



Uniwersytet  
**ŁÓDZKI**

Inżynierska praca dyplomowa

# Platformowa gra zręcznościowa z wykorzystaniem biblioteki SDL

*Autor:*

Kamil Lolo

*Promotor:*

dr.Krzysztof Podlaski

Wydział Fizyki i informatyki stosowanej

23 kwietnia 2013

# Spis treści

<b>Wstęp</b>	<b>1</b>
<b>1 Wprowadzenie</b>	<b>3</b>
1.1 Czym jest gra komputerowa? . . . . .	3
1.2 Fabuła . . . . .	5
1.3 Grafika . . . . .	6
1.4 Użyte narzędzia . . . . .	8
1.4.1 Kompilator . . . . .	8
1.4.2 Dlaczego aplikacja powstawała w Eclipse? . . . . .	10
1.4.3 Make jako narzędzie do automatyzacji budowania projektu . . . . .	11
1.4.4 Git - system kontroli wersji . . . . .	14
1.5 OpenGL jako silnik grafiki . . . . .	16
1.5.1 Czym jest OpenGL . . . . .	16
1.5.2 Wykorzystanie OpenGL do renderowania grafiki w grze . . . . .	16
1.6 Skrypty w języku Lua . . . . .	17
1.7 Wprowadzenie do biblioteki SDL . . . . .	18
1.8 Obsługa czcionek w SDL . . . . .	20
<b>2 Budowa aplikacji</b>	<b>22</b>
2.1 Główna pętla . . . . .	22
2.2 Mapa w grze . . . . .	23
2.2.1 Mapa kafelkowa . . . . .	23
2.2.2 Edytor mapy . . . . .	25
2.3 Uruchamianie aplikacji . . . . .	26
2.4 Przyjęte konwencje, logowanie i obsługa błędów . . . . .	27
2.5 Jak zbudowana jest aplikacja? . . . . .	29
<b>3 Dokumentacja użytkownika</b>	<b>33</b>
3.1 Sterowanie w grze . . . . .	33
3.2 Menu . . . . .	34
3.3 Instalacja . . . . .	36
<b>Zakończenie</b>	<b>38</b>
<b>Spis ilustracji</b>	<b>39</b>
<b>Bibliografia</b>	<b>41</b>

# Wstęp

Celem tej pracy jest stworzenie gry która będzie działać w systemie Linuks, na który to obecnie jest nieporównywalnie mniej gier niż dla komercyjnego systemu Microsoft Windows. Linuks obecnie najczęściej znajduje zastosowanie jako oprogramowanie serwera, superkomputera bądź jako system wbudowany. Rośnie jednak jego pozycja jako system dla komputerów biurkowych, chodź tutaj nadal uważany jest za system wyłącznie dla tzw. Geeków, czyli maniaków komputerowych z bardzo dużą wiedzą. Pogląd ten stopniowo zmieniany jest przez takie dystrybucje jak Ubuntu. Jest ona równie łatwa w użytkowaniu dla przeciętnego człowieka jak system Windows. Ciągle jednak Linuks nie cieszy się popularnością wśród graczy i twórców gier. Tendencja ta zaczęła się zmieniać w ciągu kilku ostatnich lat, czego dowodem może być wydanie na Linuksa przez firmę Valve Corporation systemu dystrybucji gier „Steam“. Jest to niewątpliwie krok do przodu jeżeli chodzi o zmianę poglądu twórców gier że na Linuksa nie warto wydawać gier. Warto tutaj wspomnieć jeszcze o tym że Valve nie jest mało znaną firmą, bardzo wielu graczy kojarzy ją z grą Counter Strike, w którą przez kilka lat grały setki tysięcy ludzi na całym świecie.

Założenie że aplikacja ma działać natywnie w Linuksie wykluczało użycie narzędzi nie kompatybilnych z tymże systemem, przykładem może być tutaj często wykorzystywany przy tworzeniu gier DirectX firmy Microsoft. Wybór padł natomiast na wieloplatformową bibliotekę Simple Direct Media Layer (w skrócie SDL) której lista docelowych platform jest bardzo długa. Zaczynając od Linuksa, poprzez Windows, Mac OS aż do takich egzotycznych systemów jak Amiga OS. Dodatkową zaletą biblioteki SDL jest fakt że stanowi ona wolne oprogramowanie open source na licencji „zlib“. SDL posiada także kilka dodatkowych bibliotek stanowiących rozszerzenie jej możliwości. W aplikacji zostaną wykorzystane dodatkowe moduły rozszerzające API SDL-a o obsługę dźwięku

oraz czcionek. Nie są to jednak wszystkie dostępne wtyczki do SDL-a, warto wspomnieć też o bibliotece `SDL_Net` dzięki której możliwe jest stworzenie wieloosobowej gry sieciowej. Najważniejszymi możliwościami jakimi dysponuje `SDL` jest utworzenie kontekstu graficznego i obsługa zdarzeń, biblioteka pozwala również za pomocą zestawu funkcji renderować obraz. Rysowanie za pomocą `SDL`-a często okazuje się jednak zbyt wolne, twórcy biblioteki pozwolili obejść ten problem poprzez wykorzystanie do renderowania niskopoziomowej biblioteki `OpenGL`, która również jest kompatybilna z Linuksem.

Następnym założonym celem aplikacji było wykorzystywanie zewnętrznych skryptów (tzw. skryptowanie) które dawałyby możliwość manipulowania pewnymi danymi aplikacji bez potrzeby jej re-kompilacji. Skrypty te będą napisane w języku `Lua`, który został zaimplementowany w `ANSI C`, dzięki czemu zapewnia wysoką wydajność i przenośność na wiele platform. Ogromnym plusem połączenia programu napisanego w `C++` oraz `Lua` jest to że z poziomu aplikacji `C++` można wywoływać funkcje zadeklarowane w skrypcie, a mogą one być zmieniane bez potrzeby rekompilacji całej aplikacji. Funkcje umieszczone w skrypcie są uruchamianie podczas działania aplikacji przez maszynę wirtualną `Lua`. Cały ten mechanizm działa również w drugą stronę z poziomu skryptu `Lua` można wywołać funkcję `C++`, co też zostanie wykorzystane w aplikacji.

Stworzona na potrzeby pracy gra będzie posiadać grafikę 2D, a dedykowanym systemem operacyjnym będzie Linuks, chodź dzięki zastosowaniu wieloplatformowych narzędzi pozostaje możliwość uruchomienia jej w innych systemach np. `Microsoft Windows`, bądź też `Mac OS`. Warto tutaj wspomnieć także o mobilnym systemie `Blackberry` który to wspiera wszystkie technologie które będą użyte w pracy. Daje to możliwość umieszczenia gry w `Black Berry App Word` – markcie z aplikacjami na urządzenia mobilne z tymże systemem. Co wiąże się z możliwością zarobienia pieniędzy na tej grze. Reasumując, w pracy zostanie przedstawiona aplikacja pokazująca możliwości `SDL`-a jako biblioteki do tworzenia gier. Przedstawiona zostanie również możliwość wykorzystania skryptów w języku `Lua` jako narzędzi pozwalającego przenieść część logiki aplikacji poza skompilowany program.

# Rozdział 1

## Wprowadzenie

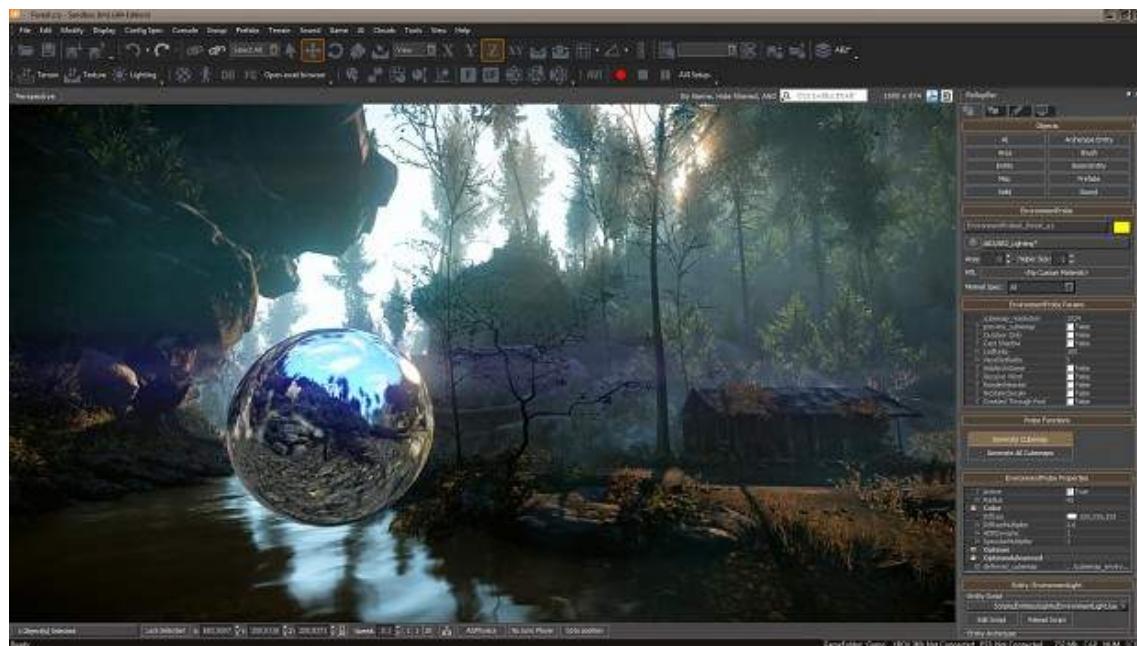
### 1.1 Czym jest gra komputerowa?

Gra komputerowa jest specyficznym rodzajem programu komputerowego łączącego w sobie elementy grafiki i dźwięku. Program ten ma za zadanie dostarczać użytkownikowi, bądź też kilku użytkownikom interaktywnej rozrywki przy komputerze, bądź też innym urządzeniem multimedialnym np. konsoli, tablecie. Gry można sklasyfikować na bardzo wiele gatunków i podgatunków np. gry zręcznościowe, gry z grafiką 3D, strategiczne, FPS-y (ang. first person shooter) itd. Można je podzielić również na to do kogo są kierowane tzn. czy gra jest edukacyjną aplikacją dla dzieci, czy też jest brutalną grą wojenną dla dorosłych.

W czasach kiedy powstawały pierwsze gry (lata 70), oraz w czasach kiedy komputery klasy PC dopiero stawały się popularne, gry były tworzone przez jedną bądź kilka osób. Dziś w produkcji gier bierze udział niejednokrotnie kilkanaście osób, a w tym nie tylko programiści ale także projektanci, reżyserowie, dźwiękowcy, graficy. Proces ten nabiera często ogromnego rozмаłu i przynosi duże pieniądze ze sprzedaży gotowego produktu. Produkcja gry komputerowej jest tak bardzo złożonym procesem że powstają filmy dokumentalne o tym jak się odbywa ten proces. Przykładem może być tutaj film z serii [megafabryki](#) pokazujący proces tworzenia gry Fifa- realistycznego symulatora piłki nożnej.

W dzisiejszych czasach coraz częściej można spotkać się z organizowaniem turniejów w różnych grach komputerowych, a nawet mistrzostwa świata. Współzawodnictwo takie nazywane jest e-sportem, i dla wielu graczy stanowi źródło dochodów. Przykładem może gra League of Legends, w której grają [miliony ludzi na świecie](#) i regularnie odbywają się zawody w tą grę, oraz istnieją ligi w ramach których gracze zmagają się między sobą.

Gry komputerowe w dzisiejszych czasach to nie tylko rozrywka, ale także duże pieniądze. Przykładem może być tutaj gra Call of Duty Modern Warfare 2 która wygenerowała [setki milionów dochodu](#).



RYSUNEK 1.1: Silnik Cry Engine

Omawiając gry komputerowe nie można pominąć zagadnienia silników gier. Silnik gry jest oprogramowaniem które dostarcza twórcy gry gotowych rozwiązań w kwestii np. wykrywania kolizji, oświetlenia, obsługi dźwięku oraz wielu innych elementów. Dzięki zastosowaniu gotowego silnika developer nie traci czasu na pisanie np. modułu do wyświetlanego animacji, tylko korzysta z gotowego rozwiązania skupiając się na fabule, bądź też jakości wyświetlonej grafiki. Takie podejście znacząco przyśpiesza Tworzenie gry, oraz zwiększa jej jakość. Osobny zespół pracuje nad silnikiem, a inny nad stworzeniem na jego podstawie gry. Duże studia tworzą własne silniki, na bazie którego Tworzą kolejne gry, przykładem może być tutaj Cry Engine dający fotorealistyczną grafikę. Został on wykorzystany do stworzenia polskiej produkcji „Sniper Ghost Warrior 2“ korzystającej z 3 wersji tego silnika. Silnik ten dostępny jest do kupienia bądź też do ściągnięcia za

darmo w celach nie komercyjnych ze strony twórców. Silnik posiada środowisko graficzne w którym w sposób wizualny można tworzyć świat dostępny w grze. Co ciekawe taki edytor znajduje zastosowanie nie tylko jako narzędzie do tworzenia gier ale także jako aplikacja do tworzenia realistycznych wizualizacji budynków przez architektów. Minusem większości komercyjnych silników jest jednak brak wsparcia dla systemu Linuks. Ewentualne dostępne dla tego systemu silniki nie dorównują możliwością tym płatnym. Stało się to też przyczyną wybrania grafiki 2D jako podstawy do napisania omawianego projektu, oraz do stworzenia gry od podstaw, bez użycia gotowych rozwiązań.

## 1.2 Fabuła

Omawiany projekt będzie grąręcznościową, celem gracza będzie przebiec bohaterem jak największy dystans. Bohaterem tym będzie astronauta który wylądował na obcej planecie, i musi on zbierać bańki z tlenem żeby się nie udusić. Bańki te uzupełniają poziom życia wskazywany przez pasek życia w lewym górnym rogu. Dodatkowym utrudnieniem będą przeszkody w postaci meteorytów które należy omijać żeby nie pomniejszyć ilości życia.



