Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(УНИВЕРСИТЕТ ИТМО)

Факультет «Систем управления и робототехники»

ОТЧЕТ

О ЛАБОРАТОРНОЙ РАБОТЕ №3

По дисциплине «Практическая линейная алгебра»
на тему: «Матрицы в 3D-графике»

Студенты:
Бахтаиров Р.А. группа R3243
Сайфуллин Д.Р группа R3243
Симонов И.А. группа R3236
Холухина Д.Е. группа R3240

Проверил:
Догадин Егор Витальевич, ассистент

Санкт-Петербург
2024

Содержание

1 Введение

В данной лабораторной работе исследуются основные преобразования в 3D-графике с использованием матриц. В рамках работы изучаются такие виды преобразований, как масштабирование, перемещение, вращение и перспективное проецирование. Это поможет понять, как создаются и визуализируются трехмерные сцены, и станет хорошей основой для дальнейшего изучения графических технологий.

2 Ход работы

2.1 Создание куба

Для выполнения задачи по построению куба в 3D-пространстве на языке Python используется библиотека `matplotlib`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

vertices_cube = np.array([
    [-1, 1, 1, -1, -1, 1, 1, -1],
    [-1, -1, 1, 1, -1, -1, 1, 1],
    [-1, -1, -1, -1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1]
])
```

В данной части кода определяются координаты вершин куба с использованием массива `vertices_cube`. Куб состоит из 8 вершин, каждая из которых описана в гомогенных координатах (четырехкомпонентный вектор). Последний столбец каждой вершины равен 1, что позволяет использовать однородные координаты для матричных преобразований.

```
faces_cube = np.array([
    [0, 1, 5, 4],
    [1, 2, 6, 5],
    [2, 3, 7, 6],
    [3, 0, 4, 7],
    [0, 1, 2, 3],
    [4, 5, 6, 7]
])
```

Массив `faces_cube` описывает грани куба. Каждая грань задается индексами четырех вершин, которые её образуют. Использование индексов упрощает создание граней и позволяет легко изменять порядок вершин, если требуется.

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d', proj_type = 'ortho')

def draw_shape(vertices, faces, color):
    vertices = (vertices[:3, :] / vertices[3, :]).T
    ax.add_collection3d(Poly3DCollection(vertices[faces],
    facecolors=color, edgecolors='k', linewidths=0.2))
```

```

draw_shape(vertices_cube, faces_cube, 'blue')

ax.set_box_aspect([1,1,1])
ax.set_xlim(-1, 1); ax.set_ylim(-1, 1); ax.set_zlim(-1, 1)
ax.view_init(azim=-37.5, elev=30)
ax.set_xticks(np.linspace(-1, 1, 5)); ax.set_yticks(np.linspace(-1, 1, 5)); ax.set_zticks(np.linspace(-1, 1, 5))

plt.show()
```

Здесь создается новая фигура для графика и добавляется трехмерная ось с ортогональной проекцией. Функция `draw_shape` вызывается для отрисовки куба с заданными вершинами и гранями, что позволяет визуализировать куб в 3D-пространстве. Эта функция выполняет несколько шагов:

- Сначала вершины переводятся в трехмерные координаты, так как изначально они заданы в гомогенных координатах.
- Затем создается коллекция граней с помощью `Poly3DCollection`, и каждая грань добавляется на график.
- Устанавливается равномерный масштаб осей с помощью `set_box_aspect([1, 1, 1])`.
- Границы осей устанавливаются от -1 до 1 по каждой из трех осей.
- Вид камеры на график задается азимутальным углом -37.5° и углом возвышения 30°, что позволяет получить объемное представление куба.
- Настраиваются метки осей для удобства визуального восприятия.
- `plt.show()` отображает график.

Таким образом, в результате выполнения данного кода получается изображение куба, отрисованного в трехмерной ортогональной системе координат.

