POLITECHNIKA ŚLĄSKA

WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

KIERUNEK INFORMATYKA

Projekt inżynierski

Opracowanie biblioteki narzędziowej dla kostek Sifteo

Autor: Krzysztof Kundera

Kierujący pracą: dr inż. Krzysztof Dobosz

Gliwice, luty 2018

Spis treści

1. Wstęp

1.1. Wprowadzenie

Kostki *Sifteo* stworzone zostały przez David'a Merrill'a i Jeevana Kalanithi'ego z MIT Media Lab pod nazwą „*Siftables*". Pierwszy raz kostki ujrzały światło dzienne jako prototypy w 2009 roku na konferencji TED. Merrill i Kalanithi założyli własną firmę, „*Sifteo Inc*." której pierwszym produktem były kostki *Sifteo*. Debiut kostek był sporym wydarzeniem. Odbył się w 2011 roku na targach elektroniki użytkowej CES (ang. „*Consumer Electronics Show"*) który ma miejsce co roku w styczniu w Las Vegas. Po konferencji ruszyła sprzedaż kostek.

Następcą kostek *Sifteo* był „*Sifteo Cubes Interactive Game System*" (pol. Interaktywny system gier na kostki *Sifteo*) wydany w 2012 roku. Niestety, *Sifteo* nigdy nie okazały się marketingowym sukcesem. W początkowych prezentacjach zainteresowanie kostkami było spore. Była to bowiem innowacyjna na tamte czasy technologia, coś takiego nie było wcześniej spotykane. Kostki wydawały się dobrym narzędziem do zabawy i nauki. Pokazuje to wiele prezentacji które były wtedy przeprawadzane z grami edukacyjnymi, ćwiczącymi pamięć i kreatywność. Twórcy kostek zadbali też o różnorodność gier. Jednak kostki nigdy nie osiągnęły zapowiadanego sukcesu. W połowie roku 2014 firma „*Sifteo Inc.*" została przejęta przez firmę „3D Robotics", a po paru miesiącach części oprogramowania zostały opublikowane jako otwarte biblioteki.

Sifteo SDK jest obecnie dostępne jako otwarte oprogramowanie. Używane jest w projektach naukowych oraz innowacyjnych projektach rehabilitacyjnych. Takie użycie jest możliwe dzięki niewielkiemu rozmiarowi kostek. Od 2009 roku rynek mocno się zmienił. Ekrany dotykowe w smartfonach oraz tablety stały się powszechne. *Sifteo* było innowacyjnym produktem wtedy, lecz teraz przegrywa z nowymi urządzeniami z powodu ograniczonej funkcjonalności. Urządzenia, takie jak tablety, posiadają obecnie oprogramowanie które pozwala na uruchamianie bardzo dużej ilości typów aplikacji. Projekty gier na *Sifteo* mają niestety swoje, inne zasady. Nie mogą być one przeniesione na inne urządzenia ze względu na wyjątkowy zestaw cech, które posiada *Sifteo*.

Motywacją dla powstania pracy jest obecny brak interfejsu programistycznego wysokiego poziomu, który jest wygodny i przystępny w obsłudze. Byłby on niezmiernie przydatny na zajęciach laboratoryjnych Politechniki.

1.2. Cel pracy, zakres pracy

Celem pracy jest opracowanie API (ang. *Application Programming Interface*), czyli interfejsu służącego do programowania aplikacji dla technologii kostek *Sifteo*. Interfejs ten ma być ulepszeniem istniejącego już interfejsu, który został przygotowany w języku C++. [1] Interfejs powinien ułatwiać użytkownikowi tworzenie aplikacji oraz zapewniać wysoki poziom abstrakcji przy zachowaniu intuicyjności tworzenia projektów. Interfejs powinien wykorzystywać istniejący już, niskopoziomowy interfejs C++.

1.3. Przewodnik po treści pracy

Drugi rozdział przedstawia obszernie problematykę projektu, wstępną analizę oraz definicje wszystkich pojęć potrzebnych do zrozumienia treści pracy. Opisuje tematykę pracy, wyjaśnia użyte wewnątrz niej pojecia oraz przedstawia stan technologii w obecnej chwili. Wewnątrz rozdziału zawarty został obszerny opis istniejącego API *Sifteo* oraz przedstawienie wykorzystywanych w pracy wzorców projektowych. Informuje również o technologiach, które zostały wykorzystane podczas prac nad interfejsem.

Trzeci rozdział przedstawia projekt implementowanego przeze mnie API. Rozdział zawiera diagram sekwencji, schemat klas interfejsu oraz krótkie omówienie decyzji dotyczących struktury architektury aplikacji oraz jej działania.

Czwarty rozdział zawiera przykłady wypracowanych rozwiązań. Pokazuje on implementację dwóch najważniejszych elementów interfejsu:

- metodę umożliwiającą wypisywanie ciągów znakowych i jej szerokie możliwości manipulacji przetwarzanymi ciągami znakowymi,
- metodę obsługującą wczytywanie obrazów i przetwarzanie plików opisu zasobów.

Piąty rozdział przedstawia interfejs z punktu widzenia użytkownika i zawiera instrukcje dotyczące pierwszego użycia, instalacji oraz uruchamiania projektów *Sifteo.* Rozdział skupia się na krokach, które użytkownik musi podjąć aby poprawnie korzystać z aplikacji.

Szósty rozdział przedstawia utworzone projekty przykładowe, służące za wprowadzenie w technologię *Sifteo.* Tutaj znajduje się opis projektów które mają być pierwszymi krokami dla programistów, którzy nie mieli jeszcze styczności z tą technologią. W tym rozdziale opisane są również wymagania jakie projekt musi spełniać, by móc wyświetlać tekst i obrazy na ekranie kostek.

Zawartość rozdziału siódmego, dotyczącego weryfikacji, obejmuje przedstawienie i instrukcje uruchomienia testów jednostkowych oraz porównanie wcześniejszych implementacji projektów do implementacji wykorzystujących wytworzony, nowy interfejs.

2. Analiza zadania

2.1. Omówienie istniejących już technologii

2.1.1. Platforma Sifteo Cubes

Sifteo to interaktywne kostki rozmiaru około 4x4 centymetry. Kostki posiadają wiele wbudowanych czujników: m.in. akcelerometr. Wykrywają m.in. pochylenie, położenie względem innych kostek, obrót a także przyciśnięcie. [1] Każda z kostek jest wyposażona między innymi w:

- 32-bitowy CPU,
- akcelerometr pracujący w trzech osiach,
- własnościową technologię wykrywania obiektów w sąsiedztwie,
- 8MB pamięci Flash,
- ekran LCD o rozmiarze 128x128 pikseli.

Kostki porozumiewają się poprzez sieć bezprzewodową z Bazą *Sifteo,* która przechowuje gry oraz aplikacje i dzieli się z kostkami danymi tych aplikacji. Baza ma kształt podłużnego, czarnego prostopadłościanu, który posiada przycisk służący do włączania i wyłączania systemu. Jest ona odpowiedzialna za przechowywanie aplikacji oraz zarządzanie systemem *Sifteo*. Baza odpowiada za pobieranie i instalację nowych aplikacji oraz dystrybucję danych w czasie działania systemu *Sifteo*.

Aplikacje użytkowe w *Sifteo SDK* są kompilowane do standardowych plików binarnych o rozszerzeniu *.elf. Takie pliki wykonywalne zawierają kod aplikacji, jej dane oraz pliki opisu zasobów wykorzystywanych przez aplikację. Wielkość pliku binarnego posiada ograniczenie 16 MB.

Aplikacje uruchamiane są w maszynie wirtualnej *Sifteo,* SVM.

Wyświetlacze LCD kostek posiadają rozmiar 128 na 128 pikseli, każdy obsługujący 16 bitową głębię kolorów. Wyświetlacze obsługują format RGB565, co oznacza że 5 bitów z 16 jest przeznaczonych na kanał czerwony oraz niebieski, podczas gdy kanał zielony jest zapisywany na 6 bitach. Podczas przygotowywania obrazów powinniśmy pamiętać, że przejdą one przez wiele poziomów optymalizacji. Należy również starać się, by obrazy posiadały wysoki kontrast, by były widoczne na wyświetlaczach LCD pod każdym kątem.

2.1.2. Sifteo SDK i zawarte wewnątrz narzędzia

Sifteo SDK (ang. *Software Development Kit)*, czyli zbiór narzędzi służących do tworzenia oprogramowania, została udostępniona w wolnych bibliotekach. [1] Obejmuje to symulator

„*Siftulator*", dokumentację API *Sifteo* C++, przykładowe projekty i gry na *Sifteo*, nagłówki C++, narzędzie do budowania projektów oraz mnóstwo bibliotek dynamicznych. Udostępnione w SDK narzędzia pozwalają na kompletną symulację fizycznych kostek *Sifteo* w aplikacji, która tworzy pole gry dla kostek. Aplikacja pozwala nam manipulować kostkami. Dostępne są wszelkie gesty, które były dostępne w kostkach. Gestów można używać poprzez skróty klawiszowe podczas symulacji. Aplikacja symulatora pozwala również na wgranie swojego projektu oraz tworzenie i usuwanie wirtualnych kostek z pola gry. Możliwe jest nawet symulowanie stanu niskiej baterii w urządzeniu.

Proces kompilacji projektów *Sifteo* przebiega następująco [1]:

- każdy projekt posiada kod aplikacji napisany w C++, który jest kompilowany przez Clang,
- kompilacja jest wspomagana przez narzędzie optymalizacji *slinky,* które wspomaga linkowanie,
- podczas kompilacji wykorzystywane jest również narzędzie *stir,* odpowiadające za kompresję plików opisu zasobów poprzez generację plików C++ zawierających zasoby. *Stir* przyjmuje instrukcje przetwarzania plików w formacie skryptów *.lua,
- pliki z opisem zasobów wygenerowane w ten sposób są wbudowywane przez linker *slinky* do pliku binarnego projektu.

Zadaniem aplikacji *Stir* jest wykrywanie podobnych części obrazu i zastępowanie ich jedną, która jest używana przez oba pliki opisów zasobów. Stopniem dozwolonej kompresji każdego obrazu możemy sterować przy wywołaniu tej aplikacji i podaniu jej odpowiedniego przełącznika w zakresie 1 .. 10, gdzie 10 oznacza całkowity brak kompresji. a 1 pozwolenie na kompresję w najwyższym możliwym stopniu. Dokumentacja *Sifteo* zaleca, aby nie używać wartości 10, a zamiast tego użyć np. 9.98. [1] *Stir* jest potrzebny w systemie, ponieważ pamięć jest bardzo ważnym parametrem przy programowaniu na systemy wbudowane. Pamięć urządzeń jest zazwyczaj mocno ograniczona, co powoduje, że wszelkie możliwe optymalizacje zwiększają nam znacznie wydajność aplikacji. Im mniej danych musimy przesłać, tym mniej operacji musimy poświęcać na kontrolę przesyłania i wyświetlania obrazów a urządzenie jest w stanie wykonać więcej innych działań. Znaczna kompresja obrazu oznacza również, że wiele obrazów dzieli ten sam kawałek obrazu. Dzięki temu przełączanie pomiędzy obrazami będzie szybsze, gdyż część potrzebnych danych znajduje się już w kostce.

System nie jest w stanie usunąć pojedynczego pliku opisu zasobów jak i grupy takich plików w obiekcie typu *AssetGroup* ze względu na poczynione optymalizacje pamięciowe. Jeśli chcemy dodać plik opisu, możemy to zrobić tworząc nową grupę plików opisu zasobów oraz przyporządkowując ją do obiektu klasy *AssetSlot*. Jeśli chcemy usunąć grupę lub plik opisu zasobów, musimy niestety usunąć cały obiekt klasy *AssetSlot*.

Podczas kompilacji projektów, środowisko uruchomieniowe *Sifteo* używa plików *makefile*. Pliki te opisują i wspomagają przebieg kompilacji poprzez przechowywanie informacji o obiektach tymczasowych (*.o) oraz plikach źródłowych (tekstowych i graficznych) dołączonych przez użytkownika. [10] Pliki te uruchamiane są poprzez komendę *make* w konsoli *Sifteo.*

2.1.3. Silnik graficzny i jego działanie

System kostek *Sifteo* cechuje się rozproszonym renderowaniem grafiki. [1] Oznacza to, że każda kostka posiada wbudowane urządzenia odpowiedzialne za składowanie grafiki, tworzenie jej w silniku oraz wyświetlanie. Aplikacja udostępnia kostkom bufor danych do wyświetlenia, które każda kostka pobiera poprzez przesył radiowy z *Bazy*. Każda kostka zawiera dwa typy pamięci. Pierwszy to pamięć zasobów. Jej wielkość to 8 MB. Zawiera nieskompresowane dane. Drugi typ pamięci to pamięć wideo o wielkości 1 kB. Pamięć ta, w odróżnieniu od pamięci zasobów, zawiera metadane oraz komendy. Zawartość tej pamięci jest cały czas synchronizowana z pamięcią *Bazy*.