RYSUNEK 1.2: Postać astronauty

Astronauta podczas gry będzie cały czas biec do przodu, zwalniając nieznacznie w przypadku niskiego poziomu życia. Zwolnienie te będzie odbywać się poprzez zmianę położenia bohatera względem lewego brzegu ekranu. Prędkość przesuwania mapy nie ulega w tym przypadku zmianom. Poziom życia będzie ciągle spadał w regularnych odstępach czasu, natomiast bańki z tlenem będą go zwiększać. Ilość życia które doda jedna zebrana bańka jest większa niż ilość życia zmniejszona w jednorazowo w równych odstępach czasu. Gracz będzie miał możliwość podskakiwania astronautą oraz wznoszenia się nim do góry Sporadycznie na mapie będą się pokazywać bonusy które astronauta będzie mógł zebrać (maksymalnie 3 naraz) i wykorzystać

później do uzupełnienia ilości życia. Taki bonus będzie dawał także nieśmiertelność przez kilka sekund, wtedy to na brzegach ekranu pojawi się charakterystyczna niebieska obwódka. Kiedy gracz zakończy grę, wtedy jego wynik, czyli ilość przebytych metrów zapisywany będzie na liście 10 najlepszych wyników, o ile ilość punktów będzie większa od najniższego wyniku na liście. Kiedy zostanie spełniony ten warunek, gracz zostanie poproszony o podanie swojego imienia. Wyniki te będą zapisywane w osobnym pliku na dysku, tak żeby dane nie zostały stracone po wyłączeniu aplikacji. Listę najlepszych wyników będzie można obejrzeć wybierając z głównego menu pozycje highscore.

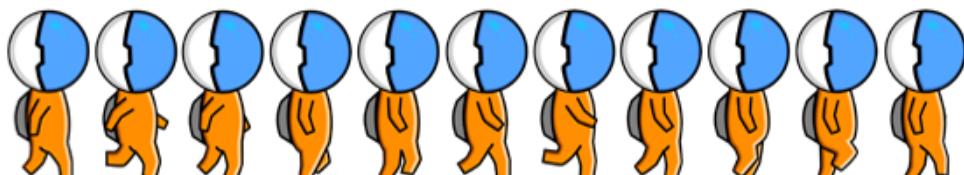


RYSUNEK 1.3: Pasek życia wyświetlany w lewym górnym rogu

Ze względu na fabułę z biegnącym astronautą, oraz osadzenie zdarzeń na obcej planecie, gra została nazwa „Astro Rush“. Tytuł ten nawiązuje również do gry Dino Rush dostępnej dla urządzeń z systemem iOS, która była inspiracją do stworzenia Astro Rush.

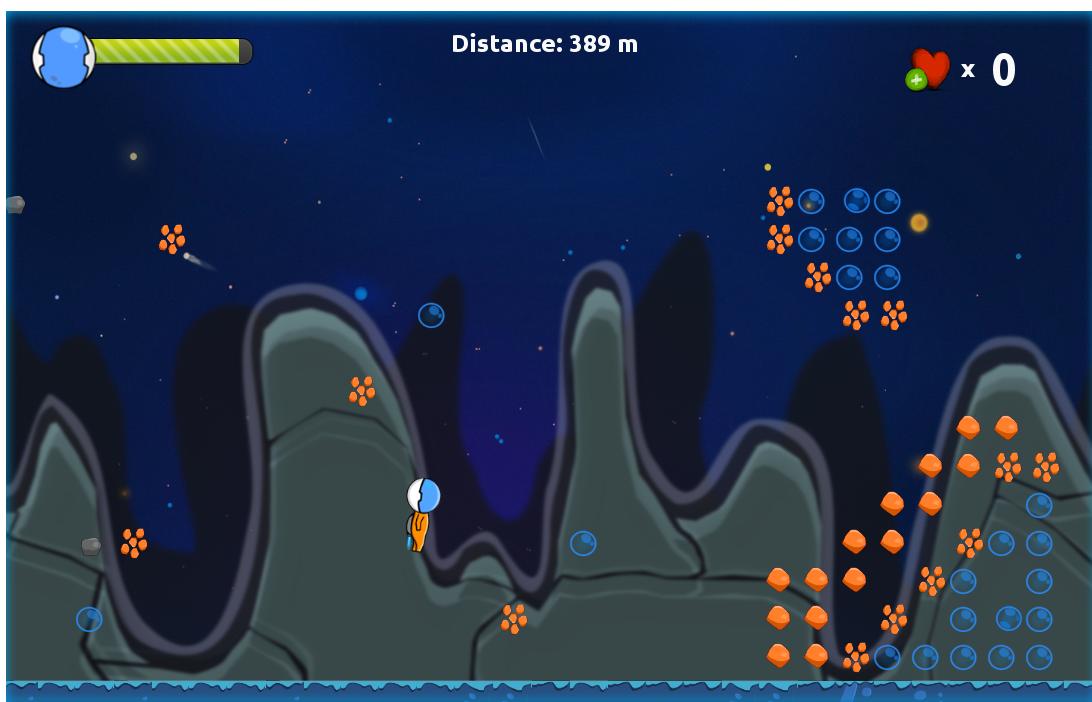
### 1.3 Grafika

Grafika w grze będzie się opierać o technikę tworzenia animacji, gdzie pojedynczy animowany element nazywany jest sprite. Technika ta polega na rysowaniu animacji za pomocą serii klatek które są przechowywane bezpośrednio obok siebie w jednym pliku graficznym. Korzystając z tej techniki można renderować duże obrazki za pomocą serii mniejszych grafik, co pozwala na oszczędzenie miejsca pamięci, na dysku oraz na zwiększenie wydajności aplikacji. W przypadku gry Astro Rush animowany jest bieg astronauty, poszczególne klatki tej animacji narysowanej w programie Inkscape przedstawia poniższy rysunek.

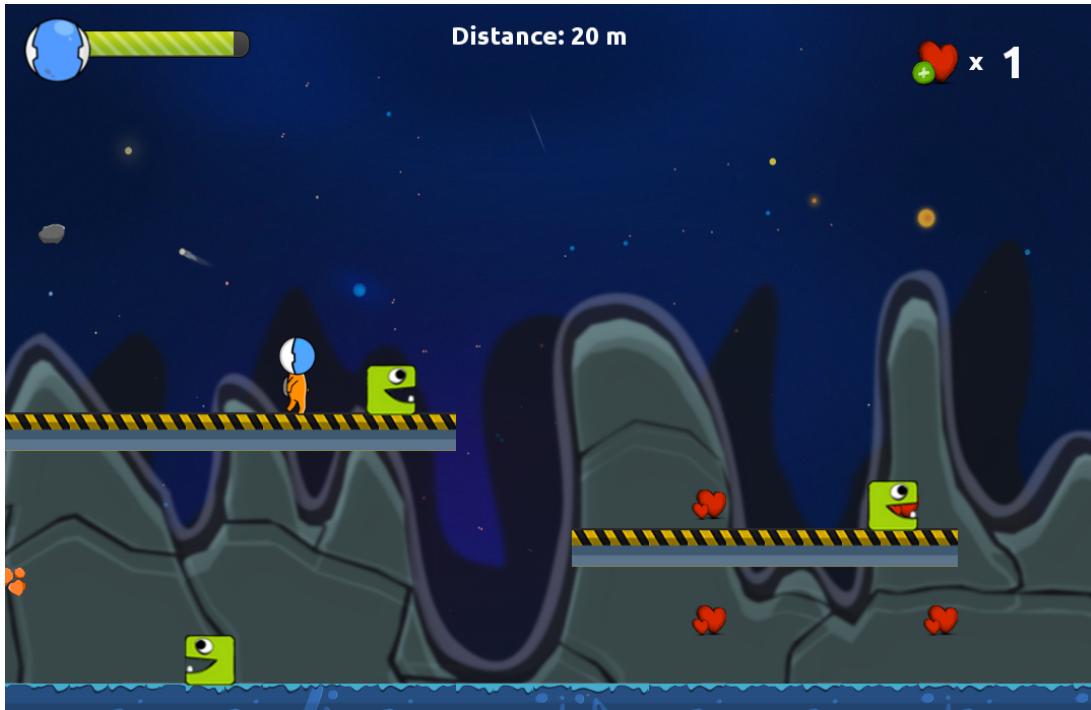


RYSUNEK 1.4: Animacja biegu astronauty

Wszystkie animacje w grze są przechowywane w jednym głównym pliku „atlas.png”. Skąd wyświetlany jest tylko fragment odpowiadający danemu sprite-owi. Po upływie określonego czasu następuje przejście do następnej klatki animacji, czyli zazwyczaj przesuniecie współrzędnej X o szerokość obrazka. W tym algorytmie współrzędna Y nie zmienia się. Cały algorytm wyświetlania animacji opartej przedstawia schemat 1. Warto tutaj wspomnieć o układzie współrzędnych jaki jest używany w bibliotece SDL. Otóż punkt początkowy (0,0) znajduje się w lewym górnym rogu, prawy górny wierzchołek to ( szerokość okna, 0 ), natomiast lewy dolny to : (0, wysokość okna ). Grafika na potrzeby gry została częściowo stworzona w edytorze grafiki wektorowej Inkscape, który oparty jest na licencji GPL i działa pod takimi systemami operacyjnymi jak np. Windows, Linux. Narysowanie części obrazków jako grafiki wektorowej pozwoliło zachować pełną skalowalność w dalszym procesie tworzenia grafiki. Utworzone grafiki wektorowe były składane i poprawiane w Adobe Photoshop – bardzo rozbudowanej aplikacji do obróbki grafiki rastrowej. Photoshop jest aplikacją płatną, jednak istnieje możliwość użycia 30 dniowej wersji Trial, co też zostało zrobione podczas tworzenia gry. W atlasie grafiki znalazły się także ikony z kolekcji „Hand drawn icon set” które autor opublikował w internecie na darmowej licencji.



RYSUNEK 1.5: Przykładowy screen z gry



RYSUNEK 1.6: Przykładowy screen z gry

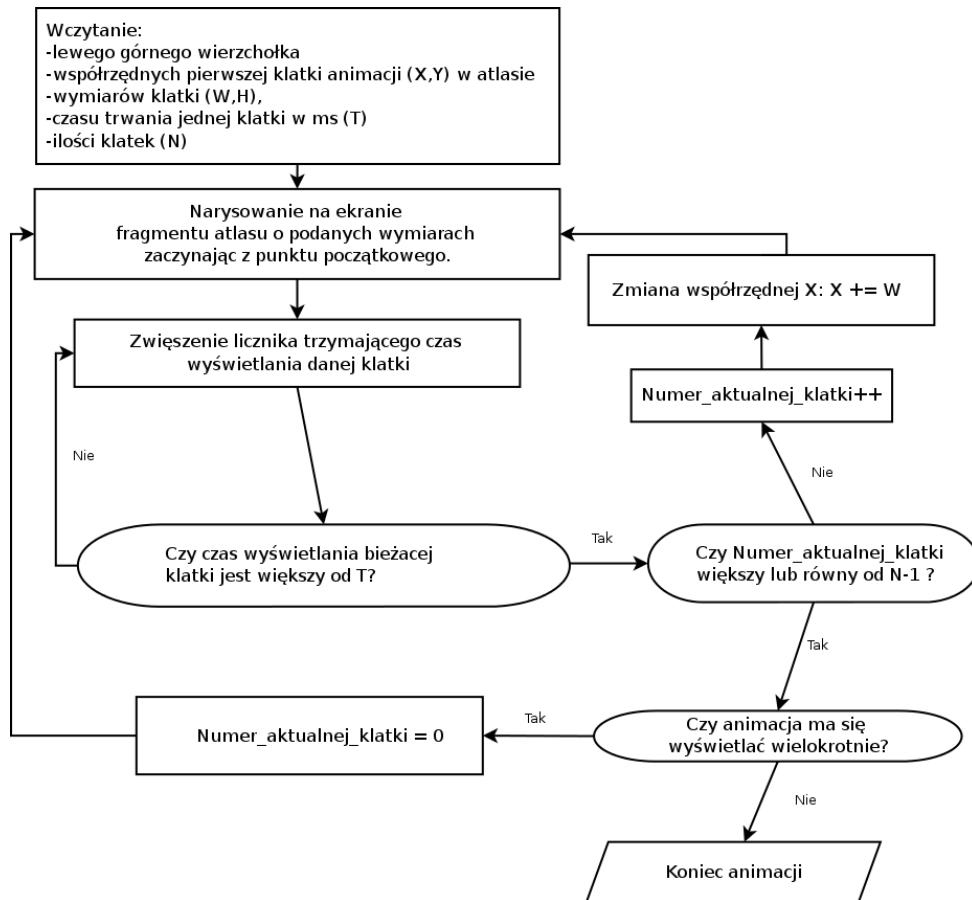
## 1.4 Użyte narzędzia

### 1.4.1 Kompilator

Do zbudowania aplikacji został wykorzystany kompilator g++ dostępny w ramach GNU GCC - zestawu kompilatorów dla popularnych języków. GCC dostępny na licencji GPL pozwala na komplikacje aplikacji napisanej w min. w Java, C/C++, Ada. Dostępny jest na wiele platform i architektur. Ponadto w systemach Linuks jest często zainstalowany domyślnie np. w Gentoo gdzie każda aplikacja jest kompilowana ze źródeł. G++ posiada kompatybilność z językiem C, dzięki czemu nie ma problemu z komplikacją programu korzystającego z biblioteki SDL, która jest napisana właśnie w C. Kompilator ten jest aplikacją konsolową, a skompilowanie zwykłego pliku źródłowego (main.cpp) do pliku wynikowego (main.bin) można wykonać za pomocą polecenia:

```
g++ -c ./main.cpp -o main.bin
```

W przypadku aplikacji składającej się z jednego pliku i nie korzystającego z żadnych bibliotek taka komplikacja wydaje się łatwa. Kiedy natomiast mamy kilka plików do skompilowania, wtedy należy wszystkie je wpisać jako parametr, a dodatkowo trzeba ustawić



RYSUNEK 1.7: Algorytm wyświetlania animacji w oparciu o tzw. „sprite“

flagi linkera, który łączy skompilowane pośrednie pliki w plik wykonywalny. Projekt Astro Rush składa się z około 20 klas, wpisywanie takiego polecenia za każdym razem byłoby bardzo nie wygodne, dlatego też zostało użyte narzędzie make usprawniające budowanie gry. Zostanie ono opisane w jednym z kolejnych rozdziałów.