2.2 Изменение масштаба кубика

Первое преобразование, которое нам предстоит сделать в пространстве — масштабирование или, по-другому говоря, умножение каждой из координат вектора на константы. К примеру, увеличим x-координату увеличим в 1.5 раза, y-координатууменьшим в 2 раза, а z-координату увеличим в 2 раза. Масштабирование реализуем с помощью матрицы A, определенной как:

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

где s_x, s_y и s_z — коэффициенты масштабирования по осям x, y и z соответственно. Если $s_x = s_y = s_z = 1$, объект сохраняет свои размеры, а если $s_x, s_y, s_z > 1$, объект увеличивается.

```python
def scale_matrix(sx, sy, sz):
    return np.array([
        [sx, 0, 0, 0],
        [0, sy, 0, 0],
        [0, 0, sz, 0],
        [0, 0, 0, 1]
    ])

def apply_transformation(vertices, transformation_matrix):
    return transformation_matrix @ vertices

sx, sy, sz = 1.5, 0.5, 2.0
S = scale_matrix(sx, sy, sz)

scaled_vertices_cube = apply_transformation(vertices_cube, S)
draw_shape(scaled_vertices_cube, faces_cube, 'blue')

ax.set_box_aspect([1,1,1])
ax.set_xlim(-2, 2); ax.set_ylim(-2, 2); ax.set_zlim(-2, 2)
ax.view_init(azim=-37.5, elev=30)
plt.show()
```

Применяя матрицу масштабирования, мы видим, как куб изменяется: он растягивается вдоль оси x, сжимается вдоль оси y и растягивается вдоль оси z. Изменение размеров куба происходит за счет умножения его координат на соответствующие коэффициенты масштабирования:

$$Sv = \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1.5x + 0y + 0z + 0w \\ 0x + 0.5y + 0z + 0w \\ 0x + 0y + 2z + 0w \\ 0x + 0y + 0z + 1w \end{bmatrix} = \begin{bmatrix} 1.5x \\ 0.5y \\ 2z \\ w \end{bmatrix}$$

Таким образом, куб становится прямоугольным параллелепипедом, если коэффициенты масштабирования различны по осям.

Результаты выполнения масштабирования продемонстрированы на рисунке:

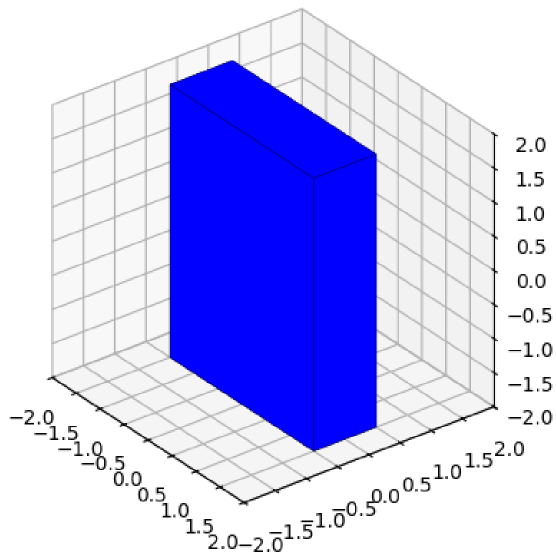


Рис. 1: Масштабированный куб, результат применения матрицы перемещения S

2.3 Перемещение кубика

В данном задании реализуется матрица перемещения T. Элементы матрицы умножаясь и складываясь поэлементно, образуют сумму $\alpha x+\beta y+\gamma z$ с некоторыми коэффициентами перед базисными векторами. Эта сумма складывается из координат исходного вектора с некоторыми коэффициентом — у нас нет независимого элемента, который мог бы прибавиться ко всем координатам, не домножив их на коэффициенты. Учитывая, что получить в произведении сумму можно только поэлементным сложением, то нам не остаётся ничего, кроме как добавить w-компоненту в векторы — вот откуда она появилась и вот зачем она нужна.

$$Tv = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1x + 0y + 0z + 2w \\ 0x + 1y + 0zw \\ 0x + 0y + 1z - 1.5w \\ 0x + 0y + 0z + 1w \end{bmatrix} = \begin{bmatrix} x + 2w \\ y + w \\ z - 1.5w \\ w \end{bmatrix}$$

