Projekt nr 45

PROSTY EDYTOR GRAFIKI 3D

Wiktor Hładki
Michalina Oleksy
Franciszek Kramarczyk

1. Opis projektu

Celem projektu jest stworzenie prostego edytora grafiki 3D. Podstawowymi funkcjonalnościami jest umożliwienie użytkownikowi stworzenie kilku prostych brył oraz wykonywanie na nich różnych przekształceń i operacji.

2. Założenia wstępne przyjęte w realizacji projektu

Interfejs graficzny składa się z czterech obszarów roboczych, które umożliwiają wyświetlenie aktualnej bryły z różnych perspektyw, konsoli, pola odpowiedzialnego za wyświetlanie danych wyjściowych z konsoli i pola odpowiedzialnego za wyświetlanie listy stworzonych obiektów wraz z ich numerami. Użytkownik komunikuje się z programem przy użyciu konsoli w programie za pomocą prostych komend.

3. Analiza projektu

Dane wejściowe i dane wyjściowe, komunikacja z użytkownikiem

Dane wejściowe są wpisywane w konsoli programu, dane wyjściowe są wyświetlane w polu danych wyjściowych - dane wyjściowe mogą zawierać jedynie listę stworzonych obiektów z ich numerami id. Do dyspozycji użytkownika jest kilka prostych komend:

- *help* - komenda help służy do wyświetlenia pomocy. Pokazuje ona listę wszystkich komend wraz z wyjaśnieniami dla każdej z nich i listą wymaganych argumentów.
- *help [nazwa komendy]* - służy do wyświetlania pomocy dla komendy określonej odpowiednią nazwą, wyświetla też wymagane argumenty.
- *set_line_color r g b* - służy do zmiany koloru rysowanych obiektów. Nie powoduje pojawienia się danych wyjściowych.
- *line x1 y1 z1 x2 y2 z2* - rysuje linię łączącą dwa punkty o zadanych współrzędnych. W obszarach roboczych pojawiają się odpowiednie widoki, a w liście obiektów pojawia się obiekt wraz z id: *id line (x1, y1, z1) (x2, y2, z2).*
- *box x1 y1 z1 x2 y2 z2* - rysuje prostopadłościan o zadanych narożach. W obszarach roboczych pojawiają się odpowiednie widoki, a w liście obiektów pojawia się obiekt wraz z id: *id box (x1, y2, z1) (x2, y2, z2).*
- *sphere x y z r n m* - rysuje sferę o środku w punkcie (x,y,z), promieniu r, (n,m) oznacza na ile „południków” i „równoleżników” ma być podzielona kula. W obszarach roboczych pojawiają się odpowiednie widoki, a w liście obiektów pojawia się obiekt wraz z id: *id sphere (x, y, z) r (n, m).*
- *cone x1 y1 z1 r1 x2 y2 z2 r2 n* - rysuje stożek o podstawach w zadanych punktach i odpowiednich promieniach podstaw podzielony na n czworokątów. W obszarach roboczych pojawiają się odpowiednie widoki, a w liście obiektów pojawia się obiekt wraz z id: *id cone (x1, y1, z1) r1 (x2, y2, z2) r2 n.*
- *cylinder x1 y1 z1 x2 y2 z2 r n* - Rysuje cylinder o zadanym promieniu podzielony na n czworokątów. W obszarach roboczych pojawiają się odpowiednie widoki, a w liście obiektów pojawia się obiekt wraz z id: *id cylinder (x1, y1, z1) (x2, y2, z2) r n.*
- *delete [id]* - usuwa obiekt o zadanym numerze id.
- *clear_all* - kasuje wszystkie obiekty, usuwa listę obiektów z konsoli. Reprezentacja 3D obiektów zostaje usunięta.
- *move [id] x y z* - przesuwa zadany obiekt o wektor (x, y, z). Reprezentacja obiektu na obszarach roboczych ulega przekształceniu - odpowiedniemu przesunięciu. Jeżeli obiekt nie został znaleziony, to użytkownik jest o tym powiadamiany za pomocą odpowiedniego komunikatu wyświetlanego w konsoli danych wyjściowych.

- *rotate [id] x y z α β γ* - obraca obiekt wokół punktu (x, y, z) o zadane kąty (α, β, γ). Reprezentacja obiektu na obszarach roboczych ulega przekształceniu - odpowiedniemu obrotowi. Jeżeli obiekt nie został znaleziony, to użytkownik jest o tym powiadamiany za pomocą odpowiedniego komunikatu wyświetlanego w konsoli danych wyjściowych.
- *save [name]* - zapisuje dane obiektu w pliku “name”.
- *load [name]* - odczytuje dane z pliku “name”. Tworzy na tej podstawie nowy obiekt i wyświetla go w liście obiektów z odpowiednim id oraz rysuje jego graficzną reprezentację z różnych perspektyw w obszarach roboczych. Jeżeli nie może stworzyć obiektu z danych, to w konsoli danych wyjściowych wyświetla odpowiednią informację.

Struktury danych

- Każdy obiekt 3D jest instancją odpowiedniej klasy. Podstawowa klasa obiektu zawiera informacje o numerze stworzonego obiektu i jego typie, a klasy wyspecjalizowane posiadają takie informacje jak na przykład charakterystyczne współrzędne punktu czy promień, czyli dane podane w komendzie przez użytkownika. Ponadto każda klasa zawiera wektor szczegółowych danych obiektu wykorzystywanych do renderowania reprezentacji 3D, a także przydatnych podczas zapisu czy odczytu z pliku.
- Komendy posiadają klasę bazową odpowiednią do zdefiniowania interfejsu komend oraz wyspecjalizowane klasy które nadpisują na swój użytek odpowiednie metody, szczególnie metodę *Execute()* odpowiedzialną za wykonanie danej komendy.
- Parser danych wejściowych od użytkownika został zaimplementowany jako część GUI, konkretnie klasy *MyFrame* jako główna funkcja parsująca, która korzysta z kilku funkcji pomocniczych, które sprawdzają czy podane przez użytkownika dane są poprawne.