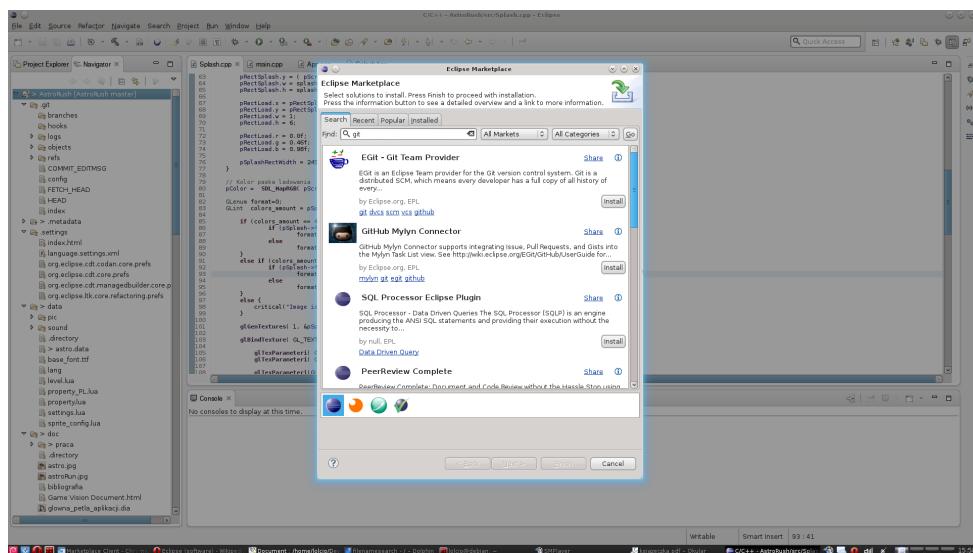
Warto wspomnieć że g++ posiada bardzo wiele przydatnych parametrów, przy komplikacji projektu zostały wykorzystane opcje:

- Wall - włącza wszystkie ostrzeżenia przy budowaniu.
- pedantic włącza wszystkie ostrzeżenia związane ze standardem ISO języka C/C++
- g komplikacja z włączonym debuggowaniem. Do skompilowanej aplikacji dołączane są dane potrzebne do debuggowania poprzez gdb.
- std=c++0x flaga wymusza użycie najnowszego standardu języka C++.
- O0 Określa poziom optymalizacji. Poziom 0 ustawiony domyślnie wyłącza optymalizacje, przyśpieszając tym samym czas komplikacji. Przed udostępnieniem aplikacji użytkownikowi należy włączyć optymalizacje np. za pomocą -O3, gdzie poziom 3 jest najwyższym poziomem optymalizacji.

Dodatkowo potrzebne okazały się następujące flagi linkera: GL lGLU lSDL

### 1.4.2 Dlaczego aplikacja powstawała w Eclipse?

Środowiskiem w którym będzie powstawać aplikacja będzie Eclipse IDE (ang. Integrated Development Environment). Aplikacja została stworzona przez firmę IBM, jednak obecnie stanowi ona darmowe oprogramowanie, dostępne na licencji Eclipse Public License (EPL). Licencja ta bardzo dobrze nadaje się do wykorzystania w celach komercyjnych, dzięki czemu Eclipse jest często platformą wykorzystywaną do tworzenia oprogramowania w korporacjach. Warto wspomnieć o tym że funkcjonalność aplikacji można rozszerzać za pomocą wtyczek. Eclipse posiada nawet market (<http://marketplace.eclipse.org/>) w którym niezależni Twórcy mogą sprzedawać stworzone przez siebie plugin-y. Dostępne są tam zarówno płatne rozszerzenia, jak i darmowe, a samą instalację można przeprowadzić z poziomu uruchomionej aplikacji wybierając w górnym menu Help a następnie Eclipse Marketplace.



RYSUNEK 1.8: Eclipse podczas instalacji wtyczki z Marketplace

Domyślnie w wersji klasycznej Eclipse możliwe jest jedynie tworzenie aplikacji w Javie, jednak instalacja wtyczki CDT, bądź też pobranie przygotowanej już odpowiedniej wersji programu pozwala na wygodne pisanie w języku C++. Konfiguracja tego środowiska do tworzenia gry Astro Rush polegała na pobraniu z oficjalnej strony spakowanej platformy dostosowanej do języka C++, wypakowaniu jej (program wymaga do pracy zainstalowanego JRE, ponieważ napisana jest w Javie), a następnie po uruchomieniu

doinstalowaniu kilku dodatkowych wtyczek które okazały się przydatne podczas tworzenia projektu np. wtyczki zapewniającej wsparcie dla repozytorium Git-a. Środowisko te posiada wbudowaną integrację z wieloma przydatnymi narzędziami, które bardzo upraszczają życie programiście np. auto uzupełnienie, formatowanie kodu. Natomiast wtyczka CDT dostarczyła przy tworzeniu gry Astro Rush taki przydatnych narzędzi jak: -Integracja Eclipse z GNU Debuggerem (w skrócie gdb). Debugger ten stworzony przez Richarda Stallmana i dostępny na licencji GPL jest aplikacją konsolową która umożliwia debugowanie aplikacji C++ w Eclipse. Najbardziej przydatna okazała się tutaj możliwość zatrzymania programu na ustawnionym breakpoincie oraz sprawdzenie stosu wywołań. Można również wykonywać kod programu krok po kroku, szukając przyczyny ewentualnych błędów. Używanie gdb z poziomu konsoli byłoby bardziej uciążliwe. -Integracja z Make (samo narzędzie zostanie omówione w kolejnym podrozdziale) dzięki czemu nie ma potrzeby wpisywania polecenia make w konsoli za każdym razem przy budowaniu, ale wystarczy w oknie Eclipsa wybrać cel budowania i dwa razy kliknąć w niego.

Trzeba również wspomnieć o tym że przy tworzeniu projektu został wykorzystany plugin integrujący Eclipse z programem Valgrind. Ta aplikacja konsolowa jest narzędziem do debugowania pamięci oraz do wykrywania wycieków pamięci. Plugin ten okazał się pomocny wielokrotnie do wyszukania miejsc w aplikacji gdzie dynamicznie przydzielana pamięć nie była zwalniania. Valgrind pozwala nawet znaleźć zmienne które nie są inicjalizowane, co pozwala uniknąć niektórych błędów związanych z tym że zmienne automatyczne zawierają po utworzeniu przypadkowe wartości z pamięci.

W ostatniej fazie tworzenia projektu zostało wykonane profilowanie, czyli dynamiczna analiza aplikacji pozwalająca znaleźć miejsca aplikacji których wywołanie trwa najdłużej w celu ich optymalizacji. Do tego celu został wykorzystany plugin łączący Eclipse z profilerem Perf. Pozwolił on na zebranie i wyświetlenie statystyk odnośnie czasu procesora jaki jest wykorzystywany przez poszczególne funkcje, klasy w programie.

#### **1.4.3 Make jako narzędzie do automatyzacji budowania projektu**

Eclipse jako platforma programistyczna dedykowany jest językowi Java, a kolejne wtyczki pozwalające pisać w innych językach to tylko rozszerzenie tego środowiska Javy.

Eclipse z wtyczką do języka C++ może używać programu Make, który automatyzuje budowanie projektu składającego się z wielu plików. Zostało to wykorzystane w projekcie. Ogromnym plusem takiego rozwiązania jest to żeby zbudować aplikacje u klienta nie potrzeba ściągać Eclipse-a i konfigurować projektu, ale wystarczy aplikacja make, która zajmuje niewiele miejsca na dysku i wystarczy wykonać jedno polecenie żeby zbudować projekt. Rozwiązanie takie jest nie zależne od systemu, ponieważ make działa zarówno w systemie Windows jak i Linux. Dodatkowo bardzo łatwo się go instaluje w większości dystrybucji Linuksa. Dla dystrybucji Debian będzie to polecenie:

```
aptitude install make
```

Po uruchomieniu programu make, szuka on domyślnie, o ile nie podaliśmy w parametrach innej nazwy - pliku Makefile. Plik ten opisuje zależności między plikami źródłowymi, i umożliwia skompilowanie tylko tych plików które uległy zmianie od ostatniego budowania. Zdecydowanie przyśpiesza to prace nad projektem w przypadku kiedy jest potrzebne testowania poprawek na bieżąco.

Domyślne wywołanie make bez żadnych parametrów spowoduje zawsze uruchomienie domyślnego celu budowania: all. Możliwe jest definiowania dowolnej ilości celów budowania w jednym Makefile-u. Dzięki zastosowaniu takiego podejścia nie ma potrzeby edycji Makefilu kiedy chce skompilować aplikacje np. z innymi flagami kompilatora. Wystarczy wtedy dopisać odpowiedni cel budowania i go uruchomić. W projekcie Astro Rush został dodany również cel budowania służący do czyszczenia gry z wszystkich skompilowanych źródeł oraz z linkowanej aplikacji, co okazuje się przydatne kiedy występuje potrzeba przebudowania całego projektu. Tutaj należy wspomnieć o tym że Makefile sprawdza które pliki wymagają ponownej komplikacji, i kompiluje tylko te które uległy zmianie od ostatniego budowania. Takie podejście znacząco skracia czas który programista musi poświecić na budowaniu aplikacji. Dodatkowo make wspiera komplikacje wielowątkowa, dzięki czemu na maszynach z procesorem wielordzeniowym można wykorzystać wszystkie rdzenie, skracając jeszcze bardziej czas budowy programu.

Napisany na potrzeby gry plik „Makefile” pokazuje również że w pliku reguły programu make można definiować swoje zmienne. Taką zmienną może być na przykład ścieżka do kompilatora, która może się zmienić w zależności od platformy. Takimi zmieniennymi mogą być również flagi linkera, kompilatora, oraz lista plików źródłowych z katalogu

src. Tak napisany Makefile okazał się bardzo uniwersalny i dodanie kolejnych nowych plików źródłowych wymaga jedynie dopisanie ich na listę w zmiennej Makefile-a.

Budowanie aplikacji poprzez make bądź też podobne narzędzie o nazwie CMake jest wyjątkowo popularne w systemie Linuks i projektach napisanych w języku C/C++. Make jest nawet wykorzystywany do budowania jądra Linuksa, które składa się z milionów linii kodu (wersja 3.2 to w przybliżeniu 15 mln) oraz setek plików, gdzie oszczędność czasu w przypadku rekompilacji jest bardzo znacząca kiedy kompilowane są tylko te moduły które zostały zmienione. Plik Makefile w grze Astro Rush wygląda następująco:

```
CXX = g++
CFLAGS = -Wall -pedantic -g -std=c++0x -I ./include -O0 -c

# flagi linkera
LIBS = -lGL -lGLU -lSDL -lSDL_mixer -lSDL_ttf -lSDL_image -lluabind -llua5.1

# lista plikow źródłowych do kompilacji
SOURCES = src/main.cpp src/App.cpp src/Property.cpp src/Resource.cpp

# jak maja się nazywać skompilowanie pliki cpp
OBJECTS=$(SOURCES:.cpp=.o)

# nazwa pliku wynikowego
EXECUTABLE = AstroRush.bin

# domyslny cel dla wywolania make bez argumentu, czyli zbudowanie projektu
all: $(SOURCES) $(EXECUTABLE)

# linkowanie aplikacji
$(EXECUTABLE): $(OBJECTS)
@echo "\n ---- Linkowanie ---- "
$(CC) $(OBJECTS) -o $(EXECUTABLE) $(LIBS)

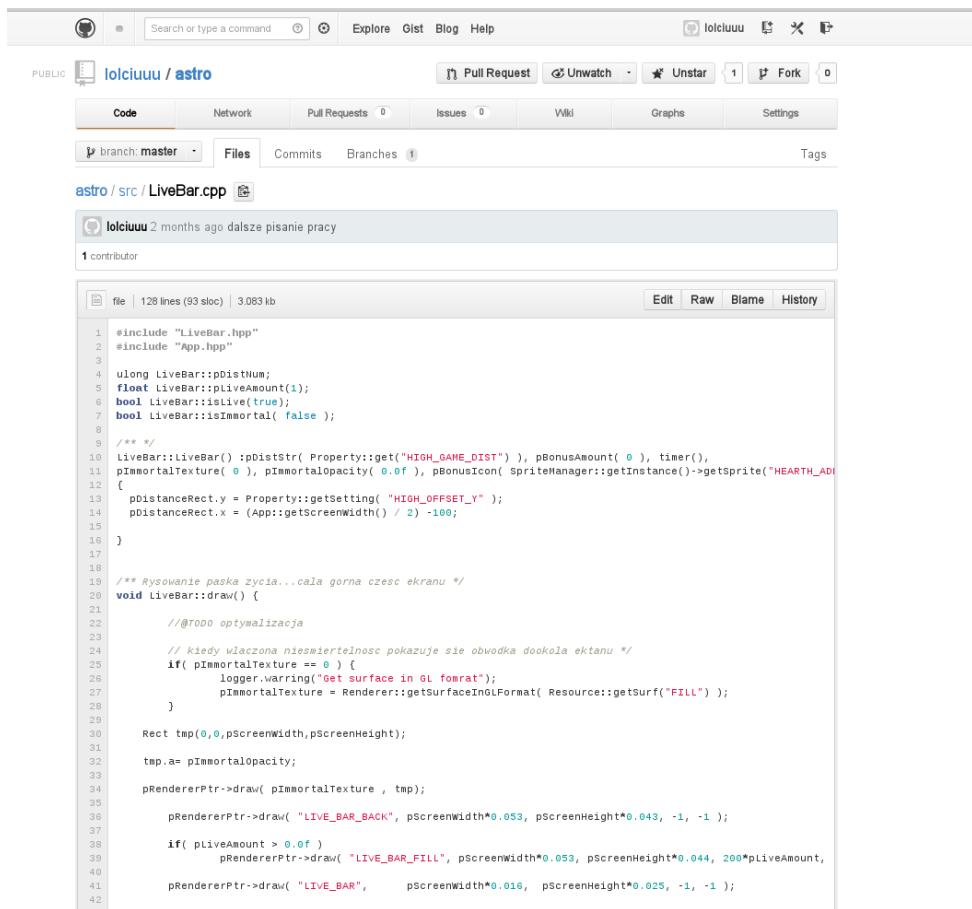
#kompilowanie plikow cpp
.cpp.o:
@$(CXX) $(CFLAGS) $< -o $@

# czyszczenie aplikacji przed zbudowaniem
clean:
rm -rf ./src/*.o
```

```
rm ./AstroRush.bin
```