W celu optymalizacji przesyłów radiowych, zasoby są przechowywane w specyficzny sposób, bo podwójnie. Podczas kompilacji *stir* analizuje zasoby które użytkownik polecił mu włączyć wewnątrz jednej grupy zasobów. Dzieli wszystkie obrazy na obszary 8x8 pikseli, po czym szuka w nich jak najwięcej podobnych obszarów. Warto zaznaczyć w tym momencie, że w pojedynczej klatce wyświetlania mieści się 18x18 takich obszarów. Wszystkie unikatowe obszary wyłuskane z grupy assetów trafiają do jednego zbioru wewnątrz *AssetGroup*. *AssetGroup* jest następnie przesyłany do docelowych kostek, które przechowują go w pamięci zasobów. Aby wyświetlić na ekranie kostki obraz, przesyłamy do niej dane w obiekcie klasy *AssetImage*. Ta klasa przetrzymuje dla nas w tablicy odnośniki do unikatowych obszarów wytworzonych przez *stir*. Tablica o której tu mówimy, zawiera oczywiście tyle pól, ile obszarów 8x8 posiadał oryginalny obraz. Kiedy silnik graficzny posiada oba zbiory, łączy je wykorzystując identyfikatory obszarów i wyświetla na ekranie LCD. Przetrzymywanie odnośników do obszarów w tablicy nie pochłania tak dużo miejsca jak unikatowe obszary zawarte w *AssetGroup*, dlatego pamięć wideo kostki nie musi być też aż tak duża.

Każda z kostek pobiera dane do pamięci grafiki o swobodnym dostępie, po czym przetwarza dane wewnątrz silnika graficznego. Wewnątrz silnika możemy wyróżnić trzy etapy renderowania obrazu: Pobieranie danych specyficzne dla aktywnego trybu (*Mode renderer*), rysowanie (*Windowing*) oraz obracanie (*rotation*).

Każda kostka *Sifteo* jest w stanie przetwarzać dane w swoim silniku graficznym na wiele sposobów – nazywamy je trybami. Tylko jeden tryb może być aktywny dla pojedynczej kostki przy pojedynczym wywołaniu funkcji *paint()*. Można za to używać różnych trybów sekwencyjnie, przy kolejnych wywołaniach funkcji odświeżającej ekran.

Występujące tryby wyświetlania [1]:

- *BG0* – wyświetla nieskończoną siatkę obszarów. Posiada wsparcie dla przesuwania obrazu we wszystkie strony o jeden piksel. Tryb ten jest bardzo szybki. Tryb wykorzystywany jest do wczytywania menu aplikacji oraz dużych, przesuwalnych map,
- *BG0_BG1* – łączy dwie, niezależne od siebie pod względem ruchu siatki obszarów na jednym obrazie. Ze względu na ograniczoną ilość pamięci, *BG1* może wyświetlać tylko 144 obszarów na siatce. To oznacza, że pozostałe 112 musi pozostać wolne. Wolne obszary są interpretowany jako przezroczyste. Siatka *BG1* posiada dodatkowy bit ustalający przezroczystość. Posiada trzy różne metody wypełniania maski oraz rysowania obrazu na *BG1*,
- *BG0_SPR_BG1* – modyfikacja poprzedniego trybu. Zamiast siatki obszarów, umożliwia nam wczytanie do ośmiu obrazów, których rozmiar jest większy niż 8x8 a wymiary są potęgą dwójki. Użycie tego trybu wymusza użycie sekwencyjnego pliku zasobów (*PinnedAssetImage*), co wydłuża obliczenia i wypełnia bardziej pamięć. Obraz o najmniejszym numerze identyfikacyjnym będzie wyświetlany zawsze na wierzchu,
- *BG2* – służy do manipulowania obrazem za pomocą obrotów i skalowania. Działania te pochłaniają dużo czasu obliczeń. Tryb ten powinien być używany do animacji wejścia i efektów specjalnych, ze względu na niską jakość obrazów po przekształceniu,
- *BG0_ROM* – tryb działający podobnie jak *BG0*, ale odczytujący zawartość pamięci stałej. Ponieważ pamięć ta służy tylko do odczytu, nie jesteśmy w stanie jej zmodyfikować. Obszary zasobów przetrzymywane w pamięci *ROM* zawierają czcionkę o stałej szerokości, która jest przydatna przy sprawdzaniu poprawności aplikacji. *BG0_ROM* może zostać użyty do wypisywania informacji oraz pasków postępu,
- *SOLID* – wyświetla jeden kolor z zakresu *RGB565* na całym ekranie. Prosty w obsłudze i wymagający mało pamięci oraz przesyłów radiowych,
- *FB32* – *FB32* (ang. *Framebuffer*) wykorzystuje przesył radiowy do transmisji kolejnych pikselów obrazu do bufora klatki zawartego w pamięci wideo. Takie działanie pozwala nam wyświetlić konkretne zestawy pikseli w czasie działania aplikacji, wymaga jednak dużej ilości pamięci. Wadą tego trybu jest niska rozdzielczość obrazu, który możemy przetwarzać w tym trybie. *FB32* pozwala nam przesłać obraz 32x32 z 16 bitami głębi koloru. Taki obraz będzie zajmował dokładnie połowę dostępnej pamięci (512B z 1kB). Pozostały obszar pamięci zawiera m.in. tabele skojarzeniowe pozwalające przekształcić kolor obrazu do

formatu *RGB565*. Obraz przesyłany w tym trybie zostanie powiększony razy 2 tak, aby zajmować całą powierzchnię wyświetlacza. Wykorzystuje klasę *FBDrawable* zawierającą funkcje udostępniające zapisywanie danych do pamięci wideo kostki,

- *FB64* – jest wariacją trybu *FB32*. Jego działanie jest podobne do poprzedniego trybu. Jeśli rozdzielczość obrazu jest ważniejsza niż głębia kolorów, możliwe jest użycie tego trybu. *FB64* przesyła obraz o wielkości 64x64 piksele z głębią kolorów na przestrzeni jednego bitu,
- *FB128* – kolejna wariacja trybu *FB32*. Przesyła obraz o rozdzielczości 128x48 z głębią kolorów zapisaną na jednym bicie. Jest to najbardziej wydajna metoda wyświetlania tekstu. Obrazy w tym trybie nie są skalowane,
- *STAMP* – tryb służący do szybkiego rysowania małej części sceny podczas kolejnych klatek animacji. Przeznaczony do efektów specjalnych. Działa wolniej od pozostałych trybów. Tryb interpretuje część pamięci wideo jako bufor klatki o dowolnej zmiennej wielkości. Wielkość bufora jest określona dwoma warunkami: Szerokość musi być liczbą parzystą oraz całkowita ilość użytych bajtów nie może przekroczyć 1536. Tryb udostępnia nam 16 bitów na zapis koloru.

2.1.4. Struktura pamięci

Pliki aplikacji, takie jak binarny plik wykonywalny oraz dane zapiasne przez aplikację, przechowywane są w głównej pamięci Flash znajdującej się w *Bazie Sifteo*. Dostęp do pamięci *Bazy* możemy uzyskać w czasie działania aplikacji poprzez klasę *StoredObject*. Podczas działania aplikacji, kod oraz dane zasobów są bez przerwy pobierane z pamięci bazy i wysyłane do kostek drogą radiową. Główna pamięć przechowuje pliki zasobów w takiej samej formie, w jakiej stworzone zostaly przez *stir*: dane w obiektach *AssetGroup* są skompresowane, a dane indeksów *AssetImage* są przechowywane w jednej z ich dostępnych postaci (image, flatImage, pinnedImage).

Strefy pamięci głównej Flash są objęte kronikowaniem (ang. *journaling*), czyli mechanizmem, który zapisuje zmiany czekające na zatwierdzenie w specjalnej strukturze, stanowiącej bufor danych. Dzięki tej strukturze, w wyniku nagłej przerwy w dostawie zasilania stracimy jedynie ostatnio zapisane dane. Taka struktura zapisu pozwala nam zachować spójność danych np. w przypadkach wyładowywania się baterii, co zdarzać się może często.

Każda z kostek posiada swoją pamięć Flash, o wielkości 4MB, w której przechowuje dane części obrazów plików zasobów. Razem z pamięcią wideo, o wielkości 1kB, są używane przez silnik graficzny do tworzenia kolejnych klatek do wyświetlenia. [1] Pamięć zasobów w kostkach jest nieulotna. To oznacza, że dane, które są zawarte w tej pamięci, pozostają tam w niezmienionej formie, aż do czasu nadpisania lub celowego wyczyszczenia. Podstawową jednostką pamięci w pamięci Flash kostki jest obiekt *AssetSlot*. W każdej kostce istnieje osiem

takich obiektów, z których na raz aplikacja może korzystać tylko z czterech. Klasa obiektów jest bardzo prostym kontenerem, który zawiera maksymalnie 24 grupy zasobów/4096 obszarów zasobów.

Aplikacja może w swoim kodzie zawrzeć instrukcje informujące system, że konkretna część zasobów powinna zostać wczytana przed przekazaniem kontroli do tej aplikacji. Taki mechanizm nazywamy *Bootstrapping*.

2.1.5. System obsługi zdarzeń

Z powodu braku wielowątkowości aplikacji (mamy do dyspozycji tylko jeden wątek w każdej aplikacj), obsługa zdarzeń została zaimplementowana jako mechanizm asynchroniczny. System działa w następujący sposób: zdarzenia są generowane przez czujniki, np. przyciśnięcie koski. Sygnał o wygenerowanym zdarzeniu zostaje odłożony do kolejki zdarzeń, która ma strukturę FIFO. Aby obsłużyć zdarzenie, w kodzie programu musi zostać wywołana, jawnie lub niejawnie, funkcja *yield()* która przestawia działania środowiska na tryb 'obsługi zdarzeń'. Są one teraz po kolei zdejmowane z kolejki. Następnie sprawdzane jest, jaka funkcja była zarejestrowana jako reakcja na to zdarzenie. Po sprawdzeniu, funkcja jest wywoływana w 'nasłuchującej' aplikacji. Jako alternatywa do tego systemu istnieje możliwość odpytywania czy zdarzenia już zaistniały. Jeśli podjętą co do obsługi zdarzeń decyzją jest wykorzystanie pierwszego systemu, musimy pamiętać o umieszczeniu w aplikacji widocznych wywołań funkcji *yield*() albo funkcji *paint*() która również wywołuje funkcję *yield*(), aby system miał czas obsłużyć zdarzenia w kolejce. Warto odnotować, że system wykrywa zmiany w sąsiedztwie kostki tylko po oddaniu kontroli przez aplikację przez funkcję *yield*(). Niektóre zadania, takie jak przesył radiowy danych lub odtwarzanie muzyki, posiadają wyższy priorytet niż uruchamiana aplikacja użytkowa. Wywoływanie funkcji *yield()* jest zalecane, aby zadania o niższym priorytecie, niż aplikacja użytkowa miały czas na swoje działania.

2.2. Elementy istniejącego już API

2.2.1. Opis modułów

Istniejące API jest podzielona na moduły. Każdy moduł zawiera klasy, które odpowiadają za poszczególne funkcje systemu. API składa się z następujących modułów [1]:

- *Array* - zawiera klasy *Array* i *BitArray* będące kontenerami,
- *Assets* - klasy służące ładowaniu, przetwarzaniu i nadzorowaniu plików opisu zasobów,
- *Audio* - interfejs dla miksera audio i odtwarzacza dźwięków,
- *Cube* - klasy wspomagające zarządzanie kostkami ich urządzeniami składowymi,

- *Event* - podsystem rejestracji zdarzeń dla programowania sterowanego zdarzeniami,
- *Filesystem* - zawiera klasy zapewniajace dostęp do składowanych danych, takich jak zapisane dane gier,
- *Macros* - makra użytkowe, logowania i skryptów,
- *Math* - działania matematyczne stało oraz zmiennoprzecinkowe, macierze oraz wektory,
- *Memory* - zawiera funkcje szybkiej kopii pamięci, wypełniania oraz operacji CRC,
- *Menu* - API wspólnego menu, opartego na przechyleniach kostek,
- *Metadata* - narzędzia do oznaczania plików binarnych ELF systemem klucz/wartość,
- *Motion* - podsystem wykrywający ruch,
- *String* - zawiera klasy formatujące ciągi znaków,
- *System* - udostępnia operacje dotyczące systemu jako całości,
- *Time* - klasy wspomagające operacje na czasie,
- *Video* - klasy podsystemu grafiki.