Podział projektu

- Opracowanie logiki projektu i wykonanie jego podstawowej struktury
- Implementacja parsera
- Implementacja podstawowej klasy komend
- Implementacja poszczególnych komend, w tym implementacja algorytmów
- Implementacja podstawowej klasy obiektu 3D
- Implementacja poszczególnych obiektów
- Napisanie dokumentacji
- Przeprowadzenie testów

Narzędzia programistyczne

Projekt tworzony był na trzech środowiskach, Windows, Linux oraz macOS ze względu na to, że każdy z nas dysponował innym. Głównie kompilowano projekt przez MSVC oraz gcc. Z dodatkowych bibliotek użyto jedynie wxWidgets.

4. Podział pracy i analiza czasowa

Pierwszy tydzień:

1. Opracowanie logiki projektu - *Wiktor Hładki*
2. Wykonanie podstawowej struktury projektu - *Wiktor Hładki*
3. Uzupełnienie struktur wszystkich modułów komend oraz obiektów - *Michalina Oleksy*

Drugi tydzień:

4. Napisanie parsera - *Wiktor Hładki*
5. Uzupełnienie szczegółowych danych właściwych dla każdego z obiektów potrzebnych do wykonywania odpowiednich komend - *Franek Kramarczyk*

Trzeci tydzień:

6. Uzupełnienie szczegółowych danych właściwych dla każdej komendy - implementacja wykonania komendy na podstawie danych obiektów - *Wiktor Hładki, Michalina Oleksy*

Czwarty tydzień:

7. Wykonanie dokumentacji - *Michalina Oleksy*
8. Napisanie logiki testów - *Franek Kramarczyk*
9. Przeprowadzenie testów i wykonanie raportu - *Franek Kramarczyk*

5. Kodowanie

Komendy

Podstawowa struktura każdej komendy mieści się w klasie BaseCommand. Każda instancja Base Command zawiera pola:

- *name* - przechowujące nazwę komendy, wypełniane w liście inicjalizacyjnej konstruktora.
- *nArguments* - przechowujące liczbę argumentów, które ma przyjmować dana komenda, wypełniane w liście inicjalizacyjnej konstruktora.

Klasa ta zawiera także proste funkcje zwracające wartości tych pól, takie jak:

- *GetName* - zwracające nazwę komendy.
- *GetArgumentsCount* - zwracające ilość argumentów, które przyjmuje dana komenda.

Oprócz tego klasa składa się także z metod, które mają być nadpisane w klasach dziedziczących:

- *Execute* - służące do wykonania danej komendy.
- *Args* - zwracające argumenty w stringu.
- *Help* - zwracające opis komendy w stringu.

Dla poszczególnych komend metoda *Execute* wygląda następująco:

- Dla klasy *ClearAll* komenda ta przechodzi po liście obiektów i usuwa każdy z nich, następnie aktualizuje wyświetlaną listę stworzonych obiektów i czyści historię komend.
- Dla klasy *Delete* komenda ta tworzy iterator i szuka obiektu o wymaganym id w liście obiektów, a następnie usuwa go i aktualizuje listę wyświetlanych obiektów.
- Dla klasy *Load* tworzone jest okno dialogowe, a następnie wybrany plik jest wczytywany linia po linii i tworzone są obiekty odpowiadające danym, które następnie są wyświetlane. Jeżeli nie uda się rozpoznać danych jako obiekt w konsoli danych wyjściowych jest wyświetlana odpowiednia wiadomość.
- Dla klasy *Save* tworzone jest okno dialogowe, a następnie wszystkie dostępne obiekty są wpisywane do wybranego pliku jako dane tekstowe.
- Dla klasy *SetLineColor* pobierane są trzy liczbowe argumenty, sprawdzane jest czy mieszczą się w przedziale wartości odpowiednich dla RGB i zmieniamy jest kolor linii dla wszystkich narysowanych obiektów.
- Dla klasy *Rotate* najpierw wczytywane są odpowiednie argumenty - współrzędne, kąty oraz id, następnie obiekt jest obracany przy użyciu metody *Rotate*, a potem lista obiektów jest aktualizowana. Jeżeli obiekt o danym id nie został znaleziony, to użytkownik jest powiadamiany za pomocą odpowiedniego komunikatu, wyświetlanego w konsoli danych wyjściowych.

- Dla klasy *Move* najpierw wczytywane są odpowiednie argumenty - współrzędne oraz id, następnie obiekt jest przenoszony przy użyciu metody *Move*, a potem lista obiektów jest aktualizowana. Jeżeli obiekt nie został znaleziony, to użytkownik jest o tym powiadamiany za pomocą odpowiedniego komunikatu wyświetlanego w konsoli danych wyjściowych.
- W klasie *PrintHelp* przechodzi ona przez wszystkie dostępne komendy i wyświetla dla nich wyjaśnienie działania. Jeśli jednak podamy w argumentach nazwę funkcji, to tworzony jest iterator, który przeszukuje listę funkcji i wyświetla odpowiednią pomoc, a jeżeli funkcja nie została znaleziona to użytkownik jest powiadamiany odpowiednim komunikatem.
- W klasach *CreateLine, CreateBox, CreateCone, CreateCylinder* oraz *CreateSphere* najpierw wczytuje się odpowiednie argumenty, takie jak na przykład wymagane współrzędne lub promień, następnie tworzony jest nowy obiekt, wyświetla się jego graficzna reprezentacja w trzech obszarach roboczych, a lista obiektów jest aktualizowana.

Obiekty

Podstawowa struktura każdego obiektu mieści się w klasie BaseObject. Każda instancja BaseObject zawiera pola:

- *objId* - numer stworzonego obiektu wypełniane w liście inicjalizacyjnej konstruktora.
- *type* - typ stworzonego obiektu wypełniany w liście inicjalizacyjnej konstruktora
- *points* - pole, które zawiera wszystkie punkty, które tworzą obiekt, potrzebne do wykonania renderowania, a także do zapisu do pliku.

Klasa ta zawiera także proste funkcje pomocnicze, które służą do ułatwienia dostępu do danych, takie jak:

- *GetId*, które zwraca id obiektu,
- *GetData*, które zwraca zawartość pola points,
- *GetCoordinatesString*, które zwraca w stringu specjalnie sformatowane współrzędne punktu podanego w wektorze jako argument metody,
- *Repr*, odpowiedzialne za zwrócenie typu w charakterze stringa.