#### 1.4.4 Git - system kontroli wersji

Projekt Astro Rush nie wydaje się zbyt duży biorąc pod uwagę fakt że nie przekroczył 10 tysięcy linii kodu. Jednak zawsze warto mieć jakąś kopię na repozytorium oraz ewentualnie możliwość poprzez historie zmian przywrócić jakiś fragmentów kodu. Narzędziem które okazało się tutaj pomocne jest rozproszony system kontroli wersji – git. Darmowe oprogramowanie stworzone przez Linusa Torvaldsa do zarządzania kodem jądra Linuksa. Git w założeniu ma być pomocny w pracy zespołowej nad dużymi projektami, jednak w przypadku tak małego projektu jak Astro Rush również okazał się pomocny. Git jest również narzędziem wieloplatformowym, chodź pod systemem Windows jego klient jest on wolniejszy niż na Linuksie.



```

PUBLIC      Search or type a command      Explore Gist Blog Help      lolciuuu      Fork      0
lolciuuu / astro      Pull Request      Unwatch      Star      1      Fork      0
Code      Network      Pull Requests 0      Issues 0      Wiki      Graphs      Settings
branch: master      Files      Commits      Branches 1      Tags
astro / src / LiveBar.cpp      lolciuuu 2 months ago dalsze pisanie pracy
1 contributor
file | 128 lines (93 sloc) | 3.083 kb      Edit      Raw      Blame      History
1 #include "LiveBar.hpp"
2 #include "app.hpp"
3
4 ulong LiveBar::pDistNum;
5 float LiveBar::pLiveAmount(1);
6 bool LiveBar::isLive(true);
7 bool LiveBar::isImmortal( false );
8
9 /**
10 LiveBar::LiveBar() :pDistNum( Property::get("HIGH_GAME_DIST") ), pBonusAmount( 0 ), timer(),
11 pImmortalTexture( 0 ), pImmortalOpacity( 0.0f ), pBonusIcon( SpriteManager::getInstance()->getSprite("HEARTH_ADD")
12 {
13     pDistanceRect.y = Property::getSetting( "HIGH_OFFSET_Y" );
14     pDistanceRect.x = (App::getScreenWidth() / 2) -100;
15 }
16
17
18 /**
19 Rysowanie paska życia... cala gorna czesc ekranu /
20 void LiveBar::draw()
21 {
22     // @TODO optymalizacja
23
24     // Kiedy włączona niesmiertelność pokazuje się obwódkę dokoła ekranu /
25     if( pImmortalTexture == 0 ) {
26         logger.warring("Get surface in GL format");
27         pImmortalTexture = Renderer::getSurfaceInGLFormat( Resource::getSurf("FILL" ) );
28     }
29
30     Rect tmp(0,0,pScreenWidth,pScreenHeight);
31     tmp.ae pImmortalOpacity;
32
33     pRendererPtr->draw( pImmortalTexture , tmp);
34
35     pRendererPtr->draw( "LIVE_BAR_BACK", pScreenWidth*0.053, pScreenHeight*0.043, -1, -1 );
36
37     if( pLiveAmount > 0.0f )
38         pRendererPtr->draw( "LIVE_BAR_FILL", pScreenWidth*0.053, pScreenHeight*0.044, 200*pLiveAmount,
39
40         pRendererPtr->draw( "LIVE_BAR", pScreenWidth*0.016, pScreenHeight*0.025, -1, -1 );
41
42

```

RYSUNEK 1.9: Portal [github.com](https://github.com). Podgląd pliku źródłowego.

Przy tworzeniu aplikacji został wykorzystany serwis hostujący git: <https://github.com>. Portal posiada wiele funkcji wspomagających pracę zespołową nad projektem. Pozwala między innymi na edycję plików źródłowych bezpośrednio na serwerze, przeglądanie i porównywanie zmian w plikach. Dzięki czemu łatwo można sprawdzić co było w poprzedniej wersji pliku, oraz kto wysłał zmiany (ang.commit) na serwer. Github dostarcza statystyk na temat projektu np. wykres jak zmieniała się ilość kodu wraz z kolejnymi wysłanymi wersjami na serwer. Ponadto można tworzyć na serwerze dokumentacje do aplikacji (tzw. Wiki) którą następnie użytkownik może przeglądać przez przeglądarkę, bez potrzeby ściągania na dysk. Hosting dla aplikacji open source jest darmowy, jednak w takim przypadku repozytorium z kodem jest publiczne. Oznacza to że każdy może sobie ściągnąć projekt jednym poleceniem:

```
git clone git://github.com/lolciuuu/astro.git
```

Analogicznie z poziomu konsoli można później wysyłać zmiany na serwer (o ile dany użytkownik został dodany na listę osób które mogą zmieniać pliki na serwerze ), bądź też pobierać zmiany wysłanych przez innych użytkowników. Warto wspomnieć o tym że serwis github mimo tego że powstał dość nie dawno bo w 2008, ma już 2 miliony repozytoriów.

Git w środowisku Linuks jest narzędziem konsolowym (W systemie Debian wystarczy zainstalować pakiet o takie samej nazwie), jednak w ramach ułatwienia podczas tworzenia aplikacji została wykorzystana wtyczka do Eclipse która pozwala w łatwy sposób synchronizować projekt który znajduje się na dysku lokalnie z tym co jest na repozytorium, oraz wysyłanie zmian na serwer. Dzięki wtyczce do tej podczas tworzenia projektu nie było potrzeby wydawać poleceń z poziomu konsoli, wszystko można wykonać z poziomu GUI.

## 1.5 OpenGL jako silnik grafiki

### 1.5.1 Czym jest OpenGL

Open Graphics Library (w skrócie OpenGL) jest niskopoziomową biblioteką graficzną 3D. Kompatybilny jest on z większością liczących się systemów operacyjnych, można z niego korzystać również na urządzeniach mobilnych np. z Androidem. OpenGL jest często wykorzystywany jako podstawowe API przy tworzeniu silników do gier 3D przykładem może być tutaj choćby nawet silnik ID tech znany min. z serii gier Quake. Mimo że OpenGL jest przystosowany do pracy z grafiką trójwymiarową to doskonale można go wykorzystać do grafiki 2D, tak jak to miało miejsce w grze Astro Rush. OpenGL posłużył do wyświetlania tekstur na ekranie, co odbywało się zdecydowanie szybciej niż poprzez funkcje do rysowania z biblioteki SDL. Różnica w szybkości renderowania wynosiła około 20 fps-ów (ang. frame per second ) na laptopie hp550 z procesorem dual core 1.4 ghz. OpenGL dostarcza także takich zaawansowanych elementów jak obsługa cieni oraz oświetlenia, jednak z racji wykorzystania grafiki 2D nie znalazło to zastosowania w projekcie. W bardzo łatwy sposób można połączyć SDL-a i OpenGL-a. W SDL podstawowym elementem graficznym na którym odbywa się rysowanie jest powierzchnia (ang. surface). Podczas inicjowania biblioteki SDL tworzona jest główna powierzchnia na której następnie będzie się odbywać rysowanie ( często też nazywane w grafice 2D „blitowaniem” ). Podczas inicjowania głównej powierzchni ekranu żeby używać do renderowania OpenGL-a wystarczy poprzez funkcję `SDL_SetVideoMode` podać flagę `SDL_OPENGL`. Trzeba jeszcze pamiętać żeby po każdym rysowaniu wywołać funkcję: `SDL_GL_SwapBuffers()` która wysyła bufor ramki do rysowania na ekranie.

### 1.5.2 Wykorzystanie OpenGL do renderowania grafiki w grze

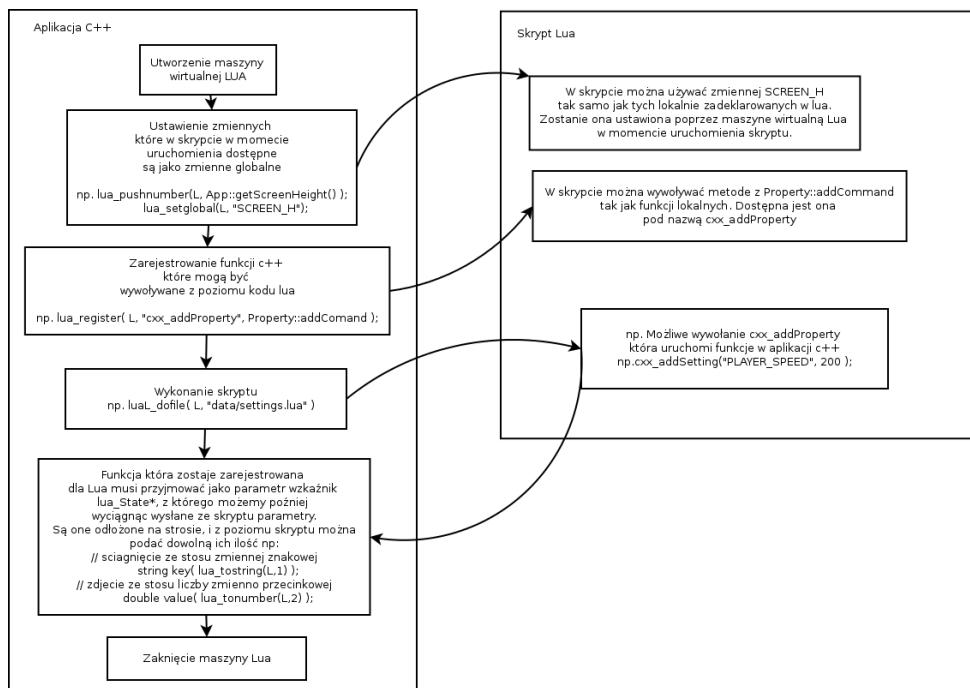
SDL posiada rozszerzenie `SDL_image` które umożliwia obsługę różnych formatów grafiki (w grze zostały wykorzystane pliki graficzne o rozszerzeniach png oraz jpeg). Pozwala one w łatwy sposób wczytać plik z dysku poprzez funkcję `SDL_Surface *IMG_Load(const char *file)` , która zwraca surface z wczytanym obrazkiem. `SDL_Surface` jest strukturą w której znajduje się wskaźnik do pamięci gdzie znajduje się wczytana z dysku grafika. Ten adres ( „`void* pixels`” ) należy przekazać jedynie do OpenGL-a podczas tworzenia

tekstury. Można w ten sposób rysować nie tylko pliki graficzne wczytane z dysku ale również obiekty graficzne utworzone w aplikacji. Taka sytuacja ma miejsce w przypadku renderowania napisów, gdzie jest tworzony surface z napisem, i następnie rysowany za pomocą OpenGL-a. W aplikacji proste prymitywy graficzne jak prostokąty wypełnione kolorem są rysowane również za pomocą API OpenGL-a.

## 1.6 Skrypty w języku Lua

Lua jest lekkim językiem skryptowym zaprojektowanym do rozszerzania możliwości innych aplikacji. Został on zaimplementowany w języku C zgodnie ze standardem ANSI, zapewnia mu to przenośność na wiele platform. Najważniejszym cechami tego języka jest to że jest dynamicznie typowany, oraz obiektowy. Jest on wykorzystywany zarówno do tworzenia rozszerzeń do różnych aplikacji, co często nazywane jest skryptowaniem (ang.scripting) oraz jako samodzielny język, w którym skrypty będą wykonywane poprzez maszynę wirtualną Lua. Samo skryptowanie wiąże się z ideą programowania sterowanego danymi (ang. data driven development). Podejście te zakłada że wszelkie stałe kontrolujące zachowanie programu oraz wybrane elementy logiki powinny być zdefiniowane poza programem. W aplikacji skrypty Lua są wykorzystywane do przechowywanie wszystkich ustawień, oraz obliczania pewnych wartości już w czasie działania aplikacji. Przykładowo rozmiar gracza jest obliczany na poziomie skryptu przy wykorzystaniu ustawionych podczas działania gry zmiennych z rozmiarami ekranu.

