# 3D-гра з нуля

Розробляємо повноцінний 3D-рушій на Pascal

Андрій Токарський

2026

# Зміст

# Вступ

Коли я був школярем, відеоігри уже були частиною культури у нашій школі. Мої однокласники грали в **Counter Strike 1.6** в комп'ютерних клубах, на мобільних телефонах уже існували прості **Java ME** ігри, й самі персональні комп'ютери, які могли підтримувати більшість популярних на той момент ігор, все частіше ставали не розвагою еліт, а частиною буденності кожного.

Моїми першими іграми були **Captain Claw**, **Freddi Fish 4**, **Doom**. Усі вони запускалися на старенькому комп'ютері із операційною системою **Windows 98**. Ближче до старших класів у нас з'явився новий сучасний комп'ютер, і я мав змогу познайомитися із новими іграми. Моєю улюбленою серією ігор стала **Half-Life**, адже вона була уособленням всього, що я полюбив у іграх. Перша частина подарувала цікавий загадковий сюжет, у якому ти відчуваєш себе частиною подій, що відбуваються під час катастрофи у науково-дослідницькому комплексі. Гра пропонувала дуже хороше поєднання цікавого ігрового процесу, нової на той момент **імерсивної** подачі сюжету і загадкової атмосфери. **Half-Life 2** до цієї формули додала надзвичайно реалістичну фізику і правдоподібні лицеві анімації.

Я буквально бачив прогрес на власні очі. Кожна наступна гра виглядала значно гарніше за попередню. Особливо я любив 3D-ігри. Тривимірність додавала такого рівня глибини і реалістичності віртуальним світам, якого жодна двовимірна гра не могла запропонувати.

І от в якийсь момент я зацікавився: як це все було влаштовано? Яким чином комп'ютер, тобто машина для оперування одиницями і нулями, може зобразити цілі тривимірні світи? Пам'ятаю, я пробував «гуглити» відповідь на це запитання, і постійно розчаровувався. Весь матеріал був дуже

складним! Я просто не розумів ті «ієрогліфи», більше того, найбільш якісна інформація була написана англійською мовою, яку на той момент я знав погано.

Зараз я розумію, що той матеріал був просто не розрахований на середньостатистичного учня. Щоб його зрозуміти, потрібно було мати глибокі знання у галузі **лінійної алгебри**, **математичного аналізу**, **структур даних** тощо, а все це вивчають уже в університеті.

Проте, будучи вже дорослим, я дізнався про підхід, який дозволяє створити псевдотривимірну гру, використовуючи лише шкільну математику. Називається він **рейкастинг** (англ. *raycasting* — «кидання променів»). Саме ця техніка лежала в основі легендарної гри **Wolfenstein 3D** (1992), яка стала однією з перших комерційно успішних 3D-ігор в історії.

Ідея рейкастингу напрочуд проста. Уявіть, що ви стоїте в лабіринті та дивитесь вперед. З вашого ока виходять невидимі промені — по одному на кожен вертикальний стовпчик екрану. Кожен промінь летить прямо, поки не зустріне стіну. Чим далі стіна — тим меншою вона виглядає на екрані, чим ближче — тим більшою. Ось і все! Жодних матриць, жодних тривимірних координат — лише відстані та пропорції, які можна порахувати за допомогою тригонометрії та теореми Піфагора.

Саме тому рейкастинг — чудова відправна точка для тих, хто хоче зрозуміти, як працює комп'ютерна графіка, але поки що не вивчав вищу математику.

**Математика: на що орієнтуватись**

Щоб читати далі без зайвого стресу, корисно мати базові уявлення про таке (звичайна шкільна програма):

- координати на площині (декартова система).
- тригонометрія на координатній площині (синус, косинус, тангенс);
- теорема Піфагора;

- відношення й пропорції, подібність трикутників;
- вектори: координати та прості операції з ними;

Якщо ж якусь тему вище не знаєте, пропустили або забули — не біда, адже у мережі є багато простих пояснень.

У цій книзі ми крок за кроком розберемо всю математику, яка стоїть за цією технікою, і напишемо власний **ігровий рушій** (що працює на базі рейкастингу) мовою програмування **Pascal**. Рушій, який ми спробуємо написати, дуже простий. Ми матимемо змогу керувати персонажем на ігровій карті, на якій також можуть бути розміщені ігрові об'єкти.

**Визначення**

**Ігровий рушій** (англ. *Game engine*) — центральна частина відеогри, що відповідає за всю її технічну сторону. Тобто за відображення об'єктів, освітлення, звук, фізику та інші ключові компоненти.

Книга іноді пропонуватиме запитання і задачі для закріплення матеріалу і перевірки знань. Деякі задачі будуть трошки «забігати наперед», щоб подальший матеріал було трохи легше засвоювати. Якщо ви відчуваєте, що запитання і задачі надто складні, спробуйте самі шукати інформацію в Інтернеті, і, що головне — **засвоїти її**. Заглядайте у розділ із відповідями тільки для того, щоб звірити уже засвоєні знання.

**Запитання 1**

Що таке **тангенс** і **арктангенс**? Чим відрізняються ці поняття і як вони пов'язані? Яка область визначення цих функцій?

**Запитання 2**

Що таке **радіан**?

**Задача 1**

Довжини катетів прямокутного трикутника — 3 і 4. Яка довжина гіпотенузи?

У книзі іноді також будуть блоки інформації, на які слід звертати особливу увагу:

**Увага**

Програмувати ми будемо мовою **Pascal**. У книзі буде короткий вступ у цю мову, у якому я швидко пройдусь по тих особливостях мови, що ми будемо використовувати. Проте все ж, величезним плюсом буде, якщо ви повторите основи алгоритмізації.

Також іноді у книзі будуть необов'язкові, але цікаві примітки:

**Примітка**

Творцем мови програмування Pascal був знаменитий швейцарський вчений-інформатик **Ніклаус Вірт** (англ. *Niklaus Wirth*). Мета, з якою створювалась ця мова, — «для навчання програмуванню як систематичній дисципліні». Першим компілятором для мови Pascal став **ETH Pascal**, створений у 1970-му році. Проте сьогодні актуальним компілятором є **FPC (Free Pascal Compiler)**. Саме ним ми будемо користуватись для розробки гри.

Розділ 1

# Як працює Wolfenstein 3D?

Для того, щоб зрозуміти ідею рейкаст-рушіїв, подивіться на скріншот гри Wolfenstein 3D (Рис. 1.1) і спробуйте виділити її характерні риси. Що характеризує цю гру? Спробуйте відповісти собі на це питання перед тим, як читати далі.


Рис. 1.1: Скріншот гри Wolfenstein 3D.

Ми бачимо тривимірне зображення. На світ ми дивимося наче очима головного героя і бачимо ряд об'єктів: деякі знаходяться ближче до нас, а деякі — далі.

Проте ми не можемо не помітити, що ця тривимірність — трохи дивна.

По-перше, всі стіни на скріншоті мають однакову висоту.

По-друге, підлога і стеля зафарбовані одноманітно: один колір для стелі, один — для підлоги.

По-третє, об'єкти (руки, вази) насправді не тривимірні, а плоскі. Більше того, якби ви зараз грали в цю гру і намагалися ходити біля вази, дивлячись на неї, ви б помітили, що вона наче завжди повернута одною стороною до вас.

Четверта особливість — дуже рівномірне світло. Ніде немає тіней! Єдине, що імітує наявність світла тут, — це те, що стіни, які «дивляться» вправо, виглядають темнішими, а ті, що вліво — світлішими.

Ви, звичайно, заперечите, що у ваз є тіні. Проте це лише «трюк» розробників гри для їх імітації. Тіні від ваз тут — частина зображення вази. Тобто вони завжди будуть повернуті вправо від вази, незалежно від того, із якого боку на вазу дивиться гравець.

Ще одну цікаву особливість можна помітити, якщо подивитися на ігрову карту зверху (Рис. 1.2).

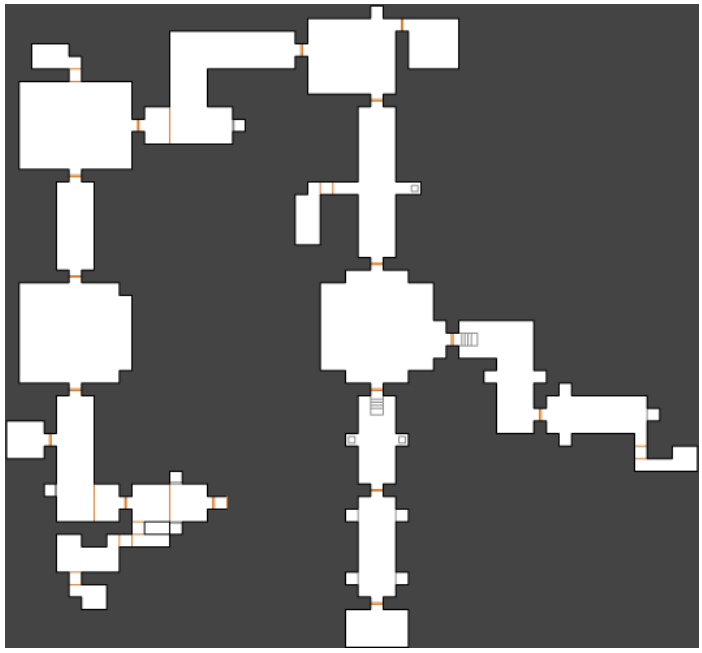


Рис. 1.2: Карта першого рівня Wolfenstein 3D (вигляд зверху).

Погодьтеся, карта виглядає так, наче уся вона складається із блоків однакової довжини і ширини. Цю карту ви неначе зможете намалювати в зошиті, зафарбовуючи клітинки-стіни.

Рис. 1.3: Зафарбована клітинка — стіна.

На нашому уявному зошитному полотні (Рис. 1.3) ви можете або зафарбувати клітинку (тоді вона буде стіною), або залишити її незафарбованою — тоді це простір, у якому персонаж може переміщуватися. Усі стіни мають однакову висоту, яка дорівнює довжині й ширині одного блоку.

Це означає, що у Wolfenstein 3D неможливі стіни, розташовані **не** під прямим кутом до інших стін, неможливі сходи, нахилені поверхні, поверхи й інша складна геометрія.

Уся карта — це по суті **двовимірна** структура. І уся ігрова логіка (положення і рух персонажа, об'єкти, стіни тощо) описується виключно планіметрією.

**Визначення**

Планіметрія (від лат. *planum* — площина) — розділ геометрії, що вивчає двовимірні (одноплощинні) фігури, тобто фігури, які можна розташувати в межах однієї площини.

Тривимірне зображення, яке ми бачимо, — це виключно результат проекції на екран.

Такі ігрові рушії, що, будучи повністю двовимірними, намагаються здаватися тривимірними, іноді називають **псевдотривимірними** або **2.5D-**

**рушіями**.

> **Запитання 3**
>
> Як ви вважаєте, у чому причина таких обмежень? Що заважало розробникам гри зробити повноцінне 3D із більш різноманітною геометрією?

## 1.1 Рейкастинг

Вигляньте у вікно, і спробуйте подивитися на якийсь близький і далекий об'єкт. Спробуйте уявити два відрізки. Перший із них знаходиться між вами і далеким об'єктом. Другий — між вами і близьким об'єктом. Який із двох відрізків буде довшим? Який із двох об'єктів здається більшим для нас?

Відповідь проста. Для першого об'єкта — довший відрізок, але сам об'єкт здаватиметься нам малим, для другого об'єкта — все навпаки. Це явище називається **перспективою** (англ. *perspective*): об'єкти, розташовані далеко, здаються меншими, ніж об'єкти, розташовані близько. Врахування перспективи при відображенні на екрані — обов'язкове для тривимірного зображення.

Тепер подумаємо, як застосувати це при відображенні карти.

На Рис. 1.4 зображена карта. Персонаж знаходиться на місці червоної точки, і дивиться у напрямку вектора $\overrightarrow{n}$. Давайте подумаємо, як буде виглядати світ очима такого персонажа на екрані, ширина якого 16 пікселів.

Для цього ми рівномірно випускаємо 16 променів від персонажа (тому що ширина екрану — 16 пікселів). Усі промені рівномірно розподілені у межах **поля зору** (англ. *field of view*, скорочено *FOV*).

Поле зору — це кутовий діапазон, у якому персонаж може бачити навколишній світ. Людина має приблизно 180° периферійного зору, але в іграх зазвичай використовують значно менший кут — від 60° до 90°. На Рис. 1.4 поле зору — це кут між першим і останнім променем.

Для кожного променя ми шукаємо найближчу стіну, із якою він перетинається. На Рис. 1.4 точки перетину зі стінами позначені маленькими червоними точками. Тобто ми бачимо набір відрізків між персонажем і стінами, які він бачить.

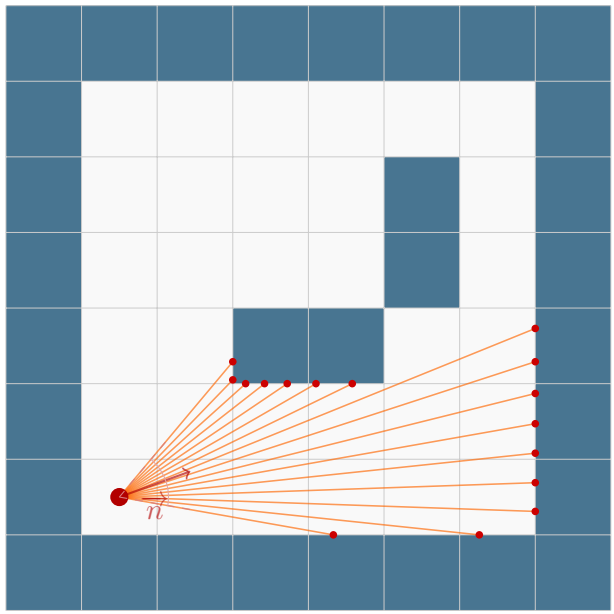



Рис. 1.4: Ігрова карта з променями рейкастингу.

Тепер розглянемо алгоритм, за яким кінцеве зображення будується на екрані.

- Ділимо екран горизонтально на дві половини (верхня відповідатиме стелі, нижня — підлозі)
- Фарбуємо стелю і підлогу у будь-який колір (на Рис. 1.5 можна помітити, що верхня половина — світло-сіра, нижня — темно-сіра).
- Для кожного відрізка (Рис. 1.4) ми на екрані малюємо вертикальну лінію: чим довший відрізок, тим менша лінія.

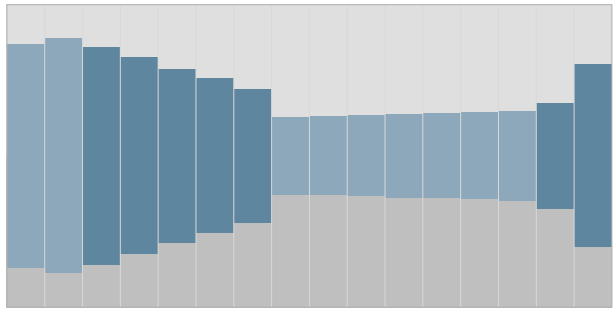


Рис. 1.5: Результат рейкастингу: псевдотривимірне зображення.

На Рис. 1.5 ми бачимо, хоч і низькоякісне, але правдоподібне 3D-зображення, яке ми отримали внаслідок виконання алгоритму вище. Зрозуміло, що чим більшою є ширина екрану, тобто чим більше променів ми випускаємо в межах поля зору, тим більш плавною буде здаватися стіна.

Спробуємо простежити зв'язок між картою та екраном на конкретному прикладі. Подивіться на найлівіший стовпчик Рис. 1.5 — він досить високий. Цей стовпчик відповідає променю, який на карті (Рис. 1.4) іде найбільше вліво від напрямку погляду. Цей промінь влучає у близьку внутрішню стіну, тому відрізок на карті короткий — а стовпчик на екрані, відповідно, високий. Натомість стовпчики в центрі екрану дуже низькі, адже відповідні промені долітають аж до далекої правої стіни-межі.

**Задача 2**

Визначте, які стовпчики на Рис. 1.5 відповідають яким променям на Рис. 1.4.

**Задача 3**

Пряма описується рівнянням $y = 2x + 3$. Визначте точку її перетину із прямими $x = 2$ та $y = 4$.

На Рис. 1.5 можна також помітити, що деякі стіни зображені світлішими, а деякі — темнішими. Це досить простий і дешевий спосіб зробити ілюзію освітлення. Якщо промені перетинають ліву чи праву частину

стіни (не забуваємо, що всі стіни у нас — квадратні), ми фарбуємо їх звичайним кольором. Якщо ж промені перетинають стіну зверху чи знизу — ми «приглушуємо» наш колір до менш інтенсивної версії, і стіни здаються темнішими. Завдяки цьому картинка виглядає більш живою, аніж версія без ілюзії освітлення (порівняйте самі Рис. 1.6 і Рис. 1.5).

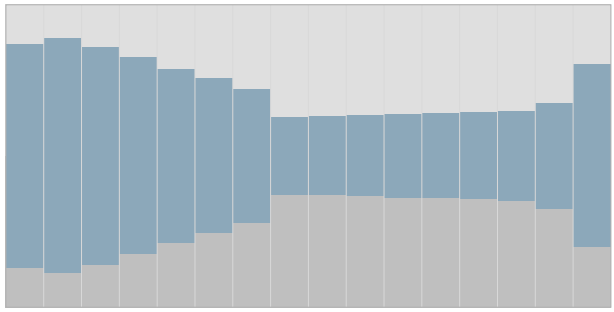


Рис. 1.6: Результат рейкастингу, але без затемлення стін.

Тепер ви знаєте загальну ідею рейкастингу. Ми побачили, що для побудови псевдотривимірного зображення достатньо вміти знаходити відстані від персонажа до стін. У наступних розділах ми крок за кроком вивчимо математику, необхідну для реалізації кожного етапу цього алгоритму: вектори, тригонометрію та перетин променів зі стінами.

Розділ 2

# Математика

У цьому розділі будуть пояснені основні математичні поняття, які будуть надалі застосовані.

Навіть якщо ви впевнені у своїх математичних здібностях, все ж рекомендую перечитати цей розділ, оскільки у комп'ютерній графіці деякі нюанси, пов'язані із геометрією, дещо відрізняються від того, що ви вивчали в школі.

Також деякі позначення, які будуть застосовані у книзі, можуть виявитись ближчими до університетського курсу **лінійної алгебри**, ніж до шкільного курсу геометрії (це ніяк не робитиме матеріал складнішим), а для багатьох понять будуть даватись англомовні іменування (що дуже важливо, оскільки більшість по-справжньому якісної інформації про розробку ігор в Інтернеті — англомовна).

## 2.1 Система координат

Якби я попросив вас зобразити точку із координатами $(3, 2)$ на координатній площині, ви би, скоріш за все, зобразили щось схоже на Рис. 2.1. Ви би побудували вісь $x$, напрямлену вправо, і вісь $y$, напрямлену вверх. Ви б розмістили поділки на координатних осях, знайшли б місце із координатами $(3, 2)$ і намалювали б там точку.

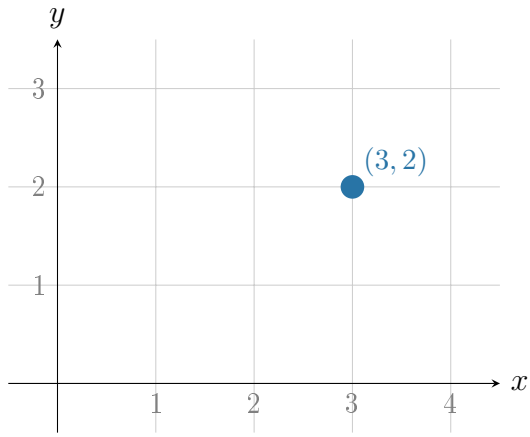



Рис. 2.1: Точка $(3, 2)$ у **класичній** системі координат: вісь $y$ зростає вгору.

Це абсолютно правильна відповідь, проте у комп'ютерній графіці часто використовують зовсім іншу систему координат, вісь $y$ якої росте **вниз** (Рис. 2.2)!

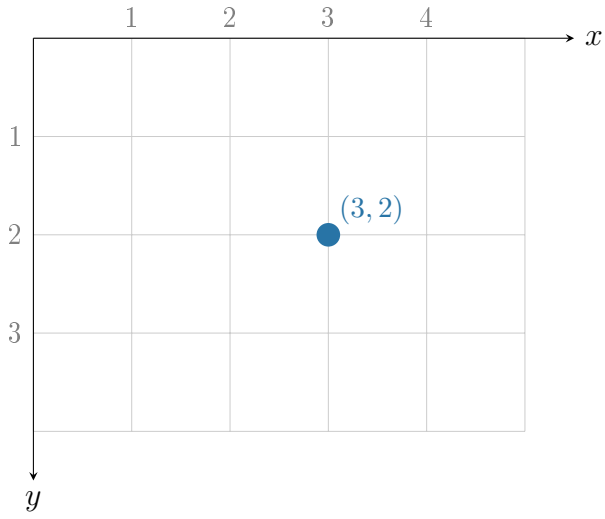



Рис. 2.2: Та сама точка $(3, 2)$ у системі, **як на екрані**: вісь $y$ зростає вниз, початок координат зверху ліворуч.

Чому так зроблено? Головна причина — монітори. Перший піксель зображення на моніторі знаходиться у верхньому лівому куті, а його номер дорівнює 0. Наступні пікселі мають номери $1, 2, 3, \ldots$ і розташовуються далі вправо, а потім — на наступному рядку. Тож вісь $x$ природно зростає

вправо, а вісь $y$ — вниз, і початок координат опинився у верхньому лівому куті.

Але такою системою координат зручно описувати не лише елементи екрану, а й елементи карти. Ви матимете змогу відчути простоту такої системи координат, коли почнете розробляти рушій в наступних розділах.

## 2.2 Вектори

### 2.2.1 Що таке вектор?

Коли я вчився у школі, мені давали таке визначення вектора:

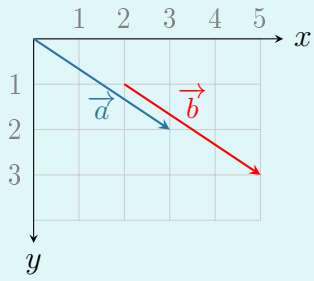
**Визначення**

**Вектор** (англ. *Vector*) — ~~це напрямлений відрізок~~.

Як бачите, я закреслив це визначення, оскільки воно не зовсім правильне. Для того, щоб зрозуміти, чому, пропоную розв'язати задачу (**перед** тим, як читати далі).

**Задача 4**

На рисунку зображені вектори $\overrightarrow{a}$ і $\overrightarrow{b}$. Які їхні координати?




Якщо ви розв'язали задачу правильно (ви ж пробували її розв'язати, правда?), ви могли помітити, що вектори $\overrightarrow{a}$ і $\overrightarrow{b}$ мають однакові координати. Більше того, куди б ви не перемістили вектор $\overrightarrow{a}$, доки він зберігає свій напрям і довжину, його координати залишаються однаковими. Але якби ве-

ктор був просто одним конкретним напрямленим відрізком, то $\overrightarrow{a}$ і $\overrightarrow{b}$ мали б бути різними векторами, адже це різні відрізки з різними початковими і кінцевими точками.

Тому я пропоную інше визначення вектора:

**Визначення**

**Вектор** (англ. *Vector*) — це множина всіх напрямлених відрізків, що мають однаковий напрямок і довжину. Два напрямлені відрізки з однаковим напрямком і довжиною вважаються одним і тим самим вектором.

Це означає, що $\overrightarrow{a}$ і $\overrightarrow{b}$ — один і той самий вектор.

Чому таке розуміння вектора корисне? Тому що вектор описує не конкретне місце, а **зміщення** — наскільки і в якому напрямку треба рухатися.

Наприклад, вектор $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$ означає «на 3 вправо і на 2 вниз», незалежно від того, звідки ми починаємо. Це дуже зручно у іграх: один і той самий вектор швидкості можна застосувати до будь-якого об'єкта на карті, а вектор напрямку погляду персонажа можна використати для побудови променів рейкастингу з будь-якої позиції.

**Примітка**

Ви могли помітити, що у книзі координати вектора записуються у стовпчик. Справа в тому, що у лінійній алгебрі існують два види векторів: вектори-стовпчики, тобто $\overrightarrow{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix}$, і вектори-рядки, тобто $\overleftarrow{a} = \begin{bmatrix} a_x & a_y \end{bmatrix}$. Рішення щодо того, які вектори використовувати як основні, — **конвенційне**, тобто ми як розробники вільні вибрати яке завгодно основне представлення вектора.

В рамках нашої книжки неважливо, рядки чи стовпчики ми використовуватимемо як основу, оскільки різниця між цими поняттями проявлятиметься в **матричних обчисленнях**, про що мова у книзі

не йтиме взагалі.

Я виберу представлення саме у вигляді стовпчика, оскільки так легше читати складні формули у книжному форматі.

Більше того, координати точок в іграх теж описуються векторами! І це звучить логічно, адже координати точки можна уявити як **зміщення цієї самої точки від початку координат**. Тобто будь-яку точку із координатами $(p_x, p_y)$ можна описати як вектор $\overrightarrow{p} = \begin{bmatrix} p_x \\ p_y \end{bmatrix}$. Такі вектори, що описують положення точки, називаються **радіус-векторами**.

**Визначення**

**Радіус-вектор** — вектор, проведений з початку координат до даної точки.

**Примітка**

Зверніть увагу, що в англомовній літературі поняття «радіус-вектор» використовується рідше. Частіше вживають термін **«position vector»**.

### 2.2.2 Додавання і віднімання векторів

Уявімо точку $p$, яка описується радіус-вектором $\overrightarrow{p}$ (надалі я буду дозволяти собі писати «точку $\overrightarrow{p}$», ви тепер розумієте, що це означає). Нехай ця точка описується деякими координатами:

$$\overrightarrow{p} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Впродовж деякого проміжку часу ця точка перемістилась вправо на 2 і вверх на 1. Тобто зміщення цієї точки:

$$\overrightarrow{\Delta p} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

**Примітка**

Грецька літера $\Delta$ («дельта») традиційно позначає **зміну** або **різницю** якоїсь величини. Запис $\overrightarrow{\Delta p}$ читається як «дельта пе» і означає «наскільки змінилось $\overrightarrow{p}$». Це позначення широко вживається у фізиці, математиці та програмуванні.

Як визначити, яким буде положення точки $\overrightarrow{p}$ після переміщення на $\overrightarrow{\Delta p}$? Все просто! Додати ці вектори (Рис. 2.3):

$$\overrightarrow{p}_{\text{new}} = \overrightarrow{p} + \overrightarrow{\Delta p} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 3+2 \\ 2+(-1) \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

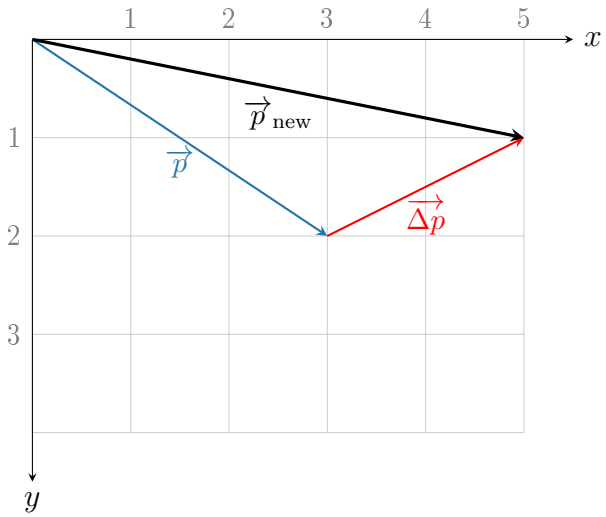



Рис. 2.3: Переміщення точки $\overrightarrow{p}$.

А як, знаючи нові координати $\overrightarrow{p}_{\text{new}}$ і старі координати $\overrightarrow{p}$, дізнатись, наскільки точка перемістилась? Все так само просто! Треба скористатись відніманням векторів:

$$\overrightarrow{\Delta p} = \overrightarrow{p}_{\text{new}} - \overrightarrow{p} = \begin{bmatrix} 5 \\ 1 \end{bmatrix} - \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 5-3 \\ 1-2 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

Як бачимо, **додавання і віднімання** (англ. *Addition and subtraction*) векторів виконується звичайним додаванням чи відніманням відповідних координат:

$$\overrightarrow{a} \pm \overrightarrow{b} = \begin{bmatrix} a_x \pm b_x \\ a_y \pm b_y \end{bmatrix}$$

**Задача 5**

Персонаж знаходиться у точці $\overrightarrow{p} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$. Спочатку він перемістився на $\overrightarrow{\Delta p_1} = \begin{bmatrix} 4 \\ -1 \end{bmatrix}$, а потім — ще на $\overrightarrow{\Delta p_2} = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$. Які координати персонажа після обох переміщень?

### 2.2.3 Модуль вектора

**Визначення**

**Модуль вектора, абсолютне значення вектора** (англ. *Vector magnitude, absolute value*) — це довжина вектора (без урахування його напрямку).

Модуль вектора $\overrightarrow{a}$ позначається як $\|\overrightarrow{a}\|$, і дуже просто обчислюється за **теоремою Піфагора**.

$$\|\overrightarrow{a}\| = \sqrt{a_x^2 + a_y^2}$$

**Задача 6**

Знайдіть модуль вектора $\overrightarrow{v} = \begin{bmatrix} 5 \\ 12 \end{bmatrix}$.

**Запитання 4**

При розробці ігор іноді виникає необхідність порівняти довжини векторів $\overrightarrow{a}$ і $\overrightarrow{b}$, тобто визначити, який із векторів довший. Як ви вважаєте, чому розробники порівнюють не модулі векторів (тобто $\|\overrightarrow{a}\|$, $\|\overrightarrow{b}\|$), а квадрати цих модулів (тобто $\|\overrightarrow{a}\|^2$, $\|\overrightarrow{b}\|^2$)?

### 2.2.4 Множення вектора на скаляр

Що робити, якщо ми хочемо збільшити чи зменшити модуль вектора, але залишити той самий напрям? Для цього використовується **операція множення вектора на скаляр** (англ. *Multiplication by scalar*), тобто на число.

Якщо ми, наприклад, хочемо збільшити вектор вдвічі, ми множимо його на 2, при цьому усі його координати множаться на 2. Зменшення вектора вдвічі — це те ж саме, що і множення вектора на 0.5, або ділення на 2.

Множення вектора на від'ємний скаляр змінює його напрям на протилежний. Наприклад, множення вектора на $-1$ змінює його напрям на протилежний, але зберігає його модуль. А якщо ви помножите вектор на $-3$, то ви не лише зміните його напрям, а й збільшите його модуль утричі.

Маємо таку загальну формулу множення вектора на скаляр:

$$\lambda\overrightarrow{a} = \begin{bmatrix} \lambda a_x \\ \lambda a_y \end{bmatrix}$$

**Задача 7**

Нехай вектор швидкості персонажа $\overrightarrow{v} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$. Обчисліть:

1. $4\overrightarrow{v}$ (персонаж прискорився вчетверо),
2. $-2\overrightarrow{v}$ (персонаж рухається у протилежному напрямку вдвічі швидше).

### 2.2.5 Нормалізація вектора

Іноді від вектора нам може бути не потрібною довжина, а лише напрям.

**Приклад**

Приклад такої ситуації — погляд персонажа в комп'ютерній грі. Персонаж дивиться у якомусь напрямку, але модуль вектора напрямку ні на що не впливає.

Якщо довжина вектора для нас неважлива, ми могли б залишити його як є, але все ж такі вектори прийнято **нормалізувати**.

**Визначення**

**Нормалізований вектор** (англ. *Normalized vector*) — це вектор, що має модуль 1.

**Визначення**

**Нормалізація** (англ. *Normalization*) — операція зменшення довжини вектора до 1 шляхом ділення на його модуль. Нульовий вектор $\overrightarrow{0}$ нормалізувати неможливо, оскільки він не має напрямку (і ділити на нуль не можна), тому результатом нормалізації $\overrightarrow{0}$ вважають $\overrightarrow{0}$.

Формула нормалізації:

$$\widehat{\vec{a}} = \begin{cases} \dfrac{\vec{a}}{\|\vec{a}\|}, & \text{якщо } \|\vec{a}\| \neq 0 \\ \vec{0}, & \text{якщо } \|\vec{a}\| = 0 \end{cases}$$

Вектори, довжина яких неважлива, варто нормалізовувати майже завжди.

**Задача 8**

Нормалізуйте вектор $\vec{a} = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$. Перевірте, що модуль результату дорівнює 1.

## 2.3 Декартові та полярні координати

Уявімо довільний вектор $\vec{a}$. Ми знаємо, що його можна записати у вигляді пари координат $a_x$ і $a_y$, тобто як зміщення вздовж осей.

$$\vec{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix}$$

Такий спосіб представлення вектора називається **декартовими координатами**:

**Визначення**

**Декартові координати** (англ. *Cartesian coordinates*) — спосіб опису положення точки (або вектора) за допомогою зміщень вздовж осей $x$ і $y$.

Але це не єдиний спосіб описати вектор. Замість того, щоб казати «на 3 вправо і на 4 вниз», можна сказати «на відстань 5 під кутом 53°». Такий спосіб називається **полярними координатами**.

**Визначення**

**Полярні координати** (англ. *Polar coordinates*) — спосіб опису положення точки (або вектора) за допомогою відстані $r$ від початку координат та кута $\theta$ відносно додатного напрямку осі $x$.

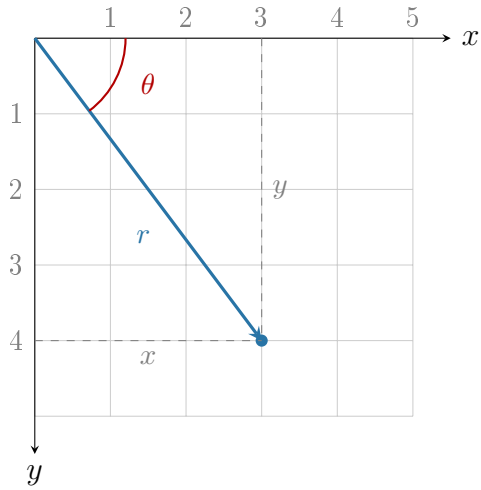



Рис. 2.4: Один і той самий вектор у декартових і полярних координатах.

Перехід між цими двома системами виконується за допомогою тригонометрії. Якщо відомі полярні координати $(r, \theta)$, то декартові обчислюються так:

$$\overrightarrow{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix} = \begin{bmatrix} r\cos\theta \\ r\sin\theta \end{bmatrix}$$

І, навпаки, якщо відомі декартові координати вектора $\overrightarrow{a}$, то полярні:

$$r = \sqrt{a_x^2 + a_y^2}, \quad \theta = \arctan2(a_y, a_x)$$

**Примітка**

Функція $\operatorname{arctan2}(y, x)$ — це «розумний» арктангенс, що враховує знаки обох координат і повертає кут у правильній чверті. На відміну від звичайного $\arctan \frac{y}{x}$, він коректно працює, навіть коли $x = 0$. У більшості мов програмування ця функція доступна «з коробки» як `atan2(y, x)`, або `arctan2(y, x)`.

Повна формула цієї функції виглядає так:

$$\operatorname{arctan2}(y, x) = \begin{cases} \arctan \frac{y}{x}, & \text{якщо } x > 0 \\ \arctan \frac{y}{x} + \pi, & \text{якщо } x < 0 \text{ і } y \geq 0 \\ \arctan \frac{y}{x} - \pi, & \text{якщо } x < 0 \text{ і } y < 0 \\ \frac{\pi}{2}, & \text{якщо } x = 0 \text{ і } y > 0 \\ -\frac{\pi}{2}, & \text{якщо } x = 0 \text{ і } y < 0 \\ \text{Заборонено}, & \text{якщо } x = 0 \text{ і } y = 0 \end{cases}$$

Але запам'ятовувати цю формулу не обов'язково. Ми можемо просто нею користуватись, оскільки стандартні бібліотеки практично всіх мов програмування надають цю функцію: вона **занадто корисна** у сфері комп'ютерної графіки.

### 2.3.1 Навіщо потрібні полярні координати?

У рейкастингу персонаж має **напрямок погляду**, який зручно описувати кутом $\theta$. Коли персонаж повертається вліво чи вправо, кут просто збільшується або зменшується. А для побудови конкретного променя потрібно перетворити цей кут назад у декартовий вектор напрямку:

$$\overrightarrow{n} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$$

**Примітка**

Зверніть увагу, що цей вектор вже нормалізований: його модуль $\|\overrightarrow{n}\| = \sqrt{\cos^2\theta + \sin^2\theta} = 1$. Це випливає з основної тригонометричної тотожності.

Обидві системи координат описують одне й те саме, але в різних ситуаціях зручніша то одна, то інша. Полярні координати спрощують роботу з кутами та обертаннями, а декартові — обчислення відстаней та переміщень. У рейкастингу ви постійно переходитимете між ними.

**Задача 9**

Персонаж дивиться у напрямку під кутом $\theta = 60°$ відносно осі $x$. Знайдіть декартові координати нормалізованого вектора напрямку його погляду.

**Задача 10**

Знайдіть полярні координати $(r, \theta)$ вектора $\overrightarrow{a} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$.

## 2.4 Скалярні і векторні величини

Перш ніж рухатись далі, варто зупинитись на важливому розмежуванні. Деякі величини описуються одним числом, а деякі — кількома.

**Визначення**

**Скалярна величина** (англ. *Scalar*) — величина, яка описується одним числом. Наприклад: відстань, кут, швидкість (без урахування напрямку), час.

**Визначення**

**Векторна величина** (англ. *Vector quantity*) — величина, яка описується вектором, тобто має і значення, і напрямок. Наприклад: положення, переміщення, швидкість із напрямком.

**Задача 11**

Визначте, які з наведених величин є скалярними, а які — векторними:

1. Маса предмета
2. Позиція персонажа на карті
3. Кут повороту персонажа
4. Переміщення персонажа за один кадр
5. Час між кадрами $\Delta t$
6. Швидкість кулі (з урахуванням напрямку)

Тепер подивимось, якими даними описується об'єкт у нашому рушії.

### 2.4.1 Позиція

**Позиція** (англ. *Position*) — це місце, де знаходиться об'єкт. Як ми вже знаємо, вона описується радіус-вектором:

$$\overrightarrow{p} = \begin{bmatrix} p_x \\ p_y \end{bmatrix}$$

Позиція — це векторна величина: нам потрібні обидві координати, щоб знати, де саме знаходиться об'єкт.

### 2.4.2 Напрямок погляду

У нашому рейкастингу персонаж може дивитися в різних напрямках. Як описати, **куди** він дивиться?

Як ми дізнались у попередньому розділі, один і той самий напрямок можна описати двома способами:

- **Кутом** $\theta$ відносно осі $x$ — це скалярна величина (одне число).
- **Нормалізованим вектором** $\vec{n} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$ — це векторна величина (два числа).

Ці два способи взаємозамінні: знаючи кут, ви завжди можете обчислити вектор (через cos і sin), а знаючи вектор — знайти кут (через arctan2). Але коли ви пишете програму, потрібно обрати, яке представлення зберігати як основне.

Кут зручніший для **обертання**: щоб повернути персонажа, достатньо додати або відняти число від $\theta$. Вектор зручніший для **побудови променів** та обчислення переміщень.

У нашій реалізації напрямок погляду персонажа ми зберігатимемо як **кут** $\theta$ відносно осі $x$, а вектор напрямку обчислюватимемо з нього за потреби.

**Запитання 5**

Уявіть, що замість кута $\theta$ ви зберігаєте напрямок погляду як нормалізований вектор $\vec{n}$. Як у такому разі повернути персонажа на 10° вправо? А як це зробити, якщо зберігається кут?

### 2.4.3 Швидкість

Взагалі, **швидкість** (англ. *Velocity*) — векторна величина, оскільки вона описує зміну положення впродовж одиниці часу (1 сек) як вздовж осі $x$, так і вздовж осі $y$:

$$\vec{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

Проте що ми знаємо про нашого персонажа? Що буде, якщо при управлінні ним ви натиснете кнопку «вперед»? Персонаж піде в напрямку свого погляду, тобто вектор швидкості завжди буде напрямлений у сторону погляду персонажа. Скориставшись множенням вектора на скаляр, виразимо швидкість через напрямок погляду:

$$\overrightarrow{v} = s\overrightarrow{n} = s \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}, \text{ де } s \geq 0$$

де $s$ — це абсолютне значення швидкості, тобто скаляр. Оскільки $\overrightarrow{n}$ — нормалізований, модуль швидкості дорівнює:

$$\|\overrightarrow{v}\| = s\|\overrightarrow{n}\| = s \cdot 1 = s$$

Згідно з правилами нашої гри, персонаж завжди рухається у напрямку свого погляду, тому нам не потрібен повний вектор швидкості — достатньо зберігати лише скаляр $s$.

> **Примітка**
>
> В англійській мові є два переклади слова «швидкість». Перший переклад — **velocity**, і це слово вживається у значенні «вектор швидкості». Другий переклад — **speed**, і це слово означає «абсолютне значення швидкості». У нашому випадку, velocity — $\overrightarrow{v}$, а speed — $s$.

Зверніть увагу, наскільки факт того, що напрям нормалізований, все спростив! Завдяки цьому ми можемо дуже легко рахувати вектор швидкості без лишніх операцій ділення на модуль вектора напряму.

### 2.4.4 Час

Час, хоча й є характеристикою не якогось окремого об'єкта, а всього світу в цілому, є надзвичайно важливою величиною, якій ми приділимо особливу увагу.

Спробуйте подивитись на об'єкти, що рухаються: автомобілі, люди на

вулиці, можливо, у вас є домашній улюбленець. Зверніть увагу, що рух у реальному світі відбувається дуже плавно. Між будь-якими двома моментами часу завжди можна знайти ще один момент посередині, і так — нескінченно. Тобто час є **неперервним**: будь-який проміжок часу можна ділити на менші частини скільки завгодно разів.

Проте комп'ютер не вміє оперувати нескінченними даними! Для цього йому потрібна нескінченна пам'ять, якої, на жаль, не існує в природі. Як же тоді комп'ютер виводить на екран відео?

Для того, щоб це зрозуміти, візьміть фотоапарат. Направте його на будь-який об'єкт, що рухається, і зробіть його фотографію. Відрахуйте дві секунди, і зробіть фотографію ще раз. Ще раз відрахуйте дві секунди, і зробіть фото. І ще раз, і ще раз...

Маючи ці фотографії, дивіться на них, міняючи фотографію на наступну кожні дві секунди.

Ви, по суті, бачите слайд-шоу, але це слайд-шоу демонструє зміну положення об'єкта кожні дві секунди.

Фотографія, яку ви тримаєте в даний момент, називається **кадром** (англ. *Frame*).

Час між появою поточного і наступного кадру називається **дельта-час** (англ. *Delta-time*) і позначається як $\Delta t$.

У експерименті, який ви провели, $\Delta t = 2$ сек. Але тепер повторіть експеримент, тільки для $\Delta t = 1$ сек, тобто фотографувати об'єкт щосекунди і міняти фотографію при перегляді щосекунди. Повторіть втретє, тільки для $\Delta t = 0.5$ сек. Що ви помітили?

Сподіваюсь, ви зробили висновок про те, що чим менше значення $\Delta t$, тим менш «рваною» і більш плавною вийде анімація.

Для того, щоб людському оку було «зручно» сприймати картинку на екрані, достатньо, щоби $\Delta t < 0.041$ сек, але до відеоігор ці вимоги жорсткіші, тому що задача відеогри — не лише вивести зображення на екран, а й обробити команди від гравця (на клавіатурі чи геймпаді). При надто високому значенні $\Delta t$ управління здається «дерев'яним». Тому для суто сюжетних ігор використовують $\Delta t \approx 0.033$ сек або $\Delta t \approx 0.016$ сек, а

для динамічних, багатокористувацьких або кіберспортивних дисциплін — $\Delta t \approx 0.016$ сек або взагалі $\Delta t \approx 0.008$ сек.

Але при вимірюванні продуктивності гри частіше використовують не метрику $\Delta t$, а **кадри за секунду** (англ. *Frames per Second*, скорочено **FPS**). Надалі я вживатиму саме абревіатуру FPS, оскільки вона загальновживана серед геймерів і розробників. FPS обчислюється як $\frac{1}{\Delta t}$, і, відповідно:

- Комфортне значення для людського ока — $\approx 24$ FPS
- Для сюжетних і наративних ігор — 30 FPS або 60 FPS
- Для динамічних, багатокористувацьких і кіберспортивних ігор — 60 FPS або 120 FPS

**Увага**

FPS — поняття, створене людьми для людей, тому що поняттям FPS легше оперувати у повсякденному житті. $\Delta t$ у будь-якому випадку залишається основою для обробки даних у рушії.

**Задача 12**

1. Частота кадрів гри — 50 FPS. Чому дорівнює $\Delta t$?
2. В іншій грі $\Delta t = 0.04$ сек. Скільки це FPS?

### 2.4.5 Переміщення персонажа

Тепер, розуміючи, що таке кадр, спробуємо визначити, як переміщується персонаж.

Нехай ми знаємо його позицію $\overrightarrow{p}$ і вектор швидкості $\overrightarrow{v}$. Якою буде його позиція в наступному кадрі?

Ми знаємо, що швидкість — це переміщення, яке здійснить персонаж за 1 секунду. Яке переміщення тоді здійснить персонаж за $\Delta t$? Цю відповідь ми знаємо ще із молодших класів:

$$\overrightarrow{\Delta p} = \overrightarrow{v}\Delta t$$

Тобто точка, у якій персонаж опиниться в наступному кадрі:

$$\overrightarrow{p}_{\text{new}} = \overrightarrow{p} + \overrightarrow{\Delta p} = \overrightarrow{p} + \overrightarrow{v}\Delta t$$

**Примітка**

До речі, те, що ми щойно зробили — обчислили нове положення, додавши швидкість, помножену на час, — є найпростішою формою **чисельного інтегрування** (англ. *Numerical integration*), відомою як **метод Ейлера**. Коли ви пізніше зустрінете символ інтеграла $\int$ у підручнику з математики, знайте: за ним стоїть та сама ідея — підсумовування маленьких кроків.

**Задача 13**

Персонаж знаходиться у точці $\overrightarrow{p} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ і дивиться під кутом $\theta = 0°$ (вздовж осі $x$). Його швидкість $s = 5$, а $\Delta t = 0.2$ сек. Обчисліть нову позицію персонажа $\overrightarrow{p}_{\text{new}}$.

Аналогічно, знаючи положення об'єкта у поточному кадрі $\overrightarrow{p}$ і попередньому кадрі $\overrightarrow{p_{prev}}$, можна визначити швидкість об'єкта.

$$\overrightarrow{v} = \frac{\overrightarrow{p} - \overrightarrow{p_{prev}}}{\Delta t} = \frac{\overrightarrow{\Delta p}}{\Delta t}$$

**Примітка**

А ця операція — обернена до інтегрування — називається **чисельним диференціюванням** (англ. *Numerical differentiation*). Ми обчислили швидкість як відношення зміни положення до зміни часу: $\frac{\overrightarrow{\Delta p}}{\Delta t}$. Саме це позначає похідна $\overrightarrow{p}'$ з підручника математики (також часто використовують позначення $\frac{\mathrm{d}\overrightarrow{p}}{\mathrm{d}t}$) — лише там $\Delta t$ прямує до нуля, а ми працюємо з конкретним, скінченним кроком.

### 2.4.6 Кутова швидкість і поворот

Ми щойно навчились переміщувати персонажа: знаючи позицію $\overrightarrow{p}$ і швидкість $s$, ми обчислюємо нову позицію. Але персонаж має не лише рухатися, а й **повертатися**: коли гравець натискає кнопку «вліво» чи «вправо», напрямок погляду має змінюватись.

Логіка тут абсолютно така сама, як і з переміщенням, тільки замість позиції — кут, а замість лінійної швидкості — **кутова швидкість**.

**Визначення**

**Кутова швидкість** (англ. *Angular velocity*) — швидкість зміни кута, тобто наскільки градусів (або радіан) за секунду повертається персонаж. Позначається $\omega$.

Порівняйте формули — вони однакові за структурою:

**Переміщення:** $\overrightarrow{p}_{\text{new}} = \overrightarrow{p} + \overrightarrow{v} \cdot \Delta t$

**Поворот:** $\theta_{\text{new}} = \theta + \omega \cdot \Delta t$

Як бачите, якщо для лінійного руху ми додаємо вектор швидкості, помножений на час, то для обертання ми додаємо кутову швидкість, помножену на час. Усе те саме, лише простіше — бо кут і кутова швидкість є скалярами, а не векторами.

**Задача 14**

Персонаж дивиться під кутом $\theta = 30°$ і повертається вправо з кутовою швидкістю $\omega = 90°/\text{сек}$. Куди він буде дивитися через $\Delta t = 0.5$ сек?

## 2.5 Промені

У рейкаст-рушіях промені відіграють ключову роль: саме за їхньою допомогою рушій визначає відстань від персонажа до стін. Кожен промінь «летить» від персонажа в певному напрямку, доки не зустріне стіну, і повідомляє, на якій відстані ця стіна знаходиться.

Отож, визначимось спочатку із тим, що таке промінь.

**Визначення**

**Промінь** (англ. *Ray*) — це частина прямої, що має точку початку і напрямок, у якому вона нескінченно продовжується.

Як описати промінь? Достатньо знати точку початку і вектор напрямку. Тобто для опису променя нам достатньо радіус-вектора початку променя $\vec{r}$ і нормалізованого вектора напрямку $\vec{n}$. Пара цих векторів повністю описує промінь.

Задача кожного променя — знайти точку перетину зі стіною, і зробити це **швидко**.

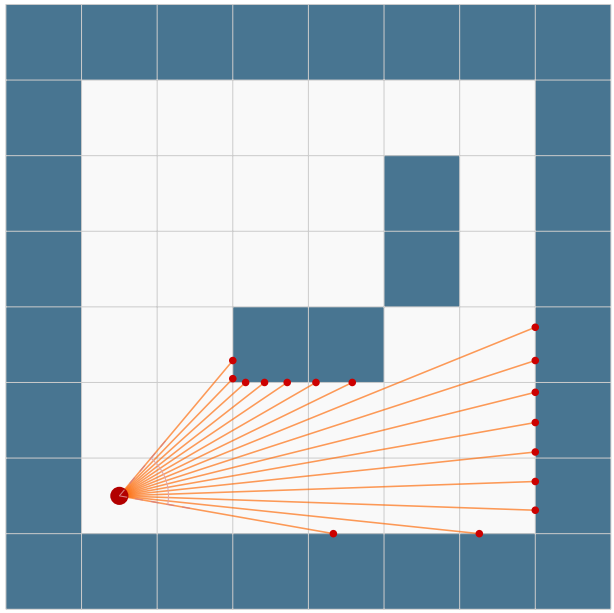


Рис. 2.5: Ігрова карта з променями рейкастингу.

А тепер давайте поглянемо на Рис. 2.5. Пригадайте обмеження нашого рушія: усі стіни — це квадратні клітинки однакового розміру. Подивіться уважно на червоні точки перетину на рисунку. Чи помічаєте закономірність?

