POLITECHNIKA KRAKOWSKA

IM. TADEUSZA KOŚCIUSZKI

WYDZIAŁ FIZYKI MATEMATYKI I INFORMATYKI

KIERUNEK FIZYKA TECHNICZNA

MATEUSZ HOFMAN

EDUKACYJNY PROGRAM DO NAUKI PROGRAMOWANIA W JĘZYKU SCHEME

PRACA INŻYNIERSKA

STUDIA STACJONARNE

Ocena:

…….

Podpis promotora:

……

Promotor: dr.inż. Radosław Kycia

Kraków 2016

Składam serdeczne podziękowania
dr inż. Radosławowi Kyci
za poświęcony czas i pomoc
w przygotowaniu niniejszej pracy

Spis treści

1. Wstęp:

Postępująca informatyzacja naszego otoczenia powoduje, że coraz młodsi ludzie interesują się przeróżnymi zagadnieniami informatycznymi. Bardzo ważną częścią tych zagadnień są języki programowania. W dzisiejszych czasach są one kluczowe do rozwijania szeroko pojętej branży IT. Nie tylko uczniowie szkół średnich aspirujący do rozpoczęcia studiów technicznych na uczelniach wyższych wykazują zainteresowanie językami programowania. We współczesnym świecie duża część dzieci ma dostęp do nowoczesnych urządzeń elektronicznych od najmłodszych lat. Używanie tego sprzętu wzmaga w nich chęć poznania zasady działania danych urządzeń. Efektem tego jest coraz większe zainsteresowanie uczniów w szkołach zajęciami pozalekcyjnymi z programowania czy nawet robotyki. Uczniowie od najmłodszych lat z zainteresowaniem starają się poznać zasady działania otaczających ich urządzeń.

Językami cieszącymi się największą popularnością są: C, C++, Java, PHP. W niniejszej pracy nacisk położono głównie na język programowania o nazwie Scheme. Można śmiało powiedzieć, że język ten jest językiem niszowym. Scheme bardzo różni się od języków typu C/C++, prawdopodobnie dlatego nie jest tak bardzo rozpowszechniony jak wyżej wymienione języki.

Wiodącym zagadnieniem w tej pracy jest tworzenie aplikacji edukacyjnej. W założeniu aplikacja stworzona na potrzeby tej pracy jest przystosowana do używania zarówno przez studentów jak i przez młodszych uczniów. Można śmiało powiedzieć, że zarówno młodsi i starsi uczniowie znajdą w tej aplikacji coś dla siebie. Młodsi użytkownicy ograniczą się do prostszych zagadnień. Jednak bardziej wprawni użytkownicy będą mogli zająć się bardziej skomplikowanymi problemami, np. rysowaniem zaawansowanych figur geometrycznych.

2. Cel i zakres pracy:

Głównym celem pracy jest stworzenie programu do nauki programowania w języki Scheme dla uczniów i studentów. Głównym elementem programu jest interpreter języka Scheme – Guile, rozszerzony o dodatkowe funkcje umożliwiające rysowanie obiektów geometrycznych. Za graficzną część programu jest odpowiedzialna biblioteka OpenGL. Do programu zostały dołączone dodatkowe skrypty umożliwiające rysowanie obiektów geometrycznych takich jak np. fraktale. Stworzony system może być również prostym edytorem graficznym sterowanym poleceniami języka Scheme. Główną problematyką programu jest osadzenie Guile w aplikacji Glut oraz funkcjionalność umożliwiająca pisanie skryptów w języku Scheme, które wyzwalają odpowiednie funkcje w OpenGL. Główny program został napisany w języku C. Zakres pracy wygląda następująco:

- Przegląd literatury związanej z interpreterem Guile;
- Przegląd literatury związanej z językiem programowania Scheme;
- Nauka osadzania interpretera Guile w programach napisanych w języku C
- Krótki opis działania biblioteki graficznej OpenGL;
- Stworzenie programu interpretującego poleceń języka Scheme przez bibliotekę OpenGL;
- Testy aplikacji;

3. INTERPRETER GUILE / SCHEME

3.1 Ogólnie o Guile

Guile jest interpreterem języka Scheme. Sam Scheme jest wysokopoziomowym językiem programowania pochodzącym z rodziny dialektów Lisp. Lisp został zaprojektowany przez Johna McCarthiego na Massachusetts Institute of Technology. Jest on jednym z pierwszych zaprojektowanych wysokopoziomowych języków programowania. Na przestrzeni ponad 50 lat powstało wiele różnych dialektów Lisp'u (John McCarthy wymyślił go w 1958 roku). Najpopularniejszymi z nich są: Common Lisp, Clojure oraz wcześniej wspomniany Scheme. [I] Różnice pomiędzy poszczególnymi dialektami są dosyć znaczące. Zarówno Scheme jak i Common Lisp używają różnych słów kluczowych przy definiowaniu funkcji, wewnątrz danego dialektu takie różnice jednak nie są spotykane. Każda z implementacji posiada zdefiniowane taki sam zestaw funkcji, dodatkowo oferując własne rozszerzenia i biblioteki dostępne w ramach danej implementacji. Twórcami dialektu Scheme są Guy Steele and Gerald Sussman.

Niestety w przeciwieństwie do innych języków programowania (tak jak np. Python) Scheme nie posiada oficjalnego projektanta – skutkuje to powstawaniem wielu implementacji samego Scheme. Wiele środowisk akademickich wprowadza własne rozwiązania dla tego dialektu co jest powodem powstawania wielu (często bardzo różnych od siebie) implementacji tego dialektu.

Scheme różni się od przykładowego Common Lisp tym, że stawia na minimalizm. Common Lisp zawiera szeroką specyfikacje – obejmuje ona wiele typów danych oraz nawet system obiektowy (system również dostępny w Scheme). Scheme udostępnia stosunkowo niewielki zbiór standardowych funkcji, rekompensuje to określonymi cechami implementacyjnymi, np. optymalizacja rekursji ogonowej czy pełne kontynuacje. Obie te opcje są niedostępne w ramach Common Lisp. Scheme charakteryzuje się również w pełni automatycznym zarządzaniem pamięcią (jest to język o dynamicznym systemie typów). [II]

Wracając do samego interpretera – Guile, został on stworzony przez twórców GNU Project (GNU jest systemem operacyjnym opierającym się tylko i wyłącznie na wolnym oprogramowaniu). Twórcy wzorując się na Emacs Lisp postanowili stworzyć dla swojego systemu operacyjnego środowisko do pisania elastycznych aplikacji

umożliwiających modyfikowanie ich przez użytkowników, bądź innych programistów za pomocą skryptów, nowych modułów itp. Funkcjonalność Guile jest zgodna z filozofią samego GNU Project, który zakłada swobodę modyfikacji otrzymanego oprogramowania przez użytkownika. Obecnie Guile jest używany w systemie GNU w aplikacjach takich jak: AutoGen, Lilypond, Denemo, TeXmacs. [1]

Jak już zostało wcześniej wspomniane, Guile różni się od płatnego oprogramowania tym, że daję swobodę modyfikacji oprogramowania. Posiada to wiele walorów – w tym również walor edukacyjny.

Od wersji 2.0, Guile umożliwia programowanie w innych językach programowania. Od tego momentu Scheme stał się zwyczajnie jednym z dostępnych języków – innymi językami są: Emacs Lisp, JavaScript, Brainfuck. Rozważane jest również wprowadzenie takich języków jak: Lua, Ruby czy Python. Guile posługuję się standardem Scheme o nazwie R5RS (Revised5 Report of the Algorithmic Language Scheme).