Powyższy rysunek pokazuje przykładowy przepływ danych między aplikacją C++ a skryptem Lua. Wykorzystane tu zostało wywołanie funkcji C++ z poziomu skryptu, aczkolwiek w drugą stronę ten mechanizm również działa. To znaczy z poziomu C++ można wykonać funkcje Lua. Lua została wykorzystana do internacjonalizacji aplikacji. Podobnie jak to jest wykorzystywane w aplikacjach webowych w języku Java, gdzie wszystkie komunikaty są przechowywane w plikach properties. Podczas uruchamiania aplikacji zostaje odczytany odpowiedni plik dla danego języka. Cały proces ilustruje rysunek 5. W rezultacie takiej budowy aplikacja może działać z dowolnym językiem. Podczas tworzenia zostały napisane komunikaty zarówno polskie jak i angielskie.



RYSUNEK 1.10: Przepływ danych między aplikacją C++ a skryptem Lua

## 1.7 Wprowadzenie do biblioteki SDL

Rozdział ten będzie zawierał wprowadzenie do tworzenia aplikacji z wykorzystaniem biblioteki SDL, nie będzie tu poruszony temat instalacji tej biblioteki. Zagadnienie te będzie znajdować się w rozdziale dotyczącym komplikacji projektu w systemie Linuks.

Simple Direct Media Layer jest biblioteką ułatwiającą tworzenie gier komputerowych, oraz różnych aplikacji multimedialnych. Umożliwia ona stworzenie okna, oraz zarządzanie nim. Dodatkowo zapewnia obsługę zdarzeń związanych z klawiaturą, myszą oraz joystickiem. Możliwa jest nawet obsługa CD-ROM-u za pomocą tego API. Największą zaletą obok dużej funkcjonalności jest prostota aplikacji pisanej z wykorzystaniem tej biblioteki. Utworzenie kontekstu graficznego wymaga jedynie kilku linijek, a najprostszy program typu "Hello Word" może wyglądać następująco:

```
#include <SDL/SDL.h>

int main(void)
{
    SDL_Init( SDL_INIT_VIDEO );
    SDL_Surface * screen = SDL_SetVideoMode( 800, 600, 32, SDL_SWSURFACE );
    SDL_Flip( screen );
```

```
    SDL_Quit();  
    return 0;  
}
```

Tak napisany kod można skompilować przy pomocy kompilatora g++ z poziomu linuksowej konsoli za pomocą polecenia:

```
g++ ./sdl_simple.cpp -o sdl_simple -lSDL
```

Aplikacja po uruchomieniu utworzy okno które zostanie natychmiast zamknięte. Tak działająca aplikacja jest mało przydatna, dlatego też konieczna jest pętla w której będzie odbywać się praca całego programu. Bardzo ważnym elementem takiej pętli jest obsługa zdarzeń, którą można zrealizować poprzez użycie funkcji `SDL_PollEvent`, przyjmującej jako parametr adres do struktury `SDL_Event`, którą to wypełnia informacjami o zdarzeniach które miały miejsce. Następnie za pomocą warunków logicznych należy sprawdzić jakie zdarzenia miały miejsce. np. warunek `if(pEvent.type == SDL_KEYDOWN && pEvent.key.keysym.sym == SDLK_ESCAPE)` udzieli informacji czy miało miejsce naciśnięcie klawisza escape. Analogicznie ze struktury `SDL_Event` możemy wyciągnąć informacje o dowolnym klawiszku, naciśnięciu myszy, itp.

SDL dostarcza również obsługę wielowątkowości oraz timerów. Dzięki czemu nie potrzebna była w projekcie żadna dodatkowa biblioteka która umożliwiała by utworzenie timera. Zadaniem timera jest uruchamianie pewnego zadania co stały określony czas, w przypadku biblioteki SDL jest wywoływana funkcja do której wskaźnik przekazujemy do funkcji `SDL_AddTimer`, drugim ważnymi parametrem tej funkcji jest czas milisekundach co ile mas się wykonać ta funkcja. Funkcja która rejestrujemy do wykonywania przez timer powinna (@TODO typ) oraz funkcja `SDL_AddTimer` zwróci nadane utworzonego timera w postaci struktury `SDL_TimerID`. W taki właśnie sposób zostały wykorzystane timery w Astro Rush do np wyłączania nieśmiertelności bohatera który zebrał i wykorzystał bonus. Żeby cały mechanizm timerów działał potrzebne jest zainicjowanie go przy tworzeniu okna poprzez użycie flagi `SDL_INIT_TIMER`.

```
SDL_INIT_TIMER SDL_TimerID  
onWithEnemy = false;
```

```
timer = SDL_AddTimer( DISABLE_COLLISION_WITH_ENEMY_TIME,
    enableEnemyDetect_callbackTimer, this );
static Uint32 enableEnemyDetect_callbackTimer(Uint32 interval, void *param);
```

## 1.8 Obsługa czcionek w SDL

W projekcie została wykorzystana biblioteka `SDL_ttf` pozwalająca używać w aplikacji czcionek w formacie True Type. Format ten stworzony przez firmę Apple przechowuje kształty poszczególnych liter jako krzywe Beziera, i jest on obsługiwany przez większość platform. Na Linuksie jest on bardzo powszechnym formatem do obsługi czcionek, dodatkowym plusem jest ogromna ilość czcionek na darmowych licencjach. W grze została wykorzystana czcionka "Ubuntu" udostępniona za darmo, i będąca domyślną czcionką w dystrybucji Linuksa o tej samej nazwie. Plik z taką czcionką (standardowo o rozszerzeniu `*.ttf`) jest wczytywany podczas uruchamiania aplikacji, następnie poprzez wywołania funkcji z biblioteki `SDL_ttf` np. `TTF_RenderUTF_Blended` zostaje utworzona powierzchnia na której narysowany jest napis o podanej treści, kolorze oraz rozmiarze.

Niestety powierzchnia taka jest zwracana jako wskaźnik na strukturę `SDL_Surface`, przez co konieczna jest konwersja na format obsługiwany przez OpenGL-a. Niedogodność taka nie występowałaby gdyby do renderowania było wykorzystywane API biblioteki SDL. Identyczny problem występuje również przy renderowaniu innych elementów graficznych które są ładowane z dysku i zwracane jako `SDL_Surface`\* (Wczytywanie takie realizowane jest poprzez kolejną bibliotekę będącą uzupełnieniem SDL-a: `SDL_image`. Służy ona do wczytywania plików graficznych w takich formatach jak np. JPEG, PNG, TIFF. W aplikacji wykorzystana jest tylko jedna funkcja z tej biblioteki stąd też nie będzie ona szerzej omawiana).

Sama konwersja `SDL_Surface`\* na `GLuint` to wygenerowanie tekstury w standardowy dla OpenGL-a sposób, wykorzystując przy tym pole `pixels` ze struktury `SDL_Surface`, które jest adresem pod którym przechowywane są poszczególne piksele obrazka. W uproszczeniu funkcja realizująca taką konwersję w grze wygląda następująco:

```
void RendererGL::create_gl(SDL_Surface * surf, GLuint * tex )
{
```

```
/** ...tutaj określenie ilości kolorów i formatu */

glGenTextures( 1, tex );
 glBindTexture( GL_TEXTURE_2D, *tex );

/** ...tutaj ustawienia parametrów tekstury */

glTexImage2D( GL_TEXTURE_2D, 0, colors_amount,
    surf->w, surf->h, 0, format,
    GL_UNSIGNED_BYTE, surf->pixels );
}
```

Warto wspomnieć że biblioteka `SDL_ttf` pozwala renderować napisy z polskimi znakami, o ile takie występują w wczytanej czcionce. Ponadto `SDL_ttf` dostępny jest podobnie jak `SDL` na darmowej licencji zlib, i jest wieloplatformowy jak wszystkie wtyczki do `SDL-a`. W niektórych dystrybucjach zainstalowanie tej biblioteki, oraz innych wspomnianych bibliotek rozszerzających `SDL-a` sprowadza się do wykonania jednego polecenia - zainstalowania pakietu z repozytorium. Dla dystrybucji Debian oraz jego pochodnych będzie to polecenie:

```
apt-get install libsdl-ttf2.0-dev
```

W przypadku innych dystrybucji kod źródłowy biblioteki można pobrać ze strony [www.libsdl.org](http://www.libsdl.org). Na stronie tej dostępne są również prekompilowane pakiety dla dystrybucji opartych na pakietach rpm.

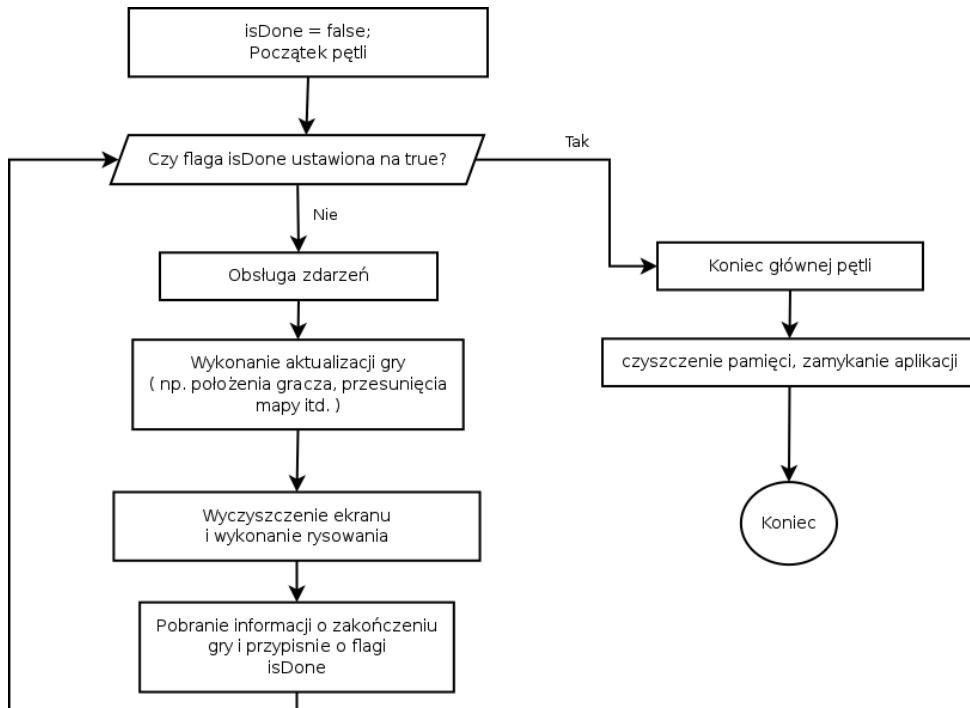
# Rozdział 2

## Budowa aplikacji

### 2.1 Główna pętla

W grach komputerowych często wykorzystywane jest tzw. programowanie sterowane zdarzeniami. Polega ono na umieszczeniu w aplikacji głównej pętli, w której to cyklicznie będzie się odbywać obsługa zdarzeń (np. naciśnięcie klawisza ), aktualizacja gry oraz rysowanie. Sama kolejność tych elementów nie odgrywa większej roli, warto natomiast zwrócić uwagę na to że po zatrzymaniu pętli następuje przygotowanie aplikacji do wyłączenia. W przypadku Astro Rush po wyjściu z głównej pętli zatrzymywany jest kontekst graficzny SDL-a, zwalniane są wszystkie zajęte zasoby, i następuje wyłączenie gry. Pętla taka najczęściej implementowana jest jako while, którego zakończeniem steruje flaga wyjścia. Przy każdym obiegu pętli wartość tej flagi wyciągana jest z klasy Game, która to decyduje kiedy należy zakończyć działanie aplikacji.

Pętla stanowi najważniejszy element większości gier, to od niej zależy czy gra będzie działać tak samo na urządzeniach różniących się wydajnością. W projekcie pętla jest dość prosta i oprócz typowych elementów typu aktualizacja, rysowanie, obsługa zdarzeń, uwzględnia jedynie sytuacje w której całość obliczeń i rysowań odbywa się zbyt szybko i należy wykonać opóźnienie. Taki problem może się pojawić na szybszych urządzeniach na których gra działała by zbyt szybko. W przypadku bardziej złożonej aplikacji można także uwzględnić sytuacje odwrotną, kiedy to ostatnia aktualizacja stanu gry odbywała się zbyt wolno i w następnym obiegu pętli należy wykonać aktualizacje kilkukrotnie żeby



RYSUNEK 2.1: Schemat działania głównej pętli

zapobiec braku płynności w renderowanym obrazie. Taka sytuacja jednak w grze Astro Rush nie powinna mieć miejsca z racji tego iż jest to gra z grafiką dwuwymiarowa i występują w niej proste obliczenia matematyczne.

## 2.2 Mapa w grze

### 2.2.1 Mapa kafelkowa

Mapa kafelkowa (ang. tiled map) jest jedną z podstawowych technik przy tworzeniu gier z grafiką dwuwymiarową. Technika ta polega na podziale świata dostępnego w grze na fragmenty (tzw. kafelki ) o tych samych rozmiarach. Najczęściej są to kwadraty, którym przypisujemy odpowiednie identyfikatory grafik. Tak utworzona mapa przechowywana jest w postaci dwuwymiarowej macierzy w osobnym pliku na dysku, i jest wczytywana podczas startu aplikacji w osobnym wątku. Macierz składająca się wyłącznie z cyfr (typu short żeby dodatkowo oszczędzić pamięć) zajmuje o wiele mniej pamięci w przeciwieństwie do rozwiązania w którym z dysku wczytywana jest cała mapa w postaci jednej grafiki. Na podstawie tej macierzy rysowana jest mapa widoczna na ekranie.