Справа в тому, що перетин променів і стін **завжди** знаходиться на сторонах клітинок-стін. А що можна сказати про координати усіх точок на сторонах клітинок? Враховуючи, що усі клітинки розташовані рівномірно, наче в зошиті в клітинку, одна із координат перетину променя зі стіною **завжди** буде кратною довжині сторони стіни. Кратна — означає, що при діленні на довжину сторони остача дорівнює нулю.

Тобто, якщо точка $\overrightarrow{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$ — перетин променя і стіни із довжиною сторони $w$, то:

$$t_x \bmod w = 0 \quad \textbf{або}$$

$$t_y \bmod w = 0$$

**Увага**

Зверніть увагу на дивне слово mod у формулах. Це слово — скорочення від англійського слова **Modulo**, що означає «остача від ділення». Формула $5 \bmod 2 = 1$ буквально означає «остача від ділення 5 на 2 дорівнює 1».

### 2.5.1 Життя окремого променя

**Увага**

Цю частину варто читати дуже уважно. Можливо, навіть кілька разів, оскільки тут буде багато формул. Спробуйте прочитати і зрозуміти все, що тут написане, оскільки без розуміння цього далі буде важко! Тримайте також в голові, що на Рис. 2.6 зображений лише **приклад** променя. У грі промені можуть рухатись у будь-яку сторону.

Знаючи, де промінь **може** перетнути стіну, ми повинні визначитись із алгоритмом пошуку точки перетину променя і стіни. Для цього візьмемо один конкретний промінь (Рис. 2.6), і розглянемо його уважніше.

Промінь проходить як крізь вертикальні, так і крізь горизонтальні лінії «сітки». Давайте спробуємо знайти на Рис. 2.6 всі перетини променя із вертикальними лініями і позначити їх точками $\overrightarrow{c_{v,0}}, \overrightarrow{c_{v,1}}, \ldots$. Буква $c$ в даному випадку означає «кандидат», і це ті точки, які **можуть** бути точкою перетину променя і стіни.

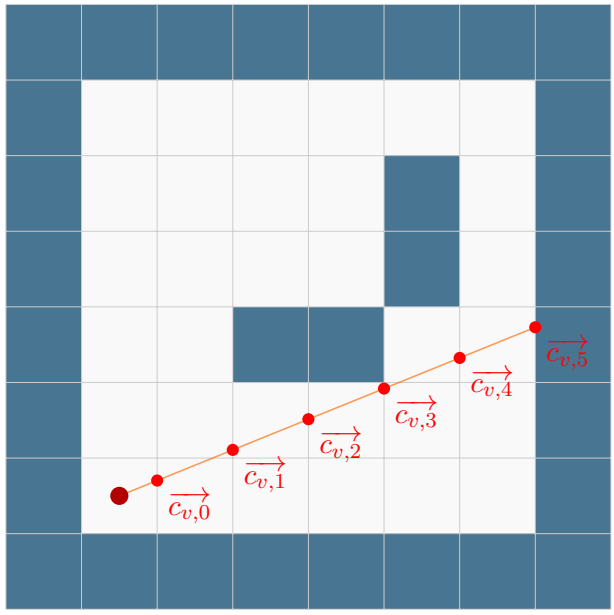



Рис. 2.6: Перетин променя з вертикальними лініями «сітки».

А тепер погляньте на ці точки. Вони розташовані рівномірно! «Рівномірно» означає, що для кожного цілого числа $i \geq 0$ виконується рівність:

$$\overrightarrow{\Delta c_v} = \overrightarrow{c_{v,i+1}} - \overrightarrow{c_{v,i}} = \overrightarrow{c_{v,i+2}} - \overrightarrow{c_{v,i+1}}$$

Тобто, різниця між усіма сусідніми кандидатами дорівнює деякому константному вектору $\overrightarrow{\Delta c_v}$:

$$\overrightarrow{\Delta c_v} = \begin{bmatrix} \Delta c_{vx} \\ \Delta c_{vy} \end{bmatrix}$$

Давайте подумаємо, чому дорівнює $\Delta c_{vx}$, тобто координата $x$ цього вектора?

Якщо наш промінь напрямлений вправо, то координата $x$ кожної наступної точки-кандидата **більша на довжину стіни** за попередню, а, якщо промінь «іде» вліво, — то **менша на довжину стіни**.

Тобто, якщо довжина сторони стіни — $w$, то ми можемо точно сказати, що:

$$|\Delta c_{vx}| = w$$

Про який факт ми ще могли забути? Про те, що, взагалі то, вектор напрямку променя $\overrightarrow{n}$ направлений в ту ж сторону, що і $\overrightarrow{\Delta c_v}$. Тобто, $\overrightarrow{\Delta c_v}$ — це вектор $\overrightarrow{n}$, але помножений на таке число, що зробить модуль його координати $x$ рівним довжині сторони стіни. Скориставшись формулою множення вектора на скаляр, отримуємо:

$$\overrightarrow{\Delta c_v} = \frac{w}{|n_x|} \begin{bmatrix} n_x \\ n_y \end{bmatrix}$$

Зверніть увагу, що ділимо ми саме на $|n_x|$, а не на $n_x$. Це тому, що ми не хочемо випадково «розвернути вектор» в протилежну сторону, якщо $n_x$ виявиться від'ємним. Якщо ж $n_x = 0$ (тобто якщо вектор напрямку є вертикальним), то така операція є забороненою, адже ділити на 0 не можна. В такому випадку ми лише будемо покладатись на перетин із горизонтальними лініями «сітки», про що мова піде трохи далі.

Проте у нас виникає ще одна проблема! Ми не можемо визначити координати першої точки $\overrightarrow{c_{v,0}}$. Насправді, можемо, але досить нестандартним методом. Давайте поділимо координату $x$ персонажа на довжину стіни:

$$\frac{r_x}{w}$$

Результатом такого ділення є деяке дробове число. Давайте відкинемо його дробову частину:

$$\text{floor}\left(\frac{r_x}{w}\right)$$

Якщо відкинути дробову частину, отримуємо **індекс стовпчика** — який за порядком це стовпчик клітинок вздовж осі $x$, рахуючи від **лівого** краю карти вправо. **Нумерація стовпчиків починається з нуля:** найлівіший стовпчик має індекс 0, наступний праворуч — 1, далі — 2 тощо. Це лише позиція вздовж горизонталі; позицію вздовж вертикалі ми не

обчислюємо.

**Увага**

Є кілька способів відкидати дробову частину числа. Перший із них - функція `floor`, яку ми щойно використали.

`floor` (укр. *Підлога*) шукає найближче менше або рівне ціле число до свого аргументу, тобто:

$$\begin{aligned} \text{floor}(3) &= 3 & \text{floor}(-6) &= -6 \\ \text{floor}(2.1) &= 2 & \text{floor}(-6.3) &= -7 \\ \text{floor}(2.7) &= 2 & \text{floor}(-6.7) &= -7 \end{aligned}$$

Але існують і інші функції, наприклад, `ceil`.

`ceil` (укр. *Стеля*) шукає найближче більше або рівне ціле число до свого аргументу, тобто:

$$\begin{aligned} \text{ceil}(3) &= 3 & \text{ceil}(-6) &= -6 \\ \text{ceil}(2.1) &= 3 & \text{ceil}(-6.3) &= -6 \\ \text{ceil}(2.7) &= 3 & \text{ceil}(-6.7) &= -6 \end{aligned}$$

Округлення ж чисел, яке ви вивчали в школі, виконується функцією `round`:

$$\begin{aligned} \text{round}(3) &= 3 & \text{round}(-6) &= -6 \\ \text{round}(2.1) &= 2 & \text{round}(-6.3) &= -6 \\ \text{round}(2.7) &= 3 & \text{round}(-6.7) &= -7 \end{aligned}$$

Знаючи це і те, що клітинки знаходяться рівномірно, можна легко знайти координату $x$ **лівої** сторони клітинки, в якій знаходиться персонаж:

$$\text{floor}\left(\frac{r_x}{w}\right) \cdot w$$

Якщо наш промінь прямує вліво, це і є $x$ координата першого кандидата на перетин зі стіною, тобто:

$$c_{v,0\ x} = \text{floor}\left(\frac{r_x}{w}\right) \cdot w$$

Але якщо промінь тягнеться вправо, то нас цікавить саме $x$ координата **правої** сторони клітинки, яку порахувати дуже легко:

$$c_{v,0\ x} = \text{floor}\left(\frac{r_x}{w}\right) \cdot w + w$$

Тобто, **повна формула $x$ координати першої точки**:

$$c_{v,0\ x} = \begin{cases} \text{floor}\left(\frac{r_x}{w}\right) \cdot w, & \text{якщо } \overrightarrow{n} \text{ прямує вліво} \\ \text{floor}\left(\frac{r_x}{w}\right) \cdot w + w, & \text{якщо } \overrightarrow{n} \text{ прямує вправо} \end{cases}$$

Координату $y$ першої точки-кандидата знаходять за **рівнянням прямої** нашого променя:

$$c_{v,0\ y} = r_y + (c_{v,0\ x} - r_x) \cdot \frac{n_y}{n_x}$$

Таким чином, ми змогли знайти координати першого кандидата:

$$\overrightarrow{c_{v,\,0}} = \begin{bmatrix} m \\ r_y + (m - r_x) \cdot \frac{n_y}{n_x} \end{bmatrix}, \quad \text{де}$$

$$m = \text{floor}\left(\frac{r_x}{w}\right) \cdot w + \text{isFacingRight}(\overrightarrow{n}) \cdot w, \quad \text{де}$$

$$\text{isFacingRight}(\overrightarrow{v}) = \begin{cases} 0, & \text{якщо } \overrightarrow{v} \text{ прямує вліво} \\ 1, & \text{якщо } \overrightarrow{v} \text{ прямує вправо} \end{cases}$$

**Запитання 6**

Як ви вважаєте, де б знаходились точки $\overrightarrow{c_{v,0}}, \overrightarrow{c_{v,1}}, \ldots$, якби рушій все-таки дозволяв би довільні форми стін? Як би від цього змінився перебір точок-кандидатів?

Тобто, алгоритм пошуку **найближчої вертикальної точки перетину** $\overrightarrow{t_v}$ досить простий:

1. Рахуємо $\overrightarrow{\Delta c_v}$ за принципом, вказаним вище.
2. Визначаємо початкового кандидата $\overrightarrow{c_{v,\,0}}$ за принципом, вказаним вище.
3. $\overrightarrow{t_v} := \overrightarrow{c_{v,\,0}}$.
4. Якщо $\overrightarrow{t_v}$ — це точка, що належить клітинці стіни, зупиняємо алгоритм, бо $\overrightarrow{t_v}$ — найближча вертикальна точка перетину променя і стіни.
5. $\overrightarrow{t_v} := \overrightarrow{t_v} + \overrightarrow{\Delta c_v}$.
6. Повертаємося до кроку 4.

> **Примітка**
>
> Запис := у алгоритмі означає операцію присвоєння значення (англ. *Value assignment*). Її суть полягає у тому, що $\overrightarrow{t_v}$ ми сприймаємо як комірку, значення якої ми міняємо, і «присвоїти значення» означає «змінити значення змінної зліва від оператора := на значення справа від :=». Саме так і виглядає операція присвоєння значення в Pascal.

Але, крім вертикальної, нам варто знайти і **горизонтальну** точку перетину $\overrightarrow{t_h}$. І тут алгоритм повністю аналогічний, але із поправкою на те, що саме крок вздовж осі $y$, а не $x$, буде дорівнювати $\pm w$.

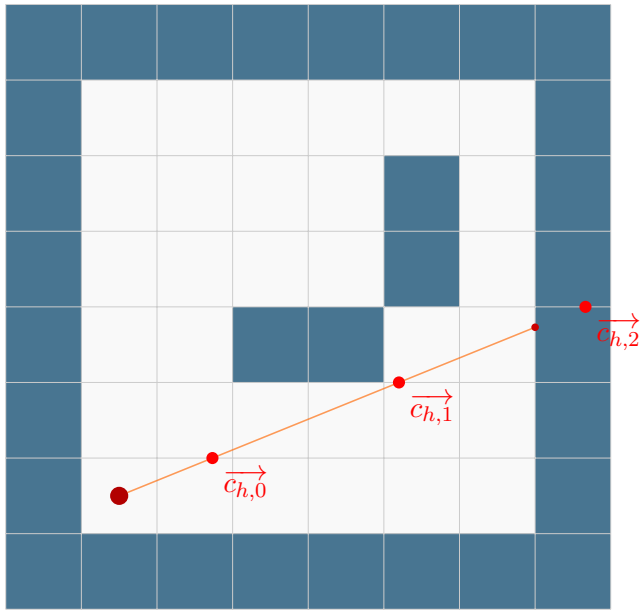



Рис. 2.7: Перетин променя з горизонтальними лініями «сітки».

Немає сенсу повторювати все вище написане, але вже для горизонтальних ліній «сітки» (якщо бажаєте, можете спробувати вивести формули самостійно). Просто напишемо необхідні формули, але уже адаптовані під горизонтальні перетини.

По-перше, нам потрібен вектор кроку $\overrightarrow{\Delta c_h}$:

$$\overrightarrow{\Delta c_h} = \frac{w}{|n_y|} \begin{bmatrix} n_x \\ n_y \end{bmatrix}$$

По-друге, вектор початкової точки:

$$\overrightarrow{c_{h,\,0}} = \begin{bmatrix} r_x + (m - r_y) \cdot \dfrac{n_x}{n_y} \\ m \end{bmatrix}, \qquad \text{де}$$

$$m = \text{floor}\left(\frac{r_y}{w}\right) \cdot w + \text{isFacingDown}(\overrightarrow{n}) \cdot w, \qquad \text{де}$$

$$\text{isFacingDown}(\overrightarrow{v}) = \begin{cases} 0, & \text{якщо } \overrightarrow{v} \text{ прямує вверх} \\ 1, & \text{якщо } \overrightarrow{v} \text{ прямує вниз} \end{cases}$$

Знаючи найближчий вертикальний $\overrightarrow{t_v}$ і горизонтальний $\overrightarrow{t_h}$ перетин, знаходимо точку, що знаходиться ближче до персонажа, і саме вона виявиться точкою перетину променя і стіни $\overrightarrow{t}$:

$$\overrightarrow{t} = \begin{cases} \overrightarrow{t_h}, & \text{якщо } \left\|\overrightarrow{t_h} - \overrightarrow{r}\right\| < \left\|\overrightarrow{t_v} - \overrightarrow{r}\right\| \\ \overrightarrow{t_v}, & \text{якщо } \left\|\overrightarrow{t_h} - \overrightarrow{r}\right\| \geq \left\|\overrightarrow{t_v} - \overrightarrow{r}\right\| \end{cases}$$

Надалі, розмір вертикальних ліній, намальованих на екрані стін, буде залежати саме від значення $\left\|\overrightarrow{t} - \overrightarrow{r}\right\|$: чим воно більше, тим меншою буде здаватись стіна.

## 2.6 Висновки

Ось і вся основна математика, яка буде нам потрібна при розробці. Того, що ми описали у цьому розділі, достатньо, щоб повністю зрозуміти головний принцип технології рейкастингу.

Це не означає, що в процесі програмування у нас не виникне проблем, для яких треба буде шукати математичне рішення, але ідеєю цього розділу було виключно пояснити основний математичний апарат і саму **ідею** рушія, який ми будемо розробляти.

До речі, програмувати уже будемо у наступному розділі!

Розділ 3

# Основи мови Pascal

**Примітка**

Цілком можливо, що ви прекрасно знаєте мову Pascal, і не бажаєте витрачати час на читання цього розділу. Може бути і таке, що ви хочете використовувати іншу більш зручну для вас мову програмування. В такому випадку перед тим, як переходити на наступний розділ, виконайте деякі задачі. Задачу 15 і Задачу 18 треба розв'язати обов'язково. Задачу 16 — за бажанням. Обов'язково також слід прочитати частину про Спільні бібліотеки.

Pascal — досить проста за своєю структурою мова. Недарма вона і досі часто використовується як засіб навчання програмуванню.

Тож у цьому розділі ми повторимо основи Pascal: я не можу бути впевненим, що її вивчають у всіх школах і що вчителі завжди правильно розставляють акценти.

Важливий нюанс: у цій книзі ми не будемо детально «розжовувати» весь синтаксис Pascal, тобто кожне його ключове слово, кожен оператор і кожен знак. Все-таки, ця книжка не називається «Основи програмування». Вміння шукати інформацію в Інтернеті, якщо щось не зрозуміло, — дуже важливе для сучасного програміста.

Зверніть також увагу, що у цьому розділі ми будемо не просто вчитись писати програми, а й робити дещо не менш важливе — розділяти програму на **логічні модулі**. Ми ж не хочемо написати код, який не зможемо підтримувати в майбутньому?

**Увага**

Найнадійніший спосіб засвоїти матеріал цього і подальших розділів — **практикуватися**. Ви повинні писати той код, який ви бачите у прикладах і розв'язувати задачі. Код, який ви **пишете**, стає значно зрозумілішим, ніж той, що ви просто пролистаєте поглядом. Засвоєння матеріалу ще покращиться, якщо ви будете **експериментувати** із кодом.

## 3.1 Встановлюємо необхідні програми

### 3.1.1 Компілятор

Перше, що нам потрібно встановити — це компілятор.

**Визначення**

**Компілятор** (англ. *Compiler*) — це програма, що перетворює програмний текст на виконуваний файл, тобто на кінцеву програму, яку ви можете запустити. Компілятор бере на себе роль перекладача, який перетворює програмний код, зрозумілий програмісту, у набір машинних інструкцій, зрозумілих комп'ютеру. Але також компілятор бере на себе ще одну задачу, а саме перевірку правильності програмного коду, який ви написали. Якщо ви написали код із помилкою, компілятор дасть про це знати!

Компілятор, яким ми будемо користуватись, називається **FPC (Free Pascal Compiler)**. Завантажити його можна по цьому посиланню:

https://www.freepascal.org/download.html

FPC доступний на більшості сучасних операційних систем і апаратних платформ: виберіть свою і завантажте інсталятор за відповідним посиланням. Якщо ви користуєтесь персональним комп'ютером із ОС Windows або

ОС на базі ядра Linux, то, найімовірніше, вашою апаратною платформою є **«AMD64/Intel 64/x86_64»** або, для старіших комп'ютерів, **«Intel x86/i386»**. Сучасні комп'ютери із macOS на базі процесорів Apple Silicon (M1 і новіші) мають платформу **«ARM64/Aarch64»**, а старші макбуки із Intel — **«AMD64/Intel 64/x86_64»**.

На момент написання книги актуальною версією FPC є **3.2.2**. Переконайтесь, що версія компілятора, який ви завантажуєте, рівна або більша за цю версію.

### 3.1.2 Чи встановився компілятор? Термінал і перша команда

Щоб переконатися, що FPC справді встановився, зручно один раз «поговорити» з комп'ютером у **терміналі** (так само кажуть **консоль** або **командний рядок**). Це звичайне вікно з текстом: ви вводите короткий наказ одним рядком, натискаєте Enter, і комп'ютер відповідає текстом. Не потрібно нічого складного — це як повідомлення в чаті, тільки співбесідник — програми на вашому ПК, а не людина.

**Як відкрити термінал.** У Windows натисніть Win+R, у вікні введіть `cmd` і Enter; або в меню «Пуск» знайдіть «Командний рядок» / «cmd» / «PowerShell». У Linux або macOS зазвичай є програма «Термінал» у списку програм.

У відкритому вікні наберіть команду `fpc -iV` і натисніть Enter. Якщо встановлення пройшло успішно, ви побачите рядок із номером версії компілятора — наприклад, щось на кшталт `Free Pascal Compiler version 3.2.2`. Якщо замість цього з'явиться повідомлення, що команда не знайдена (`is not recognized` тощо), значить, термінал не «бачить» FPC: перевірте, чи завершили інсталяцію, і чи під час установлення погодились додати FPC до змінної середовища `PATH` — це потрібно, щоб команда `fpc` працювала з будь-якої папки в терміналі.

Пізніше ми будемо збирати програми саме з термінала, тому цей крок корисний і як перевірка, і як перше знайомство з тим, як розробники за-

пускають інструменти без «чарівних» кнопок у графічному вікні.

## 3.2 Інтегроване середовище розробки

Крім компілятора, нам також потрібен якийсь текстовий редактор, щоб набирати текст програмного коду. Взагалі, для цього достатньо звичайного Блокнота. Але Блокнот використовувати дуже незручно: це просто маленька програма для редактування невеличких текстових файлів, а нам потрібне щось, що стане в нагоді саме в плані написання коду. І для цього використовують так звані **Інтегровані середовища розробки**.

> **Визначення**
>
> **Інтегроване середовище розробки** (англ. *Integrated development environment* або *IDE*) — це спеціальна програма, створена для спрощення життя розробникам. Вона зазвичай включає в себе **підсвітку синтаксису** (англ. *Syntax highlighting*), **автодоповнення** (англ. *Code completion*) та інші дуже корисні інструменти. Серед програмістів прийнято вживати скорочення «**IDE**».

IDE, яким ми будемо користуватись, називається **Visual Studio Code**, і його можна завантажити за посиланням:

https://code.visualstudio.com/Download

Коли ви встановите і відкриєте IDE, ви побачите таке вікно, як на Рис. 3.1.

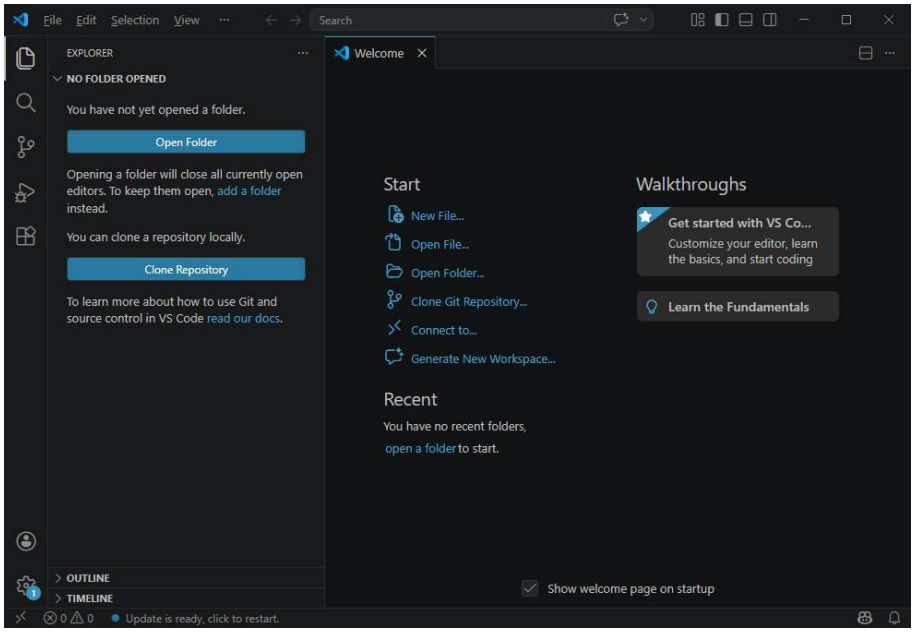



Рис. 3.1: Visual Studio Code: перший запуск

Проблема у тому, що VS Code зараз не знає про мову Pascal нічого, оскільки вона створювалась як IDE, незалежна від конкретної мови.

Щоб у VS Code з'явилась підтримка Pascal, потрібно встановити спеціальне розширення (англ. *Extension*). Для цього знайдіть кнопку із чотирма квадратами зліва, і натисніть її. У полі пошуку введіть слово «**Omni-Pascal**», і встановіть відповідне розширення. Якщо з часом це розширення зникне з маркетплейсу чи перестане оновлюватись, підійде й **інше розширення з підсвіткою синтаксису Pascal**: головне, щоб код у файлах `.pas` не лишався суцільним сірим текстом без кольорів.

Після встановлення IDE буде повністю готове для роботи із Pascal.

Ну що, спробуємо написати першу програму?

## 3.3 Перша програма

Перш за все, створіть папку, у якій вам буде зручно писати ваші тестові програми. Відкрийте цю папку за допомогою VS Code (кнопка «Open Folder»).

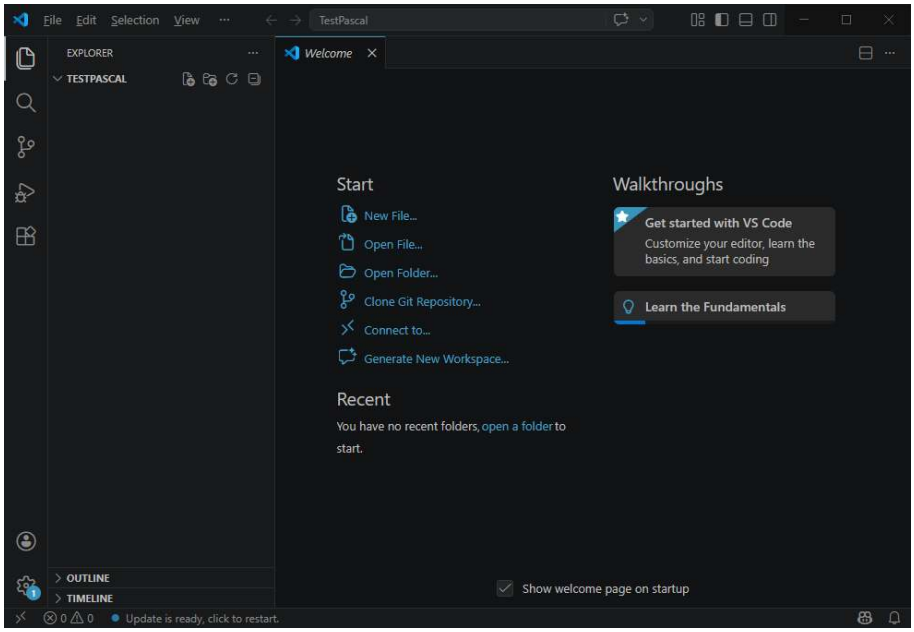



Рис. 3.2: Visual Studio Code при відкритті пустої папки

Зліва ви бачите менеджер файлів: у ньому можна переглядати всі файли з папки, яку ви відкрили. Зараз папка пуста, але спробуйте створити новий файл і назвіть його `hello.pas`. Відкрийте його, і наберіть наступний текст:

**Pascal** `hello.pas`

```pascal
program hello;

begin
  {вивести текст 'Hello, world' на екран}
  WriteLn('Hello, world');
end.
```

Збережіть файл, і натисніть комбінацію клавіш Ctrl+~ (тобто клавішу Ctrl і клавішу, що зазвичай знаходиться під кнопкою Esc). На багатьох клавіатурах це та сама клавіша, де намальована українська гривня або російська літера «Ё». Відкриється термінал прямо в IDE — не потрібно перемикатись на окреме вікно.

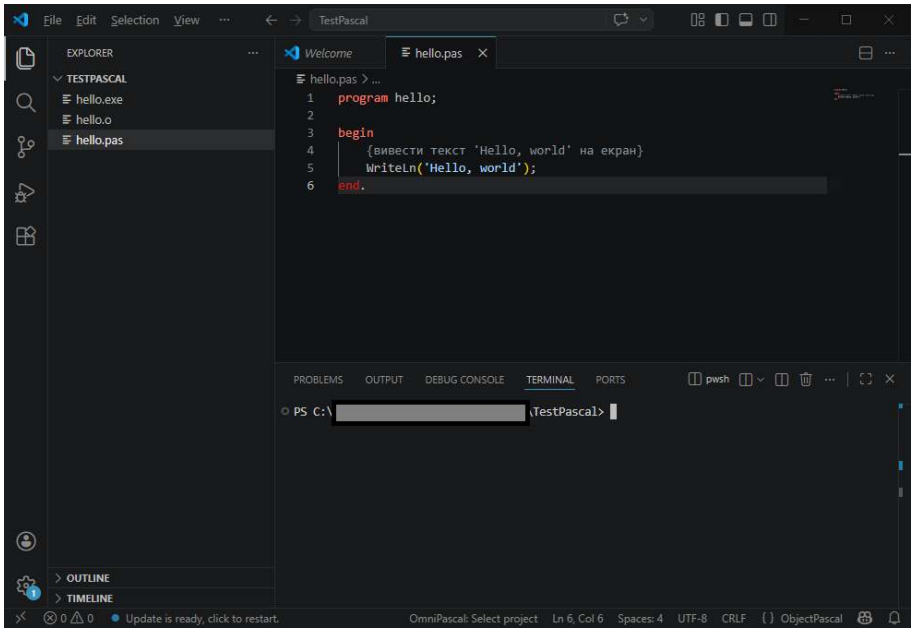



Рис. 3.3: Visual Studio Code із емулятором термінала

Спочатку скористуємся емулятором термінала, щоб скомпілювати нашу першу програму. Впишіть таку команду і запустіть її:

**Shell**

```
fpc hello.pas
```

Зверніть увагу: перед цим ви маєте перебувати у тій папці, де лежить файл `hello.pas`. VS Code зазвичай відкриває термінал у папці, де лежить файл, який ви відкрили, але є команда в терміналі, яка дозволяє перейти до будь-якої папки:

**Shell**

```
cd шлях/до/папки
```

Якщо компіляція пройшла без помилок, поруч із `hello.pas` з'явиться **виконуваний файл** — готова програма, яку можна запустити. На Windows введіть команду:

CMD

```
.\hello.exe
```

В ОС на базі Linux чи macOS команда виглядає так:

Shell

```
./hello
```

Якщо ви побачили в терміналі очікуване `Hello, world`, то ви налаштували все правильно, і ви повністю готові до нових звершень у світі програмування!

## 3.4 Структура програми

Ви написали свою першу програму, проте давайте ми розглянемо її уважніше.

Pascal hello.pas

```
program hello;

begin
  {вивести текст 'Hello, world' на екран}
  WriteLn('Hello, world');
end.
```

Програма тут має назву `hello` — її задає заголовок `program hello;`. Її задача — вивести текст на екран. Під час запуску виконуються оператори між `begin` і `end.`: це **головний блок** програми, тобто послідовність команд, які комп'ютер виконує одну за одною.

Слово `program` відкриває програму й задає її ім'я (далі можуть бути оголошення змінних тощо — про це пізніше). Пара `begin` ... `end.` обмежує **початок і кінець** цієї виконуваної частини — саме тут записано ваш алгоритм.

Ми, наприклад, можемо додати ще одну команду в наш алгоритм, і

тепер програма виведе текст двічі:

Pascal hello.pas

```pascal
program hello;

begin
  {вивести текст 'Hello, world' на екран}
  WriteLn('Hello, world');

  {вивести текст 'Goodbye, world' на екран}
  WriteLn('Goodbye, world');
end.
```

Окрім тексту, у Pascal можна виводити й числа:

Pascal printNumbers.pas

```pascal
program printNumbers;

begin
  WriteLn(4);
  WriteLn(3.5);

  {можна робити навіть так}
  WriteLn('Numbers of the week: ', 12, ' ', 34, ' ', 65, ' ', 3.14);
end.
```

**Примітка**

Дійсні числа (з **десятковою крапкою**) при виводі через `WriteLn` інколи з'являються не у звичному вигляді `123.45`, а в **експоненційній формі** — на кшталт `3.1E+003` або `1.5E-4`. Літера `E` означає «помножити на 10 у степені, записаному після знака»: `E+003` відповідає множенню на $10^3$, `E-4` — на $10^{-4}$. Наприклад, `3.1E+003` — це те саме число, що й `3100.0`; `1.5E-4` — дуже мале число порядку 0.00015. Так

комп'ютер коротко записує дуже великі або дуже малі значення; **це не означає, що програма зламалась**. Якщо потрібен саме «звичний» напис (фіксована кількість знаків після крапки), пізніше можна скористатись форматуванням рядка (наприклад, `Format` у Free Pascal).

**Примітка**

Текст, що знаходиться між фігурними дужками (`{ ... }`) в Pascal — це коментарі. Коментарі ніяким чином не впливають на виконання програми і ігноруються компілятором. Коментарі слід писати в тих місцях коду, що потребують додаткового пояснення для тих, хто буде читати ваш код. І, чесно кажучи, для себе в майбутньому теж.

`WriteLn` — це стандартна **процедура** (процедури й функції розглянемо трохи пізніше); вона виводить значення в консоль (вікно термінала) і переходить на новий рядок.

Текст для виводу треба брати в **одинарні лапки** — тоді Pascal розуміє, що це **рядок**, тобто готовий напис, а не частина «мови програмування». Без лапок ті самі літери компілятор намагатиметься прочитати по-іншому (як ім'я змінної тощо) — і часто з'явиться помилка або несподіваний результат.

Як видно з прикладу `printNumbers.pas`, у одному виклику `WriteLn` можна перелічити кілька значень через кому: і рядки, і числа виводяться поспіль в один рядок.

## 3.5 Арифметичні операції

Pascal дозволяє виконувати різноманітні арифметичні операції:

Pascal arithmetic.pas

```
program arithmetic;
```

```
begin
  WriteLn('14 + 17 --> ', 14 + 17);                  {31}
  WriteLn('14 - 17 --> ', 14 - 17);                  {-3}
  WriteLn('14 * 17 --> ', 14 * 17);                  {238}
  WriteLn('14 + 17 * 2 --> ', 14 + 17 * 2);          {48}
  WriteLn('(14 + 17) * 2 --> ', (14 + 17) * 2);      {62}

  WriteLn('1.4 + 1.7 --> ', 1.4 + 1.7);              {3.1}
  WriteLn('1.4 - 1.7 --> ', 1.4 - 1.7);              {-0.3}
  WriteLn('1.4 * 1.7 --> ', 1.4 * 1.7);              {2.38}
  WriteLn('1.4 / 1.7 --> ', 1.4 / 1.7);              {0.824}
  WriteLn('1.4 + 1.7 * 2 --> ', 1.4 + 1.7 * 2);      {4.8}
  WriteLn('(1.4 + 1.7) * 2 --> ', (1.4 + 1.7) * 2); {6.2}
end.
```

Зверніть увагу, що Pascal зберігає правильний порядок математичних операцій: спочатку виконуються операції множення чи ділення, а тільки потім — операції додавання і віднімання. За допомогою дужок порядок операцій можна змінити — все як в математиці.

## 3.6 Булева логіка

Крім числових і текстових значень, Pascal підтримує ще і так звані **логічні**, або **булеві** значення. Існують лише два булеві значення — `true` і `false`, які означають «істинне твердження» і «хибне твердження» відповідно.

Розглянемо, які операції над ними можна виконувати:

Pascal booleanLogic.pas

```
program booleanLogic;

begin
  WriteLn('true --> ', true);                        {true}
```

```
  WriteLn('false --> ', false);                          {false}

  {Операція логічного заперечення НІ (not)}
  WriteLn('not true --> ', not true);                    {false}
  WriteLn('not false --> ', not false);                  {true}

  {Операція логічного І (and)}
  WriteLn('true and true --> ', true and true);          {true}
  WriteLn('true and false --> ', true and false);        {false}
  WriteLn('false and true --> ', false and true);        {false}
  WriteLn('false and false --> ', false and false);      {false}

  {Операція логічного АБО (or)}
  WriteLn('true or true --> ', true or true);            {true}
  WriteLn('true or false --> ', true or false);          {true}
  WriteLn('false or true --> ', false or true);          {true}
  WriteLn('false or false --> ', false or false);        {false}
end.
```

## 3.7 Порівняння чисел

Оператори **відношення** (їх часто називають операторами **порівняння**) утворюють вираз, значенням якого є булева величина — `true` чи `false`, як у попередньому розділі. До них належать **рівність** (`=`), **нерівність** (`<>`), а також **менше** (`<`), **більше** (`>`), **менше або дорівнює** (`<=`) і **більше або дорівнює** (`>=`).

У Pascal символ `=` у **виразі** означає перевірку «чи рівні значення?» Оператор `<>` читають як «не дорівнює»: результат `true`, якщо значення *різні*, і `false`, якщо вони *однакові*. Оператори `<` і `>` відповідають звичним математичним «менше» та «більше»; `<=` та `>=` дозволяють і **рівність** лівого і правого значення: наприклад, `10 <= 10` дає `true`.

Pascal relationalOperators.pas

```pascal
program relationalOperators;

begin
  {Оператори Рівність / Нерівність}
  WriteLn('42 = 42  --> ', 42 = 42);  {рівні --- true}
  WriteLn('42 = 24  --> ', 42 = 24);  {не рівні --- false}
  WriteLn('42 <> 42 --> ', 42 <> 42); {рівні --- для <> це false}
  WriteLn('42 <> 24 --> ', 42 <> 24); {не рівні --- для <> це true}

  {Оператори Менше / Більше}
  WriteLn('10 < 20  --> ', 10 < 20);  {true}
  WriteLn('10 < 10  --> ', 10 < 10);  {false}
  WriteLn('20 > 10  --> ', 20 > 10);  {true}
  WriteLn('10 > 10  --> ', 10 > 10);  {false}

  {Оператори Менше / Більше або Дорівнює}
  WriteLn('10 <= 20 --> ', 10 <= 20); {true}
  WriteLn('10 <= 10 --> ', 10 <= 10); {true}
  WriteLn('10 <= 9  --> ', 10 <= 9);  {false}
  WriteLn('20 >= 10 --> ', 20 >= 10); {true}
  WriteLn('10 >= 10 --> ', 10 >= 10); {true}
  WriteLn('9 >= 10  --> ', 9 >= 10);  {false}
end.
```

## 3.8 Константи

Ну що ж, давайте напишемо програму, яка виконує реальну задачу?

Отож задача така: потрібно написати програму, яка рахує довжину кола, площу круга та об'єм кулі, якщо радіус дорівнює 5.

Нагадаю, що довжина кола дорівнює $L = 2\pi R$, площа круга — $S = \pi R^2$, а об'єм кулі — $V = \frac{4}{3}\pi R^3$.

Тобто код виглядатиме якось так:

Pascal circle.pas

```
program circle;

begin
  {довжина кола}
  WriteLn(2.0 * 5.0 * 3.1415);

  {площа круга}
  WriteLn(5.0 * 5.0 * 3.1415);

  {об'єм кулі}
  WriteLn((4.0 / 3.0) * 5.0 * 5.0 * 5.0 * 3.1415);
end.
```

Формально, задачу цей код розв'язує, але, якщо подумати, він жахливий. Чому так?

Число 5.0 зустрічається в цьому коді шість разів! Якщо вимоги зміняться і треба буде рахувати, скажімо, не для радіуса 5, а для 4, — доведеться шукати й замінювати кожне з цих входжень. І це лише в програмі на сім рядків коду. Уявіть програму на сотні або тисячі рядків: легко пропустити одне місце, переплутати 5 із коефіцієнтом $\frac{4}{3}$ у формулі об'єму кулі $\frac{4}{3}\pi r^3$ або помилитись у кількості множників $r$.

Як відповідальні програмісти, ми повинні давати числам осмислені імена. Давайте спробуємо це зробити.

Pascal circle.pas

```
program circle;

const
  pi: single = 3.1415;
  radius: single = 5.0;
```

```
begin
  {довжина кола}
  WriteLn(2 * radius * pi);

  {площа круга}
  WriteLn(radius * radius * pi);

  {об'єм кулі}
  WriteLn((4.0 / 3.0) * radius * radius * radius * pi);
end.
```

Погодьтесь, так **набагато** зрозуміліше. Числа 2 і $\frac{4}{3}$ ми лишили в коді як є: це невід'ємна частина самих формул, а не «дані задачі», як радіус чи $\pi$. (Пізніше і це можна оформити охайніше.) Поки що звернімо увагу на те, що саме ми змінили.

Ми **оголосили константи** — імена для значень, які в цій програмі не змінюються під час роботи. Тут таких констант дві: `pi` для наближення $\pi$ і `radius` для радіуса.

Головна зручність — змінити радіус або точність $\pi$ можна **в одному місці** (у блоці `const`), а не шукати однакові числа по всьому коду. Додали знаків після коми для $\pi$ — оновилось усюди; змінили радіус у задачі — знову ж таки одна правка.

Зверніть увагу на слово `single` біля назв цих констант. Це — їх **тип даних**.

> **Визначення**
>
> **Тип даних** (англ. *Data type*) — це спосіб організації та представлення даних, що визначає їхні властивості та правила використання.

Тип даних `single` поки що можемо просто сприймати як тип даних для **дійсних чисел**. Це дуже грубе спрощення, але поки що цього нам буде достатньо. Для цілих чисел будемо використовувати тип `LongInt`, для

булевих значень — `boolean`.

**Примітка**

Ось мінімальний приклад константи цілого типу (поруч із уже знайомим вам блоком `const`):

Pascal

```
const
  classSize: LongInt = 30;
  goRight: boolean = true;
```

## 3.9 Функції і процедури

Ускладнимо нашу задачу. Тепер наша задача — вивести довжину кола, площу круга і об'єм кулі для трьох радіусів 5, 8, 42.

Ну що ж, пишемо рішення задачі:

Pascal circle.pas

```
program circle;

const
  pi: single = 3.1415;
  radius1: single = 5.0;
  radius2: single = 8.0;
  radius3: single = 42.0;

begin
  {перший радіус}
  WriteLn(2 * radius1 * pi);
  WriteLn(radius1 * radius1 * pi);
  WriteLn((4.0 / 3.0) * radius1 * radius1 * radius1 * pi);
```

```
  {другий радіус}
  WriteLn(2 * radius2 * pi);
  WriteLn(radius2 * radius2 * pi);
  WriteLn((4.0 / 3.0) * radius2 * radius2 * radius2 * pi);

  {третій радіус}
  WriteLn(2 * radius3 * pi);
  WriteLn(radius3 * radius3 * pi);
  WriteLn((4.0 / 3.0) * radius3 * radius3 * radius3 * pi);
end.
```

І ми знову написали жахливий код. Тільки цього разу не через те, що ми повторюємо **числа**, а через те, що ми повторюємо **поведінку**. Зверніть увагу: ми тричі пишемо формулу довжини кола, площі круга і об'єму кулі. Це — проблема, адже при розростанні програми ми знову ж можемо легко помилитися при наборі коду.

Для вирішення цієї проблеми створені функції. Давайте поглянемо, як програма виглядатиме з ними:

Pascal circle.pas

```
program circle;

const
  pi: single = 3.1415;
  radius1: single = 5.0;
  radius2: single = 8.0;
  radius3: single = 42.0;

function GetCircleLength(const radius: single): single;
begin
  GetCircleLength := 2 * radius * pi;
end;
```

```
function GetCircleArea(const radius: single): single;
begin
  GetCircleArea := radius * radius * pi;
end;

function GetBallVolume(const radius: single): single;
begin
  GetBallVolume := (4.0 / 3.0) * radius * radius * radius * pi;
end;

begin
  {перший радіус}
  WriteLn(GetCircleLength(radius1));
  WriteLn(GetCircleArea(radius1));
  WriteLn(GetBallVolume(radius1));

  {другий радіус}
  WriteLn(GetCircleLength(radius2));
  WriteLn(GetCircleArea(radius2));
  WriteLn(GetBallVolume(radius2));

  {третій радіус}
  WriteLn(GetCircleLength(radius3));
  WriteLn(GetCircleArea(radius3));
  WriteLn(GetBallVolume(radius3));
end.
```

На перший погляд здається, ніби програма виросла. Але насправді її стало значно легше читати і редагувати. До того ж, завдяки функціям у нас є лише одна точка в коді, яка рахує кожну формулу.

Давайте детальніше розглянемо синтаксис функцій. Функції приймають `radius`, який має бути дійсним числом, і мають повернути дійсне значення.

У функції виконуються всі команди, що знаходяться між `begin` і `end;`.

А значення функції має записуватись в спеціальну **змінну** (детальніше змінні будуть розглянуті згодом), назва якої така ж, як і назва функції. Для запису в змінну використовується оператор присвоєння значення `:=`.

Дивлячись на цей код, можна помітити, що він все ще не ідеальний: **логіка виведення інформації про кожний радіус все ще повторюється**! Нам потрібно створити ще одну функцію, яка виводить в консоль інформацію про фігури із вказаним радіусом. Функція, яка виводитиме цю інформацію, має прийняти `radius` у якості параметра, але їй нічого повертати. Якщо така функція нічого не повертає, її називають **процедурою**. Давайте спробуємо створити процедуру для виведення інформації про фігури із вказаним радіусом:

Pascal circle.pas

```pascal
program circle;

const
  pi: single = 3.1415;
  radius1: single = 5.0;
  radius2: single = 8.0;
  radius3: single = 42.0;

function GetCircleLength(const radius: single): single;
begin
  GetCircleLength := 2 * radius * pi;
end;

function GetCircleArea(const radius: single): single;
begin
  GetCircleArea := radius * radius * pi;
end;

function GetBallVolume(const radius: single): single;
begin
```

```
  GetBallVolume := (4.0 / 3.0) * radius * radius * radius * pi;
end;

procedure PrintInfoAboutShapesWithRadius(const radius: single);
begin
  WriteLn(GetCircleLength(radius));
  WriteLn(GetCircleArea(radius));
  WriteLn(GetBallVolume(radius));
end;

begin
  PrintInfoAboutShapesWithRadius(radius1);
  PrintInfoAboutShapesWithRadius(radius2);
  PrintInfoAboutShapesWithRadius(radius3);
end.
```

Процедури й функції — головний засіб прибрати **повтори** в коді. Якщо у вашому коді багато повторів, один і той самий фрагмент у кількох місцях доводиться змінювати окремо, легко щось пропустити або зробити по-різному — і саме звідти часто беруться помилки.

Також варто добирати **імена**, які вже пояснюють, що робить процедура чи функція: по назві виклику читач розуміє намір без довгих пояснень.

Одна ясна, охайно оформлена функція або процедура з вдалою назвою часто виявляється кориснішою за багато рядків коментарів, хоч би наскільки ретельно ті коментарі описували код. Пам'ятайте про це, коли пишете програму: спершу про структуру та імена, а не лише про текст у фігурних дужках.

**Примітка**

Сучасні мови програмування майже не використовують термін «процедура». Такі мови, як C, C++, Java, C#, Rust, Zig тощо, називають процедури і функції просто «функціями». Також в об'єктно-орієнтованій парадигмі часто застосовують термін «метод».

**Визначення**

**Підпрограма** (англ. *Subroutine*) — частина програми, яка реалізує певний алгоритм і дозволяє звернення до неї з різних частин загальної (головної) програми. В мові програмування Pascal і процедури, і функції — це підпрограми.

## 3.10 Модулі (частина 1)

Нехай у нас є нова задача. Дано координати напрямку $(x, y)$, і потрібно визначити кут, який цей напрямок утворює з позитивним напрямком осі $x$, у градусах.

Для цього існує функція $\arctan2(y, x)$, яка повертає кут у радіанах для точки $(x, y)$ відносно початку координат. На відміну від звичайного арктангенса, вона правильно визначає кут у всіх чотирьох квадрантах. Щоб перевести радіани в градуси, помножимо на $\frac{180}{\pi}$:

$$\theta = \arctan2(y, x) \cdot \frac{180}{\pi}$$

Спробуємо це запрограмувати:

Pascal directionAngle.pas

```pascal
program directionAngle;

function GetAngleInDegrees(
  const x: single;
  const y: single): single;
begin
  GetAngleInDegrees := arctan2(y, x) * 180.0 / pi;
end;

begin
  WriteLn(GetAngleInDegrees(1, 0));
```

```
  WriteLn(GetAngleInDegrees(0, 1));
  WriteLn(GetAngleInDegrees(-1, 0));
end.
```

Але ця програма не скомпілюється! Справа в тому, що функція `arctan2` не є вбудованою у мову Pascal. Вона знаходиться у **модулі** під назвою `math`. В мові Pascal модулі — це просто набір функцій, змінних, типів даних, процедур, констант. Ідея модулів полягає у тому, що ми як програмісти хочемо логічно **організувати** код нашої програми. Тому ми розділяємо програму на файли: кожен файл повинен відповідати за свої задачі.

Наш рейкаст-рушій теж буде розділений по модулях: буде модуль, відповідальний за вектори, модуль, відповідальний за промені, за карту тощо.

Модуль `math` відповідає за різні «математичні штуки». Він надає нам можливість користуватись популярними математичними функціями (в тому числі тригонометричними), і все це об'єднано разом.

Для підключення модуля використовується ключове слово `uses`. Виправимо нашу програму:

Pascal directionAngle.pas

```
program directionAngle;

uses math;

function GetAngleInDegrees(
  const x: single;
  const y: single): single;
begin
  GetAngleInDegrees := arctan2(y, x) * 180.0 / pi;
end;

begin
  WriteLn(GetAngleInDegrees(1, 0));   {0 градусів}
  WriteLn(GetAngleInDegrees(0, 1));   {90 градусів}
```

```
  WriteLn(GetAngleInDegrees(-1, 0));  {180 градусів}
end.
```

Нам вдалось! Ми успішно підключили модуль `math` і використали його функцію `arctan2` для обчислення кута напрямку. Зверніть також увагу, що константу `pi` ми тут **не оголошували самі**: модуль `math` надає свою константу `pi` з високою точністю, тому окремий `const` нам більше не потрібен.

**Примітка**

Англійською модулі мови Pascal називаються **Units**. Слово «Модуль» — найкращий переклад для цього поняття, який я зміг знайти, але він не ідельний, тому що він вносить деяку «плутанину» в термінах.

По-перше, модулі в Pascal не мають нічого спільного із математичним модулем $|a|$. У Pascal є функція, що повертає математичний модуль числа: `abs`. Назва цієї функції походить від англійського виразу «absolute value» (укр. *Абсолютне значення*).

По-друге, в мові Pascal є оператор `mod`, який позначає остачу від ділення. Вираз `5 mod 2` повертає `1`, а вираз `(5 mod 2) = 1` повертає `true`. Назва цього оператора походить від англійського слова `modulo`.

По-третє, Ніклаус Вірт (автор мови Pascal), створив ще три мови програмування: **Modula**, **Modula-2** і **Modula-3**.

Загалом, спробуйте не заплутатись у всіх цих поняттях.

## 3.11 Змінні

Розглянемо ще одну задачу. Нам потрібно знайти площу трикутника зі сторонами 5, 6 і 7. Для цього скористаємось формулою Герона:

$$S_{\triangle} = \sqrt{p(p-a)(p-b)(p-c)}$$

У цій формулі $a$, $b$, $c$ — довжини сторін трикутника, а $p$ — половина

його периметра:

$$p = \frac{a + b + c}{2}$$

Спробуємо написати відповідну функцію:

Pascal triangles.pas

```
program triangles;

function GetTriangleArea(
  const a: single;
  const b: single;
  const c: single): single;
begin
  {Рахуємо площу трикутника формулою Герона}
  GetTriangleArea :=
    sqrt(
      ((a + b + c) / 2.0) *
      ((a + b + c) / 2.0 - a) *
      ((a + b + c) / 2.0 - b) *
      ((a + b + c) / 2.0 - c)
    );
end;

begin
  WriteLn(GetTriangleArea(5, 6, 7))
end.
```

Зверніть увагу на коментар у цьому коді. Він — корисний, тому що він описує метод, яким ми рахуємо площу. Людина, яка читає цей код, у разі якихось питань просто «загуглить» поняття «формула Герона» для того, щоб дізнатись про цей метод детальніше.