2.2.2. Moduł *Array*

Moduł „*Array*" zawiera w sobie szablony dwóch kontenerów tablic: *Array*<> i *BitArray*<>. Kontenery te posiadają funkcje dostępowe, zwracające liczebność oraz funkcje czyszczące przechowywane tablice.

Array przypomina klasę *Vector* z biblioteki STL C++. Zawiera tablicę do której możemy odnosić się poprzez nawiasy kwadratowe (symbole „[„ oraz „]") tak jak w klasycznych tablicach. Podobieństwo do wektora występuje przy funkcjach push_back() i pop_back(). *Array* pozwala nam usuwać pojedyncze elementy tablicy, przestawiając pozostałe, tak aby nie występowały puste przestrzenie wewnątrz tablicy. Jest skierowna na przetrzymywanie niezależnych od siebie elementów.

BitArray jest kontenerem zawierającym bity i manipulującym bitami. Posiada funkcje wykorzystujące operatory logiczne (and, or, not) oraz funkcje do ustawiania/zerowania bitów. Wykorzystywany jest przy niskopoziomowych działaniach na bitach. Składowane w nim są liczby i flagi.

2.2.3. Moduł *Assets*

Moduł „*Assets*" (pol. zasoby, pliki opisu zasobów) zawiera klasy obrazów (*AssetImage, FlatAssetImage, PinnedAssetImage*) i grupowania obrazów (*AssetGroup*), klasy ładowania plików opisu zasobów (*AssetSlot, AssetLoader, AssetConfiguration*) i klasy zasobów dźwiękowych (*AssetAudio, AssetTracker*).

Każdy z obrazów zawiera funkcje pozwalające na zarządzanie zawartym wewnątrz obrazem, funkcje informujące o własnościach obrazu takich jak wysokość czy ilość klatek w przypadku animacji oraz referencję na grupę w której znajduje się dany plik opisu. Klasy obrazów celowo nie tworzą hierachii. Obrazy są rzutowane jeśli podczas działania programu występuje taka konieczność. [1]

Obiekt obrazu „płaski" (*FlatAssetImage*) zawiera w sobie części obrazu które są ułożone w tablicy, jedna po drugiej, bez kompresji. To daje nam łatwy dostęp do części kosztem czasu wykoniania i zajętości pamięci.

Obiekt obrazu „przybitego" (*PinnedAssetImage*) zawiera w sobie sekwencyjnie ułożone w pamięci wszystkie części obrazu.

Ze względu na działanie na tak małej ilości pamięci, daleko posunięta optymalizacja obrazów jest konieczna. Obrazy są obsługiwane przez *Stir*, niskopoziomową aplikację transformującą obrazy do odpowiednich, zoptymalizowanych dla *Sifteo* form. Aplikacja działa na bardzo niskim poziomie. Jest to konieczne, aby jej operacje były precyzyjne a jej efekty zajmowały mało miejsca. To wymusza na środowisku aby ono również operowało na obrazach na niskim poziomie, czego efektem jest działanie na POD-ach. Dlatego też podczas przeglądania dokumentacji *Sifteo* natkniemy się na wiele struktur będących POD-ami, szczególnie w module służącym do przetwarzania plików opisu zasobów.

Oprócz klas obrazów, moduł zawiera klasy zarządzające plikami opisu zasobów: *AssetGroup* i *AssetSlot*. Te drugie mogą zawierać w sobie wiele grup plików opisu. Grupy służą do składowania plików opisu zasobów i ich części – właśnie tutaj mechanizmy optymalizacji pamięciowej umieszczają przetworzone z obrazów dane, które powinny być co najwyżej takie samej wielkości co suma rozmiarów wszystkie obrazów w grupie. Pożądanym wynikiem jest oczywiście jak najmniejszy rozmiar.

AssetLoader jest odpowiedzialny za zarządzanie ładowaniem plików do grup kostek. Grupą kostek mogą być kostki 'połączone' (*CubeSet::connected()*) lub kostki należące do podanej grupy. Posiada funkcje odpowiedzialne za sprawdzanie postępu ładowania danych. Funkcja ładowania danych do grup kostek może być wywoływana wiele razy w czasie działania aplikacji. Istnieje jeszcze druga klasa ładująca dane, *ScopedAssetLoader,* która posiada wbudowane w konstruktor i destruktor wywołania funkcji *start*() i *finish*() – funkcji służących do kontroli stanu przesyłu danych. Podstawowa klasa ładująca nie ma takich wywołań, ponieważ musi być kompatybilna z obiektami typu POD (np. Unie).

W module "Asset" istnieje również klasa wspomagająca pracę *AssetLoader*'a nazwana *AssetConfiguration*. Jest to klasa składująca w tablicy *Array* węzły *AssetConfigurationNode,* które stanowią połączenie pomiędzy grupą kostek a grupą plików opisu. Jedna klasa tego typu może zostać zaaplikowana do wszystkich kostek, lub wiele takich klas może zostać użytych jednocześnie by wczytać całkowicie różne dane na różne grupy kostek.

Ostatnimi klasami istniejącymi w tym długim module są *AssetAudio* i *AssetTracker*. Są to klasy służące do przechowywania odtwarzalnych dźwięków. *AssetAudio* służy do pojedynczego odtworzenia ścieżki, podczas gdy *AssetTracker* może zostać zapętlony, oszczędzając w ten sposób pamięć urządzenia.

2.2.4. Moduł *Cube*

Moduł odpowiada za kontrolę i wykrywanie stanu kostek. Zawiera typ wyliczeniowy *Side* (pol. Strona), określający konkretny bok kostki. Klasa *CubeID* zawiera metody udostępniające wszelkie informacje o kostce: jej sąsiadach, poziomie naładowania, odczyt akcelerometru.

CubeID jest kontenerem na kostkę i może zostać użyty by wprowadzać zmiany na skojarzonej z nim kostce. Klasa *CubeSet* jest używana do przechowywania kolekcji kontenerów kostek. Korzysta z *BitArray*.

Obiekty klasy *Neighborhood* odpowiadają obiektom klasy *CubeID*. Przedstawiają one stan całego sąsiedztwa dla kostki określonej przez powiązane *CubeID*. W tym module istnieje jeszcze klasa pomocnicza *NeighborID* która odpowiada za symulację miejsca w którym może być jakiś sąsiad danej kostki. Posiada metody pozwalające nam sprawdzić, czy miejsce sąsiada jest wypełnione i czym. Może być również rzutowane na *CubeID*.

2.2.5. Moduł *Filesystem*

Zawiera klasy informujące o właściwościach systemu. Klasa *FilesystemInfo* daje nam dostęp do informacji o zajętości pamięci urządzeń. Aby zebrać znaczące informacje, należy wywołać funkcję *gather()* przed odczytaniem jakiejkolwiek wartości. Klasa *Volume* stanowi reprezentanta zewnętrzengo obszaru pamięci zawierającego dane innej aplikacji. Poprzez tę klasę możemy przekazać kontrolę do innej aplikacji lub odczytać dane z osobno zarządzanego przedziału pamięci. Klasa *StoredObject* stanowi reprezentanta obiektu, który chcemy odczytać lub zapisać do strefy pamięci Flash *Bazy*. Obiekt reprezentowany przez tę klasę możemy również usuwać. Domyślnie, zapisujemy lub odczytujemy obiekt z pomocą tej klasy do strefy pamięci uruchomionej w tej chwili aplikacji. Obiekty ze stref pamięci innych aplikacji mogą być czytane, lecz nie mogą być nadpisywane.

Klasa *MappedVolume* reprezentuje obszar pamięci, który znajduje się w drugim poziomie pamięci Flash. W jednym momencie może istnieć tylko jeden *MappedVolume* – tworząc nowy, pozostałe tracą swoje referencje. Może zostać utworzony tylko dla obszarów pamięci, które posiadają odpowiedni plik *.elf.

2.2.6. Moduł *Math*

Biblioteka zawierająca stałę matematyczne – pi, wartości logarytmów oraz pierwiastków. Zawiera definicje wszystkich typów matematycznych – głównie wektorów liczb całkowitych i zmiennoprzecinkowych.

Biblioteka zawiera w sobie elementy niezbędne do działań matematycznych [1]:

- klasę macierzy afiinicznych do skalowania, przesuwania i rotacji wykorzystywanych w trybie renderowania BG2,
- klasę generatora liczb pseudolosowych,
- typy szablonów struktury wektorowych dwu- i trójelementowych,
- stałe matematyczne, takie jak Pi, logarytmy i pierwiastki,
- definicje typów wektorowych liczb całkowitych i zmiennoprzecinkowych (Vector2<float>, Vector3<uint8_T>),
- funkcje działań matematycznych, takich jak wartość absolutna *abs()*, obliczanie pierwiastków, logarytmów, zaokrąglania oraz funkcji trygonometrycznych,
- szablon funkcji porównujący dwie wartości podane w parametrze.

2.2.7. Moduł *Time*

Zawiera zbiór klas odpowiedzialnych za rejestrowanie i przetwarzanie czasu w aplikacji. Najważniejszą klasą jest klasa *SystemTime*. W tej klasie zapisywana jest dyskretna wartość zegara systemowego zliczającego nanosekundy informujące o czasie działania systemu od uruchomienia. Urządzenie nigdy nie powinno mieć problemu z przepełnieniem licznika. Zawiera ona funkcje *now()* oraz *optimeMS()* które informują nas o obecnym odczycie czasu. Obiekty tej klasy możemy ze sobą porównywać i odejmować, czego efektem będzie obiekt klasy *TimeDelta*. Klasa *TimeStep* implementuje funkcjonalność stopera. Mierzy ona interwały czasu i zawiera funkcję zwracającą ostatnią wartość pomiaru czasu. Klasa ta jest kontenerem dla klasy *TimeDelta*, która jest reprezentacją różnicy dwóch pomiarów czasu. *TimeDelta* zawiera funkcje informujące o ilośći klatek, które zostały wytworzone w reprezentowanym przedziale czasu, funkcje zwracające wartość przedziału czasowego w nano-, mili- lub sekundach oraz funkcje pomocnicze pozwalające określić, czy przedział czasu ma wartość pozytywną czy negatywną. *TimeDelta* może zostać skonstruowany z wartości zmiennoprzecinkowej której jednostką są sekundy, ale również milisekundy a nawet Hertze. Ostatnim narzędziem jest *TimeTicker*, który konwertuje strumień przedziałów czasowych w strumień dyskretnych cyklów procesora.

2.2.8. Inne moduły

Udostępniona poprzez *Sifteo* SDK biblioteka posiada wiele modułów. Moduły przedstawione do tej pory wymagały dokładnego omówienia z racji na ich rozmiar i funkcjonalność. Moduły przedstawione poniżej zostały zebrane w jednym miejscu, ponieważ wymagają mniej dokładnych wyjaśnień. [1]

- *Audio* – Zawiera klasy *AudioTracker* i *AudioChannel*. *AudioTracker* posiada funkcje pozwalające na swobodne kontrolowanie odtwarzania dźwięku (*start, setPosition, setVolume* etc) pochodzącego z *AssetTrackerów* opisanych w pkt. 2.4.2. *AudioChannel* jest obiektem odpowiadającym za miksowanie dźwięków. Jest w stanie odtwarzać obiekty typu *AssetAudio*. Format pliku to ADPCM lub PCM. Użycie klas typu *Tracker* jest zalecane do odtwarzania utworu który będzie się zapętlał, gdyż posiadają związane z tym optymalizacje,
- *Event* – Moduł składa się z przestrzeni nazw możliwych do zarejestrowania zdarzeń, oraz z trzech klas będących szablonami kolejek zdarzeń. Podstawową klasą jest *EventVector<>,* który pozwala nam ustawić funkcję nasłuchującą. Pozostałe dwie klasy, *GameMenuEventVector* oraz *NeighborEventVector* są podobnymi do *EventVector* kolejkami zdarzeń. Różnią się przeznaczeniem: pierwsza kolejka posiada zalecenie, aby istniał tylko jeden obiekt o tym typie, dla głównego menu. Druga kolejka to kolejka, która rejestruje zdarzenia, które dzieją się w sąsiedztwie danej kostki,
- *Macros* – Ten moduł zawiera makra, które wykorzystać można w projekcie. Wśród makr występują między innymi: funkcja wypisujaca informacje na konsoli *LOG,* zmienna przedstawiająca limit ilości fizycznych kostek oraz uruchamianie skryptów,
- *Memory* - Biblioteka zawierające funkcje kopiujące bloki pamięci w kawałkach po 8, 16 lub 32 bity przetwarzanych na raz danych oraz funkcje ustawiające bity w pamięci (również z szerokością 8, 16 lub 32 bity) lub czyszcząca bity w pamięci (funkcje *bzero()*). Oprócz funkcji manipulujących pamięcią biblioteka ta wyposażona jest również w funkcję obliczającą kod CRC 32 bitowy,
- *Menu* - Moduł zawiera jedną klasę – *Menu* – oraz pare funkcji pomocniczych służących do ustawienia początkowego ekranu menu oraz inicjalizacji menu,
- *Metadata* - Moduł dający możliwość opisu pliku binarnego za pomocą ikony, tytułu oraz przyporządkowania konkretnych identyfikatorów kostek,
- *Motion* - Podsystem udostępniający funkcje śledzenia zmiany położenia kostki. Moduł udostępnia klasy odpowiadające za rozpoznawanie gestów oraz odczytywanie ich ostatnich wystąpień. Udostępnia statysyki gestów oraz szablon klasy do składowania zarejestrowanych danych, *MotionBuffer<>*,

