Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«ПОБУДОВА В HASKELL ПРАВИЛЬНИХ БАГАТОГРАННИКІВ (НА БАЗІ OPEN GL)»**

Виконав: студент 4-го року навчання
освітньої програми «Прикладна математика»,
спеціальності 113 Прикладна математика

Бікчентаєв Микола Олексійович

Керівник: Проценко В. С.,
кандидат фіз.-мат. наук, доц.

Рецензент_____
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК _____
«____» _____ 2021 р.

Київ – 2021

ЗМІСТ

ВСТУП

Коли мова йде про роботу з комп'ютерною графікою, часто згадують такі графічні рушії як *Unity, Unreal Engine* і т. д.

Ці рушії надають безліч готових інструментів для роботи з графікою, звуком чи фізикою графічних об'єктів. Часто ці інструменти використовують[1-2] ту чи іншу реалізацію OpenGL – специфікації програмного інтерфейсу, що містить визначення функцій для роботи з графікою.

Знати OpenGL досить корисно, оскільки це дає можливість зрозуміти принципи роботи сучасних графічних рушіїв. З'являється гнучкість при роботі з їх інструментами: ми можемо змінити існуючі, або ж розробити власні. Використовуючи OpenGL, ви також можете розробити власний графічний рушій, якщо існуючі не відповідають вашим вимогам.

OpenGL реалізований на багатьох мовах програмування[3], зокрема на Java, Python, C++ та Haskell.

Великою перевагою Haskell є її лаконічність та простота. При роботі з OpenGL досить багато часу займає робота з пам'яттю. У Haskell ці деталі сховані від користувача, що дає можливість зосередитись на написанні коду, який відповідає безпосередньо за роботу з графікою.

Для демонстрації можливостей реалізації OpenGL на мові Haskell гарно підходять багатогранники. Багатогранники мають велику різноманітність форм та видів, проте складаються вони завжди з багатокутників[11, с.13]. Зображуючи багатокутники, ми можемо побудувати досить складні багатогранники. Це відповідає одному з головних принципів OpenGL – візуалізації складних об'єктів через велику кількість простіших.

Отже, **метою даної роботи** є розгляд однорідних багатогранників (у множину яких входять правильні багатогранники) та їх візуалізація із застосуванням реалізації OpenGL на мові Haskell, а також опис можливостей OpenGL й створення графічного застосунку для перегляду тривимірних моделей однорідних багатогранників.

Об'єкт дослідження: однорідні багатогранники, специфікація програмного інтерфейсу OpenGL.

Методи дослідження: аналіз наукової літератури.

Завдання дослідження:

1. Дослідити однорідні багатогранники та розглянути способи їх побудови.

2. Розглянути специфікацію програмного інтерфейсу OpenGL та її реалізацію на мові Haskell.

3. Використовуючи OpenGL, розробити графічний застосунок для перегляду тривимірних моделей однорідних багатогранників.

Робота складається зі вступу, чотирьох розділів, висновків та списку використаних джерел.

У *першому розділі* розглядається специфікація програмного інтерфейсу OpenGL, мова програмування Haskell та взаємодія між Haskell та OpenGL. Наводиться визначення декількох типів, що будуть використані при візуалізації багатогранників.

У *другому розділі* наводиться визначення однорідних багатогранників та розглядаються дві групи цих багатогранників: Платонові та Архімедові тіла. Описуються принципи побудови Платонових та Архімедових тіл.

Третій розділ присвячено побудові багатогранників із використанням реалізації OpenGL на мові Haskell.

У *четвертому розділі* розглядаються зірчаті однорідні багатогранники, а також об’єднання однорідних багатогранників.

Наукова новизна одержаних результатів: у роботі наведено спосіб візуалізації однорідних багатогранників із використанням реалізації OpenGL на мові Haskell. Описано основні види однорідних багатогранників, їх влаштування та принципи побудови.

Практичне значення одержаних результатів: робота буде корисною тим, хто цікавиться комп’ютерною графікою, зокрема OpenGL та його можливими застосуваннями, або однорідними багатогранниками та їх побудовою.

1. Haskell та OpenGL. Підготовка до візуалізації графічних об’єктів

1.1. Програмний інтерфейс OpenGL

OpenGL – це програмний інтерфейс (англ. *API*), який використовується для створення комп’ютерної графіки та містить визначення 250 команд для роботи з нею[4, с. 23].

OpenGL містить тільки визначення цих операцій. Їх реалізація лягає на плечі розробників, якими зазвичай є виробники графічних карт. Тобто, будь-яка придбана вами відеокарта буде підтримувати певну реалізацію OpenGL, розроблену спеціально для неї або для її серії. Тому, OpenGL – це *специфікація* програмного інтерфейсу[5].

OpenGL є *апаратно-* та *програмно-незалежним* інтерфейсом. Він не містить визначень команд для роботи з вікнами чи користувацьким вводом, оскільки ці операції можуть по-різному працювати на різних пристроях. Проте, існує безліч бібліотек, які можуть допомогти впоратись з цими задачами[4, с. 23].

Візуалізація (англ. *rendering*) складних тривимірних об’єктів, таких як моделі техніки чи предметів, у OpenGL відбувається за допомогою *графічних примітивів*: точок, ліній та багатокутників. Сам OpenGL не містить команд, які б давали можливість описати та відразу отримати складні тривимірні об’єкти [4, с. 23].

1.2. Бібліотеки для OpenGL

Для OpenGL існує безліч бібліотек, які можуть допомогти зі створенням вікон, де ми будемо візуалізувати графічні об’єкти, чи зі зчитуванням користувацького вводу. Деякі з найпопулярніших бібліотек це *SDL*, *SFML*, *GLFW* та *GLUT*[4, с. 33-34]. Ми будемо використовувати останню.

GLUT (*OpenGL Utility Toolkit*) – це незалежний від операційної системи набір інструментів, який використовується для створення вікон, отримання вводу з клавіатури чи миші, візуалізації деяких геометричних примітивів (куб, сфера та інші) та створення простих контекстних меню.

1.3. OpenGL – скінчений автомат

OpenGL можна уявити у вигляді скінченого автомату. Ми переводимо його у різні стани, змінюючи *контекст* OpenGL – набір змінних, значення яких визначають поведінку програмного інтерфейсу, або ж те, як саме OpenGL зображує графічні об'єкти[4, с. 29].

Наприклад, якщо замість зафарбованого куба ми забажаємо намалювати лише його каркас, то нам буде достатньо змінити лише одну зі змінних контексту. При цьому, ця зміна вплине не тільки на цей куб, а і на усі інші об'єкти, що будуть зображені після нього.

1.4. Мова програмування Haskell

Haskell – це назва *чистої функціональної мови програмування*. Різниця між імперативними мовами програмування (Java, Python, C++ і т. д.) та функціональними полягає у тому, що програма, написана імперативною мовою, представляє собою послідовність інструкцій, після виконання яких комп'ютером, ми отримаємо результат. Під час виконання цих інструкцій стан програми може бути змінений. Наприклад, ми можемо присвоїти змінній `counter` певне значення, та потім змінити його на якесь інше[6].

У функціональній мові ми не використовуємо послідовності інструкцій, а описуємо правила за якими можна отримати бажаний результат [6]. Наприклад, сума чисел у списку – це перше число плюс сума чисел, що залишилися. Кожне таке правило є функцією. Поєднуючи правила, ми поєднуємо функції.

Окрім цього, чисті функціональні мови програмування мають певні, важливі для нас, властивості[7, с. 16-21]:

a) Усі функції є *чистими*;

b) Змінна, після її ініціалізації, не може бути змінена.

Чисті функції для одного й того ж набору даних повертають однакові значення (що називається *детермінованістю*) та не мають побічних ефектів[7, с. 16]. Чиста функція не може змінювати змінні поза її областю видимості, виконувати вивід у консоль або читати файли. Вона може тільки отримати дані та повернути результат обрахунків. Звідси, функції у Haskell повинні мати хоча б один параметр та обов'язково щось повертати.

Зі змінними все простіше: ми можемо створювати нові змінні, проте не можемо змінювати вже існуючі[7, с. 18].

1.5. Взаємодія між Haskell та OpenGL

Після ознайомлення з двома попередніми параграфами може виникнути питання щодо того, чи можуть Haskell та OpenGL взагалі бути сумісними? Адже принцип роботи OpenGL зовсім не відповідає концепціям, яких притримується Haskell.

На щастя, Haskell має *клас типів* під назвою `Monad` (*монада*)[7, с. 381]. Клас типів визначає операції (функції) які можна застосувати до типів даних, що належать цьому класу[7, с. 134]. Наприклад, `Num` – це клас типів, члени якого повинні реалізовувати операцію «+». Членом цього класу є, наприклад, тип `Int`.

Типи даних класу `Monad` у Haskell дозволяють функціям виконувати I/O операції, змінювати значення змінних, що знаходяться поза областю видимості функції, і так далі[8, с. 6]. Окрім цього, завдяки монадам, з'явилась можливість реалізації OpenGL на мові Haskell.

1.6. Визначення необхідних типів. Імпортування необхідних модулів

У цій роботі ми будемо використовувати модуль під назвою `Graphics.UI.GLUT`, який є реалізацією бібліотеки GLUT на мові Haskell. Окрім функцій для роботи з вікнами та користувацьким вводом, він також містить всі функції програмного інтерфейсу OpenGL. Зокрема ті, що можуть змінити положення *камери* на *сцені*.

Сценою ми будемо називати те, що користувач бачить у вікні нашої програми. Сцену користувач бачить через *камеру*. Змінюючи положення камери, ми можемо переглядати сцену з різних ракурсів.

Ми також будемо використовувати модуль `Data.IORef`, що дозволяє створювати змінні, значення яких можна змінити під час виконання програми, використовуючи спеціальні функції[6].

Створимо модуль `Main`, у файлі `Main.hs`, та імпортуємо у нього модулі, згадані вище.

```
import Data.IORef
import Graphics.UI.GLUT
```

Лістинг 1.1 Імпортування необхідних модулів

Визначимо, також, декілька нових типів даних.

Перший типу має назву `State` та виглядає наступним чином:

```
data State = State
  { polyhedraId :: IORef Int,
    cameraPos :: IORef (Int, Int, Double)
  }
```

Лістинг 1.2 Тип даних State

Конструктор типу `State` має два параметри:

a) `polyhedraId` – номер багатогранника який відображається на сцені у даний момент;

b) `cameraPos` – позиція камери у *сферичних координатах*.

Після «`::`» зазначений тип кожного параметру[7, с. 129].

Два останніх типи мають назви `MyPoint` та `PolyFace`. Їх визначення буде наведене у модулі `RenderHelper` з файлу `RenderHelper.hs`.

```
data MyPoint = MyPoint
  { xC :: GLfloat,
    yC :: GLfloat,
    zC :: GLfloat
  }
```

```
data PolyFace = PolyFace
  { polyFaceIndices :: [Int],
    polyFaceColor :: Color3 GLfloat
  }
```

Лістинг 1.3 Типи даних MyPoint та PolyFace

Тип `MyPoint` визначає точку у тривимірному просторі, а тип `PolyFace` – грань багатогранника.

Параметри конструктора `MyPoint` мають тип `GLfloat`, визначення якого міститься у модулі `Graphics.UI.GLUT`. Цей тип є аналогічним звичайному типу `Float`[4, с. 28].

Конструктор `PolyFace` приймає два значення: масив індексів, що відповідають координатам вершин грані, та колір грані, що має тип `Color3 GLfloat`. Визначення типу `Color3` міститься у модулі `Graphics.UI.GLUT`. Значення цього типу задають колір у форматі *RGBA*[4, с. 148-150].

Використовуючи цей формат, ми представляємо колір у вигляді послідовності з чотирьох чисел. Перші три числа відповідають за інтенсивність червоного (R), зеленого (G) та синього (B) кольорів та набувають значень від 1 до 225. Четверте відповідає за прозорість кольору (A) та приймає значення від 0 до 1.

Особливістю типу `Color3` є те, що компонента «A» кольору, по замовчуванню, рівна 1[4, с. 155]. Таким чином, нам потрібно вказати лише три числа.

2. Випуклі однорідні багатогранники

2.1. Означення однорідних багатогранників

Фігуру на площині ми можемо уявити як множину відрізків, що обмежують частину цієї площини. Таку фігуру ми будемо називати *багатокутником*[10, с. 1].

Багатогранник – це множина багатокутників, що обмежують частину тривимірного простору[10, с. 1].

Багатогранники можуть досить сильно відрізнятися один від одного, проте певні схожі риси вони мають. Зокрема, кожен багатогранник складається з трьох компонент[11, с. 13]:

a) Багатокутники, які формують багатогранник, називаються *гранями*;

b) Відрізок прямої, спільний для двох граней, називається *ребром*;

c) Точка, де сходяться декілька граней та ребер, називається *вершиною*.

Багатокутник називається *правильним*, якщо всі його кути та сторони рівні[7, 1].

Багатогранник називається *правильним*[12, с. 265], якщо:

a) Він *випуклий*, тобто кожен його *двогранний кут* (кут сформований двома сусідніми гранями) не більший за 180°[10, с. 3];

b) Всі його грані – це правильні багатокутники одного типу (тільки трикутники, тільки чотирикутники і т. д.);

c) У кожній його вершині сходиться однакова кількість ребер;

d) Усі його двогранні кути рівні.

Правильних багатогранників існує всього п’ять і називають їх *Платоновими тілами*[10, с. 2-3].

Існує також множина з тринадцяти багатогранників, що називаються *напівправильними* (бо їх гранями є правильні багатокутники декількох типів) або *Архімедовими тілами*[10, с. 2-3].

Об’єднання Платонових та Архімедових тіл називають *випуклими однорідними багатогранниками*[10, с. 3].

Багатогранник називають *однорідним* якщо його грані – це правильні багатокутники, і всі *багатогранні кути*[12, с. 232] рівні.

Кожну вершину однорідного багатогранника оточують багатокутники у одному й тому ж порядку[10, с. 3]. Наприклад, для *кубооктаедра* порядок граней такий: квадрат, трикутник, квадрат, знову трикутник.

2.2. Платонові тіла

2.2.1. Загальні відомості

Доведемо, що Платонових тіл існує всього п’ять.

Теорема 2.1. Платонових тіл існує тільки п’ять, це: *тетраедр*, *октаедр*, *гексаедр* (куб), *ікосаедр* та *додекаедр*[12, с. 265-266].

Доведення. Нехай число p – це кількість ребер, яку має кожна грань правильного багатогранника, а q – це кількість граней, що сходяться у кожній з вершин цього багатогранника. Таким чином, кожний правильний багатогранник ми можемо позначати парою чисел $\{p, q\}$, що має назву символу Шлефлі (англ. *Schläfli symbol*). Наприклад, тетраедр буде позначатися як $\{3, 3\}$.

Сума внутрішніх кутів кожної грані правильного багатогранника рівна $180(p-2)$[12, с. 53]. Звідси, градусна міра кожного окремого кута грані може бути записана як $\frac{180(p-2)}{p}$.

Оскільки правильні багатогранники є випуклими, то сума пласких кутів кожного багатогранного кута менша $360°$[12, с. 234]. Отримаємо наступну нерівність:

$$\left(180 - \frac{360}{p}\right) q < 360$$
$$180\left(1 - \frac{2}{p}\right) < 360$$
$$(p-2)(q-2) < 4$$

Можна побачити, що p та q повинні бути більші двох. Тоді, якщо $p = 3$, то q може дорівнювати 3, 4 або 5. Якщо $p = 4$, то $q = 3$. Коли $p = 5$, то $q = 3$. При цьому, p не може бути більше 5.

Таким чином, ми отримали п’ять пар: $\{3, 3\}$, $\{3, 4\}$, $\{3, 5\}$, $\{4, 3\}$ та $\{5, 3\}$. Кожна пара чисел відповідає одному Платоновому тілу: *тетраедру*, *октаедру*, *ікосаедру*, *гексаедру* (кубу) та *додекаедру* відповідно.

Теорему доведено.

2.2.2. Тетраедр. Побудова тетраедру

Тетраедр – це багатогранник, що складається з 4 вершин та 6 ребер. Його гранями є чотири трикутники. Якщо ці трикутники рівносторонні, то тетраедр буде називатися *правильним* (рис. 2.1). Саме *правильний тетраедр* є Платоновим тілом[12, с. 265].

У подальшому, правильний тетраедр ми будемо називати просто тетраедром.

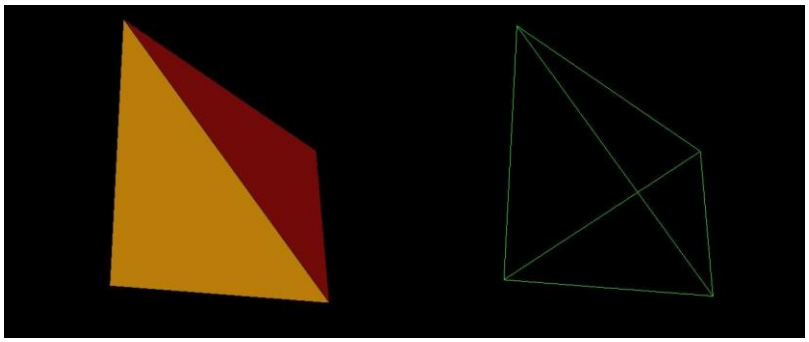


Рис. 2.1 Правильний тетраедр

Алгоритм побудови тетраедра був наведений у відомому трактаті Евкліда під назвою «*Начала*». Зокрема, на ньому базується спосіб побудови тетраедра та інших Платонових тіл, які будуть наведені нижче[11, с. 66].

Візьмемо півколо з діаметром NS. Поставимо на ньому точку Q так, щоб $NQ{:}\,NS = 2{:}\,1$. Висота тетраедру буде рівна NQ.

Проведемо перпендикуляр до NS, що проходить через точку Q, і нехай P – це точка перетину півкола та перпендикуляру. З'єднаємо цю точку з точкою N та отримаємо відрізок NP (рис. 2.2), довжина якого буде рівною довжині ребра тетраедру.

Намалюємо коло з центром у точці H, з радіусом рівним довжині відрізка QP та впишемо у це коло рівносторонній трикутник EFG. З'єднаємо центр кола з вершинами цього трикутника щоб отримати відрізки EH, FH та GH.