3. 2 WYMAGANE BIBLIOTEKI GUILE

Guile 2.0 potrzebuje pewnych zewnętrznych bibliotek do prawidłowego działania:

- GNU MP (GNU Multiple Precision Arithmetic Library) – programistyczna biblioteka odpowiedzialna za udostępnianie liczb całkowitych ze znakiem, wymierne oraz zmiennoprzecinkowe. Jej głównym walorem jest brak limitu jeśli chodzi o precyzję liczby zmiennoprzecinkowej – jedynym ograniczeniem jest tutaj ilość dostępnej pamięci operacyjnej na której działa GMP. Poza Guile jest on wykorzystywany również w takich programach obliczeniowych jak Mathematica. [III] Najnowsza wersja guile wymaga GMP w wersji przynajmniej 4.2.
- GNU Libtool – Jest to narzędzie do tworzenia zkompilowanych przenośnych bibliotek. Różne systemy operacyjne używają wieloplatformowych bibliotek w różny sposób – niektóre w ogóle nie używają tych bibliotek. Stworzenie wieloplatformowego programu może się okazać bardzo trudne. Libtool pomaga zarządzać tworzeniem statycznych i dynamicznie zmieniających się bibliotek na wielu unixopodobnych systemach operacyjnych, zacierając różnicy pomiędzy różnymi systemami operacyjnymi (np. systemy GNU/Linux kontra Solaris). W

większości przypadków narzędzie to współpracuje z innymi narzędziami ze stajni GNU: Autoconf oraz Automake, żadne z tych narzędzi nie jest jednak wymagane do działania narzędzia Libtool. Guile wymaga Libtool przynajmniej w wersji 1.5.6. [IV]

- GNU libunistring – Biblioteka ta odpowiada za przetwarzanie zestawu znaków Unicode. Guile wymaga tej biblioteki w wersji przynajmniej 0.9.3.
- Libgc – Wersja standardowerj biblioteki języka C, stworzona w ramach projektu GNU. Udostępnia funkcjonalność wymaganą przez UNIX 98, Single UNIX Specification, POSIX oraz część funkcjonalności wymaganej przez normę ISO C99, a dodatkowo rozszerzenia uznane za konieczne lub użyteczne w trakcie tworzenia GNU. Guile wymaga libgc przynajmniej w wersji 7.0. [V]
- Libffi – przenośny funkcyjny interfejs. Kompilatory języków wysokopoziomowych generują kod, który odpowiada pewnemu standardowi. Przykładowo standard wywoływania funkcji jest zbiorem założeń stworzonych przez kompilator o tym gdzie argumenty funkcji mogą się znajdować. Standard ten definiuje gdzie są umieszczone wartości zwracane przez funkcję. Niektóre programy mogą nie wiedzieć, które argumenty są przekazane do funkcji przy danej kompilacji. Libffi działa jako pomost pomiędzy kompilatorem a kompilowanym kodem. [VI]
- Pkg-config – Program który kolejkuje zainstalowane biblioteki w celu kompilacji oprogramowania z ich kodu źródłowego. [VII] Guile używa Pkg-config by uzyskać informację o odpowiednich opcjach kompilowania i linkowania dla bibliotek Libgc oraz Libffi. W tym do zmiennej PKG_CONFIG_PATH musi być dostarczona informacja o ścieżce dostępu do tych bibliotek: PKG_CONFIG_PATH=/sciezka/do/libgc/pkgconfig:/sciezka/do/libbffi/pkgconfig

4. SKŁADNIA SCHEME:

Scheme dosyć znaczączco różni się w swojej składni od większości języków programowania, które cieszą się obecnie dużym zainteresowaniem (C/C++, Java, PHP). Składnia słada się z S-wyrażeń w sposób klasyczny dla języka Lisp. Różne elementy języka – deklaracje, definicje, warunki przedstawione są za pomocą list. Lista zawiera elementy oddzielone białymi znakami (znakami odstępu, tabulacji lub kolejnego wiersza) oraz otoczona jest zwykłymi nawiasami. Niektóre implementacje dopuszczają zastosowanie nawiasów kwadratowych w celu poprawienia czytelności kodu. Pierwszym elementem listy jest jej identyfikator, po niej umieszczane są argumenty. [II]

Działania podobnie jak wszystkie inne funkcje zapisuje się w notacji prefiksowej tzn. znak działania umieszczony jest przed argumentami. Jest to uwarunkowane przynależności Scheme do rodziny języków Lisp mianowicie: są to języki funkcyjne więc nawet najprostsze dodawanie „+” traktowane jest jako funkcja, przykład:

```
( + 5 4)
```

W powyższym działaniu zostaje przywołana funkcja „+” oraz dane są dwa argumenty: 5 i 4. Wynikiem wywołania takiego wyrażenia jest 9. Deklarowanie zmiennych odbywa się w sposób następujący:

```
(define zmienna wartosc)
```

Co oznacza zdefiniowanie zmiennej „zmienna” o wartości „wartosc”. Nie jest wymagane deklarowanie typu zmiennej. Poniżej przedstawiono przykłady deklarowania dwóch zmiennych: odpowiednio przypisując im wartości liczby całkowitej oraz ciągu znaków:

```
(define x 23)
(define y „ciag_znakow”)
```

Definiowanie listy odbywa się przy użyciu procedury „quote” - nakazuje ona traktować listę jako dane, nie jako procedurę. np. wyrażenie (1 2 3 4 5) skutkuje utworzeniem listy o takiej samej strukturze. Interpreter sprawdza bowiem pierwszy element listy (w tym przypadku jest to „1) i sprawdza czy jest on procedurą i na tej podstawie podejmuje decyzję czy dane wyrażenie będzie traktowane jako procedure czy jako lista. Nie jest to jednak wystarczające ponieważ, gdy konieczne jest zdefiniowanie

listy w postaci (+ 1 2) Scheme traktuje to jako procedurę. Bez użycia „quote” pierwszy wyraz „+” jest traktowany jako procedura. Scheme dopuszcza użycie znaku ' jako zamiennik za „quote”.

(+ 1 2) → 3
(quote (+ 1 2)) → (+ 1 2)

Podstawowe operacje wykonywane na listach to „car” i „cdr”, które odpowiednio skutkują wyodrębnieniem pierwszego elementu listy oraz wyodrębnieniem wszystkich elementów poza pierwszym. Przykład:

(car '(a b c)) → (a)
(cdf '(a b c)) → (b c)

Procedura „cons” tworzy listy – wymaga dwóch argumentów. Przykład:

(cons (car '(a b c))
(cdr '(d e f))) → (a e f)

Jeżeli wymagane jest zdeklarowanie zmiennej lokalnej potrzebny jest operator „let”:

(let ((z1 wyr1) … (zn wyrn))
wyrazenie)

Powyższy kod ustawia wartość zmiennych (z1 … zn) na wyniki odpowiadających im wartości (wyr1 … wyrn) i oblicza wartość wyrażenia „wyrażenie”. Zmienne dostępne są tylko w obrębie tego kodu, „let” ogranicza ich widoczność – poza nim są przesłonięte. W ramach definiowania zmiennych można również przedefiniować podstawowe funkcje na inne:

(let ((+ *)) (+ 3 8))

Wynikiem tego wyrażenia będzie 24, gdyż w ramach tego konkretnego operatora let znak „+” odpowiada „*”. W celu zmiany wartości danej zmiennej trzeba użyć operatora „!set” i jest podobne to „zwykłych” języków programowania:

(!set x war2)

Powyższy kod zmienia wartość zmiennej „x” na „war2”. Warto dodać, że zmienna musi być wcześniej zdefiniowana poleceniem „define” lub musi być dostępna w ramach ciała innej funkcji tworzonej przez „lambda”.