Код программы, которая смещает объект на заданные значения по каждой из осей x, y и z. Перемещение используется для изменения положения объекта в пространстве без его масштабирования или искажения формы.

```python
def translation_matrix(tx, ty, tz):
    return np.array([
        [1, 0, 0, tx],
        [0, 1, 0, ty],
        [0, 0, 1, tz],
        [0, 0, 0, 1]
    ])

tx, ty, tz = 2.0, 1.0, -1.5
T = translation_matrix(tx, ty, tz)

translated_vertices_cube = apply_transformation(vertices_cube, T)

fig = plt.figure()
ax = fig.add_subplot(projection='3d', proj_type='ortho')
draw_shape(translated_vertices_cube, faces_cube, 'purple')

ax.set_box_aspect([1, 1, 1])
ax.set_xlim(-3, 3); ax.set_ylim(-3, 3); ax.set_zlim(-3, 3)
ax.view_init(azim=-37.5, elev=30)
plt.show()
```

При применении матрицы перемещения T к вершинам куба объект сдвигается на заданные расстояния вдоль каждой оси. В результате куб остаётся неизменным по форме и размерам, но его центр перемещается в новую точку, которая задается параметрами t_x, t_y и t_z. Результат выполнения показан на рисунке:

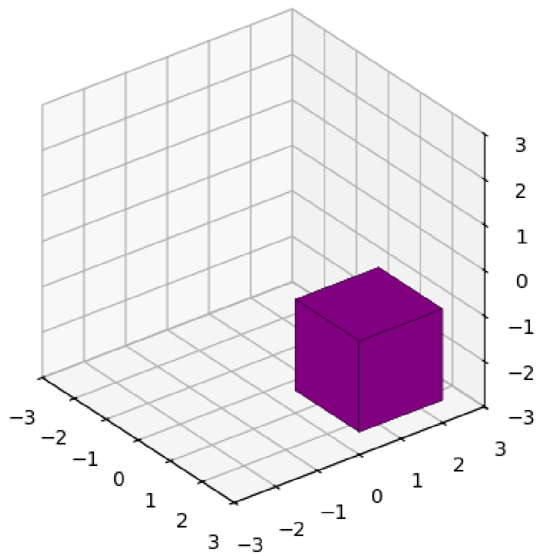


Рис. 2: Смещенный куб, результат применения матрицы перемещения T

А что насчет комбинации матриц маштабирования и перемещения? Комбинируя перемещение и масштабирование, можно исследовать различия между преобразованиями TS и ST, где:

- TS — сначала применяется масштабирование, затем перемещение,
- ST — сначала перемещение, затем масштабирование.

```
def translation_matrix(tx, ty, tz):
    return np.array([
        [1, 0, 0, tx],
        [0, 1, 0, ty],
        [0, 0, 1, tz],
        [0, 0, 0, 1]
    ])

tx, ty, tz = 2.0, 1.0, -1.5
T = translation_matrix(tx, ty, tz)

scaled_then_translated_vertices = apply_transformation(
    scaled_vertices_cube, T)

translated_then_scaled_vertices = apply_transformation(
    vertices_cube, S @ T)
```

```
fig = plt.figure(figsize=(12, 6))

ax1 = fig.add_subplot(121, projection='3d', proj_type='ortho')
draw_shape(scaled_then_translated_vertices, faces_cube, 'red')
ax1.set_title("                                          ,
                                            (ST)")
ax1.set_xlim(-3, 3); ax1.set_ylim(-3, 3); ax1.set_zlim(-3, 3)
ax1.view_init(azim=-37.5, elev=30)

ax2 = fig.add_subplot(122, projection='3d', proj_type='ortho')
draw_shape(translated_then_scaled_vertices, faces_cube, 'blue')
ax2.set_title("                                                          ,
                            (TS)")
ax2.set_xlim(-3, 3); ax2.set_ylim(-3, 3); ax2.set_zlim(-3, 3)
ax2.view_init(azim=-37.5, elev=30)

plt.show()
```

В результате выполнения комбинаций TS и ST наблюдаются различные эффекты:

- При применении ST сначала осуществляется перемещение, после чего куб масштабируется относительно нового положения.
- В случае TS, наоборот, сначала куб масштабируется, а затем перемещается, сохраняя пропорции, заложенные при масштабировании.