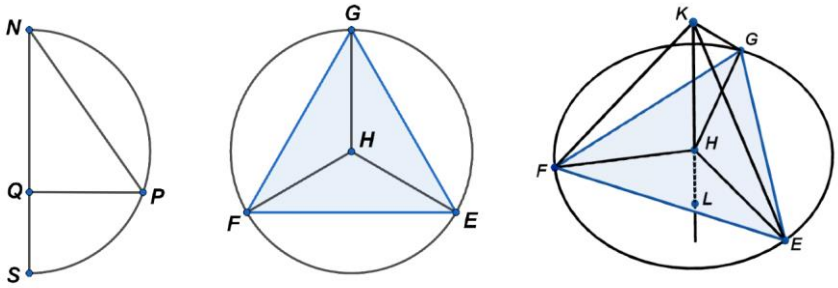


Рис. 2.2 Побудова правильного тетраедру

Через точку H проведемо пряму, перпендикулярну до площини, що містить трикутник EFG. На цій прямій, по протилежні боки від точки H, візьмемо дві точки K та L, так щоб $KH = NQ$ і $HL = SQ$. З'єднаємо точку K з вершинами рівностороннього трикутника EFG та отримаємо відрізки KE, KF та KG, що є ребрами тетраедру.

Відрізки KE, KF, KG та FE, FG, GE дорівнюють NP. Таким чином, трикутники EGF, EGK, GFK та FEK є рівносторонніми.

Побудовано правильний тетраедр.

2.2.3. Октаедр. Побудова октаедру

Октаедр – це багатогранник, гранями якого є вісім рівносторонніх трикутників. Окрім цього, він має 12 ребер та 6 вершин[12, с. 265] (рис. 2.3).

Октаедр також можна уявити у вигляді двох правильних чотирикутних пірамід, що мають одну спільну грань.

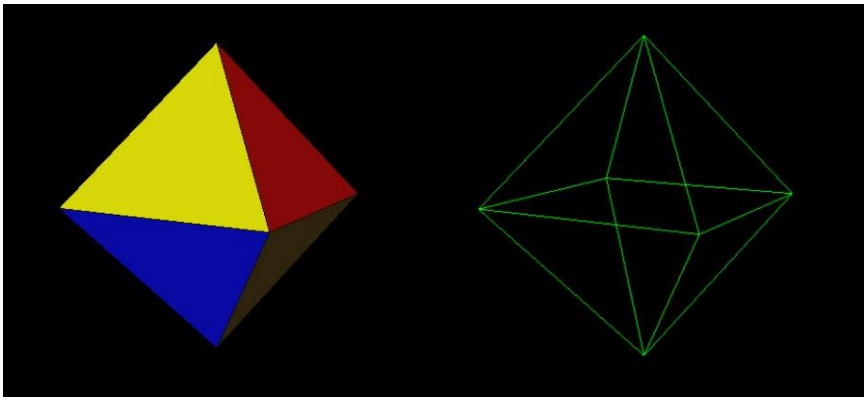


Рис. 2.3 Октаедр

Для побудови октаедра, візьмемо півколо з діаметром NS та точку Q посередині діаметру[11, с. 67-68] (рис. 2.4).

Проведемо через точку Q пряму перпендикулярну відрізку NS і нехай P – це точка перетину цієї прямої та півкола. З'єднаємо точки P та N щоб утворити відрізок NP.

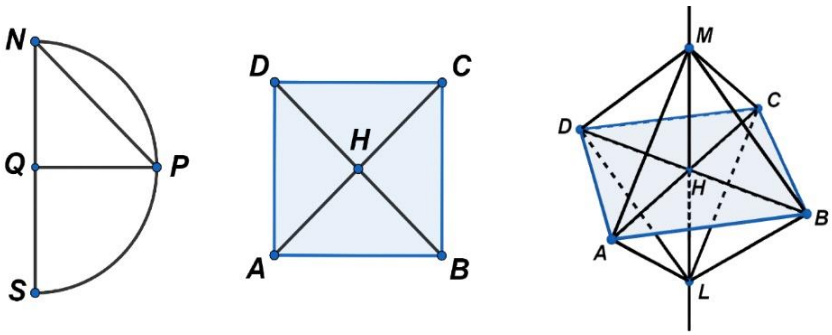


Рис. 2.4 Побудова октаедру

Намалюємо квадрат $ABCD$, довжина сторін якого рівна довжині відрізка NP.

Нехай точка H – це точка перетину діагоналей квадрата. Через цей центр проведемо пряму перпендикулярну до площини, що містить квадрат $ABCD$.

На цій прямій, по обидві сторони від точки H, візьмемо дві точки L та M так, щоб $HL = NQ$ та $HM = NQ$. З'єднавши ці точки з вершинами квадрата, ми отримаємо 12 ребер октаедра.

2.2.4. Куб. Побудова куба

Куб або *гексаедр* – це правильний багатогранник, гранями якого є шість квадратів. Він складається з 12 ребер та 8 вершин[12, с. 265] (рис. 2.5).

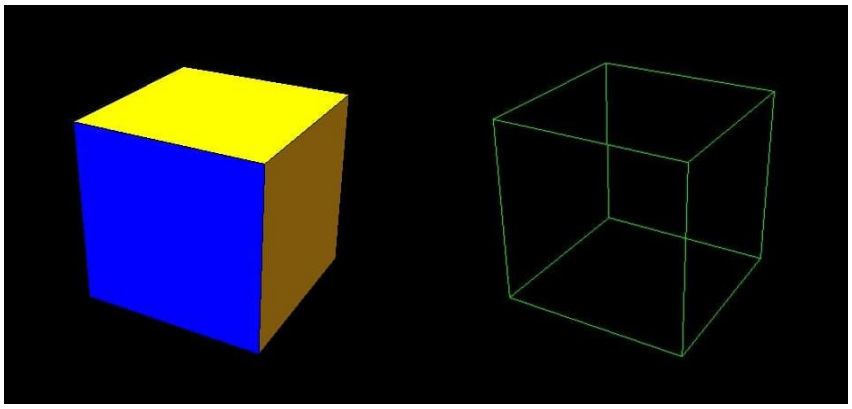


Рис. 2.5 Куб

Візьмемо півколо з діаметром NS[11, с. 68]. Поставимо на ньому точку Q так, щоб $NQ:QS = 2:1$ і проведемо через точку Q пряму перпендикулярну до NS (рис. 2.6). Точка P – місце перетину цієї прямої та півкола. Залишається з'єднати точки Q та P, щоб отримати відрізок SP.

Побудуємо квадрат $ABCD$, довжина грані якого дорівнює довжині SP.

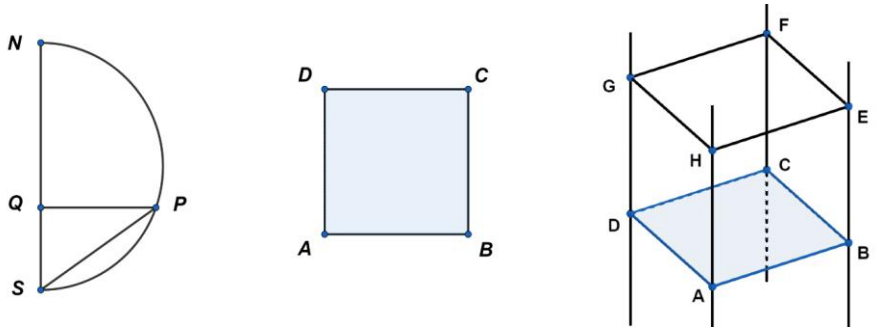


Рис. 2.6 Побудова куба

Через вершину квадрата A проведемо пряму, перпендикулярну площині, що містить квадрат $ABCD$. На цій прямій поставимо точку H, так щоб $AH = SP$. Якщо виконати аналогічні дії для інших вершин квадрата то ми отримаємо ще три точки: E, F та G.

З'єднавши отримані точки між собою, ми отримаємо куб.

2.2.5. Додекаедр. Побудова додекаедру

Додекаедр – це багатогранник гранями якого є дванадцять правильних п’ятикутників. Окрім цього, він складається з 30 ребер та 20 вершин[12, с. 266] (рис. 2.7).

Додекаедр можна отримати якщо на кожній грані куба побудувати фігуру у формі «даху»[11, с. 69-70] (рис. 2.7). Для цього нам потрібно знайти висоту та положення вершин цього даху.

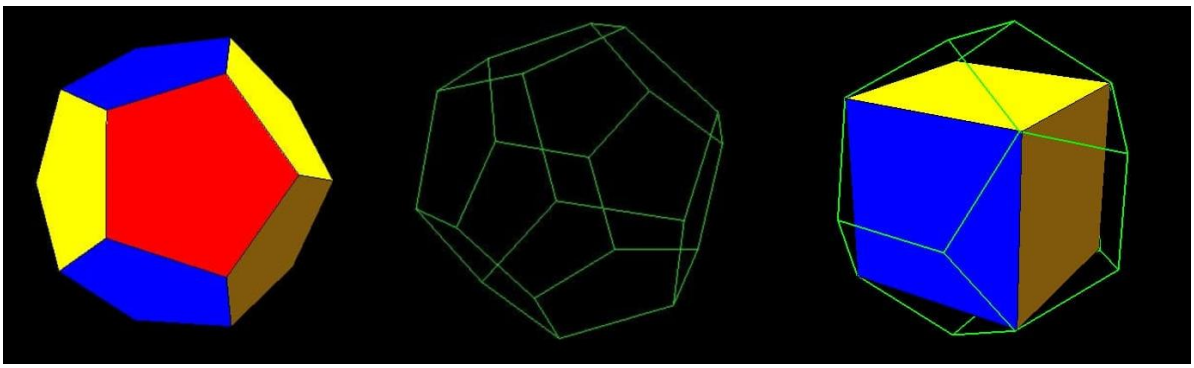


Рис. 2.7 Додекаедр

Намалюємо квадрат з вершинами $ABCD$ і нехай точки E та F – середні точки сторін AD та CB відповідно. З’єднаємо ці точки та отримаємо відрізок EF.

Нехай точка G – середина відрізку EF. Відрізок EG нам потрібно поділити по *золотому перетину*[12, с. 152-153], що дасть нам точку H, при чому відрізок EH коротший за HG (рис. 2.8). Аналогічно ми ділимо відрізок GF та отримуємо точку J, при цьому JF коротший за GJ.

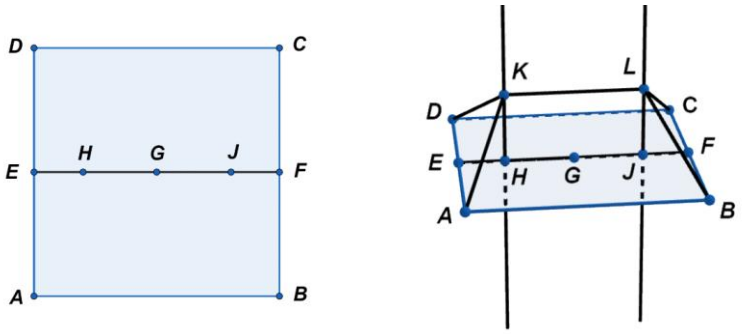


Рис. 2.8 Побудова "даху" для куба

Через точки H та J проведемо прямі перпендикулярні площині, що містить квадрат $ABCD$. На цих прямих позначимо точки K та L, так щоб довжина відрізків HK та JL була рівна довжині HG. З’єднавши точки K і L між собою та з вершинами квадрати, ми отримаємо «дах». Виконавши цю процедуру для усіх граней квадрата, ми отримаємо додекаедр. При цьому важливо звертати увагу на орієнтацію «дахів».

2.2.6. Ікосаедр. Побудова ікосаедру

Ікосаедр – це багатогранник гранями якого є 20 рівносторонніх трикутників. Також, він складається з 12 вершин та 30 ребер[12, с. 265] (рис. 2.9).

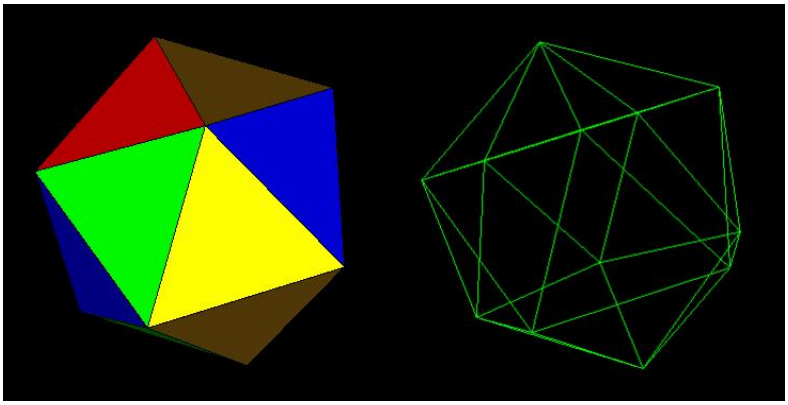


Рис. 2.9 Ікосаедр

Ікосаедр можна побудувати, взявши за основу куб[11, с. 70]. 12 вершин ікосаедра будуть знаходитись на гранях куба, по дві вершини на кожну грань.

Знайти ці вершини можна так само, як і точки H та J з алгоритму побудови додекаедра, що був наведений раніше. Виконавши його для кожної з 6 граней куба, ми отримаємо 12 вершин ікосаедра.

2.3. Архімедові тіла

2.3.1. Загальні відомості

Архімедові тіла – це група з 13 багатогранників[10, с. 2], у яких всі багатогранні кути рівні, а грані є правильними багатокутниками різних типів. На рисунку 2.10 наведені всі Архімедові тіла.

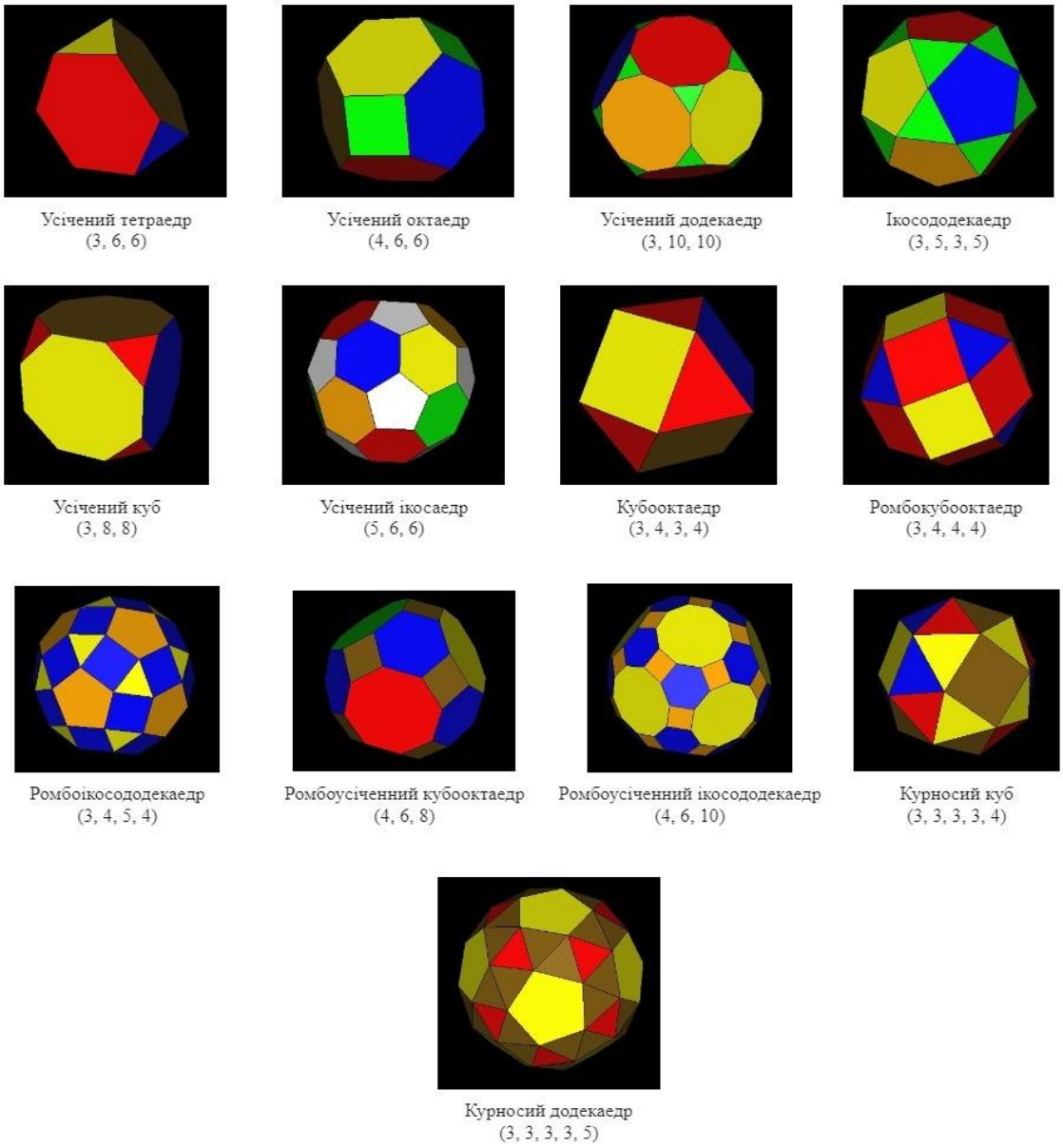


Рис. 2.10 Архімедові тіла

Теорема 2.2 Архімедових тіл існує тільки тринадцять[11, с. 158-168].

Доведення. Архімедові тіла – це випуклі однорідні багатогранники, тому порядок граней навколо кожної вершини Архімедового тіла однаковий.

Кожній вершині Архімедового тіла можна поставити у відповідність одну й ту саму послідовність натуральних чисел $(v_1, v_2, \dots, v_n)$, де v_i – це кількість ребер однієї з граней, якій належить дана вершина. Звідси, кожному Архімедовому тілу можна поставити у відповідність таку послідовність. Наприклад, *усіченому кубу* (рис. 2.10) буде відповідати послідовність $(3, 8, 8)$.

Розглянемо довільний випуклий однорідний багатогранник, що описується послідовністю $(v_1, \dots v_n)$. Кожний такий багатогранник повинен мати принаймні три грані, тому $v_1, \dots v_n \geq 3$. При цьому, у кожній вершині повинно зустрічатися не менше трьох граней, тому $n \geq 3$.

Якщо $n \geq 6$, то сума пласких кутів кожного багатогранного кута буде не менша ніж $6 * 60° = 360°$, що не можливо, оскільки багатогранник випуклий [12, с. 234]. Тому $n < 6$.

Доведення розділимо на три частини: для $n = 3$, $n = 4$ та $n = 5$ відповідно.

Частина I. Нехай, $n = 3$ та випуклий однорідний багатогранник описується послідовністю (a, b, c), де $a \leq b \leq c$.

Якщо $a \geq 6$, то сума пласких кутів кожного багатогранного кута буде не меншою ніж $3 * 120° = 360°$, що не можливо. Таким чином, $a = 3$, $a = 4$ або $a = 5$.

Нехай, $a = 3$. Покажемо, що b-реберна та c-реберна грані однакові. Для цього розглянемо довільну трикутну грань ABC (рис. 2.11).