Dla niektórych aplikacji bardziej odpowiednią formą grupowania danych niż listy są wektory. Wektory definiowane są operatorem „vector”. Długośc wektoru definiowana jest przez „length”. W Scheme wektory indeksowane są przez dodatnie liczby całkowite, pierwszym indeksem wektoru jest zawsze 0. Najwyższym możliwym indeksem w danym wektorze jest liczba o jeden mniejsza niż wartość „lenght”. [4]

Sama zmienna nie posiada ustalonego typu – jest to zależne od wartości jaką aktualnie przechowuje. Do sprawdzania jaką wartość przechowuje dana zmienna potrzebne są zapytania typu:

(numer? 5) - kod sprawdza czy 5 jest liczbą;
(number? „tekst”) - kod sprawdza czy „tekst” jest liczbą;
(string? „tekst”) - kod sprawdza czy „tekst” jest ciągiem znaków.

Pytania o typ zmiennej składają się więc z nazwy typu i znaku zapytania. Z powyższych wyrażeń zwracane są wartości #t oraz #f – oznaczają one odpowiednio prawdę oraz fałsz. Można na nich wykonać operacje koniunkcji, alternatywy oraz negacji.

Funkcje definiuje się za pomocą konstrukcji:(lambda (lista argumentów) (ciało funkcji)), przy czym może ona zostać wartością zmiennej (np. przez define lub let) albo zostać użyta bez nadawania jej nazwy. [II]

```
(define fib
(lambda (n)
(cond ((= n 0) 0)
((= n 1) 1)
(else (+ (fib (- n 1)) (fib (- n 2)))))))
```

Powyższy kod przedstawia sposób obliczenia ciągu Fibonacciego z danej liczby. Wyrażenie „cond” zastępuje w tym przypadku konieczność dwukrotnego umieszczenia wyrażenia zwrotu składniowego „if”. Innym przykładem jest funkcja obliczająca wynik równania kwadratowego na podstawie współczynników podanych przez użytkownika:

```
(define quadric-formula
(lambda (a b c)
(let ([root1 0] [root 2 0] [minusb 0] [radical 0] [divisor 0])
(set! Minusb (- 0 b))
(set! Radical (sqrt (- (* b b) (* 4 (* a c)))))
(set! Divisor (* 2 a))
(set! Root1 (/ (+ minusb radical) divisor))
(set! Root2 (/ (- minusv radical) divisor))
(cons root1 root2))))
```

Powyższy przykład pokazuje również, że Scheme pozwala na używanie takich funkcji matematycznych jak „sqrt” (pierwiastek).

Zamiast pętli wykorzystuje się możliwości stwarzane przez rekurencję, szczególnie przez rekurencję ogonową w której wywołanie rekurencyjne występuje jako ostatnie wyrażenie, funkcje z rekurencją ogonową zamieniane są na pętle przez kompilator. [II] Poniżej przedstawiono przykład rekurencji ogonowej przy obliczaniu silni:

```
(define (! n)
(let iter ((n n) (product 1))
if (zero? n)
product
(iter (- n 1) (* product n)))))
```

Dzięki tak zdefiniowanej rekurencji parametry nie są odkładane na stos, zupełnie jak w przypadku zwykłej iteracji.

5. PROGRAMOWANIE W C

5.1 Osadzanie Guile w C

Ten podrozdział pokazuje w jaki sposób przebiega osadzanie Guile w programie napisanym w języku C. Umożliwia to wywoływanie funkcji języka Scheme w ramach programu napisanego w C, oraz poszerzanie biblioteki Guile o nowe funkcje i procedury. Pierwszym krokiem jest dodanie do kodu programu nagłówka <libguile.h>. [1] Zapewnia ona dostęp do wszystkich funkcji i stałych dostępnych w ramach Guile. <libguile.h> nie jest jednak domyślną ścieżką dla nagłówka – poniższe komendy zapewniają ścieżkę dostępu potrzebną do kompilacji programu używając Guile 2.0:

```
pkg-config guile-2.0 - - cflags
pkg-config guile-2.0 - - libs
```

W celu zainicjowania Guile można użyć kilku funkcji – pierwszą z nich jest scm_with_guile. Jest ona najbardziej przenośnym sposobem na przywołanie Guile. Funkcja ta zainicjuje Guile w razie potrzeby i wywoła funkcję, którę może określić użytkownik. Wiele wątków może korzystać z scm_with_guile równocześnie, odwołanie do niego może również wystąpić więcej niż raz w ramach danego wątku. Globalny stan scm_with_guile przetrwa do następnego wywołania tej funkcji. Funkcje są wywoływane z wnętrza scm_with_guile, ponieważ Guile musi wiedzieć gdzie jest umieszczony stos każdego wątku.

Drugą funkcją jest scm_init_guile – odpowiada za zainicjowanie Guile. Funkcja po zwróceniu umożliwia użycie Guile API w ramach danego wątku. Funkcja ta zawiera pewne rozwiązania, które uznawane są jako „non-portable”, przez co może być niedostępna na niektórych platformach. [1]

Głównym sposobem użycia Guile w programie C jest napisanie funkcji, które mogą być wywoływane ze Scheme oraz połączyć program z Guile. Ostatecznie otrzymuje się po prostu Guile, poszerzony jednak o nowe funkcje w ramach danej aplikacji. W efekcie czego wiersz poleceń aplikacji zachowuje się w taki sam sposób jak interpreter Guile. Funkcjami odpowiedzialnymi za to są: scm_boot_guile i scm_shell. [1] Poniższy kod przedstawia prosty przykład osadzenia interpretera Guile w aplikacji napisanej w języku C:

```
#include <libguile.h>
static void inner_main (void *closure, int argc, char **argv)
```

```
{
scm_shell (argc, argv);
}
int main (int argc, char **argv)
{
scm_boot_guile (argc, argv, inner_main, 0);
return 0;
}
```

Główna funkcja wywołuje scm_boot_guile, w celu wywołania Guile. Po wywołaniu tej funkcji scm_boot_guile odwołuje się do funkcji inner_main, która wywołuje scm_shell w celu wywołania wiersza poleceń.

5.2 GUILE API

Guile's Aplication Programming Interface (Guile API) to interfejs programistyczny pozwalający na używanie dwóch języków: C i Scheme. Wiele jego elementów, np. Procedura assq w Scheme jest również dostępna jako scm_assq w języku C. Element nazwany w Scheme jest powiązany z elementem w języku C w regularny sposób. Funkcja w C zawsze bierze stałą wartość argumentów typu SCM, nawet jeśli analogiczna funkcja w Scheme korzysta z dynamicznie zmieniającej się liczbie argumentów. Dla niektórych funkcji w Scheme ostatni argument może być opcjonalny – powiązana funkcja w C musi wiedzieć o wszystkich określonych opcjonalnych argumentach. Wówczas, jeżeli argument nie został określony – jego wartość ustawia się jako SCM_UNDEFINED. Wartość zwrotna funcji w C, która komunikuje się z funkcją w Scheme musi być zawsze typu SCM. [1]