Z racji tego że gracz ciągle wędruje przez mapę, ta cały czas jest przesuwana, a dokładniej to inkrementowany jest indeks kolumny w macierzy kafelków od której zaczynamy rysowanie. Kolumna o takim indeksie rysowana jest na ekranie jako pierwsza z lewej strony, następnie rysowane są obok (po prawej stronie) kolejne kolumny aż do momentu w którym mapa pokrywa cały ekran. Kolumna od której zaczyna się rysować od lewego brzegu ekranu przesunięta jest w lewo o pewien offset, który zwiększany jest podczas biegu gracza do przodu. Offset ten sprawia że współrzędna na osi X skrajnej kolumny zostaje przesunięta w lewo poza ekran, tak że widoczny jest tylko fragment kolumny na ekranie. Przejście do następnej kolumny następuje w momencie kiedy skrajna kolumna znajduje się całkiem poza ekranem. Dzięki zastosowaniu takiego algorytmu następuje płynne przesuwanie mapy, bez widocznych przeskoków pomiędzy kolejnymi kolumnami.



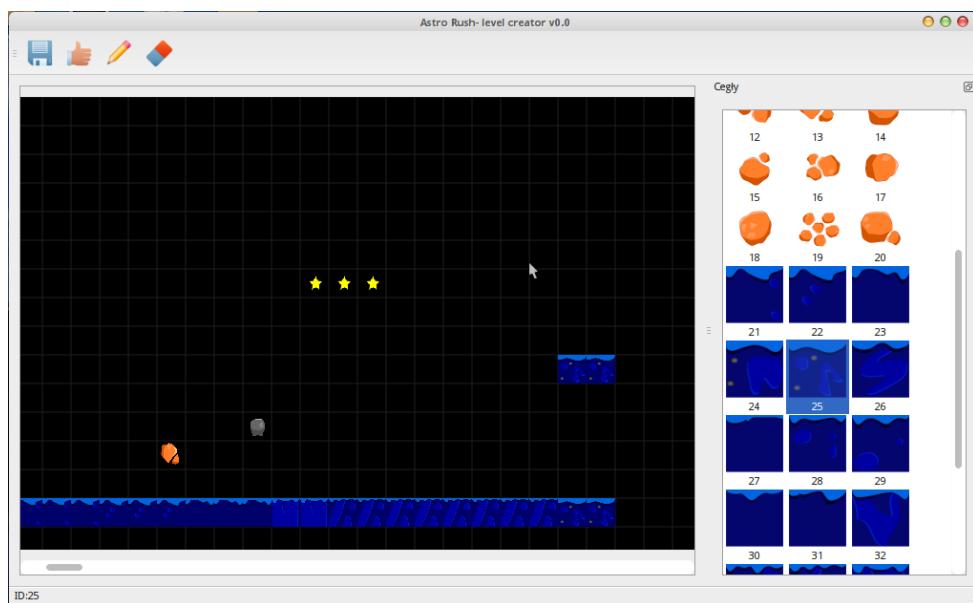
RYSUNEK 2.2: Przewijane tło wyświetlane w grze

Warto wspomnieć o tle w grze. Tłem mapy jest grafika wykonana w programie Inkscape, przedstawiająca kosmos. Wymiary tej grafiki to 1920 px na 1080 px, w przypadku uruchomienia gry na mniejszej rozdzielczości obraz zostaje przeskalowany, natomiast większa rozdzielcość ( 1920x1080 to popularnie nazywa rozdzielcość HD ) nie jest dopuszczalna w aplikacji i program zakończy się z informacją że taka rozdzielcość nie jest wspierana. Ograniczenie takie powstało w celu zapewnienia jednolitego wyglądu na różnych wielkościach monitorów. Dolną graniczną rozdzielczości z jaką może pracować program jest 800 na 600. Wymiary te są zaszyte w kodzie źródłowym jako stałe, przez co nie mogą być modyfikowane przez potencjalnego użytkownika któremu zostanie dostarczona aplikacja. W celu nadania większej atrakcyjności wyświetlane jest drugie tło składające się ze skał obcej planety. Tło te przesuwa się w tym samym kierunku co mapa, jednak z inną prędkością żeby nadać większej atrakcyjności wyświetlaniu obrazowi. Szerokość grafiki to ponad 2500 px, jednak jest to zbyt mało żeby skały mogły się przewijać przez cały czas trwania rozrywki, dlatego też wyświetlanie tego tła ulega zapętleniu, i kiedy na ekranie pojawia się brzeg grafiki, wtedy zaraz za nim pokazuje się następny obrazek

tła rysowany od początku. Dzięki czemu zachowana jest ciągłość i płynność wyświetlanej grafiki.

## 2.2.2 Edytor mapy

Opisana w poprzednim podrozdziale macierz kafelków w grze Astro Rush ma wymiary: 30000 x 15. Stąd też pojawił się problem edycji tak dużej ilości danych. Zmiana poszczególnych wpisów ręcznie nie wchodziła w grę, dlatego też powstała dodatkowa aplikacja do edycji mapy. W wizualny sposób, z wykorzystaniem jedynie myszy można w niej stworzyć w kilkanaście minut całą mapę, rozmieszczając na niej dostępne rodzaje kafelków. Edytor umożliwia także wczytanie stworzonej wcześniej mapy i jej edycje. Aplikacja została napisana z wykorzystaniem biblioteki Qt udostępnionej na licencji LGPL. Biblioteka ta jest zestawem przenośnych narzędzi do tworzenia między innymi interfejsu użytkownika, obsługi sieci, grafiki trójwymiarowej (OpenGL), plików i wielu innych.



RYSUNEK 2.3: Edytor mapy podczas pracy

Edytor mapy wyświetla całą planszę w postaci siatki na której naniesione są kafelki. Poprzez kliknięcie w daną komórkę możemy zmienić rodzaj kafelka który w danym miejscu ma się wyświetlić, bądź też wyczyścić daną komórkę. Do pliku zapisywane są tylko numery odpowiadającym poszczególnym kafelkom, na bazie których gra rozpoznaje jaką grafikę w danym miejscu wstawić. Numeracja kafelków rozpoczyna się od 0, natomiast wartość -1 oznacza że w danym miejscu nie ma kafelka i taki fragment nie jest rysowany. Edytor jest

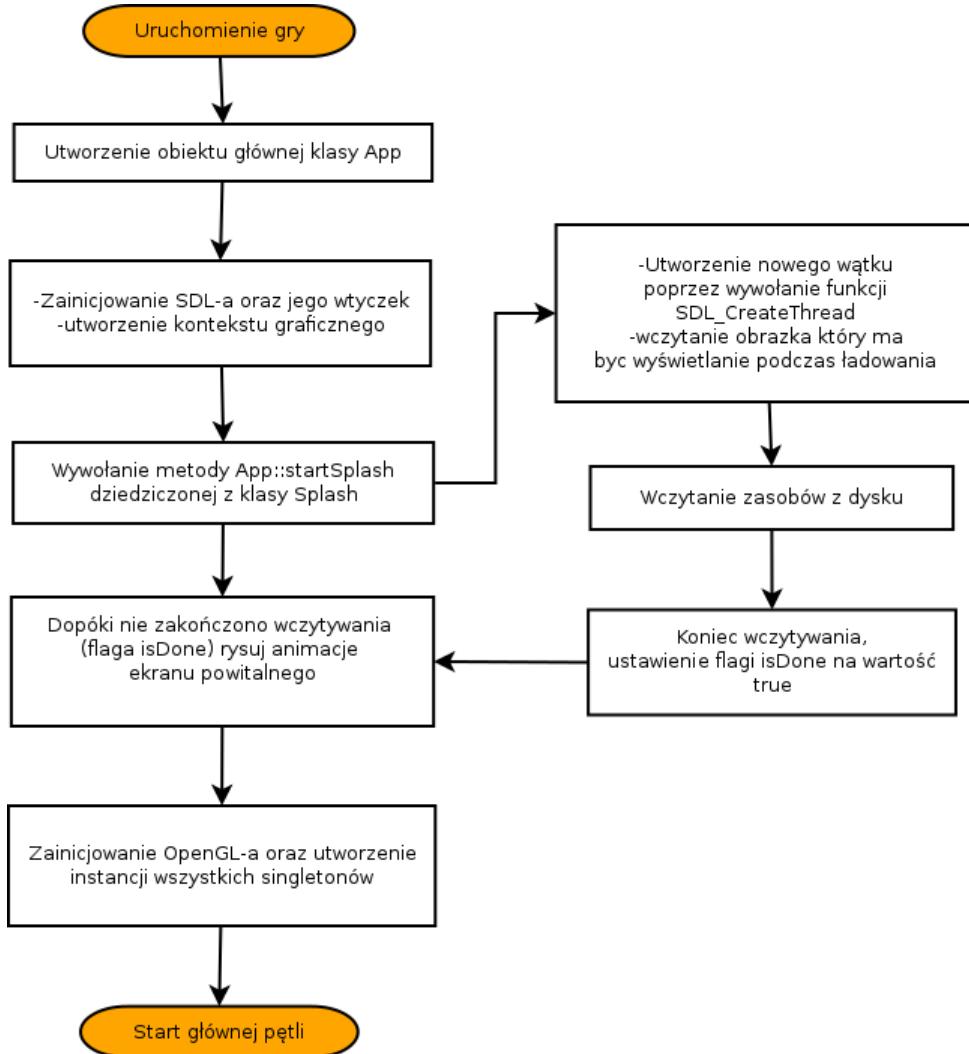
aplikacją bardzo prostą, wszelkie jego modyfikacje np. dodanie nowego rodzaju kafelka wymaga ręcznych zmian w kodzie programu, jednak na potrzeby pracy takie rozwiązanie okazało się wystarczające.

## 2.3 Uruchamianie aplikacji

Aplikacje takie jak gry wymagają do pracy zewnętrznych zasobów które są przechowywane na dysku. Mogą to być grafiki, dźwięki, czcionki. Wraz ze wzrostem ilości materiałów jakie muszą być załadowane do aplikacji przy jej starcie rośnie też czas oczekiwania gracza na to aż aplikacja będzie gotowa do pracy. Dlatego też wczytywanie zasobów, oraz inne czynności które trwają długo są wykonywane podczas uruchamiania, a użytkownikowi prezentowany jest w tym czasie ekran powitalny (ang. Splash screen). W grze Astro Rush taki ekran również jest prezentowany. Pokazuje się na nim również pasek postępu obrazujący ile czasu pozostało jeszcze do uruchomienia właściwej części aplikacji. Żeby samo rysowanie ekranu powitalnego odbywało się płynnie wczytywanie danych z dysku odbywa się w osobnym wątku.

Wielowątkowość w projekcie jest możliwa dzięki wykorzystaniu specjalnej funkcji z biblioteki SDL. Funkcja `SDL_CreateThread` przyjmująca jako argument wskaźnik do funkcji tworzy nowy wątek w którym zostaje uruchomiona ta funkcja. W funkcji takiej zostało umieszczone wczytywanie wszystkich zasobów z dysku. W momencie kiedy funkcja wczyta wszystkie dane, wtedy ustawi flagę oznaczającą zakończenie ładowania na wartość `True`. Zmiana wartości tej flagi spowoduje zakończenie wyświetlania ekranu powitalnego i uruchomi główną pętle gry. Cała funkcjonalność związana z ekranem powitalnym została umieszczona w osobnej klasie `Splash` która jest dziedziczona przez główną klasę aplikacji - `App`. Dzięki takiej implementacji klasa `App` zajmuje się jedynie inicjowaniem bibliotek, oraz obsługą głównej pętli.

Należy wspomnieć ile miejsca zajmują omawiane zasoby na dysku. W przypadku dźwięków jest to około 10 MB, grafika to 3 MB, a sam plik z mapą to 1.3 MB. Na wczytanie takich ilości danych w zależności od sprzętu może być potrzebne nawet do kilku sekund.



RYSUNEK 2.4: Schemat uruchamiania gry

## 2.4 Przyjęte konwencje, logowanie i obsługa błędów

W fazie analizy projektu zostały przyjęte pewne konwencje które na celu miały przyśpieszyć proces usuwania błędów w kodzie, zapewnić jego czytelność i wysoki poziom. Dodatkowo zostały wprowadzone pewne mechanizmy które miały uspójnić projekt, zostanie to przybliżone w tym rozdziale.

Nazwy metod oraz pól w aplikacji zostały w większości przypadku zapisane w formie "camelCase", czyli notacji polegającej na pisaniu nazwy składającej się z kilku wyrazów łącznie zaczynając każdy kolejny wyraz z dużej litery. Wyjątek stanowi pierwsza litera z nazwy, która jest małą ( w przeciwieństwie do notacji PascalCase gdzie początek nazwy też jest pisany z dużej litery ). Dodatkowo przyjętą konwencją było pisanie przy nazwach pól litery "p" na początku, bądź też "is" w przypadku zmiennych logicznych. Taki zapis

pozwala odróżnić nazwy funkcji od pól. Akcesory do pól prywatnych są nazywane z konwencją przyjętą w języku Java tzn. z przedrostkami get, set, is. Zmienne stałe pisane są drukowanymi literami, a odstępy między wyrazami w takiej nazwie stanowi twarda spacja.

W celu zachowania wysokiej jakości kodu w programie nie są używane takie elementy języka jak skoki goto, zmienne globalne (z wyjątkiem stałych), zaprzyjaźnianie klas. Same klasy natomiast są częściowo opisane za pomocą komentarzy doxygena.

Kwestia dołączania bibliotek do poszczególnych plików źródłowych została rozwiązana za pomocą umieszczenia wszystkich importów bibliotek (ang.include) w jedynym wspólnym pliku, dzięki czemu nie był potrzeby dołączania do każdego pliku np. biblioteki Lua. Plik z potrzebnymi importami to Headers.hpp. Dodatkowo w pliku tym zdefiniowane zostały stałe globalne używane w aplikacji, a nie podlegające modyfikacji z poziomu skryptów Lua np, szybkość biegu gracza. Plik ten dodatkowo zawiera przestrzeń nazw GameSpace zawierającą enumeratory używane np. do określenia rozmiaru czcionki wypisywanego na ekranie tekstu.

