Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ПОБУДОВА ГЕОМЕТРИЧНИХ ФІГУР У ДОПОВНЕНІЙ РЕАЛЬНОСТІ ІЗ ЗАСТОСУВАННЯМ ФРЕЙМВОРКУ ARCORE

Текстова частина до курсової роботи за спеціальністю „Інженерія програмного забезпечення"

Керівник курсової роботи
к.т.н. ст. викл. Бучко О. А.

____
(підпис)
"____" __________ 2020 р.

Виконав студент Філоненко М. І.
"____" __________ 2020 р.

Київ 2020

Зміст

Анотація 5

Вступ 6

Аналіз та класифікація задач зі стереометрії. Створення універсального формату даних 9

Аналіз шкільної програми зі стереометрії 10-11 класів. Виділення основних типів задач 9

Створення моделі даних стереометричної задачі 10

Створення універсального формату даних стереометричних задач 14

Розробка інтерфейсу користувача 18

Побудова абстрактної схеми інтерфейсу 18

Реалізація графічного інтерфейсу користувача 20

Відображення геометричних моделей у доповненій реальності 24

Висновки 29

Список джерел 30

Додатки 31

XSD-схема розробленого формату даних.................................... 31

Опис моделі даних у мові програмування Kotlin із використанням анотацій SimpleXML 32

Приклади роботи програми 37

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

доцент, к.ф.-м.н.

____________ С. С. Гороховський

(підпис)

"____"_______________ 20__ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Філоненку Михайлу

3-го курсу факультету інформатики

ТЕМА: Побудова геометричних фігур у доповненій реальності із застосуванням фреймворку ARCore

Вихідні дані:

Зміст ТЧ до курсової роботи:

Вступ

Анотація

1. Аналіз та класифікація задач зі стереометрії. Створення універсального формату даних

2. Розробка інтерфейсу користувача

Висновки

Список джерел

Додатки

Дата видачі "___" _________ 201_ р.

Керівник _____________ Завдання отримано __________

Календарний план виконання курсової роботи

Тема: Побудова геометричних фігур у доповненій реальності із застосуванням фреймворку ARCore

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	листопад 2019 р.	
2.	Огляд літератури за темою роботи	листопад - грудень 2019 р.	
3.	Оволодіння навичками розробки вільних від блокувань структур даних	грудень - лютий 2019 р.	
4.	Створення практичної частини роботи	лютий - березень 2020 р.	
5.	Написання пояснювальної роботи	березень 2020 р.	
6.	Створення слайдів для доповіді та написання доповіді	березень 2020 р.	
7.	Надання роботи керівнику для перевірки	березень - квітень 2020 р.	
8.	Корегування роботи за результатами перевірки керівником	квітень 2020 р.	
9.	Остаточне оформлення пояснювальної роботи та слайдів	квітень-травень 2020 р.	
10.	Захист курсової роботи	15 травня 2020 р.	

Студент ____ Філоненко М.І. ____ ____________

Керівник ____ Бучко О.А. ____ ____________

“______” ____________________ _________ р.

Анотація

Стереометрія – одна з найскладніших частин шкільної математичної програми. При роботі з об'ємними побудовами учням доводиться додумувати конфігурацію геометричних фігур у просторі, що створює великі труднощі, особливо на початковому етапі. Ця робота присвячена способу подолання таких проблем за допомогою сучасних технологій, таких як просторове моделювання та використання технологій доповненої реальності.

Вступ

Актуальність теми. Сучасні технології значно полегшують наше життя. Вони автоматизують ті процеси, що раніше доводилося виконувати вручну, а також допомагають підвищити рівень кваліфікації у недосяжних раніше сферах. Поступово, рок за роком, новітні інформаційні технології починають використовуватися у сфері освіти та науки. В деяких дисциплінах їх застосування доволі очевидне, в інших - становить велику проблему з точки зору організації роботи та створення відповідної матеріально-технічної бази.

Однією з дисциплін, де наразі іде пошук застосування новітніх технологій, є шкільна математика. Раніше виконувалися деякі напрацювання у цьому напрямку: видавався методичний посібник із застосування інформаційних технологій у сільських школах [2], а також підтримка математичних уроків із застосуванням Paint та офісного пакету Windows [1]. Проте значного поширення подібні способи викладання не набули, зважаючи на такі проблеми:

1. **Складна та неоднорідна система запису завдань.** Математична нотація – одна з найбільш складних та неоднорідних з-поміж усіх, що можна зустріти у рамках шкільної програми. Можна погодитися з думкою, що запис математичних знаків набагато зручніший у паперовому, ніж в електронному вигляді.
2. **Високий поріг входження для більшості математичних програм.** Для роботи з програмами, присвяченими створенню формул, моделюванню чи кресленню необхідна значна кваліфікація, якою не володіє більшість учнів, і здобуття якої не є ціллю шкільної математичної програми.
3. **Низька ілюстративність відповідного ПЗ, його недостатня адаптованість під потреби шкільної програми**. Зазвичай математичне ПЗ створюється для професійних потреб, тому не враховує потреби

математичної програми, даючи велику кількість непотрібних можливостей і обмежуючи чи ускладнюючи потрібні для учнів речі.

4. **Незрозумілість вирішуваної проблеми**. Перенесення математичної нотації у програми загалом ускладнює її, майже не створюючи додаткових переваг. Всі учні з молодших класів володіють вмінням креслення і лише незначна частина може виконувати геометричні побудови на комп’ютері з такою ж швидкістю. Тобто, замість вирішення певних проблем, використання інформаційних технологій призводить до ускладнення навчального процесу.

Особливі проблеми у вивченні математики виникають серед учнів 10-11 класів, у зв'язку з ускладненням матеріалу. Одним з розділів, що вивчаються в цей час, є стереометрія. Учні мають значні проблеми із її вивченням, у зв'язку з ускладненням математичних побудов та слабким розвитком просторового мислення.

Оскільки значна кількість проблем у галузі освіти вже була вирішена за рахунок новітніх технологій, ми зробили припущення, що проблема із освоєнням стереометрії теж може бути вирішена таким самим чином. Ми взяли реально існуючу проблему, яку досить складно вирішити без застосування технологій, а саме, просторове мислення та сприйняття просторових побудов, і спробували розв’язати її.

Мета роботи: створення застосунку, що дозволить відображати, створювати, редагувати та маніпулювати стереометричними кресленнями у середовищі доповненої реальності.

Завдання:

1. Аналіз та обробка інформації, що стосується шкільної програми стереометрії 10-11 класів. Аналіз та виділення типів задач.
2. Знаходження способів структурування задач та побудови їх моделей на основі інформаційних технологій.

3. Створення універсального формату для зберігання та відтворення моделей, що буде зрозумілий для користувача, що має середню технічну кваліфікацію і буде зручним для обробки обчислювальною системою.
4. Побудова простого інтерфейсу, зрозумілим для користувача, що має технічну кваліфікацію на рівні 10-11 класів, що дозволяє взаємодіяти з моделями у відповідному форматі (вводити нові та змінювати існуючі дані).
5. Вивчення технологій доповненої реальності та побудова необхідних для роботи програми абстракцій.
6. Створення інтерпретатора, що дозволить відобразити модель у просторі за допомогою технологій доповненої реальності.

Об'єкт дослідження: матеріал шкільного курсу геометрії 10-11 класів, бібліотека доповненої реальності ARCore.

Предмет: додаток для відображення стереометричних побудов у середовищі доповненої реальності.

Наукова новизна: дана робота є першою зі спроб адаптації технологій доповненої реальності для використання у цілях навчання стереометрії із застосуванням зрозумілого та спрямованого на середню кваліфікацію учня інтерфейсу користувача.

Практичне значення роботи полягає у можливості застосування розробленого додатка учнями для покращення знань зі стереометрії та кращого володіння навичками з вирішення стереометричних задач.

Структура курсової роботи. Курсова робота складається з двох розділів та підрозділів, списку використаних джерел (9 найменувань), 3 додатків.