- *String* - Moduł zawiera funkcje wspierające działania na ciągach znakowych. Zawiera szablon klasy ciągu znakowego o podanej długości oraz klasy opakowujące klasę ciągu znaowego, obsługujące stałoprzecinkowy i heksadecymalny zapis liczb. Szablon jest wyposażony w dużą ilość funkcji operatorowych,
- *System* - Zawiera tylko jedną klasę, *System*, która udostępnia szeroki zakres funkcji zwracających informacje o systemie. Są to informacje takie jak: poziom baterii, stan gry, wersja systemu, czy obecnie uruchomiona aplikacja jest w trybie produkcyjnym. Klasa zawiera również funkcje zarządzające systemem, których zadaniem jest zakończenie uruchomionej w tej chwili aplikacji, rysowanie następnej klatki, oddanie kontroli nad procesorem oraz natychmiastowe wyłączenie urządzenia,
- *Video* - Ten moduł zawiera klasy odpowiedzialne za obsługę grafiki i wspomaganie silnika graficznego kostek. Można tu znaleźć typy wyliczeniowe zawierające tryb tworzenia grafiki i rotacje kostki. Oprócz tego znajdziemy tutaj klasy odpowiadające za kontrolę poszczególnych trybów renderowania grafiki (*BG0Drawable, BG1Mask, StampDrawable*), typy trybów wykorzystujących bufor klatek wewnątrz pamięci wideo (*FB32Drawable*) oraz struktury buforów grafiki i przekształceń kolorów. Klasy trybów renderowania są używane przez silnik graficzny w celu przetwarzania danych.

2.3. Analiza istniejących projektów

Wraz z SDK, *Sifteo* przygotowało projekty przykładowe, które demonstrują funkcjonalności API. [1] Projekty te przedstawiają podstawową konfigurację i metody implementacji. Podczas analizy problematyki brane były pod uwagę również projekty opracowane na uczelni. Wszystkie projekty zostały przeanalizowane pod kątem możliwych do wprowadzenia usprawnień. W dalszej części przedstawiono nazwę projektu oraz szczegółowo określone problemy w plikach źródłowych.

2.3.1. Projekty przykładowe z Sifteo SDK

Poniżej zostały przedstawione projekty przykładowe stworzone przez *Sifteo Inc.* Projekty te stanowią część *Sifteo* SDK. [1] Zawarte w nich przykładowe implementacje zagadnień zostały wykorzystane podczas projektowania rozszerzonego interfejsu programistycznego.

- *Accelchart*

  - wypełnienie metadanych w nowo utworzonym obiekcie jest niepotrzbnym narzutem kodu. Można je zamknąć w jednej funkcji, ale dalej powinna ona pozostać opcjonalna,
  - alokacja buforów, iteratora oraz inicjalizacja może zostać połączona,
  - określanie kolorów w mapie kolorów powinno być bardziej przejrzyste,
  - rysowanie linii wymaga znajomości trybu graficznego silnika graficznego.
- *Assetslot*
  - alokacja obiektów klasy *AssetSlot* może zostać zakryta,
  - obiekty typu *AssetSlot* muszą być ręcznie zarządzane przez użytkownika,
  - wywołanie menu posiada niezrozumiałe parametry, bufor wideo musi być ręcznie zarządzany przez użytkownika.
- *Connection*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - zmienne globalne loader i config zarządzane przez użytkownika,
  - rysowanie postępów za pomocą BG0_ROM zarządzane przez użytkownika,
  - wyodrębnienie silnika graficznego za pomocą indeksowanego pola bufora,
  - wypisanie tekstu przez wyodrębnioną zmienną i typ trybu.
- *Mandelbrot*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - ręczne zarządzanie rysowaniem w trybie *SOLID, STAMP, BG0_ROM,*
  - ręczne tworzenie kolorów przez wartości szesnastkowe.
- *Membrane*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - ręczne zarządzanie klasami loader, config i trybem graficznym,
  - dostęp do obrazów typu *sprite* poprzez bufor graficzny.
- *Menudemo*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - ręczne zarządzanie klasą bufora graficznego.
- *Sensors*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - ręczne zarządzanie klasą bufora graficznego oraz czyszczenie pamięci gdy połączona zostanie nowa kostka,
  - wypisywanie tekstu za pomocą bufora graficznego,
  - rysowanie linii.
- *Stampy*
  - wypełnienie metadanych w nowo utworzonym obiekcie.

2.3.2. Projekty wykonane na uczelni

Projekty te zostały wykonane przez członków kadry akademickiej Politechniki Śląskiej. Wykorzystane zostały podczas określania wymagań rozszerzonego interfejsu programistycznego. Poniżej wypisane zostały możliwe ulepszenia wyróżnione po analizie projektów.

- *Floatingmobiletech*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - ręczne zarządzanie klasą bufora graficznego i inicjalizacja buforów wideo,
  - ustawienie początkowego tła oraz logo,
  - odczyt pomiaru z sensorów,
  - przenoszenie obrazu z kostki na inną kostkę.
- *Memory*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - ręczne zarządzanie klasą bufora graficznego i inicjalizacja buforów wideo i klasy *AssetSlot*,
  - dodawanie nasłuchiwania do nowej kostki,
  - inicjalizacja czujników do odczytu ich pomiarów.
- *Pgum-cubes*
  - wypełnienie metadanych w nowo utworzonym obiekcie,
  - ręczne zarządzanie klasą bufora graficznego i inicjalizacja buforów wideo,
  - ustawienie tła i logo,
  - skomplikowane odczyty aktualnego czasu,
  - funkcje pseudogenerujące.

2.4. Określenie wymagań

W wyniku analizy tematu pracy, w celu osiągnięcia wyznaczonego celu, zdefiniowano następujące wymagania funkcjonalne:

- uproszczenie tworzenia kodu źródłowego nowych projektów polegające m.in. na:
  - skróceniu wywołań funkcji, zmniejszeniu liczby lub zmiany typu parametrów,
  - zamknięciu grupy wywołań funkcji o wspólnym celu w jednej klasie osiągającej ten sam cel, za pomocą wywołań tych samych lub podobnych funkcji (enkapsulacja i hermetyzacja działań).
- podniesienie poziomu abstrakcji aplikacji, co oznacza że jego użycie powinno być bardziej intuicyjne dla użytkownika implementującego projekt, bardziej klarowne i przystępne do analizy,

- zastosowanie modułowości, czyli podział oraz enkapsulacja implementacji funkcjonalności pomiędzy klasy,
- zastosownie drobnoziarnistości podziału funkcjonalności API. Analiza wewnętrzna API nie powinna stwarzać problemów, niejasności i zbędnych zawiłości, tak aby w przyszłości osoby zainteresowane mogły wprowadzać swoje modyfikacje posiadając jak najlepsze zrozumienie przebiegu działania wszystkich klas,
- zastosowanie wzorców projektowych, które są sprawdzonym i znanym pozostałym programistom standardem – nawet dla tych, którzy wcześniej nie mieli styczności z kostkami *Sifteo,*
- zwiększenie liczby komentarzy, które powinny przedstawić przeznaczenie klas dla użytkownika, który pragnąłby pomóc rozwijać opisywaną w tej pracy technologię,
- przygotowanie przykładów dydaktycznych odpowiadających funkcjonalnie projektom przykładowym stworzonym przez *Sifteo* oraz podczas zajęć Politechniki Śląskiej. Projekty zostaną wykorzystane aby uwydatnić zmiany poczynione w API i poprawę w czytelności kodu. Treść projektów zostanie porównana ze sobą i wskazane zostaną miejsca w których zauważyć można poprawę,
- opracowanie nowych projektów przykładowych, demonstrujących programistom co można zrobić z API i jak ono działa,
- opracowanie testów jednostkowych.

2.4.1. Rozpoznane przypadki użycia

W tym rozdziale przedstawione są przypadki użycia, które wyodrębnione zostały po analizie wymagań funkcjonalnych. Podczas analizy uwzględniona została również architektura istniejącego API oraz problemy i usprawnienia widoczne w kodzie źródłowym stworzonych wcześniej projektów:

- programista chce wyświetlić prosty komunikat na wyświetlaczu pojedynczej kostki,
- programista chce przenieść obraz wyświetlany na jednej kostce na drugą,
- programista chce użyć zestawu obrazów na wielu kostkach,
- programista chce odróżniać kostki od siebie,
- programista chce wyświetlić obraz na kostce,
- programista chce wyspecyfikować tło które wyświetla kostka,
- programista chce przygotować ekran ładowania aplikacji,
- programista chce opisać swoją aplikację, specyfikując jej tytuł, ikonę oraz opis aplikacji.

2.5. Podnoszenie abstrakcji API

2.5.1. Obiekty POD

POD (ang. „*Plain Old Data object*”, pol. Obiekt zawierający zwykłe, stare dane) jest oznaczeniem struktury danych która jest możliwa do skopiowania przez użycie funkcji *memcpy* w taki sposób, by kopia mogła być przetworzona i użyta w taki sam, nienaruszony sposób co oryginał. [6] PODy są niskopoziomowymi strukturami. Stworzone zostały w celu kompatybilności języków wysokiego poziomu (np. C++) z językami niższego poziomu np. zwykłym C. Znajdują zastosowanie tam, gdzie wymuszone jest programowanie niskopoziomowe m.in. na systemach wbudowanych. W takich systemach używamy programowania na niskim poziomie abstrakcji aby jak najbardziej oszczędzać miejsca w pamięci oraz czas działania procesora. [9] Systemy wbudowane posiadają znacznie większe ograniczenia w kwestii posiadanych parametrów sprzętowych, gdyż na ogół są to urządzenia o małym rozmiarze. Aby struktura mogła być nazwana PODem, musi spełniać dwa warunki: [7] pierwszy, warunek trywialności, zapewnia programistę, że struktura nie wykorzystuje funkcji wirtualnych, alokowanej pamięci i wskaźników na pamięć dynamiczną. Takie struktury przestaną działać poprawnie po zastosowaniu funkcji *memcpy* do obiektu. Drugą zasadą jest prostota struktury. Ta zasada zapewnia nas, że wszystkie składniki struktury posiadają ten sam poziom dostępu.

2.5.2. Pojęcia enkapsulacji i hermetyzacji

Enkapsulacja oznacza zamknięcie funkcjonalności w pojedynczej klasie, bez wykorzystania klas zewnętrznych. Klasa taka może korzystać z obiektów klas które zawiera. Całe działanie powinno jednak zostać wewnątrz tej klasy, tak aby nie istniała konieczność informowania klas zewnętrznych do tej klasy o kontekście działań aby mógł zostać wypracowany wynik. [4][5]

Hermetyzacja oznacza zamknięcie funkcjonalności w pojedynczej klasie, tak aby klasy zewnętrzne nie potrzebowały nigdy informacji o tym, jak dana klasa działa wewnątrz i w jaki sposób osiąga wynik swoich działań.

Hermetyzacja dotyczy widoczności danych, ich bezpieczeństwa wewnątrz systemu i podziału obowiązków wewnątrz aplikacji. Natomiast enkapsulacja dotyczy przepływu kontroli. [8] Enkapsulacja jest obserwacją ze strony danej klasy, a hermetyzacja jest obserwacją ze strony środowiska klasy.

