Afro Derby

Dokumentacja powykonawcza

Zespół Afrodyta TEAM, grupa RCC

Paweł Aszklar

Mikołaj Boć

Przemysław Czatrowski

Aleksander Kauch

Michał Kujaszewski

1 Opis projektu

Celem projektu było napisanie gry komputerowej, która obrazuje działanie silnika fizycznego, wykorzystuje dostępne biblioteki graficzne oraz sieciowe. Tematem gry jest wyścig buggy po terenie pustynnym. Zadaniem gracza jest w jak najkrótszym czasie przejechać przez wszystkie punkty kontrolne. Zwycięża pojazd, który jako pierwszy ukończy wyścig. Gra z założenia przeznaczona jest do rozgrywki sieciowej, chociaż został zaimplementowany autopilot, który może zastępować żywego gracza.

Pełny opis funkcjonalności znajduje się w dokumentacji projektowej

2 Różnice w stosunku do dokumentacji projektowej

2.1 Powody wystąpienia różnic

Niestety nie udało się zrealizować całości tego co zostało napisane w pierwotnej wersji dokumentacji projektowej. Powodów było bardzo wiele, ale kilka z nich zasługuje na szczególną uwagę, ponieważ często się o nich zapomina pisząc pobożne życzenia jakimi jest dokumentacja projektowa. Oto lista dręczących nas problemów (szerzej opisane będą w rozdziałach poświęconych poszczególnym członkom zespołu):

2.1.1 Problem z poprawną konfiguracją środowiska – integracja OGRE z Visual Studio była prosta, podobnie z biblioteką RakNET, trochę gorzej wypada w tej klasyfikacji CeGUI, ale zdecydowanie najwięcej problemu było z ODE i OGREODE.

2.1.2 „Bałagan” w repozytorium – zanim udało się skonfigurować środowisko poszczególni członkowie zespołu pracowali na tym co działało – można było implementować moduł sieciowy, interfejs oraz grafikę oddzielnie bez potrzeby łączenia wszystkich wyżej wymienionych bibliotek. Następnie te skrawkowe projekty trzeba było połączyć w jedną całość. Dodatkowym problemem było poprawne umieszczenie bibliotek w repozytorium

2.1.3 Niekompletna dokumentacja – niestety nie wszystkie biblioteki mają kompletną dokumentację. Szczególnie uciążliwy był brak przykładów w CeGUI i OGREODE.

2.1.4 Inne projekty – niestety natłok pracy nad innymi projektami był strasznie duży. Często nie starczało czasu na napisanie czegokolwiek w ciągu tygodnia. Można oczywiście powiedzieć, że tak naprawdę ten problem nazywa się lenistwo, ale w tym wypadku ilość czasu spędzana przy klawiaturze jest naprawdę olbrzymia.

2.2 Najważniejsze odstępstwa

2.2.1 Brak możliwości strzelania – Początkowo pojazdy miały mieć możliwość eliminowania rywali przy pomocy szybkostrzelnego działka przypominającego broń o wdzięcznej nazwie minigun. Niestety priorytetem była implementacja samego wyścigu, więc funkcjonalność ta nie znalazła się, nawet w szczątkowej formie w ostatecznej wersji projektu.

2.2.2 Brak kolizji z elementami terenu – Funkcjonalność ta miała zostać dodana w końcowym etapie projektu, ale okazało się, że biblioteka ODE nie radzi sobie z taką ilością obiektów na raz. Oczywiście jest możliwe napisanie małego silnika, który przekazywałby ODE tylko najbliższe obiekty, ale uniemożliwiłoby to oddanie projektu na czas.

2.2.3 Brak punktów i hali sławy – związane jest to z brakiem możliwości strzelania i brakiem sensownego pomysłu na realizację punktacji. Czas przejazdu pełni rolę punktów wystarczająco dobrze. Hala sławy nie znalazła się w ostatecznej wersji ze względu na brak czasu.

2.2.4 Brak oddzielnych współczynników tarcia dla trasy i pustyni – dobranie współczynników wiązałoby się z koniecznością dodatkowego testowania gry oraz ingerencji w kod – na co oczywiście brakło czasu.