Аналіз та класифікація задач зі стереометрії. Створення універсального формату даних

Аналіз шкільної програми зі стереометрії 10-11 класів. Виділення основних типів задач

Перед початком роботи над програмою потрібно чітко зрозуміти обсяг завдання, вимоги до кінцевого продукту та очікування користувачів. Достатньо очевидно, що, створюючи програму для подання геометричних моделей у доповненій реальності, ми маємо зрозуміти, які умови можуть бути задані в цих моделях, та, враховуючи ці вимоги, створити застосунок, що буде надавати можливість задавати всі, або більшість з цих умов.

Спосіб аналізу, що був обраний – перегляд матеріалів шкільної програми (підручників, посібників та збірників задач).

Для проведення аналізу були використані рекомендовані МОН України підручники, а також деякі класичні збірники задач [3; 4].

Завдяки проведеному аналізу були виокремлені такі типи задач:

5. **Задачі з площинами та прямими.** Тут задаються відношення площин, кути та відстані між ними, конфігурація площин та прямих у просторі. У якості шуканого значення виступає кут або відстань між елементами, або загальне питання (скільки існує елементів, що відповідають певній умові).
6. **Задачі з пірамідами.** У таких задачах комбінуються елементи піраміди, площини, відрізки та прямі. У якості шуканого значення може бути деякий кут між елементами, сторона або інший елемент піраміди, довжина деякого відрізку.
7. **Задачі з призмами.** Крім попередніх типів комбінацій елементів, у таких задачах також додаються перерізи фігур та робота з ними, знаходження деяких величин за допомогою перерізів.

8. **Задачі з сферами, циліндрами та конусами.** Зазвичай у них використовуються попередні математичні знання, що були застосовані при роботі з пірамідами та призмами, а також додаються формули для роботи зі сферами та колами (площа, довжина кола, площа сектора або сегменту).
9. **Задачі, що суміщують декілька типів, зазначених вище.** Між геометричними фігурами задаються деякі відношення, що дозволяють або встановити значення у одній з фігур, або ж шуканим значенням виступають саме ці відношення.

Матеріали та задачі, що містяться у підручниках, можуть також не належати до жодної з цих категорій та оперувати деякими незвичними просторовими об’єктами. Такі задачі, а також задачі олімпіадного рівня, що не належать до шкільної програми, у рамках цієї роботи розглядатися не будуть.

Ми виявили, що всі задачі зі стереометрії, що були дані у рамках ЗНО, належать до того або іншого типу даної класифікації. Отже, ми можемо використати дану класифікацію для виконання наступного кроку – побудови моделі даних на основі інформаційних технологій.

Створення моделі даних стереометричної задачі

Для того, щоб створити необхідний застосунок, мало виділити типи задач. Щоб надати зрозумілий користувачу інтерфейс, захистити його від ненавмисного некоректного вводу даних, а також створити зрозумілий для комп'ютера механізм відображення деякої задачі, необхідно створити додатковий шар абстракції, а саме типізовану модель.

Завдяки класифікації, яку ми виконали у попередньому розділі нашої роботи, ми можемо зрозуміти, які типи даних є спільними для усіх типів

задач, які параметри та обмеження застосовуються будь-де та визначити їх типами у нашій програмі.

Серед таких типів є:

10. **Просторові форми (Shape)**. Загальний тип, що включає піраміди, призми, сфери, конуси та площини. Незважаючи на те, що всі ці фігури мають різні визначення, вони є спільними за основним параметром – вони є просторовими, тобто при їх побудові мають значення всі три виміри. Також всі ці фігури мають спільні риси. Всі вони мають основу, що є пласкою фігурою. Вони можуть мати ім'я, їх кількість в задачі в цілому необмежена (зазвичай 1 або 2). Ці форми мають вершини, можуть мати обмеження по кутах або сторонах. Всі вони можуть бути правильними або неправильними. Такі форми є верхнім рівнем моделі, тому що включать в себе всі інші елементи побудови.
11. **Основа (Base).** Ця сутність зазвичай визначає планіметричну частину стереометричної задачі. Основа може включати багатокутники (Polygon), відрізки та прямі (Line) та еліпси (Ellipse). Вона може також накладати обмеження (Constraint), що відносяться до зв'язку між цими фігурами.
12. **Еліпс (Ellipse).** Така фігура принципово відрізняється від багатокутників як способом побудови, так і умовами задач, що з нею пов'язані. Еліпс може бути правильним або довільним, а також мати інші обмеження (наприклад, на розмір та перетини з іншими фігурами).
13. **Відрізок (Line).** Найпростіша з усіх фігур, за допомогою якої можуть бути побудовані інші фігури, такі як багатокутники та просторові форми, а також елементи фігур (висоти, медіани, бісектриси, діагоналі тощо). Може мати ім'я, що зазвичай задається малими літерами. Окрім довжини, відрізок не може мати ніяких обмежень відносно себе, лише обмеження, задані фігурами, до яких він включений.

14. **Багатокутник (Polygon).** Фігура, що входить до більшості стереометричних задач. Може бути правильним, мати обмеження на кути, сторони, внутрішні елементи. Також є специфічні типи багатокутників, що часто зустрічаються у задачах, такі як паралелограм, ромб, рівнобедрений чи прямокутний трикутник, трапеція тощо.

Також э елементи, що, хоча і присутні на побудові, не мають самостійного геометричного змісту. Вони завжди включаються у інші об'єкти. Серед них:

7. **Сторона (Edge).** Задає обмеження або детальну інформацію по стороні деякої фігури. Може мати довжину, або бути необхідною для задання кута.
8. **Група вершин (VertexGroup).** Часто кожна вершина не має самостійного значення, а в задачі просто зазначається деяка сукупність точок. У групи є тип (вершини основи, вершина піраміди).
9. **Кут (Angle).** Кут – досить складна побудова. Він може бути просторовим або пласким, мати значення, бути кутом між прямими або площинами. Кут має посилатися на те, якими об'єктами він заданий, а також через що проходить (точка або пряма).
10. **Обмеження (Constraint).** Тут подаються всі обмеження, що не мають відображення на рисунку задачі, але впливають на відношення об'єктів у ній. Наприклад, це може бути відстань між площинами, відношення фігур як вписаної та описаної тощо. Обмеження також можуть задаватися як кроки до вирішення певної задачі, або для моделювання ситуацій, які можуть виникнути у задачі.

Завдяки тому, що ми виділили набір сутностей і в загальному вигляді надали характеристику кожної з них, ми можемо тепер перейти до проектування структур даних.

При проектуванні ми можемо помітити декілька закономірностей:

1. Деякі об’єкти не мають сенсу без інших. Наприклад, основа форми не має сенсу без самої форми, кут не має сенсу без елементів, що його утворюють тощо.
2. Деякі об'єкти мають відображатися на рисунку (фігури, форми, кути), деякі лише впливають на рисунок, але не відображаються (обмеження).
3. Деякі об'єкти у своєму визначенні посилаються на інші об'єкти, що їх утворюють (кути на елементи, що їх утворюють, обмеження на елементи, які до них відносяться).

Ієрархічну природу стереометричної побудови (закономірність 1) зручно відобразити за допомогою деревоподібної структури даних, коли деякі елементи включаються до складу інших. Строга типізація дає нам можливість обмежити також вкладання елементів. Наприклад, до плаской фігури не може бути включена об'ємна, що ми показуємо вже на рівні моделі.

Для того, щоб розмежувати два типи об'єктів, визначених за закономірністю 2, ми можемо створити два загальні інтерфейси. Один з них (GeometryObject) буде визначати всі елементи, що відображаються на рисунку. Другий (Affector) – ті, що тільки впливають на відображення. При цьому один і той самий тип може визначати обидва ці інтерфейси, якщо він одночасно і впливає на побудову, і може бути показаний на рисунку.

Для того, щоб задовольнити потребу в ідентифікації об'єктів, ми створимо закриту змінну id, що буде визначена для кожного GeometryObject. Ця змінна недоступна користувачу, вона унікально ідентифікує кожен об'єкт (на відміну від імені об'єкта, що задається користувачем).

Композиція об'єктів та діаграма їх властивостей, що були визначені на попередньому кроці, показана на ілюстрації.