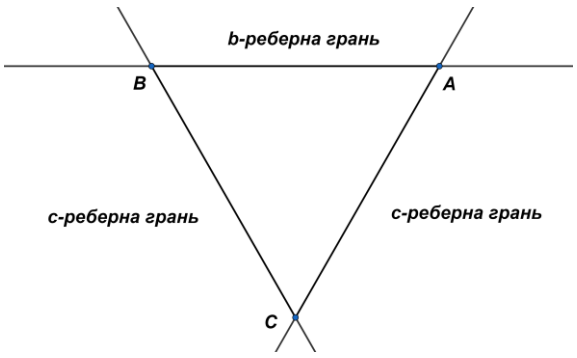


Рис. 2.11 Довільна трикутна грань ABC

Не втрачаючи загальності, нехай ребро AB прилегле до b-реберної грані. Тоді інші два ребра, BC та AC, повинні бути прилеглими до c-реберної грані. Таким чином, вершину C будуть оточувати дві c-реберні грані. Оскільки багатогранник однорідний, $b = c$.

Якщо $b = 3$, то ми отримаємо послідовність $(3, 3, 3)$, що відповідає тетраедру, який не є Архімедовим тілом. Таким чином, $b \geq 4$, при цьому b буде парним числом. Покажемо, що це так.

Розглянемо частину довільної b-реберної грані та позначимо на ній вершини $ABCD$ (рис. 2.12).

Вершину B, окрім b-реберної грані, оточує трикутна та c-реберна грань. Не втрачаючи загальності, нехай ребро BC прилегле до трикутної грані

FBC і нехай AB прилегле до c-реберної грані. З цього слідує, що AD буде прилеглим до трикутної грані AGD.

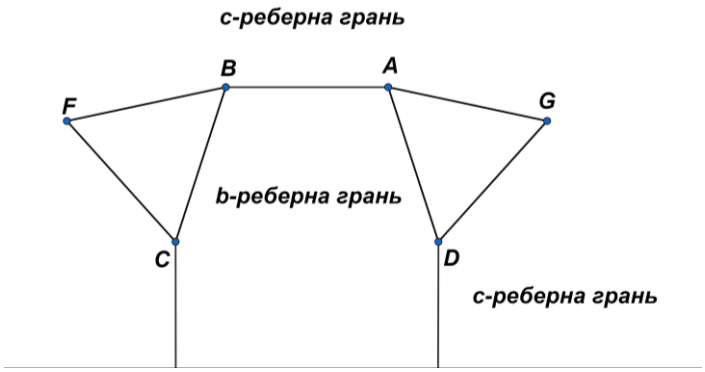


Рис. 2.12 Грань ABCD…

Таким чином, частина b-реберної грані $ABCD$ почергово суміжна з трикутною гранню та з c-реберною гранню. Інакше кажучи, вона має парну кількість суміжних з нею граней. Звідки можна зробити висновок, що b – це парне число.

Якщо $b \geq 12$, то сума пласких кутів кожного багатогранного кута буде не меншою за $60° + 150° + 150° = 360°$, що не можливо. Звідси, можливі значення для b це 4, 6, 8 або 10.

Отже, ми отримаємо наступні фігури: $(3, 6, 6)$ – *усічений тетраедр*, $(3, 8, 8)$ – *усічений куб*, $(3, 10, 10)$ – *усічений додекаедр* та $(3, 4, 4)$ – *трикутна призма*[11, с. 85].

Нехай $a = 4$.

Якщо $b = 4$, то ми отримаємо послідовності виду $(4, 4, n)$, де $n \geq 4$ які визначають куб та нескінчену кількість призм.

Якщо $b \geq 4$, то ми можемо показати, що b та c – парні, використовуючи міркування, які ми раніше навели при розгляді b-реберної грані з вершинами $ABCD$.

Якщо $b \geq 8$, то сума пласких кутів буде не меншою ніж $90° + 135° + 135° = 360°$, що не можливо. Таким чином, $b = 6$.

Розглянемо можливі значення для c. Якщо $c \geq 12$, то сума пласких кутів кожного багатогранного кута буде не меншою ніж $90° + 120° + 150° = 360°$, що не можливо. Таким чином, можливими значеннями будуть для c будуть 6, 8 та 10.

Отже, ми отримаємо наступні фігури: $(4, 6, 6)$ – *усічений октаедр*, $(4, 6, 8)$ – *ромбоусічений кубооктаедр* та $(4, 6, 10)$ – *ромбоусічений ікосододекаедр*.

Нехай $a = 5$.

Так само як і у випадку, коли a дорівнювало 3, ми можемо показати, що $b = c$.

Якщо $b \geq 7$, то сума пласких кутів кожного багатогранного кута буде не меншою за $108° + 128° + 128° = 364°$, що не можливо, оскільки багатогранник випуклий. Таким чином b дорівнює 5 або 6.

Отже, ми отримаємо наступні фігури: $(5, 5, 5)$ – *додекаедр* та $(5, 6, 6)$ – *усічений ікосаедр*.

Частина II. Нехай, $n = 4$ та випуклий однорідний багатогранник описується послідовністю (a, b, c, d), де $a \leq b \leq c \leq d$.

Якщо $a \geq 4$, то сума пласких кутів буде не меншою за $4 * 90° = 360°$, що не можливо. Тому $a = 3$.

Якщо $b \geq 5$, то сума пласких кутів буде не меншою за $60° + 3 * 180° = 360°$, що не можливо. Тому $b = 3$ або $b = 4$. Обидва випадки ми розглянемо окремо.

Нехай $b = 3$.

Якщо $c \geq 6$ тоді сума пласких кутів буде не меншою за $60° + 60° + 2 * 120° = 360°$, що не можливо. Тому $c = 3, 4$ або 5.

Коли $c = 3$, ми отримаємо $(3, 3, 3, 3)$ – *октаедр* та нескінчену кількість *антипризм*[11, с. 85] $(3, 3, 3, n)$, $n > 3$.

Якщо $c = 4$ або 5, то наші фігури будуть визначатися послідовностями, що є перестановками $(3, 3, c, d)$, де $c \leq d$.

Доведемо, що дві трикутні грані не можуть бути розташовані одна за одною навколо довільної вершини багатогранника.

Доведемо від супротивного. Нехай багатогранник задається послідовністю $(3, 3, c, d)$. Розглянемо довільну трикутну грань ABC (рис. 2.13).

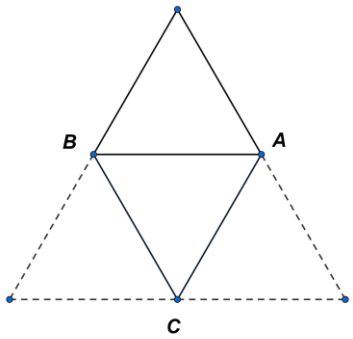


Рис. 2.13 Довільна трикутна грань ABC

За початковим припущенням, навколо кожної вершини грані, одна за одною, будуть розташовані дві трикутні грані. Інакше кажучи, одне із ребер грані ABC буде суміжне до іншої трикутної грані. Не втрачаючи загальності, нехай це буде ребро AB.

Вершина C, у свою чергу, буде суміжна до двох інших трикутних граней, що на рисунку 2.12 позначені пунктирними лініями.

Звідси, або навколо вершини B або навколо вершини A будуть послідовно розташовані три трикутні грані. Маємо суперечність.

Таким чином, багатокутники будуть задаватися послідовностями виду $(3, c, 3, d)$, де $c = 4$ або $c = 5$, при цьому $c \leq d$.

Покажемо, що $c = d$.

Знову розглянемо довільну трикутну грань ABC (рис. 2.14) багатогранника, що задається послідовністю виду $(3, c, 3, d)$.

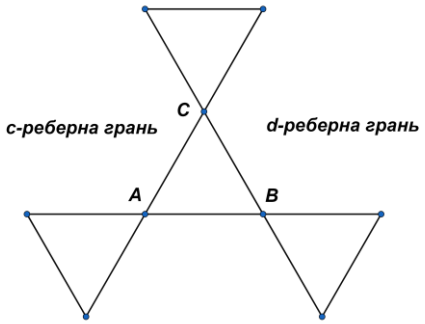


Рис. 2.14 Довільна трикутна грань ABC

Не втрачаючи загальності, нехай ребро AC суміжне c-реберній грані і нехай CB суміжне d-реберній грані. Тоді, ребро AB повинно бути суміжне як c-реберній грані та і d-реберній грані, тому $c = d$.

Отже, ми отримаємо наступні фігури: $(3, 4, 3, 4)$ – *кубооктаедр* та $(3, 5, 3, 5)$ – *ікосододекаедр*.

Нехай $b = 4$.

Якщо $c \geq 5$, то сума пласких кутів буде не меншою за $60° + 90° + 2 * 108° = 366° > 360°$, що не можливо. Тому, $c = 4$.

Якщо $d \geq 6$, то сума пласких кутів буде не меншою за $60° + 90° + 90° + 120° = 360°$, що, знову, не можливо. Тому допустимими значеннями для d є 4 та 5.

Покажемо, що послідовність $(3, 4, 4, 5)$ не відповідає жодному Архімедовому тілу. Для цього розглянемо довільну трикутну грань ABC (рис. 2.15).

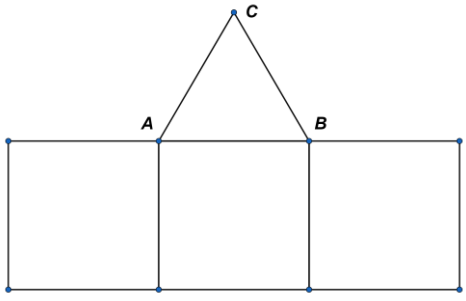


Рис. 2.15 Довільна трикутна грань ABC

Не втрачаючи загальності, нехай AB суміжне з чотирикутною гранню. З цього випливає, що у вершинах A та B, окрім трикутної грані, повинні

зустрічатися чотирикутна та п’ятикутна грані відповідно. Таким чином, ребра AC та CB суміжні з п’ятикутними гранями.

У результаті маємо, що у вершині C будуть зустрічатися дві п’ятикутні грані, що не можливо.

Таким чином, у нас залишаються дві наступні фігури: $(3, 4, 4, 4)$ – *ромбокубооктаедр* та $(3, 4, 5, 4)$ – *ромбоікосододекаедр*.

Частина III. Нехай, $n = 5$ та випуклий однорідний багатогранник описується послідовністю (a, b, c, d, e), де $a \leq b \leq c \leq d \leq e$.

Якщо $d \geq 4$, тоді сума пласких кутів кожного багатогранного кута буде не меншою за $60° + 60° + 60° + 2 * 90° = 360°$, тому $a = b = c = d = 3$.

Якщо $e \geq 6$, тоді сума пласких кутів буде не меншою за $60° + 60° + 60° + 60° + 120° = 360°$, що не можливо. Тому, $e = 3$, 4 або 5.

Отже, ми отримаємо наступні три фігури: $(3, 3, 3, 3, 3)$ – *ікосаедр*, $(3, 3, 3, 3, 4)$ – *курносий куб* та $(3, 3, 3, 3, 5)$ – *курносий додекаедр*.

Теорему доведено.

2.3.2. Побудова Архімедових тіл

Архімедові тіла можна розбити на декілька груп. Першу групу складають багатогранники, які ми отримуємо за рахунок *усікання* Платонових тіл[10, с. 2] (рис. 2.16).

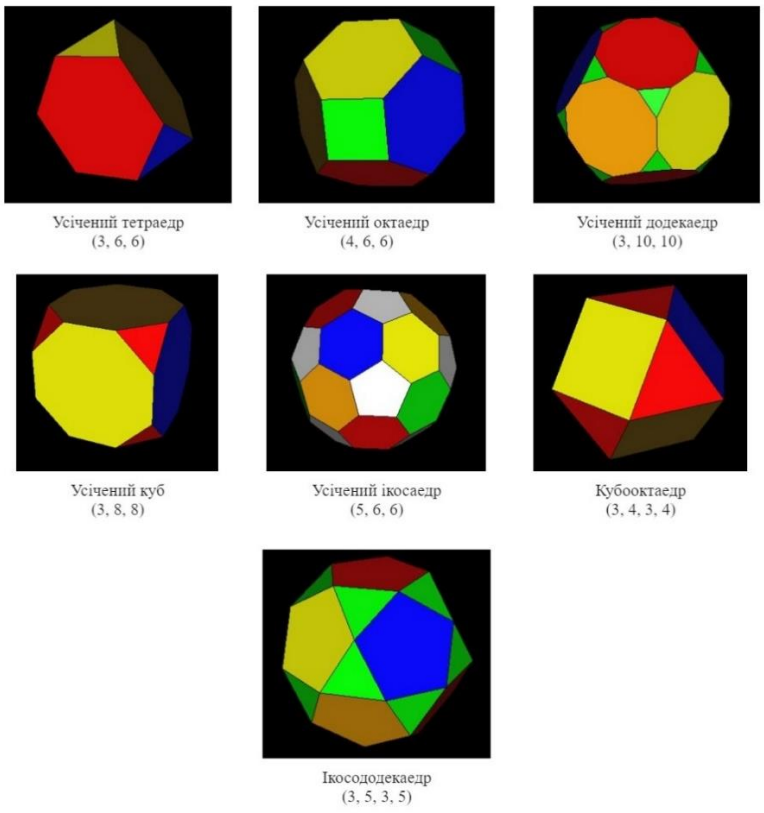


Рис. 2.16 Архімедові тіла отримані через усікання Платонових тіл

Процес *усікання* полягає у тому, що ми прибираєм вершину багатогранника і замість неї, на кожному з ребер що зустрічаються у цій вершині, обираємо точки так, щоб з'єднавши їх ми отримали правильний багатокутник[10, с. 2]. Цей багатокутник буде гранню усіченого багатогранника.

Усічений тетраедр (рис 2.17) має 8 граней, 12 вершин та 18 ребер. Його можна отримати «зрізавши» вершини тетраедру таким чином, щоб замість них залишились рівносторонні трикутники[13, с. 5-6]. Отриманій фігурі буде відповідати послідовність $(3, 6, 6)$.

Тетраедр має 4 грані, 4 вершини та 6 ребер. Після усікання кожної вершини ми отримаємо 4 нові трикутні грані. Кількість вершин та ребер збільшиться втричі, а граней стане на чотири більше.

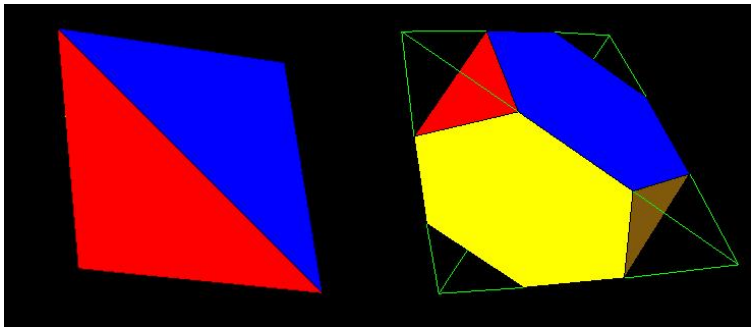


Рис. 2.17 Усічений тетраедр

Застосувавши усікання до октаедру, ми отримаємо *усічений октаедр* (рис 2.18), що задається послідовністю $(4, 6, 6)$ та має 14 граней (6 чотирикутних та 8 шестикутних), 24 вершини та 36 ребер[13, с. 6].

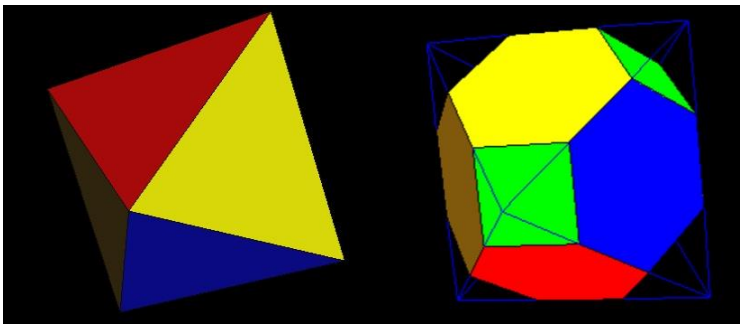


Рис. 2.18 Усічений октаедр

Октаедр має 8 граней, 6 вершин та 12 ребер. Для того щоб отримати усічений октаедр, нам потрібно зрізати кожну вершину так, щоб у результаті отримати квадратну грань. З'явиться шість нових квадратних граней, а кількість вершин збільшить у чотири рази. Кількість ребер збільшиться на 24, оскільки після усікання кожної вершини ми будемо отримувати по чотири нових ребра.

Усічений куб (рис. 2.19), якому відповідає послідовність $(3, 6, 6)$, має 14 граней (8 трикутних та 6 восьмикутних), 24 вершини та 36 ребер[13, с. 4-5].

Для того щоб отримати усічений куб ми зрізаємо кожну вершину куба так, щоб отримати трикутну грань.

Куб має 6 граней, 8 вершин та 12 ребер. Після усікання його вершин, кількість граней збільшиться на вісім, а кількість вершин та граней збільшиться втричі.

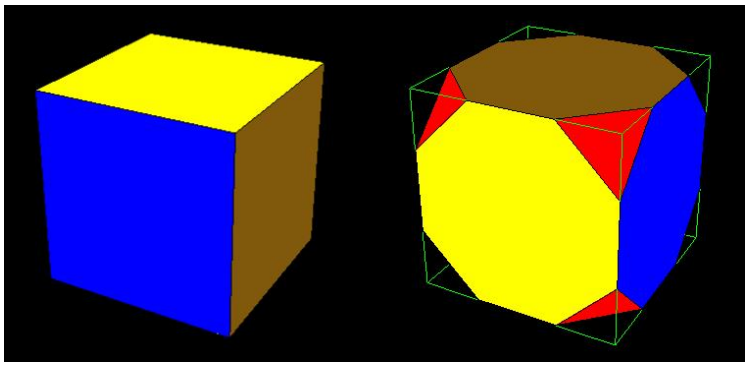


Рис. 2.19 Усічений куб (гексаедр)

Усікання вершин ікосаедру дасть нам фігуру під назвою *усічений ікосаедр* (рис. 2.20), яка позначається послідовністю $(5, 6, 6)$[13, с. 7-8].

Ікосаедр має 20 граней, 12 вершин та 30 ребер. Зрізавши вершини ікосаедра так, щоб на їх місці залишилися правильні п’ятикутники, ми отримаємо усічений ікосаедр.