2.2.5 Brak punktów i hali sławy – związane jest to z brakiem możliwości strzelania i brakiem sensownego pomysłu na realizację punktacji. Czas przejazdu pełni rolę punktów wystarczająco dobrze. Hala sławy nie znalazła się w ostatecznej wersji ze względu na brak czasu.

3 Realizacja poszczególnych części aplikacji

Ze względu na to, że praktycznie każda osoba zajmowała się inną częścią gry rozdział ten został podzielony na sprawozdania z pracy poszczególnych członków zespołu. Należy tutaj zaznaczyć, że wszystko poza modułem sieciowym, który jest w zasadzie czarną skrzynką zostało zaprojektowane przez zespół jako całość lub przez dwuosobowe podzespoły. Prezentowany podział dotyczy jedynie samej implementacji projektu.

Każdy członek zespołu osobiście opisał swoje wrażenia z pracy.

3.1 Symulacja, stany gry, sterowanie – Paweł Aszklar

Pierwszym moim zadaniem było takie skonfigurowanie projektu, aby można było korzystać z bibliotek ODE oraz OgreODE. Biblioteki te dostępne były w wersjach skompilowanych, lecz były to starsze wersje z różnych okresów i nie współpracowały one ze sobą poprawnie. Należało więc je samodzielnie skompilować, co było o tyle trudne, że dokumentacja do OgreODE jest szczątkowa, a ja miałem długą przerwę w używaniu C/C++ .

Tworzenie symulacji świata fizycznego na potrzeby gry zacząłem tworzyć w oddzielnym projekcie, na podstawie przykładowych projektów OgreODE (które stanowiły główne, jeśli nie jedyne źródło informacji o użyciu tej biblioteki). Powstały w ten sposób klasy świata, pojazdu i terenu, które zostały przeniesione później do głównego projektu. Wzorując się na przykładach stworzyłem także klasę wczytującą elementy świata gry (klasa wczytująca, oraz klasy terenu były później rozwijane przez Michała).

Model fizyczny ogólnie zgodny jest z założeniami projektowymi. Ze względu na ograniczenia czasowe nie udało się jednak zaimplementować kilku szczegółów, w tym różnicy w wychyleniu kół w zależności od prędkości, czy też różnicy we właściwościach jezdnych w zależności od tego czy pojazd znajduje się na trasie. Brak również mechanizmu uniemożliwiającego pojazdowi wywrócenie się na dach ani powrotu na trasę w miejscu jej opuszczenia.

Nie udało się ponadto włączyć do symulacji zderzeń z przeszkodami. Próba dodania do elementów terenu geometrii odpowiedzialnych za wykrywanie kolizji spowodowała znaczne obniżenie wydajności symulacji do poziomu, gdzie rozgrywka nie mogła być w żaden sposób prowadzona (długie odstępy pomiędzy klatkami poza tym, że fatalnie wpływają na jakość rozgrywki, utrudniały sterowanie i powodowały niestabilność symulacji). Rozwiązaniem mogło być napisanie własnego modułu do silnika fizycznego, który odpowiadałby za znajdowanie znajdujących się blisko siebie obiektów. Z braku czasu oraz bardziej szczegółowego opisu jak taki moduł należałoby napisać zmuszony byłem zrezygnować z tej części funkcjonalności.

Na potrzeby synchronizacji sieciowej stworzyłem klasy przechowujące stan wewnętrzny pojazdu (położenie, prędkość nadwozia, kół itp.). Komentarze na forum przestrzegały przed ręcznym ustawianiem takich wartości w trakcje symulacji, gdyż powodować to miało jej niestabilność. Na szczęście okazało się, że wartości pobrane w trakcie symulacji na jednym komputerze i wstawione na innym nie powodowały żadnych problemów. Rozwiązanie proponowane przez użytkowników forum

polegające na korekcji wartości poprzez dodawanie odpowiednich sił i momentów, które byłoby bardzo trudne do implementacji w celu poprawnej synchronizacji pojazdów okazało się niepotrzebne.