Oprócz tego klasa składa się również z innych metod:

- *Rotate,* odpowiedzialnej za obrócenie obiektu,
- *Move*, odpowiedzialnej za przeniesienie obiektu.

Te metody są bardzo istotne dla funkcji move i rotate. Oprócz tego w klasie są jeszcze następujące metody:

- *MoveOrigins*, nadpisywana przez klasę dziedziczącą i używana przez metodę Move,
- *GeneratePoints*, także do nadpisania przez klasę dziedziczącą, która po prostu generuje punkty właściwe dla danego obiektu 3D.
- *Copy*, również nadpisywane przez klasę dziedziczącą pozwala na utworzenie nowego obiektu tego samego typu, jest wykorzystywane przez funkcję load.
- *SetData* inicjalizuje dane obiektu za pomocą dwóch stringów, jest wykorzystywana w funkcji load do tworzenia nowych obiektów z wczytywanego pliku
- *GetSaveData* przygotowuje reprezentację obiektu dla funkcji save.

Specyficzne klasy obiektów zawierają pola z danymi odpowiednimi dla typu:

- Klasa *Line* posiada dwa wektory *start* i *end*, które określają początek i koniec linii
- Klasa *Box* zawiera dwa wektory *start* i *end*, które określają naroża obiektu
- Klasa *Sphere* zawiera pole *pos*, w którym znajdują się współrzędne środka sfery, *radius*, które zawiera wartość promienia oraz *meridians* i *parallels*, czyli równoleżniki i południki, które określają jak ma być podzielona kula
- Klasa *Cone* zawiera wektory *start* i *end*, które określają dwie podstawy o promieniach odpowiednio *radius1* i *radius2* oraz pole *nTetragons*, które określa na ile czworokątów podzielony jest stożek
- Klasa *Cylinder* również zawiera wektory *start* i *end*, które określają podstawy, a także pole *radius*, które zawiera wartość promienia oraz *nTetragons*, które tak jak w przypadku stożka określa na ile czworokątów podzielona jest figura

Reszta schematu poszczególnych obiektów jest bardzo prosta. Są to:

- Własny konstruktor, który najpierw tworzy *BaseObject* z odpowiednim typem i wypełnia pola właściwe dla obiektu opisane powyżej
- Nadpisana metoda *MoveOrigins* używa w funkcji move
- Nadpisana metoda *GeneratePoints* właściwa dla obiektu danego typu
- Metodę *Repr*, która używa *Repr* należącego do *BaseObject*, a potem dopisuje właściwości odpowiednie dla konkretnego obiektu, czyli przetwarza jego pola, które opisano wcześniej do stringa

Oprócz tego w klasie *MainFrame* znajdują się dwie metody, które są wykorzystywane przez komendy programu:

- *UpdateObjList* nieprzyjmujące żadnych argumentów, które czyści listę obiektów i wyświetla ją ponownie z uwzględnieniem wszystkich zmian naniesionych przez komendy.
- *UpdateObjList*, które jako argument przyjmuje nowy obiekt i dodaje go na koniec listy wszystkich istniejących obiektów i wyświetla go na liście obiektów.

Parser

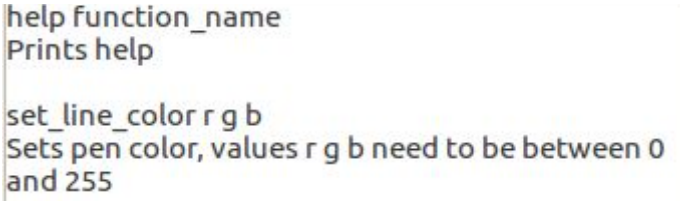
Parser został zaimplementowany jako część klasy *MainFrame*. Główną metodą, która tworzy parser jest *ParseConsoleInput*. Najpierw pobiera ona wszystko co wpisał użytkownik w konsoli danych wejściowych i tworzy ze wszystkiego string *input*. Następnie string ten dzielony jest na słowa i umieszczany w wektorze stringów *args.* Kolejno wszystko co wypisuje ta metoda jest przekierowywane na konsolę danych wyjściowych. Tworzony jest iterator, który przechodzi po liście komend i szuka tej, która została podana przez użytkownika. Jeżeli komenda nie została znaleziona wyświetlany jest odpowiedni komunikat. Jeśli jednak udało się znaleźć komendę, to sprawdzana jest liczba podanych argumentów. W przypadku, gdy nie jest ona odpowiednia użytkownik jest o tym powiadamiany, a jeżeli jest inaczej sprawdzany jest typ argumentów. Dla wszystkich funkcji oprócz *help, save* oraz *load* argumenty muszą być typu int lub double i za sprawdzenie tego odpowiedzialna jest funkcja *CheckArgumentsIfIntOrDouble*, która z kolei wykorzystuje funkcję *isNumber*. Jeżeli argumenty mają nieodpowiedni typ wyświetla się komunikat informujący o tym użytkownika. W przeciwnym wypadku odpowiednia komenda zostaje wykonana, a konsola danych wejściowych jest czyszczona.

6. Testowanie, wdrożenie, raport i wnioski

Poniżej znajdują się przykłady wszystkich wykonanych funkcji pokazujących w jaki sposób zadziałało wpisanie odpowiednich danych przez użytkownika.

Help

Po wpisaniu w konsoli danych wejściowych komendy *help* wyświetlają się wszystkie dostępne komendy wraz z ich opisem. W celu przejrzenia wszystkich informacji konieczne jest przesunięcie w dół w konsoli danych wyjściowych.