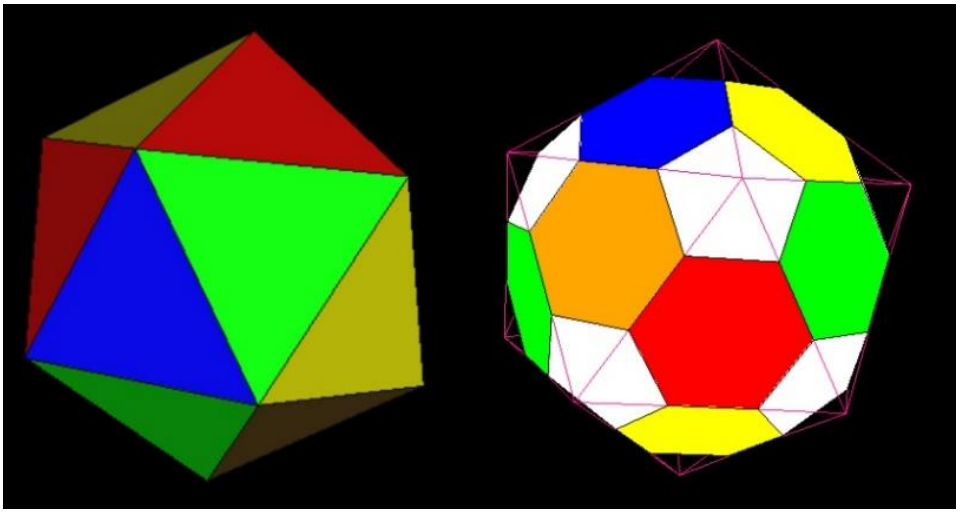


Рис. 2.20 Усічений ікосаедр

Правильних п’ятикутних граней буде 12 і загальна кількість граней буде рівна $20 + 12 = 32$, де 20 – це кількість шестикутних граней, що утворились в результаті операції усікання.

Кожне ребро шестикутної грані суміжне з п’ятикутною гранню, тому для підрахунку кількості вершин буде достатньо порахувати вершини п’ятикутних граней. Таким чином, кількість вершин усіченого ікосаедра рівна $12 * 5 = 60$.

Кількість ребер, з додавання 12 п’ятикутних граней, збільшиться на 60 і буде рівною $60 + 30 = 90$, де 30 – це кількість ребер ікосаедра.

Додекаедр має 12 граней, 20 вершин та 30 ребер. Застосувавши операцію усікання до вершин додекаедру так, щоб на їх місці отримати рівносторонні трикутники, ми отримаємо *усічений додекаедр* (рис. 2.21), який задається послідовністю $(3, 10, 10)$[13, с. 7].

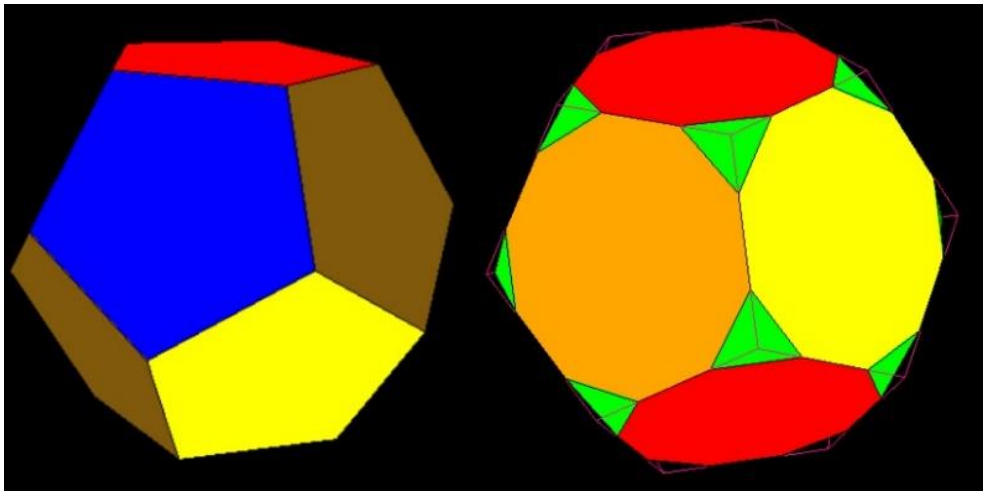


Рис. 2.21 Усічений додекаедр

Оскільки після усікання кожної вершини додекаедра ми отримує одну нову грань, то усічений додекаедр має $12 + 20 = 32$ грані (20 трикутних та 12 десятикутних), 60 вершин та 90 ребер.

Кубооктаедр та *ікосододекаедр*, можна отримати скориставшись *повним усіканням* під час якого ребро початкового багатогранника переходить у точку.

Ми прибираємо вершину багатогранника і по середині ребер, що зустрічаються у цій вершині, ми ставимо точку. З’єднавши ці точки, ми отримаємо грань усіченого багатогранника.

Кубооктаедр (рис. 2.22), якому відповідає послідовність $(3, 4, 3, 4)$, можна отримати виконавши повне усікання куба[13, с. 8-9].

Кубооктаедр буде мати 24 ребра, 12 вершин (по одній вершині на середину ребра куба) та 14 граней (8 трикутних та 6 чотирикутних).

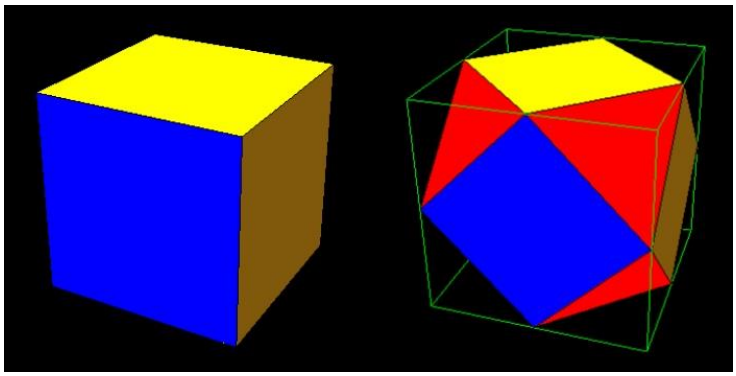


Рис. 2.22 Кубооктаедр

Ікосододекаедр (рис. 2.23), якому відповідає послідовність $(3, 5, 3, 5)$, є результатом повного усікання ікосаедру[13, с. 9-10].

Ікосододекаедр має 30 вершин (по одній вершині на середину ребра ікосаедра), 32 грані (12 п'ятикутних, що з'являються на місці вершин ікосаедра, та 20 трикутних) та 60 ребер.

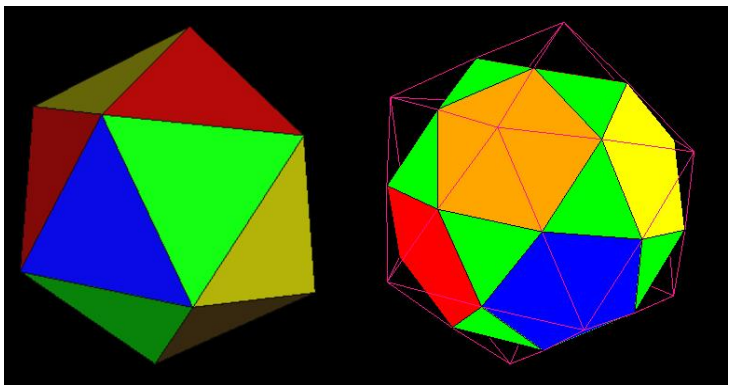


Рис. 2.23 Ікосододекаедр

Останні шість Архімедових тіл, що наведені на рисунках 2.24 та 2.25, вже не можна отримати лише за рахунок усікання Платонових тіл[10, с. 2].

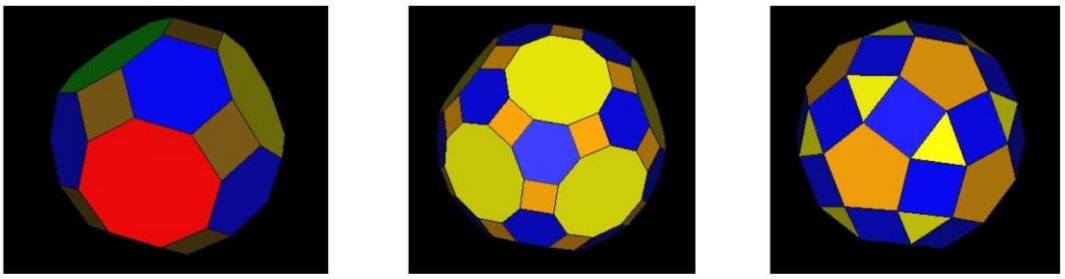


Рис. 2.24 Архімедові тіла, що не можна отримати за рахунок усікання (а)

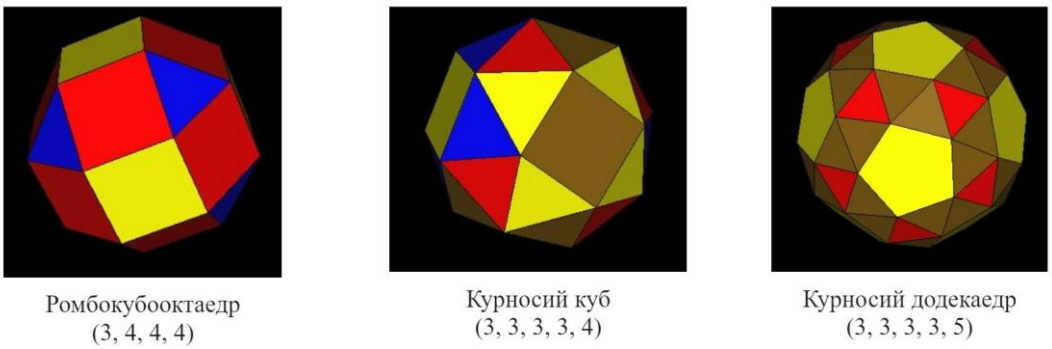
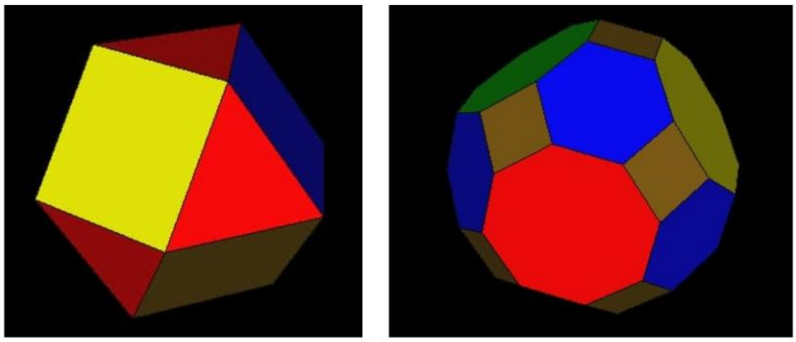


Рис. 2.25 Архімедові тіла, що не можна отримати за рахунок усікання (б)

Ромбоусічений кубооктаедр (рис. 2.26) ми отримуємо методом усікання кубооктаедра, проте, якщо обмежитись лише цим, то на місці зрізаних вершин ми отримаємо прямокутники. Тому, ми змінюємо отримані

грані так, щоб вони мали рівні сторони. Інакше кажучи, перетворюємо їх на квадрати[13, с. 12].

Ромбоусічений кубооктаедр позначається послідовністю $(4, 6, 8)$, та має 26 граней (12 чотирикутних, 8 шестикутних та 6 восьмикутних), 48 вершин та 72 ребра.



Кубооктаедр
(3, 4, 3, 4)

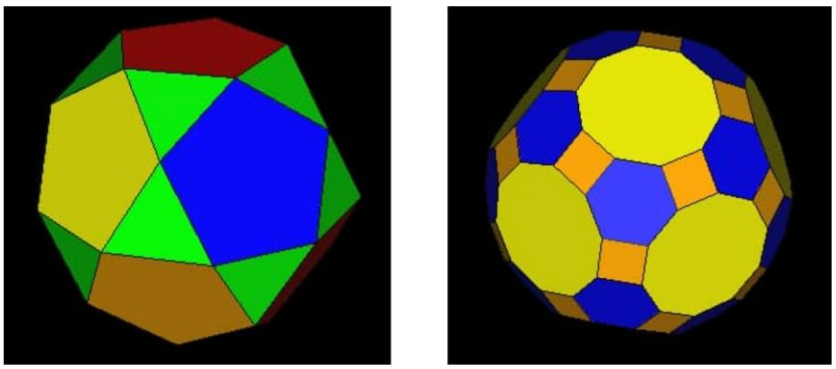
Ромбоусіченний кубооктаедр
(4, 6, 8)

Рис. 2.26 Кубооктаедр та ромбоусічений кубооктаедр

Аналогічним чином, застосовуючи процес усікання до вершин та перетворюючи отримані грані на квадрати, з ікосододекаедра ми отримуємо *ромбоусічений ікосододекаедр* (рис. 2.27), що позначається послідовністю $(4, 6, 10)$[13, с. 11].

Ікосододекаедр має 30 вершин. Після дій, описаних вище, на місці кожної вершини ми отримаємо квадратну грань і кількість граней збільшить на 30. Кількість вершин збільшиться у чотири рази (так як з'являться нові чотирикутні грані), а кількість ребер збільшиться на $4 * 30 = 120$.

Отже, ромбоусічений ікосододекаедр буде мати 62 грані (30 чотирикутних, 20 шестикутних та 12 десятикутних), 120 вершин та 180 ребер.



Ікосододекаедр
(3, 5, 3, 5)

Ромбоусіченний ікосододекаедр
(4, 6, 10)

Рис. 2.27 Ікосододекаедр та ромбоусічений ікосододекаедр

Інші дві фігури: *ромбокубооктаедр* та *ромбоікосододекаедр* ми отримуємо за рахунок операції *розтягнення*[13, с. 12].

Для прикладу, розглянемо розтягнення квадрата.

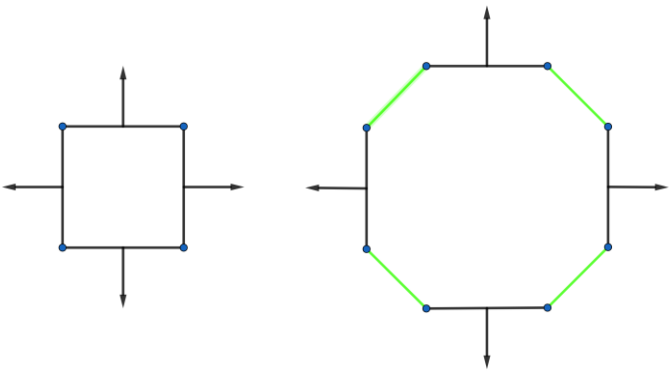


Рис. 2.28 Розтягнення квадрата

На рисунку 2.28 ми проводимо *нормалі*[4, с. 72] до сторін квадрату та починаємо одночасно та рівномірно рухати їх від центру квадрата в напрямку на який вказують нормалі. Сторони, що знаходяться поруч, ми з'єднуємо відрізками для того, щоб утворити багатокутник. На рисунку 2.28 вони позначені зеленим. У результаті ми отримаємо восьмикутник. Якщо сторони отриманої фігури рівні, то таке розтягнення називається *однорідним*.

Таким чином, для того, щоб отримати *ромбокубооктаедр* (рис. 2.29), ми застосовуємо операцію розтягнення до октаедра або до куба[13, с. 14-15].

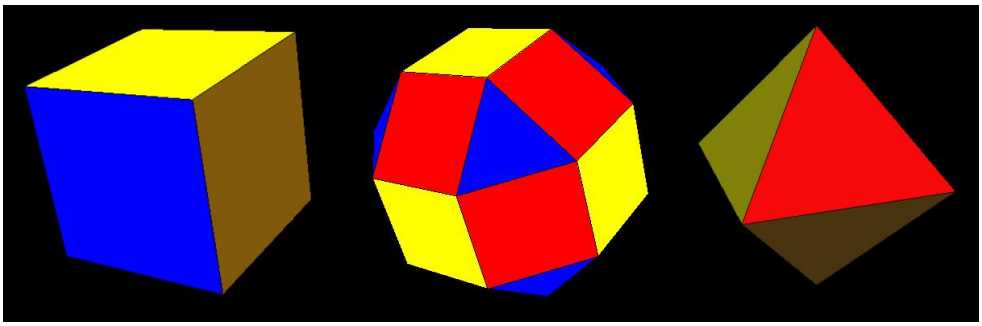


Рис. 2.29 Ромбокубооктаедр

На рисунку 2.29, однорідне розтягнення куба дасть нам жовті грані кубооктаедра. З'єднавши ці грані відрізками, ми отримаємо червоні (квадратні) та сині (трикутні) грані.

Ромбокубооктаедр позначається послідовністю $(3, 4, 4, 4)$ і має 26 граней (8 трикутних та 18 чотирикутних), 24 вершини та 48 ребер.

Ромбоікосододекаедр (рис. 2.30) позначається послідовністю $(3, 4, 5, 4)$ і має 62 грані (20 трикутних, 30 чотирикутних та 12 п'ятикутних), 60 вершин та 120 ребер[13, с. 13].

Цю фігуру можна отримати виконавши однорідне розтягнення додекаедра або ікосаедра.

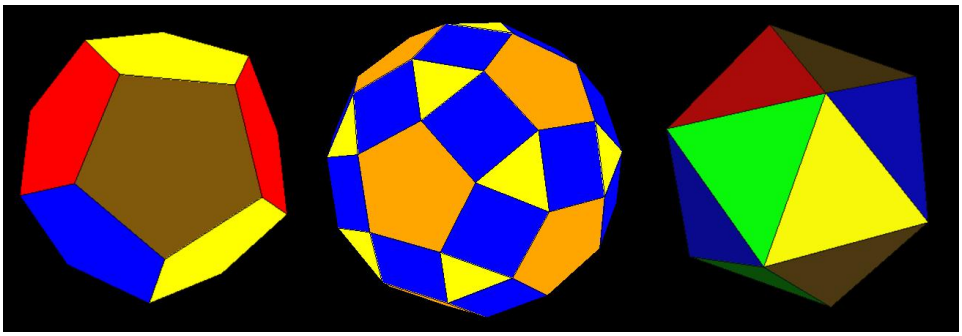


Рис. 2.30 Ромбоікосододекаедр

На рисунку 2.30, якщо у ромбоікосододекаедра прибрати всі чотирикутні та трикутні грані, то «стиснувши» п’ятикутні грані, що залишились, ми отримаємо тетраедр. Аналогічно, якщо прибрати п’ятикутні та чотирикутні грані та «стиснути» ті, що залишились, можна отримати ікосаедр.

Останні два Архімедових тіла, *курносий куб* та *курносий додекаедр*, ми отримуємо застосувавши операцію «*snub*»[14].