W celu ułatwienia implementacji sterowania w grze napisałem szereg klas, zwanych akcjami, które reprezentować miały czynności jakie wykonać może użytkownik. Powstało też wiele metod ich tworzenia, czy to w odpowiedzi na zdarzenia, czy też przy wczytywaniu całej konfiguracji. Opierając się o dziedziczenie i polimorfizm umożliwiło to napisanie prostego mechanizmu tłumaczącego stan urządzeń wejściowych (takich jak mysz, klawiatura, pad, kierownica, joystick) na stan sterowania pojazdu.

Rozszerzenie mechanizmu sterowania pociągnęło za sobą rozszerzenie obsługi zdarzeń w aplikacji. Początkowo klasa obsługująca zdarzenia dziedziczyła po adekwatnej klasie używanej w różnych przykładowych programach biblioteki Ogre. Okazała się ona niewystarczająca, a dokładna analiza kodu wykazała, że wykonuje one sporo niepotrzebnych operacji (np. utrzymuje własną, nie używaną w naszej aplikacji kamerę, oraz aktualizuje jej położenie). Zastąpiłem ją implementacją części potrzebnej funkcjonalności w naszej klasie odpowiednio dostosowując ją do potrzeb naszej gry. Następnie zmieniłem klasy menadżera stanów oraz same stany, aby poprawnie obsługiwały rozszerzony mechanizm sterowania.

Pod koniec skupiłem się na umożliwieniu przeprowadzenia pełnej rozgrywki pomiędzy wieloma graczami od momentu przejścia z lobby do gry aż do zakończenia wyścigu i powrotu do lobby. Stan gry podzielony został na mniejsze stany odpowiedzialne z poszczególne etapy tego procesu:

- Stan odpowiedzialny za wczytanie zasobów, oraz synchronizację momentu rozpoczęcia wyścigu.
- Stan właściwy gry, w którym gracz steruje swoim własnym pojazdem.
- Stan oczekiwania na zakończenie wyścigu przez innych graczy. Gracz, który ukończył wyścig może w tym czasie obserwować poczynania przeciwników.
- Stan prezentujący wyniki oraz odpowiedzialny za zwolnienie zasobów.

Implementacja polegała głównie na zebraniu i połączeniu ze sobą różnych modułów aplikacji napisanych przez różne osoby. Jako, że części z nich nie można było do końca sprawdzić wcześniej głównym problemem była potrzeba częstej komunikacji z innymi członkami zespołu i naprawianie przez nich różnych błędów.

Widoczne to było zwłaszcza w przypadku wykorzystania modułu sieciowego napisanego przez Mikołaja.

W mechanizmie synchronizacji, zgodnie z projektem, za symulację wszystkich pojazdów odpowiadał serwer, rozsyłając co jakiś czas wyliczone przez siebie stany do pozostałych graczy. Okazało się, że powoduje to skakanie pojazdów innych graczy, co negatywnie wpływało na jakość rozgrywki oraz utrudniało sterowanie. Dlatego razem z Mikołajem wymyśliliśmy modyfikację tego mechanizmu, w którym każda gra odpowiada tylko za symulację pojazdu swojego lokalnego gracza i rozsyła co jakiś czas swój stan do pozostałych. Mimo prostoty zmian, po ich wprowadzeniu przez Mikołaja do modułu sieciowego uzyskaliśmy znaczącą poprawę jakości rozgrywki.

Wszystkie stany związane z właściwą rozgrywką część funkcjonalności miały wspólną (zwłaszcza wyliczanie kolejnych kroków symulacji oraz sporą część obsługi ruchu sieciowego). Wyodrębnienie ich do klasy bazowej pozwoliło znacznie uprościć kod. Ponadto okazało się, że stany te powinny być w stanie aktualizować symulację nawet gdy nie są aktywne. Wymusiło to modyfikację menadżera stanu w taki sposób, aby wszystkie stany na stosie, które tego potrzebują otrzymywały powiadomienia o zdarzeniach związanych z przetwarzaniem kolejnych klatek, lecz tylko aktywny stan miał możliwość reakcji na stan urządzeń wejścia.