Таким образом, очередность применения матриц может существенно влиять на итоговое положение и размеры объекта в пространстве, что иллюстрирует важность порядка операций при работе с матрицами преобразований. Результаты выполнения показаны на рисунке:

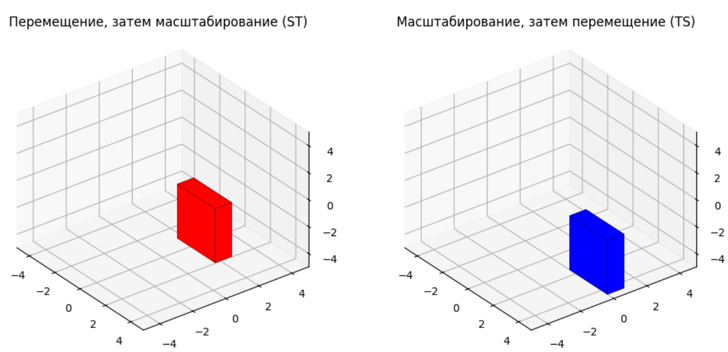


Рис. 3: Результат выполнения комбинаций ST и TS

2.4 Вращение кубика

Для выполнения четвертого задания используется матрица вращения, которая позволяет повернуть куб вокруг заданного вектора v на угол θ. Вращение в 3D-пространстве задается с использованием матрицы поворота $R_v(\theta)$, которая может быть получена с использованием оси и угла вращения.

Вектор $v = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}$ определяет направление оси вращения. При этом матрица поворота $R_v(\theta)$ может быть получена как экспонента от матрицы J, умноженной на угол θ.

Для вектора $v = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}$ матрица J определяется как:

$$J = \frac{1}{\|v\|} \begin{pmatrix} 0 & -v_z & v_y & 0 \\ v_z & 0 & -v_x & 0 \\ -v_y & v_x & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Кроме всем известных матриц поворота вокруг оси для двухмерного пространства существуют также полноценные её аналоги для трёхмерного пространство, которая и представлена выше в матричной форме. J- представление векторного произведения нормированного вектора v в виде линейного преобразования. $R_v(\theta) = I + sin(\theta)J + (1 - cos(\theta))J^2 = e^{J\theta}$ Вместо произвольного вектора подставим базисные вектора декартовой системы координат. Тогда вращение вокруг осей примет вид:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\phi) = \begin{pmatrix} \cos(\phi) & 0 & \sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\psi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Также мы можем утверждать, что тройки матриц $R_x(\theta)$, $R_y(\phi)$ и $R_z(\psi)$ достаточно для описания всех возможных вращений в 3D пространстве. Это утверждение следует из теоремы Эйлера о вращениях, которая гласит, что любое вращение в трёхмерном пространстве можно представить как последовательность трёх вращений вокруг различных осей. Эта последовательность вращений часто называется "углами Эйлера". Таким образом, с помощью комбинации матриц R_x, R_y и R_z можно задать любое возможное вращение в 3D пространстве.

Следующий код иллюстрирует результат воздействия матрицы поворота на куб:

```python
def rotation_matrix(vx, vy, vz, theta):
    v = np.array([vx, vy, vz])
    v = v / np.linalg.norm(v)
    vx, vy, vz = v

    J = np.array([
        [0, -vz, vy, 0],
        [vz, 0, -vx, 0],
        [-vy, vx, 0, 0],
        [0, 0, 0, 0]
    ])

    R = np.eye(4) + np.sin(theta) * J + (1 - np.cos(theta)) * (J @
    J)
    return R

vx, vy, vz = 1, 1, 0
theta = np.pi / 4

R = rotation_matrix(vx, vy, vz, theta)
rotated_vertices_cube = apply_transformation(vertices_cube, R)

fig = plt.figure()
ax = fig.add_subplot(projection='3d', proj_type='ortho')
draw_shape(rotated_vertices_cube, faces_cube, 'orange')

scale_factor = 3
origin = np.array([[0, 0, 0], [scale_factor * vx, scale_factor * vy
    , scale_factor * vz]])
ax.quiver(*origin[0], *origin[1], color='red', arrow_length_ratio
    =0.2, linewidth=2)

ax.set_box_aspect([1, 1, 1])
ax.set_xlim(-2, 2); ax.set_ylim(-2, 2); ax.set_zlim(-2, 2)
ax.view_init(azim=-37.5, elev=30)
plt.show()
```

Применяя матрицу поворота $R_v(\theta)$, мы наблюдаем, как куб изменяет ориентацию, вращаясь вокруг оси, заданной вектором v. В результате вращения его форма и размеры остаются неизменными, но его положение в пространстве поворачивается относительно исходной оси. Это иллюстрирует, как изменение угла θ и направления оси влияет на ориентацию куба.