5.3 TYP SCM

Guile reprezentuje wszystkie wartość z Scheme w języku C jednym typem – SCM. Jest to typ abstrakcyjny dostępny z poziomu użytkownika. Najważniejszą zasadą typu SCM jest to, że nie ma możliwości odniesienia się bezpośrednio do argumentu typu SCM, a jedynie do funkcji lub makra w libguile. Prostym przykładem może być próba dodania dwóch zmiennych typu SCM, które zawierają w sobie dwie liczby całkowite. Niemożliwe jest wykonanie operacji dodawania operatorem "+" z języka C – konieczne jest użycie funkcji libguile - w tym przypadku scm_sum. Innym przykładem może być

zmienna, która przechowuje wartość logiczną - #f. Dla Scheme oczywiście oznacza to fałsz, nie ma jednak gwaracji, że reprezentacja typu SCM dla fałszu wygląda tak samo w języku C. Konieczne jest użycie funkcji scm_is_true lub scm_is_false w celu sprawdzenia wartości SCM. Odpowiednikiem definiowania zmiennych w interfejsie API w języku C (w języku Scheme jest to operator "define") jest scm_define oraz scm_c_define. W zalezności od tego czy nazwa zmiennej jest określona jako symbol SCM czy nieokreślony ciąg znaków: [1]

```
scm_define (symbol, value)
scm_c_define (const char *name, value)
```

5.4 MAKE – TWORZENIE MAKEFILE

W celu skompilowania powyższego programu użyto reguły Makefile oraz narzędzia make. Narzędzie to służy do zarządzania kompilacją programów, na które składa się kilka plików źródłowych. Program make przyjmuje na wejściu plik makefile (zwykle o nazwie Makefile), który opisuje zależności i relacje między komponentami tworzącymi gotowy program. [3] Następnie przy pomocy polecenia make możliwa jest kompilacja projektu. Narzędzie to znacznie usprawnia kompilację, ponieważ samodzielnie decyduje o tym, które pliki mają zostać zkompilowane. Ponowna kompilacja zachodzi bowiem tylko w przypadku plików, które zostały w jakikolwiek sposób zmienione po ostatniej kompilacji. Skrypt Makefile składa się w większości z reguł, które mają następującą budowę:

```
[TUTAJ MOŻE GRAFICZNY SCHEMAT?]
CEL: SKŁADNIKI
KOMENDA
```

CEL jest nazwą pliku docelowego – jest on tworzony z plików wymienionych jako SKŁADNIKI. KOMENDA oznacza komendę, która tworzy plik docelowy z plików składowych SKŁADNIKI. Najprostrzym przykładem pliku Makefile jest:

```
hello: hello.c aux.c
gcc hello.c aux.c -o hello
```

Jest to reguła określająca sposób tworzenia pliku hello na bazie plików hello.c i aux.c. Make tworzy plik docelowy znajdujący się w pierwszej regule w skrypcie Makefile. Pozostałe reguły mają charakter pomocniczy – podają sposób stworzenia składników reguł znajdujących się nad nimi.[VIII] Rozwijając powyższy przykład:

```
hello: hello.o aux.o
gcc hello.o aux.o -o hello
hello.o: hello.c
gcc -c hello.c -o hello.o
aux.o aux.c
gcc -c aux.c -o aux.o
```

Dodano tu kolejne dwie reguły tworzenia plików hello.o i aux.o. W tym przypadku głównym plikiem docelowym jest plik hello. W pierwszej kolejności narzędzie make szuka danych plików w katalogu docelowym – jeśli ich nie znajduje szuka reguł umożliwiających stworzenie tych plików. Make tworzy składniki reguły głównej, a później plik docelowy.

Narzędzie make posługuje się wieloma zmiennymi, które są predefiniowane – można jednak w nie ingerować. Przykładami tych zmiennych są: CC – odpowiada nazwie kompilatora języka C; CFLAGS – odpowiada opcjom kompilatora języka C; LFLAGS – odpowiada opcjom dla linkera. Dodatkowo możliwe jest odwołanie do tzw. zmiennych automatycznych – zmienne, które są dynamicznie używane podczas wykonywania pliku Makefile, np. < - aktualnie przetwarzany plik z listy składników; @ - nazwa pliku docelowego; ^ - składniki. Przykład użycia zmiennych:

```
CC = gcc
CFLAGS = -g
OBJS = hello.o aux.o
LFLAGS=

hello: $(OBJS)
$(CC) $(LFLAGS) $^ -o $@
hello.o: hello.c
$(CC) $(CFLAGS) -c $< -o $@
aux.o: aux.c
$(CC) $(CFLAGS) -c $< -o $@
```

W tym przypadku jako kompilatora użyto gcc (pełna nazwa: the GNU Compiler Collection). Przed użyciem każdej zmiennej wymagany jest znak dolara ($). Wracając do przykładu osadzania Guile w programie napisanym w C – plik Makefile do kompilacji tego programu będzię miał postać:

```
CC = gcc
CFLAGS = 'pkg-config – cflags guile-2.0'
LIBS = 'pkg-config –libs guile 2.0'

simple-guile: simple-guile.o
```

```
${CC} simple-guile.o ${LIBS} -o simple-guile
simple-guile.o: simple-guile.c
${CC} -c ${CFLAGS} simple-guile.c
```

Zmiennej CFLAGS została przypisana ścieżka dostępu do <libguile.h> , zmienna LIBS natomiast mówi linkerowi, których bibliotek użyć oraz gdzie ich szukać.[1]

Każdy dobry plik Makefile powinien posiadać regułę, która usuwa pliki pośrednie powstałe podczas kompilacji. W opisywanym programie takim plikiem pośrednim są pliki main.x oraz main.o. Powszechnie przyjęło się, że taka reguła ma nazwę 'clean'. Przykład:

```
clean:
rm -f main.x main.o
```

Reguła clean ma pustą listę składników – do wywołania tej reguły służy polecenie make clean. Jest to reguła specjalna w tym sensie, że "clean" nie jest nazwą pliku. Gdyby jednak zdarzyło się, że w katalogu bieżacym istnieje plik o nazwie "clean" to reguły te mogłby nie działać. Aby temu zapobiec trzeba poinstruować narzedzie make że "clean" nie jest nazwą pliku. [VIII] Do tego celu służy specjalna reguła o nazwie ".PHONY":

```
.PHONY clean build run
```

5.5 OPENGL

Za część graficzną programu jest odpowiedzialna biblioteka OpenGL – w tym celu zostanie użyta implementacja biblioteki GLUT o nazwie Freeglut. OpenGL (z ang. Open Graphics Library) jest to specyfikacja otwartego i uniwersalnego API do tworzenia grafiki. Zaletą OpenGL jest jego wieloplatformowość – programy z korzystające z tej biblioteki działają praktycznie we wszystkich współczesnych systemach operacyjnych. [2] Zestaw funkcji składa się z 250 podstawowych wywołań, umożliwiających budowanie złożonych trójwymiarowych scen z podstawowych figur geometrycznych. Głównym celem jest tworzenie grafiki. Dzięki temu, że polecenia są realizowane przez sprzęt (procesor graficzny = GPU) tworzenie grafiki następuje szybciej niż innymi sposobami. [IX]

OpenGL działa na zasadzie klient-serwer. Klientem w tym przypadku jest aplikacja, która przesyła rządania do wykonania przez serwer - aktualnie używaną implementację OpenGL. Sam OpenGL charakteryzuje się tym, że jest on maszyną stanu

tzn. posiada on wiele parametrów stanu oraz trybów działania. Raz ustawione parametry są ustawiane w stosie – ich wartość pozostaje taka sama aż do następnej zmiany. Przykładem może być tutaj parametr odpowiedzialny za kolor rysowania linii. Funkcją odpowiedzialną na zapamiętywanie aktualnego stanu jest glPushAttrib(). Funkcja do niej przeciwna – glPopAttrib() odpowiada za pobieranie ze stosu zapamiętany stan. Nie wymaga więc żadnych argumentów. Do włączania i wyłączania znacznej części parametrów służą funkcję glEnable() oraz glDisable().