2.5.3. Wzorce projektowe

Wzorce projektowe są szablonami dobrych praktyk tworzenia oprogramowania, które należy powielać w stosownych, przeanalizowanych przypadkach. Każde zastosowanie wzorca projektowego powinno być uzasadnione, aby programista był świadomy jak wzorzec wpasowuje się w system i jak w nim funkcjonuje. [2][3] Stosując wzorce projektowe nie powielamy tylko dobrego, dobrze udokumentowanego działania modułu. Zyskujemy również modułowość aplikacji oraz podział obowiazków wewnątrz niej na klasy. Zwiększamy ziarnistość aplikacji. Poprzez podział obsługi funkcjonalności aplikacji między funkcjami zyskujemy również odseparowanie działania obiektów od siebie – co upraszcza analizę systemu i pozwala na częstsze wykorzystywanie gotowych już klas w innych projektach.

Wzorce są zbiorem szablonów klas o specyficznych właściwościach, które porządkują nam działanie aplikacji. Są przetestowane i szeroko znane w środowisku. Właściwie zastosowane są w stanie zaoszczędzić wiele czasu w fazie projektowania, implementacji i testowania, oraz ułatwiają poznanie systemu programiście, który nie ma doświadczenia z naszym systemem. Sam fakt znajomości ze wzorcami projektowymi pozwala mu trafnie przewidywać możliwe połączenia w sytemie w który jest wdrażany. Przez to wzorce projektowe są tak ważne podczas projektowania systemów informatycznych.

Nieprzemyślane kopiowanie wzorców w każdej możliwej sytuacji doprowadza do sytuacji, w których osiągamy efekty odwrotne do oczekiwanych. [3] Na przykład - systemy zamiast być bardziej czytelne stają się tylko bardziej zawiłe w analizie działania. Dzieje się tak, ponieważ nie istnieje jeden wzorzec pasujący do każdej sytuacji. Wzorce nachodzą na siebie w zakresie swoich funkcjonalności, przez co nie istnieje uniwersalny wzorzec projektowy spełniający wszystkie możliwe wymagania. Wybór wzorca jest bardzo odpowiedzialnym zadaniem. Dlatego programista powinien rozumieć powody użycia wzorca i możliwości jakie nam daje.

2.5.4. Wzorzec projektowy „Fasada”

Wzorzec ten określa strukturę, która stanowi interfejs pomiędzy klientem lub aplikacją a systemem lub podsystemem o skomplikowanym, niejasnym lub złożonym działaniu. [2] Fasada jest zazwyczaj reprezentowana przez pojedynczą klasę, która zawiera w sobie funkcjonalności jakiegoś systemu lub podsystemu. Elementy takiego systemu posiadają złożone, wzajemne zależności. Te zaś powodują, że osiągnięcie pożądanego efektu przez klienta lub aplikację kliencką wymaga kompletnej znajomości samych elementów systemu, wspomnianych wcześniej wzajemnych zależności elementów oraz skomplikowanej implementacji. Każda zmiana w systemie powodowałaby również zmianę w każej aplikacji klienckiej, co wymagałoby ciągłych aktualizacji, pochłaniałoby czas i tworzyło koszta.

Aby usprawnić implementację, skrócić cykl tworzenia oprogramowania oraz uniknąć nadmiarowych kosztów, stosuje się Fasady. Fasady zawierają w sobie proste w wywołaniu metody, które wykonują odpowiednie działania na danym systemie aby uzyskać pożądany efekt. W takiej sytuacji, gdy zmienią się elementy systemu, zmiany wprowadzać trzeba tylko raz – w Fasadzie.

Fasady mogą mieć szeroki lub wąski zasięg funkcjonalności. Wiele Fasad może istnieć wewnątrz systemu lub udostępniać dane systemu zewnętrznym podmiotom.

2.5.5. Wzorzec projektowy „Fabryka”

Wzorce projektowe posiadają określone funkcjonalności, które zostały dobrane do nich według określonego zestawu zasad. Zasady te zostały określone w celu zwiększenia czytelności i prostoty wzorców, tak aby stanowiły one dobre przykłady zastosowań. Jedną z tych zasad jest oddzielenie klas manipulujących na obiektach od klas przetwarzających i wykorzystujących obiekty. Zadaniem pierwszej grupy klas jest tworzenie, inicjalizacja i usuwanie obiektów. [2] Zadaniem drugiej jest przetwarzanie obiektów, aby spełnić określone funkcjonalności.

Fabryka należy do pierwszej grupy klas, czyli do klas manipulujących na obiektach. Posiada ona metody do tworzeniaz nowych obiektów. Aby użyć tego wzorca, tworzymy obiekt tej klasy, po czym wywołujemy jego metodę tworzącą obiekt. Referencja na obiekt zostanie nam zwrócona jako rezultat funkcji. Taka implementacja dostarcza nam rozwiązań wielu problemów – na przykład klasy dziedziczące mogą zmienić wirtualną/abstrakcyjną funkcję tworzenia obiektów. W takiej sytuacji nazwa funkcji pozostaje ta sama, a otrzymać możemy obiekt z grupy typów implementujących dziedziczenie. Jest to przydatne, gdy nie musimy dokładnie wiedzieć, jaki typ ma stworzony przez nas obiekt.

2.6. Narzędzia wykorzystane w implementacji

Notepad++

Otwarte, wielofunkcyjne narzędzie, zdolne wspomagać pracę programisty w szerokim zakresie. Obsługuje podświetlanie składni mnóstwa języków programowania, pozwala na szukanie i zamianę w plikach oraz udostępnia skróty edycyjne zdolne edytować wiele wierszy na raz. Bardzo dobrze służy podczas pisania skryptów *make* oraz analizowania plików *.lua, które to pliki są niezbędne do wczytywania plików opisu zasobów do aplikacji *Sifteo*.

Lua

Dynamicznie typowany język programowania, używany w systemie *Sifteo* do wczytywania plików zasobów do grup zasobów. Twórcy tego języka, przy jego definiowaniu, kierowali się prostotą implementacji funkcjonalności. W efekcie otrzymano prosty w użyciu, elastyczny język programowania.

Sifteo SDK

SDK zawiera większość niezbędnych narzędzi, bez których praca nad oprogramowaniem dla kostek *Sifteo* byłaby bardzo uciążliwa. Należy tutaj wymienić:

- *Stir* - narzędzie do wykrywania wspólnych części obrazów, które wykonuje tym samym bardzo ważne prace nad optymalizacją pamięciową i przesyłową w systemie,
- *Slinky* – narzędzie linkujące, wprowadzające dodatkowe optymalizacje,
- *Siftulator* – wielofunkcyjny symulator środowiska *Sifteo* bez którego bardzo trudno byłoby wprowadzać kolejne iteracje aplikacji oraz zajęło by to na pewno o wiele więcej czasu,
- kompilator C++ ARM *Clang* – przetwarzający kod C++ na kod maszynowy zrozumiały dla kostek *Sifteo.*

Microsoft Visual Studio Community 2015

Podczas prac nad API korzystałem również z darmowej wersji Visual Studio firmy Microsoft. Visual Studio to IDE, czyli zintegrowane środowisko tworzenia oprogramowania, w które jest wbudowane mnóstwo przydatnych narzędzi. Można je również zintegrować z systemami kontroli wersji, takimi jak *Git*. Wybrałem to narzędzie do pracy, gdyż posiada ono bardzo dobre i intuicyjne środowisko tworzenia oprogramowania w języku C++. Opócz tego mam z nim duże doświadczenie i tworzenie w nim jest dla mnie wygodne. Visual Studio udostępnia szereg przydatnych funkcji, takich jak na przykład wspomagane tworzenie klas.

3. Projektowanie

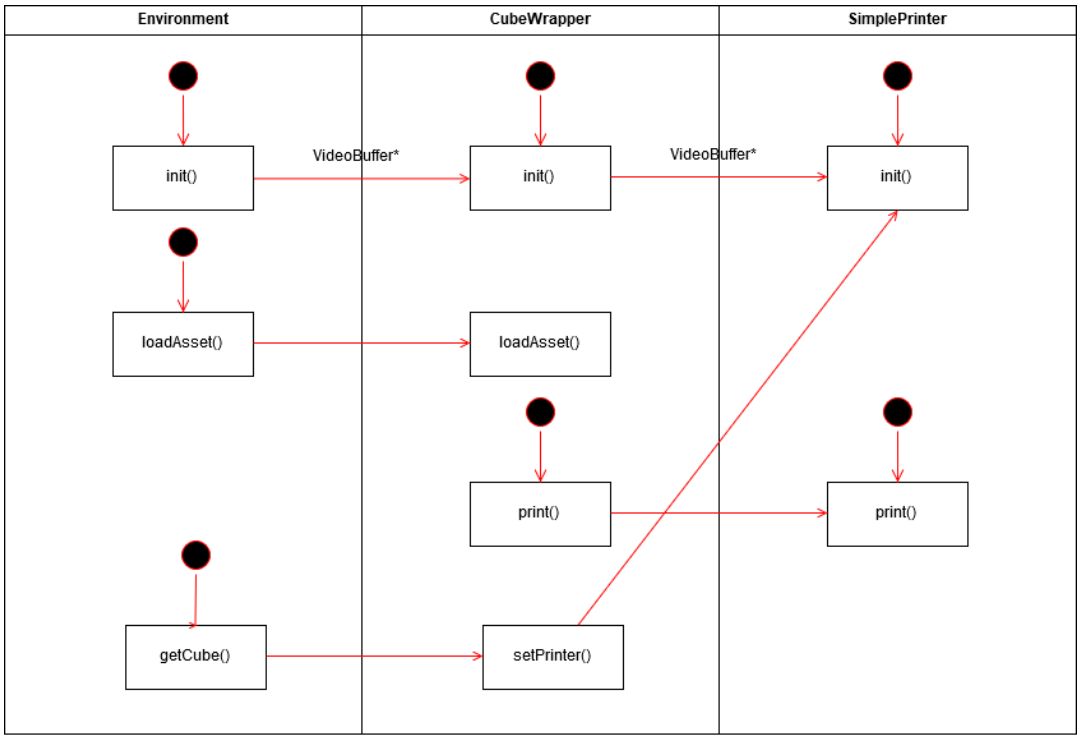
W tym rozdziale omówiony został etap projektowania API. Etap ten wykorzystywał dane uzyskane z analizy technologii i istniejących materiałów, takich jak projekty przykładowe *Sifteo* oraz dokumentacja kostek. W czasie projektowania, powstały dwa diagramy:

- sekwencji, przedstawiający przepływ sterowania wewnątrz interfejsu,
- klas, przedstawiający wzajemne zależności pomiędzy klasami interfejsu, oraz istniejącymi już klasami interfejsu *Sifteo*.

3.1. Diagram sekwencji API

Diagram na rysunku 3.1 przedstawia przepływ sterowania wewnątrz opracowanego interfejsu. Podczas etapu projektowania wyróżnione zostały trzy klasy:

- *Environment*, która jest punktem odniesienia i korzeniem interfejsu. Jej zadaniem jest bezpośrednie udostępnienie funkcjonalności,
- *CubeWrapper*¸ która utożsamiana jest z pojedynczą kostką *Sifteo* i reprezentuje możliwości tej kostki,
- *SimplePrinter*, wyspecjalizowana klasa odpowiedzialna głównie za wypisywanie tekstu na ekran oraz jego formatowanie.