Результат выполнения поворота показан на рисунке:

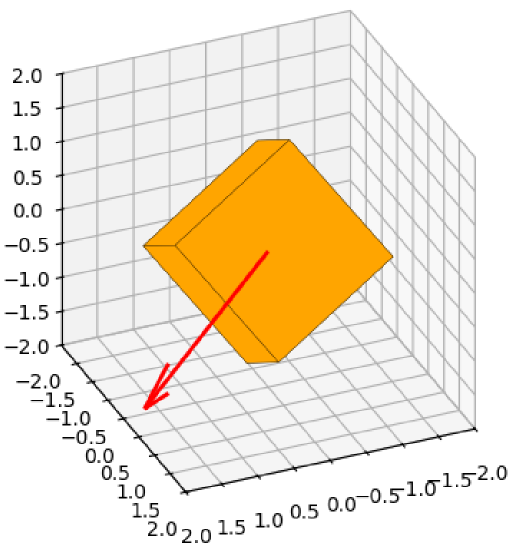


Рис. 4: Вращенный куб, результат применения матрицы вращения $R_v(\theta)$

А можно ли найти ось вращения, имея только матрицу поворота? Да, можно. Для этого необходимо найти собственые вектора матрицы поворота. Но стоит учесть, что собственные векторов у данной матрицы может быть больше чем 1. В качестве оси вращения необходимо взять собственный вектор с действительными числами, соответсвующий направлению оси поворота. Вот к примеру собственный вектор-ось вращения, найденная численно, для примера сверху:

$$v = \begin{pmatrix} 0.7071 \\ 0.7071 \\ 1.22 * 10^{-16} \\ 0 \end{pmatrix} \approx \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

2.5 Вращение кубика около одной вершины

При применении линейного отображения на пространство векторов единственный вектор, который точно не поменяет своего положения, это нулевой вектор. Значит необходимо лишь перенести вершину в центр координат и поворачивать кубик уже там. Для этого воспользуемся матрицами из задания 2 из задания 4(матрицу поворота по оси z). К примеру сместим кубик на 1 по каждой координате и повернём его по оси z на 45 градусов.

$$M = RT = R_z(\frac{\pi}{4})T = \begin{pmatrix} \cos(\frac{\pi}{4}) & -\sin(\frac{\pi}{4}) & 0 & 0 \\ \sin(\frac{\pi}{4}) & \cos(\frac{\pi}{4}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

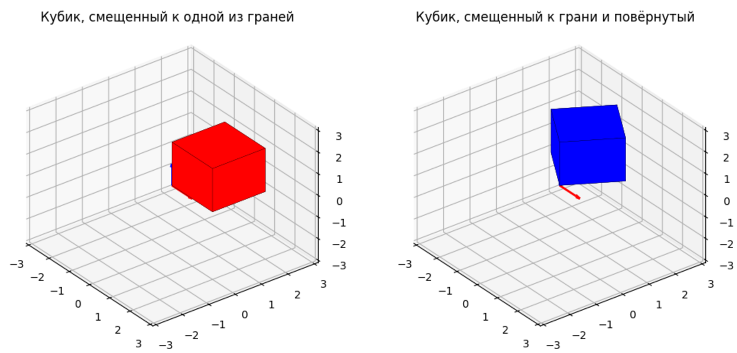


Рис. 5: Результат применения матрицы M

2.6 Реализация камеры

Для начала необходимо выбрать матрицу преобразования, которая "условно"перенесёт камеру из одного пложения в другое. Из условия, что камера смотрит на плоскость снизу можно выбрать такую матрицу для её переноса:

$$TR_y(\frac{4\pi}{3}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\frac{4\pi}{3}) & 0 & \sin(\frac{4\pi}{3}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\frac{4\pi}{3}) & 0 & \cos(\frac{4\pi}{3}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Получается, что камера сначала провернётся и будет смотреть на кубики сверху под углом 45 градусов, а затем сместится по y на 4.