**Примітка**

Функція `sqrt` повертає квадратний корінь з числа. На відміну від інших математичних функцій, вона доступна без підключення додаткових модулів типу `math`.

Але із цим кодом щось не так: у нас повтори — ми чотири рази рахуємо півпериметр. Було б добре, якби ми його просто порахували один раз і використовували уже пораховане значення. Для цього використовуються змінні.

**Визначення**

**Змінна** (англ. *Variable*) — це комірка, у якій зберігається значення. Значення змінної може мінятись під час виконання програми.

Давайте спробуємо створити змінну, яка «житиме» впродовж виконання функції `GetTriangleArea`, і триматиме значення половини периметра трикутника:

Pascal triangles.pas

```pascal
program triangles;

function GetTriangleArea(
  const a: single;
  const b: single;
  const c: single): single;
var
  halfPerimeter: single;
begin
  halfPerimeter := (a + b + c) / 2.0;

  {Рахуємо площу трикутника формулою Герона}
  GetTriangleArea :=
    sqrt(
```

```
      halfPerimeter *
      (halfPerimeter - a) *
      (halfPerimeter - b) *
      (halfPerimeter - c)
    );
end;

begin
  WriteLn(GetTriangleArea(5, 6, 7))
end.
```

**Запитання 7**

Як ви вважаєте, чому ми не можемо зробити `halfPerimeter` константою?

## 3.12 Потік керування програми

Для розуміння концепції потоку виконання програми, давайте розглянемо такий приклад. Нам потрібно розробити програму, яка рахуватиме **найбільший спільний дільник** (англ. *Greatest Common Divisor* або *GCD*) двох чисел.

Алгоритм Евкліда для знаходження найбільшого спільного дільника двох чисел $a$ і $b$ можна описати так. Нехай $a$ і $b$ — натуральні числа.

1. Якщо $a = b$, то це спільне значення і є **НСД** — на цьому алгоритм закінчується.
2. Якщо $a > b$, замінюємо $a$ на різницю $a - b$ (від більшого віднімаємо менше).
3. Якщо ж $b > a$, замінюємо $b$ на різницю $b - a$.
4. Переходимо до кроку 1 і повторюємо, доки числа не стануть рівними.

Ідея в тому, що **найбільший спільний дільник пари чисел не змі-**

**нюється**, якщо від більшого відняти менше: НСД$(a, b) =$ НСД$(a-b, b)$ при $a > b$ і симетрично для $b > a$. Тому поступово пара $(a, b)$ «стискається», поки обидва числа не збіжуться — тоді це і є шуканий НСД.

Давайте спробуємо написати програму, яка реалізує цей алгоритм:

Pascal gcdGoto.pas

```
program gcdGoto;

function Gcd(
  const a: LongInt;
  const b: LongInt): LongInt;
var
  aTemp: LongInt;
  bTemp: LongInt;
label
  LoopStart, AGreater, BGreater, Done;
begin
  aTemp := a;
  bTemp := b;

  {Застосовуємо метод Евкліда}
  LoopStart:
    if aTemp = bTemp then goto Done;
    if aTemp > bTemp then goto AGreater;
    goto BGreater;
  AGreater:
    aTemp := aTemp - bTemp;
    goto LoopStart;
  BGreater:
    bTemp := bTemp - aTemp;
    goto LoopStart;
  Done:
  Gcd := aTemp;
```

```
end;

begin
  WriteLn('GCD = ', Gcd(1071, 462));
end.
```

Отож, ми створили наш алгоритм. НСД успішно рахується і виводиться в консоль. Завдяки нашій програмі ми дізнались, що НСД$(1071, 462) = 21$.

**Увага**

Зверніть увагу, що хоч НСД ми рахуємо лише один раз, але все одно створили функцію, `Gcd`. Чому ми так зробили? У першу чергу, тому, що `Gcd` ми в майбутньому **можемо** перевикористовувати, але є ще одна не менш важлива причина. Підпрограми не повинні брати на себе багато відповідальності!

Щоб зрозуміти це, давайте подумаємо, якою була наша задача. Порахувати і вивести НСД двох чисел. Підрахунок даних і їх виведення в консоль — це **дві різні задачі**.

**Хорошою практикою є розділення програми на такі підпрограми, що беруть на себе по одному зобов'язанню і виконують тільки ті задачі, що описані в їх назві.**

Давайте подивимось детальніше на наш код. `LoopStart`, `AGreater`, `BGreater`, `Done` у ньому — це **мітки**. Мітки — це спеціальні точки в коді програми, до яких ми можемо «перескочити»; оголошуються вони схожим чином, як і змінні, тільки ключовим словом `label`. А для того, щоб «стрибнути» до мітки, можна використати команду `goto`.

Конструкція `if ··· then` задає **умову**. Ідея полягає в тому, що команда справа від `then` виконається лише тоді, якщо **булеве** значення між `if` і `then` дорівнює `true`. Тобто рядок коду `if aTemp = bTemp then goto Done;` означає «якщо змінна `aTemp` дорівнює змінній `bTemp`, то ми переходимо до точки `Done`, у іншому випадку ми цього не робимо».

Цей код цікавий тим, що він описує алгоритм **буквально** неначе ми

його описали словами: є команди, і є опис того, куди за яких умов «стрибнути».

Проблема полягає у тому, що цей код поганий ...

Уявіть, що у вашому коді — тисячі таких міток. Ви дуже швидко заплутаєтесь у ньому! Немає гарантії, що ви не заплутались навіть у коді вище! Мітки дають свободу переходити куди завгодно в функції, і код, який використовує `goto`-переходи, **дуже швидко перетворюється на спагетті**, тобто на код, який настільки заплутаний, що ви не можете навіть знайти його початку і кінця!

Цю проблему програмісти помітили не сьогодні, і навіть не вчора. Ще у 1968 році програміст і вчений Едсгер Дійкстра написав статтю-лист «Go To Statement Considered Harmful», у якій розкритикував використання команди `goto`, і закликав її не використовувати.

Замість цього він пояснив, що значно кращий і підтримуваний код вийде, якщо використовувати всього три структури керування: послідовне виконання підпрограм, умови, цикли (ми зараз дізнаємось, що це все таке).

У своїй роботі Дійкстра посилався на **Структурну теорему Бьома—Якопіні** (*Böhm—Jacopini*), про яку можна прочитати по посиланню:

https://uk.wikipedia.org/wiki/Структурна_теорема_Бьома_—_Якопіні

Давайте спробуємо переписати нашу програму відповідним чином:

Pascal gcdStructured.pas

```pascal
program gcdStructured;

function Gcd(
  const a: LongInt;
  const b: LongInt): LongInt;
var
  aTemp: LongInt;
  bTemp: LongInt;
```

```
begin
  aTemp := a;
  bTemp := b;

  while aTemp <> bTemp do
    if aTemp > bTemp then
      aTemp := aTemp - bTemp
    else
      bTemp := bTemp - aTemp;

  Gcd := aTemp;
end;

begin
  WriteLn('GCD = ', Gcd(1071, 462));
end.
```

Ми написали код без жодної мітки! І, знаючи, як працюють `else` і `while`, ви зможете значно легше такий код прочитати! Розглянемо, що у ньому відбувається.

Цикл `while умова do ...` знову й знову виконує **тіло циклу** (те, що йде після `do`), доки **умова** лишається істинною. Коли вона стає хибною, програма переходить до наступного рядка *після* циклу — у нашому випадку до присвоєння `Gcd := aTemp`. У версії з `goto` те саме «обертання» давали перехід `goto LoopStart`: ми повторювали блок віднімань, поки `aTemp` і `bTemp` не стали рівними. `while` говорить про це прямо: «повторюй, доки числа не рівні», без стрибків по мітках.

Конструкція `if ··· then ··· else ···` читається так: якщо умова після `if` істинна, виконується гілка `then`; якщо хибна — виконується гілка `else`. (Якщо `else` немає, при хибній умові нічого не відбувається — як у ранніх прикладах із `goto`.) Тут `else` відповідає другому випадку порівняння `aTemp` і `bTemp`: саме його в попередньому коді ми обходили через мітку `BGreater` і `goto` після перевірки `aTemp > bTemp`.

**Увага**

Маючи на озброєнні підпрограми, умови і цикли, у більшості випадків можна обійтися без `goto`. Коли є альтернатива у вигляді умови чи циклу, краще обирати саме її: такий код простіше читати, перевіряти і підтримувати.

## 3.13 Записи

Спробуємо розв'язати ще одну задачу. Дані три вектори $\overrightarrow{a}$, $\overrightarrow{b}$, $\overrightarrow{c}$. Знаючи їхні координати, нам потрібно знайти $\|\overrightarrow{a} + \overrightarrow{b} + \overrightarrow{c}\|$.

Будучи відповідальними програмістами, ми вже знаємо, що нам точно потрібні будуть підпрограми для додавання векторів і для пошуку довжини вектора. Але вектор — це два числа! Як написати функцію, яка повертає два числа? Ми вміємо повертати лише одне значення!

Для таких, і не тільки, цілей існує концепція **записів** (англ. *Record*). Запис — це об'єднання кількох елементів в один тип даних. Як це працює у випадку із векторами? Ми кажемо «нам треба об'єднати два числа із типом даних `single`, перше із них — це $x$, а друге — це $y$; це об'єднання називатиметься вектором».

Давайте спробуємо вирішити нашу задачу із використанням записів:

Pascal vec3mag.pas

```pascal
program vec3mag;

type
  TVector = record
    x: single;
    y: single;
  end;

function VectorCartesian(
```

```
  const x: single;
  const y: single
): TVector;
begin
  VectorCartesian.x := x;
  VectorCartesian.y := y;
end;

function VectorAdd(
  const lhs: TVector;
  const rhs: TVector
): TVector;
begin
  VectorAdd.x := lhs.x + rhs.x;
  VectorAdd.y := lhs.y + rhs.y;
end;

function VectorMagnitude(const vec: TVector): single;
begin
  VectorMagnitude := sqrt(vec.x * vec.x + vec.y * vec.y);
end;

function Sum3Magnitude(
  const a: TVector;
  const b: TVector;
  const c: TVector
): single;
var
  sum: TVector;
begin
  sum := VectorAdd(a, VectorAdd(b, c));
  Sum3Magnitude := VectorMagnitude(sum);
end;
```

```
var a: TVector;
var b: TVector;
var c: TVector;

begin
  a := VectorCartesian(10, 20);
  b := VectorCartesian(13, 65);
  c := VectorCartesian(1.2, 56);

  WriteLn('Result = ', Sum3Magnitude(a, b, c));
end.
```

Ключове тут — **власний тип** `TVector`. У блоці `type` ми описали, з чого він складається: два поля `x` і `y` типу `single`. Далі `TVector` можна використовувати так само, як `LongInt` чи `single`: оголошувати змінні (`var a: TVector`), передавати в функції й повертати з функцій.

Функція `VectorCartesian` — це зручний «конструктор»: з двох чисел вона збирає один вектор-запис.

`VectorAdd` приймає два вектори й повертає новий вектор: координати складаються окремо ($x$ з $x$, $y$ з $y$). `VectorMagnitude` навпаки повертає одне число — довжину, використовуючи вбудовану функцію `sqrt`.

Нарешті, `Sum3Magnitude` показує, як з типами «складати» програму шарами: спочатку `VectorAdd(b, c)` дає суму двох векторів, потім `VectorAdd(a, ...)` додає третій, і вже для підсумкового `sum` викликається `VectorMagnitude`. У головній програмі три вектори створюються через `VectorCartesian`, а далі вся «геометрія» ховається за викликом однієї функції — читач одразу бачить головну думку: знайти довжину суми трьох векторів.

## 3.14 Масиви

Ускладнимо нашу задачу. Нехай нам потрібно порахувати модуль суми не трьох, а **десяти** векторів.

На перший погляд здається, що для цього варто уже тримати десять змінних — `a, b, c, d, e, ...`. А для того, щоб рахувати суму, нам довелось би викликати функцію `VectorAdd` цілих дев'ять разів. Якось так:

Pascal

```
sum := VectorAdd(
  a,
  VectorAdd(
    b,
    VectorAdd(
      c,
      VectorAdd(
        d,
        ...))));
```

І ми знову пишемо код із одноманітними повторами. Це — сигнал того, що ми робимо щось не так.

Давайте вирішимо цю проблему. Що ми знаємо про нашу задачу і наші дані? Нам потрібно знайти суму **списку векторів**. Тобто потрібні не десять окремих імен змінних, а список із десяти векторів. Для цього в Pascal використовуються **масиви** (англ. *Array*). Давайте спробуємо вирішити нашу задачу, використовуючи масиви:

Pascal vecnmag.pas

```
program vecnmag;

type
  TVector = record
    x: single;
```

```
    y: single;
  end;

function VectorCartesian(
  const x: single;
  const y: single
): TVector;
begin
  VectorCartesian.x := x;
  VectorCartesian.y := y;
end;

function VectorAdd(
  const lhs: TVector;
  const rhs: TVector
): TVector;
begin
  VectorAdd.x := lhs.x + rhs.x;
  VectorAdd.y := lhs.y + rhs.y;
end;

function VectorMagnitude(const vec: TVector): single;
begin
  VectorMagnitude := sqrt(vec.x * vec.x + vec.y * vec.y);
end;

function SumNMagnitude(const vecs: array of TVector): single;
var
  sum: TVector;
  i: LongInt;
begin
  i := Low(vecs);
  sum := VectorCartesian(0, 0);
```

```
  while i <= High(vecs) do
  begin
    sum := VectorAdd(sum, vecs[i]);
    i := i + 1;
  end;

  SumNMagnitude := VectorMagnitude(sum);
end;

var
  vecs: array[0..9] of TVector;

begin
  vecs[0] := VectorCartesian(10, 20);
  vecs[1] := VectorCartesian(13, 65);
  vecs[2] := VectorCartesian(1.2, 56);
  vecs[3] := VectorCartesian(98, 12);
  vecs[4] := VectorCartesian(5.6, -4);
  vecs[5] := VectorCartesian(1.4, 5.6);
  vecs[6] := VectorCartesian(1, 2);
  vecs[7] := VectorCartesian(1, 3);
  vecs[8] := VectorCartesian(2, 8);
  vecs[9] := VectorCartesian(10, 20);

  WriteLn('Result = ', SumNMagnitude(vecs));
end.
```

У головній програмі змінна `vecs` має тип `array[0..9] of TVector`: це **масив фіксованої довжини** — рівно десять елементів, пронумерованих від `0` до `9`. Елемент із номером `i` записується як `vecs[i]`: так ми заповнюємо масив десятьма векторами, не заводячи десять окремих імен.

У функції `SumNMagnitude` параметр оголошено інакше:

```
const vecs: array of TVector
```

Такий запис означає **відкритий масив**: функція приймає масив *будь-якої* довжини (з тими самими типами елементів) і не повинна знати заздалегідь, скільки їх буде — десять, сто чи нуль. Межі індексів у такому випадку зручно брати через `Low(vecs)` і `High(vecs)`: для `array[0..9]` це буде `0` і `9`, а якщо пізніше ви зміните розмір масиву в головній програмі, цикл залишиться правильним без правок усередині функції.

Перед циклом `sum` ініціалізують нульовим вектором `VectorCartesian(0, 0)` — це «нейтральний» елемент для додавання. Далі в `while` для кожного `i` до суми додається `vecs[i]`; після обходу `VectorMagnitude` рахує довжину вже повної суми.

## 3.15 Цикл for

Для того, щоб обходити масив, ми використовували конструкцію `while`, вручну збільшуючи значення змінної `i` на одиницю на кожній ітерації.

Але існує простіший спосіб — через цикл `for`. Давайте спробуємо замінити `while` на `for`.

Pascal vecnmag.pas

```
function SumNMagnitude(const vecs: array of TVector): single;
var
  sum: TVector;
  i: LongInt;
begin
  sum := VectorCartesian(0, 0);

  for i := Low(vecs) to High(vecs) do
    sum := VectorAdd(sum, vecs[i]);

  SumNMagnitude := VectorMagnitude(sum);
end;
```

Так виглядає значно охайніше! `for i := A to B do ...` означає: «послідовно взяти для `i` кожне ціле значення від `A` до `B` **включно** і кожного разу виконати команду після `do`». У нашому випадку `A` — це `Low(vecs)`, `B` — `High(vecs)`; збільшувати `i` вручну більше не потрібно.

## 3.16 Модулі (частина 2)

Розглянемо детальніше нашу програму, що рахує модуль суми векторів. У цій програмі є кілька функцій: `VectorCartesian`, `VectorAdd`, `VectorMagnitude` і `SumNMagnitude`.

Давайте подумаємо, скільки із цих функцій напряму стосуються умови нашої задачі? Якщо так подумати, то лише одна — `SumNMagnitude`. Всі інші функції визначають **типові** операції, які ми можемо виконати над векторами. Уявіть, що у вас є нова задача, пов'язана із векторами. Із надзвичайно високою ймовірністю функції `VectorCartesian`, `VectorAdd` і `VectorMagnitude` вам знадобляться. І вам що, треба тримати один і той самий код у тексті кожної задачі? Ви ж не робите такого із математичними функціями типу `sqrt`, ви користуєтесь **модулем**, який надали вам автори мови Pascal!

Чому б нам не зробити те ж саме для векторів? Давайте створимо окремий файл `vectors.pas` і перенесемо туди наші функції для роботи із векторами:

Pascal vectors.pas

```pascal
unit vectors;

interface
uses math;

type
  TVector = record
    x: single;
```

```
    y: single;
  end;

{ Формує вектор із координат x, y }
function VectorCartesian(
  const x: single;
  const y: single
): TVector;

{ Повертає суму двох векторів }
function VectorAdd(
  const lhs: TVector;
  const rhs: TVector
): TVector;

{ Рахує довжину вектора }
function VectorMagnitude(const vec: TVector): single;

implementation

function VectorCartesian(
  const x: single;
  const y: single
): TVector;
begin
  VectorCartesian.x := x;
  VectorCartesian.y := y;
end;

function VectorAdd(
  const lhs: TVector;
  const rhs: TVector
): TVector;
```

```
begin
  VectorAdd.x := lhs.x + rhs.x;
  VectorAdd.y := lhs.y + rhs.y;
end;

function VectorMagnitude(const vec: TVector): single;
begin
  VectorMagnitude := sqrt(vec.x * vec.x + vec.y * vec.y);
end;

end.
```

Файл починається з ключового слова `unit`: так ми даємо модулю ім'я (`vectors`). Далі йдуть два великі блоки — `interface` і `implementation`.

У **інтерфейсі** ми оголошуємо *те, чим модуль ділиться з іншими частинами програми*: підключення `uses math;` (для тригонометричних функцій, які знадобляться у повній реалізації модуля), тип `TVector` і **заголовки** функцій — сигнатури з типами параметрів і типом результату, але *без* тіл функцій. Це схоже на зміст: читач одразу бачить, що можна викликати, не занурюючись у реалізацію. Короткі коментарі в фігурних дужках нагадують, що робить кожна функція.

У блоці `implementation` записано вже *повні* визначення — те саме, що ми раніше писали в одному файлі: `begin` ... `end;` для кожної функції. Компілятор звіряє їх із оголошеннями з інтерфейсу: імена, параметри й типи повинні збігатися.

Останній рядок `end.` (з крапкою) завершує весь модуль. Далі головна програма або інший модуль зможе написати `uses vectors;` і користуватися `TVector` та функціями, не копіюючи їхній текст знову й знову. Давайте, власне, спробуємо це і зробити із нашою основною програмою:

Pascal vecnmag.pas

```pascal
program vecnmag;

uses vectors;

function SumNMagnitude(
  const vecs: array of TVector
): single;
var
  sum: TVector;
  i: LongInt;
begin
  sum := VectorCartesian(0, 0);

  for i := Low(vecs) to High(vecs) do
    sum := VectorAdd(sum, vecs[i]);

  SumNMagnitude := VectorMagnitude(sum);
end;

var
  vecs: array[0..9] of TVector;

begin
  vecs[0] := VectorCartesian(10, 20);
  vecs[1] := VectorCartesian(13, 65);
  vecs[2] := VectorCartesian(1.2, 56);
  vecs[3] := VectorCartesian(98, 12);
  vecs[4] := VectorCartesian(5.6, -4);
  vecs[5] := VectorCartesian(1.4, 5.6);
  vecs[6] := VectorCartesian(1, 2);
  vecs[7] := VectorCartesian(1, 3);
  vecs[8] := VectorCartesian(2, 8);
```

```
  vecs[9] := VectorCartesian(10, 20);

  WriteLn('Result = ', SumNMagnitude(vecs));
end.
```

Ми тільки що підключили наш модуль `vectors`, і можемо користуватися типом `TVector` і всіма функціями, які ми там визначили **без дублювання коду** із будь-якого іншого файлу, який підключає цей модуль.

### Задача 15

Ми створили модуль `vectors`, але нам для нашого рушія потрібно реалізувати більше операцій, які ми можемо виконати над векторами.

Вашим завданням є повністю реалізувати модуль `vectors` (нижче — заготовка інтерфейсу).

Це перша по-справжньому важлива задача! **Бо цей код використовуватиметься у нашій грі**.

Також перевірте правильність того, як ви написали свої функції. Для цього створіть окрему програму `vectorsTest.pas`, у якій із допомогою процедури `Assert` перевірте те, чи правильно ви реалізували свої функції (можливість дослідити, як працює `Assert`, я залишаю вам).

Pascal — vectors.pas

```
unit vectors;

interface
uses math;

type
  TVector = record
    x: single;
    y: single;
  end;
```

```
{ Формує вектор із Декартових координат x, y }
function VectorCartesian(
  const x: single;
  const y: single
): TVector;

{ Формує вектор із полярних координат radius, angle }
function VectorPolar(
    const radius: single;
    const angle: single
): TVector;

{ Повертає суму двох векторів }
function VectorAdd(
  const lhs: TVector;
  const rhs: TVector
): TVector;

{ Повертає різницю двох векторів }
function VectorSub(
  const lhs: TVector;
  const rhs: TVector
): TVector;

{ Повертає результат множення вектора на число }
function VectorMulScalar(
  const vec: TVector;
  const scalar: single
): TVector;

{ Повертає результат ділення вектора на число }
function VectorDivScalar(
```

```
  const vec: TVector;
  const scalar: single
): TVector;

{ Рахує довжину вектора }
function VectorMagnitude(const vec: TVector): single;

{ Рахує квадрат довжини вектора }
function VectorMagnitudeSquared(const vec: TVector): single;

{ Повертає нормалізований вектор }
function VectorNormalize(const vec: TVector): TVector;

{ Рахує кут у радіанах відносно осі x }
function VectorAngle(const vec: TVector): single;

implementation

{TODO: Тут має бути ваша реалізація функцій}

end.
```

## 3.17 Числа

Повернемось до простих прикладів. Розглянемо дуже просту програму, і спробуємо перевірити, як вона спрацює.

Pascal integernumbers.pas

```
program integernumbers;

var n: LongInt;

begin
```

```
  n := 76;
  WriteLn(n);

  n := n + 1;
  WriteLn(n);
end.
```

Очікуваний результат — 76 і 77. Запустивши таку програму, ми переконаємось, що все спрацювало, як і очікувалось.

Але давайте спробуємо запустити такий код:

Pascal integernumbers.pas

```
program integernumbers;

var n: LongInt;

begin
  n := 2147483647;
  WriteLn(n);

  n := n + 1;
  WriteLn(n);
end.
```

Спочатку ми бачимо 2147483647 — це найбільше значення, яке ще поміщається в `LongInt`. Після `n := n + 1` інтуїтивно хотілося б отримати «наступне» число 2147483648, але в типі `LongInt` для нього просто немає місця: результат «згортається», і на екрані з'являється −2147483648. Це явище називають **переповненням** цілого типу (англ. *Integer overflow*). Логіка цього «згортання» схожа на те, як працює лічильник води чи газу: після відмітки 999999 лічильник просто скидається в 000000.

У кожного типу даних у пам'яті є фіксований розмір, тому й існує обмежена область значень. Тип `LongInt` зазвичай займає 32 біти, отже допустимі цілі від −2147483648 до 2147483647 включно — саме це пояснює

переповнення в прикладі вище.

Але в мові Pascal є дуже багато цілочисельних типів даних. Давайте їх розглянемо:

| **Тип** | **Розмір** | **min** | **max** |
|---|---|---|---|
| `ShortInt` | 8 біт | $-128$ | 127 |
| `SmallInt` | 16 біт | $-32768$ | 32767 |
| `LongInt` | 32 біти | $-2147483648$ | 2147483647 |
| `Int64` | 64 біти | $-2^{63}$ | $2^{63}-1$ |
| `NativeInt` | $S_{\text{Pointer}}$ біт** | $-2^{S_{\text{Pointer}}-1}$ | $2^{S_{\text{Pointer}}-1}-1$ |
| `Byte` | 8 біт | 0 | 255 |
| `Word` | 16 біт | 0 | 65535 |
| `LongWord` | 32 біти | 0 | 4294967295 |
| `UInt64` | 64 біти | 0 | $2^{64}-1$ |
| `NativeUInt` | $S_{\text{Pointer}}$ біт** | 0 | $2^{S_{\text{Pointer}}}-1$ |
| `Integer` | $S_{\text{Integer}}$ біт* | $-2^{S_{\text{Integer}}-1}$ | $2^{S_{\text{Integer}}-1}-1$ |

* $S_{\text{Integer}}$ — розмір типу `Integer` в поточному режимі компілятора (на сучасних 32/64-бітних платформах у Free Pascal найчастіше 32 біти; на деяких 16-бітних цілях може бути 16).

** $S_{\text{Pointer}}$ — розмір машинного слова (розрядність цільової архітектури): зазвичай 32 біти на x86 і 64 біти на x86_64.

У таблиці наведено типові розміри та діапазони для **знакових** (`ShortInt`, `SmallInt`, ...) і **беззнакових** (`Byte`, `Word`, ...) цілих типів. Беззнакові типи зберігають лише невід'ємні значення, тому їхній мінімум завжди 0.

Але є в цих типах одна проблема: вони жахливо називаються. Зв'язати назву типу, його розмір і наявність від'ємних значень в області значень просто нереально! Ще більш заплутаною ситуація буде, якщо ви програмуєте на інших мовах програмування. Наприклад, в мові C# тип `long` — 64-бітний, а `short` — 16-бітний.

Зате подивіться, як називаються типи даних у таких мовах програмування, як **Zig** чи **Rust**. Типи даних із знаком: `i8`, `i16`, `i32`, `i64`, `isize` (буква «i» — скорочення від слова «integer», тобто «ціле число»). Типи даних без знаку: `u8`, `u16`, `u32`, `u64`, `usize` (буква «u» — скорочення від слова «unsigned integer», тобто «беззнакове ціле число»).

Із допомогою псевдонімів типів у Pascal можна зробити так само! Давайте створимо модуль `types`, у якому зберігатимемо зручні і читабельні псевдоніми для типів даних:

Pascal types.pas

```pascal
unit types;

interface

type
  u8 = Byte;
  i8 = ShortInt;
  u16 = Word;
  i16 = SmallInt;
  u32 = LongWord;
  i32 = LongInt;
  u64 = UInt64;
  i64 = Int64;
  f32 = Single;
  f64 = Double;
  usize = NativeUInt;
  isize = NativeInt;
  c_int = i32;
  c_uint = u32;
  {$IF DEFINED(WINDOWS) OR NOT DEFINED(CPU64)}
  c_long  = i32;
  c_ulong = u32;
  {$ELSE}
```

```
  c_long = i64;
  c_ulong = u64;
  {$ENDIF}

implementation

end.
```

Цей модуль, як і `vectors`, **ми перенесемо у код нашої гри**. Із цього моменту, ми його підключатимемо у всі наші програми і використовуватимемо саме ці типи даних, оскільки із цими назвами нам легше буде контролювати розміри даних і сумісність із кодом, написаним мовою **C**.

До речі, щодо мови C. Зверніть увагу на дивні типи, які були додані у `types`, такі як `c_int`, `c_uint`, `c_long` і `c_ulong`.

**Примітка**

Не лякайтесь конструкції `{$IF DEFINED...}` у визначенні `c_long` і `c_ulong` — це так звана **умовна компіляція** (англ. *Conditional compilation*): компілятор сам вибере потрібний розмір типу залежно від операційної системи й апаратної платформи. Вам не потрібно розуміти цей механізм зараз.

Справа у тому, що нам доведеться взаємодіяти із кодом на C. І передбачається, що ці типи матимуть такий же розмір, як і типи даних `int`, `unsigned int`, `long int` і `unsigned long int` у мові програмування C.

**Задача 16**

У модулі `vectors`, який ви розробляли в задачі Задачі 15, замініть тип `single` на `f32`. Переконайтесь, що тести, які ви написали для операцій із векторами, все ще виконуються.

**Задача 17**

В задачі Задачі 15 ви писали тести із використанням функції `Assert`. Напишіть ще одну програму-тест, у якій переконайтесь, що типи даних, які ми оголосили в модулі `types`, мають правильні розміри. В Pascal для визначення розміру варто користуватись функцією `SizeOf` — вона повертає розмір типу даних у байтах.

## 3.18 Зберігання двовимірних даних

У нашому рушії, як ми уже не раз згадували, карта нагадує зошит в клітинку: кожна клітинка позначає або стіну, або зону, по якій можна переміщуватись. Це означає, що ми повинні зберігати інформацію про те, яка із них — стіна, а яка — ні. Для цього нам потрібен масив, але такий, що зберігає двовимірну структуру.

Хоча в Pascal і можна працювати із «штатними» двовимірними масивами, ми скористаємось іншим способом: ми використаємо звичайний одновимірний масив для зберігання карти. Давайте подумаємо, як це працює.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

Рис. 3.4: Нумерація клітинок карти.

На Рис. 3.4 показано, як упакувати двовимірну сітку $8 \times 8$ у одновимірний масив: верхній рядок містить індекси $0, \ldots, 7$, наступний — $8, \ldots, 15$, і так далі до 63. Якщо позначити номер стовпчика через $x$ і номер рядка через $y$, то індекс елемента масиву дорівнює $8y + x$.

**Pascal** map1d.pas

```pascal
program map1d;

uses types;

const
  mapWidth = 8; {ширина карти (розмір вздовж координати x)}
  mapHeight = 8;  {висота карти (розмір вздовж координати y)}

var
  {0 - порожньо, 1 - стіна}
  map: array[0..mapWidth * mapHeight - 1] of u8 = (
    1,1,1,1,1,1,1,1,
    1,0,0,0,0,0,0,1,
    1,0,0,0,0,1,0,1,
    1,0,0,0,0,1,0,1,
    1,0,0,1,1,0,0,1,
    1,0,0,0,0,0,0,1,
    1,0,0,0,0,0,0,1,
    1,1,1,1,1,1,1,1
  );

function CellIndex(const x, y: isize): isize;
begin
  CellIndex := y * mapWidth + x;
end;

begin
```

```
  WriteLn(
    'map[0] = ', map[0],
    ', map[54] = ', map[54],
    ', map[63] = ', map[63]
  );
end.
```

**Запитання 8**

Спробуйте з'ясувати, у чому переваги і недоліки використання одновимірного масиву для зберігання двовимірних даних, у порівнянні із двовимірним масивом.

**Задача 18**

Ми щойно змогли представити карту із допомогою масиву. Кожен елемент масиву описує клітинку. Уявіть, що довжина сторони стіни — 50. Напишіть функцію, яка перевірить, чи є точка з вказаними координатами точкою стіни. Не забувайте користуватись для цього модулем `vectors`, який ви розробили в Задачі 15.

Підказка: пригадайте, як ми шукали першу точку-кандидат $\overrightarrow{c_{v,0}}$ або $\overrightarrow{c_{h,0}}$ із допомогою одної незвичайної функції.

## 3.19 Вказівники

Чи малювали ви коли-небудь родинне дерево? Уявіть, що ви хочете зберегти його в програмі: у кожної людини є мама, тато, можливо — чоловік або дружина, і список дітей. Звучить просто, поки не виникає запитання, як це правильно зробити.

Ми уже знаємо, що для того, щоб описати якусь сутність, використовуються **записи**.

Створимо запис людини:

Pascal familytree.pas

```pascal
program familytree;

uses types;

type
  TPerson = record
    { ім'я }
    firstName: ShortString;

    { прізвище }
    lastName: ShortString;

    mother: TPerson;
    father: TPerson;
  end;
begin

end.
```

Виглядає логічно: усі ж люди мають батьків, і, маючи цю інформацію про всіх людей, можна визначати всі інші родинні зв'язки.

Але ця програма навіть не скомпілюється. Давайте подумаємо, чому.

Візьмемо спочатку запис `TVector` із модуля `vectors`. Який його розмір? Скільки місця він займає у пам'яті? Оскільки вектор — це два поля типу `f32`, розмір вектора у бітах:

$$S_{\text{TVector}} = S_{\text{f32}} + S_{\text{f32}} = 32 + 32 = 64$$

Знаючи, що тип даних `ShortString` займає 256 байт, спробуємо порахувати розмір типу `TPerson`:

$$\begin{aligned} S_{\text{TPerson}} &= S_{\text{ShortString}} + S_{\text{ShortString}} + S_{\text{TPerson}} + S_{\text{TPerson}} \\ &= 512 \cdot 8 + 2S_{\text{TPerson}} \\ &= 4096 + 2S_{\text{TPerson}} \end{aligned}$$

У нас утворилось рівняння, корінь якого — $S_{\text{TPerson}} = -4096$, тобто від'ємне число. **Це абсурд**, адже розмір типу в пам'яті не може бути від'ємним. Причина в тому, що в нашому визначенні `TPerson` запис містить два поля того самого типу `TPerson` за значенням (`mother` та `father`). Щоб зберегти одну людину, треба зберегти двох її батьків; щоб зберегти кожного з батьків, треба зберегти ще їхніх батьків, і так далі без кінця. Тобто такий тип вимагає насправді не від'ємного, а **нескінченного** обсягу пам'яті, тому компілятор не може обчислити його розмір і відмовляється компілювати програму.

Як же вирішити таку проблему? Замість того, щоб тримати повну інформацію про батьків у типі `TPerson`, нам достатньо всього лише зберегти посилання на них. Для цього використовуються типи-вказівники.

Давайте зробимо невеличку зміну в програму, щоб вона скомпілювалась:

Pascal familytree.pas

```pascal
program familytree;

uses types;

type
  TPerson = record
    firstName: ShortString;
    lastName: ShortString;

    mother: ^TPerson;
    father: ^TPerson;
  end;
```

```
begin

end.
```

Символ `^` тут знаходиться не просто так. Він означає «вказівник на значення, тип якого — `TPerson`». Усі вказівники зберігають не значення, а лише адресу у пам'яті, де саме знаходиться те значення. Це легше продемонструвати на прикладі:

Pascal familytree.pas

```
program familytree;

uses types;

type
  TPerson = record
    firstName: ShortString;
    lastName: ShortString;

    mother: ^TPerson;
    father: ^TPerson;
  end;

var
  oleksii: TPerson;
  galyna: TPerson;
  davyd: TPerson;
  anna: TPerson;
  anatolii: TPerson;

begin
  oleksii.firstName := 'Oleksii';
  oleksii.lastName := 'Romanchenko';
```

```
  oleksii.mother := nil;
  oleksii.father := nil;

  galyna.firstName := 'Galyna';
  galyna.lastName := 'Romanenko';
  galyna.mother := nil;
  galyna.father := nil;

  davyd.firstName := 'Davyd';
  davyd.lastName := 'Romanchenko';
  davyd.mother := @galyna;
  davyd.father := @oleksii;

  anna.firstName := 'Anna';
  anna.lastName := 'Romanchenko';
  anna.mother := @galyna;
  anna.father := @oleksii;

  anatolii.firstName := 'Anatolii';
  anatolii.lastName := 'Melnyk';
  anatolii.mother := nil;
  anatolii.father := nil;
end.
```

У цьому прикладі тип `TPerson` описує людину з ім'ям, прізвищем і двома посиланнями на батьків. Ключова відмінність від попередньої невдалої версії: поля `mother` та `father` мають тип `^TPerson`, тобто це *вказівники*, а не вкладені записи `TPerson`. Тому розмір запису фіксований: два рядки і два вказівники. Оскільки розмір кожного вказівника дорівнює $S_{\text{Pointer}}$ розмір запису у бітах дорівнює:

$$\begin{aligned} S_{\text{TPerson}} &= S_{\text{ShortString}} + S_{\text{ShortString}} + S_{\text{ptrTPerson}} + S_{\text{ptrTPerson}} \\ &= 512 \cdot 8 + 2S_{\text{ptrTPerson}} \\ &= 4096 + 2S_{\text{Pointer}} \end{aligned}$$

Далі у секції `var` створюються конкретні люди: `oleksii`, `galyna`, `davyd`, `anna`, `anatolii`. Для кожної людини задаються `firstName` і `lastName`.

Значення `nil` означає, що посилання на одного з батьків відсутнє (інформація невідома або не задана). Наприклад, у `oleksii`, `galyna` та `anatolii` обидва посилання — `nil`.

Оператор `@` бере адресу змінної. Тому рядки `davyd.mother := @galyna` і `davyd.father := @oleksii` означають, що матір'ю Давида є Галина, а батьком — Олексій. Аналогічно встановлюються зв'язки для `anna`.

Отже, програма будує маленьке родинне дерево через посилання між записами: у дітей поля `mother` / `father` вказують на вже створені об'єкти батьків.

Щоб продемонструвати силу такого інструменту, як вказівники, розгляньмо невеликий приклад. Уявімо, що під час заповнення дерева в Галини помилково вказали прізвище `Romanenko` замість `Romanchenko`. Тепер виправимо це значення і подивімося, як зміняться `galyna.lastName`, `anna.mother^.lastName`, `davyd.mother^.lastName`. Тобто після опису сім'ї допишемо такий код:

**Pascal** familytree.pas

```pascal
WriteLn('Before: ');
WriteLn('  ', galyna.lastName); {Romanenko}
WriteLn('  ', anna.mother^.lastName); {Romanenko}
WriteLn('  ', davyd.mother^.lastName); {Romanenko}

galyna.lastName := 'Romanchenko';

WriteLn('After: ');
```

```
WriteLn('  ', galyna.lastName); {Romanchenko}
WriteLn('  ', anna.mother^.lastName); {Romanchenko}
WriteLn('  ', davyd.mother^.lastName); {Romanchenko}
```

Ефект тут дуже важливий: ми змінили прізвище лише в одному місці — у записі `galyna`. Але `anna.mother` і `davyd.mother` зберігають адресу цього самого запису, тому під час звернення через `^` вони читають уже оновлене значення. Тобто вказівники не копіюють об'єкт, а лише посилаються на нього, і всі такі посилання бачать одні й ті самі актуальні дані.

**Визначення**

Операція `a^`, тобто доступ до значення за вказівником `a`, називається **розіменуванням вказівника** (англ. *pointer dereferencing*).

**Примітка**

Взагалі, основним символом вказівника в Pascal вважається не символ каретки `^`, а символ стрілки `↑`. Тобто записи `↑TPerson` і `^TPerson` означають одне й те саме. Основна причина, чому був вибраний символ `↑` у якості вказівника — читабельність коду. Символ стрілки — чудова візуальна метафора про те, що тип на щось **вказує**. Було зручно роздрукувати код на папері та бачити зрозумілий символ вказівника у вигляді стрілки вгору. Аналогічно, символ `@` — таке ж саме наглядне позначення **адреси**.

Символ `^` — лише альтернатива стрілці вгору. Додана вона була тому, що існували комп'ютери, у наборі символів яких просто не було стрілки вгору. Символ каретки просто візуально схожий на стрілку (теж наче вказує вгору), тому і був вибраний як додатковий символ вказівника.

Оскільки сучасні клавіатури мають саме символ `^`, а не символ `↑`, програмісти стали частіше набирати саме `^`, а про стрілку вгору вже майже всі і забули.

### 3.19.1 Розіменування нульового вказівника

Що буде, якщо спробувати отримати прізвище батька людини, батько якої не вказаний (тобто посилання на батька дорівнює `nil`)?

Pascal familytree.pas

```
WriteLn(oleksii.father^.lastName);
```

Програма успішно скомпілюється, але зламається під час виконання. Справа в тому, що вказівники зберігають адресу. Адреса — це звичайне число, яке позначає комірку в пам'яті, де знаходиться значення за вказівником.

Але існує спеціальна адреса — адреса 0. Її ми й позначаємо ключовим словом `nil`. В інших мовах програмування частіше зустрічається слово `null`. Передбачається, що адреса 0 позначає, що вказівник не вказує на жодні дані. І при розіменуванні такого вказівника програма завершується **з помилкою виконання** (англ. *Runtime error*), тобто, як кажуть в повсякденному житті, «вильотом».

**Примітка**

Багато мов програмування мають схожу ідею роботи з `null` / `nil`, але механізм захисту може відрізнятися. Наприклад, у C розіменування `NULL` має невизначену поведінку: програма може аварійно завершитися, а може поводитися непередбачувано. Тому в C особливо важливо вручну перевіряти вказівники перед розіменуванням.

Ми не хочемо, щоб наша програма вилітала. Тому ми повинні додати перевірку на значення `nil`.

Pascal familytree.pas

```
if oleksii.father <> nil then
  WriteLn(oleksii.father^.lastName)
else
```

```
  {обробка випадку нульового вказівника}
  WriteLn('No information about father.');
```

**Задача 19**

Напишіть функцію, яка перевіряє, чи є дві людини братом і сестрою (або братами/сестрами).

### 3.19.2 Дикі вказівники

Додамо у нашу програму функцію, яка повертає людину:

Pascal familytree.pas

```
function CreateImportedPersonFromArchive(
  const recordId: i32): TPerson;
var
  mother: TPerson;
  father: TPerson;
begin
  mother.firstName := 'Olha';
  mother.lastName := 'Koval';
  mother.mother := nil;
  mother.father := nil;

  father.firstName := 'Oleksandr';
  father.lastName := 'Koval';
  father.mother := nil;
  father.father := nil;

  CreateImportedPersonFromArchive.firstName := 'Iryna';
  CreateImportedPersonFromArchive.lastName := 'Koval';
  CreateImportedPersonFromArchive.mother := @mother;
  CreateImportedPersonFromArchive.father := @father;
```

```
end;
```

На перший погляд, все прекрасно. Ми повертаємо людину із іменем Ірина, її батьків звуть Олександр і Ольга.

Давайте спробуємо вивести ім'я всіх цих трьох людей:

Pascal familytree.pas

```
iryna := CreateImportedPersonFromArchive(1);
WriteLn(iryna.firstName, ' ', iryna.lastName);
WriteLn(iryna.mother^.firstName, ' ', iryna.mother^.lastName);
WriteLn(iryna.father^.firstName, ' ', iryna.father^.lastName);
```

Усе логічно. Дані виводяться, як очікувалось...

Але давайте створимо нову процедуру:

Pascal familytree.pas

```
procedure ClobberStack;
var
  i: isize;
  noise: array[0..2000] of u8;
begin
  for i := Low(noise) to High(noise) do
    noise[i] := i mod 256;
end;
```

Назва `ClobberStack` буквально означає щось на кшталт «перезатерти стек». Нам неважливо, що саме ця процедура робить. Важливо, що вона буде викликатись між створенням людини і виведенням інформації про неї і її батьків.

Pascal familytree.pas

```
iryna := CreateImportedPersonFromArchive(1);
ClobberStack;
WriteLn(iryna.firstName, ' ', iryna.lastName);
WriteLn(iryna.mother^.firstName, ' ', iryna.mother^.lastName);
```

```
WriteLn(iryna.father^.firstName, ' ', iryna.father^.lastName);
```

Якщо ви спробуєте запустити таку програму, то спочатку в консолі ви побачите повне ім'я Ірини. А потім станеться щось **СТРАШНЕ**. Ви побачите безліч хаотичних і незрозумілих символів, виглядатимуть вони, наче це «глітчі» із якихось науково-фантастичних фільмів про хакерів.

Ви побачите все, крім потрібних даних.

Справа у тому, що вказівники `iryna.mother` і `iryna.father` стали **дикими** (англ. *Wild pointers*, але в англомовній літературі частіше вживається термін *Dangling pointers*, тобто «вказівники, що звисають»).

Уявіть, що ви записали олівцем відповідь на аркуші, а потім хтось стер ваш запис і написав щось своє на тому ж місці. Ви все ще пам'ятаєте, **де** був запис, але того, що ви написали, там уже немає. Саме це і сталось із нашими вказівниками.

Подумаємо, як так сталося. Ми знаємо, що тип `TPerson` має фіксований розмір $S_{TPerson}$. Тобто, передбачається, що функція `CreateImportedPersonFromArchive` має повернути одну людину. Проте, фактично, разом із нею ви отримали вказівники на ще двох людей — її батьків! Проте на яку область пам'яті посилаються ці вказівники? На змінні `father` і `mother` функції `CreateImportedPersonFromArchive`, виконання якої завершилось! Змінні функції — це локальні дані, які живуть лише впродовж виконання функції. Після завершення функції ця пам'ять уже не належить її локальним змінним. Будь-яка інша функція може зайняти цю область і вільно перезатерти її, що і зробила функція `ClobberStack`.

**Увага**

**Ніколи не повертайте вказівники на локальні змінні!!!** Це погано завершиться! Проблеми від цієї дії, які ви собі створите, значно більші, ніж від розіменування вказівника `nil`. Відслідковувати джерело проблем, спричинених дикими вказівниками, — справжня мука.

## 3.20 Динамічна пам'ять

Ми вже бачили, як працювати із локальними змінними: вони з'являються на вході у функцію і зникають після її завершення. Це швидко й передбачувано, але накладає обмеження: ми не можемо «взяти з собою» об'єкт довше, ніж триває виклик функції (згадайте проблемну функцію `CreateImportedPersonFromArchive`), і не можемо легко змінювати розмір даних «на льоту» — наприклад, коли кількість елементів залежить від вводу користувача чи вмісту файлу.

Отож, до цього моменту ми користувались переважно тими даними, розмір яких **відомий під час компіляції**: фіксовані масиви, записи фіксованого розміру, числа тощо. Проте реальні програми рідко обмежуються лише такими випадками.

Наступний крок — навчитися виділяти пам'ять **динамічно**: тоді розмір і тривалість життя даних визначаються вже під час виконання програми, а не лише на етапі компіляції.

Із допомогою динамічного виділення пам'яті ми зможемо досить легко виправити функцію `CreateImportedPersonFromArchive`:

Pascal familytree.pas

```
function CreateImportedPersonFromArchive(
  const recordId: i32): TPerson;
var
  motherPtr: ^TPerson;
  fatherPtr: ^TPerson;
begin
  New(motherPtr);
  New(fatherPtr);

  motherPtr^.firstName := 'Olha';
  motherPtr^.lastName := 'Koval';
  motherPtr^.mother := nil;
```

```
  motherPtr^.father := nil;

  fatherPtr^.firstName := 'Oleksandr';
  fatherPtr^.lastName := 'Koval';
  fatherPtr^.mother := nil;
  fatherPtr^.father := nil;

  CreateImportedPersonFromArchive.firstName := 'Iryna';
  CreateImportedPersonFromArchive.lastName := 'Koval';
  CreateImportedPersonFromArchive.mother := motherPtr;
  CreateImportedPersonFromArchive.father := fatherPtr;
end;
```

Ключова різниця з попередньою версією — у тому, *де* живуть батьки. Раніше змінні `mother` і `father` були **локальними**: їхня пам'ять належала функції й переставала бути «вашою» одразу після її завершення. Тепер виклики `New(motherPtr)` і `New(fatherPtr)` просять у програми окремі шматки пам'яті в **купі** (англ. *heap*) — області, де дані не прив'язані до життєвого циклу конкретної функції. Після цього `motherPtr` і `fatherPtr` вказують уже не на тимчасові локальні змінні, а на два об'єкти `TPerson`, виділені динамічно. Тому поля `mother` і `father` у поверненій Ірині зберігають **коректні** адреси: навіть якщо пізніше викликати `ClobberStack` (вона не перезапише пам'ять в купі), тому уже все працюватиме коректно:

Pascal familytree.pas

```
iryna := CreateImportedPersonFromArchive(1);
ClobberStack;
WriteLn(iryna.firstName, ' ', iryna.lastName);
WriteLn(iryna.mother^.firstName, ' ', iryna.mother^.lastName);
WriteLn(iryna.father^.firstName, ' ', iryna.father^.lastName);

Dispose(iryna.mother);
Dispose(iryna.father);
```

**Увага**

Варто пам'ятати про зворотний бік: те, що виділили через `New`, треба колись звільнити (у Pascal для цього є `Dispose`). Інакше програма буде накопичувати «зайву» пам'ять, і її споживання пам'яті, яке буде показуватись у диспетчері завдань, буде постійно зростати!

Поки що головне: динамічне виділення дозволяє тримати об'єкти довше, ніж триває один виклик функції, і саме тому цей прийом прибирає проблему диких вказівників у нашому прикладі.

## 3.21 Динамічні масиви

Вказівники — це не єдиний метод роботи із динамічною пам'яттю. Ще одним, не менш важливим методом для нас, є **динамічні масиви** (англ. *Dynamic arrays*), тобто масиви, розмір яких відомий не під час компіляції, а саме під час виконання програми.

Давайте спробуємо створити функцію, яка згенерує нам карту для нашого рейкаст-рушія, але із розмірами, які будуть передані в тіло функції. Для простоти ми згенеруємо пустий простір, обмежений стінами із чотирьох боків.

Pascal dynamicmap.pas

```
program dynamicmap;

uses types;

type
  {
  Компілятор FPC вимагає псевдонім типу динамічного
  масиву, який буде повертати функція.
  }
  TMapFlat = array of u8;
```

```
function GenerateMap(width: isize; height: isize): TMapFlat;
var
  i: isize;
  r: TMapFlat;
begin
  SetLength(r, Integer(width * height));

  for i := 0 to width - 1 do
  begin
    r[0 * width + i] := 1;
    r[(height - 1) * width + i] := 1;
  end;

  for i := 0 to height - 1 do
  begin
    r[i * width + 0] := 1;
    r[i * width + (width - 1)] := 1;
  end;

  GenerateMap := r;
end;

const
  mapWidth = 20;
  mapHeight = 10;

var
  map: TMapFlat;
  i: isize;
  j: isize;

begin
```

```
  map := GenerateMap(mapWidth, mapHeight);

  for i := 0 to mapHeight - 1 do
  begin
    for j := 0 to mapWidth - 1 do
      Write(map[i * mapWidth + j]);

    WriteLn();
  end;
end.
```

Розглянемо функцію `GenerateMap`. Вона повертає `array of u8`, але компілятор вимагає створити для цього псевдонім для типу масивів, який повертається функцією.

А далі відбувається **магія**: у функції GenerateMap, ми встановлюємо розмір цього масиву функцією `SetLength`. Зверніть увагу, що і параметр `width`, і параметр `height` можуть мати довільні значення, тобто компілятор не може знати наперед їхніх значень. Тобто функція `SetLength` встановлює розмір масиву прямо в процесі виконання програми! Далі ви можете працювати із масивом так само, як і раніше, усе настільки легко! Дані динамічних масивів, так само, як і дані, виділені функцією `New`, зберігаються у купі.