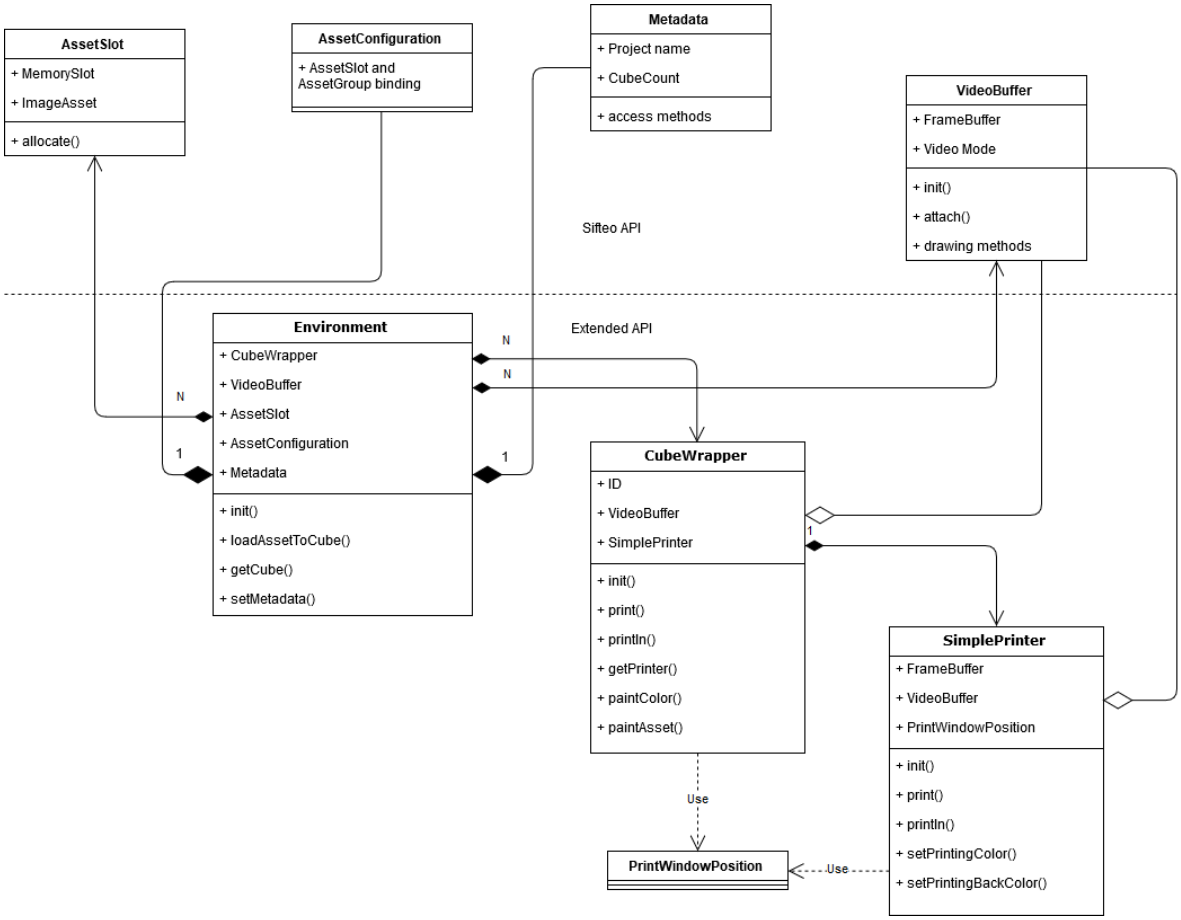
Rysunek 3.1 Diagram przepływu sterowania wewnątrz interfejsu

Obiekty porozumiewają się między sobą swobodnie i istnieje wiele dróg dostępu pomiędzy obiektami. Obiekt klasy *Environment* powinien być globalnie dostępny, bowiem z niego można dostać się w każde pozostałe miejsce API. Obiekt ten odpowiada globalnemu stanowi systemu. Obiekty klasy *CubeWrapper* odpowiadają poszczególnym kostkom obecnym w systemie.

3.2. Zależności między klasami

Na schemacie 3.2 przedstawione zostały podstawowe zależności pomiędzy klasami oryginalnego, podstawowego API *Sifteo* oraz rozwiniętego w ramach tej pracy rozszerzonego interfejsu programistycznego. Powyżej przerywanej linii umieszczone zostały klasy pochodzące z *Sifteo* API. Poniżej linii znajdują się klasy rozszerzonego interfejsu.

Podczas projektowania API uwzględniona została głównie klarowność zastosowanego rozwiązania oraz spójność logiczna interfejsu.



Rysunek 3.2 Diagram klas interfejsu

Schemat przedstawia kaskadowe połączenie trzech klas API, z których każda kolejna klasa jest coraz bardziej wyspecjalizowana.

Klasa *Environment* przeznaczona została do przetrzymywania wszystkich informacji związanych z globalnym zasięgiem projektu. Takie informacje to np. dane przechowywane w klasie *Metadata* opisujące tytuł projektu oraz ilość wykorzystywanych urządzeń. Przechowuje ona obiekty *CubeWrapper*. Obiekty klasy *Environment* zajmują się inicjalizacją danych i kontrolą przebiegu aplikacji. Umożliwiają również na ładowanie grup plików opisu zasobów do kostek.

Obiekty klasy *CubeWrapper* reprezentują pojedyncze kostki używane w aplikacji. Obiekty wyposażone zostały w funkcje:

- wypisujące ciągi znaków na ekran,
- zmieniające wyświetlaną grafikę na jednolity kolor lub obraz,
- udostępniające obiekt *SimplePrinter*.

Klasa *SimplePrinter* pozwala nam tworzyć obiekty zdolne do wypisywania ciągów znakowych na ekran kostki. Dzięki zastosowanym wewnątrz rozwiązaniom, pozwala ona na szereg zabiegów i modyfikacji tekstu. Aby działać poprawnie, obiekty tej klasy wymagają, aby wraz z plikiem *SimplePrinter.h* dołączony został także nagłówek *font_data.h*.

Przedstawiona struktura stanowi szkielet interfejsu. Dzięki prostej i zrozumiałej strukturze, rozwijanie interfejsu w przyszłości nie powinno stanowić problemu.

4. Implementacja

4.1. Przykładowy kod aplikacji

4.1.1. Wypisywanie tekstu na ekran

Opracowany został element biblioteczny zdolny do wypisywania tekstu na ekran. Klasa zawierająca tą funkcjonalność nazywa się *SimplePrinter*. Obiekty tej klasy są zdolne do wypisywania tekstu podanego przez użytkownika w kodzie projektu. Wypisywanie tekstu tą metodą nie wymaga posiadania obrazów liter ani plików opisu zasobów. Prawie cały proces jest wykonywany automatycznie, użytkownik musi tylko utworzyć obiekt klasy *SimplePrinter* i wywołać funkcję, podając jednocześnie jako parametr tekst który ma zostać wyświetlony.

Podany tekst może zawierać dowolną ilość znaków z zakresu symboli ASCII. Inne symbole powodują niepożądane zachowania (najczęściej błąd aplikacji). Obiekty tej klasy automatycznie wykrywają przepełnienie linii tekstu znakami oraz przepełnienie okna wypisywania liniami tekstu, co powoduje przeniesienie pozostałego ciągu znaków do nowej linii w pierwszym przypadku oraz zawinięcie wypisywanej linii z powrotem na początek okna w drugim.

Klasa *SimplePrinter* wykorzystuje tryb graficzny FB128, który pozwala nam na jednoczesne wyświetlenie obrazu na około 2/5 długości wyświetlacza LCD. Z tego powodu wewnątrz implementacji wyróżnione zostały trzy obszary („okna”) wyświetlacza:

- góra (granice okna wynoszą 0 i 48 pikseli wysokości),
- środek (granice okna wynoszą 40 i 88 pikseli wysokości),
- dół (granice okna wynoszą 80 i 128 pikseli wysokości).

Użytkownik podaje przy inicjalizacji obiektu klasy *SimplePrinter* który obszar ma zostać wykorzystany aby wypisać tekst. Użytkownik może używać wielu obiektów klasy *SimplePrinter* w jednym projekcie. Obiekty te mogą manipulować tym samym obszarem wyświetlacza.

```
            if (charInLineCount > maxCharInLine)// safety check against too many characters
in a single line
            {
                charInLineCount = 1;
                lineCount++;
                currentPosition.x = startPosition.x;
                currentPosition.y += 8;
            }

            if (lineCount > maxLinesInWindow)       // safety check against too many lines
to draw
```

```
        {
            currentPosition.y = startPosition.y;
            lineCount = 1;
            charInLineCount = 1;
        }

         // Specifics of our font format
        uint8_t index = ch - 32;
        LOG("SimplePrinter: index = %d\n", index);
        const uint8_t *data = font_data + (index << 3) + index;
        LOG("SimplePrinter: *data = %d\n", *data);
        uint8_t escapement = *(data++);
        LOG("SimplePrinter: escapement = %d\n", escapement);
        const Int2 size = {8, 8};
        LOG("SimplePrinter: X: %d, Y: %d\n", currentPosition.x, currentPosition.y);
        fb->bitmap(currentPosition, size, data, 1);
        currentPosition.x += escapement;
}
```

Listing 4.1.1.1. Wykrywanie granic ekranu oraz wyświetlanie tekstu

Liczba wyświetlanych na raz znaków obarczona jest ograniczeniami powierzchni wyświetlacza. Po wielu testach ustalono, że maksymalna ilość dowolnych znaków w pojedynczej linii wynosi 25. Wewnątrz każdego okna o wysokości 48 pikseli można wypisać jednocześnie 6 linii tekstu.

Klasa *SimplePrinter* została wyposażona w dwie metody:

- *print*() – funkcja przyjmująca ciąg znaków jako parametr, który zostanie wyświetlony na ekranie (znaki z przedziału znaków ASCII),
- *println*() – rozszerzona funkcja wypisująca, zakańczajaca każde wypisanie za jej pomocą znakiem nowej linii.

```
        switch (ch)
        {
            case '\t':
            {
                LOG("SimplePrinter: detected a tabulation %d %d %d\n",
currentPosition.x, startPosition.x, size.x);
                if (currentPosition.x >= startPosition.x + size.x - 8) // safety
check against tab count over 6 (more than a single line can handle)
                {
                    charInLineCount = 0;
                    lineCount++;
                    currentPosition.x = startPosition.x;
                    currentPosition.y += 8;
                }

                for (unsigned nextTabulationAt = 20; nextTabulationAt <
startPosition.x + size.x; nextTabulationAt += 20) // loop that checks tabulation
slots
                {
                    LOG("SimplePrinter: Inside the loop, x: %d, nextTabulationAt
value: %d\n", currentPosition.x, nextTabulationAt);
                    if (currentPosition.x < nextTabulationAt)   // if we enter this,
slot is found
                    {
                        LOG("SimplePrinter: Setting the x (current %d) to %d
pixels\n", currentPosition.x, nextTabulationAt);
                        currentPosition.x = nextTabulationAt;
                        charInLineCount = nextTabulationAt/5;
```

```
                        LOG("SimplePrinter: Current x: %d pixels\n",
currentPosition.x);
                        break;
                    }
                }
                continue;
            }
            case '\n':
            {
                LOG("SimplePrinter: detected a newline\n");
                charInLineCount = 0;
                lineCount++;
                currentPosition.x = 0;
                currentPosition.y += 8;
                continue;
            }
}
```

Listing 4.1.1.2. Wykrywanie i obsługa znaków nowej linii i tabulacji

Obie metody zostały przygotowane do przyjmowania podstawowych znaków formatujących tekst – czyli znaku nowej linii „\n” oraz tabulacji „\t”. Klasa *SimplePrinter* posiada również wbudowaną możliwość zmiany koloru tła oraz tekstu wewnątrz manipulowanego przez dany obiekt obszaru.

```
    void paintPrintingColor(RGB565 newBGColor)
    {
        localVideoBuffer->colormap[1] = newBGColor;
        System::paint();
        System::finish();
    }

    void paintPrintingBackColor(RGB565 newBGColor)
    {
        backgroundColor = newBGColor;
        System::paint();
        System::finish();
        refreshModeAndWindowPosition();
    }
```

Listing 4.1.1.3. unkcje obsługujące kolor tekstu oraz jego tła

4.1.2. Wyświetlanie obrazu na ekranie

Załadowanie i wyświetlenie obrazu na ekranie kostki jest możliwe jedynie poprzez obiekt klasy *Environment*, ponieważ obiekty tylko tej klasy kontrolują ładowanie zasobów do pamięci kostek. Jeśli chcemy wyświetlić wcześniej wczytany obraz, możemy użyć funkcji z obiektu klasy *CubeWrapper*.

```
    int loadAssetToCube(unsigned CubeID, unsigned assetSlotNumber, AssetGroup assetgroup,
AssetImage image)
    {
        if (CubeID > 2) return 1;    //at this moment, we have only 3 cubes active: 0, 1
and 2
        CubeSet cubes(CubeID);
        switch (assetSlotNumber)
        {
        case 0:
```

```
            assetConfiguration.append(assetSlot0, assetgroup);
            break;
        case 1:
            assetConfiguration.append(assetSlot1, assetgroup);
            break;
        case 2:
            assetConfiguration.append(assetSlot2, assetgroup);
            break;
        default:
            return -1; //error code
        }
        ScopedAssetLoader loader;
        loader.start(assetConfiguration, cubes);
        loader.finish();
        videoBufferArray[CubeID].initMode(BG0);
        videoBufferArray[CubeID].attach(CubeID);
        videoBufferArray[CubeID].bg0.image(vec(0, 0), image);
        System::paint();
        System::finish();
        return 1;
    }
```

Listing 4.1.2. Ładowanie oraz polecenie wyświetlenia obrazu na wyświetlaczu

Metoda, która jest odpowiedzialna za ładowanie i wyświetlenie obrazu, nazywa się *loadAssetToCube()*. Przyjmuje ona cztery parametry:

- numer porządkowy kostki na której grupa obrazów ma być wyświetlona,
- numer porządkowy obiektu klasy *AssetSlot*, czyli numer porządkowy sekcji pamięci, do której ma być wczytana grupa obrazów,
- nazwę grupy plików opisu zasobów,
- oraz zmienną reprezentującą obraz, który chcemy załadować.