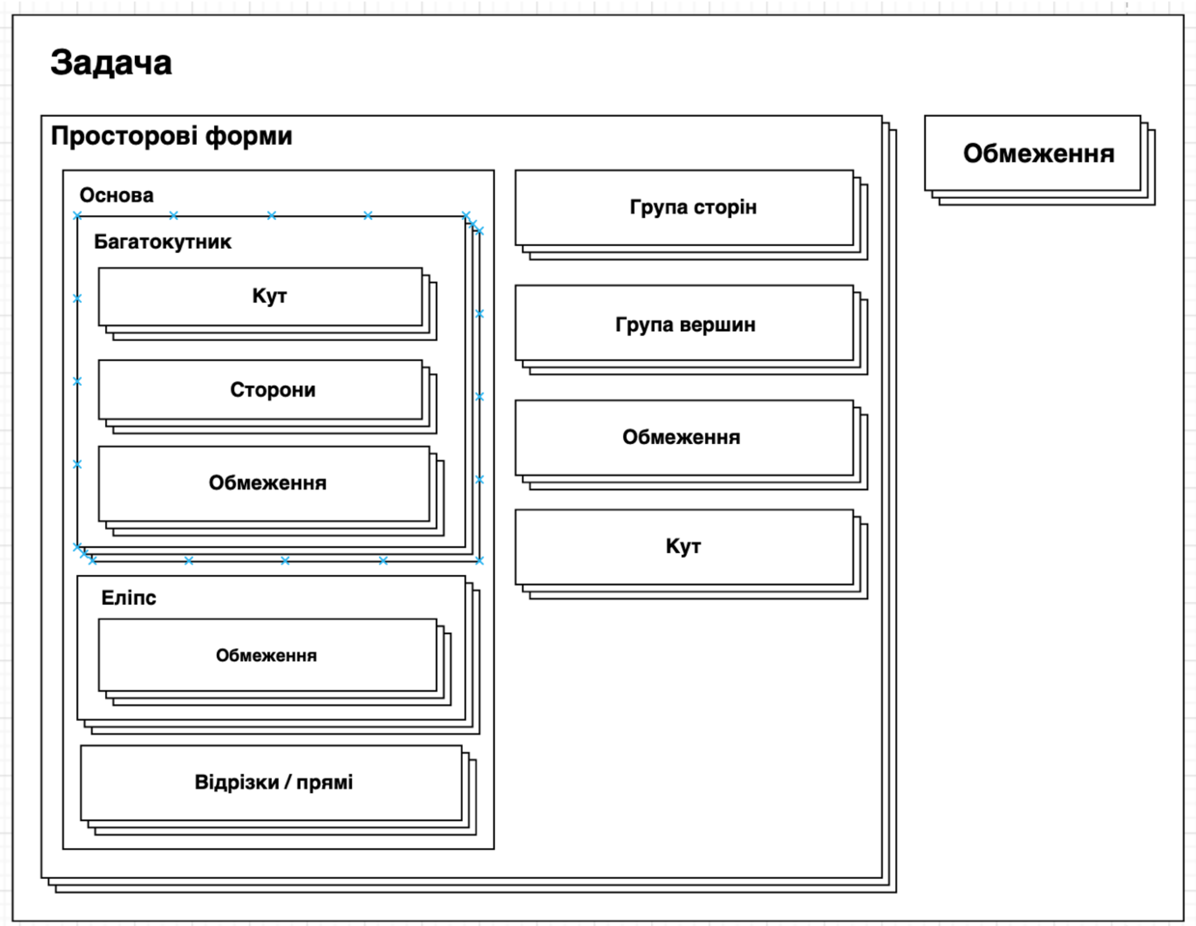


Рисунок 1. Схема композиції геометричних сутностей.

Створення універсального формату даних стереометричних задач

Першим етапом створення формату даних є вибір моделі кодування. Як ми бачимо, модель даних, яка буде побудована, має бути ієрархічною, при цьому у кожного об’єкту існують як властивості, такі як розмір або назва, так і вкладені об'єкти (сторони, кути, вершини тощо).

Таку модель даних добре передає стандарт XML. Нехай атрибутами буде визначено властивості кожного об'єкту, а вкладеними тегами – ієрархічний зв'язок між ними.

Формат XML широко розповсюджений у світі інформаційних технологій та застосовується для подання великої кількості даних. Він

органічно показує вкладеність об'єктів та множинну вкладеність, дає можливість задання атрибутів та відділення їх від вкладених сутностей. Не менш важливим є те, що для обробки цього формату існують бібліотеки для кожної розповсюдженої мови програмування, що дозволяють як передати довільну інформацію у вигляді XML (серіалізувати її), так і виконати десеріалізацію, тобто перетворити структуру XML у сукупність об'єктів такої мови.

Через свою розповсюдженість та зрозумілість як для обчислювальної системи, так і для людини, ми обрали такий формат подання нашої моделі даних.

Наприклад, так буде описана одна з перших задач з курсу стереометрії.

```
<exercise task="Намалюйте площину a та пряму b, що їй  належить">
    <scene>
        <shapes>
            <shape>
                <base name="α">
                    <line name="b">
                    </base>
            </shape>
        </shapes>
    </scene>
</exercise>
```

Тут ми можемо побачити ієрархічну структуру нашого формату даних. Кореневим тегом виступає exercise, де задається текст завдання. Далі надається список тегів scene – наразі ми вважаємо, що геометрична побудова одна, але в деяких задачах буває корисно виконати декілька

побудов, розміщуючи фігури під різними кутами, тому залишимо тут можливості для розширення моделі. У масиві shapes знаходяться власне просторові форми, серед яких і shape без типу, що означає пусту форму. У ній є тег base, що відповідає за основу, і, оскільки тип основи не вказано, ми вважаємо, що це площина. На площині є пряма b, що органічно включається в неї.

Це досить тривіальний приклад задачі. Теги вкладаються один в одного, створюючи ілюзію надлишковості. Давайте розглянемо складніший приклад, де стане очевидно, навіщо потрібен такий рівень вкладеності.

```
<exercise task="Плаский кут при вершині правильної трикутної піраміди дорівнює 90 градусів. Знайти відношення бічної поверхні піраміди до площі її основи">
    <scene>
        <shapes>
            <shape type="pyramid" regular="true">
                <base>
                    <polygon type="triangle"/>
                </base>
                <vertices>
                    <group id="v0" type="low" csv="A,B,C"/>
                    <group type="high" csv="S"/>
                </vertices>
                <angles>
                    <angle from="v0" to="v0" through="S"/>
                </angles>
            </shape>
        </shapes>
    </scene>
</exercise>
```

Тут вже можна побачити силу декларативного представлення у такому форматі. Незважаючи на досить складну умову задачі, ми описуємо лише її зміст, не створюючи зайвих сутностей, що має надавати користувач. Наприклад, ми не задаємо розмірність піраміди, а лише її форму та те, що вона є правильною. Ми не згадуємо і про те, що трикутник у її основі правильний – адже це зайве дублювання інформації. Подивіться, наскільки добре зрозуміле задання кутів. Для цього ми використали лише дві змінні-ідентифікатори, показавши, що кут при вершині – це кут, що задається двома довільними вершинами основи піраміди. Тут стає зрозуміло, навіщо виділяти і окремі сутності груп вершин, адже це дозволяє не специфікувати, між якими саме вершинами цей кут, залишаючись на тому ж рівні абстракції, що наданий умовою задачі у вільному вигляді.

Однією з рис XML є його строгість та можливість задати правила, за якими можна визначити коректність даних, тобто специфікувати свій формат даних на рівні мови. Специфікація XML, що бла розроблена у рамках даної роботи, надається в додатку.

Розробка інтерфейсу користувача

Побудова абстрактної схеми інтерфейсу

Якими б досконалими не були алгоритми, закладені в інформаційну систему, саме інтерфейс користувача задає бачення її якості для користувача. Однією з рис зручного інтерфейсу є його відповідність певній парадигмі, тому, що користувач вже бачив в інших програмах, щоб його досвід використання був органічно перенесений на новий додаток.

Однією з перших частин проектування інтерфейсу є визначення цільової платформи. В даному випадку платформа вже задана умовою задачі: оскільки для роботи програми нам необхідна робота з камерою, найкращим варіантом для нас будуть пристрої з найбільш просунутими камерами споживацького класу на ринку – смартфони.