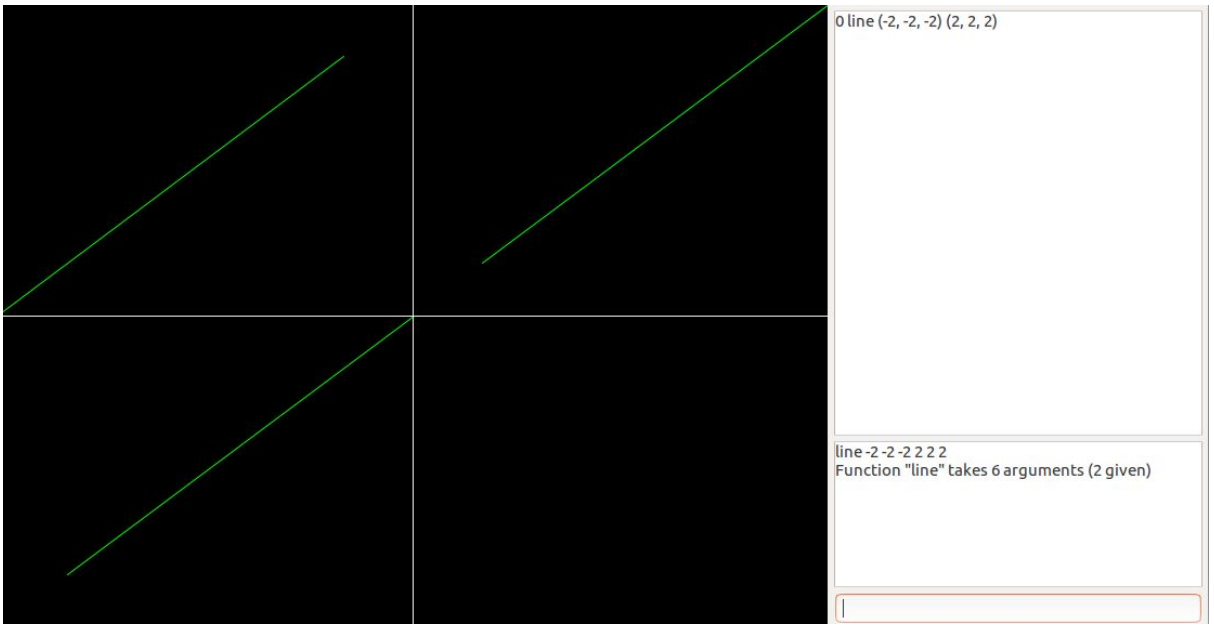


Następnie przetestowano komendę help dla wszystkich dostępnych funkcji - wszystkie wyświetliły się poprawnie. Poniżej można zobaczyć przykład dla *help set_line_color:*

Jeżeli w argumentach funkcji help podano nieprawidłową nazwę funkcji wyświetlany jest odpowiedni komunikat:

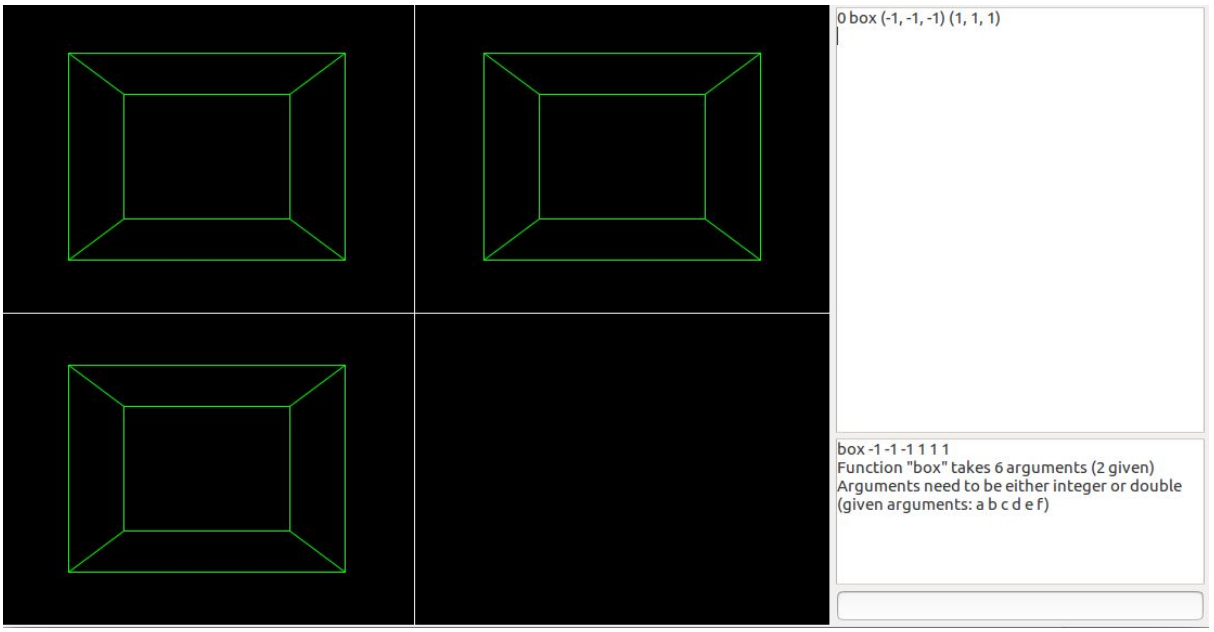
Line

Po poprawnym utworzeniu obiektu linii wyświetla się jego reprezentacja graficzna w trzech obszarach roboczych. Jeżeli podamy nieprawidłową liczbę argumentów lub argumenty w złym formacie, jesteśmy o tym informowani.