Операція «*snub*» схожа на розтягнення: ми рухаємо грані багатокутника від його цента у напрямку нормалей, після чого з’єднуємо ці грані відрізками, щоб утворити нові грані. Однак, після розтягнення, ми повертаємо грані, які ми рухали, за чи проти годинникової стрілки відносно їх центрів.

Застосувавши цю операцію до куба чи октаедра, ми отримаємо *курносий куб*[13, с. 15-16]. Також, курносий куб можна отримати просто повернувши грані ромбокубооктаедра, які ми рухали від центра початкового багатогранника (рис. 2.31). Ці грані ми повертаємо допоки суміжні чотирикутні грані не стануть парами рівносторонніх трикутників.

Курносому кубу відповідає послідовність $(3, 3, 3, 3, 4)$. Він має 38 граней (6 чотирикутних та 32 трикутних), 24 вершини та 60 ребер.

Курносий додекаедр (рис. 2.32), у свою чергу, задається послідовністю $(3, 3, 3, 3, 5)$ та має 92 грані (80 трикутних та 12 п’ятикутних), 60 вершин і 150 ребер.

Для того, щоб утворити *курносий додекаедр* ми застосовуємо операцію «snub» до додекаедра або ікосаедра[13, с. 16-17], або можемо просто повернути грані ромбоікосододекаедра за чи проти годинникової стрілки.

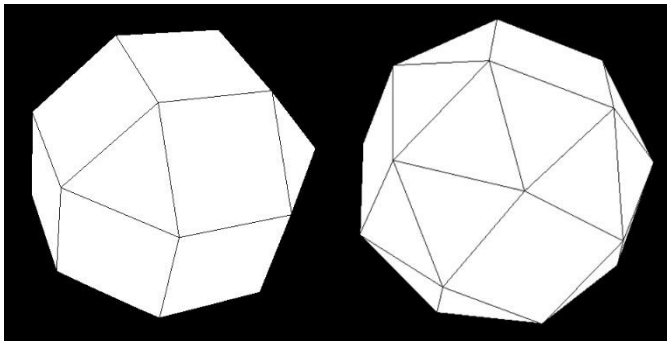


Рис. 2.31 Ромбокубооктаедр та курносий куб

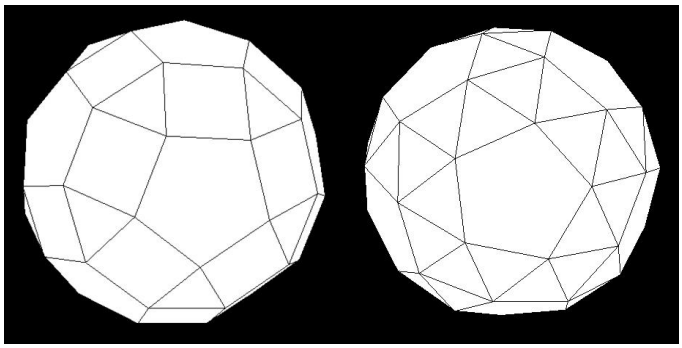


Рис. 2.32 Ромбоікосододекаедр та курносий додекаедр

Щодо того, чи є різниця між тим повертаємо ми грані за, чи проти годинниковою стрілкою, то вона несуттєва. Фігура отримана поворотом граней за годинниковою стрілкою буде дзеркальним відображенням фігури отриманої поворотом граней проти годинникової стрілки.

Такі варіанти фактично одного й того ж тіла називають *енантіоморфними*[10, с. 2-3].

3. Візуалізація багатогранників за допомогою OpenGL

3.1. Створення вікна

Для візуалізації об’єктів за допомогою OpenGL нам потрібно створити вікно, де вони будуть зображені.

Спочатку нам потрібно ініціалізувати деякі змінні з контексту OpenGL. Цю роль візьме на себе функція `main`, яку ми визначимо у модулі `Main`.

```
main :: IO ()
main = do
  (progName, _args) <- getArgsAndInitialize
  initialDisplayMode $= [WithDepthBuffer, RGBMode, DoubleBuffered]
  initialWindowSize $= Size 500 500
```

Лістинг 3.1 Функція main

Як зазначалося у першому розділі, усі функції в Haskell обов’язково повинні приймати та повертати значення. Дивлячись на `main`, можна помітити, що вона не має параметрів, а також не повертає значень. Вона просто виконує послідовність дій.

Таким чином, `main` не є функцією, оскільки порушує правила, які накладає Haskell. `main` є *IO-дією*[7, с. 250-252] (проте інколи, для зручності, ми будемо називати IO-дії функціями). У мові Haskell всі IO-дії мають тип `IO`.

Тип `IO` є параметричним. Таким чином, якщо у результаті виконання IO-дії ми отримуємо, наприклад, ціле число, її тип буде `IO Int`. Якщо у результаті ми нічого не отримуємо, тип буде вказаний як `IO ()`.

Перше, що ми робимо у `main` – викликаємо функцію `getArgsAndInitialize`, що ініціалізує модуль `Graphics.UI.GLUT` та повертає назву програми й аргументи командного рядка[15].

Після цього, ми встановлюємо значення змінних `initialDisplayMode` та `initialWindowSize` з контексту OpenGL, використовуючи оператор «$=».

Оператор «$=» визначений у модулі `Data.StateVar` як функція класу типів `HasSetter`[8, с. 7-8]. Як і `main`, він є IO-дією типу `IO ()`, проте, на відміну від `main`, має два параметри: змінну, з контексту OpenGL, та значення, яке потрібно їй присвоїти.

Ввівши поняття IO-дії, ми можемо розглядати `main` як комбінацію цих дій, при якій вони виконуються одна за одною. Для комбінації IO-дій Haskell має *do-нотацію*, яку можна помітити по ключовому слову `do` на початку `main`.

Змінна `initialWindowSize` відповідає за розміри вікна[4, с. 36].

Змінна `initialDisplayMode` відповідає за властивості вікна [4, с. 36].

У лістингу 3.1 ми вказуємо, що вікно використовує *кольорову модель RGBA*, *подвійну буферизацію*, а також *буфер глибини*.

Кольорова модель[4, с. 148-149] визначає як ми будемо задавати кольори у програмі.

Інформацію про об’єкти, що зображені у вікні, OpenGL зберігає у буферах. *Подвійна буферизація* – це принцип при якому OpenGL використовує два буфери так, щоб для зображення об’єктів у вікні використовувався вміст першого буферу, в той час як у другий буфер будуть записуватися нові дані[4, с. 38-41]. Після того, як ці дані будуть записані, OpenGL почне використовувати дані другого буферу, а нові дані вже будуть записуватись у перший буфер. Таким чином, ми можемо уникнути «миготіння» зображення, коли об’єкти, що зображені у вікні, змінюють своє положення у просторі.

Буфер глибини або z-*буфер* зберігає значення *глибини* для кожної точки об’єкту, який зображений у вікні програми[4, с. 209-210]. Зазвичай, *глибина* – це z-координата точки.

Точки об’єкту OpenGL перетворює у пікселі на екрані комп’ютеру. Значення з буферу глиби використовуються для того, щоб визначити які пікселі потрібно зобразити поверх інших.

Для того, щоб створити вікно, нам потрібно викликати функцію `createWindow`[4, с. 36] та передати їй назву для вікна (лістинг 3.2).

Окрім цього, нам потрібно визначити функцію, яку OpenGL буде викликати для зміни вмісту вікна. Ця функція також буде виконана при першій появі вікна на екрані комп’ютера.

Посилання на неї потрібно присвоїти змінній `displayCallback`[4, с. 36].

```
display :: IO ()
display = do
  clear [ColorBuffer, DepthBuffer]
  swapBuffers

main :: IO ()
main = do
  . . .
  _ <- createWindow "OpenGL Polyhedrons"
  clearColor $= Color4 0.0 0.0 0.0 1.0
  displayCallback $= display
  mainLoop
```

Лістинг 3.2 Створення вікна

Змінна `clearColor` встановлює колір по замовчуванню для кожного пікселя у вікні[4, с. 46-47]. У лістингу 3.2 це чорний колір.

Функція `display` поки що тільки видаляє всі значення, що збережені у *буфері кольору*[4, с. 148] та буфері глибини, використовуючи функцію `clear`.

Буфер кольору містить колір кожного пікселя. Видаливши все з цього буферу, всі пікселі у вікні будуть мати колір по замовчуванню.

Функція `swapBuffers`[4, с. 41] дає OpenGL сигнал про те, що потрібно використати дані з другого буферу для зображення об'єктів у вікні та почати записувати нові дані у перший буфер, або навпаки. Це залежить від того який буфер використовувався OpenGL для запису в момент виклику цієї функції.

Наостанок ми викликаємо функцію `mainLoop` для того щоб OpenGL створив вікно та почав викликати функцію, яку ми присвоїли `displayCallback`[4, с. 36].

У результаті ми отримаємо вікно з чорним фоном, розміром 500 на 500 пікселів та назвою «OpenGL Polyhedrons».

3.2. Візуалізація багатогранника

Процес візуалізації для різних багатогранників не сильно відрізняється, тому його можна розглянути на прикладі одного конкретного. Наприклад куба.

Створимо модуль під назвою `P3_Cube`, що буде містити дані й функції, необхідні для візуалізації куба у вікні.

```
module P3_Cube where

import Graphics.UI.GLUT
import RenderHelper

vertices :: [(GLfloat, GLfloat, GLfloat)]
vertices =
  [ (1.0, 1.0, -1.0), (-1.0, 1.0, -1.0),
    (-1.0, 1.0, 1.0), (1.0, 1.0, 1.0),
    (1.0, -1.0, 1.0), (-1.0, -1.0, 1.0),
    (-1.0, -1.0, -1.0), (1.0, -1.0, -1.0)]

indices :: [[Int]]
indices =
  [[0, 1, 2, 3], [4, 5, 6, 7], [3, 2, 5, 4],
   [7, 6, 1, 0], [2, 1, 6, 5], [0, 3, 4, 7]]

faces :: [PolyFace]
faces =  [ PolyFace (indices !! 0) yellow,
           PolyFace (indices !! 1) yellow,
           PolyFace (indices !! 2) blue,
           PolyFace (indices !! 3) blue,
           PolyFace (indices !! 4) brown,
           PolyFace (indices !! 5) brown ]
```

Лістинг 3.3 P3_Cube.hs

Масив кортежів `vertices` містить координати вершин куба. У масиві `indices` ми зберігаємо інформацію про грані куба. Кожний елемент `indices` містить індекси елементів з `vertices`.

У лістингу 3.3 ми також імпортуємо модуль `RenderHelper`, що містить визначення типу `PolyFace` та трьох констант, які мають тип `Color3`.

```
blue, yellow, brown :: Color3 GLfloat
blue = Color3 0.0 0.0 1.0
yellow = Color3 1.0 1.0 0.0
brown = Color3 0.5 0.35 0.05
```

Лістинг 3.4 Константи для трьох кольорів

Перейдемо тепер до функцій, що будуть зображувати куб у вікні.

```
renderCubeFrame :: Color3 GLfloat -> IO ()
renderCubeFrame color = do
  polygonMode $= (Line, Line)
  let faces = makeSimilarFaces indices color
  renderShadowedPolyFaces faces vertices
  polygonMode $= (Fill, Fill)

renderCube :: IO ()
renderCube = do
  renderShadowedPolyFaces faces vertices
  renderPolygonBoundary $ renderCubeFrame black
```

Лістинг 3.5 Функції renderCube та renderCubeFrame

Функція `renderCube` зображує куб у вікні, використовуючи дані масивів `faces` та `vertices`.

Візуалізація відбувається в два кроки: спочатку ми викликаємо функцію `renderShadowedPolyFaces`, яка зображує грані куба, після чого, поверх граней, ми візуалізуємо каркас куба, скориставшись функціями `renderPolygonBoundary` та `renderCubeFrame`. Каркас куба утворює його контур, що спрощує сприйняття фігури.

Змінна `polygonMode`[4, с. 66] відповідає за те, як OpenGL зображує об'єкти у вікні. Присвоївши їй значення `(Line, Line)`, ми даємо OpenGL вказівку зображувати **всі** графічні об'єкти у вигляді ліній.

Таким чином, знову зобразивши грані, ми отримаємо тільки їх границі, що разом утворять каркас куба.

Після зображення каркасу важливо присвоїти `polygonMode` значення `(Fill, Fill)`, для того щоб графічні об'єкти зображувались повністю зафарбованими.

Змінивши `polygonMode`, ми викликаємо функцію `makeSimilarFaces`, яка повертає масив на кшталт `faces`, у якому для всіх елементів буде визначений один і той самий колір.

```
renderShadowedPolyFaces :: [PolyFace]->[(GLfloat,GLfloat,GLfloat)]->IO ()
renderShadowedPolyFaces polyFaces vertices =
 mapM_ (`renderShadowedPolyFace` vertices) polyFaces

renderShadowedPolyFace :: PolyFace -> [(GLfloat,GLfloat,GLfloat)] -> IO ()
renderShadowedPolyFace (PolyFace faceIndices faceColor) vertices =
 renderPrimitive Polygon $
  do color faceColor
     let faceVertices = polyIndicesToVertices faceIndices vertices
     mapM_ (\(x, y, z) -> vertex $ Vertex3 x y z) faceVertices
```

Лістинг 3.6 Функція renderShadowedPolyFace та renderShadowedPolyFaces

У `renderShadowedPolyFace` (лістинг 3.6) ми викликаємо функцію `renderPrimitive`[8, с. 12], що міститься у модулі `Graphics.UI.GLUT`.

Ця функція малює графічні примітиви у вікні та приймає два аргументи: тип графічного примітива, який потрібно намалювати, та послідовність IO-дій, яка обов'язково повинна містити виклики IO-дії `vertex`[8, с. 12]. Задача `vertex` – зобразити у вікні точку. Для цього вона приймає координати точки у тривимірному просторі.

Зображені точки будуть використані для утворення графічного примітива, який ми вказали першим аргументом при виклику `renderPrimitive`. У нашому випадку це буде `Polygon` – випуклий багатокутник.

У лістингу 3.6 ми встановлюємо колір грані, використовуючи IO-дію `color`[4, с. 59], перетворюємо масив індексів, що відповідають вершинам грані, на масив кортежів з координатами вершин, після чого викликаємо функцію `vertex`.

`renderPolygonBoundary` викликає функцію, що зображує каркас багатогранника, та застосовує до цього каркасу операцію *масштабування*, викликаючи функцію `scale`[4, с. 111-112], що «розтягує» фігуру по осям XYZ.

```
renderPolygonBoundary :: IO () -> IO ()
renderPolygonBoundary renderPolygonFrame = do
  scale 1.0053 1.0053 (1.0053 :: GLfloat)
  renderPolygonFrame
```

Лістинг 3.7 Функція renderPolygonBoundary

Якщо ми просто намалюємо каркас поверх граней багатогранника, то зіштовхнемось з явищем під назвою «*z-fighting*»[16] або «*bleeding*», суть якого полягає у тому, що коли для двох точок графічного об'єкта значення глибини майже однакове, OpenGL не вистачає точності, з якою зберігаються значення у буфері глибини, для того щоб визначити який з пікселів, що відповідають цим точкам, потрібно відобразити поверх іншого на екрані комп'ютера. Тому вони будуть постійно мінятися місцями: інколи перший піксель буде поверх

другого, інколи навпаки. Для запобігання цьому, ми робимо так, щоб точки, що утворюють каркас, були вище граней куба.

У модулі `Main`, доповнимо функції `main` та `display`.

```
display :: State -> DisplayCallback
display state = do clear [ColorBuffer, DepthBuffer]
                   showPolyhedra state
                   swapBuffers
main :: IO ()
main = do . . .
          _ <- createWindow "OpenGL Polyhedrons"
          state <- constructState
          clearColor $= Color4 0.0 0.0 0.0 1.0
          displayCallback $= display state
          reshapeCallback $= Just reshape
          mainLoop
```

Лістинг 3.8 Доповнені функції main та display

Функція `showPolyhedra` визначена у модулі `Main`. Її ми використовуємо для виклику функцій, що зображують той чи інший багатогранник.

```
import P3_Cube

showPolyhedra :: State -> DisplayCallback
showPolyhedra state = do
  polyhedra <- get $ polyhedraId state
  case polyhedra of
    0 -> renderCube
  postRedisplay Nothing
```

Лістинг 3.9 Функція showPolyhedra

У `main` ми створюємо змінну `state` за допомогою функції `constructState` (лістинг 3.10) та присвоюємо змінній `reshapeCallback` посилання на функцію `reshape` (лістинг 3.11).

```
constructState :: IO State
constructState = do
  polyhedra <- newIORef 0
  camera <- newIORef (90 :: Int, 270 :: Int, 8.0)
  return $
    State
      { polyhedraId = polyhedra,
        cameraPos = camera }
```

Лістинг 3.10 Функція constructState

`reshapeCallback`[8, с. 47] містить посилання на функцію, що приймає розмір вікна у вигляді аргументу та визначає яку частину вікна ми використовуємо для візуалізації об'єктів, а також те як ці об'єкти будуть відображатися на екрані комп'ютеру.

За допомогою змінної `viewport`[8, с. 48-49] ми визначаємо частину вікна, яку ми хочемо використати для зображення графічних об'єктів. Ми присвоюємо їй кортеж з двох значень: перше значення – відступ від верхнього лівого краю вікна, друге – розмір області для візуалізації у пікселях. Окрім цього, ми також вказуємо *проекційну матрицю*[4, с. 119-120].

Екран комп'ютера – це двовимірна площина, проте моделі, які ми будуємо за допомогою OpenGL – тривимірні. Проекційна матриця визначає як тривимірні об'єкти будуть спроектовані на екран комп'ютера: за допомогою *ортогональної* чи *перспективної* проекції.

```
reshape :: ReshapeCallback
reshape s@(Size w h) = do
  viewport $= (Position 0 0, s)
  matrixMode $= Projection
  loadIdentity
  ortho (-2) 2 (-2) 2 (-2) 2
  matrixMode $= Modelview 0
```

Лістинг 3.11 Функція reshape

У лістингу 3.12, викликаючи функцію `ortho`, ми вказуємо на те, що буде використана *ортогональна проекція*[4, с. 122-123].

Функція `ortho` приймає шість аргументів, що задають на сцені область у формі прямокутного паралелепіпеда. Кожна пара аргументів – це межі області по осях OX, OY та OZ відповідно. Об'єкти (або їх частини), що потраплять у цю область, будуть зображені на екрані за допомогою ортогональної проекції.

На рисунку 3.1 можна побачити як виглядає ця область на сцені.