3.2 Połączenie sieciowe, synchronizacja – Mikołaj Boć

Zakres obowiązków

- Ustalenie modelu komunikacji sieciowej
- Wybór i ewaluacja odpowiedniej biblioteki zarządzającej warstwą sieciową
- Ustalenie protokołu komunikacji sieciowej
- Szybkie stworzenie prototypu komunikacji sieciowej jako samodzielna aplikacja
- Implementacja modułu komunikacji sieciowej w docelowej aplikacji

Założenia

W projekcie wymagana była rozgrywka sieciowa oraz mechanizm zbierania graczy w tzw. lobby. Do gier z kategorii symulatorów niespecjalnie nadaje się komunikacja połączeniowa TCP, tak więc naturalnym wyborem jest w tym przypadku komunikacja za pomocą datagramów UDP. Ustalono model połączenia zakładający inicjalizowanie gry przez jednego z graczy, zwanego graczem

serwerowym oraz łączeniu się do gry innych graczy, którzy wyrażają chęć do rozgrywki.

W ogólności przebieg komunikacji sieciowej w module miał w założeniu:

- Pozwolić na zebranie graczy w lobby (gracze tylko w stanie serwera lobby mogą się łączyć do gry)
- Każdemu z graczy dać możliwość wysyłania w lobby wiadomości do innych graczy
- Każdemu z graczy dać możliwość przejścia w stan gotowości do rozgrywki; gra może zostać uruchomiona jedynie przez gracza serwerowego i tylko wtedy, kiedy wszyscy gracze są gotowi do rozgrywki
- Umożliwić odliczanie czasu do rozpoczęcia wyścigu (zmiana œwiateł)
- synchronizować rozgrywkę, wysyłając stan wejścia poszczególnych graczy oraz co pewien dość długi okres narzucać graczom klienckim stan obiektów istotnych dla rozgrywki widziany po stronie serwera

Po sprawdzeniu funkcjonalności oraz intuicyjności kilku bibliotek, wybrano bibliotekę RakNet do zarządzania ruchem sieciowym. Taki wybór w założeniu miał pozwolić na skupienie się jedynie na logice przetwarzania sieciowego, minimalizując prace nad kwestiami technicznymi.

Ustalono protokół oparty na stanach modułu sieciowego oraz odpowiednich dla stanów modułu reakcjach na wiadomości wysyłane przez poszczególne aplikacje. Wiadomości miały być serializowane do strumieni bitowych za pomocą odpowiednich struktur biblioteki RakNet. Szczegóły projektowe protokołu są przedstawione w odpowiednim, załączonym dokumencie.

W aplikacji docelowej moduł ma w założeniu być jak najbardziej zautomatyzowany, wymagając od osób użytkujących go minimum operacji. Ten cel ma być osiągnięty poprzez zaimplementowanie singletonowych klas pośredniczących (API modułu) w stanie lobby i dalszej rozgrywki.

Wyniki

Projekt został wykonany w dużej zgodności z założeniami. Wykonano prototyp komunikacji zgodnie z założeniami z dokumentacji modułu sieciowego. Różnice w ostatecznej wersji modułu, będącego częścią aplikacji, wylistowano poniżej:

- W pierwotnej wersji nie zakładano przesyłania kolorów pojazdów graczy; takie dane przesyłane są w ostatecznej wersji modułu po zaakceptowaniu nowego gracza do lobby

- Zmieniono koncepcję synchronizacji, ze względu na niezadowalające wyniki osiągane przez założoną implementację (prawdopodobnie poprzez nieodpowiednią dla tego modelu synchronizację wyświetlania i przetwarzania logiki w innych modułach). W ostatecznej wersji uproszczono mechanizm synchronizacji poprzez przesyłanie stanu pojazdu od graczy odpowiedzialnych za dany pojazd do pozostałych. Osiągnięte wyniki są zdecydowanie bardziej zadowalające.
- Drobne zmiany w komunikatach, polegające na przesyłaniu innych typów danych, niż założone.

Minimalizacja operacji wymaganych od użytkownika modułu może zostać uznana za udaną. Użytkownicy nie muszą ingerować w wewnętrzne sprawy modułu, a jedynie wywoływać sprawnie działające metody singletonowych klas API modułu.