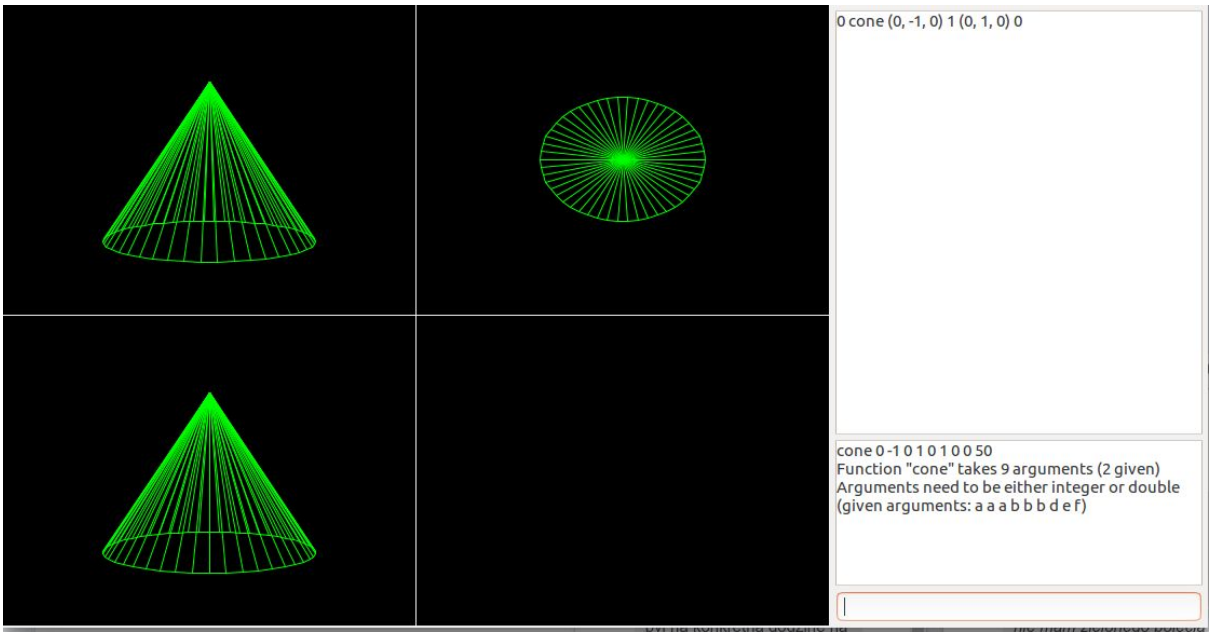
Box

Po poprawnym utworzeniu obiektu prostopadłościanu wyświetla się jego reprezentacja graficzna w trzech obszarach roboczych. Jeżeli podamy nieprawidłową liczbę argumentów lub argumenty w złym formacie, jesteśmy o tym informowani.



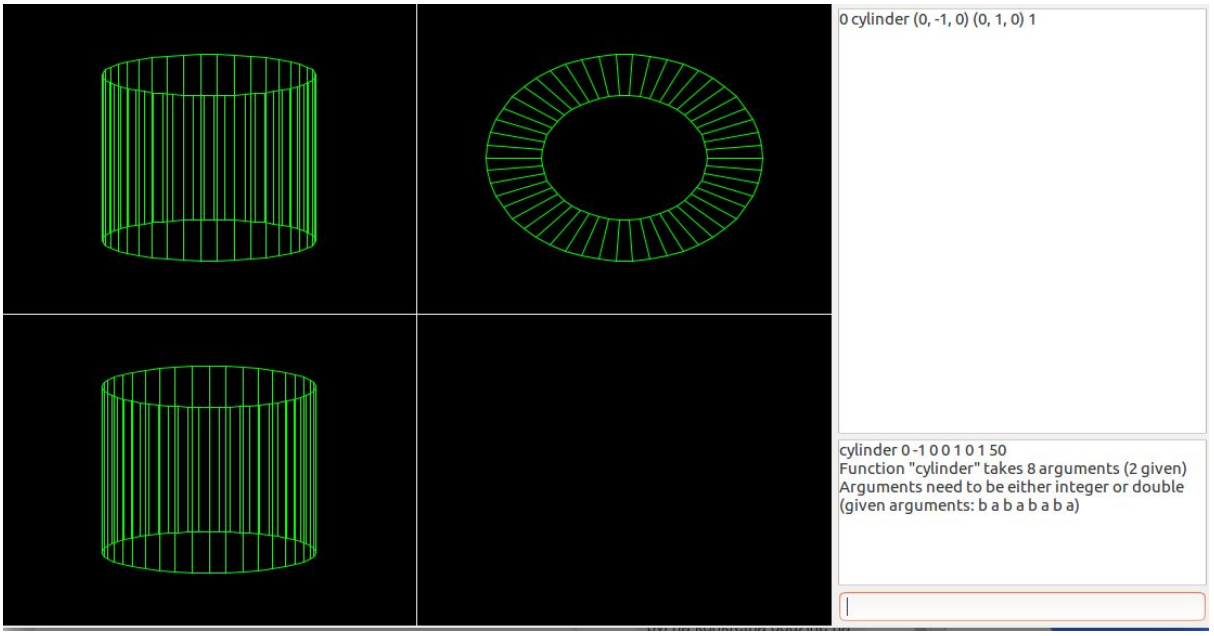
Cone

Po poprawnym utworzeniu obiektu stożka wyświetla się jego reprezentacja graficzna w trzech obszarach roboczych. Jeżeli podamy nieprawidłową liczbę argumentów lub argumenty w złym formacie, jesteśmy o tym informowani.