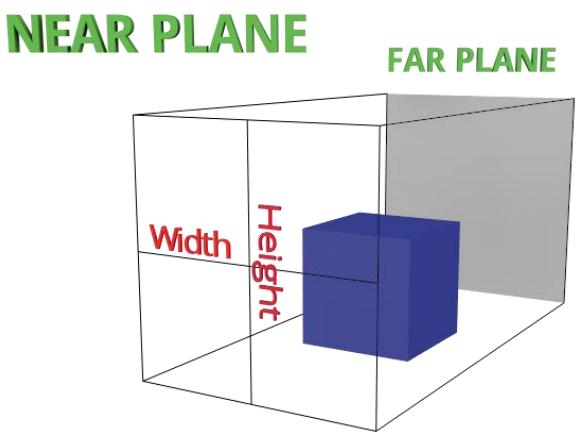


Рис. 3.1 Область, яку задає функція ortho

Частина області, що знаходиться якнайближче до камери, має назву *ближня площина* (англ. *near plane*), а та, що знаходиться якнайдалі, називається *дальня площина* (англ. *far plane*).

Варто також звернути увагу на змінну `matrixMode` та функцію `loadIdentity`[4, с. 105-106] (лістинг 3.12).

Змінна `matrixMode` визначає тип матриці, яку ми змінюємо. Це може бути *модельно-видова* або *проекційна* матриця.

Модельно-видова матриця[4, с. 107-108] змінюється, коли ми викликаємо функції, що виконують геометричні перетворення. Їх всього три[4, с. 110-111]: `scale` (масштабування або «розтягнення» об'єкта по осям XYZ), `translate` (переміщення об'єкта у напрямку заданого вектора) та `rotate` (поворот об'єкта навколо заданої осі).

Кожне геометричне перетворення може бути представлене у вигляді матриці. Ця матриця множиться на модельно-видову матрицю, що потім застосовується до всіх точок об'єкта[4, с. 107-108].

По замовчуванню `matrixMode` має значення `Modelview`, тому, для зміни проекційної матриці, нам потрібно змінити його на `Projection`

Перед викликом `ortho` ми викликаємо функцію `loadIdentity`, яка перетворює проекційну матрицю на одиничну матрицю. Таким чином, ми будемо певні, що функція `ortho` дасть бажаний результат, оскільки проекційну матрицю, як і модельно-видову, можна змінити з будь-якого місця у програмі.

У результаті виконання програми ми отримаємо зображення з рисунку 3.2.

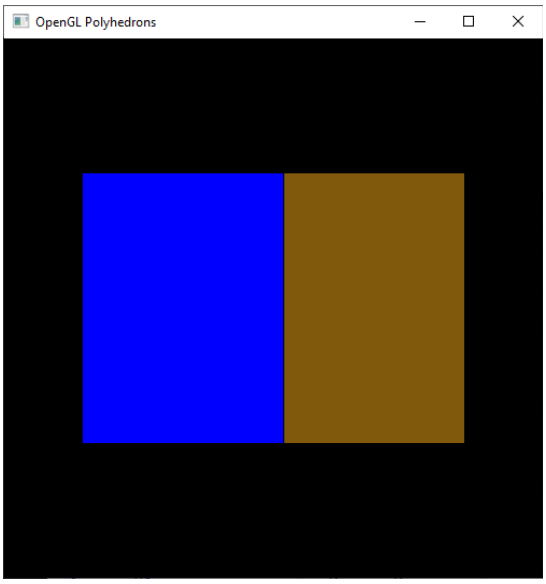


Рис. 3.2 Результат роботи програми

3.3. Перспективна проекція та робота з камерою

Для того, щоб зображення виглядало більш «реалістично», тобто так, щоб частини об’єкту, що знаходяться далі від камери, виглядали меншими ніж ті, що знаходяться ближче, нам потрібно скористатись *перспективною проекцією*[4, с. 120-122].

Для задання перспективної проекції ми використаємо функцію `frustum`[4, с. 120], що задає на сцені область у формі зрізаної чотирикутної піраміди.

`frustum` має шість параметрів: перші чотири, під назвою `left`, `right`, `top` та `bottom`, задають межі ближньої площини по осях OX та OY, останні два параметри, `near` та `far`, задають положення ближньої та дальньої площини на осі OZ.

На рисунку 3.3 можна побачити як виглядає ця область. Чорною сферою на малюнку позначена камера.

Об’єкти (або їх частини), що потраплять у цю область, будуть зображені на екрані за допомогою перспективної проекції.

Область також можна задати за допомогою функції `perspective`[4, с. 121], яка має всього чотири параметри: `fov`, `aspect`, `near` та `far`.

`fov` або «*field of view*» – це кут, що визначає поле зору камери. На рисунку 3.3 він позначений червоним трикутником. `aspect` – це відношення ширини вікна програми до його висоти. Воно потрібне для того щоб враховувати розміри вікна під час обрахунку ширини та висоти ближньої та дальньої площини.

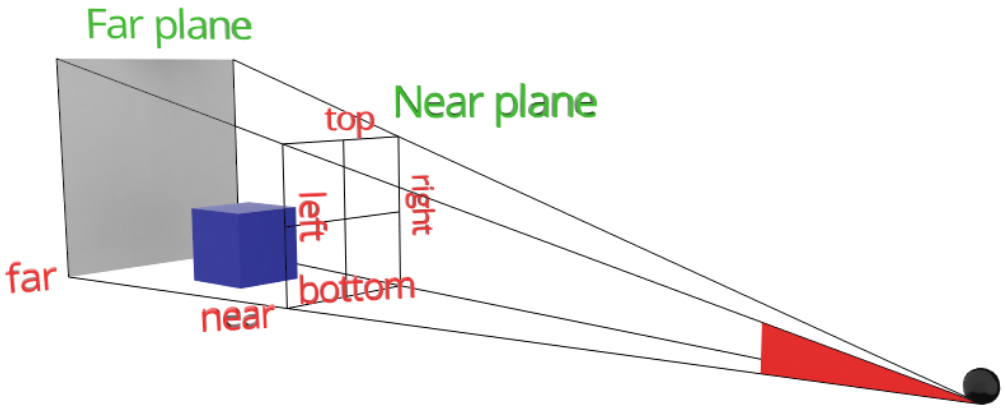


Рис. 3.3 Область, яку задає функція perspective

Недоліком функції `frustum` є те, що вона не враховує розміри вікна при побудові області, а також не дає можливості встановити поле зору камери. Таким чином, функцію `perspective` можна розглядати як певну надбудову над `frustum`.

У функції `reshape` замінимо виклик функції `ortho` на виклик функції `perspective`.

```
reshape :: ReshapeCallback
reshape s@(Size w h) = do
  viewport $= (Position 0 0, s)
  matrixMode $= Projection
  loadIdentity
  let near = 0.05
      far = 10
      fov = 45
      aspect = fromIntegral (w) / fromIntegral (h)
  perspective fov aspect near far
  matrixMode $= Modelview 0
```

Лістинг 3.12 Функція reshape

Повторно запустимо нашу програму. У результаті отримаємо зображення з рисунку 3.4.

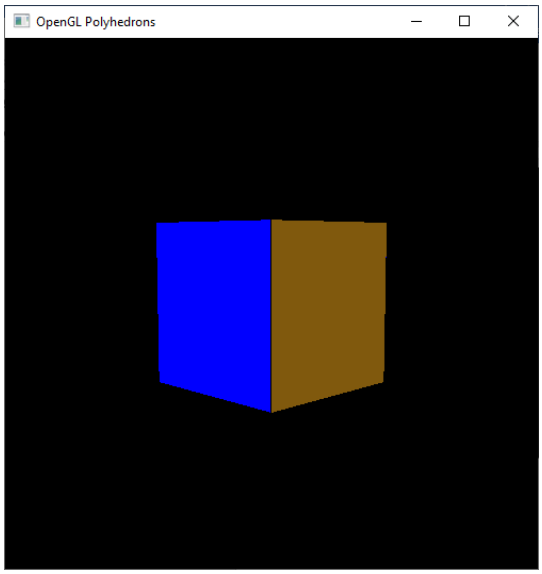


Рис. 3.4 Результат роботи програми

Тепер ми можемо налаштувати камеру, яка дозволить нам оглядати багатогранник з різних сторін.

Камера буде рухатись по сфері, у центрі якої буде знаходитись наш багатогранник. Позиція камери, у *сферичних координатах*, буде зберігатися у змінній `state`, яку ми створили у `main` (лістинг 3.8).

Сферичними координатами[17] точки $A(x, y, z)$ (рис. 3.5) називаються три числа: ρ, β, α, де ρ – це довжина радіус-вектора точки A, β – кут між проекцією вектора $\overrightarrow{OA}$ на площину OXY та віссю OX, α – кут між додатнім напрямом осі OZ та радіус-вектором $\overrightarrow{OA}$.

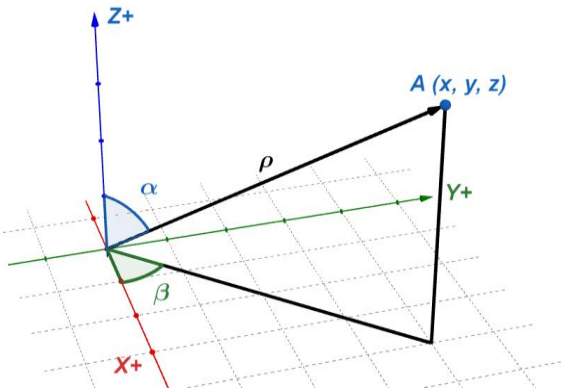


Рис. 3.5 Сферичні координати

Сферичні та декартові координати точки A пов'язані наступними співвідношеннями:

$$x = \rho cos\beta sin\alpha,\ \ y = \rho sin\beta sin\alpha,\ \ z = \rho cos\alpha$$
$$\rho \geq 0,\ \ 0 \leq \beta \leq 2\pi,\ \ 0 \leq \alpha \leq \pi$$

В OpenGL використовується *права декартова система координат*[18], яка зображена на рисунку 3.6.

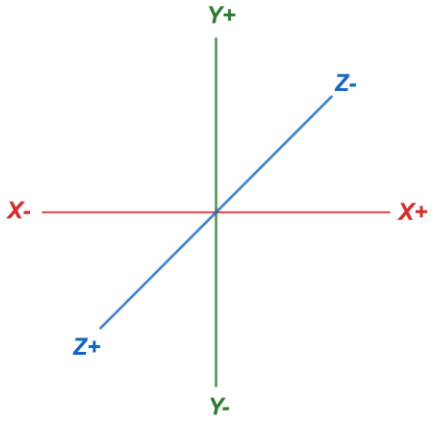


Рис. 3.6 Система координат OpenGL

Якщо вважати, що екран комп'ютера знаходиться в центрі цієї системи координат, то додатно напрямлена вісь OX буде знаходитись праворуч, додатно напрямлена вісь OY буде вказувати вгору, а додатно напрямлена вісь OZ буде перпендикулярна екрану та вказуватиме на користувача.

На рисунках 3.5 та 3.6 можна помітити, що система координат яку використовує OpenGL, відрізняється від звичайної положенням осей. Таким чином, декартові та сферичні координати будуть пов'язані наступними рівностями:

$$x = \rho cos\beta sin\alpha,\ \ z = \rho sin\beta sin\alpha,\ \ y = \rho cos\alpha$$
$$\rho \geq 0,\ \ 0 \leq \beta \leq 2\pi,\ \ 0 \leq \alpha \leq \pi$$

Створимо модуль `OrbitPointOfView` та перенесемо туди функцію `reshape`. Додатково визначимо функції `setPointOfView`, `calculatePointOfView`, `keyForPos` та `modPos`.

Функція `calculatePointOfView` отримує на вхід сферичні координати точки та повертає її координати у декартовій системі.

```
calculatePointOfView :: Int->Int->GLdouble->(GLdouble,GLdouble,GLdouble)
calculatePointOfView alp bet r =
  let alpha = fromIntegral alp * 2 * pi / 360
      beta = fromIntegral bet * 2 * pi / 360
      y = r * cos alpha
      u = r * sin alpha
      x = u * cos beta
      z = u * sin beta
   in (x, y, z)
```

Рис. 3.7 Функція calculatePointOfView

Функція `setPointOfView` обраховує координати камери у декартовій системі координат та координати вектора, що вказує вгору відносно положення камери. На рисунку 3.8 цей вектор позначений зеленим, у той час як камера – це фіолетовий куб.

```
setPointOfView :: IORef (Int, Int, GLdouble) -> IO ()
setPointOfView pPos = do
  (alpha, beta, r) <- get pPos
  let (x, y, z) = calculatePointOfView alpha beta r
      (x2, y2, z2) = calculatePointOfView ((alpha + 90) `mod` 360) beta r
  lookAt (Vertex3 x y z) (Vertex3 0 0 0) (Vector3 x2 y2 z2)
```

Лістинг 3.13 Функція setPointOfView

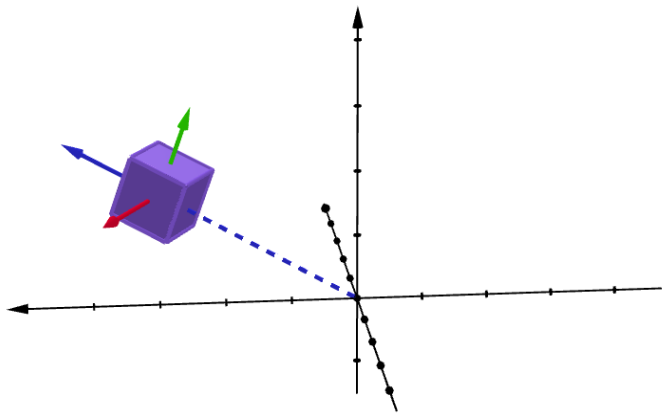


Рис. 3.8 Камера

Обраховані значення передаються у вигляді першого та третього аргументів в функцію `lookAt`[4, с. 116-117]. Другим аргументом є кінець вектора, що визначає напрямок камери.

В OpenGL камера не має графічного представлення. Це просто абстрактне поняття, яким ми користуємось, коли хочемо вказати звідки та з якого ракурсу ми спостерігаємо за об'єктами на сцені[19]. Таким чином, ми не можемо змінити позицію камери, проте ми можемо симулювати цю зміну. Наприклад, коли нам потрібно перемістити камеру праворуч, всю сцену ми перемістимо ліворуч. У цьому нам може допомогти функція `lookAt`.

Використовуючи передані їй значення, вона обраховує два вектори, які на рисунку 3.8 позначені синім та червоним. Разом з зеленим вектором вони утворюють ортонормований базис нової декартової прямокутної системи координат[19].

Наостанок функція створює матрицю переходу до цієї системи координат, яка буде застосована до всіх об'єктів на сцені[19]. Вона має наступний вигляд:

$$lookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Вектори R, U та D на рисунку 3.8 позначені червоним, зеленим та синім кольором відповідно. Вектор P – це позиція камери.

Правий множник матриці `lookAt` перемістить сцену так, щоб камера була в її центрі. Лівий множник поверне сцену так, щоб додатній напрям осі OZ початкової системи координат збігався з напрямом вектора D.

У результаті, камера буде знаходитись у центрі сцени, а об'єкти на сцені тепер будуть розташовані відносно камери.

Останні дві функції з модуля `OrbitPointOfView`: `keyForPos` та `modPos` (лістинг 3.14) будуть зчитувати ввід користувача з клавіатури та змінювати положення камери відповідно до того, які клавіші було натиснуто.

Функція `keyForPos` приймає на вхід позицію камери у сферичних координатах та клавішу, натиснуту користувачем, після чого вона викликає функцію `modPos` для зміни позиції камери.

При натисканні клавіш «+» та «-», а також при прокрутці коліщатка миші буде змінюватись радіус сфери по якій рухається камера. Якщо на клавіатурі натискати на клавіші зі стрілками, то камера буде рухатись по сфері вліво/вправо або вгору/вниз.

`modPos` приймає позицію камери у сферичних координатах та кортеж з трьох лямбда-функцій, які ми використовуємо для зміни сферичних координат камери.

```
keyForPos :: IORef (Int, Int, GLdouble) -> Key -> IO ()
keyForPos pPos (Char '+') = modPos pPos (id, id, \x -> x -0.1)
keyForPos pPos (Char '-') = modPos pPos (id, id, (+) 0.1)
keyForPos pPos (MouseButton WheelUp) = modPos pPos (id, id, \x -> x -0.3)
keyForPos pPos (MouseButton WheelDown) = modPos pPos (id, id, (+) 0.3)
keyForPos pPos (SpecialKey KeyLeft) = modPos pPos (id, (+) 359, id)
keyForPos pPos (SpecialKey KeyRight) = modPos pPos (id, (+) 1, id)
keyForPos pPos (SpecialKey KeyUp) = modPos pPos ((+) 1, id, id)
keyForPos pPos (SpecialKey KeyDown) = modPos pPos ((+) 359, id, id)
keyForPos _ _ = return ()

modPos :: IORef (Int, Int, GLdouble) -> (Int -> Int, Int -> Int, GLdouble -
> GLdouble) -> IO ()
modPos pPos (ffst, fsnd, ftrd) = do
  (alpha, beta, r) <- get pPos
  pPos $= (ffst alpha `mod` 360, fsnd beta `mod` 360, ftrd r)
  postRedisplay Nothing
```

Лістинг 3.14 Функції keyForPos та modPos

Оскільки `modPos` змінює позицію камери, то нам потрібно повідомити OpenGL про те, що зміст вікна потрібно перемалювати. Для цього ми користуємось функцією `postRedisplay`[4, с. 36].

Наостанок, доповнимо функції `main` та `display` з модуля `Main`.

```
import OrbitPointOfView

myKeyboardCallback :: State -> KeyboardMouseCallback
myKeyboardCallback _ (Char '\27') Down _ _ = exitSuccess
myKeyboardCallback State {cameraPos = pPos} key _ _ _ = keyForPos pPos key

main :: IO ()
main = do
  (progName, _args) <- getArgsAndInitialize
  initialDisplayMode $= [WithDepthBuffer, RGBMode, DoubleBuffered]
  initialWindowSize $= Size 500 500
  _ <- createWindow "OpenGL Polyhedrons"
  state <- constructState
  clearColor $= Color4 0.0 0.0 0.0 1.0
  displayCallback $= display
  reshapeCallback $= Just reshape
  keyboardMouseCallback $= Just (myKeyboardCallback state)
  mainLoop
```

Лістинг 3.15 Функція main та myKeyboardCallback

```
display :: State -> DisplayCallback
display state = do clear [ColorBuffer, DepthBuffer]
                   loadIdentity
                   setPointOfView $ cameraPos state
                   showPolyhedra state
                   swapBuffers
```

Лістинг 3.16 Функція display

Ми присвоюємо змінній `keyboardMouseCallback` посилання на функцію `myKeyboardCallback`[4, с. 38]. Її OpenGL буде викликати для обробки користувацького вводу.