6. Projekt programu

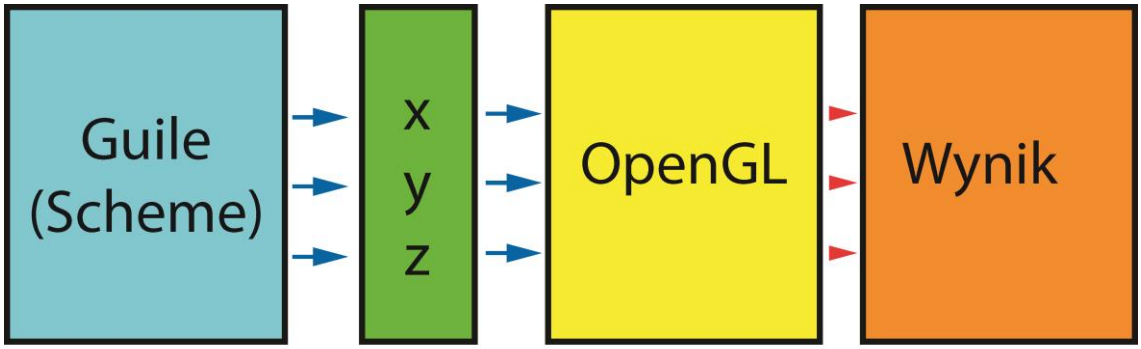
6.1 Założenia programu:

Celem pracy było stworzenie narzędzia do nauki programowania w języku Scheme dla uczniów i studentów. Języka Scheme używa się na MIT (Massachusetts Institute of Technology) w celu nauczenia nowych studentów podstaw z zakresu szeroko pojętego programowania. Jest więc on również dobrym podłożem do nauki podstaw programowania młodszych uczniów. Stworzony program został przystosowany do tego, aby mogli z niego korzystać zarówno studenci jak i uczniowie, którzy są na niższych szczeblach edukcji. Młodsi mogą zagłębiać tajniki rysowania podstawowych figur geometrycznych – starsi zaś (używając tych samych prostych komend odpowiedzialnych za rysowanie) będą w stanie stworzyć bardziej skomplikowane obrazy graficzne. Rysowanie zostało w dużej mierze oparte na tak zwanej „Grafice Żółwia" (eng. Turtle Graphics). Jest to dość często poruszany termin przy tworzeniu grafiki komputerowej tego typu. Polega to na osadzeniu hipotetycznego żółwia w punkcje o danych współrzędnych, w którym znajduje się aktualnie użytkownik. [5] Wprowadzając komendy odpowiedzialne za sterowanie żółwiem, można obserwować w oknie graficznym efekty rysowania wyznaczonego przez tor poruszania się żółwia.

6.2 Wymagania programu:

Program do prawidłowego działania wymaga systemu operacyjnego UNIX/Linux (z tego względu, że Guile jest przeznaczony właśnie na tego typu platformy). Ponadto wymagane są oczywiście sam interpreter Guile (zalecana jest wersja 2.0.11) oraz biblioteka OpenGL – w tym wypadku jest to freeglut3. Freeglut jest otwartą implementacją biblioteki GLUT. GLUT (a więc również freeglut) pozwala użytkownikowi tworzyć i zarządzać oknami używającymi OpenGL na wielu platformach. Freeglut jest w zamierzeniu dokładnym zastępstwem dla GLUT i różni się w zaledwie kilku punktach. [X]

6.3 Schemat działania programu



Rys.1 Schemat przedstawiający ogólną zasadę działania programu.

Korzystając z wcześniej opisanej możliwości zaimplementowania Scheme do progamu C można umożliwić użytkownikowi korzystanie z wiersza poleceń Guile podczas działania programu C. Ponadto implementacja ta umożliwia dodawanie własnych komend do Guile. W Rys.1 przedstawiona została ogólna zasada wywoływania funkcji Scheme oraz interpretowanie nowych komend dodanych przez program. Scheme wysyła argumenty (np. Współrzędne rysowania nowej linii), które są następnie interpretowane przez OpenGL, który Tworzy nowe okno oraz rysuje prymitywy geometryczne na bazie wcześniej otrzymanych współrzędnych.

6.4 Rejestrowanie nowych funkcji

Program przedstawiony w pracy rejestruje nowe funkcje, które są dostępne dla użytkownika z poziomu wiersze poleceń Guile. Głównym celem tych funkcji jest manipulowanie zmienną call – jest ona zmienną typu enum, która przyjmuje tylko określone wartości:

```
enum wartosc {CLEAR, MOVE, TURN, POLYGON, HUDGE_LINE, SMALL_LINE,
V_POLYGON, TEAPOT};

enum wartosc call;
```

Wartości przedstawione w kodzie powyżej uaktywniają odpowiednie funkcje w OpenGL. Nie wszystkie funkcje jednak zmieniają tą zmienną – niektóre funkcje funkcje zmieniają tylko wartości kluczowe dla samego rysowania. Przykładem będzie zmienna mpendown,

która zależnie od tego czy jest interpretowana jako prawda lub fałsz umożliwia rysowanie linii na ekranie. Prostym przykładem definiowania funkcji może być funkcja cclear:

```
static SCM cclear()
{call = CLEAR;
return SCM_UNSPECIFIED;}
```

Wszystkie rejestrowane funkcje są typu SCM, wymaga tego Guile API. Funkcja po wywołaniu zmienia wartość zmiennej call na CLEAR oraz zwraca typ SCM_UNSPECIFIED. Jest to spowodowane tym, że w języku Scheme każda funkcja musi zwracać jakąś wartość, co nie jest wymagane w języku C. Funkcja cclear() zwraca więc wartość SCM_UNSPECIFIED, aby nie wywoływać błędu składniowego. Funkcja cclear() służy do czyszczenia okna na którym wyświetla się grafika. Kolejna funkcja jest odpowiedzialna za rysowanie linii, jest to funkcja mmove:

```
static SCM mmove(SCM length1)
{length = scm_to_double (length1);
newX = x + length * cos (direction);
newY = y + length * sin (direction);
call = MOVE;
return SCM_UNSPECIFIED;}
```

Funkcja pobiera jeden argument lenght1 – jest to długość rysowanej linii wybrana przez użytkownika. Length z kolei jest zmienną używaną przez OpenGL do narysowania linii. W funkcji znajdują się również zmienne newX i newY – odpowiadają one punktom końcowym rysowanej linii. Na końcu zmienna call zmienia wartość na MOVE – inicjuje to odpowiednie funkcje w OpenGL. Tak jak poprzednio funkcja ta nie zwraca żadnej wartości. Kolejną funkcją jest turn() – odpowiedzialne za obrót „żółwia" w punkcie w którym aktualnie się znajduje".

```
static SCM turn (SCM degrees)
{const double value = scm_to_double (degrees);
direction += M_PI / 180.0 * value;
return scm_from_double (direction * 180.0 / M_PI);}
```

Turn() pobiera jedną wartość – użytkownik ustala w niej o ile stopni ma się obrócić „żółw. Funkcja ta zwraca ilość stopni o jakie obróci się „żółw". Liczba tam będzie wyświetlana w wierszu poleceń po wykonaniu wcześniej wpisanego polecenia. Funkcje pendown() oraz penup() odpowiadają za zamianę wartości logicznej zawartej w mpendown.

```
static SCM pendown ()
{SCM result = scm_from_bool (mpendown);
mpendown = 1;
return result;}
static SCM penup ()
{SCM result = scm_from_bool (mpendown);
mpendown = 0;
return result;}
```