W aplikacji w przypadku błędów krytycznych, w przypadku których aplikacja ma zakończyć działanie natychmiast zostaje wyrzucony wyjątek z dowolnego miejsca w aplikacji. Wszystkie nie złapane wyjątki są obsługiwane i logowane w funkcji main. W celu ujednolicenia rzucanych wyjątków wyrzucenie wyjątku zawsze ma następujący wygląd: `throw std::runtime_error(<nazwa_klasy>:<nazwa_metody>)`. Gdzie przed wyrzuceniem wyjątku logowane jest co się dokładnie stało. W main po złapaniu wyjątku logowana jest treść wyjątku, czyli nazwa metody z której został wyrzucony, następnie zostaje usunięty obiekt klasy App i zwrócony kod błędu EXIT\_FAILURE.

W celu ujednolicenia mechanizmu logowania została utworzona klasa Logger, która posiada zestaw funkcji wypisujących na ekranie komunikat z odpowiednim priorytetem (DEBUG, INFO, WARNING, ERROR, CRITICAL, FATAL). Z tym że funkcja wypisująca w informacje z priorytetem debug wypisze komunikat tylko wtedy kiedy zdefiniowana jest stała DEBUG. Klasa ta posiada dodatkowo pole na nazwę klasy która używa logowania, pole te może zostać ustawione za pomocą konstruktora w klasie która rozszerza klasę Logger. Informacja z nazwą klasy dopisywana jest do każdego użycia loggera, dzięki czemu łatwiej odnaleźć w logach przyczynę błędu. Takie rozwiązanie na potrzeby

tak małego projektu okazało się wystarczające, w przypadku większych projektów zdecydowanie lepszym pomysłem może okazać się użycie dodatkowej biblioteki np. log4c (dla języka C++ jest to odpowiednik znanej z Javy biblioteki log4j).

Warto wspomnieć także o tym że duża część klas miała w założeniu używać tych samych elementów, pozwalających wypisywać tekst na ekranie, rysować grafikę, oraz mieć dostęp do takich danych jak np. wymiary ekranu. Dlatego też żeby ograniczyć powtarzanie się tych samych elementów kodu wspólne elementy zostały przeniesione do klasy StandardReference. Podejście takie przy projektowaniu systemów informatycznych nazywane jest często nazywane jako DRY (ang. Don't Repeat Yourself). Zamiast w każdej klasie implementować te same metody i wpisywać te same pola, umieszcza się je w jednym miejscu. Pozwala to zmniejszyć ilość kodu, a co za tym idzie jego złożoność. Dodatkowo ewentualne poprawki wymagają zmiany w jednym miejscu, a nie w każdej klasie. Klasa StandardReference, zawiera wskaźniki do takich klas jak Renderer, Writer, SoundManager, dzięki czemu każda inna klasa dziedzicząca po niej ma możliwość używania zestawu funkcji dostępnych w tych klasach, pozwalającego rysować na ekranie, obsługiwać dźwięki oraz wypisywać tekst na ekranie.

## 2.5 Jak zbudowana jest aplikacja?

Logika aplikacji została rozbita na 3 główne warstwy. Dzięki czemu aplikacja mimo zwiększonej złożoności stała się bardziej elastyczna na przyszłe modyfikacje. Pierwszą z nich stanowi uruchamiana z poziomu funkcji main klasa App, której głównym zadaniem jest utworzenie okna aplikacji, uruchomienie głównej pętli i sterowanie jej przebiegiem.

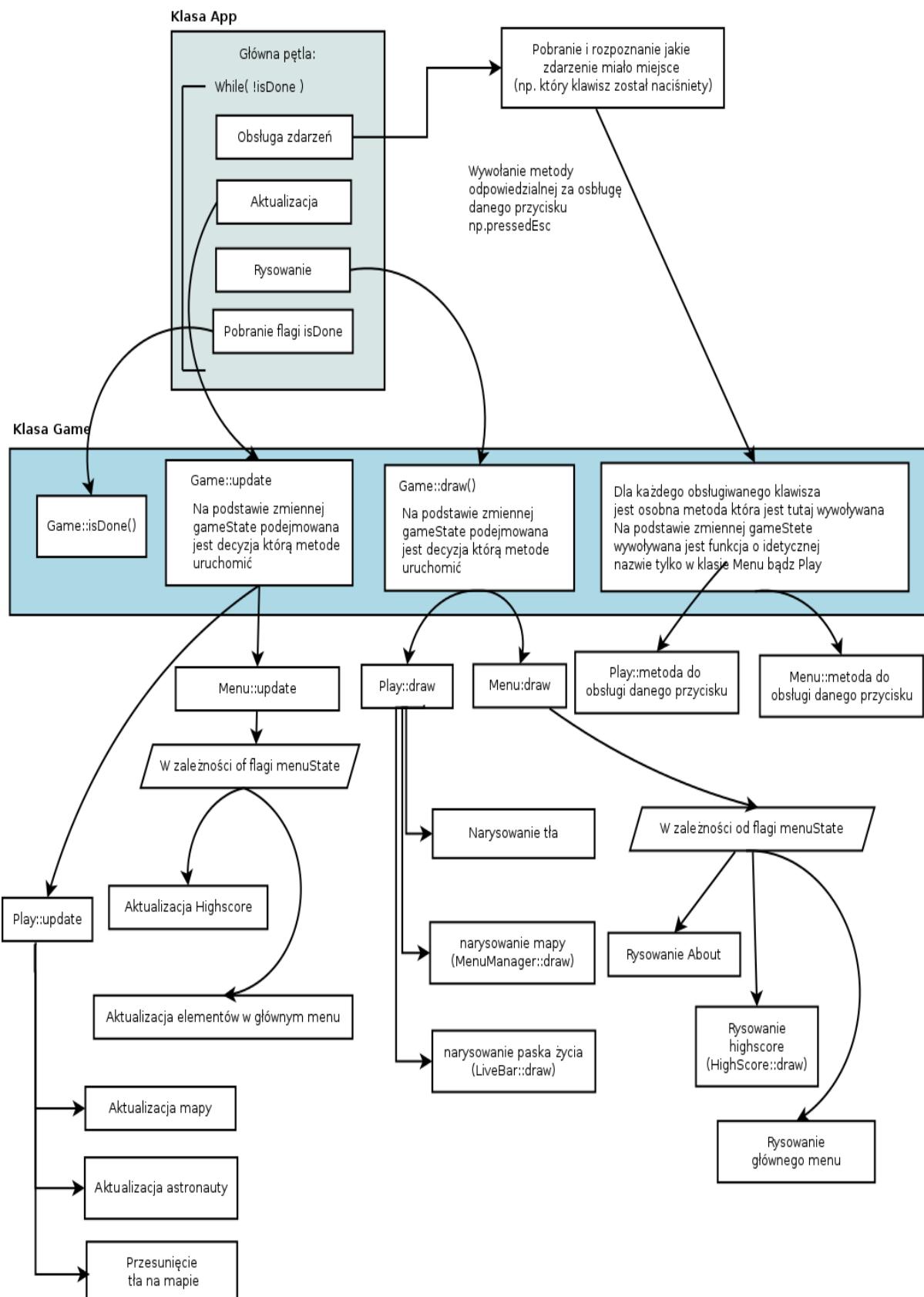
Klasa App oddziela pozostałe moduły aplikacji od zarządzania oknem, oraz zdarzeniami. Zdarzenia są pobierane w metodzie App::processEvent, a następnie w zależności od rodzaju zdarzenia wywoływana jest odpowiednia metoda klasy Game, będącej fasadą ujednolicającą dostęp do modułów odpowiedzialnych za rysowanie, odświeżanie menu, gry bądź też highscore. Klasa ta zarządza przepływem informacji decydując gdzie przekazać dane na podstawie stanu w jakim znajduje się aplikacja. Stan ten opisuje zmienna typu enum z polami: MENU oraz PLAY, to właśnie na jej podstawie wywoływane są metody np. draw, update w kolejnej warstwie. Klasa Game jest więc warstwą pośrednią między

warstwą okna aplikacji a poszczególnymi modułami gry, stanowi też implementacje części logiki gry, ponieważ rozdziela menu od gry. Najważniejszymi modułami poniżej warstwy pośredniej jaką jest Game, są klasy Play oraz Menu. Są one najważniejszą warstwą aplikacji, zarządzającej pozostałymi modułami, takimi jak np. dźwięk, pasek życia.

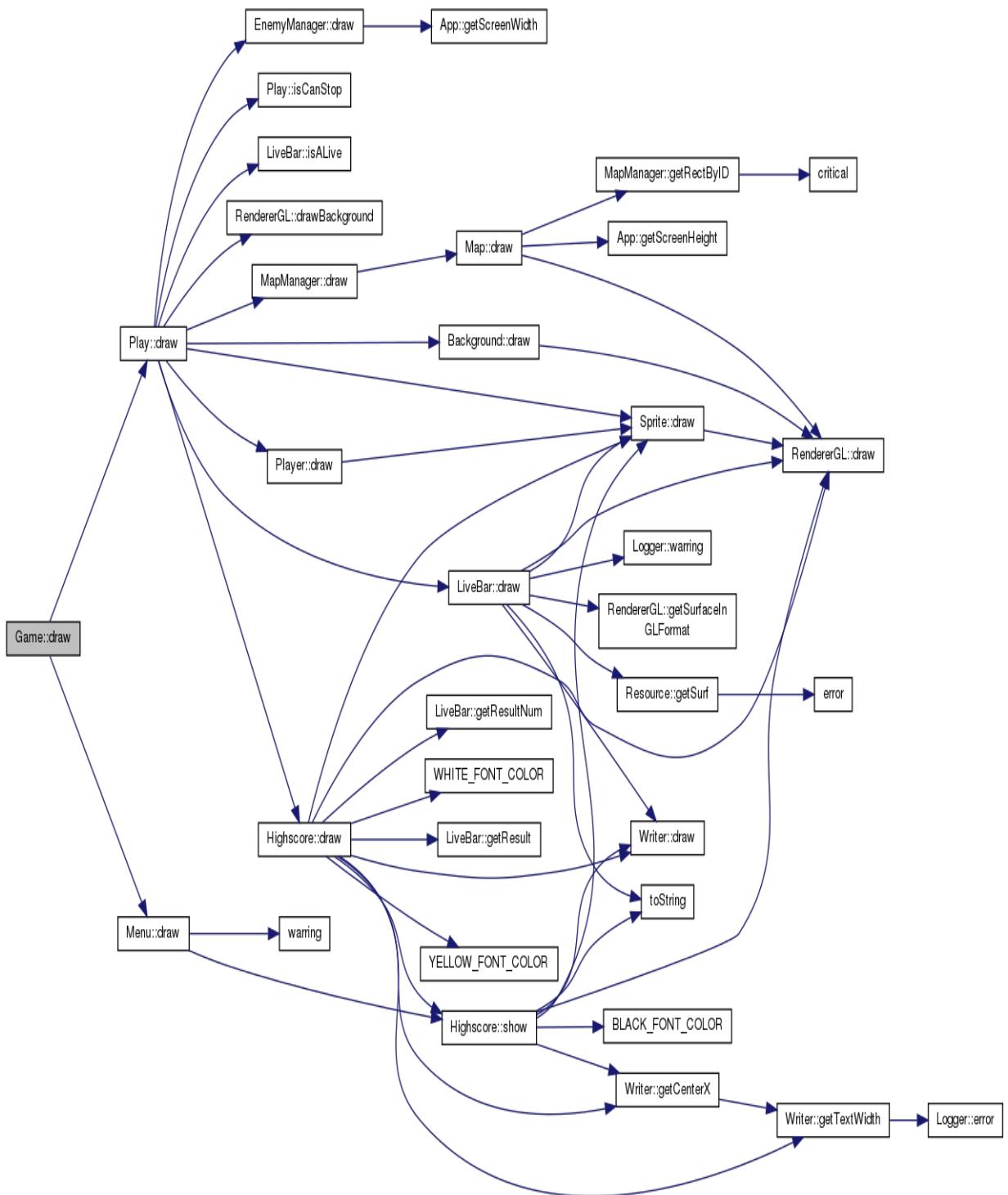
Klasa Play odpowiada jak sama nazwa wskazuje ze wszystko co się dzieje po przejściu z menu do gry. Jest do zarządzania mapą, paskiem życia gracza, samym graczem. Na poziomie tej właśnie klasy odbywa się przepływ informacji między klasą Player a klasami odpowiadającymi za ilość życia kosmonauty i klasą Highscore przechowującą ilość zdobytych punktów. Podczas rysowania, aktualizacji wołane są metody w modułach powiązanych z tą klasą. Przykładem może być tutaj klasa MapManager, która to zarządza mapami (w przypadku Astro Rush wykorzystana jest tylko jedna mapa, choć istnieje możliwość rozbudowania o gry kolejne mapy nakładające się na siebie, i przewijające jednocześnie z różną szybkością ).

Klasa Menu odpowiada jak sama nazwa wskazuje za wyświetlanie menu głównego aplikacji, oraz za wyświetlanie Highscore. Jest ona dużo prostsza niż klasa Play, bowiem oprócz klasy Highscore nie zawiera już pod modułów którymi zarządza. Samo rysowanie menu odbywa się bezpośrednio w metodzie draw, przy pomocy vector-a obiektów MenuItem. Klasa Menu posiada flagę typu enum, która określa czy gracz aktualnie znajduje się w menu głównym, czy też w menu z listą najlepszych wyników, które stanowi osobny moduł. Klasa Highscore, bo właśnie o niej mowa posiada swoje metody typowe dla większości modułów gry tj. update, draw. W klasie Menu następuje wybór metod, np. czy rysowanie ma odbyć się za pomocą draw z klasy Menu, czy z klasy Highscore. Analogicznie jak to ma miejsce w klasie Game.