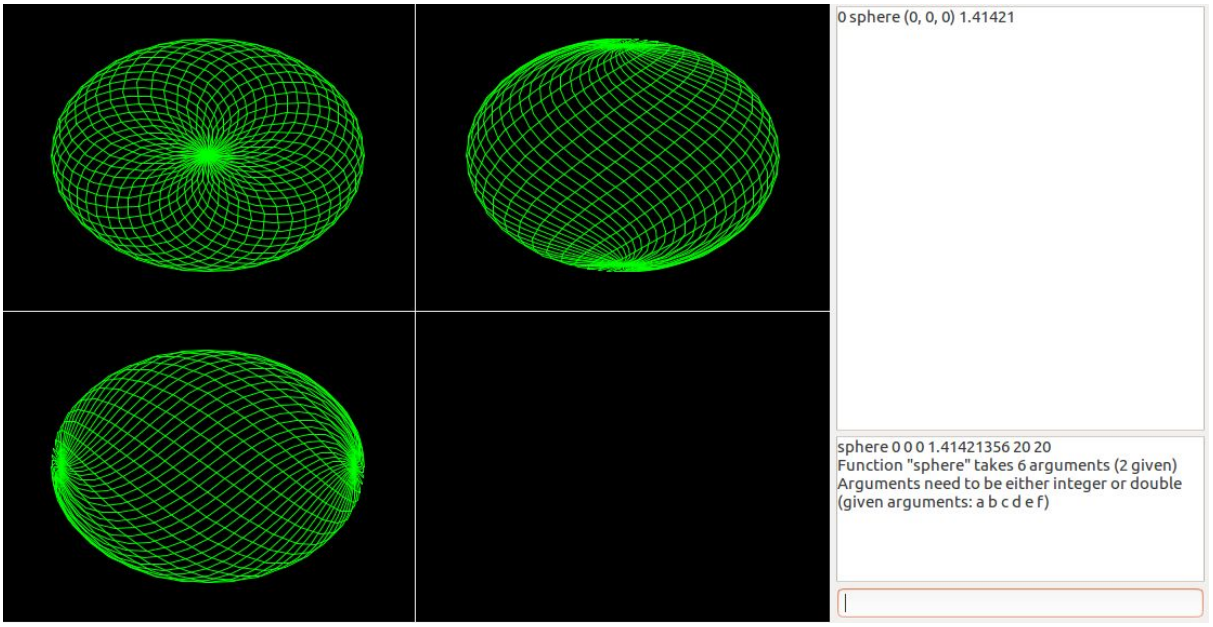
Cylinder

Po poprawnym utworzeniu obiektu cylindra wyświetla się jego reprezentacja graficzna w trzech obszarach roboczych. Jeżeli podamy nieprawidłową liczbę argumentów lub argumenty w złym formacie, jesteśmy o tym informowani.



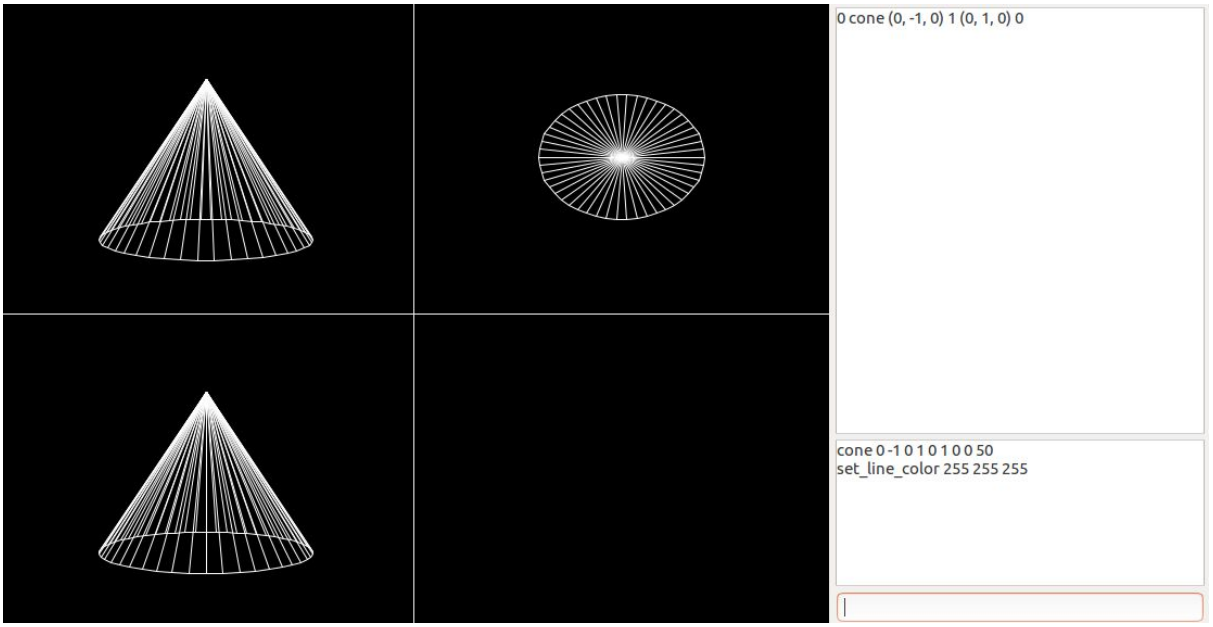
Sphere

Po poprawnym utworzeniu obiektu kuli wyświetla się jego reprezentacja graficzna w trzech obszarach roboczych. Jeżeli podamy nieprawidłową liczbę argumentów lub argumenty w złym formacie, jesteśmy o tym informowani.



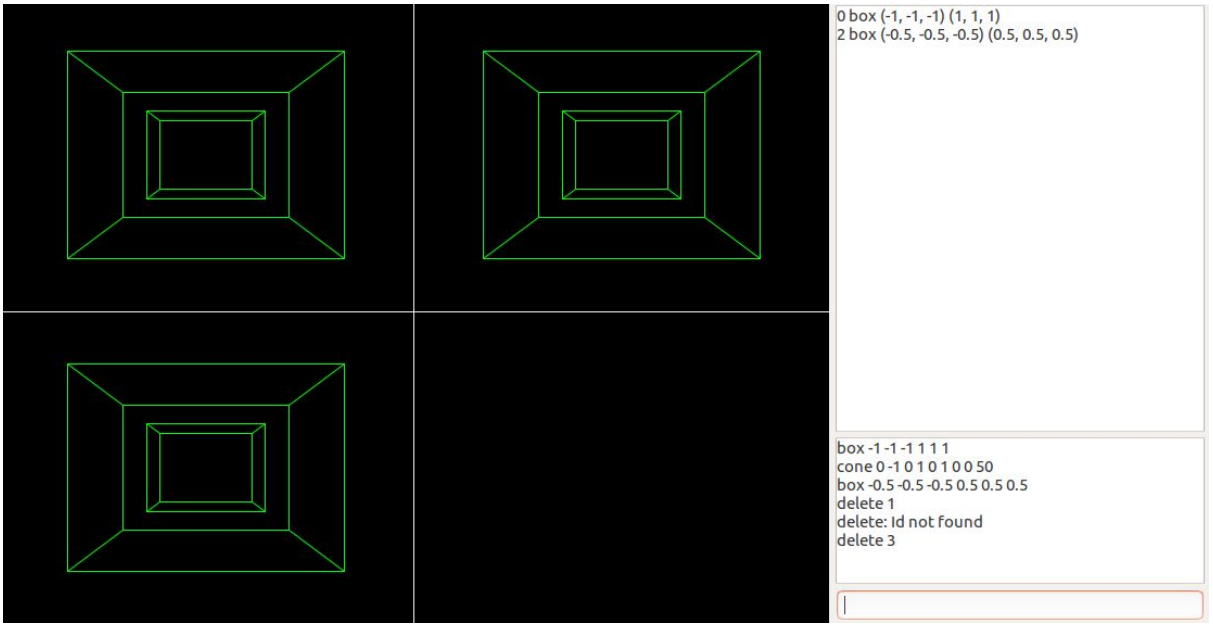
Set Line Color

Po wpisaniu komendy z odpowiednimi wartościami (trzy wartości liczbowe od 0 do 255 kolor rysowanego obiektu zmienia się, a w polu danych wyjściowych widzimy wpisaną komendę.