Смартфони є найбільш вживаними пристроями на планеті. Незважаючи на те, що за їх допомогою виконуються найрізноманітніші операції, інтерфейси мільйонів створених програм для них вкладаються у загальний шаблон. Так, основним елементом інтерфейсу є список елементів. В нашому випадку, очевидно, має бути показаний список сутностей, що були виділені та формалізовані на попередніх кроках.

З інтерфейсом смартфонів існує певна проблема. Досі не сформульована чітка концепція, за якою мають відображатися ієрархічно згруповані об'єкти. Для цього не існує ні офіційних пояснень, ні реальних робочих прикладів. В нашому випадку ієрархія потрібна, що було показано на попередніх кроках. Тому ми вдамося до прийому лінеаризації, показавши наше ієрархічне дерево як список, звичний для користувача, проте кожен елемент буде мати відступ, що і буде вказувати на вкладеність.

Така структура, при низькому рівні вкладеності, загалом нагадує користувача ієрархічне дерево коментарів, тобто вкладається в деякий шаблон, напрацьований іншими додатками.

Іншим шаблоном сприйняття є клік для перегляду деталей. Тут ми втілюємо варіант, вже напрацьований в багатьох інших додатках, коли клік на елемент не показує окреме вікно, а лише відкриває додаткові опції роботи з існуючим вікном. Користувачу потрібно здійснювати менше кліків для переходу між вікнами і він може легко додавати та видаляти елементи, при цьому економиться місце на екрані.

Рисунок 2. Схема розташування елементів інтерфейсу. Елементи в згорнутому та розгорнутому стані.

Таким чином ми вирішили найголовнішу проблему інтерфейсу – взаємодію з ієрархією геометричних сутностей. Редагування, видалення та додавання логічно реалізувати через модальні вікна, а список задач – через стандартний список. Інтерфейс взаємодії з камерою також стандартизований і не потребує пояснень.

Реалізація графічного інтерфейсу користувача

Як від абстрактного формату даних ми перейшли до конкретного універсального формату даних XML у попередньому вище, тепер ми маємо перейти від абстрактного опису інтерфейсу до його конкретної реалізації. Загалом, у сфері мобільних додатків є три варіанти: iOS, Android та платформи універсальної розробки під обидві операційні системи. У нашому випадку останній варіант не підходить, адже кожна з платформ має свою унікальну бібліотеку для підтримки роботи з доповненої реальністю. Враховуючи популярність Android та власні матеріально-технічні умови, ми обрали розробку під Android, тому у даному розділі піде мова про реалізацію інтерфейсу саме під цю платформу. Мовою програмування для розробки обрано Kotlin як рекомендовану мову для Android.

Структура даних була створена як відображення формату даних XML, описаного вище. Для органічної серіалізації та десеріалізації була обрана бібліотека SimpleXML, що для ідентифікації даних використовує сукупність анотацій. Так, наприклад, буде виглядати клас Ellipse.

```
@Root(name="ellipse", strict = false)
public data class Ellipse(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:Attribute(required=false, name="regular")
    @param:Attribute(required=false, name="regular")
    var regular: Boolean? = false,

    @field:ElementList(required=false, name="constraints")
    @param:ElementList(required=false, name="constraints")
```

```
    var constraints: List<Constraint>? = null
)
```

Анотація @Rott тут позначає ім'я елемента, який описується. Також ми вставлюємо режим strict = false, що означає допустимість пустих тегів та атрибутів XML, що не знайдуть свого відображення у моделі. Парами атрибутів field та param встановлюємо правила десеріалізації та серіалізації відповідно. Також зверніть увагу на те, що кожному атрибуту ми встановлюємо значення за замовчанням.

Повний опис моделі даних подається у додатку.

Модель даних має якось взаємодіяти з інтерфейсом. Ми не маємо однозначної відповіді на питання, які саме елементи можна додавати, які редагувати і які видаляти. Очевидно, ці правила гри має встановлювати сама модель. Саме тому у класі GeometryObject, від якого наслідуються усі геометричні сутності, встановлені декілька методів для цього.

```
interface GeometryObject {
    public fun subElements(): List<GeometryObject>;
    public fun canEdit(): List<InputVariant> {
        return emptyList()
    }
    public fun edit(values: List<InputVariant>) {


}
    public fun canAdd(): List<GeometryObjectType>
    {
        return emptyList()
    }
    public fun add(type: GeometryObjectType) {


    }
```

```
    public fun canBeDeleted(): Boolean {
        return true
    }

    public fun info(): Map<String, String> {
        return mapOf()
    }
}
```

Тут стає зрозуміло, що модель керує змістом інтерфейсу, він не заданий статично, а динамічно, за правилами моделі, генерується під час виконання. Так, модальне інформаційне вікно, що показує деталі деякої сутності, наповнюється з методу info, також я пари методів canAdd - add та canEdit -> edit. Перший метод з такої пари дає інформацію, ключі, за якими можна звертатися до об'єкту з проханням додати чи редагувати об'єкт. Поле редагування може бути декількох типів: текстове поле, список, перемикач. Також список має пропонувати декілька типів сутностей. Така структура добре вкладається у ООП-модель і реалізована через спільний інтерфейс InputType та його реалізації.

```
interface InputType {  }
class IntInputType :InputType { }
class OptionInputType(public val options: List<String>): InputType {}
```

Ще одне питання полягає в тому, як повідомляти окремим незалежним частинам інтерфейсу про те, що щось було змінено і потребує перебудови. Для цього було застосоване делегування. При створенні кожен Activity додає себе до списку делегатів сінглтон-класу, до якого за необхідності звертаються методи моделі. Сінглтон-клас відсилає

повідомлення по групах класів, чи потрібна повна перебудова моделі, редагування окремого елемента або його видалення.

Не менш важливим є метод subElements. Він дозволяє нам побудувати дерево геометричних сутностей, з якими взаємодіє користувач. Кожен клас сам показує, які його елементи слід відображати шляхом реалізації цього методу, а далі в дію вступає функція лінеаризації такого дерева.

```
private fun allElements(obj: LGO) : List<LGO> {
  return listOf(obj) + obj.second?.subElements().orEmpty().map
{ allElements(LGO(obj.first + 1, it)) }.flatten()
}
```

Тут ми бачимо також елементи функціонального стилю.

Давайте подивимося, як формуються модальні вікна в нашій програмі. Як було показано, модель надає дані, а відповідний клас створює елементи вводу користувача (текстові поля, перемикачі, списки вибору) за цими даними. Для того, щоб відслідковувати дії користувача, ми додаємо Listener до кожного з таких елементів вводу та при зміні даних змінюємо і даний нам методом canEdit масив ключів та значень. У разі підтвердження дії користувачем, той самий масив тепер передається у edit, де у дію знову ж вступає клас моделі, що аналізує ввід та редагує відповідні елементи.

У випадку додавання, ми маємо створювати нову сутність, що не так просто зробити, оскільки кожен підклас GeometryObject має декілька параметрів. Саме тому для нас настільки важлива наявність значень за замовчуванням для кожного поля. Знову ж, сам підклас визначає, якого значення він набуде при створенні конструктором без параметрів.

Наостанок опишемо протокол Affector, що застосовується для обмежень. Обмеження враховуються одне за одним, ланцюжком, саме таким чином і працює цей інтерфейс.

```
interface Affecter {
```

```
fun affect(objects: List<GeometryObject>, initial: List<RenderOptions>) : List<RenderOptions>;
}
```

Кожне обмеження – це один крок у ланцюжку.

Таким чином, інтерфейс нашого додатку повністю визначає модель. Всі форми та дерево сутностей формуються відповідно до побажань моделі, що додає інтерфейсу гнучкості та зменшує дублювання коду.

Відображення геометричних моделей у доповненій реальності

Отже, ми сформулювали ідеї інтерфейсу та моделі даних для нашої програми, тепер перейдемо до реалізації основної частини – відображення моделей у доповненій реальності. Тут ми маємо декілька труднощів.