Динамічні масиви точно стануть вам в нагоді якщо, наприклад, ви захочете читати карту із файла, і заздалегідь не знаєте її розмірів.

**Запитання 9**

У динамічних масивів може бути будь-який розмір, а розмір статичних — обмежений тим, що ми сказали компілятору. Дослідіть, навіщо нам тоді потрібні статичні масиви, якщо існують динамічні?

**Примітка**

У вас може виникнути таке питання: чому нам потрібно вручну звільняти пам'ять, виділену функцією `New` (для цього ми використовували функцію `Dispose`), а динамічні масиви — ні?

Справа у тому, що вказівник — це всього лише одне число, що позначає деяку адресу в пам'яті. Такі вказівники називають **сирими вказівниками** (англ. *Raw pointers*), і вони вимагають ручного керування: ви вручну перевіряєте, щоб вони не стали дикими, ви їх вручну створюєте і вручну звільняєте, коли вони більше не потрібні. Загалом, ви повністю відповідальні за їхній цикл життя. Пам'ять, яка управляється повністю програмістом, — це **некерована пам'ять** (англ. *unmanaged memory*).

Динамічний масив у Free Pascal — це теж свого роду вказівник, але із додатковим механізмом: компілятор і рантайм самі стежать за тим, щоб звільнити пам'ять під масив, коли він більше не потрібен (зазвичай через **лічильник посилань**, англ. *reference counting*, на тіло масиву). Тому для динамічного масиву зазвичай не викликають `New` і `Dispose` вручну — на відміну від «сирого» вказівника. Динамічна пам'ять, що позбавляє програміста необхідності думати про управління її життєвим циклом, — це **керована пам'ять** (англ. *managed memory*).

## 3.22 Передача параметрів по посиланню

Давайте спробуємо написати процедуру, яка поміняє місцями значення двох аргументів. Тобто після виклику змінна `y` матиме те, що було в `x` *до* виклику, а `x` — те, що було в `y` до виклику.

За замовчуванням програма приймає параметри за значенням, всередині неї існують лише копії чисел: навіть після обміну зовнішні `x` і `y` залишаться незмінними:

Pascal numswap.pas

```pascal
program numswap;

uses types;

procedure Swap(a: i32; b: i32);
var temp: i32;
begin
  temp := a;
  a := b;
  b := temp;
  end;

var x, y: i32;
begin
  x := 4;
  y := 8;
  Swap(x, y);

  WriteLn(x, ' ', y); {4 8}
end.
```

Потрібно, щоб процедура читала й записувала саме ті комірки пам'яті, які належать виклику — тобто передавати їх *по посиланню*.

Маючи на озброєнні такий інструмент, як вказівники, ми можемо легко написати таку процедуру.

Pascal numswap.pas

```pascal
program numswap;

uses types;

{
```

```
Pascal вимагає створити псевдонім для типу вказівника,
якщо він використовуватиметься як параметр чи вихідне значення
підпрограми.
}
type
  Pi32 = ^i32;

procedure Swap(const a: Pi32; const b: Pi32);
var temp: i32;
begin
  temp := a^;
  a^ := b^;
  b^ := temp;
end;

var x, y: i32;
begin
  x := 4;
  y := 8;
  Swap(@x, @y);

  WriteLn(x, ' ', y); {8 4}
end.
```

Тут у виклик передаються адреси (`@x`, `@y`), а в тілі процедури через `a^` і `b^` ми звертаємось до вмісту тих самих змінних у коду, що викликав підпрограму. Модифікатор `const` при цьому стосується самих параметрів-*вказівників*: усередині `Swap` не можна перепризначити `a` чи `b` на іншу адресу, але змінювати `a^` і `b^` дозволено — ми змінюємо числа за вже переданими адресами.

Проте є значно легший спосіб — через `var`-параметри. Давайте розглянемо, як це виглядає:

Pascal numswap.pas

```pascal
program numswap;

uses types;

procedure Swap(var a: i32; var b: i32);
var temp: i32;
begin
  temp := a;
  a := b;
  b := temp;
end;

var x, y: i32;
begin
  x := 4;
  y := 8;
  Swap(x, y);

  WriteLn(x, ' ', y); {8 4}
end.
```

Для параметрів, оголошені як `var`, компілятор сам забезпечить передачу за посиланням: не потрібно писати ні `@`, ні `^`. У порівнянні із вказівниками такий підхід має ще одну перевагу — **він безпечніший**, адже в такому випадку компілятор не дозволить передати `nil`, тобто перевірок на `nil` не потрібно!

Таким чином, крім звичайних параметрів без модифікатора, ми можемо явно задати `const` або `var`.

`const`-параметри можуть передаватися як за значенням, так і за посиланням (компілятор сам вирішить, який спосіб кращий). Проте компілятор гарантує, що відповідний аргумент не зміниться протягом виконання функції.

`var`-параметри можуть передаватися лише за посиланням. Але компілятор дозволяє змінювати такі параметри всередині підпрограми.

Ми часто будемо використовувати `var`-параметри, оскільки вони більш зручні і зрозумілі, ніж вказівники. Наприклад, нам якщо потрібно перемістити персонажа, ми можемо написати процедуру, яка приймає параметр `var player: TPlayer` і змінює його положення.

## 3.23 Спільні бібліотеки

Остання «фіча», про яку нам варто дізнатись — це **спільні бібліотеки**.

Для того, щоб зрозуміти, що це таке, давайте згадаємо, що ми взагалі розробляємо. Ми пишемо свою гру!

А для гри потрібне вікно, у якому ми зможемо «малювати картинку». Для цього нам потрібно, щоб операційна система виділила нам ресурс пам'яті для малювання, а також бажано нам мати і набір функцій, які полегшать нам це саме малювання. Тобто нам потрібен якийсь модуль на кшталт `math`, але для графіки.

Хотілось би скористатись якимось модулем із стандартної бібліотеки Pascal (як і випадку із тим же `math`), але такого не існує. Теоретично, ми могли б розробити такий модуль самі, але це **дуже складна робота**, яка виходить далеко за рамки просто створення ігор (це уже область операційних систем).

Залишається лише один вихід: скористатись кодом, який уже написали інші програмісти. Такий код, написаний іншим розробником для того, щоб його могли використовувати інші, називається **бібліотекою** (англ. *Library*).

Одною із найпопулярніших бібліотек, які надають таке «полотно для малювання», є Simple DirectMedia Layer (або просто **SDL**).

Давайте спробуємо її підключити у наш проект.

Спочатку бібліотеку потрібно завантажити. Знайти бібліотеку можна по посиланню:

https://github.com/libsdl-org/SDL/releases

Просто вибираєте свою операційну систему і апаратну платформу і завантажуєте архів. Після розпакування архіва ви маєте побачити файл нашої бібліотеки. Якщо ваша ОС — Windows, це буде файл `SDL3.dll`, якщо операційна система на базі ядра Linux, то файл `libSDL3.so`, а для користувачів macOS — `libSDL3.dylib`.

Ці файли — уже скомпільований код, але не у вигляді виконуваного файлу (тобто кінцевої програми), а у вигляді **спільної бібліотеки** (англ. *Shared Library*). Це чимось схоже на модулі, але підключаються до програми вони **динамічно**. Один із найпростіших і найнадійніших способів підключення — покласти файл бібліотеки поруч із виконуваним файлом.

Але як нам викликати підпрограми, що знаходяться всередині цих бібліотек? Для цього нам потрібно створити інтерфейс **зв'язки** (англ. *Binding*). Створіть для цього файл `sdl3.pas`, і перепишіть туди код, що знаходиться у Додатку 1.

Тепер — створимо нашу основну програму:

Pascal — game.pas

```
program game;

uses
  sdl3, types;

const
  {
  Назва вашої гри.
  Можете пофантазувати і назвати свою гру як завгодно!
  }
  gameName = 'Dark corridor';

var
  window: PSDL_Window;
```

```
  renderer: PSDL_Renderer;
  event: SDL_Event;
  running: Boolean;
  rect: SDL_FRect;

begin
  if SDL_Init(SDL_INIT_VIDEO or SDL_INIT_EVENTS) <> SDL_TRUE then
  begin
    WriteLn('SDL_Init failed: ', SDL_GetError);
    Halt(1);
  end;

  window := SDL_CreateWindow(
    gameName,
    800, 600,
    SDL_WINDOW_RESIZABLE);

  if window = nil then
  begin
    WriteLn('SDL_CreateWindow failed: ', SDL_GetError);
    SDL_Quit;
    Halt(1);
  end;

  renderer := SDL_CreateRenderer(window, nil);
  if renderer = nil then
  begin
    WriteLn('SDL_CreateRenderer failed: ', SDL_GetError);
    SDL_DestroyWindow(window);
    SDL_Quit;
    Halt(1);
  end;
```

```
  rect.x := 200;
  rect.y := 150;
  rect.w := 400;
  rect.h := 300;

  running := True;
  while running do
  begin
    while SDL_PollEvent(@event) = SDL_TRUE do
    begin
    if event.event_type = SDL_EVENT_QUIT then
      running := False;
    end;

    SDL_SetRenderDrawColor(renderer, 25, 30, 40, 255);
    SDL_RenderClear(renderer);

    SDL_SetRenderDrawColor(renderer, 90, 190, 255, 255);
    SDL_RenderFillRect(renderer, @rect);

    SDL_RenderPresent(renderer);
    SDL_Delay(16);
  end;

  SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);
  SDL_Quit;
end.
```

Зверніть увагу, що у цій програмі ми підключаємо модуль `sdl3` і користуємось його функціями. Детальніше про те, що, власне, роблять ці функції, ми дізнаємось далі.

Спробуємо зараз запустити цю програму:

Shell

```
fpc -Px86_64 game.pas
```

**Увага**

Зверніть увагу на опцію компілятора `-Px86_64`. Текст після `-P` — це ваша апаратна платформа, і ми повинні точно вказати ту ж саму платформу, для якої створена бібліотека.

Наприклад, на комп'ютері із архітектурою x86_64 без цієї опції компілятор може створити виконуваний файл для архітектури i386: все одно x86_64 здатна його відкрити.

Але при наявності спільних бібліотек це вже може перестати працювати, оскільки спільні бібліотеки одразу скомпільовані під конкретну платформу. Тому код і бібліотека, зібрані під різні платформи, просто не зможуть між собою спілкуватись.

Тому правильно вкажіть вашу апаратну платформу! Подивитись список підтримуваних платформ можна командою `fpc -help` (шукайте текст `-P<x> Target CPU / compiler related options`).

Запустіть скомпільовану програму. Якщо ви побачили зображення, як на Рис. 3.5, то ви зробили все вірно. **Цю програму ми використаємо як скелет для нашої гри**!

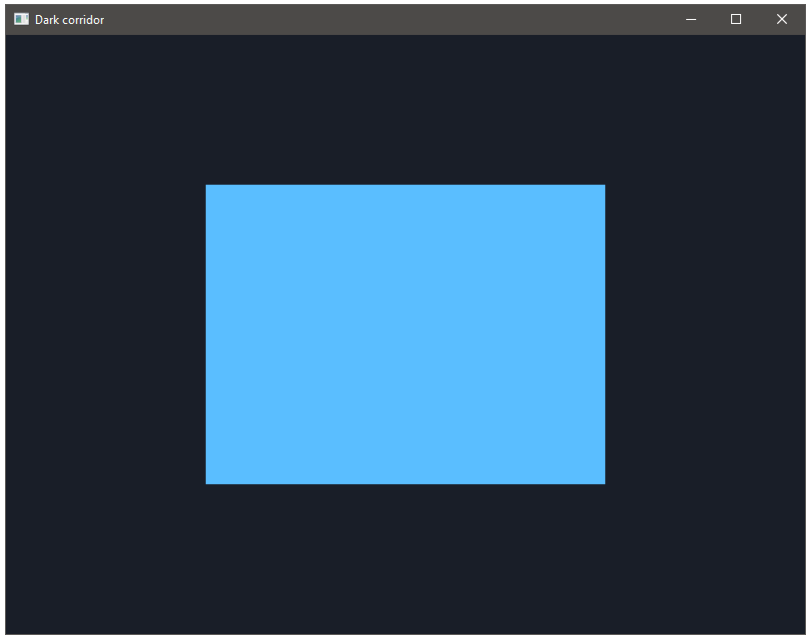



Рис. 3.5: Тестова програма, що використовує SDL3.

Залишилось нам тільки зрозуміти, як працює модуль `sdl3`. Давайте розглянемо детальніше його довільну підпрограму:

Pascal

```
function SDL_CreateWindow(
  title: PAnsiChar;
  w: c_int;
  h: c_int;
  flags: SDL_WindowFlags): PSDL_Window; cdecl; external SDL3_LIB;
```

Як бачимо, функція `SDL_CreateWindow` приймає чотири параметри. `title` — це назва вікна, `w` і `h` — ширина і висота вікна, про опції `flags` ми будемо говорити детальніше у наступному розділі. Якщо ви подивитесь на тип `PSDL_Window`, то ви помітите, що це просто вказівник на запис вікна (`PSDL_Window` — це псевдонім типу `^TSDL_Window`).

А далі ідуть дві нові для нас конструкції: `cdecl` і `external SDL3_LIB`.

Почнемо із другої, бо тут все простіше. `SDL3_LIB` — це константа, що тримає в собі назву файлу динамічної бібліотеки SDL3. Розглянувши де-

тальніше цю константу, ви помітите, що її значення різне для Windows, Linux, MacOS. `external` дозволяє явно вказати назву тої спільної бібліотеки, у якій знаходиться реалізація функції `SDL_CreateWindow`.

Тепер розглянемо `cdecl`. Якби SDL3 була написана мовою Pascal, `cdecl` нам би навіть не знадобився. Проте SDL3 написана на **C**. «Внутрішня кухня» того, як викликаються функції в мові C відрізняється від «внутрішньої кухні» Pascal. `cdecl` прямо каже, що ми, коли викликатимемо функції SDL3, будемо спілкуватись із бібліотекою тими правилами, які встановлені саме мовою C.

> **Примітка**
>
> Правила, яким повинна слідувати програма при виклику функцій, називаються **конвенцією виклику** (англ. *Calling convention*).

Саме тому ми і використовуємо тип `c_int` для параметрів `w` і `h`: вони в мові C мають тип `int`, тому `c_int` в модулі `types` буквально означає «тип `int` мови C».

Головне для нас — нам не потрібно думати про те **як реалізовані** підпрограми всередині SDL3. Ми ними можемо просто користуватись!

Ще одною перевагою спільних бібліотек є те, що файл одної бібліотеки може використовуватись багатьма програмами одночасно.

## 3.24 Висновки

У цьому розділі ми вивчали мову Pascal. Як бачите, при своїй простоті вона пропонує багато можливостей, щоб навіть ті, хто тільки вивчає основи програмування, могли розробити досить складні програми.

Але ми не лише вчили мову. Ми також створили файли, які в майбутньому сформують «скелет» нашого рушія, а саме: `game.pas`, `types.pas`, `vectors.pas` і `sdl3.pas`.

У наступному розділі ми, власне, і почнемо по-справжньому серйозну розробку...

Розділ 4

# Ігрова логіка

У цьому розділі ми нарешті почнемо нашу довгоочікувану розробку.

По-перше, ми зробимо налаштування нашого проекту. Оскільки ми хочемо, щоб наш проект був підтримуваним, ми повинні дотримуватись деяких стандартів, що стосуються не стільки якості самого коду, скільки якості проекту в цілому. Подивіться на ту папку, в яку ви зберігали тестові файли в попередньому розділі: вона перетворилась на звалище! В одному місці там і файли коду, і файли динамічних бібліотек, і виконувані файли, а також різні тимчасові файли. Такого ми хочемо уникнути. Для цього ми налаштуємо **систему збірки** (англ. *Build system*), а також розділимо наш проект на папки.

По-друге, ми почнемо розробку! У цьому розділі ми сфокусуємося на ігровій логіці: ми реалізуємо переміщення персонажа по карті, а також опишемо логіку перевірки того, де саме промені перетинають стіни. Ми поки що не будемо відображати світ в 3D. Ми будемо малювати карту (у класичному Doom клавішею ⇆Tab можна було перемикатися між видом з першої особи та видом карти — ми зробимо щось схоже). Наприкінці цього розділу ви зможете переміщувати персонажа по карті і бачити, як промені знаходять стіни.

## 4.1 Налаштування проекту

Проект ми почнемо із чистого листа, а точніше із чистої папки. Перш за все, створіть нову папку і дайте їй назву вашої гри. Я назву свою гру

«Dark Corridor» (я ніколи не вмів придумувати цікаві назви). Ця папка називатиметься **кореневою папкою** (англ. *Root folder*), тому що вона в собі міститиме всю структуру нашого проекту.

Тепер всередині папки створіть дві нові папки: `src` і `libs`. У папку `src` перемістіть чотири файли, які ми розробили у попередньому розділі: `types.pas`, `vectors.pas`, `game.pas` і `sdl3.pas`.

У папку `libs` перемістіть файл вашої динамічної бібліотеки SDL3.

Оскільки наш проект буде складнішим, ніж просто пара файлів, нам би хотілося, щоб, по-перше, він не захаращувався різними незрозумілими файлами, а, по-друге, хотілося б полегшити його збірку. Набридає постійно вводити одну й ту саму команду `fpc -Px86_64 src/game.pas`. Нам варто подумати про те, як цей процес автоматизувати. І для цього використовуються **системи збірки**.

Однією із найпростіших систем збірки є xmake. У ній все просто: невеличкий код мовою програмування Lua виконує важливу роботу: визначає папку, у яку треба скомпілювати програму під потрібну платформу, викликає команди компіляції, переміщує динамічні бібліотеки в потрібну папку.

Завантажити і встановити на свою систему можна по інструкції:

https://xmake.io/guide/quick-start.html

Наступний крок — створити файл `xmake.lua` у кореневій папці, і додати туди такий код:

**Lua** xmake.lua

```lua
-- Власне правило для компіляції Pascal-файлів,
-- оскільки xmake не підтримує Pascal "з коробки".
rule("pascal")
    set_extensions(".pas")
    on_build(function (target)
        import("core.project.config")
        local sourcefile = target:sourcefiles()[1]
        local targetfile = target:targetfile()
        local targetdir = target:targetdir()
```

```
        -- Очищуємо папку збірки перед компіляцією
        if os.isdir(targetdir) then
            os.rm(targetdir)
        end

        os.mkdir(targetdir)

        -- Викликаємо FPC; вкажіть свою платформу після -P
        os.execv("fpc", {
            "-o" .. targetfile,
            "-Px86_64",
            sourcefile
        })
    end)
rule_end()

-- Замініть назву на назву вашої гри
target("DarkCorridor")
    set_kind("binary")
    add_rules("pascal")
    add_files("src/game.pas")

    -- Копіюємо динамічну бібліотеку SDL3
    -- поруч із виконуваним файлом
    after_build(
        function(target)
            local targetdir = target:targetdir()

            os.cp("libs/*.so", targetdir)
            os.cp("libs/*.dll", targetdir)
            os.cp("libs/*.dylib", targetdir)
        end)
```

**Примітка**

Виклик `os.rm(targetdir)` повністю очищує папку збірки перед кожною компіляцією — це зручно для «чистої» збірки, але не зберігайте там нічого важливого вручну. Достатньо вказати в `add_files` головний файл (`src/game.pas`): Free Pascal сам підтягне інші модулі згідно з `uses`.

Із цим скриптом ми можемо забути про прямий запуск FPC із консолі: тепер вся збірка проекту автоматизована!

Для того, щоб скомпілювати проект, достатньо викликати команду. Зверніть увагу: вводити її потрібно, перебуваючи у **кореневій папці проекту**, тобто в тій папці, де лежить файл `xmake.lua`.

Shell

```
xmake
```

Після цього з'явиться папка із виконуваним файлом і динамічною бібліотекою автоматично. Ще одна перевага — разом із файлами вихідного коду немає різного сміття у вигляді проміжних файлів.

Крім того, xmake дозволяє швидко запускати програму. Достатньо ввести команду

Shell

```
xmake run
```

На цьому етапі структура вашого проекту має виглядати приблизно так:

```
DarkCorridor/
├── xmake.lua
├── src/
│   ├── game.pas
│   ├── sdl3.pas
│   ├── types.pas
│   └── vectors.pas
├── libs/
│   └── SDL3.dll
└── build/
    └── ...
```

**Примітка**

Назва кореневої папки у вас може бути іншою — такою, яку ви обрали для своєї гри. Також у папці `libs` буде лише **один** файл динамічної бібліотеки: `SDL3.dll` для Windows, `libSDL3.so` для Linux або `libSDL3.dylib` для macOS.

## 4.2 Що відбувається в game.pas?

Маючи гарно налаштований проект, ми можемо легко збирати програму. xmake запустить команду, що компілює файл `game.pas` за нас.

Але що відбувається у цьому файлі? У частині про Спільні бібліотеки ми вже написали цей код, але відклали пояснення на потім. Час розібратись!

У розділі про Динамічну пам'ять ми обговорювали функції `New` і `Dispose`, які відповідальні за виділення і очищення пам'яті в купі.

Схожим чином працюють ресурси в SDL. Ми повинні виділити динамічну пам'ять для кількох ресурсів, а коли вони перестануть бути потрібними, звільнити їх. Але всередині SDL використовується власна стратегія управління ресурсами, тому ми управляємо ними не за допомогою функцій

`New` і `Dispose`, а за допомогою функцій, які надає сама бібліотека SDL.

### 4.2.1 Ініціалізація SDL

У першу чергу, нам потрібно ініціалізувати саму бібліотеку SDL. Функція `SDL_Init` робить саме це. Вона ініціалізує **глобальні** ресурси, до яких ми, як користувачі бібліотеки, не матимемо доступу. Вона повертає `SDL_TRUE` у випадку успіху і `SDL_FALSE` у випадку невдалої ініціалізації. У другому випадку ми аварійно завершимо програму із допомогою функції `Halt`. Для того, щоб звільнити глобальні ресурси, виділені бібліотекою, користуємось функцією `SDL_Quit`.

У `game.pas` ця частина виглядає ось так:

Pascal game.pas

```
if SDL_Init(SDL_INIT_VIDEO or SDL_INIT_EVENTS) <> SDL_TRUE then
begin
    WriteLn('SDL_Init failed: ', SDL_GetError);
    Halt(1);
end;

...

SDL_Quit;
```

**Увага**

Зверніть увагу, що у виразі `SDL_INIT_VIDEO or SDL_INIT_EVENTS` використовується оператор `or`. Тут `or` не означає «або одне, або інше». Уявіть бланк із галочками: кожна константа — це одна галочка, а `or` дозволяє поставити кілька галочок одночасно. Тобто цей вираз означає «увімкнути і відео, і обробку подій».

**Примітка**

Зверніть увагу, що в бібліотеці SDL є власний тип даних, що відповідає за булеві значення: він називається `SDL_Bool`, але насправді це просто псевдонім до типу цілого числа, для якого визначені константи: `SDL_TRUE`, яка дорівнює одиниці, і `SDL_FALSE`, яка дорівнює нулю. Така сама «внутрішня кухня» влаштована і у типі `Boolean` в `Pascal`.

### 4.2.2 Ініціалізація вікна

Дивимось далі в наш файл `game.pas` ... Нам потрібне вікно, у якому будемо малювати зображення нашої гри. Але доступ до ресурсу вікна, яке ми створимо, у нас вже буде: ми отримаємо вказівник на нього. Функція `SDL_CreateWindow`, власне, і виділяє ресурс, після чого повертає вказівник на ресурс вікна у разі успіху, або `nil` у разі невдачі.

Pascal game.pas

```
...
window := SDL_CreateWindow(gameName, windowWidth, windowHeight, 0);
if window = nil then
begin
  WriteLn('SDL_CreateWindow failed: ', SDL_GetError);
  SDL_Quit;
  Halt(1);
end;

...

SDL_DestroyWindow(window);
SDL_Quit;
```

**Увага**

Зверніть увагу, як ми звільняємо ресурси, а саме в зворотному порядку: ресурс, що виділяється першим, звільняється останнім, а ресурс, що виділяється останнім, — навпаки, звільняється першим.

Також зверніть увагу, що `SDL_Quit` ми викликаємо незалежно від того, успішно ми виділили ресурс вікна, чи ні.

### 4.2.3 Ініціалізація рендерера

Саме по собі вікно — це лише порожня рамка. Щоб малювати у ньому, нам потрібен ще один ресурс — **рендерер** (англ. *renderer*). Рендерер — це об'єкт, який відповідає за все малювання: він уміє заповнювати вікно кольором, малювати лінії, прямокутники та виводити текстури.

Створюємо рендерер за допомогою функції `SDL_CreateRenderer`, якій передаємо вказівник на вікно. Якщо створити рендерер не вдалося, функція поверне `nil`.

Pascal game.pas

```
...
renderer := SDL_CreateRenderer(window, nil);
if renderer = nil then
begin
  WriteLn('SDL_CreateRenderer failed: ', SDL_GetError);
  SDL_DestroyWindow(window);
  SDL_Quit;
  Halt(1);
end;

...

SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);
```

```
SDL_Quit;
```

**Примітка**

Другий параметр `SDL_CreateRenderer` — це назва конкретного графічного бекенду, який відповідатиме за рендеринг (наприклад, `'opengl'` або `'direct3d'`). Якщо ви геймер, ці слова можуть бути вам знайомі. Якщо передати `nil`, SDL обере найкращий доступний бекенд автоматично.

### 4.2.4 Ігровий цикл

Якби ви запустили програму, яка лише ініціалізує бібліотеку, вікно і рендерер, ви б побачили, як дуже швидко з'являється і пропадає вікно, а потім програма швидко завершується.

Це так працює тому, що програма виконує рівно те, що їй і сказали: «ініціалізуй все, звільни все, завершись».

Для того, щоб програма не закривалась, потрібен процес, який буде тривати доти, доки користувач не натисне «хрестик». Ми повинні постійно перевіряти, чи не натискає користувач кнопку із «хрестиком», і тільки тоді закрити вікно. Для цього ми і використовуємо цикл:

Pascal game.pas

```
...

running := True;
while running do
begin
  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    if event.event_type = SDL_EVENT_QUIT then
      running := False;
```

```
  end;

  ...

  SDL_Delay(16);
end;

...
```

Ми створили змінну `running`. Як тільки вона отримає значення `False`, цикл завершиться, і програма перейде до стадії звільнення ресурсів.

Всередині нашого циклу є ще один цикл. Його потрібно розглянути детальніше. В SDL є система **подій** (англ. *Events*). Подіями є будь-які дії користувача: натиснення кнопки клавіатури, нахилення стіку геймпада, рух миші, чи, як у нашому випадку, закриття вікна. За один крок основного циклу може накопичитися кілька подій, тому ми повинні перевірити їх усі. Для цього ми використовуємо внутрішній цикл із функцією `SDL_PollEvent`: вона забирає одну подію з глобальної черги та повертає `SDL_TRUE`, якщо подія була, або `SDL_FALSE`, якщо черга порожня. Сама інформація про подію записується у змінну `event` — ми передаємо її адресу за допомогою оператора `@`. Далі нам залишається перевірити, чи є подія закриттям вікна. Якщо так, то наступної ітерації циклу вже не відбудеться: `running` дорівнюватиме `False`.

`SDL_Delay(16)` змушує програму зачекати 16 мілісекунд перед наступною ітерацією циклу. Без цього виклику цикл крутився б із високою швидкістю, навантажуючи процесор. Значення 16 мс приблизно відповідає 60 FPS ($1000 \div 60 \approx 16.7$), але цей код ми ще покращимо.

### 4.2.5 Малювання фігур

Нарешті, ми можемо малювати. Код нижче малює прямокутник голубого кольору на фоні темно-синього фону:

Pascal game.pas

```
...

rect.x := 200;
rect.y := 150;
rect.w := 400;
rect.h := 300;

running := True;
while running do
begin
  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    if event.event_type = SDL_EVENT_QUIT then
      running := False;
  end;

  SDL_SetRenderDrawColor(renderer, 25, 30, 40, 255);
  SDL_RenderClear(renderer);

  SDL_SetRenderDrawColor(renderer, 90, 190, 255, 255);
  SDL_RenderFillRect(renderer, @rect);

  SDL_RenderPresent(renderer);

  SDL_Delay(16);
end;

...
```

Перед циклом ми задаємо координати і розміри прямокутника: `x` і `y` — це позиція лівого верхнього кута, а `w` і `h` — ширина та висота.

Тепер подивимося на те, що відбувається всередині циклу після обробки подій. Малювання кадру складається з трьох етапів:

1. **Очищення екрану.** Функція `SDL_SetRenderDrawColor` встановлює поточний колір малювання. Вона приймає чотири числа від 0 до 255: червоний, зелений, синій та **альфа-канал** (прозорість, де 255 — повністю непрозорий). Цей формат називається **RGBA**. Після цього `SDL_RenderClear` заповнює **весь** екран цим кольором — так ми стираємо попередній кадр.
2. **Малювання об'єктів.** Ми знову викликаємо `SDL_SetRenderDrawColor`, цього разу з голубим кольором, і малюємо заповнений прямокутник функцією `SDL_RenderFillRect`.
3. **Показ кадру.** Функція `SDL_RenderPresent` виводить все намальоване на екран **відразу**! До цього моменту ми малювали у невидимому буфері — це потрібно для того, щоб користувач не бачив процес малювання покроково, а бачив лише готовий кадр.

> **Примітка**
>
> Техніка, коли малювання відбувається у прихованому буфері, а потім готовий кадр показується користувачу, називається **подвійною буферизацією** (англ. *Double buffering*). Без неї зображення на екрані «мерехтіло» б, бо користувач бачив би, як фігури малюються одна за одною.

Зверніть увагу на порядок: спочатку очищуємо, потім малюємо, потім показуємо. Цей порядок повторюється кожну ітерацію циклу, тобто кожен кадр.

## 4.3 Правильний підрахунок дельта-часу

Розглянемо детальніше код нашого ігрового циклу, який все ж має один суттєвий недолік.

Pascal game.pas

```pascal
...

running := True;
while running do
begin
  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    if event.event_type = SDL_EVENT_QUIT then
      running := False;
  end;

  ...

  SDL_Delay(16);
end;

...
```

Хоч цей код і працює, давайте пригадаємо частину про Час і спробуємо визначити **дельта-час** $\Delta t$.

Інтуїтивно здається, наче $\Delta t = 16$ мс $= 0.016$ с: ми ж між кожним кадром чекаємо рівно 16 мілісекунд, тому здається наче $\Delta t$ дорівнює цьому значенню. Але, крім виклику функції `SDL_Delay` у нашому ігровому циклі викликається ще цілий ряд функцій: для обробки подій, для малювання, а в майбутньому — і для оновлення стану нашого світу. **І час виконання цих функцій не нульовий**, а дорівнює:

$$\Delta t \approx t_{\text{frameprocess}} + 0.016 \text{ с}.$$

Час $t_{\text{frameprocess}}$, тобто час виконання всього в поточному кадрі, може складати кілька мілісекунд, що є помітною частиною від 16 мс і може суттєво спотворити час у нашій грі.

Саме тому час $\Delta t$ ми повинні **не встановлювати вручну, а рахувати**. Давайте спробуємо це зробити.

В першу чергу, створимо дві нові константи:

Pascal game.pas

```
const
  ...

  targetFps = 60;
  targetDeltaTimeMilliseconds: u64 = Round(1000.0 / targetFps);
```

Перша константа, `targetFps`, створена лише для нас (для програмістів), а не для комп'ютера. Нам як геймерам і розробникам просто ментально легше працювати із поняттям FPS, аніж із $\Delta t$. Тому `targetFps` використовується тільки для того, щоб порахувати значення константи `targetDeltaTimeMilliseconds`. А ця константа уже значно важливіша для програми! Вона позначає те значення $\Delta t$, яке ми **хочемо** досягти!

Тепер ми повинні навчитись рахувати, скільки часу тривав один кадр. Якщо говорити простими словами, то ми беремо годинник, фіксуємо час на початку виконання першого кадру, потім фіксуємо час на початку виконання другого кадру і шукаємо різницю між двома моментами часу. Роль такого годинника бере функція `SDL_GetTicks`. Подивимось, як це виглядає.

Для фіксації часу нам потрібні дві нові змінні:

Pascal game.pas

```
var
  ...

  {який момент часу був на початку попереднього кадру?}
  lastFrameTimeMillis: u64;

  {скільки часу пройшло із моменту попереднього кадру?}
  deltaTimeMillis: u64;
```

Тепер спробуємо у нашому ігровому циклі рахувати дельта-час.

Pascal game.pas

```
running := True;
lastFrameTimeMillis := SDL_GetTicks;

while running do
begin
  deltaTimeMillis := SDL_GetTicks - lastFrameTimeMillis;
  lastFrameTimeMillis := lastFrameTimeMillis + deltaTimeMillis;

  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    if event.event_type = SDL_EVENT_QUIT then
      running := False;
  end;

  SDL_SetRenderDrawColor(renderer, 25, 30, 40, 255);
  SDL_RenderClear(renderer);

  SDL_SetRenderDrawColor(renderer, 90, 190, 255, 255);
  SDL_RenderFillRect(renderer, @rect);

  SDL_RenderPresent(renderer);
end;
```

Цей код вже рахує $\Delta t$, але має одну проблему: якщо кадр виконається дуже швидко (наприклад, за 2 мс), то цикл крутитиметься із занадто високою швидкістю, даремно навантажуючи процесор. Нам потрібно обмежити частоту кадрів: якщо кадр завершився швидше, ніж наша ціль ( `targetDeltaTimeMilliseconds` ), ми повинні зачекати решту часу. Ось покращений варіант:

Pascal game.pas

```
running := True;
lastFrameTimeMillis := SDL_GetTicks;

while running do
begin
  deltaTimeMillis := SDL_GetTicks - lastFrameTimeMillis;
  if deltaTimeMillis < targetDeltaTimeMilliseconds then
  begin
    SDL_Delay(targetDeltaTimeMilliseconds - deltaTimeMillis);
    deltaTimeMillis := targetDeltaTimeMilliseconds;
  end;
  lastFrameTimeMillis := lastFrameTimeMillis + deltaTimeMillis;

  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    if event.event_type = SDL_EVENT_QUIT then
      running := False;
  end;

  SDL_SetRenderDrawColor(renderer, 25, 30, 40, 255);
  SDL_RenderClear(renderer);

  SDL_SetRenderDrawColor(renderer, 90, 190, 255, 255);
  SDL_RenderFillRect(renderer, @rect);

  SDL_RenderPresent(renderer);
end;
```

Розглянемо, що тут відбувається. На початку кожної ітерації ми рахуємо, скільки часу минуло з попереднього кадру:

```
deltaTimeMillis := SDL_GetTicks - lastFrameTimeMillis
```

Якщо цей час менший за наше цільове значення (тобто кадр виконав-

ся швидше, ніж потрібно), ми «досипаємо» залишок часу за допомогою `SDL_Delay` і встановлюємо `deltaTimeMillis` рівним цільовому значенню. Якщо ж кадр тривав довше за ціль (наприклад, через складну сцену), ми нічого не чекаємо і зберігаємо реальний час кадру. Після цього оновлюємо `lastFrameTimeMillis` для наступної ітерації. Далі йде обробка подій, потім ігрова логіка (яку ми додамо пізніше), і нарешті — рендеринг. Такий порядок гарантує, що `deltaTimeMillis` завжди доступний для ігрової логіки.

Тепер у нас є коректно обчислений дельта-час, який ми зможемо використовувати для переміщення персонажа та інших обчислень, що залежать від часу.

Але у змінній `deltaTimeMillis` є одна проблема: нею просто незручно користуватись. Проблема у тому, що нам, як програмістам, незручно оперувати мілісекундами. Швидкість ми вимірюємо в метрах за секунду (або, у випадку гри, у **одиницях за секунду**), а не за мілісекунду. Тому давайте створимо ще одну змінну типу `f32`, у яку ми запишемо дельта-час саме в секундах.

Тому створюємо нову змінну:

Pascal game.pas

```pascal
var
  ...

  {який момент часу був на початку попереднього кадру?}
  lastFrameTimeMillis: u64;

  {
  скільки часу у МІЛІСЕКУНДАХ пройшло
  із моменту попереднього кадру?
  }
  deltaTimeMillis: u64;

  {
```

```
  скільки часу у СЕКУНДАХ пройшло
  із моменту попереднього кадру?
  }
  deltaTimeSeconds: f32;
```

А після того, як ми повністю визначили `deltaTimeMillis`, рахуємо дельта-час у секундах:

Pascal game.pas

```
deltaTimeSeconds := deltaTimeMillis * 0.001;
```

Тепер ми можемо порухати наш прямокутник. Давайте скажемо йому рухатись вправо зі швидкістю 30 одиниць за секунду. Для цього у нашому ігровому циклі потрібно додати рядок коду:

Pascal game.pas

```
rect.x := rect.x + 30 * deltaTimeSeconds;
```

Тепер наш прямокутник рухається!

## 4.4 Новий модуль — Ігровий менеджер

У попередніх частинах ми реалізували виділення ресурсів, ігровий цикл та коректний розрахунок $\Delta t$. Звучить круто, але давайте поглянемо на **весь** наш файл `game.pas` в цілому:

Pascal game.pas

```
program game;

uses
    sdl3, types;

const
    gameName = 'Dark corridor';
```

```
    windowWidth = 800;
    windowHeight = 600;

    targetFps = 60;
    targetDeltaTimeMilliseconds: u64 = Round(1000.0 / targetFps);

var
    window: PSDL_Window;
    renderer: PSDL_Renderer;
    event: SDL_Event;
    running: Boolean;
    rect: SDL_FRect;

    {який момент часу був на початку попереднього кадру?}
    lastFrameTimeMillis: u64;

    {
    скільки часу у МІЛІСЕКУНДАХ пройшло
    із моменту попереднього кадру?
    }
    deltaTimeMillis: u64;

    {
    скільки часу у СЕКУНДАХ пройшло
    із моменту попереднього кадру?
    }
    deltaTimeSeconds: f32;

begin
    if SDL_Init(SDL_INIT_VIDEO or SDL_INIT_EVENTS) <> SDL_TRUE then
    begin
        WriteLn('SDL_Init failed: ', SDL_GetError);
        Halt(1);
```

```
  end;

  window := SDL_CreateWindow(
    gameName,
    windowWidth,
    windowHeight,
    0);

  if window = nil then
  begin
      WriteLn('SDL_CreateWindow failed: ', SDL_GetError);
      SDL_Quit;
      Halt(1);
  end;

  renderer := SDL_CreateRenderer(window, nil);
  if renderer = nil then
  begin
      WriteLn('SDL_CreateRenderer failed: ', SDL_GetError);
      SDL_DestroyWindow(window);
      SDL_Quit;
      Halt(1);
  end;

  rect.x := 200;
  rect.y := 150;
  rect.w := 400;
  rect.h := 300;

  running := True;
  lastFrameTimeMillis := SDL_GetTicks;

  while running do
```

```
    begin
        deltaTimeMillis := SDL_GetTicks - lastFrameTimeMillis;
        if deltaTimeMillis < targetDeltaTimeMilliseconds then
        begin
            SDL_Delay(targetDeltaTimeMilliseconds - deltaTimeMillis);
            deltaTimeMillis := targetDeltaTimeMilliseconds;
        end;
        lastFrameTimeMillis := lastFrameTimeMillis + deltaTimeMillis;
        deltaTimeSeconds := deltaTimeMillis * 0.001;

        while SDL_PollEvent(@event) = SDL_TRUE do
        begin
            if event.event_type = SDL_EVENT_QUIT then
                running := False;
        end;

        rect.x := rect.x + 30 * deltaTimeSeconds;

        SDL_SetRenderDrawColor(renderer, 25, 30, 40, 255);
        SDL_RenderClear(renderer);

        SDL_SetRenderDrawColor(renderer, 90, 190, 255, 255);
        SDL_RenderFillRect(renderer, @rect);

        SDL_RenderPresent(renderer);
    end;

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit;
end.
```

Бачите проблему цього коду? Зверніть увагу, наскільки **величезним** став головний блок програми. Там відбувається все: ініціалізація і звіль-

нення ресурсів, аварійне завершення програми, ігровий цикл, управління станом світу (у нашому випадку, прямокутника), малювання і все інше. Загалом, ми повністю порушили те правило, яке проголосили раніше: **усі підпрограми повинні мати лише одну відповідальність**.

Цю помилку треба виправляти, і терміново. Для цього ми створимо новий модуль, що відповідатиме за всі аспекти гри в цілому. Тому ми назвемо його `gameManager`, тобто ігровим менеджером. У ньому будуть окремі підпрограми для всіх аспектів гри окремо.

Тому давайте створимо новий файл із іменем `gameManager.pas`, у якому буде створений новий модуль. Тепер почнемо розробляти наш модуль. В першу чергу розглянемо його інтерфейс:

Pascal gameManager.pas

```pascal
unit gameManager;

interface

uses types, vectors, sdl3;

type
    {
    Запис, який тримає інформацію про дії, які зробив
    користувач.

    Поки все, що може користувач зробити ---
    вийти із програми.
    }
    TGameEvents = record
        exitApp: Boolean;
    end;

    {
    Запис, який зберігає всю інформацію про гру:
```

```
    стан світу, розмір вікна, ресурси від SDL.
    }
    TGameManager = record
        windowSize: TVector;
        targetDeltaTimeMilliseconds: u64;
        events: TGameEvents;
        window: PSDL_Window;
        renderer: PSDL_Renderer;
    end;

    {
    Вказівник на запис ігрового менеджера.
    }
    PGameManager = ^TGameManager;

    {
    Параметри гри. Це ті параметри, із допомогою
    яких програміст може конфігурувати гру.
    }
    TGameManagerCreateParameters = record
        gameName: PChar;
        windowWidth: c_int;
        windowHeight: c_int;
        targetFps: i32;
    end;

    {
    Можливі варіанти завершення функції ініціалізації
    гри.
    }
    TGameManagerCreateStatus = (
        {Гра успішно ініціалізувалась}
        Ok,
```

```
        {Невдала ініціалізація SDL}
        SdlNotInitialized,

        {Невдала ініціалізація вікна}
        SdlWindowNotInitialized,

        {Невдала ініціалізація рендерера}
        SdlRendererNotInitialized
    );

{
Ініціалізація гри. У разі успіху змінна `gm` буде
заповнена даними гри, а функція поверне `Ok`.

Інакше функція просто поверне статус помилки.
}
function GameManagerInitialize(
    const params: TGameManagerCreateParameters;
    var gm: TGameManager
): TGameManagerCreateStatus;

{ Звільнення ресурсів гри }
procedure GameManagerDestroy(var gm: TGameManager);

{ Запуск ігрового циклу }
procedure GameManagerRun(var gm: TGameManager);

implementation

{ TODO: Тут має бути реалізація функцій інтерфейсу. }

end.
```

Розглянемо інтерфейс по черзі.

**Запис** `TGameEvents`**.** Це невелика «коробка» для того, що *зробив* користувач між кадрами: натиснув клавішу, закрив вікно тощо. Поки ми зберігаємо лише один прапорець `exitApp`: якщо він `True`, ігровий цикл має завершитись. Пізніше сюди легко додати, наприклад, натиснуті стрілки або клавішу ⇆Tab для перемикання виду карти — не змінюючи загальну ідею: спочатку події з SDL перетворюються на поля цього запису, а вже потім логіка гри читає їх і реагує.

**Запис** `TGameManager`**.** Це головний «стан гри» у вигляді одного запису. Тут лежить усе, що має бути «під рукою» під час кадру: розмір вікна (`windowSize`), цільовий крок часу в мілісекундах (`targetDeltaTimeMilliseconds`), згадані події (`events`), а також вказівники на ресурси SDL — вікно і рендерер. Тримати їх разом зручніше, ніж розкидати глобальні змінні по програмі: функції менеджера отримують один параметр `var gm: TGameManager` і завжди знають, з чим працюють.

**Тип** `PGameManager`**.** Це просто «ім'я для вказівника» `^TGameManager`. Він знадобиться пізніше модулю `world`, щоб безпечно перетворювати `Pointer` на типізований вказівник після того, як ми розберемось із циклічними залежностями.

**Запис** `TGameManagerCreateParameters`**.** Це не «стан під час гри», а **налаштування на старті**: назва вікна, ширина й висота, бажана частота кадрів (`targetFps`). Програміст заповнює такий запис один раз перед викликом `GameManagerInitialize`, тож параметри не змішані з внутрішніми полями `TGameManager` і їх простіше змінювати в одному місці.

**Перелік** `TGameManagerCreateStatus`**.** Замість того щоб при кожній помилці викликати `Halt` глибоко всередині модуля, функція ініціалізації повертає **статус**: `Ok`, якщо все вдалось, або одне з значень на кшталт `SdlWindowNotInitialized`, якщо щось пішло не так. Виклик у головній програмі може розгалузитись за цим результатом (написати повідомлення, не запускати цикл тощо) — це зручніше для читання і для подальшого розширення, ніж аварійний вихід «нізвідки».

**Функція** `GameManagerInitialize`**.** Приймає `const params`: вхі-

дний запис конфігурації функція не змінює. Другий параметр — `var gm: TGameManager`: сюди при успіху запишуться заповнені поля менеджера (розмір вікна, `targetDeltaTimeMilliseconds`, вказівники `window` і `renderer` тощо). Повертається значення типу `TGameManagerCreateStatus`.

**Процедура** `GameManagerDestroy`**.** Звільняє ресурси в зворотному порядку до створення (рендерер, вікно, виклик `SDL_Quit` — як ми робили в `game.pas`).

**Процедура** `GameManagerRun`**.** Тут зосередиться ігровий цикл: обробка подій, оновлення стану, малювання кадру та обмеження частоти кадрів.

### 4.4.1 Реалізація модуля

А тепер давайте спробуємо реалізувати наш модуль. Почнемо із функції `GameManagerInitialize`. Тут усе досить просто, і працює майже так само, як і працює в нашому файлі `game.pas`:

Pascal gameManager.pas

```
function GameManagerInitialize(
  const params: TGameManagerCreateParameters;
  var gm: TGameManager
): TGameManagerCreateStatus;
var
  window: PSDL_Window;
  renderer: PSDL_Renderer;
begin
  if SDL_Init(SDL_INIT_VIDEO or SDL_INIT_EVENTS) <> SDL_TRUE then
  begin
    Exit(TGameManagerCreateStatus.SdlNotInitialized);
  end;

  window := SDL_CreateWindow(
    params.gameName,
    params.windowWidth,
```

```
    params.windowHeight,
    0);

  if window = nil then
  begin
    SDL_Quit;

    Exit(TGameManagerCreateStatus.SdlWindowNotInitialized);
  end;

  renderer := SDL_CreateRenderer(window, nil);
  if renderer = nil then
  begin
    SDL_DestroyWindow(window);
    SDL_Quit;

    Exit(TGameManagerCreateStatus.SdlRendererNotInitialized);
  end;

  gm.window := window;
  gm.renderer := renderer;
  gm.windowSize := VectorCartesian(
    params.windowWidth,
    params.windowHeight
  );

  gm.targetDeltaTimeMilliseconds :=
    Round(1000.0 / params.targetFps);

  Exit(TGameManagerCreateStatus.Ok);
end;
```

Проте є кілька відмінностей. У разі невдачі ми не викликаємо `Halt`, а повертаємо статус помилки. Наразі у нас є чотири статуси завершення

функції: одна для успіху, і три для помилок, але ми в майбутньому додамо ще кілька статусів. Зверніть увагу, що `targetDeltaTimeMilliseconds` — уже не константа, а повноцінне поле. Ми це зробили тому, щоб гру можна було конфігурувати без переписування її внутрішньої частини: ви просто у файлі `game.pas` задаєте значення FPS. Такими ж параметрами конфігурації тепер у нас є `windowWidth` і `windowHeight`, а також назва гри `gameName`. Красу такого підходу ви побачите, коли почнете переписувати `game.pas`.

Тепер переходимо до функції `GameManagerDestroy`, відповідальної за звільнення ресурсів. Тут все значно простіше:

**Pascal** gameManager.pas

```pascal
procedure GameManagerDestroy(var gm: TGameManager);
begin
  if gm.renderer <> nil then SDL_DestroyRenderer(gm.renderer);
  if gm.window <> nil then SDL_DestroyWindow(gm.window);
  SDL_Quit;

  {скидаємо невалідні вказівники до значення nil для безпеки}
  gm.renderer := nil;
  gm.window := nil;
end;
```

Як і в `game.pas`, звільняємо ресурси і, оскільки вказівники `renderer` і `window` стали дикими (вони вже не вказують на пам'ять, якою ми володіємо), в цілях безпеки скидаємо їх до значення `nil`.

Ігровий цикл, на відміну від попередніх функцій, зазнає великих змін. Нашою задачею є розділити його на підпрограми, кожна з яких виконує окрему задачу. Розглянемо, як це відбуватиметься:

**Pascal** gameManager.pas

```pascal
procedure GameManagerPollEvents(var gm: TGameManager);
var
  event: SDL_Event;
```

```
begin
  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    if event.event_type = SDL_EVENT_QUIT then
      gm.events.exitApp := True;
  end;
end;

procedure GameManagerUpdate(
  var gm: TGameManager;
  const deltaTimeSeconds: f32
);
begin
  { TODO: Оновити світ }
end;

procedure GameManagerRender(var gm: TGameManager);
begin
  { TODO: Намалювати світ }

  SDL_RenderPresent(gm.renderer);
end;

procedure GameManagerRun(var gm: TGameManager);
var
  lastFrameTimeMillis: u64;
  deltaTimeMillis: u64;
  deltaTimeSeconds: f32;
begin
  gm.events.exitApp := False;
  lastFrameTimeMillis := SDL_GetTicks;

  while not gm.events.exitApp do
```

```
  begin
    deltaTimeMillis := SDL_GetTicks - lastFrameTimeMillis;
    if deltaTimeMillis < gm.targetDeltaTimeMilliseconds then
    begin
      SDL_Delay(gm.targetDeltaTimeMilliseconds - deltaTimeMillis);
      deltaTimeMillis := gm.targetDeltaTimeMilliseconds;
    end;

    lastFrameTimeMillis := lastFrameTimeMillis + deltaTimeMillis;
    deltaTimeSeconds := deltaTimeMillis * 0.001;

    GameManagerPollEvents(gm);
    GameManagerUpdate(gm, deltaTimeSeconds);
    GameManagerRender(gm);
  end;
end;
```

Почнемо із того, що функція `GameManagerRun` у нас — повністю реалізована! У ній виконується абсолютно вся логіка, потрібна ігровому циклу: розрахунок дельта-часу, затримка до цільового значення дельта-часу і виклик функцій, відповідальних за отримання подій (`GameManagerPollEvents`), оновлення світу `GameManagerUpdate` і малювання (`GameManagerRender`), яке надалі будемо називати **рендерингом** (англ. *Rendering*). Дуже важливо ці функції відокремити одну від одної, оскільки ці три стадії повинні обов'язково іти окремо: ми спочатку перевіряємо, що гравець «наклацав», потім — оновлюємо стан світу, а потім — рендеримо все у фінальну картинку.