Metoda wczytuje grupę obrazów poprzez obiekt *assetConfiguration*, który utworzony został w obiekcie *Environment*. Poprzez tę funkcję następuje również wiązanie grupy obrazów z sekcją pamięci w kostce. Po wczytaniu grupy obrazów do pamięci, metoda zleca buforowi graficznemu (przyporządkowanemu do wskazanej przez nas kostki) aby wyświetlił wskazany obraz w trybie BG0.

Metoda kończy swoje działanie po wywołaniu funkcji *paint*() i *finish*() z biblioteki systemowej *Sifteo*. Dzięki temu wymuszamy dokończenie rysowania na ekranie kostki przez naszą metodę, zyskując pewność że nasz obraz będzie widoczny na kostce przez zauważalną chwilę. W innym wypadku, nasz obraz mógłby pozostać bardzo szybko nadpisany przez następne metody zlecone przez użytkownika.

Metoda oddaje sterowanie, jednocześnie zwracając prostą, całkowitą wartość kontrolną. Wartość 0 oznacza poprawne wykonanie wszystkich instrukcji zawartych w metodzie. Wartość 1 oznacza niespodziewany błąd (np. niepoprawny numer porządkowy w parametrze).

5. Specyfikacja zewnętrzna

Aplikacja była rozwijana z wykorzystaniem symulatora technologii kostek *Sifteo*, który jest częścią *Sifteo* SDK. Aby uruchomić projekt w symulatorze:

5.1. Instalacja SDK i interfejsu

Aby rozpocząć pracę z SDK i rozszerzonym interfejsem programistycznym, należy postępować zgodnie z poniższą instrukcją:

1. Jeśli *Sifteo* SDK nie jest jeszcze zainstalowane na maszynie, może być pobrane ze strony: https://github.com/sifteo/thundercracker,
2. Pliki powinny zostać rozpakowane do jednego, dowolnego folderu, wspólnego dla wszystkich plików.
3. Jeśli Extended API nie jest jeszcze zainstalowane na maszynie, może być pobrane ze strony: https://github.com/krzykun/sifteo-api-extended,
4. Pliki nagłówkowe z EAPI powinny być umieszczone wewnątrz projektu, w którym będą użyte.

Aby korzystać z rozszerzonego interfejsu programistycznego, nie jest konieczna instalacja żadnych dodatkowych komponentów. Pliki rozszerzonego interfejsu programistycznego wymagają tylko dołączenia pliku nagłówkowego „*Environment.h”* za pomocą standardowej dyrektywy *#include* w pliku *main.cpp* który posiada każdy projekt *Sifteo.*

Listing 5.1. Przykładowe dołączenie pliku nagłówkowego opracowanej biblioteki:

```
#include <sifteo.h>
#include "assets.gen.h"

#include "Environment.h"

using namespace Sifteo;
```

5.2. Tworzenie nowego projektu

Aby utworzyć nowy projekt wykorzystujący interfejs *Sifteo*, należy postępować zgodnie z krokami opisanymi poniżej:

1. Najprostszym sposobem na utworzenie nowego projektu jest skopiowanie jednego z szablonowych projektów z folderu *examples*. Aby skorzystać z rozszerzonej biblioteki, zalecane jest skopiowanie projektu *Hello World* do nowego katalogu,
2. Po skopiowaniu projektu, należy zmienić nazwę folderu z projektem oraz nazwę projektu w pliku *Makefile*, który znajduje się wewnątrz projektu,

3. Jeśli pliki nagłówkowe nie są jeszcze dołączone, należy zrobić to za pomocą dyrektywy *#include* w taki sposób, jaki ukazano na *Listingu 5.1*.

Jeśli użytkownik nie chce kopiować żadnego z obecnych projektów, możliwe jest stworzenie projektu od podstaw. W tym celu należy stworzyć folder projektowy, a w nim zawrzeć co najmniej dwa pliki: *main.cpp* oraz *Makefile*. W celu przybliżenia wyglądu takiego projektu, stworzony został projekt *BareMinimum*, który jest częścią projektów przykładowych.

5.3. Uruchomienie projektu

Aby uruchomić utworzony wcześniej projekt, należy postępować zgodnie z krokami wymienionymi poniżej:

1. Aby uruchomić projekt należy najpierw upewnić się, że wszystkie pliki zostały wypakowane z archiwum w jeden katalog na dysku,
2. Po rozpakowaniu, należy przejść do katalogu *sdk* a potem *sifteo-sdk-windows,*
3. Wewnątrz katalogu znajduje się konsola *Sifteo,* o nazwie *sifteo-sdk-shell.cmd,*
4. Uruchamiamy konsolę,
5. Nawigujemy do katalogu z pożądanym projektem,
6. Uruchamiamy projekt wpisując „*make* <nazwa projektu>”,
7. Jeśli podczas budowania projektu nie wystąpią błędy - uruchomi się nam symulator *Sifteo.* W odwrotnym przypadku, błędy zostaną wypisane w konsoli.

Oczywiście, aby skorzystać z rozszerzonego interfejsu programistycznego, należy spełniać wymagania sprzętowe *Sifteo* SDK. SDK jest udostępniane na licencji wolnego oprogramowania.

6. Przykładowe projekty

Podczas tworzenia rozszerzonego API stworzone zostały projekty przykładowe, mające na celu zaznajomienie użytkowników *Sifteo* z technologią, środowiskiem oraz interfejsem. Wszystkie projekty mają charakter dydaktyczny i stanowią przykłady implementacji konkretnych zagadnień.

6.1. ”BareMinimum”

Projekt stworzony został w celu przedstawienia absolutnie minimalnej liczby plików oraz linii kodu, która jest konieczna do poprawnego działania aplikacji na kostkach *Sifteo*. Może stanowić szablon dla nowych aplikacji. Projekt zawiera dwa pliki: *main.cpp* oraz *Makefile*. Pierwszy plik zawiera kod aplikacji, kompilowany do kodu maszynowego. Drugi z plików służy za zbiór instrukcji dla kompilatora, według których będzie przebiegała kompilacja. Aby zauważyć uruchomienie projektu, konieczne jest zdefiniowanie obiektu metadanych oraz umieszczenie pętli głównej wewnątrz funkcji *main*.

6.2. ”Template”

Projekt jest wyposażony w podstawowe środowisko. Ma on służyć jako szablon na projekty. Posiada zdefiniowany obiekt metadanych, środowiska oraz dołączone pliki nagłówkowe rozszerzonego interfejsu programistycznego. Szablon aplikacji posiada pętlę główną, w której umieszczona została pętla opóźniająca, stanowiąca licznik czasu. Dzięki tej pętli, aplikacja może zatrzymywać się w określonych przez programistę momentach.

Listing 6.2. Zawartość pliku Main.cpp szablonu projektu template

```
/*=========================================================

    template

Project template.

By:
Version:
=========================================================*/
#include <sifteo.h>
#include "Environment.h"
using namespace Sifteo;

static Metadata M = Metadata()
    .title("New project")
    .package("TBD", "1.0");
    .cubeRange(1);
```

```
    Environment env; // global environment object, for accessing the EAPI

void main()
{
    while (true)
    {
        for (unsigned i = 0; i < 0x200; i += 4)  //waiting loop, does nothing but
paints for a second
        {
            System::paint();
        }
    }
}
```

6.3. "Hello World"

Jest to projekt stanowiący przykład wykorzystania rozszerzonego interfejsu programistycznego w celu wypisywania tekstu na ekran. Projekt jest przystosowany do przeprowadzenia testów jednostkowych związanych z wyświetlaniem tekstu. Aby skorzystać z tej funkcji, projekt został wyposażony w następujące pliki:

- EAPI, czyli pliki nagłówkowe:
  - Environment.h,
  - CubeWrapper.h,
  - SimplePrinter.h,
- Pliki obsługi tekstu:
  - fontdata.h,
  - fontgen.py,
  - 04B_03_.TTF.

Pliki są niezbędne do prawidłowego działania aplikacji.

6.4. "Hello Asset"

Projekt stanowi przykład wykorzystania plików opisu zasobów. Aby korzystać z plików zasobów, konieczne są pewne zmiany w projekcie:

- dodanie plików obrazu do folderu projektu (sugerowane jest użycie plików o formacie *.png lub podobnym formacie bez kompresji danych),
- dodanie pliku assets.lua do projektu. Pliki te służą do tworzenia grup plików zasobów, które to grupy są potem przetwarzane razem i ładowane na kostki przez EAPI,
- zmiany w makefile:
  - dodanie pliku obiektów $(ASSETS).gen.o przy wymienianiu plików obiektów,

  - dodanie do zmiennej ASSETDEPS źródeł plików oraz pliku .lua,
  - dodanie zależności od nagłówków C++.
- Aby korzystać z plików opisu zasobów wewnątrz projektu, projekt musi zaimportować wygenerowany w czasie kompilacji plik nagłówkowy poprzez dyrektywę `#include "assets.gen.h"`.

Listing 6.4. Zawartość pliku Makefile projektu HelloAsset

```
APP = HelloAsset

include $(SDK_DIR)/Makefile.defs

OBJS = $(ASSETS).gen.o main.o
ASSETDEPS += *.png $(ASSETS).lua
CDEPS += *.h

# build assets.html to proof stir-processed assets.
# comment out to disable.
ASSETS_BUILD_PROOF := yes

include $(SDK_DIR)/Makefile.rules
```

7. Weryfikacja

7.1. Uruchamianie testów jednostkowych

Aby uruchomić którykolwiek z przedstawionych poniżej testów, należy postępować zgodnie z instrukcjami zawartymi w podpunkcie „Przygotowanie testu”. Po wykonaniu zawartych tam kroków, należy uruchomić projekt zawierający kod testowy.

7.2. Testy Jednostkowe, wypisywanie na ekran

Proponowanym projektem do celów testowania jest projekt „*Hello World”* ponieważ przedstawiono w nim podstawową funkcjonalność wypisywania tekstu na ekran.

Poniżej zostały zaprezentowane testy jednostkowe interfejsu, ktore zostały opracowane:

- Pierwszy z testów pozwala sprawdzić prawidłowe zawijanie wyświetlanego tekstu, w sytuacji gdy liczba znaków przekracza pojemność wyświetlanej linii (25 znaków). Aby przygotować test, należy skopiować następujący kod do projektu:

```
env.getCube(0).print("12345678901234567890123456");
```

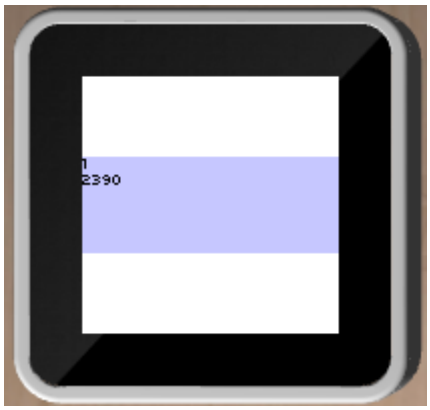
Oczekiwany wynik przedstawia rys. 7.1.

Rysunek 7.1 Test prostego wypisywania na ekran

- Następny test pozwala sprawdzić, czy znak nowej linii jest poprawnie interpretowany. Aby przygotować test, nalezy skopiować następujący kod do projektu:

```
env.getCube(0).print("1\n2390");
```

Oczekiwany wynik przedstawia rys. 7.2.

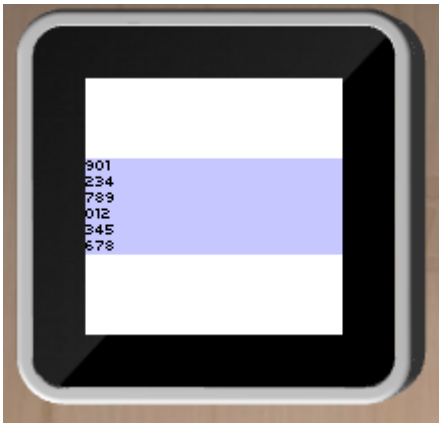


Rysunek 7.2 Test znaku nowej linii

- Trzeci test pozwala sprawdzić prawidłowe nadpisywanie linii w przypadku wypisywnia większej ilości linii niż może pomieścić pojedyncze okno wypisywania. Aby przygotować test, należy skopiować następujący kod do projektu:

```
env.getCube(0).print("123\n456\n789\n012\n345\n678\n901\n234");
```

Oczekiwany wynik przedstawia rys. 7.3.