Неповнота даних. Наша модель задана декларативно, у нас немає точних даних відносно розташування тих чи інших фігур. При тому, що це є перевагою нашого підходу для роботи з кінцевим користувачем, це також є проблемою при роботі обчислювальної системи.

Обмеження можуть бути зовнішніми і внутрішніми. Зовнішні обмеження (кути, відстані)– зрозуміла для нас річ. Ми можемо застосувати їх ланцюжком. З іншої сторони, піделементи фігури можуть задавати батьківські елементи. Наприклад, конфігурація точок трикутника в основі піраміди задає координати відрізків, що утворюють сторони піраміди. Такі залежності не є тривіальними, і, в загальному випадку, можуть бути кільцевими. Тоді для того, щоб виконати правильну побудову, ми фактично маємо розв'язати задачу і знайти всі значення. Оскільки створення такої системи штучного інтелекту не входить до обсягу нашої роботи, скористаємося спрощеним механізмом, а саме послідовним відображенням піделементів із поверненням значень батьківському елементу, що він застосує для власного відображення.

Так чи інакше, для того, щоб відобразити деякі геометричні сутності, нам необхідно скористатися тими примітивами, що надає стандартна бібліотека. Клас ShapeFactory, що був використаний у роботі, надає такі примітиви:

15. Паралелепіпед
16. Циліндр
17. Сфера

Також окремо виступають текстові вузли, що відображають пласкі фігури у просторі.

За допомогою цих примітивів а також перетворень, таких як паралельне перенесення та обертання, можна створити будь-яку геометричну побудову.

Загалом у рамках бібліотеки ARCore ключовою одиницею є TransformableNode. Це вузол побудови, що посилається на свого батька та має дочірні вузли. Така структура допомагає абстрагуватися від абсолютної системи координат та позиціонувати кожен підвузол у залежності від координат та параметрів повороту батьківського вузла, для цього існують властивості localPosition та worldPosition. Вузол посилається на Renderable – власне ту геометричну форму, яку ми хочемо відобразити. Вузол верхнього рівня посилається на екземпляр бібліотечного класу ArFragment.

Також у бібліотеці є генератор текстур – MaterialFactory. Він дозволяє надати об'єктам колір чи текстуру, у нашому випадку – напівпрозору та одного з вибраних кольорів. Деякі методи працюють асинхронно, тому для отримання результату на повернення параметрів (внутрішніх обмежень) на рівень вище, нам знадобиться клас CompletableFuture, що дозволяє встановити зворотній виклик (callback) при виконанні асинхронної частини коду. Також дуже корисною є можливість поєднувати декілька CompletableFuture в один за допомогою методу CompletableFuture.allOf.

Для того, щоб ефективно виконувати побудови, необхідно спочатку створити зручні для нас примітиви, тобто новий шар абстракції над вже існуючим. Так, в процесі виконання були створені такі примітиви, як текстовий блок, пряма, що задається через точки, трикутник тощо.

Текстові вузли в нашому випадку не можуть бути просто пласким текстом у просторі. Для того, щоб користувач коректно бачив креслення з усіх сторін, вони мають повертатися відносно камери користувача. Для цього кожного разу при повороті камери ми маємо корегувати їх параметр worldRotation, враховуючи поточні параметри камери. В роботі були використані деякі стереометричні обрахунки:

```
fragment.arSceneView.scene.addOnUpdateListener { frameTime ->
    val cameraPosition = fragment.arSceneView.scene.getCamera().getWorldPosition()
    val cardPosition = textNode.getWorldPosition()
    val direction = Vector3.subtract(cameraPosition, cardPosition)
    val lookRotation = Quaternion.lookRotation(direction, Vector3.up())
    textNode.setWorldRotation(lookRotation)
}
```

Тут ми зустрічаємось з таким ключовим поняттям побудови, як кватерніон. Цей параметр задає поворот об'єкту і широко застосується у різноманітних системах моделювання. Кватерніон складається з чотирьох параметрів: просторовий вектор (x, y, z) та ступінь повороту w. У прикладі, наведеному вище, кожного разу при повороті камери ми беремо її позицію, обраховуємо різницю між існуючим поворотом та поворотом камери, та встановлюємо цю різницю у якості необхідного нам повороту текстового блоку.

Такою ж абстракцією можна задати не тільки поворот текстового блоку, але і відображення відрізків. В основу ми беремо паралелепіпед який необхідним чином переносимо та повертаємо у просторі.

```
fun renderLine(from: Vector3, to: Vector3, parent: Node, fragment: ArFragment, context: Context) {
    var node = TransformableNode((fragment.transformationSystem))
    node.setParent(parent)
    val diff = Vector3.subtract(from, to)
    val diffNormalized = diff.normalized()
    val rotation = Quaternion.lookRotation(diffNormalized, Vector3.up())
    node.worldRotation = rotation
    node.worldPosition = (Vector3.add(from, to).scaled(.5f)
…
}
```

Після того, як ми абстрагувалися від досить незручних для нашої задачі примітивів, можемо виконувати побудови, обраховуючи координати точок об'єктів. Дерево TransformableNode прекрасно вкладається у нашу ієрархічну модель геометричних сутностей. Ми рекурсивно відображаємо поточний елемент, додаємо його вузол до батьківського, а потім викликаємо таку саму операцію для піделементів.

Для того, щоб правильно застосовувати обмеження, нам необхідно довільно міняти порядок побудови піделементів. Наприклад, для головного вузла порядок не має значення і ми довільно відображаємо всі елементи. Для піраміди доречно відобразити спочатку її основу, забрати у неї параметри і застосувати їх для побудови решти відрізків.

Робота з відображення кожного елементу загалом спирається на наші математичні пізнання з шкільного курсу. Для ілюстрації такого підходу наведемо процес знаходження точок багатокутника на площині.

```
val angles by lazy {
    if(!(regular ?: false)) {
```

```
            val values = (0..sides!! - 1).map { Random.nextDouble(it + 0.2, it + 0.8) }
            val sum = values.sum()
            values.map { it / sum * 4 * PI}.sorted()
        }
        else {
            (0..sides!! - 1).map { it * angleIncrement }
        }
    }
    val a: Float = (Math.sqrt(2 * d * d - d * d * Math.sqrt(3.0))).toFloat()
    val points = angles.map { options.position.translate((a * cos(it).toFloat()), 0f, (a * sin(it).toFloat())) }
```

Тут можна побачити, що ми оперуємо лише двома координатами, залишаючи третю завжди нульовою. Але це не значить, що такий многокутник буде відображатися в одній площині. Він буде будуватися відносно батьківського вузла, що може бути поверненим. Такий підхід є ще однією ілюстрацією того, наскільки важливі для відображення складного дерева елементів поняття відносної позиції та оберту.

Приклади схем та їх відображення можна знайти у додатку.

Висновки

У рамках даної роботи було виконано декілька важливих завдань. Був проаналізований матеріал курсу стереометрії 10-11 класів та структуровані типи задач. Задачі були структуровані за сутностями, були знайдені загальні риси сутностей, за рахунок чого вдалося створити абстрактну модель даних, що покриває потреби курсу. Використавши формат XML як основу, ми розробили власний формат даних, що ґрунтується на ієрархічній моделі відносин об'єктів.

У другому розділі ми показали втілення нашого абстрактного проекту та створення робочого застосунку. Ми спроектували інтерфейс, що є інтуїтивно зрозумілим та відповідає сучасним стандартам. Основною частиною роботи стала реалізація побудови геометричних фігур у доповненій реальності, для чого був створений новий шар абстракцій, а на його основі – відтворені основні стереометричні фігури та їх відношення. Були продумані типи обмежень та можливості для коректного відображення моделі із урахуванням цих обмежень.

Були показані загальні риси програмування та моделювання у доповненій реальності, описана робота з основними абстракціями бібліотеки, принципи серілазіації та десеріалізації даних.

У рамках виконання роботи виникли деякі труднощі. Вони пов'язані із існуванням циклічних залежностей, коли для отримання коректних даних фактично потрібно спершу розв'язати задачу. Для того, щоб просунутися у цьому напрямку і коректно виконувати складні геометричні побудови, необхідно додатково розробити систему розв'язання задач у рамках заданого формату.