Funkcje te nie pobierają żadnych wartości, zmieniają jedynie wartość pendown na 0 lub 1 i zwracają w wierszu poleceń wartości logiczne w Scheme (odpowiednio #f i #t). Funkcja scm_from_bool() przekształca zmienną result tak aby była możliwa do zinterpretowania przez Scheme. Ccolor() jak sama nazwawskazuje zmienia kolor rysowanych obiektów:

```
static SCM ccolor(SCM rcolor, SCM gcolor, SCM bcolor)
{
SCM result;
testrcolor = scm_to_double (rcolor);
testbcolor = scm_to_double (bcolor);
testgcolor = scm_to_double (gcolor);
if (testrcolor <= 1 && testrcolor >= 0 && testgcolor <=1 && testgcolor >= 0 &&
testbcolor<=1 && testbcolor >= 0)
{ SCM result = scm_from_bool (1);
rrcolor = scm_to_double (rcolor);
bbcolor = scm_to_double (bcolor);
ggcolor = scm_to_double (gcolor);
return scm_from_bool(1); }
else { return scm_from_bool(0); }}
```

Funkcja pobiera 3 wartości – po jednak dla każdej składowej RGB (Red, Green, Blue), Wartości te musza być w przedziale od 0 do 1, gdzie 0 odpowiada brakowi występowania danego koloru w danej bardziej, a 1 odpowiada jego pełnemu występowaniu. Kolory mieszają się i powstaje jeden konkretny kolor. W kodzie funkcji znajduje się również zabezpieczenie, które nie dopuszcza, aby użytkownik wprowadził wartość większą od 1 lub mniejszą od 0. Odpowiedzialne za to są zmienne testrcolor, testgcolor i testbcolor – są one konwertowane z typu SCM do double za pomocą funkcji scm_to_double(). Jeżeli ich wartości tych zmiennych nie są błędne, wartości podane wcześniej przez użytkownika

zostają przekazane do zmiennych, które są później użyte przez komendy OpenGL. Powyżej zostały przedstawione tylko niektóre zarejestrowane funkcje – w samym programie można znaleźć również funkcje resetujące położenie x i y żółwia, zmieniające grubość linii czy rysujące proste prymitywy nie wymagające wprowadzania żadnych parametrów.

Poniżej znajduje się funkcja, której wywołanie w późniejszym etapie w funkcji main() powoduje zarejestrowanie wszystkich funkcji, które się w niej znajdują:

```
static void* register_functions (void* data){
scm_c_define_gsubr ("exit_mode", 0, 0, 0, &exit_mode);
scm_c_define_gsubr ("move", 1, 0, 0, &mmove);
scm_c_define_gsubr ("clear", 0, 0, 0, &cclear);
(…)
scm_c_define_gsubr ("turn", 1, 0, 0, &turn);
scm_c_define_gsubr ("recolor", 3, 0, 0, &ccolor);
return NULL;}
```

Komenda scm_c_define_gsubr() rejestruje wcześniej wymienione funkcje – jej argumentami są odpowiednio: nazwa rejestrowanej funkcji w Scheme, wymagane argumenty, argumenty opcjonalne, wartość, która definiuje czy są dostępne argumenty „rest-list” oraz odnośnik do rejestrowanej funkcji. Wymagana jest również funkcja runScheme(), która wywołuje Guile oraz jego wiersz poleceń. Argumentem komendy scm_with_guile() jest odnośnik do funkcji register_functions, dzięki czemu wcześniej zdeklarowane funkcje będą dostępne w ramach wiersza poleceń Guile:

```
void *runScheme( void *ptr )
{char *message;
message = (char *) ptr;
printf("%s \n", message);
scm_with_guile (&register_functions, NULL);
scm_shell (0, NULL);}
```

Funckcja main() jest dwuwątkowa – w jednym działa Guile (umożliwia korzystanie z wiersza poleceń), w drugim wykonywane są odpowiednie komendy w OpenGL:

```
int main (int argc, char* argv[]){
pthread_t thread1, thread2;
```

```
const char *message1 = "Thread 1";
const char *message2 = "Thread 2";
int iret1, iret2;
iret1 = pthread_create( &thread1, NULL, runScheme, (void*) message1);
if(iret1){fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
exit(EXIT_FAILURE);}
glutInit( &argc, argv );
glutInitWindowSize(500, 500);
glutCreateWindow("Rysowanie");
glutDisplayFunc(mydisplay);
glutIdleFunc(idle);
glutMainLoop();
return EXIT_SUCCESS;}
```

Polecenie glutIdleFunc(idle) na bierząco sprawdza i aktualizuje obraz w przypadku braku wprowadzania zmian przez użytkownika.

6.5 Rysowanie wywołanych funkcji

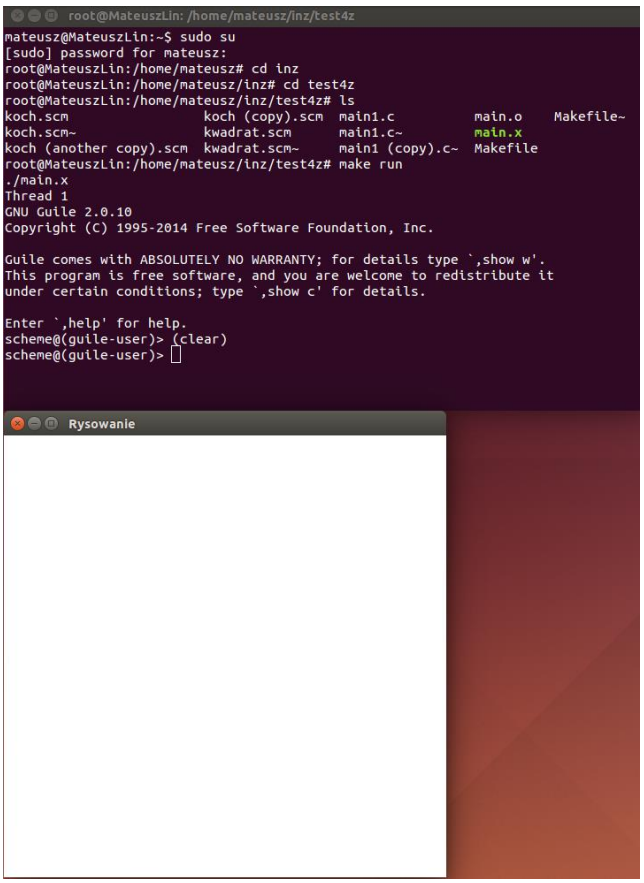
Całość kodu związana z rysowaniem zawarta jest w funkcji MyDisplay():

```
if (call == MOVE)
{if (newX <= 1 && newY <=1 && newX >= -1 && newY >= -1)
{if (mpendown)
{glLineWidth(rwidth);
glColor3f(rrcolor, ggcolor, bbcolor);
glBegin(GL_LINES);
glVertex3f(x, y, 0.0);
glVertex3f(newX, newY, 0);
glEnd();}
x = newX;
y = newY;}}
(…)
if (call == CLEAR) {
glClearColor(1, 1, 1, 1 );
glClear( GL_COLOR_BUFFER_BIT );
x = y = 0.0;
direction = 0.0;
mpendown = 1;}
glFlush(); }
```

Powyżej przedstawiono, niektóre fragmenty kodu zawarte w tej funkcji. Zasada działania jest prosta – funkcja jest na bieżąco wywoływana, więc przy każdej zmianie zmiennej call, na ekranie od razu pojawia się efekt wywołanej przez użytkownika funkcji. We fragmencie kodu odpowiedzialnego za rysowanie linii zostało zamieszczone zabezpieczenie, które uniemożliwia użytkownikowi narysowanie obiektu poza oknem. Obie współrzędne mogą mieć maksymalne wartości równe 1 lub -1. Zostało również zamieszczone zabezpieczenie, aby funkcja nie rysowała linii gdy zmienna mpendown ma wartość 0 – wówczas jedynie współrzędne zmieniają się na te pożądane przez użytkownika. Na samym końcu funkcji znajduje się komenda glFlush(), która odpowiada za wymuszenie wykonania wszystkich podanych wcześniej komend (tych które są dostępne w ramach wywoływania funkcji Scheme).