Функція `myKeyboardCallback` зчитує ввід з клавіатури. Якщо користувач натисне клавішу «Esc», то програму буде завершено. Інакше, викликається функція `keyForPos` для зміни позиції камери.

Тепер, після запуску програми, ми можемо змінювати позицію камери, використовуючи клавіші на клавіатурі.

3.4. Освітлення

Для обрахунків, пов’язаних з освітленням, OpenGL використовує *нормалі*[4, с. 72]. У нашій програмі ми будемо обраховувати нормаль окремо для кожної грані багатогранника, використовуючи *метод Ньюелла*[20].

Нехай нормаль N, довільної грані багатокутника, що задається вершинами $(p_1, \dots, p_n)$, де $p_i = (x_i, y_i, z_i)$, має координати (n_x, n_y, n_z), тоді:

$$n_x = \sum_{i=0}^{N-1} (y_i - y_{i+1})(z_i + z_{i+1})$$

$$n_y = \sum_{i=0}^{N-1} (z_i - z_{i+1})(x_i + x_{i+1})$$

$$n_z = \sum_{i=0}^{N-1} (x_i - x_{i+1})(y_i + y_{i+1})$$

Для обрахунку нормалей ми використовуємо функцію `calculateSurfaceNormal`, що приймає на вхід масив точок та повертає нормалізований вектор нормалі.

Обрахунки координат вектора відбуваються за формулами вище. У процесі ми користуємось функцією `makePointPairs`, яка зі списку з n точок утворює пари точок вигляду $(p_1, p_2), \dots, (p_n, p_1)$, де $p_i = (x_i, y_i, z_i)$.

```
calculateSurfaceNormall :: [(GLfloat, GLfloat, GLfloat)] -> Vector3 GLfloat
calculateSurfaceNormall points = normalizeVector (Vector3 nX nY nZ)
  where pointPairs = makePointPairs points 0 []
        nX = foldl (\res (current, next) -> res + …) 0.0 pointPairs
        nY = foldl (\res (current, next) -> res + …) 0.0 pointPairs
        nZ = foldl (\res (current, next) -> res + …) 0.0 pointPairs
```

Лістинг 3.17 Функція calculateSurfaceNormal

`calculateSurfaceNormal` ми будемо викликати у `renderShadowedPolyFace` для обрахунку нормалей багатогранника.

```
renderShadowedPolyFace :: PolyFace -> [(GLfloat,GLfloat,GLfloat)] -> IO ()
renderShadowedPolyFace (PolyFace faceIndices faceColor) vertices =
 renderPrimitive Polygon $
  do color faceColor
     let faceVertices = polyIndicesToVertices faceIndices vertices
     let (Vector3 nX nY nZ) = calculateSurfaceNormal faceVertices
     normal (Normal3 nX nY nZ)
     mapM_ (\(x, y, z) -> vertex $ Vertex3 x y z) faceVertices
```

Лістинг 3.18 Функція renderShadowedPolyFace

Обраховані координати нормалі ми передаємо функції `normal`[4, с. 72]. Вона присвоїть нормаль графічному примітиву, що з'явиться у результаті виклику функції `renderPrimitive`.

Обрахувавши нормалі для кожної грані багатогранника, нам залишається тільки налаштувати освітлення.

У OpenGL освітлення працює за рахунок зміни кольору об'єкта[4, с. 161]. Кінцевий колір об'єкта буде залежати від кольору самого об'єкта, від кольору джерела світла та від положення об'єкта відносно джерела світла. Колір джерела світла складається з кольорів його трьох компонент: *фонового світла*, *розсіяного світла*, та *відбитого світла*[4, с. 164-165].

Фонове світло відображає особливість світла відбиватись від різних поверхонь, тим самим досягаючи місць, які не можна побачити напряму з позиції джерела світла. Таким чином, об'єкти на сцені, що не освітлюються прямими променями з джерела світла, завжди будуть освітлені фоновим світлом.

Розсіяне світло надходить безпосередньо з джерела світла. Принцип роботи розсіяного світла у OpenGL проілюстрований на рисунку 3.9.

Жовтим кругом позначене джерело світла. З нього виходить промінь та потрапляє на поверхню об'єкта. Нормаль поверхні – це вектор $\overline{N}$, α – це кут між нормаллю та променем світла. Чим менший цей кут, тим більший вплив буде мати розсіяне світло на колір об'єкта.

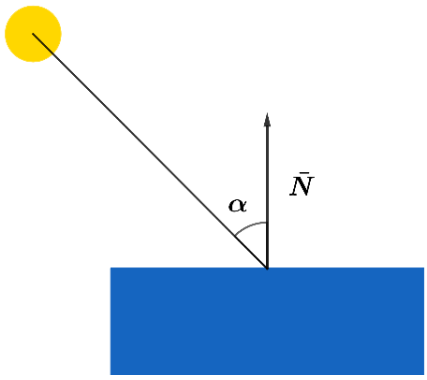


Рис. 3.9 Принцип роботи розсіяного світла

Наприклад, нехай на сцені знаходиться куб синього кольору. Грані куба, розташовані безпосередньо перед джерелом світла, будуть мати яскравий синій колір, водночас, грані, розташовані під кутом до джерела світла, будуть мати менш яскравий колір.

Відбите світло характеризує властивість поверхні відбивати світло. Якщо ми освітлимо повністю гладку поверхню, то на цій поверхні можна буде побачити відблиск. Чим краще поверхня відбиває світло, тим менший та яскравіший цей відблиск.

Принцип роботи відбитого світла у OpenGL можна проілюструвати наступним чином:

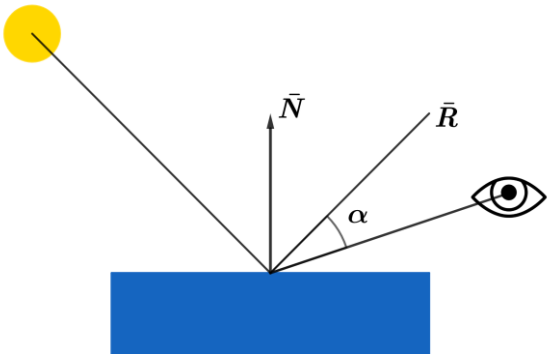


Рис. 3.10 Принцип роботи відбитого світла

З джерела світла виходить промінь та потрапляє на поверхню об’єкта у точці з якої виходить нормаль $\bar{N}$. Після цього він відбивається від поверхні. Відбитий промінь позначено як $\bar{R}$.

З права знаходиться камера, напрямлена на те місце, куди потрапив промінь світла. α – це кут між відбитим променем світла та напрямом камери. Чим менший цей кут, тим яскравішим буде відблиск на поверхні об’єкта.

Для того щоб налаштувати освітлення нам потрібно вказати колір який буде мати кожна з складових джерела світла. Ми також можемо вказати який колір буде мати сам об’єкт під впливом, окремо, фонового, розсіяного та

відбитого світла. Всі ці кольори будуть враховані під час визначення кінцевого кольору об’єкта.

Вказати кольори можна змінивши значення деяких змінних з контексту OpenGL (лістинг 3.19).

Присвоївши змінній `rescaleNormal` значення `Enabled`, ми повідомили OpenGL про те, що, коли до об’єкта на сцені застосовується операція масштабування, його нормалі теж потрібно масштабувати та нормалізувати[1, 73]. Якщо цього не зробити, то після операції масштабування нормалі об’єкта вже не будуть перпендикулярними (рис. 3.11).

```
main :: IO ()
main = do
  . . .
  state <- constructState
  rescaleNormal $= Enabled
  materialSpecular Front $= Color4 0.5 0.5 0.5 1
  materialShininess Front $= 128
  colorMaterial $= Just (Front, AmbientAndDiffuse)
  position (Light 0) $= Vertex4 0 0 2 0
  lightModelAmbient $= Color4 0.1 0.1 0.1 1
  light (Light 0) $= Enabled
  . . .
```

Лістинг 3.19 Функція main

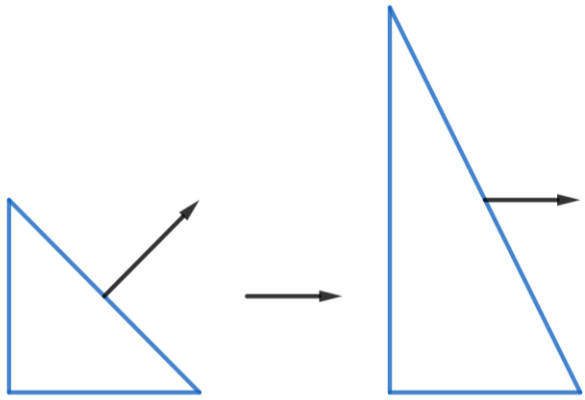


Рис. 3.11 Вплив операції масштабування на нормаль

Змінна `materialSpecular`[4, с. 183] встановлює колір, який буде мати об’єкт під впливом відбитого світла. `materialShininess`[4, с. 183] визначає наскільки добре об’єкт відбиває світло. Ця змінна приймає значення від 0 до 128[4, с. 186]. Чим більше значення, тим меншим і яскравішим буде відблиск на поверхні.

Змінна `position` встановлює позицію джерела світла. Для цього потрібно вказати координати та назву джерела світла[4, с. 170-171]. У нашому випадку, джерело світла під назвою `Light 0` буде мати координати $(0, 0, 2)$. Всього у OpenGL наявно 8 джерел світла з назвами `Light 0`, …, `Light 7`.

`lightModelAmbient` встановлює колір фонового світла одночасно для всіх джерел[4, с. 181].

Наостанок, за допомогою функції `light`, ми активуємо джерело світла `Light 0`. Якщо тепер запустити програму, то можна буде побачити освітлений куб (рис. 3.12).

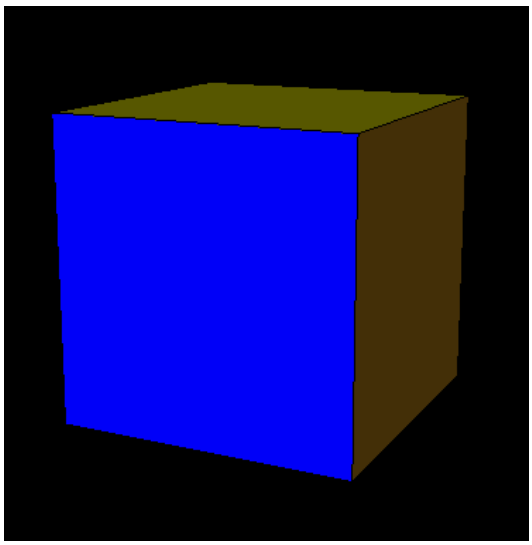


Рис. 3.12 Освітлений куб

4. Зірчаті тіла та об’єднання

4.1. Загальні відомості

Зірчатий багатокутник – це фігура яку можна отримати, продовживши сторони правильного багатокутника допоки вони не перетнуться[10, с. 34]. Точки перетину будуть вершинами зірчатого багатокутника, а сам зірчатий багатокутник буде називатися *зірчатою формою* правильного багатокутника, з якого він був отриманий.

Варто зазначити, що не з кожного правильного багатокутника можна отримати зірчастий. Наприклад продовження сторін рівностороннього трикутника або квадрата перетинатися не будуть (рис. 4.1).

Проте, якщо продовжити сторони правильно п’ятикутника, отримаємо п’ятикутну зірку, або ж *пентаграму* (рис. 4.2).

Пентаграму ми можемо розглядати у вигляді багатокутника з 5 сторонами, при цьому, ці сторони перетинаються[10, с. 34].

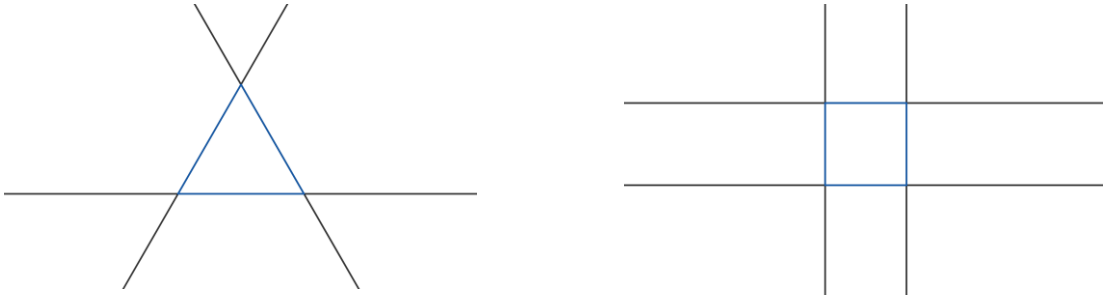


Рис. 4.1 Продовження сторін рівностороннього трикутника та квадрата

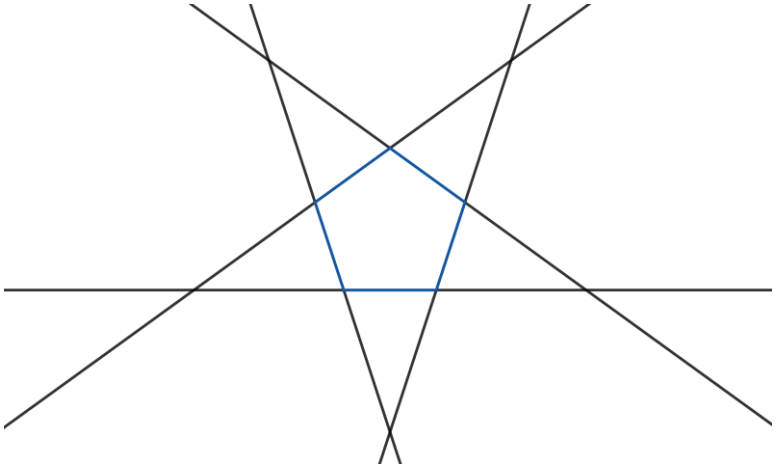


Рис. 4.2 Пентаграма

Ми також можемо задати обхід пентаграми по її вершинам навколо її центру. Порядок обходу вершин зображений на рисунку 4.3. Під час цього обходу ми оминаємо центр пентаграми 2 рази. Варто також зазначити, що внутрішні точки перетину сторін пентаграми ми не вважаємо вершинами.

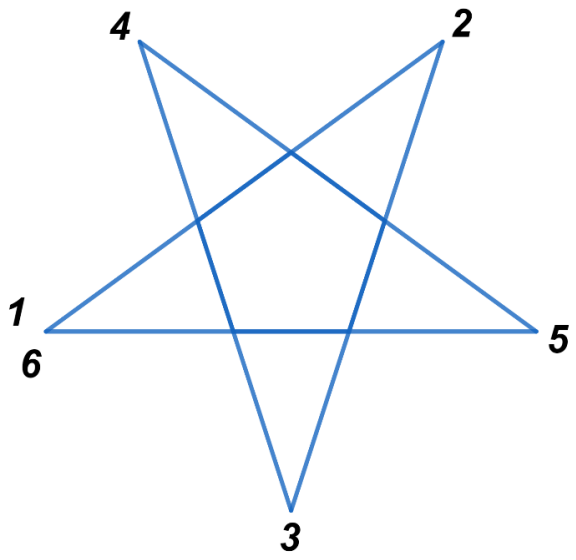


Рис. 4.3 Обхід вершин пентаграми

Символічно пентаграму можна позначити парою чисел $5/2$, де 5 – це кількість вершин, а 2 – це стільки разів ми оминаємо центр під час обходу цих вершин[10, с. 35].

Зірчатим багатогранником називається фігура, що була отримана за рахунок продовження граней багатогранника допоки вони не перетнуться. При перетині граней утворюються відрізки, що будуть ребрами зірчатого багатогранника. Ці ребра також будуть перетинатися, проте внутрішні точки перетинів не будуть вважатися вершинами зірчатого багатогранника[10, с. 35].

Правильним зірчатим багатогранником називається зірчатий багатогранник, гранями якого є однакові правильні або зірчаті багатокутники. Правильні зірчаті багатогранники не є випуклими[10, с. 35].

4.2. Тіла Кеплера-Пуансо та об’єднання однорідних багатогранників

Розглянемо зірчаті форми Платонових тіл.

Якщо ми продовжимо грані тетраедра, то перетин чотирьох площин буде обмежувати частину простору, яка збігається з самим тетраедром. Аналогічний результат ми отримаємо і для куба.

Продовживши грані октаедра ми побачимо, що ребра, по яким вони перетинаються, є ребрами малих правильних тетраедрів які розташовані на гранях цього октаедру (рис. 4.4).

Отриманий правильний зірчатий багатогранник називається *зірчатим октаедром*[10, с. 35] або *stella octangula* (лат. восьмикутна зірка). Його можна побачити на рисунках 4.4 та 4.5.

Цю фігуру також можна отримати об’єднавши два правильних тетраедри, центри яких збігаються з центром октаедра (рис. 4.5). Інакше кажучи, зірчатий октаедр можна розглядати як *об’єднання однорідних багатогранників*.

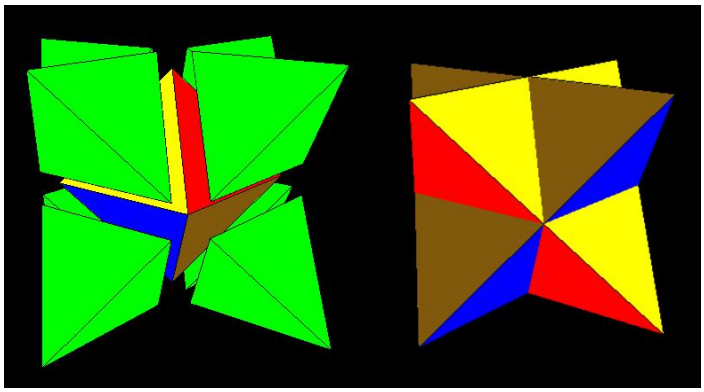


Рис. 4.4 Зірчатий октаедр (а)

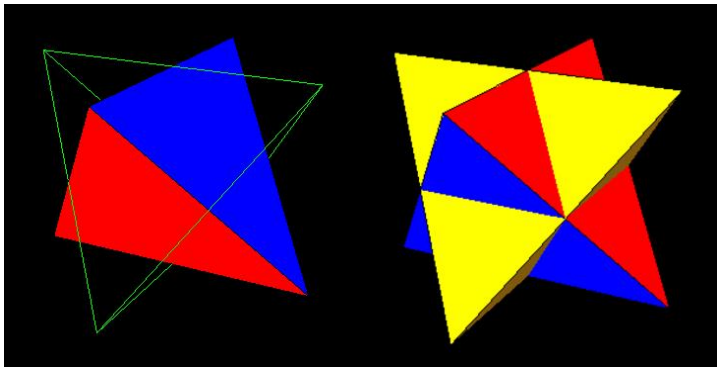


Рис. 4.5 Зірчатий октаедр (б)

Гранями зірчатого октаедра є 8 рівносторонніх трикутників. Він має 12 ребер та 8 вершин[21]. Зірчатий октаедр також є єдиною зірчатою формою октаедра[10, с. 35].