Таким чином, ми виконали основні завдання, поставлені на початку роботи, а також довели можливість застосування інформаційних технологій у курсі стереометрії для кращого освоєння матеріалу учнями.

Список джерел

1. Використання та впровадження ІКТ на уроках інформатики – Кадієвич О.С, 2018.
2. Інформаційні технології на уроках інформатики. ІКТ-супровід – Руденко В.Об 2009.
3. Математика. Алгебра і початки аналізу та геометрія. Рівень стандарту – А. Г. Мерзляк, 2018.
4. Математика. Алгебра і початки аналізу та геометрія. Рівень стандарту – О. С. Істерб 2018.
5. https://developers.google.com/ar/discover/concepts - ARCore Fundamental Concepts.
6. https://blog.griddynamics.com/latest-arcore-and-sceneform-features-take-creation-of-ar-apps-to-the-next-level/ - Take your apps to the next level with 3D augmented reality.
7. https://creativetech.blog/home/ui-elements-for-arcore-renderable – How to attach UI elements to ARCore Node.
8. https://proandroiddev.com/arcore-cupcakes-4-understanding-quaternion-rotations-f90703f3966e – ARCore Cupcakes #4 - Understanding Quaternion Rotations.
9. https://citrusbits.com/how-to-create-arcore-app-using-kotlin/ – How to create ARCore App Using Kotlin.

Додатки

XSD-схема розробленого формату даних.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema"> <xs:element name="exercise"> <xs:complexType> <xs:sequence> <xs:element name="scene"> <xs:complexType> <xs:sequence> <xs:element name="shapes"> <xs:complexType> <xs:sequence> <xs:element name="shape"> <xs:complexType> <xs:sequence> <xs:element name="base"> <xs:complexType> <xs:sequence> <xs:element name="polygons"> <xs:complexType> <xs:sequence> <xs:element name="polygon"> <xs:complexType> <xs:sequence> <xs:element name="edges"> <xs:complexType> <xs:sequence> <xs:element name="edge-group"> <xs:complexType> <xs:sequence> <xs:element type="xs:string" name="edge" maxOccurs="unbounded" minOccurs="0"/> </xs:sequence> <xs:attribute type="xs:string" name="id"/> <xs:attribute type="xs:string" name="type"/> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="vertices"> <xs:complexType> <xs:sequence> <xs:element name="vertex-group"> <xs:complexType> <xs:sequence> <xs:element name="vertex" maxOccurs="unbounded" minOccurs="0"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute type="xs:string" name="type" use="optional"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element type="xs:string" name="constraints"/> </xs:sequence> <xs:attribute type="xs:string" name="id"/> <xs:attribute type="xs:string" name="type"/> <xs:attribute type="xs:byte" name="area"/> <xs:attribute type="xs:string" name="regular"/> <xs:attribute type="xs:byte" name="sides"/> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="ellipses"> <xs:complexType> <xs:sequence> <xs:element name="ellipse"> <xs:complexType> <xs:sequence> <xs:element type="xs:string" name="constraints"/> </xs:sequence> <xs:attribute type="xs:string" name="id"/> <xs:attribute type="xs:string" name="regular"/> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="lines"> <xs:complexType> <xs:sequence> <xs:element name="line"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute type="xs:string" name="id"/> <xs:attribute type="xs:string" name="name"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> <xs:attribute type="xs:string" name="id"/> <xs:attribute type="xs:string" name="name"/> </xs:complexType> </xs:element> <xs:element name="vertices"> <xs:complexType> <xs:sequence> <xs:element name="group" maxOccurs="unbounded" minOccurs="0"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute type="xs:string" name="id" use="optional"/> <xs:attribute type="xs:string" name="type" use="optional"/> <xs:attribute type="xs:string" name="csv" use="optional"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="angles"> <xs:complexType> <xs:sequence> <xs:element name="angle"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute type="xs:string" name="id"/> <xs:attribute type="xs:string" name="name"/> <xs:attribute type="xs:byte" name="value"/> <xs:attribute type="xs:string" name="from"/> <xs:attribute type="xs:string" name="to"/> <xs:attribute type="xs:string" name="through"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="edges"> <xs:complexType> <xs:sequence> <xs:element name="edge"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute type="xs:byte" name="id"/> <xs:attribute type="xs:string" name="from"/> <xs:attribute type="xs:string" name="to"/> <xs:attribute type="xs:string" name="name"/> <xs:attribute type="xs:byte" name="length"/> </xs:extension> </xs:simpleContent>
```

```
</xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element>
<xs:element name="constraints"> <xs:complexType> <xs:sequence> <xs:element
name="constraints"> <xs:complexType> <xs:simpleContent> <xs:extension
base="xs:string"> <xs:attribute type="xs:string" name="type"/> <xs:attribute
type="xs:byte" name="value"/> <xs:attribute type="xs:string" name="from"/>
<xs:attribute type="xs:string" name="to"/> </xs:extension> </xs:simpleContent>
</xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element>
</xs:sequence> <xs:attribute type="xs:string" name="id"/> <xs:attribute
type="xs:string" name="type"/> <xs:attribute type="xs:string" name="regular"/>
</xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element>
<xs:element type="xs:string" name="constraints"/> </xs:sequence> </xs:complexType>
</xs:element> </xs:sequence> <xs:attribute type="xs:string" name="task"/>
</xs:complexType> </xs:element> </xs:schema>
```

Опис моделі даних у мові програмування Kotlin із використанням анотацій SimpleXML

```
typealias  LGO = Pair<Int, GeometryObject?>

data class RenderOptions(var position: Vector3,
                         var rotation: Quaternion?) {}
@Root(strict = false)
public data class Exercise(
    @field:Attribute(name="task", required=false)
    @param:Attribute(name="task", required=false)
    var task: String? = "Default task name",

    @field:Element(name="scene", required=false)
    @param:Element(name="scene", required=false)
    var scene: Scene? = Scene()
): GeometryObject, Serializable {}
public data class Scene(
    @field:ElementList(required=false, name="constraints",
inline = false)
    @param:ElementList(required=false, name="constraints",
inline = false)
    var constraints: MutableList<Constraint>? = null,

    @field:ElementList(required=false, name="shapes")
    @param:ElementList(required=false, name="shapes")
    var shapes: MutableList<Shape>? = mutableListOf(Shape())
): GeometryObject, Serializable {}

@Root(name="shape", strict = false)
public data class Shape(

    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,
```

```
    @field:Attribute(required=false, name="type")
    @param:Attribute(required=false, name="type")
    var type: String? = null,

    @field:Attribute(required=false, name="regular")
    @param:Attribute(required=false, name="regular")
    var regular: Boolean? = null,

    @field:ElementList(required=false, name="edges")
    @param:ElementList(required=false, name="edges")
    var edges: MutableList<EdgeGroup>? = null,

    @field:ElementList(required=false, name="vertices")
    @param:ElementList(required=false, name="vertices")
    var vertices: MutableList<VertexGroup>? = null,

    @field:ElementList(required=false, name="angles")
    @param:ElementList(required=false, name="angles")
    var angles: MutableList<Angle>? = null,

    @field:ElementList(required=false, name="constraints")
    @param:ElementList(required=false, name="constraints")
    var constraints: MutableList<Constraint>? = null,

    @field:Element(name="base")
    @param:Element(name="base")
    var base: Base = Base()
): GeometryObject, Serializable { }

@Root(name="base", strict = false)
public data class Base(

    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:Attribute(required=false, name="name")
    @param:Attribute(required=false, name="name")
    var name: String? = null,

    @field:ElementList(required=false, name="ellipses",
inline=true)
    @param:ElementList(required=false, name="ellipses",
inline=true)
    var ellipses: MutableList<Ellipse>? = null,

    @field:ElementList(required=false, name="lines",
inline=true)
    @param:ElementList(required=false, name="lines",
```

```
inline=true)
    var lines: MutableList<Line>? = null,

    @field:ElementList(required=false, name="polygons",
inline=true)
    @param:ElementList(required=false, name="polygons",
inline=true)
    var polygons: MutableList<Polygon>? =
mutableListOf(Polygon())