6.6 Testy programu

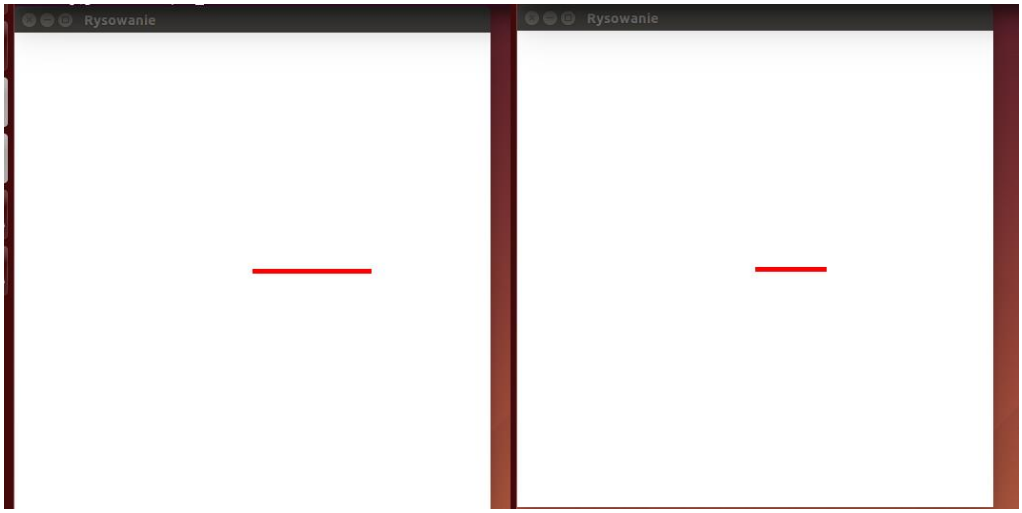
W celu sprawdzenia poprawności działania programu przeprowadzono testy rysowania linii oraz podstawowych prymitywów dostępnych w ramach dodanych wcześniej funkcji do Guile:



Rys.2 Start programu

Jak widać na obrazie zamieszonym na Rys.2 po włączeniu programu pojawia się okno do rysowania obiektów geometrycznych. Ponadto wiersz poleceń jest teraz

połączony z Guile i ten reaguje na komendy tam wpisywane. Wyświetlana jest również nazwa „Thread 1” co jest równoznaczne z uruchomieniem się Guile. Po włączeniu aplikacji wyświetlana jest wersja Guile (w tym przypadku 2.0.10).



Rys.3 Rysowanie linii o różnych długościach.

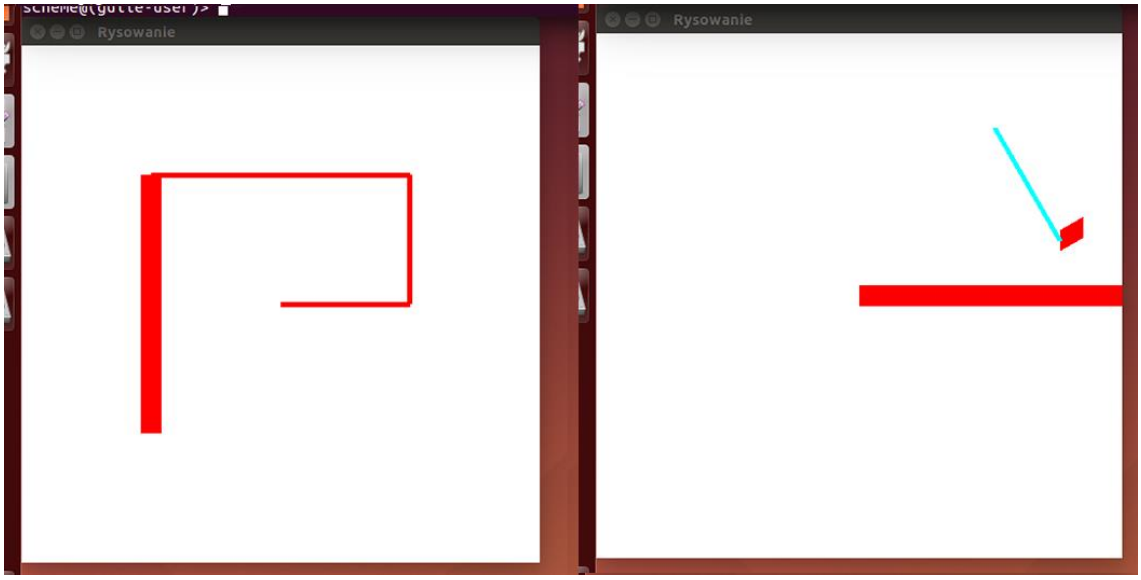
Opcje rysowania linii działają bez zarzutu – co widać na Rys.3. Powyższe linie są wynikiem wywołania komend (move 0.5) i (move 0.3) Po lewej widoczna jest linia, której długość wynosi 0,5 – długości linii po prawej wynosi 0,3. Oczywiście oba obrazy przedstawiają dwa różne wywołania programów – wywołanie powyższych komend po sobie spowodowałoby powstanie jednolitej linii o długości 0,8.

Rys.4 Zmiana koloru rysowania oraz skręcanie.

Rys.4 jest wynikiem działania komend (na obrazie po prawej):

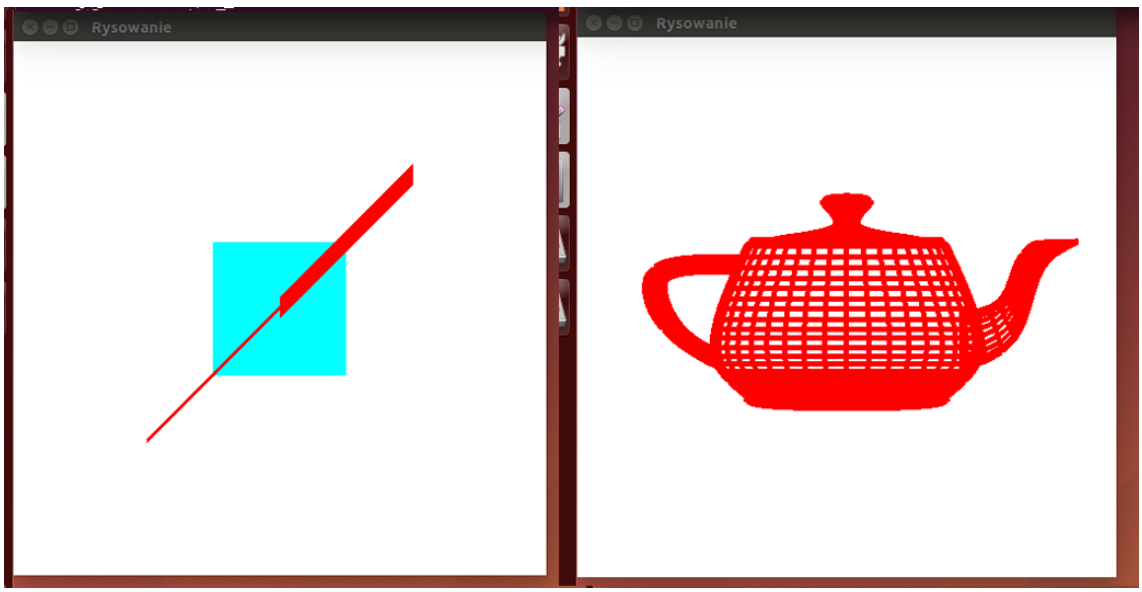
- (move 0.3) – powstanie czerwonej linii,
- (recolor 1 1 0) - wymieszanie kolorów spowodowało powstanie koloru żółtego
- (turn 90) – zmiana kierunku o 90 stopni w kierunku przeciwnym do ruchu wskazówek zegara ,
- (move 0.5) – powstanie żółtej linii,
- (turn 120) – ponowny obrok o 120 w lewo,