Додекаедр має три зірчаті форми: *малий зірчатий додекаедр*, *великий додекаедр* та *великий зірчатий додекаедр*[10, с. 35].

Малий зірчатий додекаедр (рис. 4.6) має символ Шлефлі $\{5/2, 5\}$ та складається з 12 вершин та 30 ребер[21]. Його гранями є 12 пентаграм.

Перетинаючи один одного, пентаграми утворюють п’ятикутні піраміди, тому, малий зірчатий додекаедр можна також розглядати як додекаедр на гранях якого було побудовано 12 п’ятикутних пірамід (рис. 4.7).

Великий додекаедр (рис. 4.8) має символ Шлефлі $\{5,5/2\}$. Він складається з 12 вершин та 30 ребер[21]. Його гранями є 12 п’ятикутників, які, перетинаючись, утворюють візерунок схожий на пентаграму.

Порівнюючи великий додекаедр та малий зірчатий додекаедр, можна помітити, що вершинами п’ятикутних граней великого додекаедра є вершини п’ятикутних пірамід малого зірчатого додекаедра (рис. 4.9).

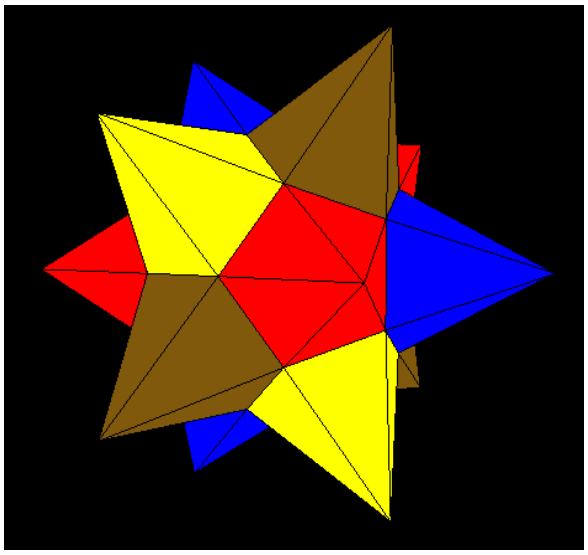


Рис. 4.6 Малий зірчатий додекаедр (а)

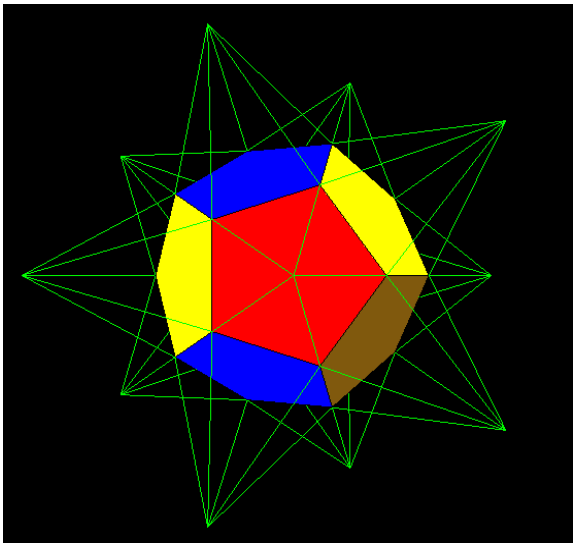


Рис. 4.7 Малий зірчатий додекаедр (б)

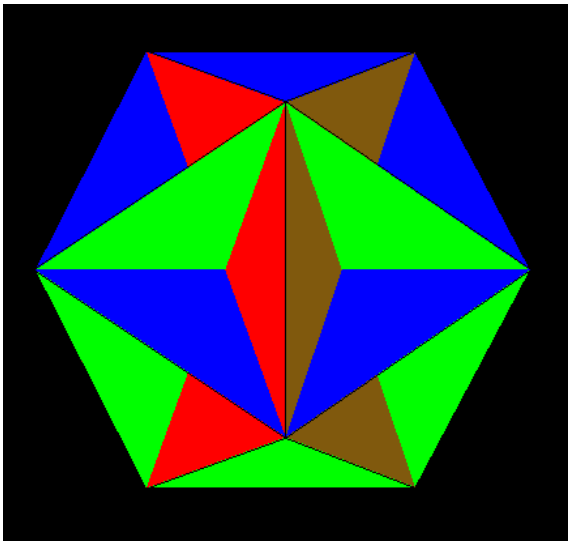


Рис. 4.8 Великий додекаедр

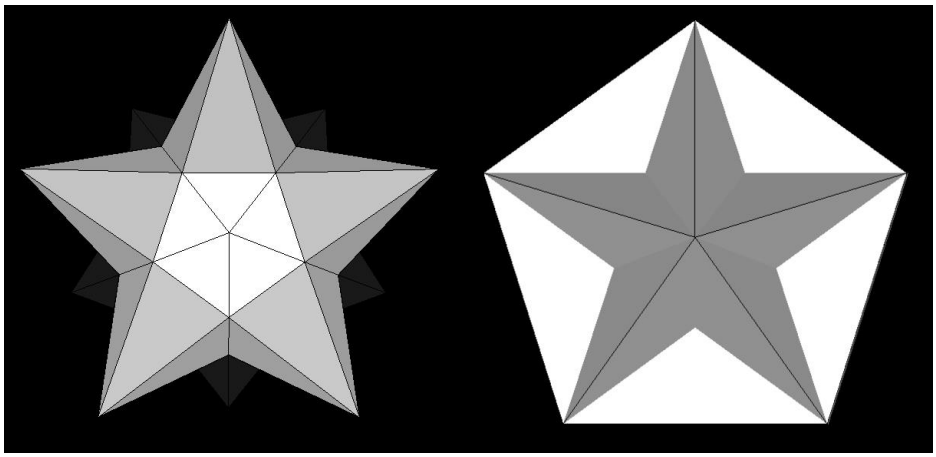


Рис. 4.9 Малий зірчатий додекаедр та великий додекаедр

Великий зірчатий додекаедр (рис. 4.10) має символ Шлефлі $\{5/2, 3\}$ та складається з 12 вершин і 30 ребер. Його гранями є 12 пентаграм які, перетинаючись, утворюють трикутні піраміди[21].

Цікавою особливістю цієї фігури є те, що, якщо зрізати всі трикутні піраміди, то ми отримаємо ікосаедр (рис. 4.11).

Всі зірчаті форми додекаедра є правильними зірчатими багатогранниками, які не можна представити у вигляді об'єднання однорідних багатогранників[10, с. 35].

Ікосаедр має 20 граней, продовжуючи які ми можемо отримати безліч правильних зірчатих тіл. У цій роботі ми розглянемо одне з них: *великий ікосаедр*[10, с. 41].

Великий ікосаедр (рис. 4.12) має символ Шлефлі $\{3,5/2\}$. Він складається з 12 вершин та 30 ребер. Його гранями є 20 рівносторонніх трикутників, що в перетині утворюють візерунок схожий на пентаграму[21].

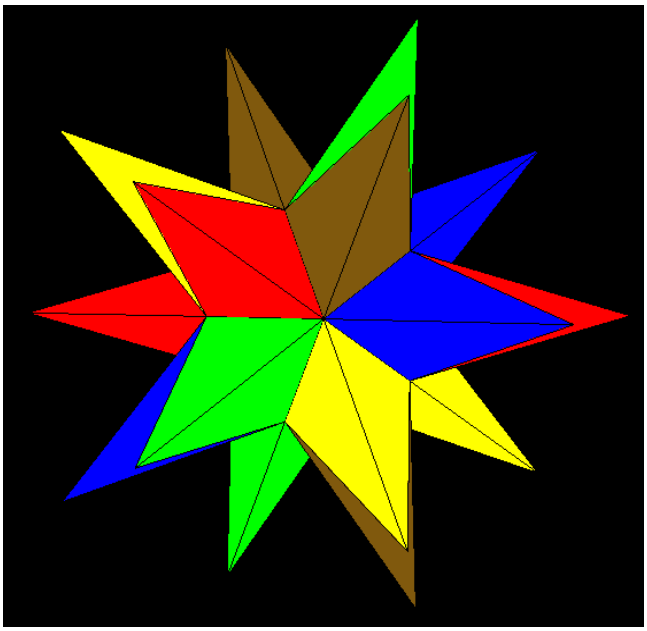


Рис. 4.10 Великий зірчатий додекаедр

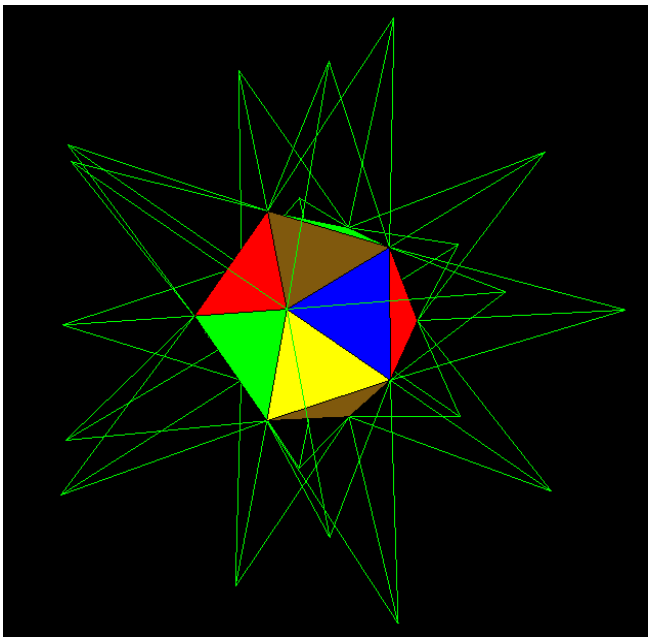


Рис. 4.11 Великий зірчатий додекаедр зі зрізаними трикутними пірамідами

Три зірчаті форми додекаедра та великий ікосаедр утворюють групу правильних зірчатих тіл під назвою *тіла Кеплера-Пуансо*, при цьому тіла Кеплера-Пуансо не можна представити як об'єднання Платонових або зірчатих тіл[10, с. 35].

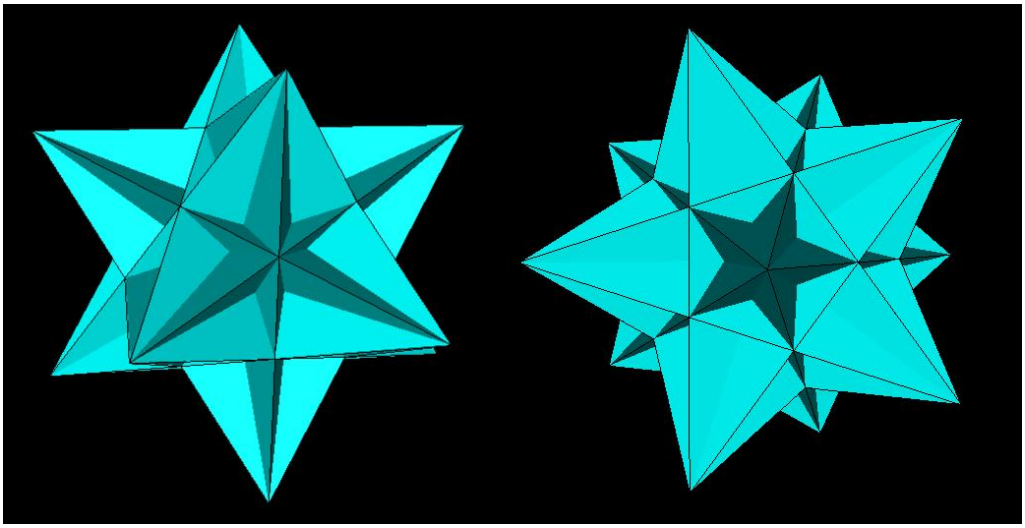


Рис. 4.12 Великий ікосаедр

Наостанок наведемо зірчату форму одного з Архімедових тіл.

Першою зірчатою формою кубооктаедра є *об'єднання куба та октаедра*[10, с. 68] (рис. 4.13).

Цей зірчатий багатогранник можна представити як об'єднання двох однорідних багатогранників: куба та октаедра.

Продовження трикутних граней кубооктаедра при перетині утворюють чотирикутні піраміди, а продовження квадратних – трикутні (рис. 4.14). У результаті отримане зірчате тіло схоже на об'єднання куба та октаедра.

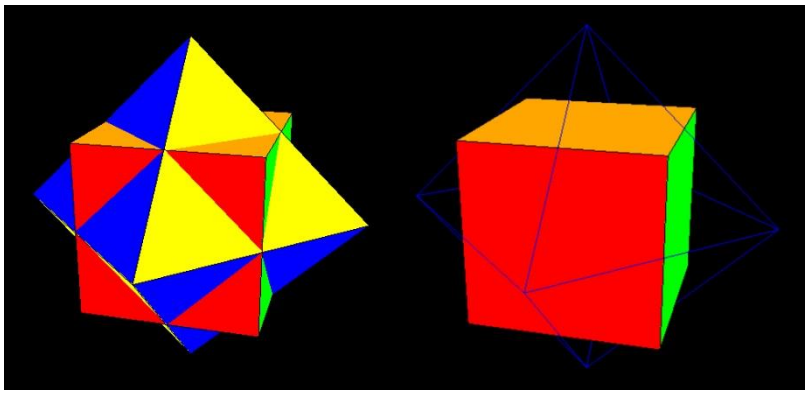


Рис. 4.13 Об'єднання куба та октаедра

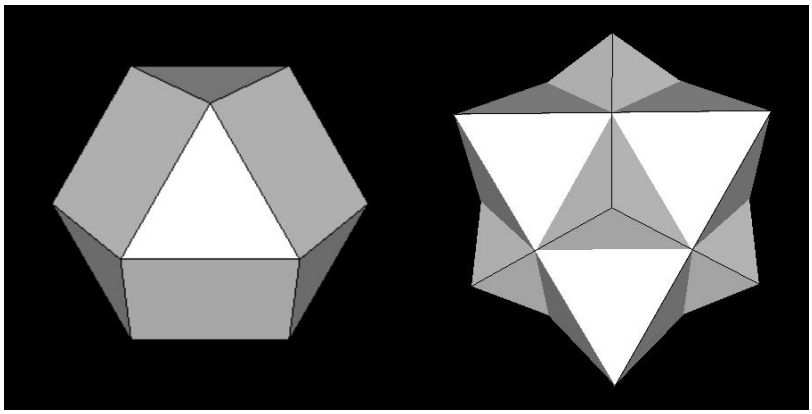


Рис. 4.14 Кубооктаедр та об’єднання куба й октаедра

ВИСНОВКИ

У даній роботі були розглянуті однорідні багатогранники (Платонові та Архімедові тіла) та однорідні зірчаті багатогранники (тіла Кеплера-Пуансо та деякі об’єднання однорідних багатогранників), їх влаштування та побудова.

У роботі розглянуто специфікацію програмного інтерфейсу OpenGL та її реалізацію на мові Haskell. Були розглянуті можливості OpenGL та принцип його роботи.

Використовуючи реалізацію OpenGL на мові Haskell, був розроблений графічний застосунок для перегляду тривимірних моделей Платонових та Архімедових тіл, деяких зірчатих однорідних багатогранників. Майбутнім покращенням цієї програми може слугувати збільшення кількості моделей зірчатих однорідних багатогранників.

Haskell – це не найпопулярніша мова, проте вона не стоїть на місці та постійно розвивається. Наразі для Haskell вже реалізовані деякі основні бібліотеки для роботи з OpenGL, такі як OpenGLRaw, GLUT та GLU[22]. Це дає гарне підґрунтя для подальшої розробки та розвитку графічних бібліотек на мові Haskell.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Unity. Documentation. OpenGL Core [Електронний ресурс] – https://docs.unity3d.com/Manual/OpenGLCoreDetails.html
2. Unreal Engine 4 Documentation. FOpenGLBase [Електронний ресурс] – https://docs.unrealengine.com/en-US/API/Runtime/OpenGLDrv/FOpenGLBase/index.html
3. OpenGL Wiki. Language binding [Електронний ресурс] – https://www.khronos.org/opengl/wiki/Language_bindings
4. OpenGL. Руководство по программированию. Библиотека программиста. 4-е издание / М. Ву, Т. Девис, Дж. Нейдер, Д. Шрайер – СПб.: Питер, 2006 – 23-210 с.
5. Learn OpenGL. OpenGL / Joey de Vries [Електронний ресурс] – https://learnopengl.com/Getting-started/OpenGL
6. Learn You a Haskell for Great Good! / Miran Lipovača – No Starch Press – с. 1
7. Will Kurt. Get Programming with Haskell / Will Kurt – Shelter Island, New York: Manning Publications Co., 2018 – 16-389 с.
8. HOpenGL – 3D Graphics with Haskell. A small Tutorial / Sven Eric Panitz – 2004 – 5-49 с.
9. Data.IORef [Електронний ресурс] – https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-IORef.html
10. Polyhedron Models / Magnus J. Wenninger – New York: Cambridge University Press – 1-68 с.
11. Polyhedra / Peter R. Cromwell – Cambridge: Cambridge University Press – 12-168 с.
12. Геометрия по Киселёву / А. П. Киселёв – 53-266 с.
13. Archimedean Solids / Anna Anderson – University of Nebraska-Lincoln – 2008 – 5-16 с.
14. Polyhedra Viewer / Nat Alison [Електронний ресурс] – https://polyhedra.tessera.li/
15. Graphics.UI.GLUT.Initialization [Електронний ресурс] – https://hackage.haskell.org/package/GLUT-2.7.0.16/docs/Graphics-UI-GLUT-Initialization.html
16. Learn OpenGL. Depth Testing / Joey de Vries [Електронний ресурс] – https://learnopengl.com/Advanced-OpenGL/Depth-testing
17. MathWorld--A Wolfram Web Resource. Spherical Coordinates / Eric Weisstein [Електронний ресурс] – https://mathworld.wolfram.com/SphericalCoordinates.html
18. Learn OpenGL. Coordinate Systems / Joey de Vries [Електронний ресурс] – https://learnopengl.com/Getting-started/Coordinate-Systems
19. Learn OpenGL. Camera / Joey de Vries [Електронний ресурс] – https://learnopengl.com/Getting-started/Camera

20. OpenGL Wiki. Calculating a Surface Normal [Електронний ресурс] – https://www.khronos.org/opengl/wiki/Calculating_a_Surface_Normal
21. Visual Polyhedra / David I. McCooey [Електронний ресурс] – http://dmccooey.com/polyhedra/
22. GitHub. Haskell OpenGL [Електронний ресурс] – https://github.com/haskell-opengl/