Факт в том, что мы не можем применить данную трансформацию на нашу камеру, вместо этого мы можем применить обратную от нашей матрицы на объекты сцены и таким образом разложить объекты сцены,чтобы смоделировать перемещение камеры с помощью нашей матрицы, не перемещая её

```
T2 = translation_matrix(6,-2, 0)
T3 =translation_matrix(-2,4,0 )
cube1 = vertices_cube.copy()
cube2 = T2@vertices_cube.copy()
cube3 = T3@vertices_cube.copy()
##
```

```
fig = plt.figure(figsize=(12, 6))
ax1 = fig.add_subplot(121, projection='3d', proj_type='ortho')
draw_shape(ax1, cube1, faces_cube, 'blue')
draw_shape(ax1, cube2, faces_cube, 'red')
draw_shape(ax1, cube3, faces_cube, 'green')
ax1.set_title("
                                                                ")
ax1.set_xlim(-6, 6); ax1.set_ylim(-6, 6); ax1.set_zlim(-6, 6)
ax1.view_init(azim=0, elev=-90)
drowAxis(ax1)

#

CameraMatrix = translation_matrix(0,4,0)@rotation_matrix(0,1,0,4*np
     .pi/3)
inverseCameraMatrix = np.linalg.inv(CameraMatrix)

ax2 = fig.add_subplot(122, projection='3d', proj_type='ortho')
draw_shape(ax2, inverseCameraMatrix@cube1, faces_cube, 'blue')
draw_shape(ax2, inverseCameraMatrix@cube2, faces_cube, 'red')
draw_shape(ax2, inverseCameraMatrix@cube3, faces_cube, 'green')
ax2.set_title("
                                                                ")
ax2.set_xlim(-6, 6); ax2.set_ylim(-6, 6); ax2.set_zlim(-6, 6)
ax2.view_init(azim=0, elev=-90)
drowAxis(ax2)
plt.show()
```

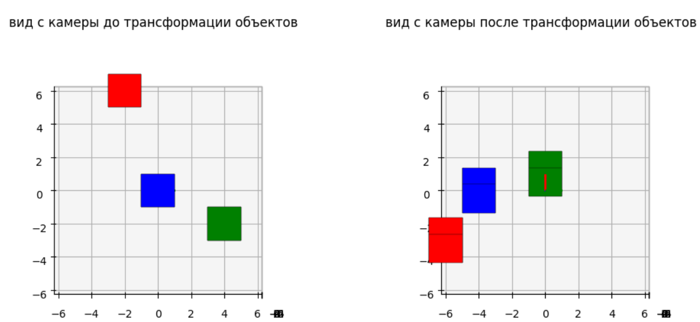


Рис. 6: Результат применения матрицы $(TR_y(\frac{4\pi}{3}))^{-1}$на объекты сцены

На картинке хорошо видно, как у нас получилось "переместить"камеру от центра в синем кубе на центр в зелёном кубе

2.7 Перспектива

Рассмотрим матрицу P

$$P = \begin{pmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & -\frac{f}{(f-n)} & -\frac{fn}{(f-n)} \\ 0 & 0 & -1 & 0 \end{pmatrix} S = \frac{1}{\frac{fov}{2} * \frac{\pi}{180}}$$

Где fov - угол обзора, f - расположение дальней обрезающей плоскости, а n - расположение ближайшей.

Каким образом работает данная матрица? Для начала стоит определить, что конкретно она записывает. Главная её идея заключается в том, чтобы спроецировать точки декартовых координат на плоскость изображения (в нашем примере плоскостью изображения будет плоскость Oxy). Значения координат проекции в общем случае определяется таким образом: $x' = \frac{x}{w'}, y' = \frac{y}{w'}, z' = \frac{-z}{w'} = 1, w' = -z$ Значения S - коэффициент, который зависит от угла обзора. А коэффициенты $-\frac{f}{(f-n)}$ и $-\frac{fn}{(f-n)}$ служать, чтобы нормализовать значения z от 0 до 1 в соответсвии с ближней и дальней обрезающими плоскостями.