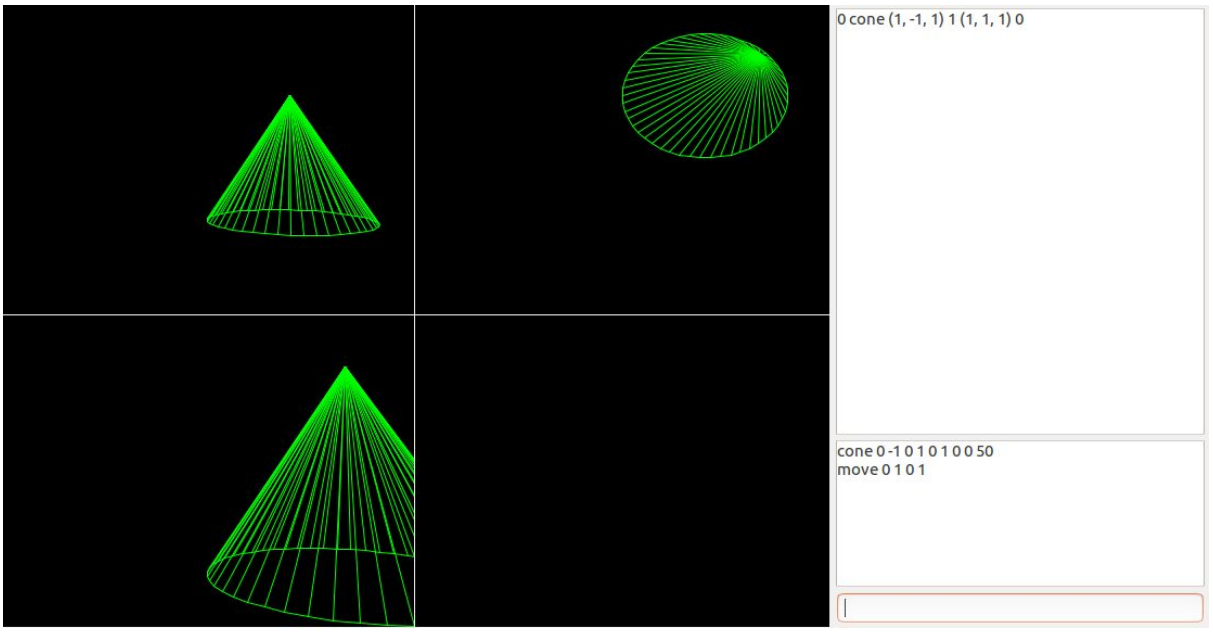
Delete

Po wpisaniu komendy *delete* i odpowiedniego id obiektu dany obiekt znika z ekranów roboczych i listy obiektów. Jeżeli podamy złe id jesteśmy o tym informowani. Na obrazku widać informację o niepoprawnym id. W konsoli danych wyjściowych widać utworzone 3 obiekty, a w liście obiektów istnieją tylko dwa, na obszarach roboczych również widać tylko dwa z nich - to efekt wykonania komendy delete.



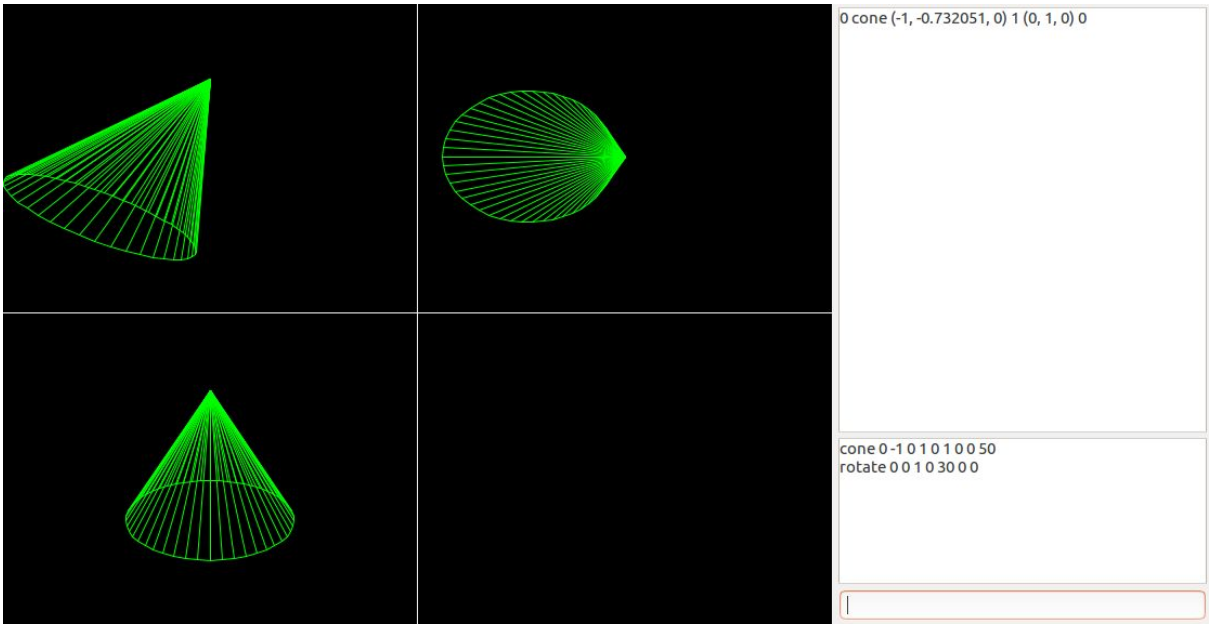
Move

Utworzono taki sam obiekt stożka jak w poprzednio, a następnie przesunięto go odpowiednio. Efekt operacji można zobaczyć na poniższym obrazku:



Rotate

Utworzono taki sam obiekt stożka jak w poprzednio, a następnie obrócono go odpowiednio. Efekt operacji można zobaczyć na poniższym obrazku:

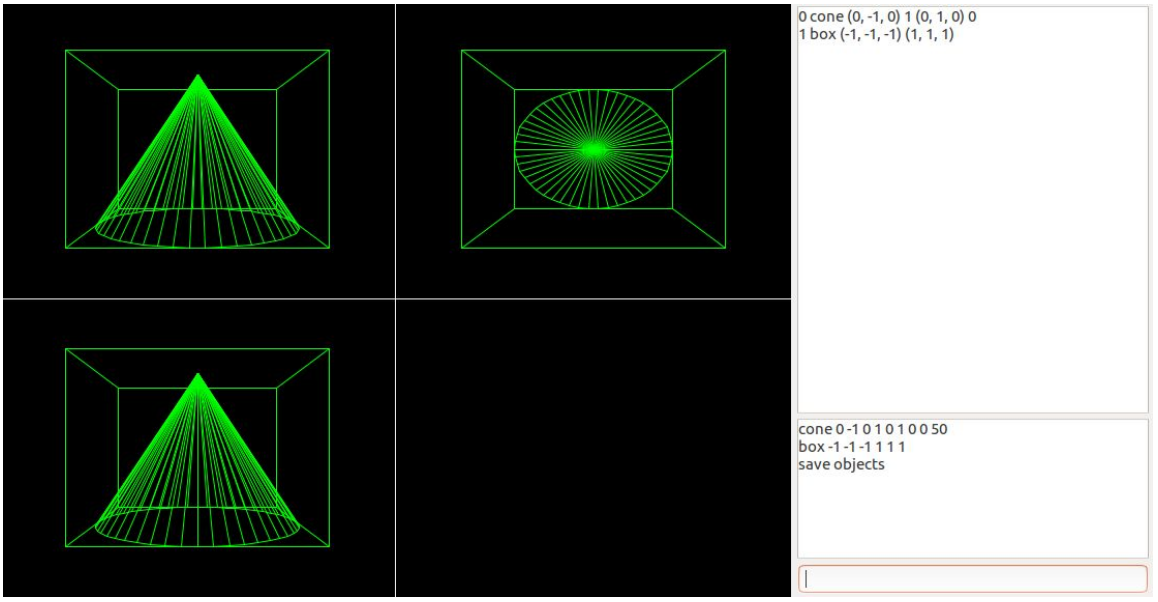


Save

Po stworzeniu obiektów i zapisaniu ich został utworzony odpowiedni plik, który zawierał dane obiektów - *objects.geo.* Na obrazku widać utworzony obiekt, a w konsoli danych wyjściowych widać komendę *save*. Plik wyglądał następująco:

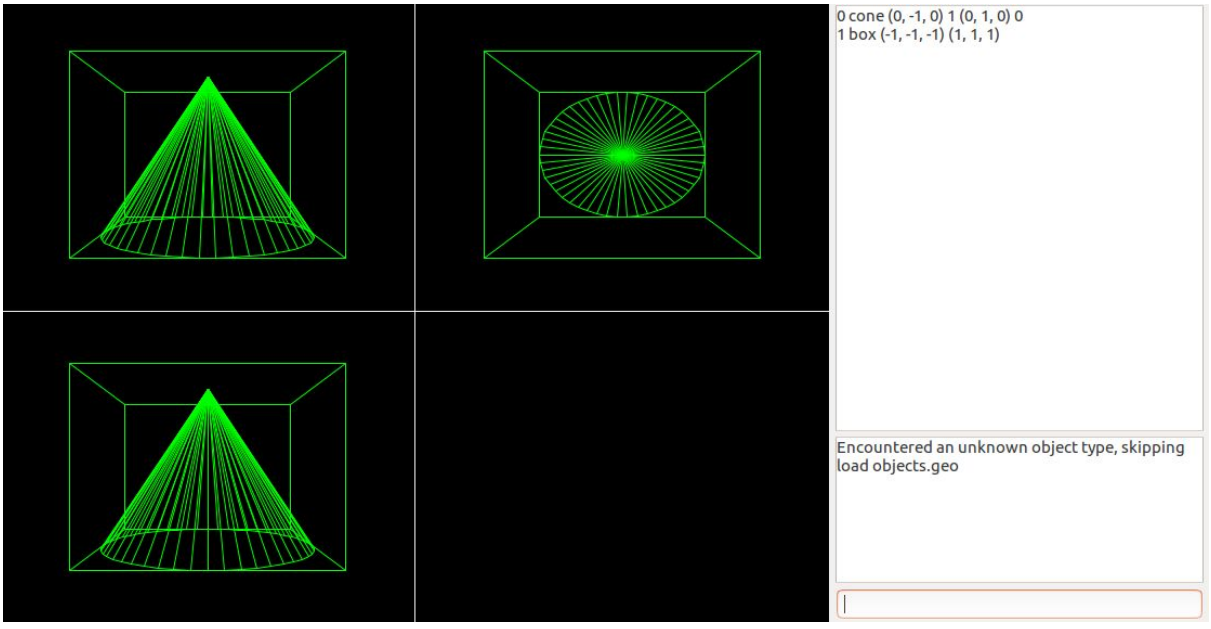
cone
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
0 -1 0 0 1 0 1 0 50
box
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

-1 -1 -1 1 1 1



Load

Po wczytaniu kilku obiektów z pliku na obszarach roboczych wyświetla się ich reprezentacja i tworzona jest lista obiektów. Jeżeli któreś dane były niepoprawne użytkownik jest o tym informowany w konsoli danych wyjściowych. Do pliku na końcu wpisano ciąg znaków “aaaaa”.



W projekcie udało się dobrze zaimplementować wszystkie wymagania podstawowe. Obiekty wyświetlają się poprawnie, a jeżeli dane wejściowe były niepoprawne, to użytkownik jest o tym informowany i nie następuje próba wykonania komend ze złymi danymi. W przyszłości można pomyśleć nad lepszym formatowaniem danych w konsoli danych wyjściowych oraz nad dodaniem dodatkowych komend, które ułatwiłyby pracę w edytorze.

Clear All

Po użyciu komendy *clear_all* z obszarów roboczych znikają wszystkie obiekty, a lista obiektów i konsola są czyszczone.