> **Увага**
>
> Ні в якому разі не можна змішувати оновлення світу і рендеринг! Поганим тоном є впродовж одного кадру спочатку щось намалювати, потім — оновити стан світу, а потім — знов щось намалювати. Це — джерело непередбачуваних помилок у комп'ютерній графіці.

Функція `GameManagerPollEvents` працює так само, як і в `game.pas`. А от функції `GameManagerUpdate` і `GameManagerRender` ми поки що залишимо пустими: прямокутник, який ми малювали в `game.pas`, не повинен оброблятись модулем `gameManager`. За світ і те, що у ньому відбувається, відповідатиме зовсім **інша частина нашої програми**, про що мова піде трішки пізніше.

### 4.4.2 Використання модуля

Ну що, готові побачити, як виглядатиме наша програма `game.pas` після змін, що ми зробили? Дивіться і насолоджуйтесь:

Pascal game.pas

```pascal
program game;

uses gameManager, vectors;

var
  gm: TGameManager;
  gmCreateParams: TGameManagerCreateParameters;
  gmCreateStatus: TGameManagerCreateStatus;

begin
  gmCreateParams.gameName := 'Dark Corridor';
  gmCreateParams.windowWidth := 800;
  gmCreateParams.windowHeight := 600;
  gmCreateParams.targetFps := 60;

  gmCreateStatus := GameManagerInitialize(gmCreateParams, gm);
  if gmCreateStatus <> TGameManagerCreateStatus.Ok then
  begin
    WriteLn('Could not create game manager!');
    Halt(1);
```

```
  end;

  GameManagerRun(gm);
  GameManagerDestroy(gm);
end.
```

Головне — цей файл маленький. Все, що ми у ньому робимо, — це кажемо: «зроби мені гру із такою назвою, таким розміром екрану і таким FPS». Ми не думаємо у цьому файлі про реалізацію **взагалі**! Ми вказуємо лише конфігурацію, і кажемо програмі «запустись».

У цьому і полягає перевага **модульності** програм: ми розбиваємо одну велику і складну задачу на кілька маленьких і простих, а потім ці частинки складаємо, як кубики Lego!

Наразі «ланцюжок» можна уявити так: `game.pas` налаштовує параметри й викликає `gameManager`, а той веде цикл кадрів; далі до нього підключиться `world` (і згодом модулі `player`, `walls` тощо), які відповідатимуть за зміст гри.

## 4.5 Новий модуль — Світ

Ігровий менеджер уже вміє завести вікно, крутити цикл кадрів і розділити події, оновлення та малювання. Але **що саме** ми оновлюємо і **що** малюємо, він поки не знає: той голубий прямокутник з `game.pas` ми свідомо залишили за межами `gameManager`. Тепер настав час повернути його в програму — не як окрему змінну «десь у головному файлі», а як частину цілого.

Уявіть собі аркуш паперу, на якому ви малюєте гру. Прямокутник — це не «сам по собі» об'єкт у порожнечі: він лежить *на* цьому аркуші, має позицію відносно країв, може зіткнутися з іншими фігурами, рухатись по карті. У термінах розробки ігор усе це входить у поняття **світу гри** (англ. *game world*): набір об'єктів, їхніх координат, правил руху та (згодом) стін і клітинок карти. Менеджер питає: «що змінилось за цей кадр?», а **світ**

відповідає: «ось де зараз персонаж, ось що з ним сталося, ось що треба намалювати зверху карти».

Тому ми винесемо опис нашого прямокутника (і все, що з ним пов'язано) в окремий модуль `world` — **світ**. У ньому зосередиться ігрова «геометрія» та логіка того, що існує в грі, без зайвих деталей SDL. Менеджер лишатиметься «оркестром», а світ — тим місцем, де живе наш прямокутник і де пізніше з'являться карта, промені та персонаж.

Розглянемо, як виглядатиме інтерфейс такого модуля:

Pascal world.pas

```
unit world;

interface

uses types, vectors, sdl3;

type
  TRectangle = record
    position: TVector;
    size: TVector;
    velocity: TVector;
  end;

  TWorldInitializeParameters = record
    rectanglePosition: TVector;
    rectangleSize: TVector;
    rectangleVelocity: TVector;
  end;

  TWorld = record
    rectangle: TRectangle;
    gmPtr: Pointer;
  end;
```

```
procedure WorldInitialize(
  const params: TWorldInitializeParameters;
  const gmPtr: Pointer;
  var world: TWorld);

procedure WorldDestroy(var world: TWorld);

procedure WorldUpdate(
  var world: TWorld;
  const deltaTimeSeconds: f32);

procedure WorldRender(var world: TWorld);

implementation

{ TODO: Тут має бути реалізація функцій інтерфейсу. }

end.
```

Тут все аналогічно ігровому менеджеру: у нас є функції ініціалізації і звільнення світу, а також — функції оновлення стану світу і рендеру.

Проте ви можете звернути увагу на те, що світ тримає у собі вказівник `gmPtr`. Справа у тому, що всі об'єкти, які будуть знаходитись у нашому ігровому світі, повинні якось рендеритись, тобто їм потрібен буде доступ до поля `renderer` із ігрового менеджера. Гравець при переміщенні повинен знати, де знаходяться стіни, а також повинен знати, які клавіші натискає геймер, щоб мати змогу переміщуватись. Тому вказівник на ігрового менеджера потрібен всім об'єктам: завдяки ньому об'єкти будуть бачити загальний контекст гри: вхідні дані від геймера, інформація про світ, ресурси для малювання тощо.

Але чому тоді `gmPtr` має якийсь дивний тип даних `Pointer`, а не `PGameManager`? Давайте розбиратись...

### 4.5.1 Циклічні залежності в Pascal

Якщо ви спробуєте замінити `Pointer` типом `PGameManager`, попередньо написавши `uses gameManager;` в модулі `world`, **програма не скомпілюється**.

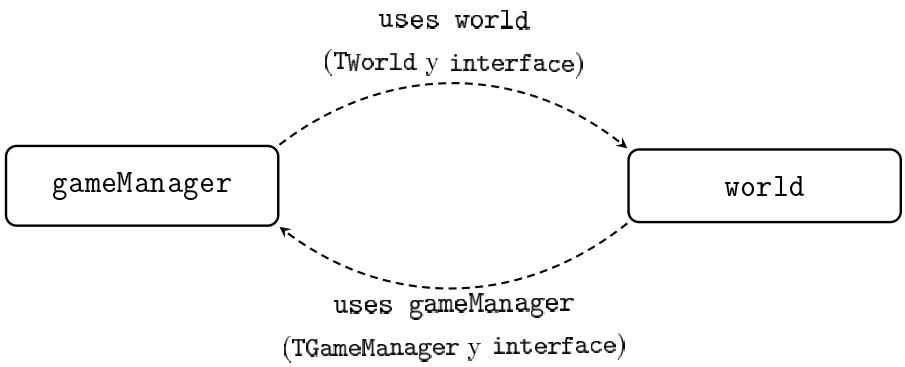



Рис. 4.1: Циклічна залежність

Давайте подумаємо, чому так. Запис `TGameManager` тримає в собі світ (поле `world` типу `TWorld`). Для цього в інтерфейс модуля `gameManager` ми імпортуємо модуль `world`. Поки що все логічно.

Але світу, тобто запису типу `TWorld`, потрібен вказівник на `TGameManager` (це ми уже проговорили в попередній частині). І для цього в інтерфейс модуля `world` ми імпортуємо модуль `gameManager`.

Виходить, що інтерфейси модулів `world` і `gameManager` **залежать один від одного**! Це як проблема «курки та яйця» — компілятор просто не знає, за обробку якого модуля взятись першим. Це і створює так звану проблему **циклічної залежності** (англ. *Circular dependency*), яку компілятор Pascal не може вирішити.

### 4.5.2 Вирішуємо проблему циклічної залежності

Вирішити цю проблему можна, і для цього нам потрібно скористатись одною цікавою особливістю ключового слова `uses`.

Справа в тому, що імпорт модуля в Pascal повністю ігнорує те, що відбувається в `implementation` частині модуля. Саме тому у модулі `world` ми

можемо перенести `uses gameManager;` в `implementation`.

Але тепер нам варто вирішити одну супутню проблему: інтерфейс модуля `world` уже не має доступу до `PGameManager`. На жаль, ідеального рішення цієї проблеми немає, тому замість `PGameManager` ми і пишемо `Pointer`.

Уявіть **конверт без напису**: усередині може бути що завгодно, а в `implementation` ви самі «підписуєте», що це саме `PGameManager`, коли робите перетворення типу.

`Pointer` — це теж тип вказівника. Але він не прив'язаний до якогось типу, тобто він може бути **вказівником на будь-що**. Оскільки всі вказівники мають однаковий розмір (розмір адреси в пам'яті), ми його зможемо приводити до будь-якого вказівника. У нашому випадку, нам треба буде **перетворити** `Pointer` на `PGameManager`. Тому в подальшому, якщо нам потрібно буде користуватись вказівником `gmPtr`, ми спочатку створюватимемо локальну змінну типу `PGameManager`, яка буде вказувати на ігровий менеджер:

```
var gm: PGameManager;
```

А потім перетворюватимемо `Pointer` на потрібний тип.

```
gm := PGameManager(world.gmPtr);
```

Таким чином, ми матимемо змогу працювати із ігровим менеджером.

**Увага**

**Перетворення вказівників — небезпечна операція**! Наприклад, вам нічого не заважає перетворити `Pointer` не на `PGameManager`, а, наприклад, на `^u64`, і далі працювати із цим вказівником як із вказівником на число. Це може створити проблеми цілісності даних: ви маєте ризик серйозно спотворити дані, і програма почне вести себе непередбачувано. **При перетворенні вказівників завжди тримайте в голові типи даних.**

### 4.5.3 Реалізація модуля

Реалізація ігрового світу поки що все значно простіша, ніж реалізація ігрового менеджера.

Функції `WorldInitialize` і `WorldDestroy` досить примітивні:

Pascal world.pas

```pascal
procedure WorldInitialize(
  const params: TWorldInitializeParameters;
  const gmPtr: Pointer;
  var world: TWorld);
begin
  world.rectangle.position := params.rectanglePosition;
  world.rectangle.size := params.rectangleSize;
  world.rectangle.velocity := params.rectangleVelocity;
  world.gmPtr := gmPtr;
end;

procedure WorldDestroy(var world: TWorld);
begin
  {
  немає ресурсів, які необхідно звільнити
  }
end;
```

Ініціалізуючи світ, ми просто переносимо характеристики прямокутника в його стан, зберігаємо вказівник на ігровий менеджер (`gmPtr`), а функція звільнення поки нічого не робить.

У функції `WorldUpdate` ми оновлюємо позицію прямокутника:

Pascal world.pas

```pascal
procedure WorldUpdate(
  var world: TWorld;
  const deltaTimeSeconds: f32);
```

```
var
  rectangleMovement: TVector;
begin
  {
  визначаємо вектор, на який прямокутник має переміститись
  протягом кадру
  }
  rectangleMovement := VectorMulScalar(
    world.rectangle.velocity,
    deltaTimeSeconds
  );

  world.rectangle.position := VectorAdd(
    world.rectangle.position,
    rectangleMovement
  );
end;
```

Цього разу ми вже застосовуємо наші векторні операції для розрахунку вектора переміщення протягом кадру і визначення наступної позиції. Код досить зрозумілий і легко читається.

І, нарешті, залишається функція `WorldRender`. На початку розділу `implementation` модуля `world.pas` (один раз, перед усіма процедурами) додайте `uses gameManager;`, щоб компілятор знав тип `PGameManager`. Далі:

Pascal — world.pas

```
procedure WorldRender(var world: TWorld);
var
  gm: PGameManager;
  sdlRect: SDL_FRect;

begin
  gm := PGameManager(world.gmPtr);
```

```
  SDL_SetRenderDrawColor(gm^.renderer, 25, 30, 40, 255);
  SDL_RenderClear(gm^.renderer);

  {
  Функція SDL_RenderFillRect вимагає типу даних ^SDL_FRect,
  тому нам потрібно привести дані із прямокутника до типу SDL_FRect
  і передати вказівник у функцію
  }
  sdlRect.x := world.rectangle.position.x;
  sdlRect.y := world.rectangle.position.y;
  sdlRect.w := world.rectangle.size.x;
  sdlRect.h := world.rectangle.size.y;

  SDL_SetRenderDrawColor(gm^.renderer, 90, 190, 255, 255);
  SDL_RenderFillRect(gm^.renderer, @sdlRect);
end;
```

### 4.5.4 Використання модуля

Модуль `world` буде використовуватись модулем `gameManager`.

По-перше, **світ** стане полем запису `ігрового менеджера`, а параметри створення світу стануть частиною параметрів створення ігрового менеджера:

Pascal — gameManager.pas

```
{
Запис, який зберігає всю інформацію про гру:
стан світу, розмір вікна, ресурси від SDL.
}
TGameManager = record
  windowSize: TVector;
  targetDeltaTimeMilliseconds: u64;
```

```
  events: TGameEvents;
  window: PSDL_Window;
  renderer: PSDL_Renderer;
  world: TWorld;
end;

{
Параметри гри. Це ті параметри, із допомогою
яких програміст може конфігурувати гру.
}
TGameManagerCreateParameters = record
  gameName: PChar;
  windowWidth: c_int;
  windowHeight: c_int;
  targetFps: i32;
  world: TWorldInitializeParameters;
end;
```

У повному файлі `gameManager.pas` у розділі `interface` також з'явиться рядок `uses types, vectors, sdl3, world;` (бо тут фігурує `TWorld`).

Тепер нам залишається викликати всі чотири функції інтерфейсу світу у відповідних функціях ігрового менеджера:

**Pascal** `gameManager.pas`

```
function GameManagerInitialize(
  const params: TGameManagerCreateParameters;
  var gm: TGameManager
): TGameManagerCreateStatus;
var
  window: PSDL_Window;
  renderer: PSDL_Renderer;
begin
  ...
```

```
  WorldInitialize(params.world, @gm, gm.world);
  Exit(TGameManagerCreateStatus.Ok);
end;

procedure GameManagerDestroy(var gm: TGameManager);
begin
  WorldDestroy(gm.world);

  ...
end;

...

procedure GameManagerRender(var gm: TGameManager);
begin
  WorldRender(gm.world);

  SDL_RenderPresent(gm.renderer);
end;

procedure GameManagerUpdate(
  var gm: TGameManager;
  const deltaTimeSeconds: f32
);
begin
  WorldUpdate(gm.world, deltaTimeSeconds);
end;
```

Це все! Ми тепер лише маємо до нашого файлу `game.pas` додати конфігурацію світу:

Pascal game.pas

```
gmCreateParams.world.rectanglePosition :=
  VectorCartesian(200.0, 150.0);

gmCreateParams.world.rectangleSize :=
  VectorCartesian(400.0, 300.0);

gmCreateParams.world.rectangleVelocity :=
  VectorCartesian(30.0, 0.0);
```

Ми, по суті, зробили все те, що було зроблено в одному файлі `game.pas`, тільки цього разу у нас все зроблено гарніше і охайніше. Кожна частина функціоналу нашої програми лежить на своєму місці. Ми прекрасно бачимо, що де оновлюється, і яким чином рендериться. Тепер ми готові до реалізації логіки світу нашого рушія.

Але пропоную перед тим закріпити знання задачею.

**Задача 20**

Переробіть програму, яку ми написали. Замість голубого прямокутника, намалюйте планетарну систему: зірку в центрі і дві планети, що рухаються навколо неї по круглих орбітах. Для малювання кругів скористуйтесь функцією `SDLHelper_FillDisk` модуля `sdl3`. Її немає серед функцій спільної бібліотеки SDL3, це просто зручна функція, створена мною для вас.

## 4.6 Нові модулі — Гравець і карта

**Увага**

Надалі я не бачу сенсу писати **весь** код усіх модулів. Логіка багатьох підпрограм, що ми будемо реалізовувати в наступних наших модулях, абсолютно аналогічна тому, що ми реалізували в `world`: всі модулі, що відповідальні за ігрові об'єкти, матимуть підпрограми для ініціалізації, оновлення, рендерингу. Ігрові об'єкти також мають конфігуруватись у файлі `game.pas`. На цей момент ви уже повинні мати інтуїтивне розуміння, як це все має реалізовуватись.

Готово! Ми змогли побудувати гарну структуру проекту. Якщо ви ще не виконали вправу Задачу 20, рекомендую це зробити: це чудовий спосіб закріпити матеріал.

Тепер нам знову потрібно зробити наш світ «порожнішим» під наступний крок. Хороша новина: це **останній раз**, коли ми свідомо «обнуляємо» світ до найпростішого каркасу — далі нарешті заповнимо його картою і додамо головного персонажа.

Якщо вам жаль вашу прекрасну сонячну систему, зробіть **резервну копію** проекта в іншу папку (резервні копії корисно робити загалом).

Тепер ми видаляємо із нашого світу усе, що не стосується рейкаст-рушія, тобто практично все. Запис нашого світу і параметри його ініціалізації мають виглядати так:

Pascal world.pas

```
type
  TWorldInitializeParameters = record
  end;

  TWorld = record
    gmPtr: Pointer;
  end;
```

Почнемо із невеличкого нагадування: ми хотіли реалізувати переключення виду, як у класичному **Doom**: якщо геймер натискає кнопку ⇆ Tab,

він бачить карту.

Тому ми трошки змінимо нашу реалізацію рендерингу світу:

Pascal world.pas

```pascal
procedure WorldMapRender(var world: TWorld);
var
  gm: PGameManager;
begin
  gm := PGameManager(world.gmPtr);

  SDL_SetRenderDrawColor(gm^.renderer, 25, 30, 40, 255);
  SDL_RenderClear(gm^.renderer);

  {
  TODO: малювання карти
  }
end;

procedure WorldRender(var world: TWorld);
begin
  WorldMapRender(world);
end;
```

Тепер конкретний рендеринг буде відбуватись у процедурі `WorldMapRender`, а `WorldRender` буде відповідальною за вибір режиму відображення (3D-режим чи режим карти). Поки що у нас лише один режим карти, але 3D-режим ми почнемо реалізовувати у наступному розділі.

Тепер нам треба створити **два нові модулі**. Один ми назвемо `player`, і він відповідатиме за нашого персонажа. Другий ми назвемо `walls`, і він представлятиме «комірки» нашої карти.

Тобто, модель даних світу має виглядати так:

Pascal world.pas

```pascal
type
  TWorldInitializeParameters = record
    walls: TWallsInitializeParameters;
    player: TPlayerInitializeParameters;
  end;

  TWorld = record
    walls: TWalls;
    player: TPlayer;
    gmPtr: Pointer;
  end;
```

Pascal player.pas

```pascal
type
  TPlayerInitializeParameters = record
    position: TVector;
    angle: f32;
    speed: f32;
    angularSpeed: f32;
  end;

  TPlayer = record
    position: TVector; { позиція гравця }
    angle: f32; { кут повороту гравця }

    {
    швидкість переміщення гравця
    (коли натиснута кнопка "вперед" чи "назад")
    }
    speed: f32;
```

```
    {
    швидкість повороту гравця
    (коли натиснута кнопка "вліво" чи "вправо")
    }
    angularSpeed: f32;

    gmPtr: Pointer; { вказівник на ігрового менеджера }
  end;
```

Pascal walls.pas

```
type
  TWallsInitializeParameters = record
    map: array of u8;
    width: isize;
    height: isize;
    tileSize: f32;
  end;

  TWalls = record
    map: array of u8; { карта "клітинок" }
    width: isize; { ширина карти }
    height: isize; { висота карти }
    tileSize: f32; { довжина сторони стіни }

    gmPtr: Pointer; { вказівник на ігрового менеджера }
  end;
```

Тепер нам потрібно подумати, **яким чином ми будемо рендерити** карту і об'єкти на ній ...

### 4.6.1 Світовий і екранний простори

Отож, нам потрібно зробити рендер карти і позиції персонажа на ній.

Давайте подумаємо, де ми будемо малювати персонажа, якщо його ко-

ординати — $\begin{bmatrix} 30 \\ 100 \end{bmatrix}$. Інтуїтивно, хотілося б намалювати персонажа в точці із такими ж координатами на екрані. Але уявіть собі світ, що описується сіткою $1000 \times 1000$! Ми просто не побачимо персонажа, якщо його перемістимо на координати, наприклад $\begin{bmatrix} 10000 \\ 5000 \end{bmatrix}$, тому що вони вийдуть далеко за межі екрану!

Тому координати на екрані — це не те ж саме, що і координати у світі, і нам потрібно придумати, яким чином ми будемо перетворювати світові координати в екранні.

Для цього ми створимо новий модуль, що і буде відповідати за перетворення координат **в режимі карти**. Назвемо цей модуль `mapRendererHelpers`, і матиме він всього одну функцію:

Pascal mapRendererHelpers.pas

```
unit mapRendererHelpers;

interface

uses vectors, types, gameManager;

function MapWorldSpaceToScreenSpace(
  point: TVector;
  gm: PGameManager
): TVector;

implementation

{ TODO: Реалізація інтерфейсу }

end.
```

Тепер нам потрібно подумати, **як** ми хочемо реалізувати наш інтерфейс.

Давайте спочатку вирішимо, що ми хочемо досягти. На карті персонаж має бути завжди у центрі. Крім того, персонаж на карті завжди дивиться вгору, а при повороті ми бачимо поворот не персонажа, а карти навколо нього.

> **Увага**
>
> Зараз буде складний для сприйняття матеріал. Тут треба читати дуже уважно, оскільки в цій частині легко заплутатись.

Перед формулами коротко зафіксуємо **порядок кроків** — його варто тримати в голові, коли читаєте далі:

1. **Зсув:** переводимо світ у систему координат персонажа (віднімаємо позицію персонажа, щоб він опинився в початку координат).
2. **Поворот:** повертаємо світ таким чином, щоб на екрані персонаж дивився **вгору**. Для цього потрібно буде змінити координатний базис.
3. **Зсув на центр вікна:** додаємо половину розміру вікна, щоб центр екрану відповідав позиції персонажа на карті.

Тепер розберемось у математиці. Нехай $\overrightarrow{r}$ — радіус-вектор точки у світових координатах, яку ми хочемо перевести в координати для малювання. Координати персонажа і **одиничний** вектор напрямку його погляду — $\overrightarrow{p}$ і $\overrightarrow{n}$ відповідно. Розмір екрану в пікселях — $W_x$ і $W_y$ по осях.

У нашому світовому просторі є два базисні вектори — $\overrightarrow{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ і $\overrightarrow{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Будь-який вектор можна описати у вигляді лінійної комбінації векторів $\overrightarrow{i}$ та $\overrightarrow{j}$ (тобто як сума добутків цих векторів на дійсні скаляри):

$$\overrightarrow{r} = \begin{bmatrix} r_x \\ r_y \end{bmatrix} = \begin{bmatrix} r_x \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ r_y \end{bmatrix} = r_x \begin{bmatrix} 1 \\ 0 \end{bmatrix} + r_y \begin{bmatrix} 0 \\ 1 \end{bmatrix} = r_x \overrightarrow{i} + r_y \overrightarrow{j}$$

Ну що, почнемо перетворювати координатний простір? Спочатку ми хочемо перемістити початок координат у те місце, де наразі знаходиться

персонаж. Для цього від точки, яку ми переводимо, ми маємо відняти позицію персонажа, поки що все просто:

$$\overrightarrow{r_{\text{rel}}} = \overrightarrow{r} - \overrightarrow{p}$$

Що змінилося в базисі після зсуву? Нічого: вектори $\overrightarrow{i}$, $\overrightarrow{j}$ лишилися векторами $\overrightarrow{i}$, $\overrightarrow{j}$.

Але далі ми хочемо зробити **поворот світу**: базис зміниться — і треба вибрати нові напрямні вектори. Тобто у новому «повернутому» координатному просторі роль векторів $\overrightarrow{i}$ та $\overrightarrow{j}$ візьмуть на себе інші вектори, і саме відносно них ми рахуватимемо нові координати! Позначимо ці вектори $\overrightarrow{v}$ та $\overrightarrow{u}$ відповідно (побачити їх можна на Рис. 4.2).

Нехай базисний вектор $\overrightarrow{u}$ відповідатиме **вертикалі екрану** після повороту. Тоді його потрібно взяти **протилежним** до напрямку погляду $\overrightarrow{n}$, оскільки вісь $y$ напрямлена **вниз**, а ми хочемо, щоб на карті персонаж дивився **вгору**.

$$\overrightarrow{u} = -\overrightarrow{n} = \begin{bmatrix} -n_x \\ -n_y \end{bmatrix}.$$

Вектор $\overrightarrow{v}$ («вправо» на екрані) має бути перпендикулярним до $\overrightarrow{u}$:

$$\overrightarrow{v} = \begin{bmatrix} u_y \\ -u_x \end{bmatrix} = \begin{bmatrix} -n_y \\ n_x \end{bmatrix}.$$

Для того, щоб зрозуміти, як це працює, ви можете глянути на Рис. 4.2: так виглядає координатна система **до повороту**. Тепер поверніть сторінку так, щоб вектор $\overrightarrow{u}$ вказував на вас, на читача. Так виглядатиме координатна система **після повороту**. У будь-якому випадку, персонаж поки що знаходиться у точці $O$.

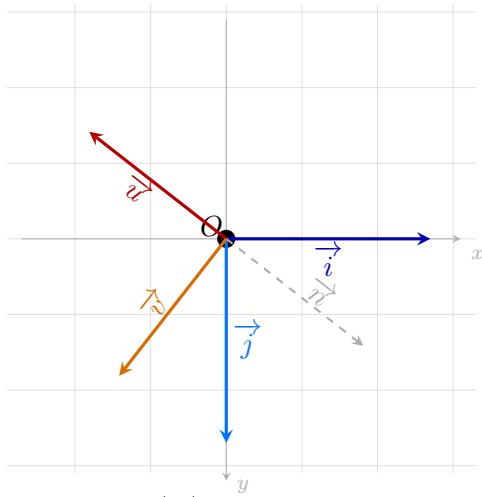



Рис. 4.2: Два базиси з одного початку $O$: світові $\{\vec{i},\vec{j}\}$ ($y$ у грі вниз) і екранні $\{\vec{v},\vec{u}\}$ для мінікарти; $\vec{n}$ — напрямок погляду.

Для повороту карти, тобто переведення координат у новий базис, скористаємось формулою (нижче $r_{\text{rel},x}$ і $r_{\text{rel},y}$ — це компоненти $\vec{r}_{\text{rel}}$, тобто $x$ і $y$ після зсуву до персонажа):

$$\overrightarrow{r_{\text{rotated}}} = \begin{bmatrix} r_{\text{rel},x} v_x + r_{\text{rel},y} v_y \\ r_{\text{rel},x} u_x + r_{\text{rel},y} u_y \end{bmatrix}$$

Нарешті, треба зробити останній крок, а саме перемістити центр координат (тобто персонажа) на середину екрану:

$$\overrightarrow{r_{\text{screen}}} = \overrightarrow{r_{\text{rotated}}} + \frac{1}{2}\begin{bmatrix} W_x \\ W_y \end{bmatrix}.$$

Тепер подивимось на реалізацію:

**Pascal** mapRendererHelpers.pas

```
const
  mapScale = 1;
```

```
function MapWorldSpaceToScreenSpace(
  point: TVector;
  gm: PGameManager
): TVector;
var
  mapCenter: TVector;
  halfScreenSize: TVector;
  pointCenterDiff: TVector;
  basisU: TVector;
  basisV: TVector;
  rotatedScaledPointCenterDiff: TVector;
begin
  { позиція центру екрану в екранній системі координат }
  halfScreenSize := VectorMulScalar(gm^.windowSize, 0.5);

  { позиція центру екрану в світовій системі координат }
  mapCenter := gm^.world.player.position;

  { зсув точки point до системи координат, де координати
  `mapCenter` - (0, 0) }
  pointCenterDiff := VectorSub(point, mapCenter);

  { координати нового базису }
  {
  angle + PI у VectorPolar дає напрям,
  протилежний погляду (-n, <<вгору>> на мінікарті);
  те саме можна отримати,
  обчисливши VectorPolar(1, angle) і помноживши вектор на -1
  }
  basisU := VectorPolar(1, gm^.world.player.angle + pi);
  basisV := VectorCartesian(basisU.y, -basisU.x);

  { перетворення точки до екранного базису }
```

```
  rotatedScaledPointCenterDiff := VectorMulScalar(
    VectorCartesian(
      pointCenterDiff.x * basisV.x + pointCenterDiff.y * basisV.y,
      pointCenterDiff.x * basisU.x + pointCenterDiff.y * basisU.y
    ),
    mapScale
  );

  { зсув точки point до системи координат, де координати
  `mapCenter` - середина екрану }
  MapWorldSpaceToScreenSpace := VectorAdd(
    rotatedScaledPointCenterDiff,
    halfScreenSize
  );
end;
```

Відтепер при рендерингу **завжди потрібно переводити точки із світової системи координат в екранну**, що ми і спробуємо зараз зробити.

### 4.6.2 Рендеримо стіни на карті

Тепер ми готові малювати «клітинки». Функція для їх малювання, на перший погляд, здається нескладною:

Pascal walls.pas

```
procedure WallsMapRender(var walls: TWalls);
var
  i, j: isize;
begin
  for i := 0 to walls.height - 1 do
    for j := 0 to walls.width - 1 do
      TileMapRender(walls, i, j);
end;
```

Ми просто проходимось по кожній клітинці і малюємо її із допомогою допоміжної функції `TileMapRender`.

Але із реалізацією функції `TileMapRender` може виникнути питання ...

Раніше, щоб малювати прямокутник на екрані, ми користувались функцією `SDL_RenderFillRect`. Але вона вміє заповнювати лише прямокутники, **вирівняні за осями** екрану: сторони паралельні горизонталі й вертикалі («як плитка підлоги», не повернута). Після нашого перетворення координат клітинка карти на екрані, загалом кажучи, вже **нахилена**, тому «просто передати ширину й висоту», як у `SDL_FRect`, недостатньо.

Функція `SDL_RenderGeometry` працює інакше: ви передаєте список **вершин** — точок на площині (з координатами вже в тій системі, у якій малюєте, наприклад у пікселях екрану) — і кажете рендереру, як з них скласти **трикутники**. Чому саме трикутники? Бо це найпростіша «плитка» для сучасної графіки: будь-який плоский малюнок можна скласти з трикутників, як мозаїку, а відеокарта дуже швидко вміє такі трикутники заливати кольором або текстурою.

Кожен квадрат клітинки ми представимо як **два** трикутники (розріз по діагоналі): разом вони дають той самий квадрат, але вже з довільним нахилом у просторі екрану. Далі в коді для кожної стіни ми просто обчислимо чотири кути квадрата у екранних координатах і передамо їх у `SDL_RenderGeometry` як два трикутники.

Давайте спробуємо реалізувати це:

Pascal walls.pas

```
procedure TileMapRender(
  var walls: TWalls;
  const i: isize;
  const j: isize
);
var
  gm: PGameManager;
  k: isize;
```

```
  fcolor: f32;

  worldTopLeft: TVector;
  relativeCorners: array[0..3] of TVector;
  worldCorners: array[0..3] of TVector;
  screenCorners: array[0..3] of TVector;
  sdlVertices: array[0..3] of SDL_Vertex;

  {
  Прямокутник представляється як два трикутники.
  У першого --- вершини (0, 1, 2).
  У другого --- вершини (0, 2, 3).

  Ці вершини відповідають вершинам із масиву `sdlVertices`.
  }
  indices: array[0..5] of c_int = (
    {перший трикутник} 0, 1, 2,
    {другий трикутник} 0, 2, 3
  );
begin
  gm := PGameManager(walls.gmPtr);

  if walls.map[i * walls.width + j] > 0 then
    {стіну рендеримо чорним кольором, тобто r, g, b компоненти = min}
    fcolor := 0.0
  else
    {стіну рендеримо білим кольором, тобто r, g, b компоненти = max}
    fcolor := 1.0;

  worldTopLeft := VectorMulScalar(
    VectorCartesian(j, i),
    walls.tileSize
  );
```

```
{ Наскільки кожна вершина зміщена відносно лівого верхнього краю }
relativeCorners[0] := VectorCartesian(0, 0);
relativeCorners[1] := VectorCartesian(walls.tileSize, 0);
relativeCorners[2] :=
  VectorCartesian(walls.tileSize, walls.tileSize);
relativeCorners[3] := VectorCartesian(0, walls.tileSize);

{ Визначаємо координати вершин у світовій системі координат }
for k := Low(relativeCorners) to High(relativeCorners) do
  worldCorners[k] := VectorAdd(worldTopLeft, relativeCorners[k]);

{ Визначаємо координати вершин у екранній системі координат }
for k := Low(worldCorners) To High(worldCorners) do
  screenCorners[k] := MapWorldSpaceToScreenSpace(
    worldCorners[k],
    gm);

{ Представляємо вершини у екранній системі координат у масиві типу
  `array[0..3] of SDL_Vertex` }
for k := Low(screenCorners) To High(screenCorners) do
begin
  sdlVertices[k].position.x := screenCorners[k].x;
  sdlVertices[k].position.y := screenCorners[k].y;

  sdlVertices[k].color.r := fcolor;
  sdlVertices[k].color.g := fcolor;
  sdlVertices[k].color.b := fcolor;
  sdlVertices[k].color.a := 1.0;
  sdlVertices[k].tex_coord.x := 0;
  sdlVertices[k].tex_coord.y := 0;
end;
```

```
  SDL_RenderGeometry(
    gm^.renderer,
    nil, { ми не використовуємо текстур тут, тому nil }
    @sdlVertices[0], { вказівник на ПЕРШУ вершину }
    4, { кількість вершин }
    @indices[0], { вказівник на ПЕРШИЙ індекс }
    6 { кількість індексів }
  );
end;
```

Реалізація функції в цілому нескладна, але вимагає від вас повного розуміння того, що ми обговорювали вище: тут відбувається визначення координат кожної із вершин у світовій системі координат, перетворення цих вершин у екранну систему координат, і малювання із допомогою функції `SDL_RenderGeometry`. Цій функції потрібен список самих вершин, (для цього ми використовуємо масив `sdlVertices`) а також інформацію про те, як ці вершини формують трикутники (для цього ми використовуємо масив `indices`, кожна трійка елементів якого описує один трикутник).

Єдине, що може збити тут із толку, — це те, як ми передаємо масиви у якості аргументів. Справа в тому, що `SDL_RenderGeometry` не приймає масиви в тому виді, як вони представляються в Pascal. Замість цього, функція приймає **вказівник на перший елемент масиву і кількість його елементів**. Це специфіка роботи із масивами у мові програмування C, на якій SDL і написана.

### 4.6.3 Рендеримо персонажа на карті

Персонажа малювати набагато легше: нам достатньо намалювати круг, що позначить позицію персонажа на карті і лінію, яка буде відображати напрямок його погляду.

Проблема тільки в тому, що у SDL немає вбудованої функції для малювання круга. Тому можете скористатись спеціальною функцією у модулі `sdl3`, яку розробив я. Вона вирішить цю проблему. Функція називається

`SDLHelper_FillDisk`.

Розглянемо тепер, як ми реалізуємо малювання персонажа на карті:

Pascal player.pas

```pascal
procedure PlayerMapRender(var player: TPlayer);
var
  gm: PGameManager;
  direction: TVector;
  screenPlayerPosition: TVector;
  lookAtWorld: TVector;
  lookAtScreen: TVector;
begin
  gm := PGameManager(player.gmPtr);
  screenPlayerPosition := MapWorldSpaceToScreenSpace(
    player.position,
    gm
  );

  direction := VectorPolar(10, player.angle);
  lookAtWorld := VectorAdd(player.position, direction);
  lookAtScreen := MapWorldSpaceToScreenSpace(lookAtWorld, gm);

  { малюємо маленький круг, що позначить гравця }
  SDLHelper_FillDisk(
    gm^.renderer,
    screenPlayerPosition.x,
    screenPlayerPosition.y,
    5,
    255, 0, 0
  );

  { малюємо лінію напрямку погляду гравця }
  SDL_SetRenderDrawColor(gm^.renderer, 255, 0, 0, 255);
```

```
  SDL_RenderLine(
    gm^.renderer,
    screenPlayerPosition.x,
    screenPlayerPosition.y,
    lookAtScreen.x,
    lookAtScreen.y
  );
end;
```

Тут все значно простіше. Малюємо точку, де знаходиться персонаж, із допомогою функції `SDLHelper_FillDisk`, а потім малюємо напрям його погляду із допомогою функції `SDL_RenderLine`. Не забуваємо перед цим перевести координати в екранну систему координат.

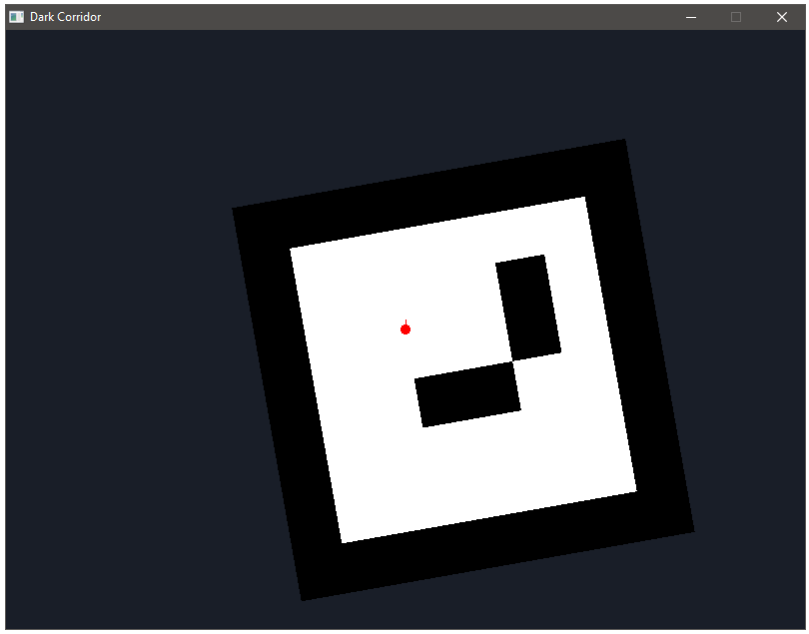



Рис. 4.3: Намальована карта і позиція персонажа.

Якщо ви запустите програму, то побачите щось схоже на Рис. 4.3. Якщо це так, то вам вдалось намалювати світ у режимі карти!

### 4.6.4 Переміщуємо персонажа

До цього моменту наш персонаж просто стояв стовпом на карті. Пора це виправляти!

Ми хочемо, щоб натискаючи стрілочки, ми переміщували нашого персонажа. Якщо ми натиснемо ↑, то персонаж піде вперед, якщо ↓, то піде назад. При натисненні ← або → персонаж повернеться у відповідну сторону.

Давайте згадаємо: інформація про дії користувача ми хочемо записувати у запис із подіями. Давайте відповідно оновимо його:

Pascal gameManager.pas

```
TGameEvents = record
  exitApp: Boolean;

  goForward: Boolean; { іти вперед }
  goBack: Boolean; { іти назад }
  turnLeft: Boolean; { повернути вліво }
  turnRight: Boolean; { повернути вправо }
end;
```

Тепер у процедурі, яка відповідає за обробку подій від SDL, ми повинні додати перевірку натиснення клавіш зі стрілками. Пропоную ще додати альтернативні кнопки управління — W, A, S та D:

Pascal gameManager.pas

```
procedure GameManagerPollEvents(var gm: TGameManager);
var
  event: SDL_Event;
  keyDown: Boolean;
begin
  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    case event.event_type of
```

```
      SDL_EVENT_QUIT:
        gm.events.exitApp := True;
      SDL_EVENT_KEY_DOWN, SDL_EVENT_KEY_UP:
        begin
          keyDown := event.event_type = SDL_EVENT_KEY_DOWN;
          case event.key.key of
            SDLK_UP, SDLK_w:
              gm.events.goForward := keyDown;
            SDLK_DOWN, SDLK_s:
              gm.events.goBack := keyDown;
            SDLK_LEFT, SDLK_a:
              gm.events.turnLeft := keyDown;
            SDLK_RIGHT, SDLK_d:
              gm.events.turnRight := keyDown;
          end;
        end;
    end;
  end;
end;
```

Зовнішній цикл лишається тим самим, що й раніше: `SDL_PollEvent` по черзі забирає всі події з черги, поки вона не спорожніє. Далі ми розбираємо `event.event_type` за допомогою `case`: якщо прийшла подія `SDL_EVENT_QUIT`, ставимо `exitApp`.

Гілка `SDL_EVENT_KEY_DOWN, SDL_EVENT_KEY_UP` обробляє і натискання, і відпускання клавіші. Змінна `keyDown` стає `True`, коли тип події саме `SDL_EVENT_KEY_DOWN` (тобто коли клавіша натиснута), і `False`, коли `SDL_EVENT_KEY_UP` (тобто коли клавіша відпущена). Тоді рядки на кшталт `gm.events.goForward := keyDown` означають просте правило: поки клавіша затиснута — прапорець увімкнений, як тільки відпустили — вимкнений. Логіка оновлення світу в наступному кадрі лише читає ці прапорці і не повинна сама вгадувати, чи ще натиснута клавіша.

Внутрішній `case event.key.key` зіставляє конкретну клавішу з полем

запису `TGameEvents`. Кілька міток через кому (`SDLK_UP, SDLK_w`) — це один і той самий код для кількох клавіш: і ↑, і W ведуть персонажа вперед, S і ↓ — назад, A/D і ←/→ відповідають поворотам.

Тепер подивимось на код оновлення персонажа:

Pascal player.pas

```pascal
function BoolToF32(b: Boolean): f32;
begin
  if b then
    BoolToF32 := 1.0
  else
    BoolToF32 := 0.0;
end;

procedure PlayerUpdate(
  var player: TPlayer;
  const deltaTimeSeconds: f32);
var
  gm: PGameManager;
  fwd: f32;
  right: f32;

  velocity: TVector;
  angularVelocity: f32;

  frameMovement: TVector;
  frameAngularMovement: f32;
begin
  gm := PGameManager(player.gmPtr);

  { 1, якщо персонаж іде вперед. -1, якщо назад }
  fwd :=
    BoolToF32(gm^.events.goForward) -
```

```
    BoolToF32(gm^.events.goBack);

  { 1, якщо персонаж повертається вправо. -1, якщо вліво }
  right :=
    BoolToF32(gm^.events.turnRight) -
    BoolToF32(gm^.events.turnLeft);

  velocity := VectorPolar(fwd * player.speed, player.angle);
  angularVelocity := right * player.angularSpeed;

  frameMovement := VectorMulScalar(velocity, deltaTimeSeconds);
  frameAngularMovement := angularVelocity * deltaTimeSeconds;

  player.position := VectorAdd(player.position, frameMovement);
  player.angle := player.angle + frameAngularMovement;
end;
```

Тут все досить просто. Спочатку рахуємо вектор поточної швидкості (`velocity`) і поточну кутову швидкість обертання (`angularVelocity`). Зверніть увагу, що ці значення уже залежать від того, які події керування персонажем у нас увімкнені.

Потім визначаємо те переміщення, яке виконає персонаж протягом кадру від цих дій: `frameMovement` для позиції, `frameAngularMovement` для обертання.

І, нарешті, додаємо ці значення до поточних значень положення і кута погляду.

Після цих дій ви нарешті зможете переміщувати персонажа по карті, але чомусь у нашого персонажа зараз забагато свободи ...

### 4.6.5 Вимикаємо «noclip»

Персонаж зараз може проходити крізь стіни. Ми повинні забрати у нього цю можливість. На щастя, робиться це досить легко. Згадайте, як у

попередній частині ми рахували точку, де буде персонаж після переміщення:

```
player.position := VectorAdd(player.position, frameMovement);
```

Замість цього нам треба спочатку передбачити точку, де буде персонаж після переміщення (тобто виконати `VectorAdd`, але поки що не присвоювати значення до `player.position`). Далі ми маємо перевірити, чи «майбутня» позиція **не** потрапляє у клітинку стіни. Ще пам'ятаєте, як ви виконували Задачу 18? Переміщувати персонажа будемо лише тоді, коли «майбутня» точка **не є точкою стіни**.

Тобто покращена реалізація переміщення персонажа виглядає так:

Pascal player.pas

```
futurePosition := VectorAdd(player.position, frameMovement);
player.angle := player.angle + frameAngularMovement;

if not IsWall(gm^.world.walls, futurePosition) then
  player.position := futurePosition;
```

Тепер наш персонаж не буде «привидом»: він не зможе проходити крізь стіни.

Таким чином, у нас наразі реалізована уся **логіка** нашої гри. Але перед тим, як переходити до 3D-рендерингу, там потрібно вирішити ще проблему променів.

## 4.7 Промені

Промені — ключовий компонент рейкаст-рушія. Саме вони допомагають визначити відстань від позиції персонажа до стін. Логіка того, як ведуть себе промені, працює у двовимірній площині, тому я вирішив, що саме поведінкою променів ми завершимо розділ про ігрову логіку перед тим, як остаточно перейти до 3D.

Отож, ми до нашого персонажа додамо промені. Оновимо модуль `player`, додавши масив променів до запису персонажа та кут огляду (**FOV**, field of view) у параметрах його ініціалізації.

Pascal player.pas

```
type
  TPlayerInitializeParameters = record
        position: TVector;
        angle: f32;
        speed: f32;
        angularSpeed: f32;
        fov: f32;
  end;

    TRay = record
    { кут відносно кута погляду гравця }
    relativeAngle: f32;

    { точка перетину із найближчою стіною }
    wallIntersection: TVector;

    { відстань до перетину із найближчою стіною }
    wallDistance: f32;

    { true, якщо перетин із стіною вертикальний,
      false, якщо горизонтальний }
    isWallIntersectionVertical: Boolean;
  end;

  TPlayer = record
        position: TVector;
        angle: f32;
        speed: f32;
```

```
        angularSpeed: f32;
    gmPtr: Pointer;
        rays: array of TRay;
  end;
```

При ініціалізації персонажа ініціалізуємо його промені: кут відносно напряму персонажа найлівішого променя має дорівнювати $\frac{-\varphi}{2}$, а найправішого — $\frac{\varphi}{2}$, де $\varphi$ — FOV. Всі інші промені рівномірно розподіляють кут:

Pascal player.pas

```
relativeAngleStep :=
  (params.fov / (raysCount - 1));

{ relativeAngle у радіанах:
  від -fov/2 (ліва колонка) до +fov/2 (права) }
if raysCount = 1 then
  player.rays[0].relativeAngle := 0
else
  begin
    player.rays[0].relativeAngle := -params.fov * 0.5;
    for i := 1 to raysCount - 1 do
      player.rays[i].relativeAngle :=
        player.rays[i - 1].relativeAngle +
        relativeAngleStep
  end;
```

**Увага**

Цей код ініціалізації променів не ідеальний. Він породить одну проблему, яку ми ще обговоримо у розділі, присвяченому 3D. Поки що рівномірного розподілу відносного кута променів нам достатньо.

Тепер нам потрібно визначити найближчу точку перетину променя із стіною. Давайте згадаємо частину про Життя окремого променя: ми по-

винні знайти найближчу вертикальну і найближчу горизонтальну точку перетину, а потім порівняти, яка із них має меншу відстань до персонажа.

І для «вертикального», і для «горизонтального» випадків алгоритм однаковий: ми шукаємо першу точку-кандидата і вектор різниці між сусідніми кандидатами. Збільшуючи кандидата на вектор різниці ми шукаємо першу точку, що виявиться стіною.

Давайте розглянемо, як це працює для вертикальної точки перетину:

Pascal player.pas

```pascal
function VectorEps(const direction: TVector): TVector;
begin
  VectorEps := VectorMulScalar(
    direction,
    0.01
  );
end;

function IsFacingRight(const vector: TVector): f32;
begin
  if vector.x > 0 then
    IsFacingRight := 1.0
  else
    IsFacingRight := 0.0;
end;

function FindNearestVerticalWall(
  var ray: TRay;
  const gm: PGameManager): TVector;
var
  firstCandidate: TVector;
  deltaCandidate: TVector;
  candidate: TVector;
  start: TVector;
```

```
  tileSize: f32;
  direction: TVector;
  eps: TVector;
begin
  start := gm^.world.player.position;
  tileSize := gm^.world.walls.tileSize;
  direction := VectorPolar(
    1,
    gm^.world.player.angle + ray.relativeAngle);
  eps := VectorEps(direction);

  firstCandidate.x := tileSize * (
    Floor(start.x / tileSize) +
    IsFacingRight(direction)
  );

  firstCandidate.y := start.y +
    (firstCandidate.x - start.x) *
    (direction.y / direction.x);

  deltaCandidate := VectorMulScalar(
    direction,
    tileSize / Abs(direction.x)
  );

  candidate := firstCandidate;
  while not IsWall(gm^.world.walls, VectorAdd(candidate, eps)) do
    candidate := VectorAdd(candidate, deltaCandidate);

  FindNearestVerticalWall := candidate;
end;
```

У цьому коді все максимально просто і зрозуміло: ми додаємо до змінної `candidate` змінну `deltaCandidate` до тих пір, поки `candidate` не стане

точкою стіни.

Тут є лише один не дуже очевидний момент. У функцію `IsWall` ми як параметр передаємо не саму точку `candidate`, а її суму зі змінною `eps`. Ідея така: `eps` — це вектор дуже маленької довжини, співнапрямлений із променем. Для вертикального променя кандидати лежать на вертикальних лініях сітки, тобто на межах сусідніх тайлів, і через похибки чисел із рухомою комою точка перевірки без додаткового зсуву могла б опинитися «між» двома клітинками. Невеликий крок на `eps` уздовж напряму променя зсуває перевірку в ту половину межі, куди промінь рухається, і тоді `IsWall` дає однозначну відповідь.

> **Задача 21**
>
> Реалізуйте самостійно функцію `FindNearestHorizontalWall`, яка шукає найближчу горизонтальну точку перетину.

Тепер нам потрібно написати функцію, яка оновить найближчу точку перетину променя і порахує відстань до стіни. Тут все досить просто, особливо якщо ви знайшли відповідь на Запитання 4:

Pascal — player.pas

```pascal
procedure UpdateNearestWallDistance(
  var ray: TRay;
  const gm: PGameManager
);
var
  horizontal: TVector;
  vertical: TVector;
  start: TVector;
  horizontalSqMag: f32;
  verticalSqMag: f32;
begin
  start := gm^.world.player.position;
  horizontal := FindNearestHorizontalWall(ray, gm);
```

```
  vertical := FindNearestVerticalWall(ray, gm);

  { Відстань від гравця до перетину, не від початку світу (0,0). }
  horizontalSqMag := VectorMagnitudeSquared(
    VectorSub(horizontal, start)
  );

  verticalSqMag := VectorMagnitudeSquared(
    VectorSub(vertical, start)
  );

  if horizontalSqMag < verticalSqMag then begin
    ray.wallIntersection := horizontal;
    ray.wallDistance := sqrt(horizontalSqMag);
    ray.isWallIntersectionVertical := False;
  end
  else begin
    ray.wallIntersection := vertical;
    ray.wallDistance := sqrt(verticalSqMag);
    ray.isWallIntersectionVertical := True;
  end;
end;

procedure RayUpdate(var ray: TRay; const gm: PGameManager);
begin
  UpdateNearestWallDistance(ray, gm);
end;

procedure PlayerUpdate(
  var player: TPlayer;
  const deltaTimeSeconds: f32);
var
    gm: PGameManager;
```

```
    ...
begin
    ...

    for i := Low(player.rays) to High(player.rays) do
        RayUpdate(player.rays[i], gm);
end;
```

Як бачимо, ми просто порівнюємо відстань від персонажа до горизонтального і вертикального перетинів, і шукаємо меншу. Це ми робимо для кожного променя.