Tak zbudowana aplikacja traci na wydajności, bowiem przy wywołaniu z głównej pętli funkcji rysującej, tak naprawdę zostaje wywołane kilka innych metod, aż do tej właściwej. Wiąże się to z narzutem czasowym odkładania parametrów podczas wywołań metod, przez co konieczne było przekazywanie bardziej złożonych danych nie przez wartość, tylko przez referencje. Przykład skompilowania takich wywołań został przedstawiony na diagramie wywołań funkcji draw. Przykładowo wywołanie draw może wykorzystywać klasy: Game, Play, MapManager, Map, zanim nastąpi właściwe rysowanie. Plusem takiego podejścia jest jednak bardziej przejrzysty i łatwy w utrzymaniu kod.



RYSUNEK 2.5: Schemat przepływu danych w aplikacji



RYSUNEK 2.6: Schemat przepływu danych w przypadku wywołania funkcji Game::draw wygenerowany za pomocą programu Doxygen.

# Rozdział 3

## Dokumentacja użytkownika

### 3.1 Sterowanie w grze

Astronauta w grze biegnie ciągle do przodu, a osoba sterująca nim ma możliwość wykonania skoku poprzez naciśnięcie klawisza spacja. W przypadku kiedy gracz chce wykonać krótki skok, tzn. nie może poczekać aż bohater przestanie się wznosić wtedy z pomocą przychodzi klawisz Ctr którym zatrzymujemy ruch pionowy kosmonauty. Zatrzymanie następuje na ułamek sekundy, jednak naciskając bardzo szybko ten klawisz można sprawić że bohater leci prawie poziomo. Może się to okazać pomocne przy przelatywaniu przez jakieś tunele na mapie.



RYSUNEK 3.1: Licznik bonusów

wytraceniu wszystkich znalezionych dodatków. Znaleziony bonus można aktywować

Podczas biegu na mapie można zebrać bonusy które przywracając pełną ilość życia i sprawiają że życie te nie opada przez kilka sekund. Po znalezieniu takie bonusu w lewym górnym rogu pojawia się informacja o tym. Maksymalnie można mieć 3 niewykorzystane bonusy, i podczas działania jednego nie da się uruchomić kolejnego, co zapobiega zbyt szybkiemu

poprzez wcisnięcie lewego klawisza shift. Po aktywowaniu bonusu dokoła ekranu pojawia się niebieska otoczka oznaczająca nieśmiertelność.

## 3.2 Menu

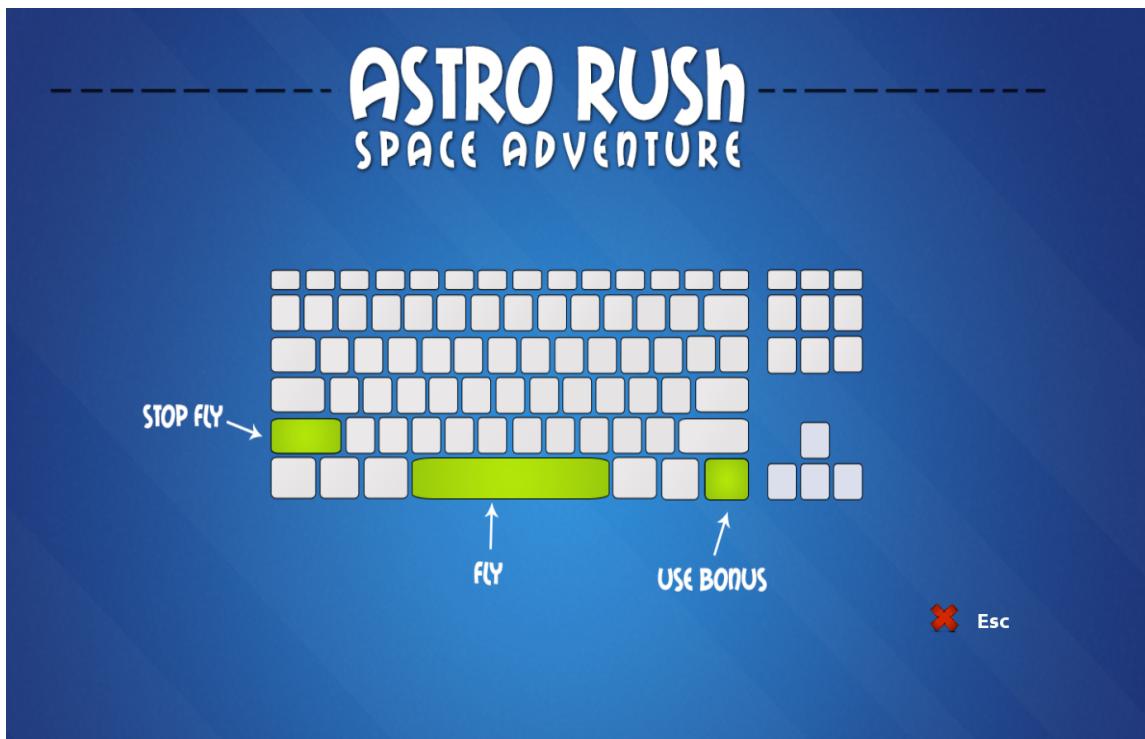
Główne menu zostało zaprojektowane w programie do tworzenia grafiki rastrowej Adobe Photoshop. Z jego poziomu można rozpoczęć nową grę, kontynuować bieżącą, obejrzeć klawiszologie, oraz listę najlepszych wyników w formie tabelki. Możliwość konstytuowania bieżącej gry staje się aktywna w momencie kiedy gracz podczas gry nacisnął klawisz escape co za skutkowało wyjściem do głównego menu, oraz zamrożeniem bieżącej rozgrywki. Nawigacja w menu odbywa się wyłącznie za pomocą klawiatury. Klawisz strzałka w góre przechodzi do następnej pozycji powyżej, strzałka w dół działa natomiast odwrotnie. Aktualnie wybrany element menu podświetlany jest za pomocą koloru pomarańczowego, nie aktywny kolorem szarym, a domyślny białym. Zatwierdzenie wybranej opcji następuje poprzez wcisnięcie klawisza enter, co stanowi standardową nawigację w większości menu do gier. Naciśnięcie klawisza w głównym menu spowoduje zamknięcie aplikacji.



RYSUNEK 3.2: Menu główne gry



RYSUNEK 3.3: Highscore w grze



RYSUNEK 3.4: Klawiszologia dostępna poprzez menu główne w grze

### 3.3 Instalacja

W tym rozdziale zostanie przedstawiona instalacja potrzebnych bibliotek oraz komplikacja aplikacji na przykładzie systemu Debian. W przypadku innych wersji systemu Linuks nie bazujących na Debianie instalacja ta może wyglądać inaczej np. w systemie Fedora gdzie nazwy pakietów są inne, oraz ścieżka do biblioteki SDL jest inna. Dla Fedory komplikacja aplikacji wymaga zmian ścieżek do bibliotek w pliku Headers.hpp. Dlatego też w przypadku wersji aplikacji który miała by być dostarczona szerszemu gronu odbiorców konieczne jest przygotowanie rozwiązania unifikującego proces komplikacji pod różnymi dystrybucjami Linuksa. Innym rozwiązaniem może przygotowanie prekompilowanych pakietów dla najpopularniejszych platform np. deb i rpm. Dodatkowym problemem może okazać się brak niektórych pakietów w repozytorium i konieczność ręcznego ichściągania, jednak taka sytuacja nie powinna mieć miejsca z racji tego że wykorzystane biblioteki są popularne wśród twórców gier dla Linuksa.

W systemie Debian żeby zainstalować wymagane pakiety należy najpierw odświeżyć listę pakietów polecienniem:

```
aptitude update
```

W przypadku braku aplikacji aptitude można użyć analogicznie apt-get. Następnym krokiem jest zainstalowanie wymaganych bibliotek w wersji developerskiej w przypadku samodzielnej komplikacji:

```
aptitude install libsdl-1.2-dev lisdl-ttf2.0-dev libsdl-sound1.2-dev  
libsdl-mixer1.2-dev libsdl-image1.2-dev libluabind-dev  
liblua5.1-0-dev
```

Jeżeli instalacja przebiegła bez problemów to wtedy można przystąpić do komplikowania gry. W katalogu z rozpakowanym kodem należy wykonać polecenie make i poczekać aż pojawi się informacja o udanym zbudowaniu programu. W przypadku komputerów z wielordzeniowym procesorem można zoptymalizować proces budowania podając parametr -j<tutaj\_liczb\_rdzeni>.

W przypadku kiedy budowanie się nie powiedzie i pojawią się błędy, należy doinstalować brakujące biblioteki bądź też zmienić ścieżki do bibliotek. Ewentualnie oczyszczenie projektu z już skompilowanych plików można wykonać za pomocą polecenia "make clean".

Po zakończeniu budowania zostanie utworzony plik AstroRush.bin (nazwa ta jest zdefiniowana w pliku makefile). Należy nadać mu prawo wykonywania:

```
chmod +x ./AstroRush.bin
```

A następnie można uruchomić aplikacje. W przypadku Linuksa należy pamiętać żeby program był uruchamiany na partycji zamontowanej z prawem do wykonywania aplikacji. Często domyślnie urządzenia przenośne nie mają ustawionego takiego prawa.

# Zakończenie

W pracy została przedstawiona budowa platformowej gry zręcznościowej mogącej pracować w systemie Linuks, gdzie brakuje gier oraz w najpopularniejszym obecnie systemie dla komputerów osobistych- Windows. Gra została stworzona w oparciu o własny silnik, pokazujący możliwości biblioteki SDL. Zaczynając od możliwości stworzenia i zarządzania oknem, poprzez integracje z OpenGL aż do obsługi dźwięku oraz czcionek. Omówiona biblioteka nie ustępuje możliwością bibliotece DirectX, której ogromnym ograniczeniem jest brak wsparcia dla systemu Linuks. Ponadto aplikacja pokazuje przykładowa budowę gry w której do renderowania grafiki wykorzystuje się sprite-y oraz mapy kafelkową. Techniki te są jednymi z najbardziej podstawowych zagadnień w grach opartych o grafikę dwuwymiarową.

W pracy został poruszony temat rozszerzenia możliwości programu w oparciu o tzw. skryptowanie. Zaprezentowane skrypty w języku Lua stanowią obecnie wtyczki rozszerzające funkcjonalność nie tylko w grach, lecz wszędzie tam gdzie przeniesienie pewnych informacji poza skompilowany program pozwala oszczędzić czas potrzebny na jego napisanie.

Stworzona aplikacja nie musi być tylko programem napisanym do pracy dyplomowej, może zostać ona upowszechniona jako projekt Open Source. Nie wykluczone jest że wtedy będzie dalej rozwijana przez osoby chętne do wolontariatu. Udostępnienie kodu jako darmowego pozwala również na stworzenie pakietu prekompilowanego dla systemu Debian ( Plik z rozszerzeniem deb) i zgłoszenie go do repozytorium dystrybucji Debian, bądź Ubuntu.

Inną możliwością co do przyszłością programu jest sprzedawanie gry jako tzw. Indie Game czyli gry niezależnej, stworzonej bez wsparcia finansowego przez jedną bądź kilka osób. Taką aplikację można sprzedawać w formie elektronicznej za nie wielkie pieniądze.

Aplikacja może zostać uruchomiona na tablecie z systemem BlackBerry i umieszczona w markecie z aplikacjami dla tego systemu. Mowa jest tutaj szczególnie o tabletach ponieważ aplikacja podlega ograniczeniom odnośnie rozdzielczości ekranu na którym może zostać uruchomiona, co zostało opisane bardziej szczegółowo w pracy.

Efektem ubocznym stworzenia gry Astro Rush było stworzenie silniki gry 2D. Silnik ten dzięki zastosowaniu zewnętrznych skryptów podatny jest na modyfikacje i zmianę. Prosta budowa pozwala na stworzenie na bazie bieżącej aplikacji kolejnej gry. Użycie biblioteki OpenGL do renderowania grafiki daje możliwość stworzenia dużo bardziej zaawansowanej grafiki, wykorzystującej cieniowanie i oświetlenie.

# Spis rysunków

1.1	Silnik Cry Engine . . . . .	4
1.2	Postać astronauty . . . . .	5
1.3	Pasek życia wyświetlany w lewym górnym rogu . . . . .	6
1.4	Animacja biegu astronauty . . . . .	6
1.5	Przykładowy screen z gry . . . . .	7
1.6	Przykładowy screen z gry . . . . .	8
1.7	Algorytm wyświetlania animacji w oparciu o tzw. „sprite“ . . . . .	9
1.8	Eclipse podczas instalacji wtyczki z Marketplace . . . . .	10
1.9	Portal github.com. Podgląd pliku źródłowego. . . . .	14
1.10	Przepływ danych między aplikacją C++ a skryptem Lua . . . . .	18
2.1	Schemat działania głównej pętli . . . . .	23
2.2	Przewijane tło wyświetlane w grze . . . . .	24
2.3	Edytor mapy podczas pracy . . . . .	25
2.4	Schemat uruchamiania gry . . . . .	27
2.5	Schemat przepływu danych w aplikacji . . . . .	31
2.6	Schemat przepływu danych w przypadku wywołania funkcji Game::draw wygenerowany za pomocą programu Doxygen. . . . .	32
3.1	Licznik bonusów . . . . .	33
3.2	Menu główne gry . . . . .	34
3.3	Highscore w grze . . . . .	35
3.4	Klawiszologia dostępna poprzez menu główne w grze . . . . .	35

# Bibliografia

- [1] Janusz Ganczarski. *OpenGL w praktyce*. Wydawnictwo BTC, 2008.
- [2] Ernest Pazera. *Focus o SDL*. Premier Press, 2003
- [3] Jacek Zagrodzki *Mapy kafelków w grach 2D*. Software Developer's Journal, 02/2010
- [4] Rafał Kocisz *Biblioteka Luabind*. Software Developer's Journal, 03/2009
- [5] Paweł Rohleder *Język skryptowy Lua*. Software Developer's Journal, 08/2009