3.3 Autopilot, kamery, skrypty – Przemysław Czatrowski

Osiągnięte cele:

- Zaprojektowanie architektury managera stanów (wraz z Aleksandrem Kauchem)
- Projekt i implementacja managera kamer + 3 typy kamer (swobodna, FPP, TPP)
- Autopilot sterujący samochodem

Na początku powstawania projektu zapoznałem się z działaniem oraz sposobami użycia biblioteki Lua oraz luabind, efektem czego były klasy pozwalające na uruchamianie skryptów w aplikacji oraz uruchamianie metod klas z C++ z poziomu skryptu.

Później zaczął powstawać kod odpowiedzialny za kamery, oparty już o skrypty (założenia były takie, że system kamer oraz autopilot zostaną wydzielone poza główny kod aplikacji).

W międzyczasie stworzyłem strukturę klas managera stanów oraz kilku podstawowych stanów, została ona potem rozszerzona przez Aleksandra i Pawła o elementy obsługujące całą logikę gry.

Około 2 tygodni przed terminem oddania projektu pojawiły się nieoczekiwane trudności – kod odpowiedzialny za wywoływanie skryptów aktualizujących położenia kamer zaczął powodować wyłączanie się aplikacji. Gra zamykała się bez rzucania jakiegokolwiek wyjątku, nie zatrzymywała się też na żadnym z ustawionych w kodzie breakpointów. W związku z tym, że na forach odnośnie Lua oraz Ogre'a nie

znalazłem odpowiedzi na problem, a debugowanie (i tym samym wykrycie błędu) byłoby bardzo trudne i czasochłonne, podjąłem decyzję o zmianie języka skryptowego. Użyłem języka AngelScript, który bardzo dobrze spełnił swoją rolę. Dokumentacja na stronie autora jest raczej mało obszerna, ale potężny dział na forum gamedev.net umożliwił poradzenie sobie z problemami.

Ostatnim elementem był autopilot – system sterowania samochodem bez ingerencji gracza. Autopilot ma dostęp do informacji o trasie (punkty trasy, checkpoint'y) oraz możliwość sterowania samochodem w taki sam sposób jak gracz. Kod pierwszej wersji, mimo widocznych niedoskonałości, spełnił swoją rolę – samochód pomyślnie kończył wyścig. Po dopracowaniu fragmentów odpowiedzialnych za ustalanie toru jazdy, autopilot w większości przypadków wygrywa z autorem kodu ;)

Niestety, z powodu wcześniej opisanych problemów z językiem skryptowym oraz ostatecznym terminie oddania projektu nie udało się wynieść kodu zarządzającego autopilotem do skryptów. Autopilot ma również prosty mechanizm ustalania toru jazdy – kieruje się po jak najkrótszej drodze do następnego niezliczonego checkpoint'a ignorując drogę.

3.4 Lider zespołu, interfejs użytkownika, zarządzanie stanami gry – Aleksander Kauch

Bycie liderem wiązało się w tym wypadku z jako taką kontrolą nad tym co się dzieje w projekcie. Na szczęście zespół sam się potrafił doskonale zorganizować i niewiele wysiłku w koordynację pracy trzeba było włożyć. Poza tym lider zwyczajowo pisze wszystkie dokumentacje i stara się dotrzymywać terminów. Dokumentacje były na czas i były wystarczająco dobre. Ostateczny termin oddania projektu przesunął się niestety o 3 tygodnie… Mimo wszystko w końcu się jednak udało =).

Zaprojektowany i wstępnie zaimplementowany przez Przemka manager stanów spisał się wyśmienicie i rozwijanie go było proste i przyjemne. Po kilku poprawkach Pawła został dostosowany do obsługi zdarzeń OIS, a po dodaniu wielu stanów zaczął pełnić funkcję koordynacji i obsługi logiki całej gry (poza rozgrywką).