І для оновлених значень променів нам залишається намалювати промені на нашій карті. Ми ж хочемо наочно бачити результат нашої роботи, чи не так? Отож, код для малювання невеликий. Нам всього лише достатньо намалювати лінію від персонажа до стіни, і зробити це для кожного променя:

Pascal player.pas

```
procedure RayMapRender(var ray: TRay; const gm: PGameManager);
var
  startWorld: TVector;
  startScreen: TVector;
  intersectionScreen: TVector;
begin
  startWorld := gm^.world.player.position;
  startScreen := MapWorldSpaceToScreenSpace(startWorld, gm);
  intersectionScreen := MapWorldSpaceToScreenSpace(
    ray.wallIntersection,
    gm
  );

  SDL_SetRenderDrawColor(gm^.renderer, 230, 230, 230, 255);
  SDL_RenderLine(
```

```
    gm^.renderer,
    startScreen.x,
    startScreen.y,
    intersectionScreen.x,
    intersectionScreen.y
  );
end;

procedure PlayerMapRender(var player: TPlayer);
var
  ...
begin
  gm := PGameManager(player.gmPtr);

  ...

  { малюємо промені }
  for i := Low(player.rays) to High(player.rays) do
    RayMapRender(player.rays[i], gm);

  { малюємо гравця далі }
  ...
end;
```

Тепер, після компіляції, ви побачите, як промені від персонажа перетинають стіни, як на Рис. 4.4.

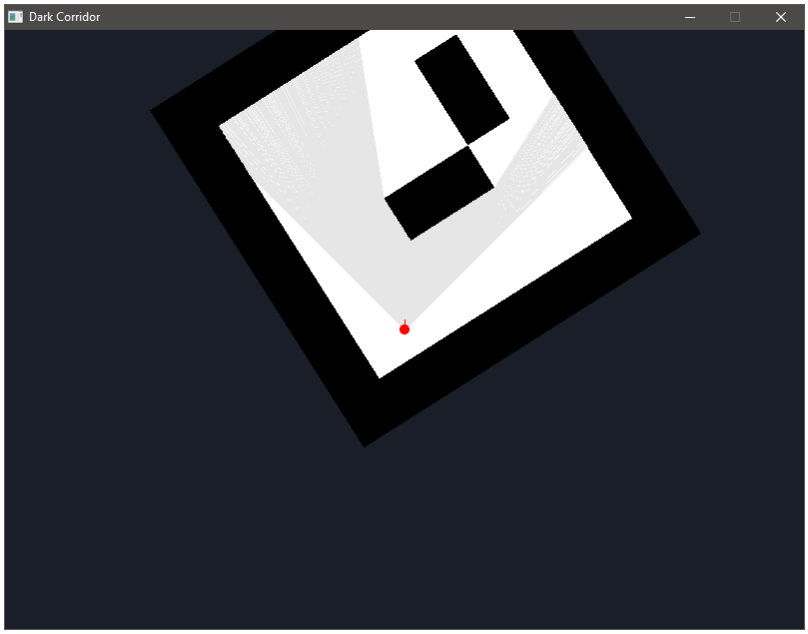



Рис. 4.4: Промені від персонажа.

## 4.8 Висновки

У цьому розділі ви реалізували все, що стосується двовимірної логіки гри: ігрову карту, персонажа, його переміщення, колізії зі стінами і, звичайно, промені. Не варто забувати й про інші важливі речі: ви навчилися налаштовувати проект і грамотно розбивати його на модулі. Ця навичка стане в нагоді будь-якому програмісту.

У наступному розділі ми вже перейдемо до 3D-частини нашого рушія.

# Розділ 5

# 3D

Тепер, маючи повністю реалізовану ігрову логіку, ми готові перейти до головної частини — 3D-рендерингу.

Тут буде найцікавіша, як на мене, частина розробки, оскільки ми будемо на очах бачити те, чого ми так очікували, — тривимірність.

## 5.1 Два режими відображення

Хоч ми і будемо реалізовувати 3D-рендеринг, все ж не хочеться позбавлятись від тої красивої карти, яку ми намалювали у попередньому розділі. Тому ми зробимо два режими відображення: основним режимом буде 3D, але, поки натиснута клавіша ⇆ Tab, світ буде відображуватись у режимі карти. Це схоже на те, як працює режим карти у класичному **Doom**.

Додамо нове поле до запису подій і нову логіку у процедурі `GameManagerPollEvents`:

Pascal — gameManager.pas

```
TGameEvents = record
  ...

  mapMode: Boolean; { режим карти }
end;

...
```

```
procedure GameManagerPollEvents(var gm: TGameManager);
var
  event: SDL_Event;
  keyDown: Boolean;
begin
  while SDL_PollEvent(@event) = SDL_TRUE do
  begin
    case event.event_type of
      SDL_EVENT_QUIT:
        gm.events.exitApp := True;
      SDL_EVENT_KEY_DOWN, SDL_EVENT_KEY_UP:
        begin
          keyDown := event.event_type = SDL_EVENT_KEY_DOWN;
          case event.key.key of
            ...
            SDLK_TAB:
              gm.events.mapMode := keyDown;
          end;
        end;
    end;
  end;
end;
```

Тепер трошки оновимо функцію `WorldRender`, щоб у ній була перевірка на режим карти:

Pascal world.pas

```
procedure World3DRender(var world: TWorld; const gm: PGameManager);
begin
  SDL_SetRenderDrawColor(gm^.renderer, 0, 0, 0, 255);
  SDL_RenderClear(gm^.renderer);

  { TODO: Тут будемо малювати 3D-зображення }
```

```
end;

procedure WorldMapRender(var world: TWorld; const gm: PGameManager);
begin
  SDL_SetRenderDrawColor(gm^.renderer, 25, 30, 40, 255);
  SDL_RenderClear(gm^.renderer);

  WallsMapRender(world.walls);
  PlayerMapRender(world.player);
end;

procedure WorldRender(var world: TWorld);
var
  gm: PGameManager;
begin
  gm := PGameManager(world.gmPtr);
  if gm^.events.mapMode then
    WorldMapRender(world, gm)
  else
    World3DRender(world, gm);
end;
```

Тепер за замовчуванням програма відображає темний екран, але ми можемо переключитися на режим карти, натиснувши ⇆Tab. Згодом `World3DRender` буде відображати 3D-зображення світу, а не монотонний чорний екран.

## 5.2 Новий модуль — 3D-рендерер

Хоча ми і можемо реалізувати 3D-рендеринг у модулі `world`, все ж краще буде створити новий модуль, що повністю відповідатиме за 3D. Справа у тому, що код, що буде відповідальним за рендеринг 3D-світу буде досить обширним, тому ми і винесемо його окремо.

Створимо новий модуль `world3drenderer`, і перенесемо туди нашу нову функцію `World3DRender`.

Pascal world3drenderer.pas

```pascal
unit world3drenderer;

interface

uses world, gameManager;

procedure World3DRender(var world: TWorld; const gm: PGameManager);

implementation

{ TODO: Реалізація інтерфейсу }

end.
```

### 5.2.1 Площина проекції

Для того, щоб зрозуміти, як працює 3D-перспектива, погляньте уважно на Рис. 5.1.

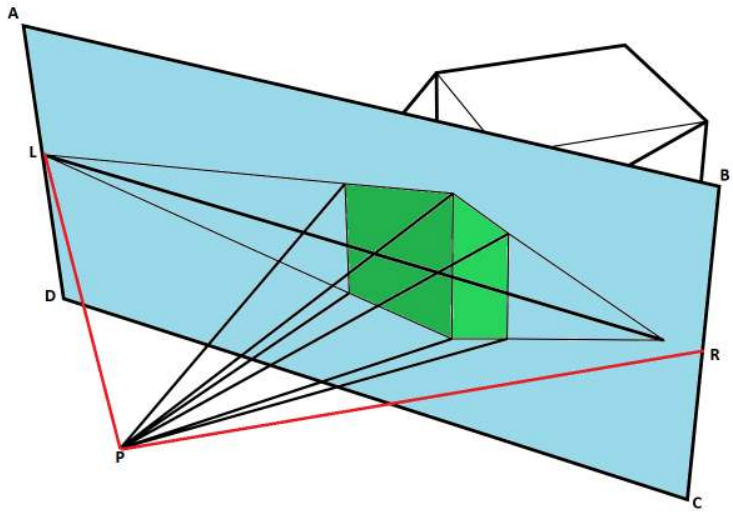



Рис. 5.1: 3D-проекція із врахуванням перспективи.

Персонаж перебуває в точці $P$ і дивиться на куб. Куб, який ви бачите, має якимось чином зобразитись на екрані. Але яким чином він переноситься на екран? Уявіть собі великий ватман формату A1, який знаходиться перед вашими очима (на рисунку це прямокутник $ABCD$).

Коли ви дивитесь на куб, уявіть відрізки між кожною точкою куба і вашою позицією. Точки, де ці відрізки перетинають ватман, і дають перспективну проекцію куба на екран; тут ватман уявляємо екраном.

Але де має бути розташований цей ватман? Подивіться на точки $L$ і $R$ — це крайні ліворуч і праворуч точки на ватмані. Ватман має бути розташованим так, щоб точки $L$ і $R$ були відповідно найлівішою і найправішою точками, які ви бачите в даний момент.

У такому випадку $\varphi = \angle LPR$ буде кутом **поля зору** (англ. *field of view*, скорочено FOV).

Площина, у якій лежить цей «ватман», називається **площиною проекції** (англ. *projection plane*).

Для подальших розрахунків нам потрібно буде визначити відстань від персонажа до площини проекції $d$. На Рис. 5.2 зображено вигляд згори:

трикутник $LPR$ і відстань $d$ від ока до площини проекції вздовж напряму погляду (відрізок $PM$ перпендикулярний до краю екрану $LR$). Розглянемо цей трикутник детальніше.

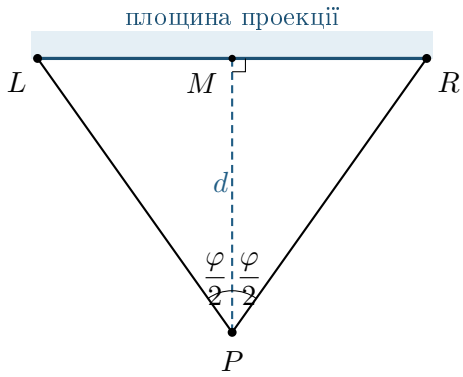



Рис. 5.2: Трикутник $LPR$ (вид згори).

Якщо детальніше розглянути наш рисунок, ми побачимо, що нам достатньо простої тригонометрії, щоб виразити $d$, через $LM$:

$$|LM| = d \tan \frac{\varphi}{2}$$

$$d = \frac{|LM|}{\tan(\varphi/2)}$$

А, оскільки точка $M$ — середина $LR$, то:

$$|LM| = \frac{1}{2}|LR|$$

Тобто:

$$d = \frac{|LR|}{2\tan(\varphi/2)}$$

Оскільки «ватман» — це екран, довжина відрізка $|LR|$ дорівнює ширині екрану в пікселях. Позначимо цю величину через $u_x$. Тоді $|LR| = u_x$, і ми легко виражаємо $d$:

$$d = \frac{u_x}{2\tan(\varphi/2)}$$

Але навіщо нам потрібна відстань до площини проекції? Для того, щоб це зрозуміти, давайте знову подивимось на нашу площину проекції, тільки цього разу не зверху, як на Рис. 5.2, а збоку, як на Рис. 5.3.

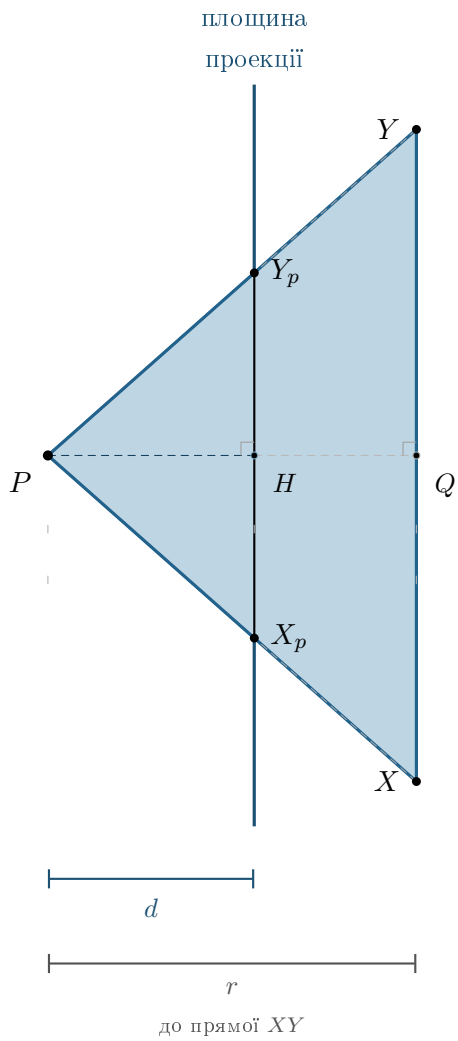



Рис. 5.3: Відрізок $XY$ паралельний площині проекції

Нехай відрізок $XY$ у просторі **паралельний** площині проекції. Промені $PX$ і $PY$ перетинають площину в точках $X_p$ і $Y_p$ — це проекція кінців відрізка. Тоді $X_pY_p$ — проектований відрізок $XY$. На Рис. 5.3 (переріз пло-

щиною, що містить $P$, $X$ і $Y$) видно, що $XY \parallel X_pY_p$, а трикутники $PXY$ і $PX_pY_p$ мають спільний кут у $P$ і відповідні кути при $X$ і $X_p$, при $Y$ і $Y_p$ як кути при січній до паралельних прямих; отже

$$\triangle PXY \sim \triangle PX_pY_p.$$

**Примітка**

$XY \parallel X_pY_p$ тільки тому, що у нашому рушії усі клітинки — це куби, вертикальні ребра яких перпендикулярні підлозі. У рушіях із більш складною геометрією, ця умова не обов'язково виконуватиметься.

Трикутники $PXY$ і $PX_pY_p$ — подібні. Отже, їх розміри пропорційні, що означає, що

$$\frac{|PH|}{|PQ|} = \frac{|X_pY_p|}{|XY|}$$

Давайте подумаємо, чому дорівнюють відрізки із формули вище: $|PH|$ — це довжина $d$, яку ми щойно обговорили. $|XY|$ нам відома як довжина сторони «клітинки» $w$ (пам'ятаєте, на карті усі клітинки репрезентують куби; для вертикальної грані стіни роль $w$ відіграє саме її висота). Тобто переписати формулу можемо так:

$$|X_pY_p| = \frac{dw}{|PQ|}$$

Але що саме в цій формулі позначає $|PQ|$? Давайте поки що спростимо собі життя. Нехай $Q_{\text{ray}}$ — точка перетину променя зі стіною; поки що вважатимемо, що $|PQ|$ у формулі можна замінити на відстань $|PQ_{\text{ray}}|$ вздовж променя (у коді це `ray.wallDistance`). Це наближення зручне, але не завжди точне — це проявляється в так званому ефекті «риб'ячого ока», який ми розглянемо трохи згодом.

А поки що можемо знову перейти до коду.

### 5.2.2 Реалізовуємо 3D-проекцію

Спочатку нам потрібно в `TGameManager` додати поля `fov` і `distanceToProjectionPlane`, які при ініціалізації розрахуємо (ці значення — однакові протягом всієї гри):

Pascal gameManager.pas

```
gm.fov := params.world.player.fov;
gm.distanceToProjectionPlane :=
  gm.windowSize.x / (2 * tan(gm.fov * 0.5));
```

Важливий практичний момент: `distanceToProjectionPlane` має бути обчислений **до** того, як ви ініціалізуєте кути променів у гравця, інакше в обчислення підставиться ще не готове значення.

Тепер реалізуємо 3D-рендеринг. Коротко про наступний блок коду: для кожної вертикальної колонки екрана ми беремо відстань до перетину зі стіною, переводимо її у висоту «смужки» стіни за формулою з $d$ і $w$, а все, що вище і нижче стіни, фарбуємо у два різні відтінки сірого («стеля» і «підлога»).

Pascal world3drenderer.pas

```
{ функція "зрізає" перший аргумент до значення між мінімумом і
  максимумом. Якщо число менше за мінімум, то функція повертає
  мінімум. Якщо число більше за максимум, то функція повертає
  максимум. В іншому випадку функція повертає саме число }
function Clamp(const num: f32; const min: f32; const max: f32): f32;
begin
  Clamp := num;
  if Clamp > max then Clamp := max;
  if Clamp < min then Clamp := min;
end;

procedure Ray3DRender(
  var ray: TRay;
```

```
  const rayIndex: isize;
  const gm: PGameManager);
var
  windowSize: TVector;
  wallScreenHeight: f32;
  wallTopY: f32;
  wallBottomY: f32;
  intensity: f32;
begin
  windowSize := gm^.windowSize;

  wallScreenHeight := gm^.distanceToProjectionPlane *
    gm^.world.walls.tileSize / ray.wallDistance;
  wallTopY := windowSize.y / 2 - wallScreenHeight / 2;
  wallBottomY := windowSize.y / 2 + wallScreenHeight / 2;
  wallTopY := Clamp(wallTopY, 0, windowSize.y);
  wallBottomY := Clamp(wallBottomY, 0, windowSize.y);
  intensity := 1.0;
  if ray.isWallIntersectionVertical then
    intensity := intensity * 0.8;

  { все, що іде від верху до початку стіни --- це стеля }
  SDL_SetRenderDrawColor(gm^.renderer, 51, 51, 51, 255);
  SDL_RenderLine(
    gm^.renderer,
    rayIndex,
    0,
    rayIndex,
    wallTopY
  );

  { тут ми малюємо стіну }
  SDL_SetRenderDrawColor(
```

```
    gm^.renderer,
    Round(34 * intensity),
    Round(56 * intensity),
    Round(114 * intensity),
    255);

  SDL_RenderLine(
    gm^.renderer,
    rayIndex,
    wallTopY,
    rayIndex,
    wallBottomY
  );

  { все, що іде від кінця стіни до низу --- це підлога }
  SDL_SetRenderDrawColor(gm^.renderer, 119, 119, 119, 255);
  SDL_RenderLine(
    gm^.renderer,
    rayIndex,
    wallBottomY,
    rayIndex,
    windowSize.y
  );
end;

procedure World3DRender(var world: TWorld; const gm: PGameManager);
var
  i: isize;
begin
  for i := Low(world.player.rays) to High(world.player.rays) do
    Ray3DRender(world.player.rays[i], i, gm);
end;
```

Функція `World3DRender` «проходить» по всіх променях і для кожного

із них викликає функцію рендера променя. Задача кожного променя тут — намалювати стовпчик шириною 1 піксель і висотою, що дорівнює висоті екрану, за що і відповідає функція `Ray3DRender`.

У цій функції і відбуваються ті підрахунки, що ми проводили вище. У змінну `wallScreenHeight` ми і записуємо висоту проекції стіни на площину екрану. Оскільки ця проекція буде знаходитись в центрі вертикальної лінії, ми розраховуємо екранні координати $y$ найвищої і найнижчої точки стіни, записуючи їх у `wallTopY` і `wallBottomY`. Оскільки проекція стіни може виявитись вищою, ніж висота екрану, ми її точки `wallTopY` і `wallBottomY` «зрізаємо» до меж екрану за допомогою функції `Clamp`. Хоча і малюватимемо стіну синім кольором, ми цей колір затемнимо (зменшивши змінну `intensity`) у випадку, якщо промінь перетинає вертикальну сторону будь-якої стіни. Це дасть нам ілюзію освітлення.

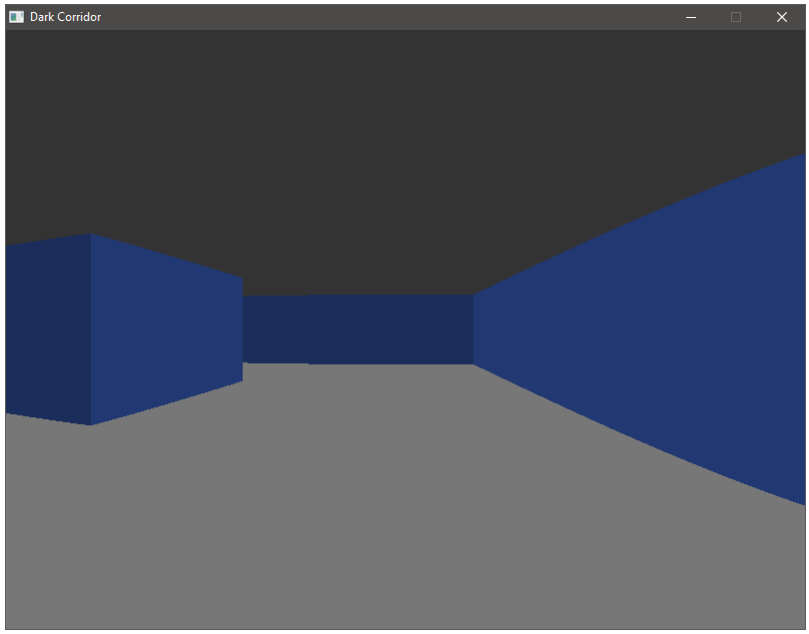



Рис. 5.4: Перший 3D-вигляд світу після проекції стовпчиків.

Після запуску коду ми отримаємо 3D-проекцію світу (Рис. 5.4)! Можете порівняти його із Рис. 1.5! Тепер ми можемо не просто переміщуватись по абстрактній і плоскій міні-карті. У нас уже є можливість повного занурення у 3D-світ: приближуючись до стіни чи віддаляючись від неї, ви бачите

зміну її розмірів за рахунок перспективи. І, хоча основна ідея рушія уже реалізована, нам є ще куди рости. І, на жаль, є що виправляти ...

### 5.2.3 «Риб'яче око»

У нашій реалізації проекції є одна проблема. Погляньте на довільну стіну у грі, і проведіть відрізки між двома кінцями стіни зверху і знизу, як на Рис. 5.5. Ви побачите, що стіни, взагалі-то, криві!

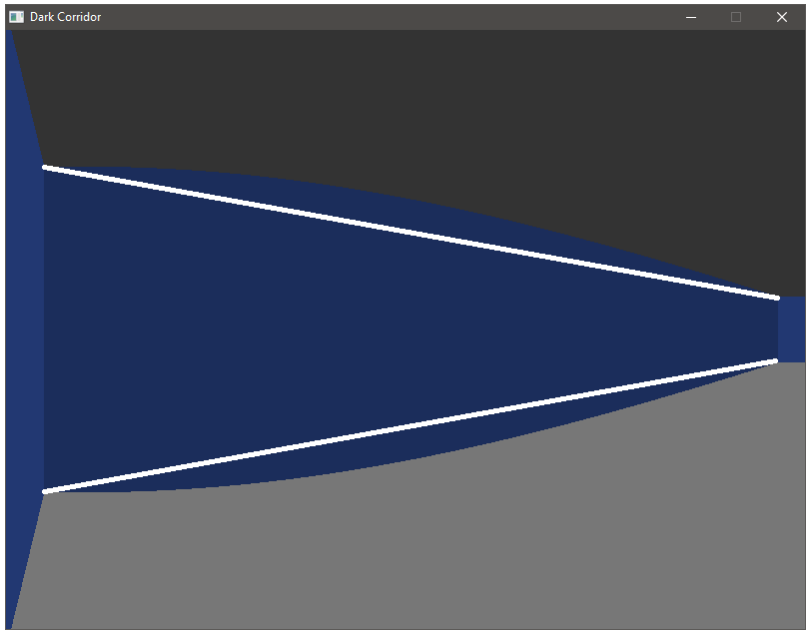



Рис. 5.5: Ефект «риб'ячого ока»

Для того, щоб зрозуміти, як так сталось, згадаймо, що ми припустили щодо значення відстані $PQ$: ми вирішили, що $PQ$ — це відстань від персонажа до точки перетину променя зі стіною, але це не зовсім правильно. Через це припущення ми і наємо такий негарний ефект. Як же тоді рахувати правильно?

На Рис. 5.6 — спрощений вид згори: стіна збігається з вертикальною прямою, з точки $P$ випущено промінь колонки до точки $Q_{\text{ray}}$ на стіні; тоді `ray.wallDistance` $= |PQ_{\text{ray}}|$.

Для того, щоб «вирівняти» стіни на 3D-проекції, нам потрібна **не** від-

стань від гравця до точки перетину променя і стіни, а її **проекція на напрям погляду**. Звучить трохи дивно, але це так.

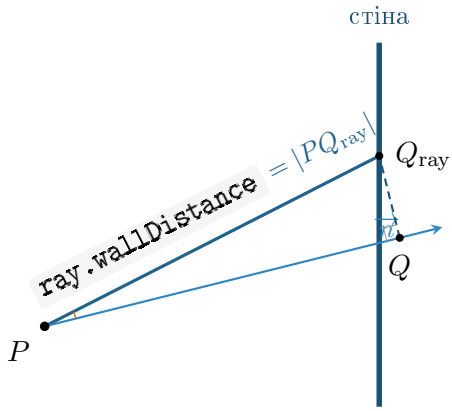



Рис. 5.6: Виправлення «риб'ячого ока»: у знаменник подібності треба підставити глибину вздовж погляду $|PQ|$

Довжина цієї проекції рахується просто:

$$|PQ| = |PQ_{\text{ray}}| \cos \delta,$$

де $\delta$ — різниця між кутом повороту персонажа і кутом нахилу променя, тобто `ray.relativeAngle` у коді.

Давайте зробимо просту зміну у нашому коді, щоб виправити проблему:

**Pascal** `world3drenderer.pas`

```
wallDistanceAcrossViewVector :=
  ray.wallDistance * cos(ray.relativeAngle);
wallScreenHeight :=
  gm^.distanceToProjectionPlane * gm^.world.walls.tileSize /
  wallDistanceAcrossViewVector;
```

Після такої простої зміни ви помітите, що картинка виглядає **значно** краще (Рис. 5.7)! Стіна більше не здається «надутою». Але все ж на краях картинки ми все ще спостерігаємо спотворення.

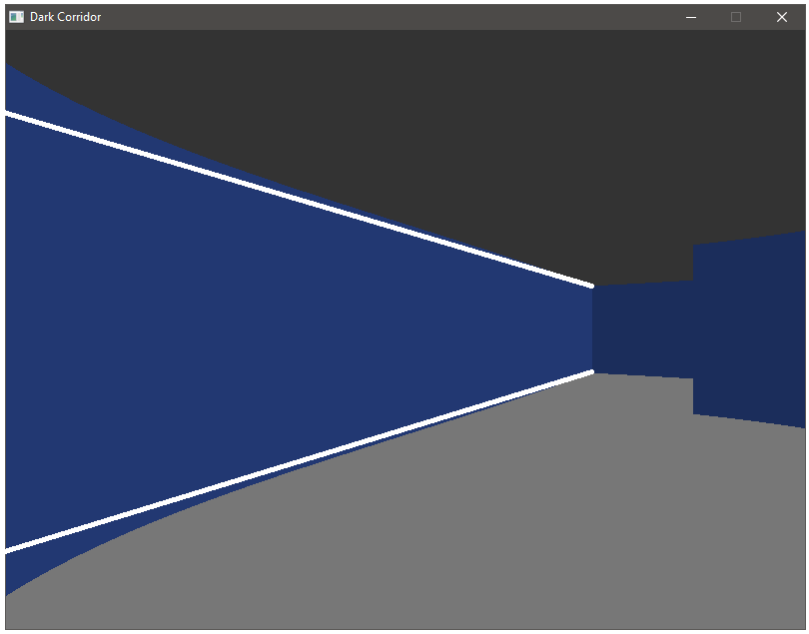



Рис. 5.7: Майже виправлений ефект «риб'ячого ока»

Причина цього другого спотворення полягає ще в одній проблемі нашого коду. Проблема лежить у променях, а точніше — у їх куті відносно напрямку погляду. Ще пам'ятаєте, як ми задавали значення поля `relativeAngle` для променів? Давайте згадаємо:

Pascal player.pas

```
relativeAngleStep :=
  (params.fov / (raysCount - 1));

...

begin
  player.rays[0].relativeAngle := -params.fov * 0.5;
  for i := 1 to raysCount - 1 do
    player.rays[i].relativeAngle :=
      player.rays[i - 1].relativeAngle +
      relativeAngleStep
```

```
end;
```

Інтуїтивно звучить правильно: ми беремо набір кутів у межах $[-\frac{\varphi}{2}; \frac{\varphi}{2}]$ і рівномірно розподіляємо їх по кожному променю.

Але цей підхід неправильний: нам не потрібно, щоб кути між сусідніми променями були однаковими, нам потрібно, щоб промені відповідали **рівномірному розташуванню колонок на площині проекції** (тобто на екрані). Якщо крок кута між сусідами однаковий, то на уявному «ватмані» сусідні промені все одно «розходяться» не так, як сусідні пікселі — тому на краях кадру залишається дивне спотворення. Тут важливо не плутати «рівномірно по куту» і «рівномірно по ширині екрана»: саме друге нам і треба.

Тому нам потрібно ініціалізувати промені по-іншому:

Pascal — player.pas

```
begin
  for i := 0 to raysCount - 1 do
    player.rays[i].relativeAngle := Arctan(
      (i - gm^.windowSize.x * 0.5) / gm^.distanceToProjectionPlane
    );
end;
```

Чому саме `Arctan`? Уявіть площину проекції на відстані, що дорівнює `distanceToProjectionPlane` від ока. Колонка з номером `i` на екрані відповідає точці на цій площині, зміщеній вбік від центру екрана на

$$\Delta c = i - \frac{u_x}{2},$$

де $u_x$ — це `windowSize.x`, тобто ширина вікна в пікселях (у нашій моделі ми вважаємо, що один піксель по горизонталі відповідає одиниці «ширини» на площині проекції). З центру ока до цієї точки йде промінь: він утворює з напрямом погляду кут $\theta$, для якого з прямокутного трикутника («висота»

$d$, «основа» $\Delta$) випливає

$$\tan\theta = \frac{\Delta c}{d}, \quad \text{тобто} \quad \theta = \arctan\frac{\Delta c}{d}.$$

Саме цей $\theta$ і є `relativeAngle` для колонки `i`: промені рівномірно розходяться **по ширині** площини проекції, а не по дузі рівних кутів.

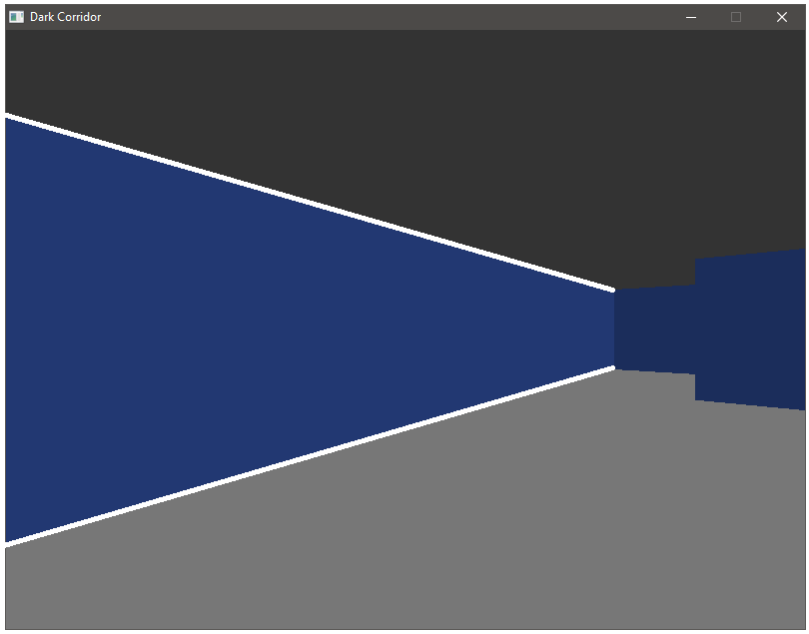



Рис. 5.8: Ефект риб'ячого ока відсутній

Тепер, після запуску програми, ми бачитимемо стіни рівними (Рис. 5.8)! Перспектива у нашому світі виглядає природно для «класичного» рушія з вертикальними стінами: ніяких викривлень, ніяких «риб'ячих очей» у тому вигляді, який ми щойно прибрали. (Звісно, це все ще спрощена модель: ми не обговорюємо тут нахил голови, складне освітлення чи криві поверхні — але для стін із квадратної сітки цього цілком достатньо, щоб повірити в 3D.) Якщо ви зуміли зрозуміти, що відбувається у цьому коді і як працює математичний апарат, то я вас вітаю! Ви тепер знаєте, як працює рейкастинг. Можна сказати, що ви навчились розуміти і, у певній мірі, навіть відчувати 3D-перспективу.

Попри це все, наша подорож ще триває. Монотонно розфарбовані стіни

— це, звичайно, цікаво, але наш світ був би значно живішим, якби стіни ми покрили **текстурами**!

## 5.3 Новий модуль — Кольори

Перед тим, як переходити до текстур, давайте виправимо одну проблему у нашому коді. У нашому файлі `game.pas` ми скрізь наче і задаємо конфігурацію нашої програми, але інформація про кольори все-одно залишається «зашитою» у код. Наприклад, якщо ви захочете змінити колір підлоги, що ви зробите? Ви полізете прямо в реалізацію алгоритму шукати, в якому місці коду ви додали значення. А що, якщо цей колір вам доведеться додати у новому місці? Загалом, від таких «магічних» чисел нам варто позбавлятися.

Ще одною проблемою нашого коду є те, що колір представляється у вигляді чотирьох чисел. З одного боку, цього вимагає SDL (функції вимагають чотири числа типу `u8`), але все ж хотілося б об'єднати це в одну структуру.

І це ще не все. У реалізації `Ray3DRender` ми рахуємо вручну значення кольору із меншою інтенсивністю:

**Pascal** world3drenderer.pas

```
if ray.isWallIntersectionVertical then
  intensity := intensity * 0.8;

...

SDL_SetRenderDrawColor(
  gm^.renderer,
  Round(34 * intensity),
  Round(56 * intensity),
  Round(114 * intensity),
  255);
```

Це дуже поганий код, оскільки, по-перше, ми не знаємо, що таке `34`, `56`, `114`. По-друге, ми делегуємо процедурі `Ray3DRender` задачу порахувати колір із меншою інтенсивністю (тобто одна функція стає відповідальною за кілька різних задач).

Тому давайте створимо новий модуль `colors`, у який перенесемо те, що стосується кольорів:

Pascal colors.pas

```
unit colors;

interface

uses types;

type
  TColor = record
    r: u8;
    g: u8;
    b: u8;
    a: u8;
  end;

  TFColor = record
    r: f32;
    g: f32;
    b: f32;
    a: f32;
  end;

function ColorRGB(r: u8; g: u8; b: u8): TColor;

function ColorWithIntensity(color: TColor; intensity: f32): TColor;
```

```
function ColorToFColor(color: TColor): TFColor;

implementation

function ColorRGB(r: u8; g: u8; b: u8): TColor;
begin
  ColorRGB.r := r;
  ColorRGB.g := g;
  ColorRGB.b := b;
  ColorRGB.a := High(u8);
end;

function ColorWithIntensity(color: TColor; intensity: f32): TColor;
begin
  ColorWithIntensity.r := Round(color.r * intensity);
  ColorWithIntensity.g := Round(color.g * intensity);
  ColorWithIntensity.b := Round(color.b * intensity);
  ColorWithIntensity.a := color.a;
end;

function ColorToFColor(color: TColor): TFColor;
begin
  ColorToFColor.r := color.r * (1.0 / High(u8));
  ColorToFColor.g := color.g * (1.0 / High(u8));
  ColorToFColor.b := color.b * (1.0 / High(u8));
  ColorToFColor.a := color.a * (1.0 / High(u8));
end;

end.
```

Тепер ви можете переробити ваш код так, щоб колір був частиною конфігурації проекту, і, наприклад, проблемний код вище виглядав би якось так:

Pascal world3drenderer.pas

```
screenFloorColor := gm^.world.floorColor;
screenWallColor := gm^.world.wallColor;
screenCeilingColor := gm^.world.ceilingColor;

if ray.isWallIntersectionVertical then
  screenWallColor := ColorWithIntensity(
    screenWallColor,
    intensityFactor);

...

SDL_SetRenderDrawColor(
  gm^.renderer,
  screenWallColor.r,
  screenWallColor.g,
  screenWallColor.b,
  screenWallColor.a);
```

Так же значно простіше все читається, правда? Загалом, якщо бажаєте, дуже рекомендую відповідним чином переробити проект, щоб більше не задумуватись про магічні значення. І всі потрібні числа будуть таким чином зберігатись у файлі `game.pas`:

Pascal game.pas

```
program game;

uses gameManager, vectors, types, math, colors;

const
  mapWidth = 8; {ширина карти (розмір вздовж координати x)}
  mapHeight = 8;  {висота карти (розмір вздовж координати y)}
```

```
var
  {0 - порожньо, 1 - стіна}
  map: array[0..mapWidth * mapHeight - 1] of u8 = (
    1,1,1,1,1,1,1,1,
    1,0,0,0,0,0,0,1,
    1,0,0,0,0,1,0,1,
    1,0,0,0,0,1,0,1,
    1,0,0,1,1,0,0,1,
    1,0,0,0,0,0,0,1,
    1,0,0,0,0,0,0,1,
    1,1,1,1,1,1,1,1
  );
  gm: TGameManager;
  gmCreateParams: TGameManagerCreateParameters;
  gmCreateStatus: TGameManagerCreateStatus;

begin
  gmCreateParams.gameName := 'Dark Corridor';
  gmCreateParams.windowWidth := 800;
  gmCreateParams.windowHeight := 600;
  gmCreateParams.targetFps := 60;

  gmCreateParams.world.wallColor := ColorRGB(34, 56, 114);
  gmCreateParams.world.mapBackgroundColor := ColorRGB(25, 30, 40);
  gmCreateParams.world.ceilingColor := ColorRGB(51, 51, 51);
  gmCreateParams.world.floorColor := ColorRGB(119, 119, 119);

  gmCreateParams.world.walls.map := map;
  gmCreateParams.world.walls.width := mapWidth;
  gmCreateParams.world.walls.height := mapHeight;
  gmCreateParams.world.walls.tileSize := 50;
  gmCreateParams.world.walls.mapWallColor := ColorRGB(0, 0, 0);
  gmCreateParams.world.walls.mapFreeZoneColor :=
```

```
    ColorRGB(255, 255, 255);

  gmCreateParams.world.player.position :=
    VectorCartesian(150, 150);
  gmCreateParams.world.player.speed := 50;
  gmCreateParams.world.player.angle := -80 * pi / 180.0;
  gmCreateParams.world.player.angularSpeed := 1.0;
  gmCreateParams.world.player.fov := 90 * pi / 180.0;
  gmCreateParams.world.player.mapColor := ColorRGB(255, 0, 0);
  gmCreateParams.world.player.rayMapColor := ColorRGB(230, 230, 230);

  gmCreateStatus := GameManagerInitialize(gmCreateParams, gm);
  if gmCreateStatus <> TGameManagerCreateStatus.Ok then
  begin
    WriteLn('Could not create game manager!');
    Halt(1);
  end;

  GameManagerRun(gm);
  GameManagerDestroy(gm);
end.
```

Тепер абсолютно всі ігрові дані знову знаходяться у конфігураційному файлі. Невже це не красиво?

## 5.4 Текстуруємо стіни

Тепер ми готові до текстур! Навіщо потрібні монотонно зафарбовані поверхні, якщо у нас є можливість перетворити абстрактні і однакові поверхні в цегляну стіну, каміння печери чи навіть обшивку міжзоряного космічного корабля?

Ключовий аспект, який потрібно розуміти для того, щоб малювати текстури — це пропорції. Але у вас із цим проблем немає: ви ж якось зрозуміли

тему подібних трикутників!

Але перед тим, як починати «кодити», нам потрібно ще дещо налаштувати ...

### 5.4.1 SDL_Image

У нас є проблема. SDL не підтримує «з коробки» завантаження зображень у звичних нам форматах JPEG чи PNG. Але нам якось треба завантажити у нашу програму ці картинки!

Для цього можемо використати SDL_Image — це розширення бібліотеки SDL, яке розповсюджується у вигляді окремої спільної бібліотеки. Завантажити її можна тут:

https://github.com/libsdl-org/SDL_image/releases

Знайдіть якусь цікаву картинку (наприклад, цегляної стіни) для текстури нашої стіни.

Ви вже знаєте що робити: вибираєте варіант для вашої апаратної платформи і перекидаєте файл бібліотеки у папку `libs`. Зв'язки для SDL_Image уже є у модулі `sdl3`, тому більше нічого робити не потрібно.

### 5.4.2 Додаємо файли текстур в проект

По-перше, вам потрібні зображення текстур. Хороша новина: ви їх можете завантажити будь-де в Інтернеті, навіть намалювати самі! Ось цікавий сайт, де можна знайти текстури на свій смак:

https://opengameart.org/content/cc0-textures-1

Тепер у кореневій папці створіть папку `textures`, яка і відповідатиме за текстури. Ви можете помістити туди ті текстури, що вам подобаються.

Тепер нам потрібно, щоб папка з текстурами, яку ми додали в наш проект, переміщувалась у папку із виконуваним файлом. Для цього нам треба додати у скрипт файл зі скриптом збірки `xmake.lua` логіку переміщення папки:

Lua xmake.lua

```lua
after_build(
  function(target)
    local targetdir = target:targetdir()

    os.cp("libs/*.so", targetdir)
    os.cp("libs/*.dll", targetdir)
    os.cp("libs/*.dylib", targetdir)
    os.cp("textures", targetdir)
  end)
```

Спробуйте скомпілювати програму командою `xmake` і переконайтесь, що папка із текстурами знаходиться разом із виконуваним файлом після компіляції.

### 5.4.3 Текстури на карті

Тепер давайте подумаємо, яким чином ми вирішуватимемо, яка текстура буде у кожної «клітинки». Для цього глянемо ще раз, як карта представляється у нашій конфігурації:

Pascal game.pas

```pascal
map: array[0..mapWidth * mapHeight - 1] of u8 = (
  1,1,1,1,1,1,1,1,
  1,0,0,0,0,0,0,1,
  1,0,0,0,0,1,0,1,
  1,0,0,0,0,1,0,1,
  1,0,0,1,1,0,0,1,
  1,0,0,0,0,0,0,1,
  1,0,0,0,0,0,0,1,
  1,1,1,1,1,1,1,1
);
```

0 на карті позначає вільну зону, а одиниця — стіну. Тепер ми переробимо правило. Число 0 працюватиме, як і працює. Воно означатиме вільну зону,

проте будь-яке число більше за нуль, означатиме «тут стіна». Тобто ми, наприклад, можемо переробити карту на щось на кшталт:

Pascal game.pas

```
map: array[0..mapWidth * mapHeight - 1] of u8 = (
  1,1,1,1,1,1,1,1,
  1,0,0,0,0,0,0,1,
  1,0,0,0,0,2,0,1,
  3,0,0,0,0,2,0,1,
  3,0,0,2,2,0,0,1,
  1,0,0,0,0,0,0,1,
  1,0,0,0,0,0,0,1,
  1,1,1,1,1,1,1,1
);
```

І наразі від такої зміни поведінка нашої програми взагалі не зміниться. Але ми можемо сказати, що 1, наприклад, це цегляна стіна, 2 — дерев'яна, а 3 — металічна.

Саме так ми і будемо присвоювати текстури нашим стінам — просто через їхній індекс.

### 5.4.4 Новий модуль — Текстури

Створимо новий модуль `textures`, у якому ми будемо реалізовувати те, що стосується текстур. Почнемо з інтерфейсу:

Pascal textures.pas

```
unit textures;

interface

uses
  sdl3,
  colors,
```

```
  types;

type
  TTexture = record
    surface: PSDL_Surface;
    width: c_int;
    height: c_int;
  end;

function TextureLoadFromFile(
  const renderer: PSDL_Renderer;
  const path: PAnsiChar;
  var tex: TTexture): Boolean;

procedure TextureDestroy(var tex: TTexture);

function TextureGetPixel(
  const tex: TTexture;
  const x, y: c_int): TColor;

function TextureCreateError(
  const renderer: PSDL_Renderer;
  var tex: TTexture): Boolean;

implementation

{ TODO: Реалізація інтерфейсу }

end.
```

Як бачимо, нам знову потрібні функції для завантаження текстур із файлу і звільнення пам'яті:

Pascal textures.pas

```pascal
function TextureLoadFromFile(
  const renderer: PSDL_Renderer;
  const path: PAnsiChar;
  var tex: TTexture): Boolean;
var
  surf: PSDL_Surface;
begin
  surf := IMG_Load(path);
  if surf = nil then
    Exit(False);
  tex.surface := surf;
  tex.width := surf^.w;
  tex.height := surf^.h;
  TextureLoadFromFile := True;
end;

procedure TextureDestroy(var tex: TTexture);
begin
  if tex.surface <> nil then
  begin
    SDL_DestroySurface(tex.surface);
    tex.surface := nil;
  end;
  tex.width := 0;
  tex.height := 0;
end;
```

Розглянемо реалізацію цих двох функцій. У SDL картинка представляється в вигляді спеціального запису, що називається **поверхня**, у SDL він називається `TSDL_Surface`. Функцією `IMG_Load` ми читаємо всі пікселі із файлу і записуємо їх у поверхню, а `SDL_DestroySurface` звільняє поверхню.

При роботі із текстурою також нам потрібна функція для зчитува-

ння кольору одного пікселя. Для цього можна використати функцію `SDL_ReadSurfacePixel`, яка зчитує колір одного пікселя, але записує чотири числа, що відповідають RGBA, через вказівники. Ми ж створили власну функцію, яка повертає колір як запис, і буде працювати поверх `SDL_ReadSurfacePixel`.

Pascal textures.pas

```
function TextureGetPixel(
  const tex: TTexture;
  const x, y: c_int): TColor;
begin
  TextureGetPixel := ColorRGB(0, 0, 0);
  if tex.surface = nil then Exit;
  if (x < 0) or (y < 0) or (x >= tex.width) or (y >= tex.height)
    then Exit;

  SDL_ReadSurfacePixel(
    tex.surface,
    x,
    y,
    @TextureGetPixel.r,
    @TextureGetPixel.g,
    @TextureGetPixel.b,
    @TextureGetPixel.a
  );
end;
```

Нарешті, нам потрібна функція, яка згенерує спеціальну текстуру для випадку помилок, пов'язаних із текстурами. Наприклад, якщо текстура не може бути завантажена, ми можемо показати чорно-рожеву текстуру-плитку, як у рушії Source (такий візерунок легко помітити, тож серед розробників він став звичним сигналом, що щось пішло не так):

Pascal textures.pas

```pascal
function TextureCreateError(
  const renderer: PSDL_Renderer;
  var tex: TTexture): Boolean;
const
  DevFallbackSize = 64;
  DevFallbackCell = 8;
var
  surf: PSDL_Surface;
  x, y, cx, cy: c_int;
  magentaCell: Boolean;

  black: TColor;
  magenta: TColor;
  wcolor: TColor;
begin
  black := ColorRGB(0, 0, 0);
  magenta := ColorRGB(255, 0, 254);

  surf := SDL_CreateSurface(
    DevFallbackSize,
    DevFallbackSize,
    SDL_PIXELFORMAT_RGBA32);

  if surf = nil then Exit(False);

  for y := 0 to DevFallbackSize - 1 do
    for x := 0 to DevFallbackSize - 1 do
    begin
      cx := x div DevFallbackCell;
      cy := y div DevFallbackCell;
      magentaCell := ((cx + cy) mod 2) <> 0;
```

```
      if magentaCell then
        wcolor := magenta
      else
        wcolor := black;

      SDL_WriteSurfacePixel(
        surf, x, y,
        wcolor.r, wcolor.g, wcolor.b, wcolor.a);
    end;

  tex.surface := surf;
  tex.width := surf^.w;
  tex.height := surf^.h;

  TextureCreateError := True;
end;
```

Зверніть увагу, що у цьому випадку ми поверхню не завантажуємо із файлу, а генеруємо процедурно. Тому у створюємо ми її іншим способом: із допомогою функції `SDL_CreateSurface`.

### 5.4.5 Масив текстур

Тепер у нашому ігровому менеджері ми повинні додати створення і очищення текстур. Текстури ми будемо зберігати в масиві. У параметри імена файлів, де перший файл буде означати текстуру із індексом 1, другий файл — 2, тощо. Файл номер 0 буде завжди означати помилкову текстуру: саме ту чорно-рожеву плитку, що ми обговорили у попередній частині.

Код ініціалізації досить простий, розглянемо його:

Pascal gameManager.pas

```
type
  TGameManager = record
```

```
    ...

    { список текстур }
    textures: array of TTexture;
  end;

  TGameManagerCreateParameters = record
    ...

    { список файлов імен текстур. Перша текстура
      буде відповідати текстурі із індексом 1 }
    textures: array of PAnsiChar;
  end;

function GameManagerInitialize(
  const params: TGameManagerCreateParameters;
  var gm: TGameManager
): TGameManagerCreateStatus;
var
  ...

  loadTextureSuccess: Boolean;
  i: isize;
begin
  ...

  { кількість елементів у масиві gm.textures ---
    на 1 більше ніж у params.textures }
  SetLength(
    gm.textures,
    Length(params.textures) + 1);

  TextureCreateError(renderer, gm.textures[0]);
```

```
  for i := Low(params.textures) to High(params.textures) do
  begin
    loadTextureSuccess := TextureLoadFromFile(
      renderer,
      params.textures[i],
      gm.textures[i - Low(params.textures) + 1]);

    if not loadTextureSuccess then
      TextureCreateError(
        renderer,
        gm.textures[i - Low(params.textures) + 1]);
  end;

  Exit(TGameManagerCreateStatus.Ok);
end;

procedure GameManagerDestroy(var gm: TGameManager);
var
  i: isize;
begin
  for i := Low(gm.textures) to High(gm.textures) do
    TextureDestroy(gm.textures[i]);

  ...
end;
```

Пригадаємо тепер, як ми визначали, чи являється деяка точка стіною. Для цього у рамках рішення Задачі 18 ви створили функцію `IsWall`.

Нам потрібно її переробити. Тепер ця функція повинна повертати не булеве значення `True`/`False`, а вказівник на текстуру стіни, яку перетинає промінь. У випадку, якщо таку текстуру неможливо визначити, то повертається помилкова чорно-рожева текстура із індексом 0.