Rysunek 7.3 Test przesycenia okna znakami nowej linii

- Następny test pozwala sprawdzić, czy znak tabulacji jest poprawnie interpretowany. Aby przygotować test, należy skopiować następujący kod do projektu:

```
env.getCube(0).print("123\t456");
```

Oczekiwany wynik przedstawia rys. 7.4.

Rysunek 7.4 Test znaku tabulacji

- Piąty test pozwala sprawdzić prawidłowe przejście do wyświetlania znaków w nastpnej linii po przepełnieniu linii znakami tabulacji. Aby przygotować test, należy skopiować następujący kod do projektu:

```
env.getCube(0).print("123\t\t\t\t\t\t456");
```

Oczekiwany wynik przedstawia rys. 7.5.

Rysunek 7.5 Test przesycenia okna znakami tabulacji

- Następny test pozwala sprawdzić działanie zmiany koloru tła okna wyświetlania tekstu. Aby przygotować test, należy skopiować nastepujący kod do projektu:

```
env.getCube(0).getPrinter().paintPrintingBackColor(RGB565::fromRGB(0x00FF00));
env.getCube(0).print("qwertyasdfghzxcvbnmasdfg");
```

Oczekiwany wynik przedstawia rys. 7.6.

Rysunek 7.6 Test zmiany koloru tła okna wypisywania tekstu

- Siódmy test pozwala sprawdzić działanie zmiany koloru tekstu, który jest wypisywany w oknie. Aby przygotować test, należy skopiować następujący kod do projektu:

```
env.getCube(0).getPrinter().paintPrintingColor(RGB565::fromRGB(0xF000F0));
env.getCube(0).print("qwertyasdfghzxcvbnmasdfg");
```

Oczekiwany wynik przedstawia rys. 7.7.

Rysunek 7.7 Test zmiany koloru wypisywanego tekstu

- Nastepny test pozwala sprawdzić poprawne tworzenie i wykorzystanie dwóch pozostałych typów okien wypisywania tekstu. Aby przygotować pierwszą część testu, tworzącą okno wypisywania na górze ekranu, należy skopiować następujący kod do projektu:

```
env.getCube(0).setPrinterTo(Top);
env.getCube(0).print("qwertyasdfghzxcvbnmasdfg");
```

Oczekiwany wynik przedstawia rys. 7.8.

Rysunek 7.8 Test zmiany pozycji okna na pozycję "Top"

Aby przygotować drugą część testu, tworzącą okno wypisywania na dole, należy skopiować następujący kod do projektu:

```
env.getCube(0).setPrinterTo(Bottom);
env.getCube(0).print("qwertyasdfghzxcvbnmasdfg");
```

Oczekiwany wynik przedstawia rys. 7.9.

Rysunek 7.9 Test zmiany pozycji okna na pozycję "Bottom"

- Dziewiąty test pozwala sprawdzić działanie wyświetlania podanej przez użytkownika grafiki. Aby przygotować test, należy skopiować następujący kod do projektu:

```
env.loadAssetToCube(0, 0, CustomGroup, gateImage);
```

Ten test można przeprowadzić na dowolnej grafice, o rozmiarze mniejszym lub równym 128x128 pikseli, w formacie *.png. Oczekiwany wynik przedstawia rys. 7.10.

Rysunek 7.10 Test wyświetlenia zadanego obrazu

- Ostatni test pozwala nam sprawdzić działanie zmiany koloru tła całego wyświetlacza. Aby przygotować test, należy skopiować następujący kod do projektu:

```
UByte3 color;
color.x = 0xA0;
color.y = 0xFF;
color.z = 0x00;
env.getCube(0).showColor(color);
```

Ten test można przeprowadzić na dowolnym kolorze, w dozwolonej formie zmiennej (RGB565, Ubyte3 etc.). Oczekiwany wynik przedstawia rys. 7.11.

Rysunek 7.11 Test wypełnienia ekranu tłem o zadanym kolorze

7.3. Porównanie starego i nowego API

7.3.1. Wyświetlanie tekstu

Narzędzia używane do wyświetlania tekstu w obu interfejsach są do siebie zbliżone. Kod wywołujący metody wyświetlające tekst na ekranie mają podobne wywołania w funkcji *main()*.

Listing 7.3.1.1 Sifteo API, projekt „text”

```
void main()
{
    /*
     * Draw some text!
     *
     * We do the drawing while the text is invisible (same fg and bg
     * colors), then fade it in and out using palette animation.
     */

    TextRenderer text(vid.fb128);
    Events::cubeRefresh.set(onRefresh);
    initDrawing();

    while (1) {
        text.position.y = 16;
        text.fb.fill(0);
        text.drawCentered("Welcome to");
        text.drawCentered("the future of text.");
        fadeInAndOut(vid.colormap);

        text.position.y = 16;
        text.fb.fill(0);
        text.drawCentered("The future of text");
        text.drawCentered("is now.");
        fadeInAndOut(vid.colormap);
```

Listing 7.3.1.2. Rozszerzone API. Projekt „Hello world”

```
#include <sifteo.h>
#include "Environment.h"
using namespace Sifteo;

static Metadata M = Metadata()
    .title("Printing example project")
    .package("com.sifteo.eapi.printing", "1.0")
    .cubeRange(3);

void main()
{
    Environment env;
    env.init(3, 1);
    env.getCube(0).print("World says hello!");
    while (true)
    {
        for (unsigned i = 0; i < 0x200; i += 4)
        {
            System::paint();
```

```
        }
    }
}
```

Kod rozszerzonego API jest bardziej skoncentrowany i nie wymaga znajomości żadnych wewnętrznych mechanizmów Sifteo. Użytkownik nie musi wiedzieć, co ile pikseli zaczyna się nowa linijka ani jakiego trybu graficznego używa. Wywołanie funkcji jest proste – wybieramy kostkę, na której chcemy wyświetlać, po czym podajemy tekst do wyświetlenia.

Jest to bardzo wygodne wywołanie dla użytkownika końcowego. Jednocześnie, rozszerzone API pozwala wyłuskać doświadczonemu użytkownikowi bardziej skomplikowane struktury, jak klasy *SimplePrinter* lub *VideoBuffer* i pozwala na modyfikację ich zachowań. Przez to, stworzone API jest przyjazne dla uzytkowników, którzy nie mieli wcześniej kontaktu z *Sifteo*, ale jednocześnie daje możliwości dostosowywania narzędzi dla doświadczonych użytkowników.

Możliwości klasy *SimplePrinter* są też o wiele większe – pozwala ona programiście zmieniać kolory tekstu i tła, pozycję okna wyświetlania tekstu na kostce, a obiekty tej klasy wykrywają znaki formatowania tekstu „*t*” oraz „*n*”.

7.3.2. Wyświetlanie obrazów

W projektach przykładowych *Sifteo* każdy obraz musiał być przygotowywany, ładowany do pamięci oraz wyświetlany przez programistę. Programista musiał zadbać o bufory graficzne, odpowiednie tryby i konieczne ładowanie plików obrazu.

Listing 7.3.2.1. Przykładowa obsługa wyświetlania obrazu. Nie wszystkie etapy są ukazane.

```
//inicjalizacja buforów video
void initVid()
{
    int id;

    for(id = 0; id <gNumCubes;id++)
    {
        vid[id].initMode(BG0);
        vid[id].attach(id);
    }
}


void main()
{
    initVid();
    CubeID currentCube = 0;
    int id, accel;


    //ustawienie poczatkowe loga i tla
    vid[0].bg0.image(vec(0,0),Logo);
```

```
    for(id = 1; id <gNumCubes;id++)
    {
    vid[id].bg0.image(vec(0,0),BlackTile);
    }
```

Rozszerzone API ułatwia dostęp programisty do obrazów i ogranicza konieczne przygotowania ze strony programisty tylko do dwóch decyzji:

- którą sekcję pamięci powinna zajmować grupa plików,
- oraz który obraz należy teraz wyświetlić.

Wszystkie inne potrzebne operacje, włącznie z ładowaniem plików i operowaniem na buforach graficznych, wykonywane są automatycznie przez interfejs programistyczny.

Listing 7.3.2.2. Kod rozszerzonego API. Wszystkie operacje potrzebne do wyświetlenia obrazu są widoczne

```
#include <sifteo.h>
#include "assets.gen.h"

#include "Environment.h"

using namespace Sifteo;

static Metadata M = Metadata()
    .title("Printing example project")
    .package("com.sifteo.eapi.printing", "1.0")
    .cubeRange(3);

void main()
{
    Environment env;
    env.init(3, 1);
    env.loadAssetToCube(0, 0, CustomGroup, gateImage);
    while (true)
    {
        for (unsigned i = 0; i < 0x200; i += 4)
        {
            System::paint();
        }
    }
}
```

Wykorzystywanie któregokolwiek z interfejsów dalej wymusza na programiście wskazanie plików, których chce używać w plikach *.lua, oraz może wymagać dodania lub modyfikacji innych plików projektu.

8. Uwagi końcowe

Opracowany został interfejs programistyczny służący ułatwieniu dostępu i uporządkowaniu istniejących metod interfejsu.

Cel projektu został osiągnięty. Interfejs jest gotowy do wykorzystania. Posiada on wiele możliwych dróg rozwoju oraz funkcjonalności które nie zostały zaimplementowane ze względu na ograniczony czas projektowania.

Interfejs wprowadza znaczne usprawnienia, co udowodniono w siódmym rozdziale pracy. Kod wygląda przejrzyściej, inicjalizacja jest niezauważalna przez programistę. Programista nie musi też już martwić się zarządzaniem strukturami takimi jak bufor graficzny albo wybór trybu grafiki (w ogólnym przypadku).

Opracowano projekty wprowadzające chętnych do zapoznania się z obsługą technologii kostek *Sifteo*. Projekty przedstawiają podstawowe funkcjonalności systemu. Dzięki rozwiniętemu interfejsowi, nauka obsługi podstawowych działań w systemie nie powinna sprawiać problemu.

Sporym problemem systemów wbudowanych jest silnie ograniczona ilość dostępnej pamięci oraz czas przesyłu danych pomiędzy dwoma punktami. Podczas prac pojawiło się sporo problemów:

- wyczerpanie pamięci podręcznej,
- niespodziewane błędy w oprogramowaniu kostek,
- spodziewane, lecz trudne do usunięcia problemy związane z ograniczeniami wyświetlacza,
- ograniczona funkcjonalność istniejącego API,
- brak dostępu do niższych poziomów oprogramowania uniemożliwa pewne rozwiązania wygodne z punktu widzenia rozszerzonego interfejsu.

Literatura

1. Dokumentacja Sifteo SDK, (dostęp: listopad 2017) https://sifteo.github.io/thundercracker/getting_started.html
2. Eric Freeman tł. Paweł Koronkiewicz, Grzegorz Kowalczyk. (2011), Wzorce Projektowe, Gliwice, Helion
3. Erich Gamma tł. Tomasz Walczak (2010), Wzorce projektowe: elementy oprogramowania obiektowego wielokrotnego użytku, Gliwice, Helion
4. Jerzy Grębosz (2015), Symfonia C++ standard: programowanie w języku C++ orientowane obiektowo. T.1, Kraków, Instytut Fizyki Jądrowej im. H. Niewodniczańskiego, Polska Akademia Nauk
5. Jerzy Grębosz (2016), Pasja C++: szablony, pojemniki i obsługa sytuacji wyjątkowych w języku C++. Kraków, Wydawnictwo Edition 2000, Oficyna Kallimach
6. Brian W. Kernighan, Dennis M. Ritchie, tł. Paweł Koronkiewicz (2010), Język ANSI C: programowanie, Gliwice, Helion
7. K.N. King, tł. Przemysław Szeremiota (2011), Język C: nowoczesne programowanie, Gliwice, Helion
8. Jerzy Grębosz (2015), Symfonia C++ standard: programowanie w języku C++ orientowane obiektowo. T.2, Kraków, Instytut Fizyki Jądrowej im. H. Niewodniczańskiego, Polska Akademia Nauk
9. Yifeng Zhu (2015), Embedded systems with ARM Cortex-M microcontrollers in assembly language and C. E-Man Press LLC
10. Robert Mecklenburg (2004), Managing Projects with GNU Make, Third Edition, O'Reilly Media, Inc.

Załączniki

1. Elektroniczna postać dokumentu pracy dyplomowej
2. Opracowany kod biblioteki rozszerzonego interfejsu
3. Przykłady dydaktyczne