Nieco gorzej było z obsługą okien w aplikacji, czyli ze stworzeniem GUI. Do realizacji tego zadania wybraliśmy bibliotekę CEGUI, co okazało się wyborem całkowicie chybionym. Niestety biblioteka jest niewygodna w użyciu, ma spore braki w dokumentacji (szczególnie jeśli chodzi o przykłady) i wiele głupich błędów jak np. brak dostępu do listy elementów w ListBox'ie czy wielokrotne rejestrowanie tych samych funkcji obsługi zdarzeń. W samej aplikacji widać mizerne efekty prób

rozmieszczenia kontrolek w sensownych miejscach niezależnie od rozdzielczości. Okazało się, że nie jest to takie proste jak mogłoby się wydawać.

W pierwszym stadium tworzenia interfejsu powstała wielka klasa managera okienek, która potem rozrosła się strasznie i zaczęła zawierać oprócz okien fragmenty logiki gry. Zostało to zmienione i obecnie całą logiką zajmuje się klasa stanu gry, a manager okienek jedynie ustawia odpowiednie okna w CEGUI i obsługuje tą bibliotekę. Manager okien stał się opakowaniem na CEGUI i doskonale sprawdza się w tej roli - w zasadzie jedyny poważny ślad korzystania z CEGUI jest w tej klasie.

Jeśli chodzi o architekturę aplikacji jako całość bardzo przydatne okazały się wszelkiego rodzaju singletony. Manager okien, manager stanów, manager kamer, klasa świata i wiele innych. W zasadzie wszystko co faktycznie w aplikacji występuje tylko raz (jako jedna instancja) zostało zaimplementowane jako singleton. Być może jest to przesada, ale wszyscy się zgodziliśmy, że jest to logiczny, prosty i bardzo wygodny sposób zarządzaniem grą.

3.5 Grafika, Dodatkowe programowanie – Michał Kujaszewski

Moim zadaniem było zaprojektowanie i realizacja elementów graficznych gry takich jak modele, tekstury, teren, efekty graficzne. W późniejszej fazie projektu pojawiła się potrzeba dodatkowego programowania procedur, które pomogły kolegom z zespołu.

3.5.1 Teren

Na początku projektu zająłem się kwestią terenu. Rozważałem użycie bardziej rozbudowanego menedżera sceny dla terenów jakim jest PagingSceneManager. Okazało się jednak, że jego funkcjonalność w większości nie jest nam potrzebna i wystarczy najprostszy wbudowany w Ogre'a menedżer scen dla terenów. Poza tym PagingSceneManager generował nieokreślony błąd w konfiguracji Debug, a nie było zbyt dużo czasu, by głębiej rozważać temat. Dalszą pracą nad terenem było przygotowanie mapy wysokości (przede wszystkim utworzenie brzegu, co okazało się czasochłonnym zadaniem), dobranie parametrów, utworzenie mapy koloru (przede wszystkim drogi, którą mają podążać pojazdy) i utworzenie lokalizacji dla checkpointów i losowych lokalizacji dla obiektów takich jak głazy i kaktusy.

3.5.2 Grafika

Udało się zaprojektować i zrealizować większość zakładanych na początku modeli. Stworzyłem model buggy i zaprogramowałem do niego procedurę umożliwiającą zmianę jego koloru. Zaprojektowane zostały modele kamieni, checkpointów i kaktusów. Do projektowania modeli użyłem wersji testowej 3ds Max 2009. Odpowiednie tekstury znalazłem w Internecie bądź we własnych zasobach na dysku.

3.5.3 Efekty graficzne

Trzeba z żalem przyznać, że ze względu na brak czasu nie udało się zastosować ciekawszych efektów graficznych. Nieudaną próbą było stworzenie efektu lens flare – niestety musiał zostać wyłączony. Włączyłem tylko podstawowe efekty takie jak sky plane i mgła i dobrałem parametry tak, żeby scena wyglądała poprawnie.

3.5.4 Programowanie dodatkowe

Pod koniec projektu wynikła potrzeba dodatkowego programowania pewnych procedur ku pomocy innym członkom zespołu. Poprawiłem procedurę wczytywania punktów lokalizacji checkpointów i obiektów na scenie, napisałem wyświetlanie dla tych obiektów. Stworzyłem procedury umożliwiające pracę z punktami lokalizacji checkpointów i ich orientację. Oprogramowałem obsługę wyświetlania nicków graczy nad ich pojazdami.