Ця функція називатиметься `WallGetTexture`:

Pascal walls.pas

```pascal
function WallGetTexture(
  const walls: TWalls;
  const point: TVector
): PTexture;
var
  gm: PGameManager;
  cellX, cellY: isize;
  textureIndex: u8;
begin
  gm := PGameManager(walls.gmPtr);
  cellX := Floor(point.x / walls.tileSize);
  cellY := Floor(point.y / walls.tileSize);

  {Область поза картою вважаємо стіною.}
  if (cellX < 0)
    or (cellX >= walls.width)
    or (cellY < 0)
    or (cellY >= walls.height)
  then
    Exit(@gm^.textures[0]);

  textureIndex := walls.map[cellY * walls.width + cellX];
  if textureIndex <> 0 then
    if textureIndex > High(gm^.textures) then
      WallGetTexture := @gm^.textures[0]
    else
      WallGetTexture := @gm^.textures[textureIndex]
  else
    WallGetTexture := nil;
end;
```

Як бачимо, у випадку, якщо точка із координатами `point` — це не стіна, функція просто поверне `nil`. Інакше — повернеться вказівник на текстуру

поточної стіни. Функція `IsWall` нам більше не потрібна, її повністю можна замінити цією.

Для того, щоб ми могли рендерити текстуровані вертикальні лінії стін, нам потрібно ще дещо: по-перше, променю треба «знати» якою є **текстура стіни**, яку він перетнув. Також йому необхідна інформація про те, наскільки далеко знаходиться точка перетину від **лівого краю сторони стіни вздовж неї** (лівого саме для гравця, а не для карти).

Тому функції `FindNearestHorizontalWall` і `FindNearestVerticalWall` повинні повертати не просто координати точки перетину, а і ті дані, що ми щойно обговорили. Для цього ми створимо новий запис для тих даних, що повертаються цими функціями і переробимо наші реалізації:

Pascal player.pas

```
type
  WallHitContext = record
    coords: TVector;
    texture: PTexture;
    offsetAlongWall: f32;
  end;

function FindNearestVerticalWall(
  var ray: TRay;
  const gm: PGameManager): WallHitContext;
var
  ...
  currentTexture: PTexture;
begin
  ...

  candidate := firstCandidate;
  while true do
  begin
    currentTexture := WallGetTexture(
```

```
      gm^.world.walls,
      VectorAdd(candidate, eps));

    if currentTexture <> nil then
    begin
      FindNearestVerticalWall.coords := candidate;
      FindNearestVerticalWall.texture := currentTexture;
      FindNearestVerticalWall.offsetAlongWall :=
        candidate.y - Floor(candidate.y / tileSize) * tileSize;
      if deltaCandidate.x > 0 then
        FindNearestVerticalWall.offsetAlongWall :=
          tileSize - FindNearestVerticalWall.offsetAlongWall;

      Exit;
    end;

    candidate := VectorAdd(candidate, deltaCandidate);
  end;
end;

function FindNearestHorizontalWall(
  var ray: TRay;
  const gm: PGameManager): WallHitContext;
var
  ...
  currentTexture: PTexture;
begin
  ...

  candidate := firstCandidate;
  while true do
  begin
    currentTexture := WallGetTexture(
```

```
      gm^.world.walls,
      VectorAdd(candidate, eps));

    if currentTexture <> nil then
    begin
      FindNearestHorizontalWall.coords := candidate;
      FindNearestHorizontalWall.texture := currentTexture;
      FindNearestHorizontalWall.offsetAlongWall :=
        candidate.x - Floor(candidate.x / tileSize) * tileSize;
      if deltaCandidate.y > 0 then
        FindNearestHorizontalWall.offsetAlongWall :=
          tileSize - FindNearestHorizontalWall.offsetAlongWall;

      Exit;
    end;

    candidate := VectorAdd(candidate, deltaCandidate);
  end;
end;
```

Поле `offsetAlongWall` — це відстань вздовж **ребра клітинки** від «початку» цієї сторони до точки влучання, у межах $[0, w]$, де $w =$ `tileSize`. Для **вертикальної** межі між клітинками («північ–південь» на карті) координата вздовж стіни зручно зчитується з `candidate.y`: залишок від ділення клітинки дає положення вздовж ребра. Для **горизонтальної** межі роль вздовж стіни відіграє `candidate.x` — там та сама ідея.

Чому ж тоді інколи беруть `tileSize - offsetAlongWall`? Текстура приклеєна до стіни у певному напрямку: ми хочемо, щоб з боку гравця «ліва» межа текстури завжди відповідала одному кінцю ребра, а «права» — іншому. Якщо промінь підходить до вертикальної стіни з боку **зростання** $x$ (`deltaCandidate.x > 0`), ми бачимо інший кінець ребра, ніж якщо він підходить з боку **спадання** $x$; тоді координату вздовж стіни треба **віддзеркалити** відносно довжини плитки. Для горизонтальної стіни те саме

робить перевірка `deltaCandidate.y > 0`: залежно від того, чи промінь іде «вгору» чи «вниз» по сітці, видимий «зріз» ребра змінюється на протилежний.

Відповідно, після того, як ми зможемо визначити, який перетин (горизонтальний чи вертикальний), є ближчим до персонажа, оновимо необхідні дані у промені:

Pascal player.pas

```
type
  ...

  TRay = record
    ...

    { де саме по довжині стіни (в межах
    однієї плитки, від 0 до її розміру)
    влучив промінь; це число перетворюють
    на номер стовпчика текстури,
    щоб малювати її не з краю,
    а з того місця, де був удар }
    offsetAlongWall: f32;

    texture: PTexture;
  end;

...

procedure UpdateNearestWallDistance(
  var ray: TRay;
  const gm: PGameManager
);
var
  horizontal: WallHitContext;
```

```
  vertical: WallHitContext;
    ...
begin
  start := gm^.world.player.position;
  horizontal := FindNearestHorizontalWall(ray, gm);
  vertical := FindNearestVerticalWall(ray, gm);

  { Відстань від гравця до перетину, не від початку світу (0,0). }
  horizontalSqMag := VectorMagnitudeSquared(
    VectorSub(horizontal.coords, start));
  verticalSqMag := VectorMagnitudeSquared(
    VectorSub(vertical.coords, start));

  if horizontalSqMag < verticalSqMag then begin
    ray.wallIntersection := horizontal.coords;
    ray.texture := horizontal.texture;
    ray.wallDistance := sqrt(horizontalSqMag);
    ray.offsetAlongWall := horizontal.offsetAlongWall;
    ray.isWallIntersectionVertical := False;
  end
  else begin
    ray.wallIntersection := vertical.coords;
    ray.texture := vertical.texture;
    ray.wallDistance := sqrt(verticalSqMag);
    ray.offsetAlongWall := vertical.offsetAlongWall;
    ray.isWallIntersectionVertical := True;
  end;
end;
```

Пам'ятаєте, як у папку `textures` ви поміщали файли зображень текстур? Пора додати назви цих файлів до конфігурації в `game.pas`:

Pascal game.pas

```
var
  ...
  textures: array[0..texturesLength - 1] of PAnsiChar = (
    'textures/redbrick.png',
    'textures/wood.png',
    'textures/mossystone.png'
  );

begin
  ...

  gmCreateParams.textures := textures;
```

Після цих досить масштабних змін ваша програма повинна повністю компілюватись. Хоч ми і додали фундамент для текстурування у рушій, поки що змін у самій програмі ви не помітите: ми ще не переробили наш рендеринг стіни. Але зараз ми це виправимо ...

### 5.4.6 Рендеримо текстуровану стіну

Наразі для того, щоб намалювати стіну, ми просто користуємось функцією для малювання лінії

Pascal world3drenderer.pas

```
SDL_SetRenderDrawColor(
  gm^.renderer,
  screenWallColor.r,
  screenWallColor.g,
  screenWallColor.b,
  screenWallColor.a
);
```

```
SDL_RenderLine(
  gm^.renderer,
  rayIndex,
  wallTopY,
  rayIndex,
  wallBottomY
);
```

Але зараз ситуація зміниться. По-перше, нам більше не потрібна змінна `screenWallColor` і поле `wallColor` запису `TWorld` взагалі! Ми їх замінимо текстурою.

На місці виклику цих двох функцій з'явиться виклик однієї, розробленої нами:

**Pascal** `world3drenderer.pas`

```
RayDrawWallVerticalLine(
  ray,
  rayIndex,
  wallTopY,
  wallBottomY,
  gm);
```

Тепер нам потрібно подумати, як же ми реалізуємо цю функцію.

Якщо подумати, у нас для цього усе є: промінь уже зберігає інформацію про те, яка текстура має малюватись, і наскільки далеко від лівого краю текстури знаходиться точка, яку ми хочемо малювати.

Текстура — це такий самий набір пікселів, як і «екранне полотно». Зазвичай текстурні координати позначають літерами $u$ (горизонтальні) та $v$ (вертикальні).

Знаючи `offsetAlongWall`, яке лежить у межах від `0` до `tileSize`, його можна перетворити у текстурну координату $u$, яка знаходиться в межах від `0` до `texture.width`.

Коли ми будемо малювати вертикальну лінію, ми, ітеруючись від `wallTopY` до `wallBottomY`, будемо перетворювати кожну $y$ у текстурну

координату $v$, яка може набути значення від `0` до `texture.height`.

Хороша новина у тому, що **обидва ці перетворення лінійні, тобто описуються звичайним рівнянням прямої**:

$$u = \text{round}\left(\frac{\text{offset} \cdot \text{textureWidth}}{\text{tileSize}}\right)$$

$$v = \text{round}\left(\text{textureHeight} \cdot \frac{y - \text{wallTopY}}{\text{wallBottomY} - \text{wallTopY}}\right)$$

На Рис. 5.9 показано, як співвідносяться координати світу і **текселі** (англ. *Texel*), тобто текстурні пікселі.

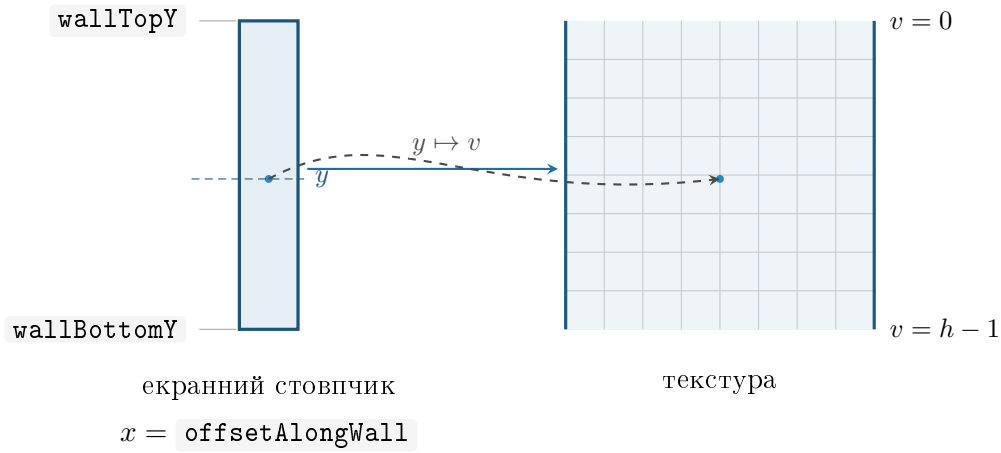



Рис. 5.9: Координати стовпчика перетворюються у координати текселів

Тепер поглянемо на реалізацію цієї функції:

Pascal world3drenderer.pas

```
procedure RayDrawWallVerticalLine(
  var ray: TRay;
  const rayIndex: isize;
  const wallTopY, wallBottomY: f32;
  const gm: PGameManager);
const
  intensityFactor = 0.6;
```

```
var
  tileSize: f32;
  tex: PTexture;
  texCol: c_int;
  texRow: c_int;
  wallH: f32;
  sy: isize;
  texel: TColor;
  screenH: isize;
begin
  if wallBottomY <= wallTopY then
    Exit;

  tex := ray.texture;
  tileSize := gm^.world.walls.tileSize;

  texCol :=
    Round((ray.offsetAlongWall / tileSize) * tex^.width);
  if texCol < 0 then
    texCol := 0;
  if texCol >= tex^.width then
    texCol := tex^.width - 1;

  wallH := wallBottomY - wallTopY;
  screenH := Round(gm^.windowSize.y);

  for sy := Max(0, Round(wallTopY))
         to Min(Round(wallBottomY), screenH - 1)
  do
  begin
    texRow := Round((sy - wallTopY) / wallH * tex^.height);
    if texRow < 0 then
      texRow := 0;
```

```
    if texRow >= tex^.height then
      texRow := tex^.height - 1;

    texel := TextureGetPixel(tex^, texCol, texRow);
    if ray.isWallIntersectionVertical then
      texel := ColorWithIntensity(texel, intensityFactor);

    SDL_SetRenderDrawColor(gm^.renderer,
      texel.r, texel.g, texel.b, texel.a);
    SDL_RenderPoint(gm^.renderer, rayIndex, sy);
  end;
end;
```

Тепер, запустивши програму, ви побачите прекрасну картинку, як на Рис. 5.10. Наші стіни повністю покриті малюнками, що зробило наш світ значно живішим і цікавішим.

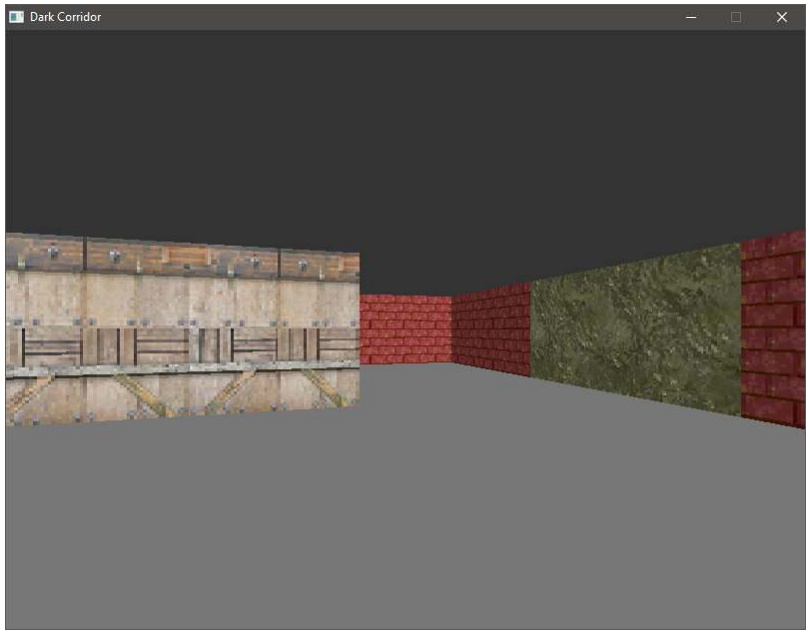



Рис. 5.10: Світ із текстурованими стінами.

Проте у нас виникла проблема: продуктивність програми дуже сильно просіла. Анімація нашого переміщення стала «рваною», рівень FPS дуже

сильно впав. Переміщуватись по нашому світу при таких умовах стає неприємно. Програма потребує оптимізації.

## 5.5 Екранна текстура

Давайте зрозуміємо, чому наша програма стала працювати погано. При програмуванні рендерингу ми до цього покладались на ряд функцій, таких як `SDL_SetRenderDrawColor`, `SDL_RenderPoint`, `SDL_RenderLine` тощо.

Ці функції добре працюють, коли нам потрібно намалювати абстрактну картинку, що складається із кількох геометричних фігур. Наприклад, для режиму карти цей підхід працює ідеально.

Проте, у випадку із текстурами, ми малюємо піксель за пікселем: для кожної точки стіни знову виставляємо колір і викликаємо `SDL_RenderPoint` — тобто звертаємося до API рендерера окремо для кожного такого пікселя, що й сповільнює програму. У режимі карти таких викликів небагато. Оцінімо масштаб: уявімо, що ми так намалювали б *кожен* піксель кадру — для вікна $800 \times 600$ це вже $800 \cdot 600 = 480000$ викликів на один кадр, тобто порядок кількості пікселів на екрані. Наш тривимірний вид наближається до такого порядку навантаження, тож викликів на кадр стає неймовірно багато. Тому ми повинні «обійти» рендерер і працювати напряму із пікселями, для чого ми і створимо власний буфер кольору.

> **Визначення**
>
> **Буфером кольору** (англ. *Color buffer*) ми називаємо масив пікселів, який ми використовуємо для зберігання кольорів пікселів на екрані.

Він має такий самий розмір, як і екран, і кожен піксель має свій колір. Для того, щоб зв'язати його з рендерером, ми створимо текстуру, але не ту текстуру `TTexture`, що ми створили в попередньому розділі, а спеціальну текстуру типу `TSDL_Texture`, яка надана нам бібліотекою SDL. Ця текстура коректно працює в тому числі із відеокартою, тому вона дуже швидко малюється на екрані.

У `TGameManager` ми додамо поля для буфера і текстури:

Pascal gameManager.pas

```
TGameManager = record
    ...

    screenTexture: PSDL_Texture;
    colorBuffer: array of TColor;
  end;

...

function GameManagerInitialize(
  const params: TGameManagerCreateParameters;
  var gm: TGameManager
): TGameManagerCreateStatus;
var
  ...
begin
  ...

  { створюємо екранну текстуру }
  gm.screenTexture := SDL_CreateTexture(
    renderer,
    SDL_PIXELFORMAT_RGBA32,
    SDL_TEXTUREACCESS_STREAMING,
    params.windowWidth,
    params.windowHeight);

  { створюємо буфер кольору }
  SetLength(gm.colorBuffer,
    params.windowWidth * params.windowHeight);
```

```
  ...
end;

procedure GameManagerDestroy(var gm: TGameManager);
var
  i: isize;
begin
  ...

  { звільняємо екранну текстуру }
  SDL_DestroyTexture(gm.screenTexture);

  ...
end;
```

Тепер ми можемо попіксельно писати у буфер кольору, а потім виводити його на екран. Розглянемо, як це робиться у функції малювання текстурованої вертикальної лінії стіни:

Pascal world3drenderer.pas

```
procedure RayDrawWallVerticalLine(
  var ray: TRay;
  const rayIndex: isize;
  const wallTopY, wallBottomY: f32;
  const gm: PGameManager);
const
  intensityFactor = 0.6;
var
  tileSize: f32;
  tex: PTexture;
  texCol: c_int;
  texRow: c_int;
  wallH: f32;
```

```
  sy: isize;
  screenW: isize;
  screenH: isize;
  bufIdx: isize;
  texel: TColor;
begin
  ...

  screenW := Round(gm^.windowSize.x);
  screenH := Round(gm^.windowSize.y);

  ...

  for sy := Max(0, Round(wallTopY)) to Min(Round(wallBottomY),
  ↪  screenH - 1) do
  begin
    ...

    bufIdx := sy * screenW + rayIndex;
    if bufIdx <= High(gm^.colorBuffer) then
      gm^.colorBuffer[bufIdx] := texel;
  end;
end;
```

Як бачите, ми просто записуємо колір у масив. Ніяких викликів функцій SDL на кожен піксель!

Розглянемо функцію `World3DRender`, яка тепер виглядає так:

**Pascal** world3drenderer.pas

```
procedure World3DRender(var world: TWorld; const gm: PGameManager);
var
  i: isize;
  colorBufferSize: isize;
```

```
begin
  colorBufferSize := Length(gm^.colorBuffer);

  { фарбуємо відразу всю стелю }
  for i := 0 to colorBufferSize div 2 - 1 do
    gm^.colorBuffer[i] := gm^.world.ceilingColor;

  { фарбуємо відразу всю підлогу }
  for i := colorBufferSize div 2 to colorBufferSize - 1 do
    gm^.colorBuffer[i] := gm^.world.floorColor;

  { рендеримо текстуровані стіни }
  for i := Low(world.player.rays) to High(world.player.rays) do
    Ray3DRender(world.player.rays[i], i, gm);

  SDL_UpdateTexture(
    gm^.screenTexture,
    nil,
    @gm^.colorBuffer[Low(gm^.colorBuffer)],
    Round(gm^.windowSize.x) * SizeOf(TColor));

  SDL_RenderClear(gm^.renderer);

  SDL_RenderTexture(gm^.renderer, gm^.screenTexture, nil, nil);
end;
```

Зверніть увагу, що ми можемо відразу заповнити першу половину буфера кольору кольором стелі, а другу половину — кольором підлоги. Це дозволяє нам заощадити час під час рендерингу променів: у функції `Ray3DRender` не потрібно знову обробляти стелю й підлогу — достатньо записати в буфер пікселі стін.

У функції `World3DRender` ми викликаємо `SDL_UpdateTexture`, щоб оновити текстуру тими кольорами, які ми записали у буфер кольору. Для

цього ми передаємо вказівник на перший елемент *буфера кольору* третім параметром. Зверніть увагу, що четвертий параметр — це розмір одного рядка в байтах. Оскільки кожен піксель має 4 байти (RGBA), то розмір одного рядка дорівнює ширині текстури, помноженій на 4. Це число 4 — результат виразу `SizeOf(TColor)`.

Сам рендеринг — це лише два виклики функцій SDL: `SDL_RenderClear` для очищення екрану і `SDL_RenderTexture` для виведення текстури на екран.

Скомпілюємо програму й переконаємося, що продуктивність значно зросла: картинка виглядає так само, як і раніше, проте переміщення по світу стало значно більш плавним.

## 5.6 Текстуруємо стелю і підлогу

Хоча Wolfenstein 3D і не підтримує текстурування підлоги і стіни, що зупиняє нас спробувати реалізувати таку «фічу»? Насправді, нічого! У 1993 році навіть вийшла гра **Blake Stone: Aliens of Gold** на модифікованій версії рушія гри Wolfestein 3D, у якій стеля і підлога уже були текстуровані! На Рис. 5.11 можна побачити, як це виглядає.


Рис. 5.11: Скріншот гри Blake Stone.

Тож ми цілком можемо це зробити; геометрія, що лежить в основі текстурування стелі і підлоги досить примітивна. Але є один нюанс:

> **Увага**
>
> Хоча підхід до текстурування стін і підлоги досить простий, варто тримати в голові, що для класичного програмного рейкаст-рендерингу текстурування підлоги і стін — найдорожча операція! Ви гарантовано відчуєте «просадку» продуктивності програми, яку буде важко оптимізувати.

### 5.6.1 Реалізовуємо текстури стелі і стін

По-перше, нам потрібно оновити інтерфейс модуля `world`:

Pascal — world.pas

```
type
  TWorldInitializeParameters = record
    walls: TWallsInitializeParameters;
    player: TPlayerInitializeParameters;
    mapBackgroundColor: TColor;

    floorTextureIndex: u8;
    ceilingTextureIndex: u8;
  end;

  TWorld = record
    walls: TWalls;
    player: TPlayer;
    gmPtr: Pointer;
    mapBackgroundColor: TColor;

    floorTextureIndex: u8;
    ceilingTextureIndex: u8;
```

```
  end;
```

Замість кольору стін ми тепер будемо використовувати номери текстур. Працюватимуть ці номери так само, як і для стін. 1 позначає першу текстуру, 2 — другу, тощо.

У модулі `world3drenderer` створимо нову функцію, яка малюватиме стелю і підлогу:

**Pascal** world3drenderer.pas

```
procedure DrawFloorAndCeiling(
  var world: TWorld;
  const gm: PGameManager);
```

Ідея функції полягає у тому, що спочатку ми проходимось по всіх координатах $y$ екрану. Протягом цього проходження ми визначаємо відстань до екранного горизонту (`pRow`), і вибираємо текстуру для стелі або підлоги.

**Pascal** world3drenderer.pas

```
tileSize := world.walls.tileSize;
proj := gm^.distanceToProjectionPlane;
horizon := screenH * 0.5;

if gm^.world.floorTextureIndex > High(gm^.textures) then
  floorTex := @gm^.textures[0]
else
  floorTex := @gm^.textures[gm^.world.floorTextureIndex];

if gm^.world.ceilingTextureIndex > High(gm^.textures) then
  ceilingTex := @gm^.textures[0]
else
  ceilingTex := @gm^.textures[gm^.world.ceilingTextureIndex];

for sy := 0 to screenH - 1 do
begin
```

```
  if Abs(sy - horizon) < 1e-3 then
    Continue;

  if sy < horizon then
  begin
    pRow := horizon - sy; { рядок над горизонтом }
    tex := ceilingTex;
  end
  else
  begin
    pRow := sy - horizon; { рядок під горизонтом }
    tex := floorTex;
  end;
```

Відстань `pRow` є дуже важливою: чим вона менша, тобто чим ближчою є вона до горизонту, тим далі від персонажа знаходиться відповідна точка підлоги або стелі.

Пам'ятаєте, як при визначення розмірів стіни на екрані ми застосовували подібність трикутників? Зараз зробимо аналогічно:

Pascal world3drenderer.pas

```
{ Базова відстань у горизонтальній площині вздовж
  "центрального" напрямку для цього рядка }
rowDist := world.walls.tileSize * 0.5 * proj / pRow;
```

Це – відстань вздовж напрямку погляду персонажа. Тепер, маючи всю інформацію поточного екранного рядка, ми повинні пройтись по всіх пікселях цього рядка:

Pascal world3drenderer.pas

```
for sx := 0 to screenW - 1 do
begin
  ...
```

Пам'ятаєте, як ми виправляли проблему «риб'ячого ока»? Ми визнача-

ли відстань вздовж напрямку погляду до точки замість відстань вздовж променя. **Зараз нам треба зробити навпаки**: визначити відстань вздовж променя до точки, тобто поділити, а не помножити на косинус відносного кута:

Pascal world3drenderer.pas

```
relAngle := world.player.rays[sx].relativeAngle;

{ Відстань у горизонтальній площині вздовж напрямку променя,
  інакше текстура "пливе" до країв. }
columnDist := rowDist / cos(relAngle);
```

Маючи відстань вздовж променя і кут напрямку променя, ми можемо визначити світові координати точки, в якій знаходиться відповідний піксель текстури.

Pascal world3drenderer.pas

```
rayAngle := world.player.angle + relAngle;
worldCoords := VectorAdd(
  world.player.position,
  VectorPolar(columnDist, rayAngle)
);
```

Далі буде все просто: маючи світові координати точки, ми можемо визначити координати текстурні. Для цього спочатку визначимо так звані UV-координати (текстурні координати, які знаходяться в межах від 0 до 1, найвища найлівіша точка текстури має координати (0,0), найправіша — (1,1)).

Pascal world3drenderer.pas

```
tu := TileFrac(worldCoords.x / tileSize);
tv := TileFrac(worldCoords.y / tileSize);
```

`TileFrac` — це допоміжна функція, яка повертає дробову частину числа. Реалізується досить просто:

Pascal world3drenderer.pas

```
function TileFrac(const coord: f32): f32;
begin
  TileFrac := coord - Floor(coord);
end;
```

Продовжуємо реалізацію основної функції. Маючи UV-координати, ми можемо визначити уже конкретні координати пікселів текстури. Це звичайна лінійна функція:

Pascal world3drenderer.pas

```
texCol := Round(tu * (tex^.width - 1));
texRow := Round(tv * (tex^.height - 1));
if texCol < 0 then
  texCol := 0;
if texRow < 0 then
  texRow := 0;
if texCol >= tex^.width then
  texCol := tex^.width - 1;
if texRow >= tex^.height then
  texRow := tex^.height - 1;
```

Тепер ми можемо записати колір пікселя текстури у буфер кольору:

Pascal world3drenderer.pas

```
texel := TextureGetPixel(tex^, texCol, texRow);

bufIdx := sy * screenW + sx;
gm^.colorBuffer[bufIdx] := texel;
```

Це все. Залишається лише викликати функцію `DrawFloorAndCeiling` у функції `World3DRender`:

Pascal world3drenderer.pas

```pascal
procedure World3DRender(var world: TWorld; const gm: PGameManager);
var
  i: isize;
begin
  DrawFloorAndCeiling(world, gm);

  { рендеримо текстуровані стіни }
  for i := Low(world.player.rays) to High(world.player.rays) do
    Ray3DRender(world.player.rays[i], i, gm);

  SDL_UpdateTexture(
    gm^.screenTexture,
    nil,
    @gm^.colorBuffer[Low(gm^.colorBuffer)],
    Round(gm^.windowSize.x) * SizeOf(TColor));

  SDL_RenderClear(gm^.renderer);

  SDL_RenderTexture(gm^.renderer, gm^.screenTexture, nil, nil);
end;
```

Тепер нам залишається лише скомпілювати програму і переконатися, що все працює. На Рис. 5.12 можна побачити, як виглядає гра після реалізації текстурування стелі і підлоги.

Тепер, маючи на озброєнні можливість текстурувати усі види поверхонь, ви можете створювати найрізноманітніші локації. Унікальні стіни, стеля і підлога нададуть вашій грі особливу атмосферу.

Ми ще, до речі, **навіть не реалізували весь потенціал текстурування стелі і підлоги**: наприклад, нам нічого не заважає зробити, як і зі стінами. Тобто не одну текстуру на всю карту, а щось унікальне на кожну клітинку (для цього, правда, потрібно трошки змінити структуру наших даних, але можливість така є). У цій книзі ми не будемо цього робити, за-

гальний підхід ви знаєте, проте чому б вам перед переходом на наступний розділ, не подумати б, як реалізувати таку «фічу»?

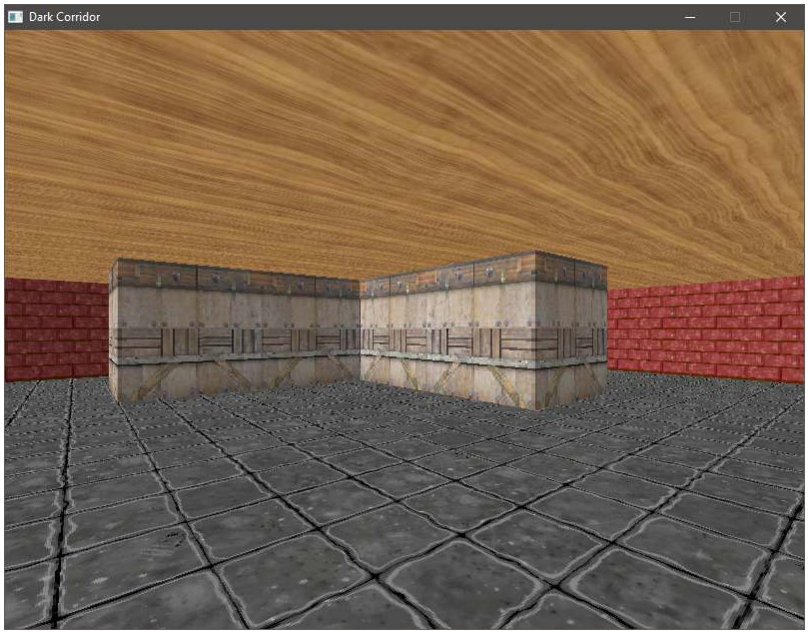



Рис. 5.12: Гра після реалізації текстурування стелі і підлоги.

На жаль, у текстуруванні підлоги і стелі є мінуси. Головний мінус — гірша продуктивність.

> **Запитання 10**
>
> Як ви вважаєте, що саме спричинює гіршу продуктивність при текстуруванні стелі і підлоги?

Існують кілька підходів до оптимізації. Наприклад, можна рідше оновлювати пікселі стелі і підлоги. Тобто міняти їх не кожен кадр, а кожні $N$ кадрів. Можна, наприклад, зменшити якість підлоги і стелі, тобто рахувати не кожен піксель, а кожен квадрат розміром $2 \times 2$ пікселів. Можна зробити припущення про те, що стеля і підлога повинні мати однакову текстуру і «дзеркалити» верхню половину екрану на іншу.

Загалом, можливі підходи до оптимізації я підказав, але їх реалізація виходитиме за рамки теми цієї книги. І, на жаль, вони найчастіше будуть

пов'язані із тим, що чимось ми жертвуємо. Втім, навіть Wolfenstein 3D, попри відсутні текстури вертикальних поверхонь, залишається чудовою грою, тому навіть якщо ми вирішите позбавитись від текстур стелі і підлоги в своїй грі, це ніяк не зробить ваш проект гіршим.

## 5.7 Ігрові об'єкти

Погодьтесь, що світ, який складається лише зі стін, трохи нуднуватий. Йому не вистачає меблів, NPC, предметів, рослин і всього іншого, що не має прямого відношення до геометрії карти.

Рейкаст-рушії, на жаль, не підтримують повноцінні 3D-моделі із власними текстурами, полігонами, анімацією, але технічні обмеження були вирішені просто: всі об'єкти у грі — це плоскі картинки, але із прозорим фоном. Більше того, усі ці об'єкти повернуті однією стороною до персонажа, і мають однаковий світовий розмір. Ширина і висота об'єкта у грі дорівнюють ширині «клітинки» на карті. Іншими словами, вони рівні `tileSize`.

Такі текстури із прозорим фоном, які зображають ігрові об'єкти, прийнято називати **спрайтами** (англ. *Sprite*).

### 5.7.1 Новий модуль — Ігровий об'єкт

Ігрові об'єкти у нашому проекті — максимально прості записи. Вони мають лише позицію у світових координатах, індекс спрайту (по-суті, індекс текстури), а також колір позначки на карті:

Pascal gameObjects.pas

```
type
  TGameObject = record
    position: TVector;
    spriteIndex: u8;
    mapColor: TColor;
```

```
  gmPtr: Pointer;
end;

TGameObjectInitializeParameters = record
  position: TVector;
  spriteIndex: u8;
  mapColor: TColor;
end;
```

**Задача 22**

Реалізуйте модуль `gameObjects`, а саме — функції ініціалізації ресурсів, звільнення ресурсів, оновлення і рендеринг в режимі карти. Список ігрових об'єктів має бути полем запису `TWorld`.

### 5.7.2 Буфер глибини

Після того, як ми намалювали стіни, нам потрібно іти далі і малювати текстури ігрових об'єктів.

Але у нас виникає проблема. Ми уже нарисували стіни! Припустимо, у нас є довільний об'єкт. Як визначити, чи ми його маємо малювати **за стіною**, чи **перед нею**? А що, якщо об'єкт виглядає з-за стіни лише частково? Для рішення цієї проблеми і створений так званий **буфер глибини** (англ. *Depth buffer*). Також можна почути термін «z-буфер».

Його ідея проста. Це звичайний числовий масив, розмір якого дорівнює розміру `colorBuffer`. Кожне число цього буфера позначає, наскільки далеко від позиції гравця знаходиться **піксель** із відповідного місця `colorBuffer`. Наприклад, якщо ми хочемо намалювати піксель, ми спочатку повинні будемо перевірити чи намальований уже піксель, що знаходиться ближче до нас? Якщо так, то `colorBuffer` не буде перезаписуватись. Все настільки просто.

Нам просто потрібно додати ще один масив в ігровий менеджер:

Pascal gameManager.pas

```
TGameManager = record
  ...
  colorBuffer: array of TColor;
  depthBuffer: array of f32;
end;
```

Ініціалізація цього буфера теж проста:

Pascal gameManager.pas

```
SetLength(gm.colorBuffer, params.windowWidth * params.windowHeight);
SetLength(gm.depthBuffer, Length(gm.colorBuffer));
```

При 3D-рендерингу ми спочатку ініціюємо буфер дуже великими значеннями. Для типу `f32` найкращим кандидатом для такого значення є `Infinity`, тобто нескінченність.

Pascal world3drenderer.pas

```
for i := Low(gm^.depthBuffer) to High(gm^.depthBuffer) do
  gm^.depthBuffer[i] := Infinity;
```

При рендерингу стелі і підлоги немає сенсу міняти значення буфера глибини: ці поверхні у будь-якому випадку мають найнижчий пріоритет показу. Будь-який об'єкт поверх стелі чи підлоги повинен бути показаний.

Зі стіною зовсім інша ситуація. При рендерингу стін ми уже перезаписуємо буфер глибини:

Pascal world3drenderer.pas

```
procedure RayDrawWallVerticalLine(
  var ray: TRay;
  const rayIndex: usize;
  const wallTopY, wallBottomY: f32;
  const gm: PGameManager);
...
```

```
begin
  ...

    bufIdx := sy * screenW + rayIndex;
    if bufIdx <= High(gm^.colorBuffer) then
    begin
      gm^.colorBuffer[bufIdx] := texel;
      gm^.depthBuffer[bufIdx] := ray.wallDistance;
    end;

  ...
end;
```

Після цих змін ми готові до основної частини рендерингу ігрових об'єктів. `depthBuffer` допоможе нам вирішувати, рисувати пікселі ігрових об'єктів чи ні.

### 5.7.3 Рендеримо ігрові об'єкти

Ну що, спробуємо намалювати спрайт?

Давайте обговоримо ідею того, як це буде робитись. Для того, щоб було легше орієнтуватись, можете поглянути на Рис. 5.13.

Ідея при малюванні досить проста: ми **взагалі не повертаємо спрайт на екрані**. Взагалі-взагалі. Все, що нас цікавить, — це наскільки далеко він знаходиться. Якщо він знаходиться близько, малюємо його великим; якщо далеко — малим. Але незмінним залишається одне — ми просто наносимо текстуру на екран без жодних поворотів.

Звичайно, як і у випадку стін, ми повинні визначати не пряму відстань між персонажем і спрайтом, а її проекцію на напрям погляду.

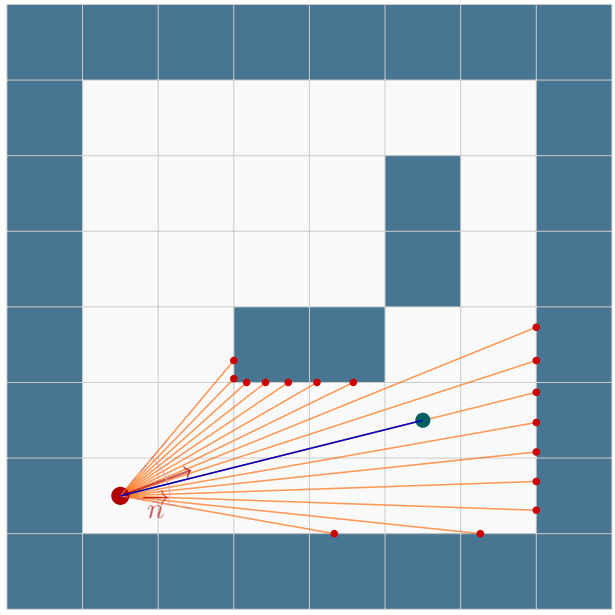


Рис. 5.13: Ігрова карта з променями рейкастингу та спрайтом.

Розглянемо тепер реалізацію.

У функції `World3DRender` після того, як ми намалювали стіни, проходимось по кожному нашому об'єкту і викликаємо функцію його малювання.

**Pascal** world3drenderer.pas

```
{ рендеримо текстуровані стіни }
for i := Low(world.player.rays) to High(world.player.rays) do
  Ray3DRender(world.player.rays[i], i, gm);

{ рендеримо ігрові об'єкти }
for i := Low(world.gameObjects) to High(world.gameObjects) do
  GameObject3DRender(world.gameObjects[i], gm);
```

Тепер перейдемо до головного — функції `GameObject3DRender`.

Спочатку нам потрібно зробити важливу оптимізацію. Ми хочемо відразу відсіяти ті об'єкти, що знаходиться не в області FOV, тобто яких персонаж не зможе побачити в принципі через те, що він повернутий у іншу сторону.

Для цього ми рахуємо кут між напрямком погляду гравця

`relativeAngle` і перевіряємо, чи знаходиться він у межав від $-\pi$ до $\pi$. Якщо не знаходиться, цей об'єкт ми можемо не малювати:

Pascal world3drenderer.pas

```
fromPlayer := VectorSub(
  gameObject.position,
  gm^.world.player.position
);

worldAngle := VectorAngle(fromPlayer);

{ нормалізація кута }
relativeAngle := VectorAngle(VectorPolar(
  1,
  worldAngle - gm^.world.player.angle));

inPlayerVisionCone := Abs(relativeAngle) < (gm^.fov / 2);
if not inPlayerVisionCone then Exit;
```

Зверніть увагу, як ми рахуємо `relativeAngle`. Замість того, щоб просто знайти різницю в кутах, ми її проганяємо через пару функцій `VectorAngle` і `VectorPolar`. Справа у тому, що ми хочемо викликати функцію arctan2, оскільки вона завжди повертає кут у межах між $-\pi$ і $\pi$. Уявіть, якщо різниця між `worldAngle` і `gm^.world.player.angle` дорівнюватиме $\frac{5\pi}{2}$. Це ж той самий кут, що і $\frac{\pi}{2}$, але одне значення більше за $\pi$, а інше — менше. Тому кути потрібно приводити до виду, зручного для порівняння. Саме проміжок від $-\pi$ до $\pi$ вважається найкращим для такого порівняння. Така операція називається **нормалізацією кута**.

Якщо ми впевнені, що персонаж повернутий у сторону спрайту, то ми уже можемо рахувати відстань до нього для того, щоб малювати. Для нас є важливою саме проекція відстані на вектор напрямку погляду:

Pascal world3drenderer.pas

```
distanceToPlayer := VectorMagnitude(fromPlayer);
distanceAcrossViewVector := distanceToPlayer * cos(relativeAngle);

{ не будемо малювати ЗАНАДТО близькі спрайти, це
  нормальна практика  }
if distanceAcrossViewVector < 0.05 then Exit;
```

Із допомогою подібності трикутників, яку ми уже не раз обговорювали, ми можемо порахувати розміри спрайту в екранному просторі.

Pascal world3drenderer.pas

```
projectedSpriteSize.x :=
  gm^.distanceToProjectionPlane *
  gm^.world.walls.tileSize / distanceAcrossViewVector;

projectedSpriteSize.y := projectedSpriteSize.x;
```

Далі визначаємо, **де** ми будемо малювати на екрані. Нам потрібно визначити вертикальну і горизонтальну межі квадрата, який ми малюватимемо.

Pascal world3drenderer.pas

```
spriteTopY :=
  Round((gm^.windowSize.y - projectedSpriteSize.y) * 0.5);

spriteBottomY :=
  Round((gm^.windowSize.y + projectedSpriteSize.y) * 0.5);

spriteScreenPositionX :=
  tan(relativeAngle) * gm^.distanceToProjectionPlane;

spriteLeftX :=
  Round(
```

```
    (gm^.windowSize.x - projectedSpriteSize.x) * 0.5 +
    spriteScreenPositionX
  );

spriteRightX :=
  Round(spriteLeftX + projectedSpriteSize.x);
```

Розрахуємо, яку текстуру хочемо рисувати, а також кілька змінних, які будуть корисні в подальшому коді.

Pascal world3drenderer.pas

```
if gameObject.spriteIndex > High(gm^.textures) then
  tex := @gm^.textures[0]
else
  tex := @gm^.textures[gameObject.spriteIndex];

screenW := Round(gm^.windowSize.x);
screenH := Round(gm^.windowSize.y);

spriteW := spriteRightX - spriteLeftX;
spriteH := spriteBottomY - spriteTopY;
```

Залишилось останнє: малювання пікселів. Тут все працює схожим чином, як і у випадку із пікселями підлоги і стелі:

Pascal world3drenderer.pas

```
if (spriteW <= 0) or (spriteH <= 0) then
  Exit;

if (spriteLeftX > spriteRightX) or (spriteTopY > spriteBottomY) then
  Exit;

for sy := Max(0, spriteTopY)
       to Min(screenH - 1, spriteBottomY - 1)
```

```
do
begin
  tv := (sy - spriteTopY) / spriteH;
  texRow := Round(tv * (tex^.height - 1));
  if texRow < 0 then
    texRow := 0;
  if texRow >= tex^.height then
    texRow := tex^.height - 1;

  for sx := Max(0, spriteLeftX)
         to Min(screenW - 1, spriteRightX - 1)
  do
  begin
    tu := (sx - spriteLeftX) / spriteW;
    texCol := Round(tu * (tex^.width - 1));
    if texCol < 0 then
      texCol := 0;
    if texCol >= tex^.width then
      texCol := tex^.width - 1;

    texel := TextureGetPixel(tex^, texCol, texRow);

    { колір 0xFF00FF вважаємо прозорим фоном }
    if (texel.r = 255) and (texel.g = 0) and (texel.b = 255) then
      Continue;

    bufIdx := usize(sy) * screenW + usize(sx);
    if bufIdx <= High(gm^.depthBuffer) then
      if distanceToPlayer < gm^.depthBuffer[bufIdx] then
      begin
        gm^.colorBuffer[bufIdx] := texel;
        gm^.depthBuffer[bufIdx] := distanceToPlayer;
      end;
```

```
  end;
end;
```

Зверніть увагу: ми не малюємо піксель, якщо уже на цьому місці був намальований піксель, що є ближчим до нас. Це ми визначаємо із допомогою буфера глибини, що ми обговорювали у попередній частині.

Також ми не малюємо прозорі пікселі. Якщо піксель прозорий, ми його просто пропускаємо і **не оновлюємо** буфер глибини. Як ми визначаємо, який піксель є прозорим?

Хороша новина: ви можете придумати будь-який «критерій прозорості». У коді вище, наприклад, прозорим являється той піксель, чий колір — magenta, тобто колір із кодом `FF00FF` (червоне — 255, зелене — 0, синє — 255). На Рис. 5.14 можна побачити приклад спрайту, який має бути прозорим.

Рис. 5.14: Спрайт стільця.

Тепер можемо спробувати запустити програму. І отримаємо зображення, схоже на Рис. 5.15. І ви просто подивіться на це! Наші спрайти працюють!

Залишається у нас лише одна проблема: коли ми повертаєтесь, і спрайти виходять за межі FOV, вони зникають занадто різко. Це напряму пов'язано із перевіркою:

```
inPlayerVisionCone := Abs(relativeAngle) < (gm^.fov / 2);
```

Кута FOV занадто мало для перевірки, чи треба малювати спрайт. Потрібно просто трошки збільшити кут, тобто дати об'єкту трохи більше простору для малювання.

```
inPlayerVisionCone := Abs(relativeAngle) < (gm^.fov / 2 + 0.3);
```

Тепер об'єкти будуть ховатись за межі екрану значно коректніше.

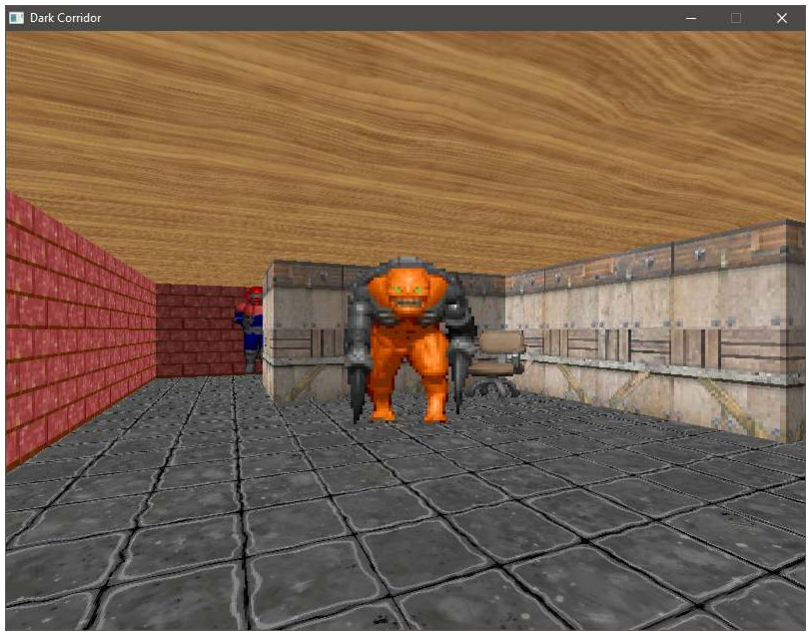



Рис. 5.15: Ігрова локація зі спрайтами об'єктів.

Головне — ми повністю завершили реалізацію відображення спрайтів. Тепер карту можна наповнювати об'єктами: додавати меблі, персонажів тощо. Загалом, робити світ «ненудним».

А найголовніше — ми повністю завершили те, що стосується 3D, на цьому матераіл книги можна вважати пройденим!

## 5.8 Висновки

У цьому розділі ми концентрувались на тривимірних аспектах світу. Ми спочатку намалювали геометрію карт. Потім ми почали текстурувати сті-

ни, стелю і підлогу. І, насамкінець, спробували рендерити спрайти ігрових об'єктів.

На жаль, рендеринг стелі і підлоги коштував нам продуктивності, проте можна або повністю позбавитись від цієї «фічі», або спробувати її оптимізувати ціною деталізації.

Ви, звичайно, можете спробувати зробити більше. Наприклад, можна додати ефект туману: чим далі знаходиться точка, яка рендериться, тим більш темною її малювати. Цілком можливо також зробити унікальні текстури стелі і підлоги для кожної клітинки, анімувати текстури теж реально. Можна навіть зробити стіни під кутом 45°. У грі Wolfenstein 3D були навіть двері, що відкриваються, і секрети, чому б не додати такі елементи на карту?

Якщо вам цікаво, можете скористатись цим посиланням:

https://lodev.org/cgtutor/index.html

Тут є серія англомовних статей про рейкастинг: тут можна знайти цікаві ідеї щодо того, що ще можна докинути у ваш рушій. Також тут можна знайти цікавий матеріал про комп'ютерну графіку загалом.

# Розділ 6

# Що далі?

У висновках розділу про тривимірний рендеринг я «накинув» кілька ідей про те, що ще можна зробити. Проте всі ті ідеї стосувалися виключно теми рендерингу і карти, і все-таки хотілося б подивитись на те, що ще можна зробити, трошки ширше.

Ви реалізували маленьку частину рушія, відповідальну за рендеринг.

Проте як щодо ігрової логіки? Ми вміємо лише ходити по статичній карті: у нас є лише перевірка перетину персонажа і стін, і все. Навіть крізь ігрові об'єкти ми проходимо, наче крізь примар. Світу потрібна інтерактивність, і тому ви можете спробувати, наприклад, реалізувати унікальну поведінку ігрових об'єктів.

Цю поведінку можна описати на Pascal або ж «прикрутити» до рушія скриптову мову. Прикладом такої мови є Lua, якою ми писали скрипт для XMake. На Lua, наприклад, пишуться макроси для **World of Warcraft** чи скрипти для **Roblox**. Також останнім часом набуває популярності мова Wren.

Можна покращити код, що стосується карти. Зараз у вас карта «зашита» у код, проте чому б не зробити можливість читати її з файлу.

Поки що ваш світ — німа кінострічка: є картинка, але немає звукового супроводу. SDL вміє відтворювати звук, тож можна додати хоча б кроки по коридору, звуки оточення чи просту мелодію на фоні — і гра одразу почне «тримати» гравця інакше.

Також можна реалізувати меню: як головне, так і меню паузи.

І, що, мабуть, є найважливішим, нашій гри не вистачає гри, як би див-

но це не звучало. Можна дати вашому персонажу здібності: навчити його стріляти, прискорюватись, використовувати магію, «хакати» комп'ютерні системи чи сповільнювати час. Можете додати RPG-систему: створити персонажів, діалоги, торгівлю тощо. А також чому б не спробувати додати простенький інтелект ворогам? Загалом, ваша гра — ваші правила і ваша фантазія!