Теперь перейдём к построению преспективы с помощь python. Для начала зафиксируем камеру в одном положении (elev = 90). Применение матрицы P отобразит объекты сцены с учётом перспективы на Oxy, мы же в свою очередь спроецируем это изображения на нашу камеру. Из-за того, что проекцию мы принимаем на камеру, нам нельзя менять её пложения. Вместо этого мы перенесём все объекты сцены в соответсвии с видом, который хотим получить (воспользуемся методом из предыдущего задания). Для корректного отображения необходимо учитывать, что все объекты сцены должны находится по одну сторону от камеры. Не стоит забывать, что в отличии от мировых координат, ось координат z камеры направлена против оси z мировых координат. Если смотреть на объекты с камеры сверху вниз, то это не влияет на изображение. Если же смотреть снизу вверх, то проекция отразится по линии (1,1,0).

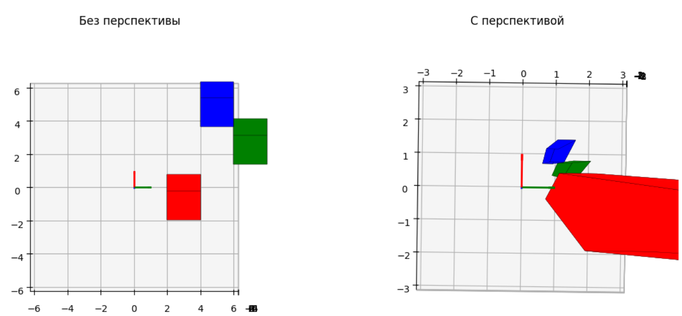
Данный код показывает работу матрицы преспективы для 1-ого кубика

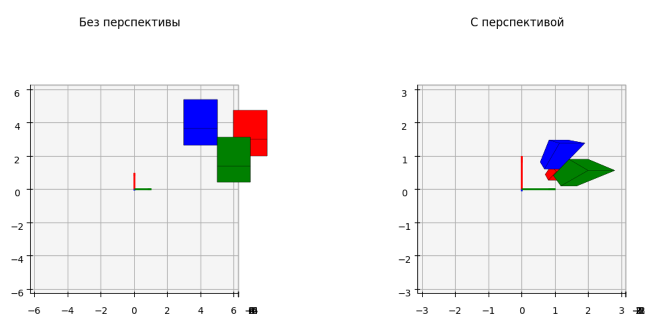
```
cube = vertices_cube.copy()
#
#
#
CameraMatrix= translation_matrix(2,2,2)@rotation_matrix(0,1,0,np.pi
    /2)
inverse_camera_matrix = np.linalg.inv(CameraMatrix)
f= 100
n= 0.1
S = np.sqrt(2)/2 #90
perspective = np.transpose(np.array([[S  ,0,0,0],
                   [0,S  ,0,0],
                   [0,0,-(f)/(f-n),-1],
                   [0,0,-f*n/(f-n),0],]))

```

```
camera_cube = inverse_camera_matrix@cube

fig = plt.figure(figsize=(12, 6))
ax1 = fig.add_subplot(121, projection='3d', proj_type='ortho')
draw_shape(ax1, camera_cube, faces_cube, 'blue')
ax1.set_xlim(-3, 3); ax1.set_ylim(-3, 3); ax1.set_zlim(-3, 3)
ax1.view_init(azim=0, elev=90)
drowAxis(ax1)
#
#
#
ax2 = fig.add_subplot(122, projection='3d', proj_type='ortho')
draw_shape(ax2, perspective@camera_cube,faces_cube, 'blue')
ax2.set_xlim(-3, 3); ax2.set_ylim(-3, 3); ax2.set_zlim(-3, 3)
ax2.view_init(azim=0, elev=90)
drowAxis(ax2)
#
#
#
plt.show()
```





3 Заключение

В ходе данной работы мы применили объекты линейной алгербры для различных преобразований 3D пространтсва. Выяснили, каким образом в принципе описываются преобразования пространства, наподобии поворота или маштабирования. Смогли описать изменеие положения камеры без его фактического изменения. Выяснили метод работы матрицы перспективы.