- (recolor 1 0.5 0) wymieszanie kolorów spowodowało powstanie koloru pomarańczowego,
- (move 0.2) powstanie pomarańczowe linii.



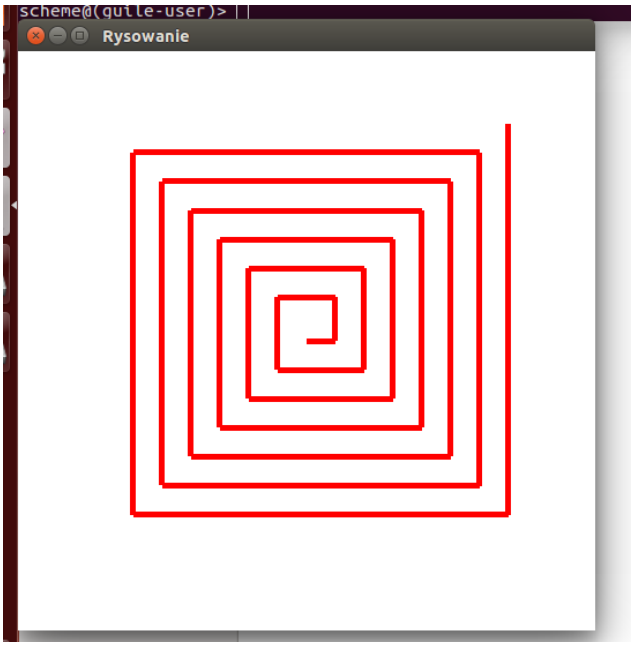
Rys.5 Sprawdzenie działania funkcji (rewidth), (penup) oraz (pendown).

W następnej kolejności przetestowano pogrubianie linii oraz przesuwanie współrzędnych bez konieczności rysowania niechcianej w danym przypadku linii.Na Rys.5 przedstawiono możliwości działania funkcji rewidth(), penup() i pendown(). Pogrubione linie są wynikiem działania funkcji (rewidth 20), która znacząco zwiększyła grubość linii przed jej narysowaniem. W obrazie po lewej można dostrzec linie oderwane od siebie, które są wynikiem użycia funkcji (penup) oraz (pendown) do zmiany współrzędnych bez konieczności zostawiania dodatkowej linii.



Rys.6 Rysowanie predefiniowanych prymitywów bez konieczności podawania jakichkolwiek parametrów.

Rysunek powyżej (Rys.6) przedstawia efekt najprostszych funkcji dodanych w ramach tej aplikacji. Po lewej można zauważyć kwadrat, który wywoływany jest poleceniem (poly) – rysowany jest jednak w oparciu o wcześniej wybrany kolor przez użytkownika. Na tym samym obrazie dostępne są dwie linie wywoływane komendami (hudge_line) oraz (small_line) – ich kolor jest predefiniowany w programie i nie zależy od preferencji użytkownika. Po prawej widoczny jest czajniczek, który wywoływany jest komendą (teapot) – on również korzysta ze zdefiniowanego wcześniej koloru oraz grubości linii.



Rys. 7 Inny przykład tworzenia kształtów.

Jak widać aplikacja działa dosyć sprawnie i nie zdarzają jej się błędy obliczeniowe. Możliwe jest rysowanie bardziej złożonych figur (tak jak przedstawiono na Rys.7)– wymagane jest już wtedy połączenie nowo dodanych komend oraz ogólnej znajomości języka Scheme.

7. Wnioski

Program wydaje się działać prawidłowo, nie jest on jednak pozbawiony wad. Zauważalne jest to, że nie ma możliwości cofnięcia operacji po jej wykonaniu. Przy popełnieniu nieznacznego błędu przy wpisywaniu argumentów danej komendy użytkownik skazany jest na wyczyszczenie całego obrazu i rozpoczęcia pracy od nowa. Szczególnie uciążliwe może to być dla młodszych uczniów, którzy nie mają jeszcze wprawy w sprawnym oporządzaniu programem. W niektórych sytuacjach możliwa jest zmiana koloru rysowania na biały i użycia komendy (move) jako gumki, ale metoda ta może się okazać zawodna w momencie nakładania się narysowanych obiektów przez użytkownika.

Kolejnym problemem jest brak możliwości zapisu wykonanego obrazu do pliku. Wyłączenie programu skutkuje utratą całej pracy, co również może zniechęcić niektórych młodszych użytkowników programu.

Jeżeli chodzi o samą funkcjonalność dodawania nowych funkcji do Guile to wydaje się to być dosyć proste, można więc założyć, że ambitniejsi użytkownicy programu sami będą dodawać nowe rozwiązania, które nie zostały zawarte w tym programie.

Literatura:

1. *Guile Reference Manual, Edition 2.0.11, revision 1, for use with Guile 2.0.11*.
2. Janusz Ganczarski OpenGL, *Podstawy programowania grafiki 3D*, Wydawnictwo Helion.
3. Willian E. Shotts JR *Wprowadzenie do wiersza poleceń*, Wydawnictwo Helion.
4. R. Kent Dybvig The Scheme Programming Language, Fourth Edition, http://www.scheme.com/tspl4/
5. H. Abelson, A. diSessa *Turtle Geometry, The Computer as a Medium for Exploring Mathematics*

Materiały uzupełniające:

I. Lisp - Wikipedia https://pl.wikipedia.org/wiki/Lisp (dostęp styczeń 2016).

II. Scheme – Wikipedia https://pl.wikipedia.org/wiki/Scheme#Elementy_j.C4.99zyka (dostęp styczeń 2016).

III. GNU Multiple Precision Arithmetic Library https://pl.wikipedia.org/wiki/GNU_Multiple_Precision_Arithmetic_Library (dostęp styczeń 2016).

IV. GNU Libtool https://en.wikipedia.org/wiki/GNU_Libtool (dostęp styczeń 2016).

V. GNU Libc https://pl.wikipedia.org/wiki/GNU_C_Library (dostęp styczeń 2016).

VI. Libffi - https://sourceware.org/libffi/_ (dostęp styczeń 2016).

VII. PKG-CONFIG https://en.wikipedia.org/wiki/Pkg-config (dostęp styczeń 2016)

VIII. Make – Tworzenie Makefile http://www.programuj.com/artykuly/linux/makefile.php (dostęp styczeń 2016)

IX. OpenGL - Wikipedia https://pl.wikipedia.org/wiki/OpenGL (dostęp styczeń 2016)

X. Freeglut – Wikipedia https://pl.wikipedia.org/wiki/Freeglut (dostęp styczeń 2016)