Не можу також не дати рекомендації щодо того, що вчити далі. Почнемо із того, що стосується програмування в цілому, не тільки ігор:

- **Англійська — ваш друг!** Уміння як мінімум читати інформацію цією мовою — це величезний плюс, адже 80% якісної літератури та ресурсів написані саме англійською.
- Якщо ви хочете зайнятись програмуванням, рекомендую опанувати мову C. Вона може здатися застарілою, проте саме на ній написані не лише легендарні ігри на кшталт Wolfenstein 3D, Doom і Quake, а й мільйони інших програм. Бібліотека SDL, якою ви вже користувались, теж написана на C. C і досі лишається головною мовою для системного програмування, тому навіть якщо ви писатимете свої проекти іншою мовою, ви все одно зустрінетесь із викликом функцій, оголошених у стилі C.
- Учіть **Алгоритми і структури даних**. Ця дисципліна дуже важлива для будь-кого, хто програмує, оскільки без цих знань майже неможливо писати швидкий і ефективний код.

Тепер щодо навичок, що стосуються ігрової розробки:

- У книзі не раз згадувалась **лінійна алгебра**. Це дуже цікава дисципліна, що вивчає векторні і матричні обчислення. Предмет цей викладають в університетах, проте не на всіх спеціальностях і факультетах. Але в Інтернеті інформації безліч!
- Спробуйте опанувати якийсь ігровий рушій. Хорошою точкою входу для новачків є рушій Godot. Але можна пробувати і Unity або Unreal.

- Серед мов програмування найчастіше в «геймдеві» використовуються C++ та C#.

Насамкінець, хотілося б побажати удачі й натхнення в майбутніх починаннях, незалежно від того, стосуються вони ігрової розробки чи ні.

# Кілька слів про Pascal

Вперше і, мабуть, востаннє я торкався мови Pascal, коли ще був школярем, тобто за 13 років до моменту, коли я вирішив написати цю книгу.

В університеті моїми основними мовами програмування були C, C++ та C#, і за цей час я уже встиг забути, як Pascal виглядає. Проте, коли я вирішив написати цей матеріал, я зрозумів, що жодна із цих мов не підійде для пояснення комп'ютерної графіки школярам. C та C++ занадто близькі до апаратного забезпечення, також мені б довелось пояснювати арифметику вказівників, препроцесор, лінкування та інші нудні концепції, які до комп'ютерної графіки мають слабке відношення. C++, C# чи Java, як мови програмування, величезні! Вони мають дуже багато «фіч», а для їх мінімального розуміння потрібно розуміти складніші парадигми розробки, такі як **об'єктно-орієнтоване програмування**.

Існують, звичайно, деякі мови нового покоління, такі як Go, Rust чи Zig. Go, наприклад, теж відрізняється винятковою простотою, і у мене виникла велика спокуса вибрати саме її, поки не почав писати код. Я зрозумів, що ці мови відразу змушують тебе написати продукт, а не експериментувати. Створив змінну, яку не використовуєш? Помилка! Назвав змінну не у правильному стилі? Попередження! Зберігаєш файл? IDE відразу автоматично відредагує вміст твого файлу, щоб код був написаний у правильному стилі. Це зручно для програмістів, які професійно займаються програмуванням, проте, все ж, шкідливо для тих, хто тільки вчиться. Мова програмування повинна давати поле для експериментів для тих, хто робить свої перші кроки.

Тож я вибрав Pascal і переконався, що не дарма. Pascal має дуже гарний синтаксис, який легко читається. Мова надає всі інструменти для того,

щоб писати навіть щось складне. Ми цілий ігровий рушій зробили! Завдяки `cdecl` ми можемо викликати код, написаний на C, що дає нам доступ до безлічі бібліотек! При цьому, пишучи цією мовою, ви не лізете в область низькорівневої розробки: в цьому плані ми використовували лише вказівники і то лише для роботи із SDL і для вирішення одної циклічної залежності. При цьому Pascal дає повну свободу «погратися» і «понаступати на граблі»: саме те, що потрібно тим, хто навчається.

Попри те, що я сказав вище, у Pascal, на жаль, є кілька проблем. Головна його проблема — це те, що мова слабко розвивається. Навколо Pascal, на жаль, немає сильної спільноти, яка б задала чіткий вектор розвитку, і вона не пропонує деяких дуже важливих речей, які існують в інших мовах десятки років. Немає, наприклад, зручного менеджера пакетів, через що нам довелось писати власні зв'язки для SDL. Немає можливості оголосити змінну й одразу її використати — таке уже давно є в усіх сучасних мовах. Стандартна бібліотека теж дуже бідна. Сучасні IDE, на жаль, теж мають дуже обмежену підтримку Pascal. Тому сьогодні програми пишуть на інших мовах: вони більш розвинуті і мають більше можливостей, проте багато із них розвивались саме, надихаючись Pascal.

Я ж не жалію про свій вибір мови для цієї книги. Як «мова для навчання програмуванню як систематичній дисципліні» вона залишається найкращою! Проте варто йти вперед і знайомитись із новими для себе мовами програмування — у кожній знайдеться своя парадигма й те бачення розробки, яке «пропагандують» її творці. Чергове таке знайомство дозволяє глянути на власний досвід під іншим кутом. Можна сказати, що кожен такий інструмент відкриває нову філософію розробки. Тому не бійтесь експериментувати, шукати нові шляхи й знаходити нові рішення!

# Додатки

## Додаток 1: Код модуля sdl3

Pascal sdl3.pas

```pascal
unit sdl3;

{$PACKRECORDS C}

interface

uses
    types;

const
    {$IFDEF WINDOWS}
    SDL3_LIB = 'SDL3.dll';
    SDL3_IMAGE_LIB = 'SDL3_image.dll';
    {$ELSE}
        {$IFDEF DARWIN}
        SDL3_LIB = 'libSDL3.dylib';
        SDL3_IMAGE_LIB = 'libSDL3_image.dylib';
        {$ELSE}
        SDL3_LIB = 'libSDL3.so';
        SDL3_IMAGE_LIB = 'libSDL3_image.so';
        {$ENDIF}
    {$ENDIF}
```

```
type
    Pc_int = ^c_int;

    SDL_Bool = i32;
    SDL_PropertiesID = u32;
    SDL_WindowFlags = u64;
    SDL_RendererFlags = u32;
    SDL_EventType = u32;
    SDL_Keycode = i32;

    PSDL_Window = ^SDL_Window;
    PSDL_Renderer = ^SDL_Renderer;
    PSDL_Texture = ^SDL_Texture;
    PSDL_Surface = ^SDL_Surface;
    PSDL_FRect = ^SDL_FRect;
    PSDL_Event = ^SDL_Event;
    PSDL_FPoint = ^SDL_FPoint;
    PSDL_FColor = ^SDL_FColor;
    PSDL_Vertex = ^SDL_Vertex;

    SDL_Window = record end;
    SDL_Renderer = record end;
    SDL_Texture = record end;

    SDL_Surface = record
        flags: u32;
        format: u32;
        w: c_int;
        h: c_int;
        pitch: c_int;
        pixels: Pointer;
        refcount: c_int;
        reserved: Pointer;
```

```
    end;

    SDL_Rect = record
        x: i32;
        y: i32;
        w: i32;
        h: i32;
    end;

    PSDL_Rect = ^SDL_Rect;

    SDL_FRect = record
        x: f32;
        y: f32;
        w: f32;
        h: f32;
    end;

    SDL_FPoint = record
        x: f32;
        y: f32;
    end;

    SDL_FColor = record
        r: f32;
        g: f32;
        b: f32;
        a: f32;
    end;

    SDL_Vertex = record
        position: SDL_FPoint;
        color: SDL_FColor;
```

```
        tex_coord: SDL_FPoint;
    end;

    SDL_CommonEvent = record
        event_type: u32;
        reserved: u32;
        timestamp: u64;
    end;

    SDL_WindowEvent = record
        event_type: u32;
        reserved: u32;
        timestamp: u64;
        windowID: u32;
        data1: i32;
        data2: i32;
    end;

    SDL_KeyboardEvent = record
        event_type: u32;
        reserved: u32;
        timestamp: u64;
        windowID: u32;
        which: u32;
        scancode: u32;
        key: SDL_Keycode;
        modifier: u16;
        raw: u16;
        down: SDL_Bool;
        repeat_: SDL_Bool;
    end;

    SDL_MouseMotionEvent = record
```

```
        event_type: u32;
        reserved: u32;
        timestamp: u64;
        windowID: u32;
        which: u32;
        state: u32;
        x: f32;
        y: f32;
        xrel: f32;
        yrel: f32;
    end;

    SDL_MouseButtonEvent = record
        event_type: u32;
        reserved: u32;
        timestamp: u64;
        windowID: u32;
        which: u32;
        button: u8;
        down: SDL_Bool;
        clicks: u8;
        padding: u8;
        x: f32;
        y: f32;
    end;

    SDL_MouseWheelEvent = record
        event_type: u32;
        reserved: u32;
        timestamp: u64;
        windowID: u32;
        which: u32;
        x: f32;
```

```
        y: f32;
        direction: u32;
        mouse_x: f32;
        mouse_y: f32;
    end;

    SDL_QuitEvent = record
        event_type: u32;
        reserved: u32;
        timestamp: u64;
    end;

    SDL_Event = record
        case i32 of
            0: (event_type: u32);
            1: (common: SDL_CommonEvent);
            2: (window: SDL_WindowEvent);
            3: (key: SDL_KeyboardEvent);
            4: (motion: SDL_MouseMotionEvent);
            5: (button: SDL_MouseButtonEvent);
            6: (wheel: SDL_MouseWheelEvent);
            7: (quit: SDL_QuitEvent);
            8: (padding: array[0..127] of u8);
    end;

const
    SDL_FALSE = 0;
    SDL_TRUE = 1;

    SDL_INIT_AUDIO = $00000010;
    SDL_INIT_VIDEO = $00000020;
    SDL_INIT_JOYSTICK = $00000200;
    SDL_INIT_HAPTIC = $00001000;
```

```
    SDL_INIT_GAMEPAD = $00002000;
    SDL_INIT_EVENTS = $00004000;
    SDL_INIT_SENSOR = $00008000;
    SDL_INIT_CAMERA = $00010000;

    SDL_EVENT_FIRST = $0000;
    SDL_EVENT_QUIT = $0100;
    SDL_EVENT_WINDOW_SHOWN = $0201;
    SDL_EVENT_WINDOW_HIDDEN = $0202;
    SDL_EVENT_WINDOW_EXPOSED = $0203;
    SDL_EVENT_WINDOW_MOVED = $0204;
    SDL_EVENT_WINDOW_RESIZED = $0205;
    SDL_EVENT_WINDOW_PIXEL_SIZE_CHANGED = $0206;
    SDL_EVENT_WINDOW_MINIMIZED = $0207;
    SDL_EVENT_WINDOW_MAXIMIZED = $0208;
    SDL_EVENT_WINDOW_RESTORED = $0209;
    SDL_EVENT_WINDOW_MOUSE_ENTER = $020A;
    SDL_EVENT_WINDOW_MOUSE_LEAVE = $020B;
    SDL_EVENT_WINDOW_FOCUS_GAINED = $020C;
    SDL_EVENT_WINDOW_FOCUS_LOST = $020D;
    SDL_EVENT_WINDOW_CLOSE_REQUESTED = $020E;
    SDL_EVENT_KEY_DOWN = $0300;
    SDL_EVENT_KEY_UP = $0301;
    SDL_EVENT_MOUSE_MOTION = $0400;
    SDL_EVENT_MOUSE_BUTTON_DOWN = $0401;
    SDL_EVENT_MOUSE_BUTTON_UP = $0402;
    SDL_EVENT_MOUSE_WHEEL = $0403;

    { SDL_Keycode values from SDL3 SDL_keycode.h (letters + arrows) }
    SDLK_a = SDL_Keycode($00000061);
    SDLK_d = SDL_Keycode($00000064);
    SDLK_s = SDL_Keycode($00000073);
    SDLK_w = SDL_Keycode($00000077);
```

```
    SDLK_RIGHT = SDL_Keycode($4000004f);
    SDLK_LEFT = SDL_Keycode($40000050);
    SDLK_DOWN = SDL_Keycode($40000051);
    SDLK_UP = SDL_Keycode($40000052);
    SDLK_TAB = SDL_Keycode($00000009);

    SDL_WINDOW_FULLSCREEN = SDL_WindowFlags($0000000000000001);
    SDL_WINDOW_OPENGL = SDL_WindowFlags($0000000000000002);
    SDL_WINDOW_OCCLUDED = SDL_WindowFlags($0000000000000004);
    SDL_WINDOW_HIDDEN = SDL_WindowFlags($0000000000000008);
    SDL_WINDOW_BORDERLESS = SDL_WindowFlags($0000000000000010);
    SDL_WINDOW_RESIZABLE = SDL_WindowFlags($0000000000000020);
    SDL_WINDOW_MINIMIZED = SDL_WindowFlags($0000000000000040);
    SDL_WINDOW_MAXIMIZED = SDL_WindowFlags($0000000000000080);
    SDL_WINDOW_MOUSE_GRABBED = SDL_WindowFlags($0000000000000100);
    SDL_WINDOW_INPUT_FOCUS = SDL_WindowFlags($0000000000000200);
    SDL_WINDOW_MOUSE_FOCUS = SDL_WindowFlags($0000000000000400);
    SDL_WINDOW_EXTERNAL = SDL_WindowFlags($0000000000000800);
    SDL_WINDOW_HIGH_PIXEL_DENSITY =
        SDL_WindowFlags($0000000000002000);
    SDL_WINDOW_ALWAYS_ON_TOP = SDL_WindowFlags($0000000000008000);
    SDL_WINDOW_UTILITY = SDL_WindowFlags($0000000000020000);
    SDL_WINDOW_TOOLTIP = SDL_WindowFlags($0000000000040000);
    SDL_WINDOW_POPUP_MENU = SDL_WindowFlags($0000000000080000);
    SDL_WINDOW_TRANSPARENT = SDL_WindowFlags($0000000000100000);
    SDL_WINDOW_NOT_FOCUSABLE = SDL_WindowFlags($0000000000200000);

    SDL_RENDERER_SOFTWARE = SDL_RendererFlags($00000001);
    SDL_RENDERER_ACCELERATED = SDL_RendererFlags($00000002);
    SDL_RENDERER_PRESENTVSYNC = SDL_RendererFlags($00000004);

    SDL_PIXELFORMAT_RGBA8888 = $16462004;
    SDL_PIXELFORMAT_ABGR8888 = $16762004;
```

```
    {$IFDEF ENDIAN_LITTLE}
    SDL_PIXELFORMAT_RGBA32 = SDL_PIXELFORMAT_ABGR8888;
    {$ELSE}
    SDL_PIXELFORMAT_RGBA32 = SDL_PIXELFORMAT_RGBA8888;
    {$ENDIF}

    { SDL_TextureAccess (SDL3 SDL_render.h) }
    SDL_TEXTUREACCESS_STATIC = 0;
    SDL_TEXTUREACCESS_STREAMING = 1;
    SDL_TEXTUREACCESS_TARGET = 2;

function SDL_Init(flags: u32): SDL_Bool; cdecl; external SDL3_LIB;
procedure SDL_Quit; cdecl; external SDL3_LIB;

function SDL_CreateWindow(
    title: PAnsiChar;
    w: c_int;
    h: c_int;
    flags: SDL_WindowFlags): PSDL_Window; cdecl; external SDL3_LIB;

procedure SDL_DestroyWindow(
    window: PSDL_Window); cdecl; external SDL3_LIB;

function SDL_GetWindowID(
    window: PSDL_Window): u32; cdecl; external SDL3_LIB;

function SDL_CreateRenderer(
    window: PSDL_Window;
    name: PAnsiChar): PSDL_Renderer; cdecl; external SDL3_LIB;

procedure SDL_DestroyRenderer(
    renderer: PSDL_Renderer); cdecl; external SDL3_LIB;
```

```
function SDL_CreateTexture(
    renderer: PSDL_Renderer;
    pixelFormat: u32;
    texAccess: c_int;
    w: c_int;
    h: c_int): PSDL_Texture; cdecl; external SDL3_LIB;

function SDL_CreateTextureFromSurface(
    renderer: PSDL_Renderer;
    surface: PSDL_Surface): PSDL_Texture; cdecl; external SDL3_LIB;

procedure SDL_DestroyTexture(
    texture: PSDL_Texture); cdecl; external SDL3_LIB;

procedure SDL_DestroySurface(
    surface: PSDL_Surface); cdecl; external SDL3_LIB;

function SDL_CreateSurface(
    width: c_int;
    height: c_int;
    format: u32): PSDL_Surface; cdecl; external SDL3_LIB;

function SDL_WriteSurfacePixel(
    surface: PSDL_Surface;
    x: c_int;
    y: c_int;
    r: u8;
    g: u8;
    b: u8;
    a: u8): Boolean; cdecl; external SDL3_LIB;

function SDL_ReadSurfacePixel(
    surface: PSDL_Surface;
```

```
    x: c_int;
    y: c_int;
    r: PByte;
    g: PByte;
    b: PByte;
    a: PByte): Boolean; cdecl; external SDL3_LIB;

function IMG_Load(
    file_: PAnsiChar): PSDL_Surface; cdecl; external SDL3_IMAGE_LIB;

function SDL_SetRenderDrawColor(
    renderer: PSDL_Renderer;
    r, g, b, a: u8): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_RenderClear(
    renderer: PSDL_Renderer): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_RenderTexture(
    renderer: PSDL_Renderer;
    texture: PSDL_Texture;
    srcrect: PSDL_FRect;
    dstrect: PSDL_FRect): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_RenderPresent(
    renderer: PSDL_Renderer): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_RenderFillRect(
    renderer: PSDL_Renderer;
    const rect: PSDL_FRect): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_RenderLine(
    renderer: PSDL_Renderer;
    x1: f32;
```

```
    y1: f32;
    x2: f32;
    y2: f32): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_RenderPoint(
    renderer: PSDL_Renderer;
    x: f32;
    y: f32): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_RenderGeometry(
    renderer: PSDL_Renderer;
    texture: PSDL_Texture;
    const vertices: PSDL_Vertex;
    num_vertices: c_int;
    const indices: Pc_int;
    num_indices: c_int): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_PollEvent(
    event: PSDL_Event): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_WaitEvent(
    event: PSDL_Event): SDL_Bool; cdecl; external SDL3_LIB;

function SDL_PumpEvents: SDL_Bool; cdecl; external SDL3_LIB;

procedure SDL_Delay(ms: u32); cdecl; external SDL3_LIB;
function SDL_GetTicks: u64; cdecl; external SDL3_LIB;
function SDL_GetError: PAnsiChar; cdecl; external SDL3_LIB;

function SDL_UpdateTexture(
    tex: PSDL_Texture;
    updateRect: PSDL_Rect;
    pixelData: Pointer;
```

```
    rowPitch: c_int): Boolean; cdecl; external SDL3_LIB;

procedure SDLHelper_FillDisk(
    renderer: PSDL_Renderer;
    const centerX: f32;
    const centerY: f32;
    const radius: f32;
    const r, g, b: u8);

implementation

procedure SDLHelper_FillDisk(
    renderer: PSDL_Renderer;
    const centerX: f32;
    const centerY: f32;
    const radius: f32;
    const r, g, b: u8);
const
    DiskSegments = 40;
var
    verts: array[0..DiskSegments] of SDL_Vertex;
    indices: array[0..3 * DiskSegments - 1] of c_int;
    i: isize;
    angle: f32;
    fr, fg, fb: f32;
    idx: isize;
begin
    if radius <= 0 then
        Exit;

    fr := r * (1.0 / 255.0);
    fg := g * (1.0 / 255.0);
    fb := b * (1.0 / 255.0);
```

```
    verts[0].position.x := centerX;
    verts[0].position.y := centerY;
    verts[0].color.r := fr;
    verts[0].color.g := fg;
    verts[0].color.b := fb;
    verts[0].color.a := 1.0;
    verts[0].tex_coord.x := 0;
    verts[0].tex_coord.y := 0;

    for i := 1 to DiskSegments do
    begin
        angle := (2.0 * Pi) * ((i - 1) / DiskSegments);
        verts[i].position.x := centerX + cos(angle) * radius;
        verts[i].position.y := centerY + sin(angle) * radius;
        verts[i].color.r := fr;
        verts[i].color.g := fg;
        verts[i].color.b := fb;
        verts[i].color.a := 1.0;
        verts[i].tex_coord.x := 0;
        verts[i].tex_coord.y := 0;
    end;

    idx := 0;
    for i := 1 to DiskSegments - 1 do
    begin
        indices[idx] := 0;
        indices[idx + 1] := i;
        indices[idx + 2] := i + 1;
        idx := idx + 3;
    end;
    indices[idx] := 0;
    indices[idx + 1] := DiskSegments;
```

```
    indices[idx + 2] := 1;

    SDL_RenderGeometry(
        renderer,
        nil,
        @verts[0],
        DiskSegments + 1,
        @indices[0],
        3 * DiskSegments);
end;

end.
```

# Відповіді

## Відповіді на запитання

**Запитання 1.** **Тангенс** кута $\alpha$ — це відношення протилежного катета до прилеглого у прямокутному трикутнику: $\tan\alpha = \frac{\text{протилежний}}{\text{прилеглий}}$. Область визначення тангенса — всі дійсні числа, крім $\alpha = \frac{\pi}{2} + \pi k$, $k \in \mathbb{Z}$, а область значень — $(-\infty, +\infty)$. **Арктангенс** — це обернена функція до тангенса: якщо $\tan\alpha = x$, то $\alpha = \arctan x$. Тобто вона дозволяє визначити кут, знаючи співвідношення катетів. Область визначення арктангенса — $(-\infty, +\infty)$, а область значень — $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$.

**Запитання 2.** **Радіан** — це одиниця вимірювання кутів. Один радіан — це центральний кут, що відповідає дузі, довжина якої дорівнює радіусу кола. Повне коло містить $2\pi$ радіан, тобто $360° = 2\pi$ рад. Звідси: 1 рад $\approx 57.3°$.

**Запитання 3.** Головна причина — обмежена обчислювальна потужність комп'ютерів початку 1990-х. Повноцінна тривимірна графіка потребує значно більше математичних операцій: множення матриць, перетворення координат у тривимірному просторі, визначення видимості поверхонь тощо. Спрощення геометрії до двовимірної сітки з однаковими блоками дозволило звести задачу до простих обчислень на площині, завдяки чому гра могла працювати плавно навіть на процесорі Intel 286. Це був свідомий компроміс між реалістичністю та швидкодією.

**Запитання 4.** Обчислення модуля вектора потребує операції квадратного кореня: $\|\overrightarrow{a}\| = \sqrt{a_x^2 + a_y^2}$. Квадратний корінь — відносно повільна операція для процесора. Але якщо нам потрібно лише **порівняти** довжини двох векторів, можна обійтись без нього: оскільки квадратний корінь — зростаюча функція, нерівність $\|\overrightarrow{a}\| > \|\overrightarrow{b}\|$ рівносильна нерівності $\|\overrightarrow{a}\|^2 > \|\overrightarrow{b}\|^2$.

При цьому $\|\overrightarrow{a}\|^2 = a_x^2 + a_y^2$ обчислюється лише додаванням та множенням, що значно швидше. У грі, де таке порівняння може виконуватись тисячі разів на кадр, ця оптимізація є дуже суттєвою.

**Запитання 5.** Якщо зберігається **кут**, повернути персонажа на 10° вправо дуже просто: $\theta_{\text{new}} = \theta + 10°$. Одне додавання — і готово.

Якщо ж зберігається **вектор** $\overrightarrow{n}$, потрібно виконати значно більше дій: спочатку знайти поточний кут $\theta = \arctan2(n_y, n_x)$, потім додати 10°, а потім обчислити новий вектор $\overrightarrow{n}_{\text{new}} = \begin{bmatrix} \cos(\theta + 10°) \\ \sin(\theta + 10°) \end{bmatrix}$. Це три операції замість однієї.

Натомість вектор зручніший, коли потрібно обчислити переміщення або побудувати промінь: маючи $\overrightarrow{n}$, можна одразу писати $\overrightarrow{p}_{\text{new}} = \overrightarrow{p} + s \cdot \overrightarrow{n}$ (де $s$ — швидкість), без додаткових викликів cos і sin.

**Запитання 6.** Без обмежень на форму стін будь-яка точка променя була б кандидатом на перетин із прямою, і нам би доводилось для кожної сторони стіни застосовувати рівняння перетину двох прямих, або перебирати мільярди точок на прямій в надії, що хоч одна виявиться перетином зі стіною (що дууууже повільно).

**Запитання 7.** **Константа** у програмі має значення, яке **відоме заздалегідь** (на етапі компіляції або як фіксоване значення на весь час роботи програми) і **не залежить** від того, хто і з якими аргументами викликав функцію. Натомість `halfPerimeter` треба порахувати з параметрів `a`, `b` і `c`, а ці числа **різні при кожному виклику** `GetTriangleArea`: то трикутник 5, 6, 7, то інший набір сторін. Тобто це не «незмінна величина всього проекту», а **проміжний результат обчислення** всередині функції. Для таких речей і потрібна **змінна** (або безіменний вираз без збереження — але тоді знову з'являться повтори в коді).

**Запитання 8.** **Переваги** одновимірного масиву: проста та щільна структура даних, краща локальність пам'яті (елементи йдуть підряд), легше передавати як звичайний буфер і часто простіше зберігати/читати з файлу.

**Недоліки**: потрібно вручну переводити координати в індекс $i = y \cdot w + x$, через що легше помилитись із межами або переплутати порядок індексації.

Двовимірний масив читається наочніше у коді, бо звернення виглядає як `map[y][x]` і краще відображає геометричний зміст задачі.

**Запитання 9.** По-перше статичні масиви значно швидші, бо їхній розмір відомий заздалегідь, і компілятор може згенерувати код для прямого доступу до елементів. По-друге, пам'ять під статичний масив повністю під нашим контролем: статичний масив може бути полем запису фіксованого розміру. Це може бути корисно, якщо заздалегідь відома кількість елементів масиву — це частина угоди про розташування полів у пам'яті.

**Запитання 10.** Основна причина — тригонометрія. Кожен піксель стелі і стіни вимагає окремий підрахунок тригонометричних функцій для того, щоб визначити точки у світових координатах.

## Відповіді на задачі

**Задача 1.** Довжина гіпотенузи — $\sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$.

**Задача 2.** Лівий край екрану відповідає лівому краю поля зору, правий — правому. Тому, у випадку наших зображень: той промінь, що знаходиться найлівіше від вектора $\overrightarrow{n}$, відповідає лівому краю екрану, кожен наступний промінь — правішому на екрані.

**Задача 3.** Перетин із прямою $x = 2$: підставимо $x = 2$ у рівняння $y = 2x + 3$, отримаємо: $y = 2 \cdot 2 + 3 = 7$. Точка перетину — $(2, 7)$. Перетин із прямою $y = 4$: підставимо $y = 4$: $4 = 2x + 3 \implies 2x = 1 \implies x = 0.5$. Точка перетину — $(0.5, 4)$.

**Задача 4.** Початок вектора $\overrightarrow{a}$ зображений у точці $(0, 0)$, а його кінець — у точці $(3, 2)$. Тобто його координати — $\begin{bmatrix} 3-0 \\ 2-0 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$. Початок вектора $\overrightarrow{b}$ зображений у точці $(2, 1)$, а його кінець — у точці $(5, 3)$. Тобто його координати — $\begin{bmatrix} 5-2 \\ 3-1 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$

**Задача 5.** $\overrightarrow{p} + \overrightarrow{\Delta p_1} + \overrightarrow{\Delta p_2} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ -1 \end{bmatrix} + \begin{bmatrix} -2 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 + 4 + (-2) \\ 3 + (-1) + 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$.

**Задача 6.** $\|\overrightarrow{v}\| = \sqrt{5^2 + 12^2} = \sqrt{25 + 144} = \sqrt{169} = 13.$

**Задача 7.**

1. $4\overrightarrow{v} = 4\begin{bmatrix}3\\1\end{bmatrix} = \begin{bmatrix}12\\4\end{bmatrix}.$
2. $-2\overrightarrow{v} = -2\begin{bmatrix}3\\1\end{bmatrix} = \begin{bmatrix}-6\\-2\end{bmatrix}.$

**Задача 8.** $\|\overrightarrow{a}\| = \sqrt{6^2 + 8^2} = \sqrt{36 + 64} = \sqrt{100} = 10$. Тоді $\widehat{\overrightarrow{a}} = \frac{1}{10}\begin{bmatrix}6\\8\end{bmatrix} = \begin{bmatrix}0.6\\0.8\end{bmatrix}$. Перевірка: $\sqrt{0.6^2 + 0.8^2} = \sqrt{0.36 + 0.64} = \sqrt{1} = 1$. ✓

**Задача 9.** $\overrightarrow{dir} = \begin{bmatrix}\cos 60°\\\sin 60°\end{bmatrix} = \begin{bmatrix}0.5\\\frac{\sqrt{3}}{2}\end{bmatrix} \approx \begin{bmatrix}0.5\\0.866\end{bmatrix}.$

**Задача 10.** $r = \sqrt{3^2 + 3^2} = \sqrt{18} = 3\sqrt{2} \approx 4.24$. $\theta = \arctan2(3, 3) = \arctan\frac{3}{3} = \arctan 1 = 45°$.

**Задача 11.**

1. Маса предмета — **скалярна** (одне число).
2. Позиція персонажа на карті — **векторна** (координати $x$ і $y$).
3. Кут повороту персонажа — **скалярна** (одне число).
4. Переміщення персонажа за один кадр — **векторна** (зміщення по $x$ і $y$).
5. Час між кадрами $\Delta t$ — **скалярна** (одне число).
6. Швидкість кулі (з урахуванням напрямку) — **векторна** (має і значення, і напрямок).

**Задача 12.**

1. $\Delta t = \frac{1}{\text{FPS}} = \frac{1}{50} = 0.02$ сек.
2. $\text{FPS} = \frac{1}{\Delta t} = \frac{1}{0.04} = 25$ FPS.

**Задача 13.** Вектор напрямку: $\overrightarrow{n} = \begin{bmatrix} \cos 0° \\ \sin 0° \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Вектор швидкості: $\overrightarrow{v} = s\overrightarrow{n} = 5\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$. Переміщення: $\overrightarrow{\Delta p} = \overrightarrow{v}\Delta t = \begin{bmatrix} 5 \\ 0 \end{bmatrix} \cdot 0.2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Нова позиція: $\overrightarrow{p}_{\text{new}} = \overrightarrow{p} + \overrightarrow{\Delta p} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$.

**Задача 14.** $\theta_{\text{new}} = \theta + \omega \cdot \Delta t = 30° + 90° \cdot 0.5 = 30° + 45° = 75°$.

**Задача 15.** Нижче — приклад, як можна реалізувати модуль, код знизу показує, як буде виглядати файл між `implementation` і `end.`.

Pascal — vectors.pas

```
implementation

const
  Epsilon: single = 1e-12;

function VectorCartesian(
  const x: single;
  const y: single): TVector;
begin
  VectorCartesian.x := x;
  VectorCartesian.y := y;
end;

function VectorPolar(
  const radius: single;
  const angle: single): TVector;
begin
  VectorPolar.x := radius * cos(angle);
  VectorPolar.y := radius * sin(angle);
end;
```

```
function VectorAdd(
  const lhs: TVector;
  const rhs: TVector): TVector;
begin
  VectorAdd.x := lhs.x + rhs.x;
  VectorAdd.y := lhs.y + rhs.y;
end;

function VectorSub(
  const lhs: TVector;
  const rhs: TVector): TVector;
begin
  VectorSub.x := lhs.x - rhs.x;
  VectorSub.y := lhs.y - rhs.y;
end;

function VectorMulScalar(
  const vec: TVector;
  const scalar: single): TVector;
begin
  VectorMulScalar.x := vec.x * scalar;
  VectorMulScalar.y := vec.y * scalar;
end;

function VectorDivScalar(
  const vec: TVector;
  const scalar: single): TVector;
begin
  VectorDivScalar.x := vec.x / scalar;
  VectorDivScalar.y := vec.y / scalar;
end;

function VectorMagnitude(const vec: TVector): single;
```

```
begin
  VectorMagnitude := sqrt(vec.x * vec.x + vec.y * vec.y);
end;

function VectorMagnitudeSquared(const vec: TVector): single;
begin
  VectorMagnitudeSquared := vec.x * vec.x + vec.y * vec.y;
end;

function VectorNormalize(const vec: TVector): TVector;
var
  magSq: single;
    magInv: single;
begin
  magSq := VectorMagnitudeSquared(vec);
  if magSq < Epsilon then
  begin
    VectorNormalize.x := 0;
    VectorNormalize.y := 0;
  end
  else
  begin
    {
    Операція ділення на квадратний корінь значно дорожча
    за просто операцію множення.
    Тому ми один раз порахуємо зворотнє значення модуля,
    і двічі помножимо на його значення
    }
    magInv := 1.0 / sqrt(magSq);
    VectorNormalize.x := vec.x * magInv;
    VectorNormalize.y := vec.y * magInv;
  end;
end;
```

```
function VectorAngle(const vec: TVector): single;
begin
  VectorAngle := arctan2(vec.y, vec.x);
end;

end.
```

Код із перевіркою правильності нашого модуля:

Pascal vectorsTest.pas

```
program vectorsTest;

uses vectors, math;

procedure AssertAlmostEq(const a: single; const b: single);
begin
  Assert(Abs(a - b) < 1e-4);
end;

procedure AssertVecEq(const u: TVector; const v: TVector);
begin
  AssertAlmostEq(u.x, v.x);
  AssertAlmostEq(u.y, v.y);
end;

var
  a: TVector;
  b: TVector;
  n: TVector;
begin
  a := VectorCartesian(3, 4);
  b := VectorAdd(VectorCartesian(1, 2), VectorCartesian(10, 20));
```

```
  AssertAlmostEq(VectorMagnitude(a), 5);

  AssertVecEq(
    VectorPolar(1, 0),
    VectorCartesian(1, 0));

  AssertVecEq(
    b,
    VectorCartesian(11, 22));

  AssertVecEq(
    VectorSub(VectorCartesian(5, 5), VectorCartesian(2, 3)),
    VectorCartesian(3, 2));

  AssertVecEq(
    VectorMulScalar(VectorCartesian(2, 3), 3),
    VectorCartesian(6, 9));

  AssertVecEq(
    VectorDivScalar(VectorCartesian(6, 8), 2),
    VectorCartesian(3, 4));

  n := VectorNormalize(VectorCartesian(3, 4));

  AssertAlmostEq(VectorMagnitude(n), 1);
  AssertAlmostEq(VectorAngle(VectorCartesian(1, 0)), 0);
  AssertAlmostEq(VectorAngle(VectorCartesian(0, 1)), Pi / 2);

  WriteLn('vectorsTest: OK');
end.
```

Для роботи `Assert` у FPC зазвичай потрібен прапорець **-Sa**, тобто для того, щоб скомпілювати програму із `Assert` потрібно писати:

**Bash**

```
fpc -Sa vectorsTest.pas
```

Інакше `Assert` буде просто ігноруватись!

Але в програму із тестами варто додати більше різних `Assert`-ів. Чим більше випадків ви покриєте тестами, тим більш ви будете впевнені в правильності вашого коду.

**Задача 16.** Після заміни типу `single` на `f32` у модулі `vectors` тести повинні лишитись зеленими без зміни математичної логіки. Це очікувано, бо `f32` — це той самий 32-бітний тип числа з плаваючою крапкою; ми лише робимо назву типу більш явною і узгодженою з підходом у книжці.

**Задача 17.** Найпростіший спосіб перевірити розміри — написати невеликий тест із `Assert` і `SizeOf` для кожного псевдоніма з модуля `types`.

**Pascal** typesSizeTest.pas

```
program typesSizeTest;

uses types;

begin
  Assert(SizeOf(u8) = 1); { 1 байт = 8 біт }
  Assert(SizeOf(i8) = 1); { 1 байт = 8 біт }
  Assert(SizeOf(u16) = 2); { 2 байти = 16 біт }
  Assert(SizeOf(i16) = 2); { 2 байти = 16 біт }
  Assert(SizeOf(u32) = 4); { 4 байти = 32 біт }
  Assert(SizeOf(i32) = 4); { 4 байти = 32 біт }
  Assert(SizeOf(u64) = 8); { 8 байтів = 64 біт }
  Assert(SizeOf(i64) = 8); { 8 байтів = 64 біт }
  Assert(SizeOf(f32) = 4); { 4 байти = 32 біт }
  Assert(SizeOf(f64) = 8); { 8 байтів = 64 біт }
  Assert(SizeOf(usize) = SizeOf(Pointer)); { розмір вказівника }
  Assert(SizeOf(isize) = SizeOf(Pointer)); { розмір вказівника }
  Assert(SizeOf(c_int) = 4); { 4 байти = 32 біт }
```

```
  Assert(SizeOf(c_uint) = 4); { 4 байти = 32 біт }

  {$IF DEFINED(WINDOWS) OR NOT DEFINED(CPU64)}
  Assert(SizeOf(c_long) = 4); { 4 байти = 32 біт }
  Assert(SizeOf(c_ulong) = 4); { 4 байти = 32 біт }
  {$ELSE}
  Assert(SizeOf(c_long) = 8); { 8 байтів = 64 біт }
  Assert(SizeOf(c_ulong) = 8); { 8 байтів = 64 біт }
  {$ENDIF}

  WriteLn('typesSizeTest: OK');
end.
```

**Задача 18.** Потрібно перевести точку зі «світових» координат у координати клітинки, а далі перевірити значення в комірці.

Pascal — iswallexample.pas

```
program iswallexample;
uses math, vectors, types;

const
  tileSize = 50.0; { розмір сторони стіни }
  mapWidth = 8; { ширина карти в клітинках }
  mapHeight = 8; { висота карти в клітинках }

var
  {0 - порожньо, 1 - стіна}
  map: array[0..mapWidth * mapHeight - 1] of u8 = (
    1,1,1,1,1,1,1,1,
    1,0,0,0,0,0,0,1,
    1,0,0,0,0,1,0,1,
    1,0,0,0,0,1,0,1,
    1,0,0,1,1,0,0,1,
    1,0,0,0,0,0,0,1,
```

```
    1,0,0,0,0,0,0,1,
    1,1,1,1,1,1,1,1
  );

function CellIndex(const x, y: isize): isize;
begin
  CellIndex := y * mapWidth + x;
end;

function IsWall(const p: TVector): Boolean;
var
  cellX, cellY: i32;
begin
  cellX := Floor(p.x / tileSize);
  cellY := Floor(p.y / tileSize);

  {Область поза картою вважаємо стіною.}
  if (cellX < 0)
    or (cellX >= mapWidth)
    or (cellY < 0)
    or (cellY >= mapHeight)
  then
    Exit(True);

  IsWall := map[CellIndex(cellX, cellY)] <> 0;
end;

procedure CalculateIsWallAndPrint(const p: TVector);
begin
  WriteLn(
    'Is point (', p.x, ', ', p.y, ') a wall? ',
    IsWall(p),
    '!'
```

```
  );
end;

begin
  CalculateIsWallAndPrint(VectorCartesian(150.0, 100.0));
  CalculateIsWallAndPrint(VectorCartesian(60.0, 30.0));
  CalculateIsWallAndPrint(VectorCartesian(-10.0, 50.0));
end.
```

**Задача 19.** Найзручніше порівнювати саме вказівники на батьків. Якщо в обох людей збігається адреса матері або адреса батька (і це не `nil`), то вони мають спільного з батьків і вважаються братами/сестрами.

Pascal — familytree.pas

```
function AreSiblings(const a, b: TPerson): Boolean;
var
  sameMother: Boolean;
  sameFather: Boolean;
begin
  {Одна й та сама людина не вважається братом/сестрою самій собі.}
  if @a = @b then
    Exit(False);

  sameMother := (a.mother <> nil) and (a.mother = b.mother);
  sameFather := (a.father <> nil) and (a.father = b.father);

  AreSiblings := sameMother or sameFather;
end;

begin
  ...

  WriteLn(AreSiblings(anna, davyd)); {True}
```

```
  WriteLn(AreSiblings(anna, oleksii)); {False}
  WriteLn(AreSiblings(anna, anna)); {False}
end.
```

### Задача 20.

У модулі `gameManager.pas` ініціалізація світу має відповідати розділу про циклічні залежності: `WorldInitialize(params.world, @gm, gm.world)` (передаємо адресу менеджера, а не `renderer` окремо).

Pascal — world.pas

```
unit world;

interface

uses types, vectors, sdl3;

type
  TStar = record
    position: TVector;
    radius: f32;
    r, g, b: u8;
  end;

  TStarInitializeParameters = record
    position: TVector;
    radius: f32;
    r, g, b: u8;
  end;

  TPlanet = record
    distanceFromStar: f32;
    angle: f32;
    radius: f32;
```

```
    angularSpeed: f32;
    r, g, b: u8;

    star: ^TStar;
  end;

  TPlanetInitializeParameters = record
    distanceFromStar: f32;
    angle: f32;
    radius: f32;
    angularSpeed: f32;
    r, g, b: u8;
  end;

  TWorldInitializeParameters = record
    star: TStarInitializeParameters;

    planet1: TPlanetInitializeParameters;
    planet2: TPlanetInitializeParameters;
  end;

  TWorld = record
    gmPtr: Pointer;
    star: TStar;
    planet1: TPlanet;
    planet2: TPlanet;
  end;

procedure WorldInitialize(
  const params: TWorldInitializeParameters;
  const gmPtr: Pointer;
  var world: TWorld);
```

```
procedure WorldDestroy(var world: TWorld);

procedure WorldUpdate(
  var world: TWorld;
  const deltaTimeSeconds: f32);

procedure WorldRender(var world: TWorld);

implementation

uses gameManager;

procedure WorldInitialize(
  const params: TWorldInitializeParameters;
  const gmPtr: Pointer;
  var world: TWorld);
begin
  world.gmPtr := gmPtr;

  world.star.position := params.star.position;
  world.star.radius := params.star.radius;
  world.star.r := params.star.r;
  world.star.g := params.star.g;
  world.star.b := params.star.b;

  world.planet1.distanceFromStar := params.planet1.distanceFromStar;
  world.planet1.angle := params.planet1.angle;
  world.planet1.radius := params.planet1.radius;
  world.planet1.angularSpeed := params.planet1.angularSpeed;
  world.planet1.r := params.planet1.r;
  world.planet1.g := params.planet1.g;
  world.planet1.b := params.planet1.b;
  world.planet1.star := @world.star;
```

```
  world.planet2.distanceFromStar := params.planet2.distanceFromStar;
  world.planet2.angle := params.planet2.angle;
  world.planet2.radius := params.planet2.radius;
  world.planet2.angularSpeed := params.planet2.angularSpeed;
  world.planet2.r := params.planet2.r;
  world.planet2.g := params.planet2.g;
  world.planet2.b := params.planet2.b;
  world.planet2.star := @world.star;
end;

procedure WorldDestroy(var world: TWorld);
begin
  {
  ми тут нічого не робимо: функція додана виключно для
  послідовності
  }
end;

procedure PlanetUpdate(
  var planet: TPlanet;
  const deltaTimeSeconds: f32
);
begin
  planet.angle :=
    planet.angle +
    planet.angularSpeed * deltaTimeSeconds;
end;

procedure WorldUpdate(
  var world: TWorld;
  const deltaTimeSeconds: f32);
begin
```

```
  PlanetUpdate(world.planet1, deltaTimeSeconds);
  PlanetUpdate(world.planet2, deltaTimeSeconds);
end;

procedure StarRender(var star: TStar; const renderer: PSDL_Renderer);
begin
  SDLHelper_FillDisk(
    renderer,
    star.position.x,
    star.position.y,
    star.radius,
    star.r,
    star.g,
    star.b);
end;

procedure PlanetRender(var planet: TPlanet; const renderer:
↪  PSDL_Renderer);
var
  planetPosition: TVector;
begin
  planetPosition := VectorAdd(
    planet.star^.position,
    VectorPolar(planet.distanceFromStar, planet.angle)
  );
  SDLHelper_FillDisk(
    renderer,
    planetPosition.x,
    planetPosition.y,
    planet.radius,
    planet.r,
    planet.g,
    planet.b);
```

```
end;

procedure WorldRender(var world: TWorld);
var
  gm: PGameManager;
begin
  gm := PGameManager(world.gmPtr);

  SDL_SetRenderDrawColor(gm^.renderer, 25, 30, 40, 255);
  SDL_RenderClear(gm^.renderer);

  StarRender(world.star, gm^.renderer);
  PlanetRender(world.planet1, gm^.renderer);
  PlanetRender(world.planet2, gm^.renderer);
end;

end.
```

**Pascal** game.pas

```
program game;

uses gameManager, vectors;

var
  gm: TGameManager;
  gmCreateParams: TGameManagerCreateParameters;
  gmCreateStatus: TGameManagerCreateStatus;

begin
  gmCreateParams.gameName := 'Dark Corridor';
  gmCreateParams.windowWidth := 800;
  gmCreateParams.windowHeight := 600;
  gmCreateParams.targetFps := 60;
```

```
gmCreateParams.world.star.position :=
  VectorCartesian(400.0, 300.0);
gmCreateParams.world.star.radius := 36.0;
gmCreateParams.world.star.r := 255;
gmCreateParams.world.star.g := 210;
gmCreateParams.world.star.b := 80;

gmCreateParams.world.planet1.distanceFromStar := 110.0;
gmCreateParams.world.planet1.angle := 0.0;
gmCreateParams.world.planet1.radius := 10.0;
gmCreateParams.world.planet1.angularSpeed := 0.9;
gmCreateParams.world.planet1.r := 120;
gmCreateParams.world.planet1.g := 180;
gmCreateParams.world.planet1.b := 255;

gmCreateParams.world.planet2.distanceFromStar := 200.0;
gmCreateParams.world.planet2.angle := 2.4;
gmCreateParams.world.planet2.radius := 14.0;
gmCreateParams.world.planet2.angularSpeed := 0.45;
gmCreateParams.world.planet2.r := 230;
gmCreateParams.world.planet2.g := 120;
gmCreateParams.world.planet2.b := 90;

gmCreateStatus := GameManagerInitialize(gmCreateParams, gm);
if gmCreateStatus <> TGameManagerCreateStatus.Ok then
begin
  WriteLn('Could not create game manager!');
  Halt(1);
end;

GameManagerRun(gm);
GameManagerDestroy(gm);
```

```
end.
```

### Задача 21.

Pascal player.pas

```
function IsFacingDown(const vector: TVector): f32;
begin
  if vector.y > 0 then
    IsFacingDown := 1.0
  else
    IsFacingDown := 0.0;
end;

function FindNearestHorizontalWall(
  var ray: TRay;
  const gm: PGameManager): TVector;
var
  firstCandidate: TVector;
  deltaCandidate: TVector;
  candidate: TVector;
  start: TVector;
  tileSize: f32;
  direction: TVector;
  eps: TVector;
begin
  start := gm^.world.player.position;
  tileSize := gm^.world.walls.tileSize;
  direction := VectorPolar(
    1,
    gm^.world.player.angle + ray.relativeAngle);
  eps := VectorEps(direction);

  firstCandidate.y := tileSize * (
    Floor(start.y / tileSize) +
```

```
    IsFacingDown(direction)
  );

  firstCandidate.x := start.x +
    (firstCandidate.y - start.y) *
    (direction.x / direction.y);

  deltaCandidate := VectorMulScalar(
    direction,
    tileSize / Abs(direction.y)
  );

  candidate := firstCandidate;
  while not IsWall(gm^.world.walls, VectorAdd(candidate, eps)) do
    candidate := VectorAdd(candidate, deltaCandidate);

  FindNearestHorizontalWall := candidate;
end;
```

### Задача 22.

Pascal — gameObjects.pas

```
unit gameObjects;

interface

uses types, vectors, colors;

type
  TGameObject = record
    position: TVector;
    spriteIndex: u8;
    mapColor: TColor;
```

```
    gmPtr: Pointer;
  end;

  TGameObjectInitializeParameters = record
    position: TVector;
    spriteIndex: u8;
    mapColor: TColor;
  end;

procedure GameObjectInitialize(
  const params: TGameObjectInitializeParameters;
  const gmPtr: Pointer;
  var gameObject: TGameObject);

procedure GameObjectDestroy(var gameObject: TGameObject);

procedure GameObjectUpdate(
  var gameObject: TGameObject;
  const deltaTimeSeconds: f32);

procedure GameObjectMapRender(var gameObject: TGameObject);

implementation

uses gameManager, sdl3, mapRendererHelpers;

procedure GameObjectInitialize(
  const params: TGameObjectInitializeParameters;
  const gmPtr: Pointer;
  var gameObject: TGameObject);
begin
  gameObject.position := params.position;
  gameObject.spriteIndex := params.spriteIndex;
```

```
  gameObject.mapColor := params.mapColor;
  gameObject.gmPtr := gmPtr;
end;

procedure GameObjectDestroy(var gameObject: TGameObject);
begin
end;

procedure GameObjectUpdate(
  var gameObject: TGameObject;
  const deltaTimeSeconds: f32);
begin
end;

procedure GameObjectMapRender(var gameObject: TGameObject);
var
  gm: PGameManager;
  screenGameObjectPosition: TVector;
begin
  gm := PGameManager(gameObject.gmPtr);
  screenGameObjectPosition := MapWorldSpaceToScreenSpace(
    gameObject.position,
    gm
  );

  { малюємо малунький круг, що позначить гравця }
  SDLHelper_FillDisk(
    gm^.renderer,
    screenGameObjectPosition.x,
    screenGameObjectPosition.y,
    5,
    gameObject.mapColor.r,
    gameObject.mapColor.g,
```

```
    gameObject.mapColor.b
  );
end;

end.
```

Pascal world.pas

```
unit world;

interface

uses types, vectors, sdl3, walls, player, colors, gameObjects;

type
  TWorldInitializeParameters = record
    ...
    gameObjects: array of TGameObjectInitializeParameters;
  end;

  TWorld = record
    ...
    gameObjects: array of TGameObject;
  end;

...

implementation

uses gameManager, world3drenderer;

procedure WorldInitialize(
  const params: TWorldInitializeParameters;
  const gmPtr: Pointer;
```

```
  var world: TWorld);
var
  i: isize;
begin
  ...

  SetLength(world.gameObjects, Length(params.gameObjects));
  for i := 0 to Length(params.gameObjects) - 1 do
    GameObjectInitialize(params.gameObjects[i], gmPtr,
    ↪  world.gameObjects[i]);
end;

procedure WorldDestroy(var world: TWorld);
var
  i: isize;
begin
  for i := 0 to Length(world.gameObjects) - 1 do
    GameObjectDestroy(world.gameObjects[i]);

  ...
end;

procedure WorldUpdate(
  var world: TWorld;
  const deltaTimeSeconds: f32);
var
  i: isize;
begin
  for i := 0 to Length(world.gameObjects) - 1 do
    GameObjectUpdate(world.gameObjects[i], deltaTimeSeconds);

  ...
end;
```

```
procedure WorldMapRender(var world: TWorld; const gm: PGameManager);
var
  i: isize;
begin
  ...

  WallsMapRender(world.walls);
  PlayerMapRender(world.player);
  for i := 0 to Length(world.gameObjects) - 1 do
    GameObjectMapRender(world.gameObjects[i]);
end;

...

end.
```