by przypominało to sprężanie i rozprężanie, co tworzy ciekawy efekt wizualny. Obiekt potwierdzający zmiany może zostać usunięty, jeżeli użytkownik wykonana akcja zresetowania repozytorium do jakiejś wcześniejszej zmiany. W tym celu napisano efektowną animację z wybuchem cząsteczek stałych, będących niewielkimi kulami. Cząsteczki są w

tym samym kolorze co 'commit'. W przypadku tworzenia stałych cząsteczek w formie wybuchu skorzystano z mechanizmów babylona, który ułatwia pracę z cząsteczkami. Na początku tworzona jest figura z 500 cząsteczek, będących sferami. Następnie definiowana jest funkcja wykonywana dla każdej cząsteczki, w której określa się prędkość oraz kierunek jej rozchodzenia. Cząsteczki rozchodzą się dynamicznie w kształcie kuli, a w każdej klatce zmniejszana jest ich skala, by zanikały w czasie, wszystko to daje efekt eksplozji ciała, co urozmaica doznania wizualne.

4.4.4 Tekst

Tekst w grafice 3D jest ukazany w postaci dwuwymiarowej z opcją billboard w trybie wszystkich osi. Oznacza to, że niezależnie jak ustawiona będzie kamera tekst będzie się odpowiednio obracał w kierunku kamery.

Aby utworzyć tekst w babylon należy zastosować kilka operacji, które w projekcie zebrano w jedną klasę. Na początku tworzona jest dynamiczna tekstura o optymalnych rozmiarach, wyliczonych na podstawie rozmiaru pojedynczego znaku, pomnożonego przez ilość liter w tekście. Następnie w podobny sposób wyliczany jest rozmiar płaszczyzny, na który będzie nakładany materiał wraz z dynamiczną teksturą. W kolejnym kroku powstaje standardowy materiał z babylon'a, w którym jako tekstura rozproszenia oraz nieprzezroczystości ustawiana jest wcześniej utworzona tekstura dynamiczna. Następnie na teksturze dwukrotnie jest rysowany tekst. Najpierw w kolorze białym z przezroczystym tłem, a następnie w kolorze przekazanym jako argument. Zabieg ten jest stosowany, gdyż by istniała możliwość sterowania kanałem alfa, np. przy animacjach, materiał musi mieć ustawioną teksturę nieprzezroczystości, lecz gdy ustawiona jest tylko ta tekstura kolory stają się wyblakłe, ponieważ !!!husaku dopiszmi tu jakos to ładnie, ze to przez to ze jest tym opacity czyli to co czarne wartosci w tych kolorach sa niewidoczne!!!. Dlatego też by tekst był w pełni koloru, rysuje się pod nim ten sam tekst, ale w kolorze białym.

Dla tekstu zdefiniowano dwie animacje pojawienia się i zniknięcia. Przy ich tworzeniu zastosowano gotowe mechanizmy z biblioteki babylon. Aby uzyskać efekt pojawiania się modyfikowana jest wartość przezroczystości materiału płaszczyzny od zera do jeden w ciągu jednej sekundy, natomiast przy znikaniu, następuje mechanizm odwrotny, polegający na zmianie tej samej wartości od jeden do zera. W ten sposób uzyskano efekt 'gaśnięcia'.

Obiekt ten jest wykorzystywany przy ukazywaniu wiadomości i tagów z wersją dotyczących zatwierdzenia zmian, a także przy wyświetlaniu nazw gałęzi.

4.4.5 Kamera

Nieodzownym elementem w projekcie z grafiką 3D jest kamera. W przypadku GitHero kamera jest kamerą śledzącą obiekt i rozszerza standardowe możliwości 'FollowCamera' z babylon'a. Określa się dla niej rotację, odległość oraz wysokość z której ma spoglądać na śledzony obiekt, a także prędkość podążania, czy przyspieszenie. Obiekt, za którym kamera ma podążać to tzw. 'followObject'. Obiekt śledzony wyznacza

Kamera reaguje również na przewijanie myszką. W momencie przewijania w tył kamera zwiększa swoją wysokość, prędkość oraz przyspieszenie. Maksymalna wysokość jest ograniczona tak, by móc przeglądać z perspektywy lotu ptaka na stan repozytorium. Na-

tomiast przewijanie w przód powoduje zmniejszanie wysokości oraz ustalenie pozycji do śledzenia obiektu w odpowiedniej odległości.

Rozdział 5

Podsumowanie

Dodatek A

Przewodnik użytkownika

Praktyczne info dla opornego użytkownika, jak ma korzystać, między innymi że jest opcja scrolla aby oddalić, jakie przyciski do obsługi helpa itp. itd., krótki opis fragmentu rozgrywki co się dzieje po czym i dlaczego i jak ma na to reagować użytkownik i takie tam.