): GeometryObject, Serializable {}

@Root(name="ellipse", strict = false)
public data class Ellipse(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:Attribute(required=false, name="regular")
    @param:Attribute(required=false, name="regular")
    var regular: Boolean? = false,

    @field:Attribute(required=false, name="constraints")
    @param:Attribute(required=false, name="constraints")
    var constraints: List<Constraint>? = null
): GeometryObject, Serializable {}

@Root(name="line", strict = false)
public data class Line(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:Attribute(name="name", required = false)
    @param:Attribute(name="name", required = false)
    var name: String? = null

): GeometryObject { }
@Root(name="polygon", strict = false)
public data class Polygon(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:Attribute(name="type", required = false)
    @param:Attribute(name="type", required =  false)
    var type: String? = null,

    @field:Attribute(required=false, name="area")
```

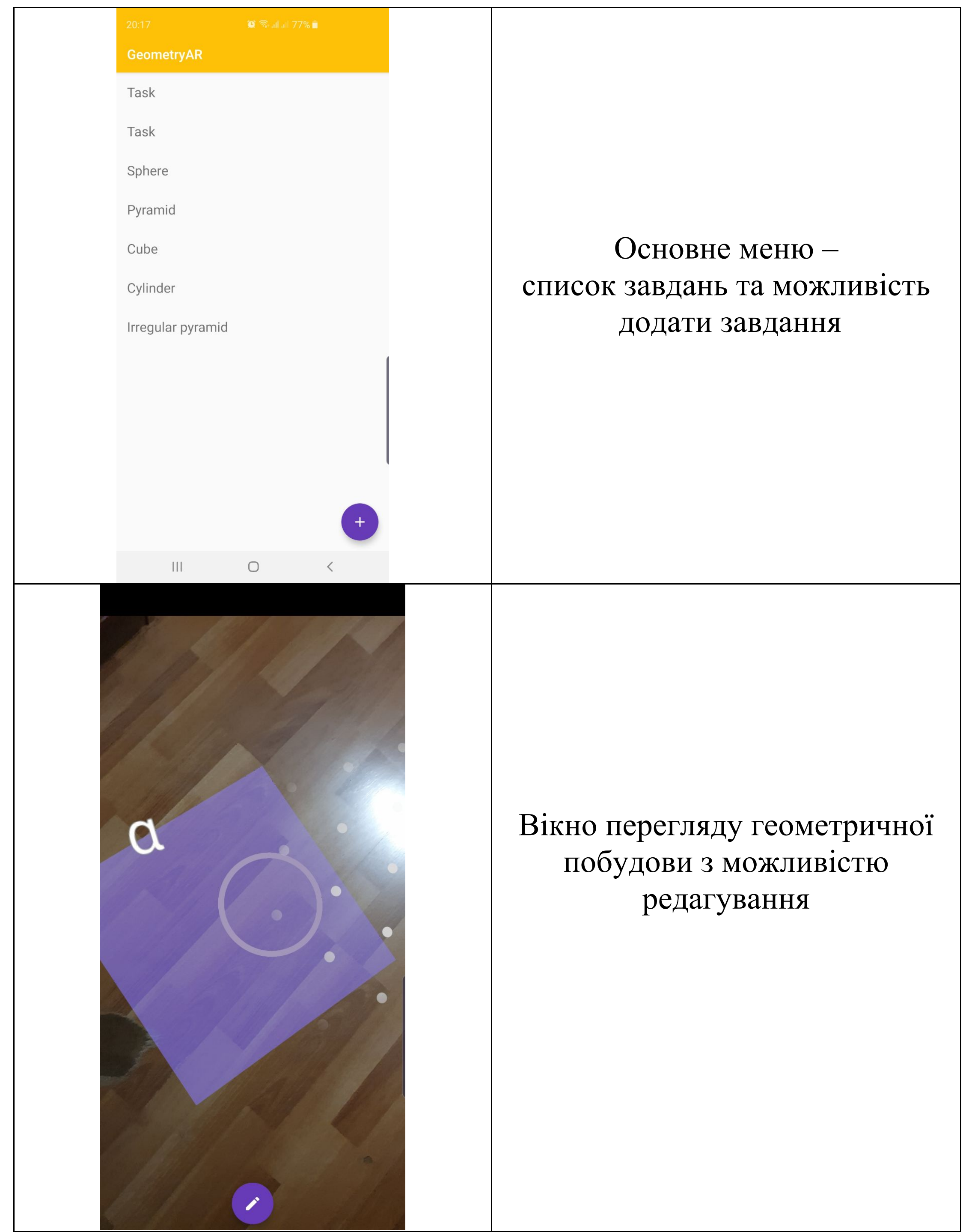
```
    @param:Attribute(required=false, name="area")
    var area: String? = null,

    @field:Attribute(required=false, name="regular")
    @param:Attribute(required=false, name="regular")
    var regular: Boolean? = false,

    @field:Attribute(required=false, name="sides")
    @param:Attribute(required=false, name="sides")
    var sides: Int? = 3,

    @field:ElementList(required=false, name="edges")
    @param:ElementList(required=false, name="edges")
    var edges: List<EdgeGroup>? = null,

    @param:ElementList(required=false, name="vertices")
    @field:ElementList(required=false, name="vertices")
    var vertices: List<VertexGroup>? = null,

    @field:ElementList(required=false, name="constraints")
    @param:ElementList(required=false, name="constraints")
    var constraints: List<Constraint>? = null

): GeometryObject {}

@Root(name="edge-group", strict = false)
public data class EdgeGroup(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:ElementList(required=false, name="edges")
    @param:ElementList(required=false, name="edges")
    var edges: String? = null,

    @field:Attribute(required = false, name="type")
    @param:Attribute(required = false, name="type")
    var type: String? = null
): GeometryObject {}

@Root(name="vertex-group", strict = false)
public data class VertexGroup(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field: ElementList(required=false, name="vertices")
    @param: ElementList(required=false, name="vertices")
    var vertices: String? = null,
```

```kotlin
    @field:Attribute(name="type")
    @param:Attribute(name="type")
    var type: String? = null
): GeometryObject {}

@Root(name="angle", strict = false)
public data class Angle(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:Attribute(required=false, name="name")
    @param:Attribute(required=false, name="name")
    var name: String? = null,

    @field:Attribute(name="from")
    @param:Attribute(name="from")
    var from: String? = null,

    @field:Attribute(name="to")
    @param:Attribute(name="to")
    var to: String? = null,

    @field:Attribute(name="through")
    @param:Attribute(name="through")
    var through: String? = null,

    @field:Attribute(required=false, name="value")
    @param:Attribute(required=false, name="value")
    var value: Int? = null
): GeometryObject {}

@Root(name="vertex", strict = false)
public data class Vertex(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,

    @field:Attribute(required=false, name="name")
    @param:Attribute(required=false, name="name")
    var name: String? = null
): GeometryObject {}

@Root(name="edge", strict = false)
public data class Edge(
    @field:Attribute(required=false, name="id")
    @param:Attribute(required=false, name="id")
    var id: String? = null,
```

```kotlin
    @field:Attribute(name="from")
    @param:Attribute(name="from")
    var from: String? = null,

    @field:Attribute(name="to")
    @param:Attribute(name="to")
    var to: String? = null,

    @field:Attribute(required=false, name="name")
    @param:Attribute(required=false, name="name")
    var name: String? = null,

    @field:Attribute(required=false, name="length")
    @param:Attribute(required=false, name="length")
    var length: Int? = null
): GeometryObject {}

@Root(name="constraint", strict = false)
public data class Constraint(
    @field:Attribute(name="type")
    @param:Attribute(name="type")
    var type: String = "spacing",

    @field:Attribute(required=false, name="value")
    @param:Attribute(required=false, name="value")
    var value: Int? = null,

    @field:Attribute(name="from", required = false)
    @param:Attribute(name="from", required = false)
    var from: String? = null,

    @field:Attribute(name="to", required = false)
    @param:Attribute(name="to", required = false)
    var to: String? = null
): GeometryObject, Affecter {}
```

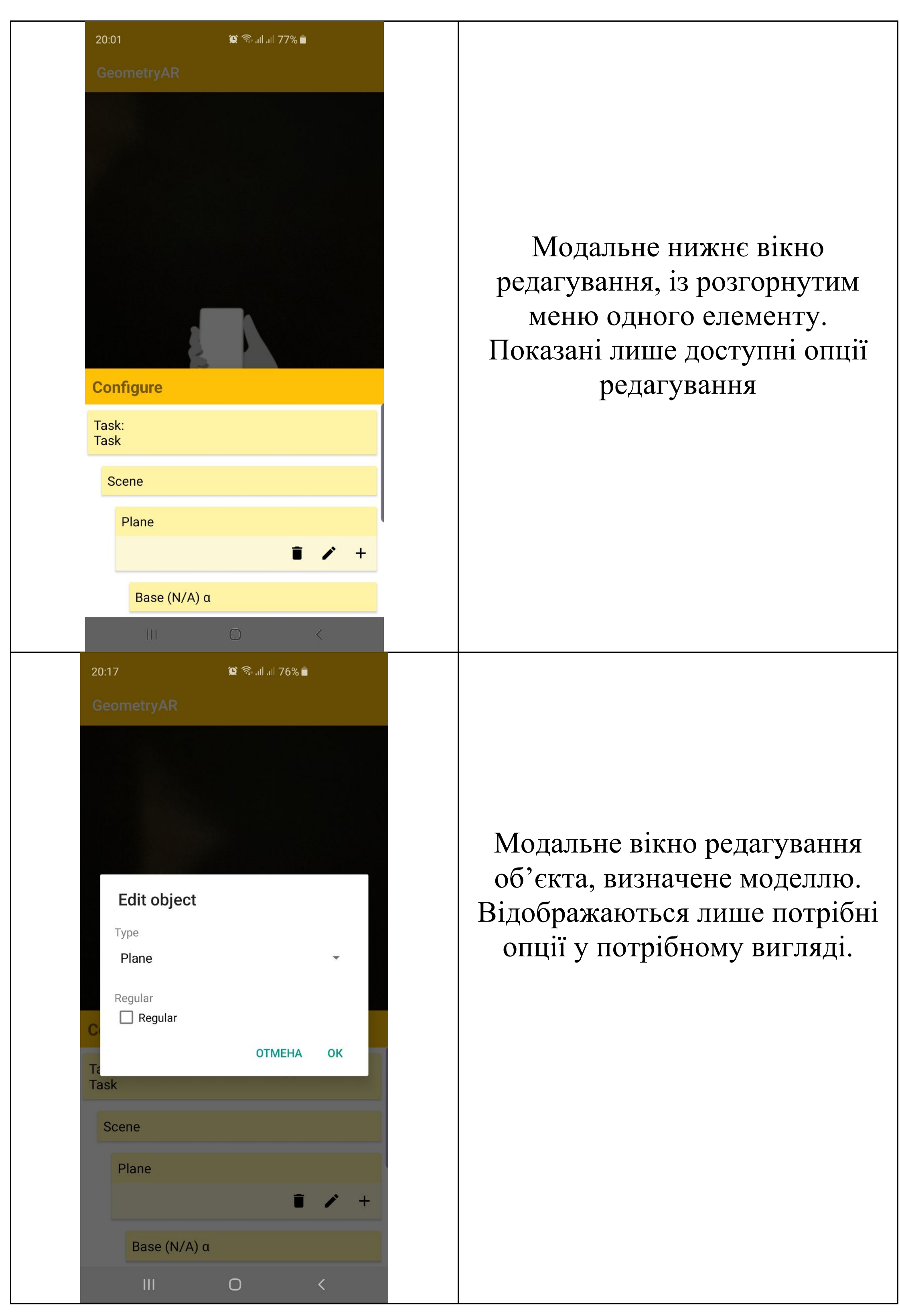
Приклади роботи програми

Знімок екрану	Опис

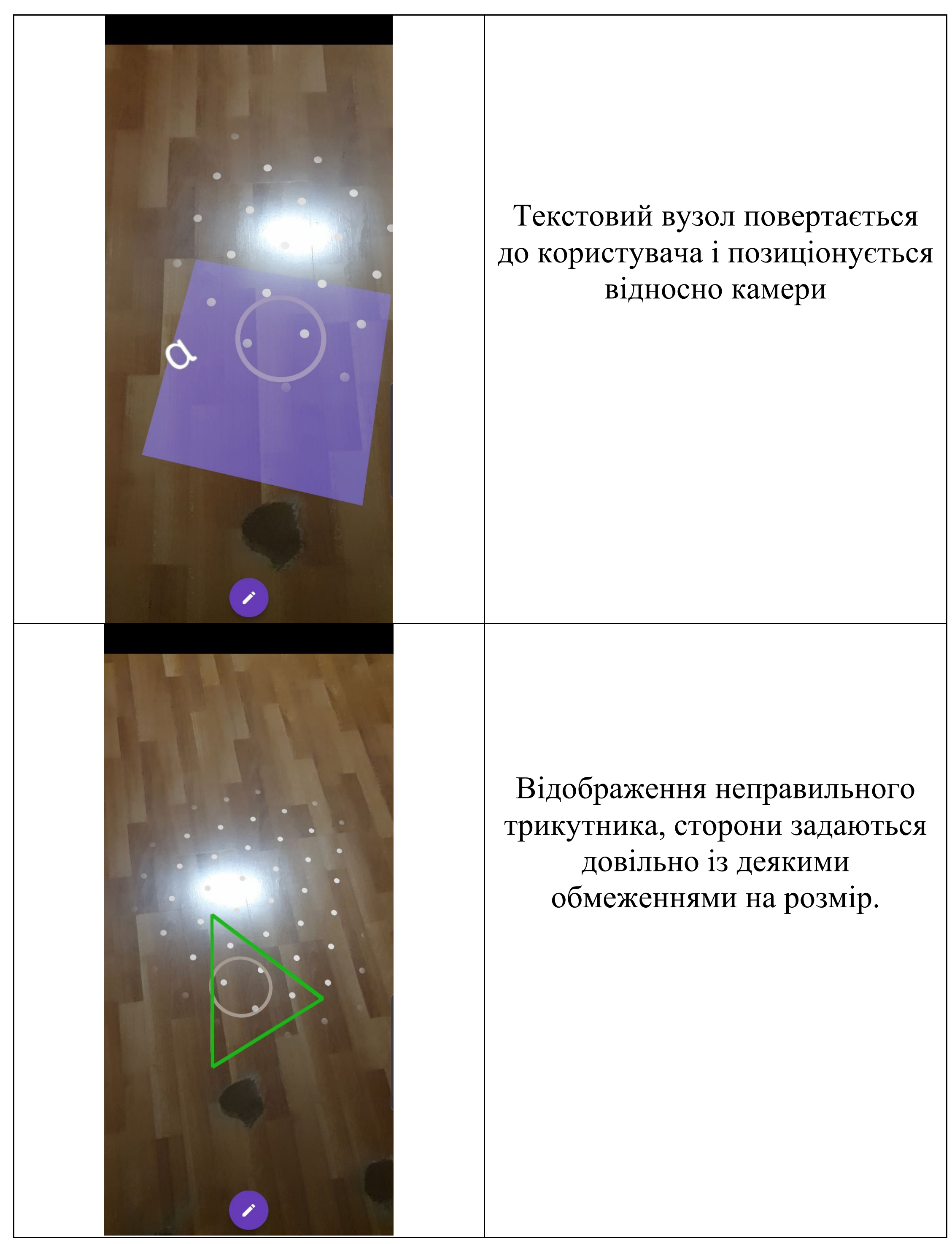


Основне меню –
список завдань та можливість
додати завдання

Вікно перегляду геометричної
побудови з можливістю
редагування



	Модальне нижнє вікно редагування, із розгорнутим меню одного елементу. Показані лише доступні опції редагування
	Модальне вікно редагування об’єкта, визначене моделлю. Відображаються лише потрібні опції у потрібному вигляді.



	Текстовий вузол повертається до користувача і позиціонується відносно камери
	Відображення неправильного трикутника, сторони задаються довільно із деякими обмеженнями на розмір.