Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ГРАФІЧНІ БІБЛІОТЕКИ У МОВІ ПРОГРАМУВАННЯ HASKELL

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
доцент Проценко В. С.
_____
(підпис)
“____” _____ 2020 р.
Виконала студент
Магур П. В.
“____” _____ 2020 р.

Київ 2020

ЗМІСТ

Анотація

Короткий зміст роботи:

- Графічні бібліотеки в Haskell

У цьому розділі коротко розглядаються основні графічні бібліотеки, доступні для використання у Haskell. Метою цього розділу є змалювання загальної картини стану графіки в Haskell для того, щоб потім об'єктивно оцінити його можливості в цій сфері.

- OpenGL і Haskell

У цьому розділі детально описується принцип роботи OpenGL і особливості його реалізації в Хаскелі. Знання фундаментальних концепцій спростить пояснення коду в розділі №3.

- Застосовуючи теорію на практиці

У цьому розділі покроково описується практична частина цієї курсової роботи. Розглядаються основні засоби, за допомогою яких можна створювати складні додатки.

- Haskell і GameDev

У цьому розділі в загальному вигляді описується подальша перспектива використання графіки в Хаскелі, а саме – GameDev. Мета розділу – це звернути увагу читача, що таке використання Хаскелю можливе і має перспективи.

Ключові слова: графічна бібліотека, шейдер, сцена, освітлення, монади, глобальний стан, GUI бібліотека, Хаскель, OpenGL, GLFW, GLUT, Yampa.

Вступ

Два десятиліття тому ніхто і подумати не міг, що за допомогою «малювання картинок» на екрані можна заробляти на життя. Проте сьогодні комп’ютерна графіка – це одна з найперспективніших сфер.

Метою цієї роботи стало спростування найголовнішого міфу про Хаскель – що це суто «математична» мова програмування, що Хаскель – це лише лямбда-вирази. Навпаки, – це повноцінна мова програмування зі своїми особливостями, яка надає широкий набір можливостей на рівні з іншими мовами програмування. В процесі написання буде дано об’єктивну оцінку можливостей Хаскелю в цій сфері та які перспективи можливі для нього.

Крім того, ще одним поштовхом для написання цієї роботи стала відсутність повних і практичних прикладів застосування графіки у Хаскелі.

Детально розібравши одну з найвідоміших графічних бібліотек – OpenGL – варіанти реалізації якої охоплюють величезний спектр різних мов програмування (і Хаскель в тому числі), ми відобразимо об’єкт на сцені, джерело світла та реалізуємо логіку камери, використовуючи засоби Хаскелю.

Надалі ця робота може бути розвинена у дослідження не менш перспективної сфери – GameDev – яка цілком може поєднуватись з функціональним стилем Хаскелю.

1. ГРАФІЧНІ БІБЛІОТЕКИ В HASKELL

Графіка – напевно одна з найменш детально описаних тем в Хаскелі. Дійсно, робіт по цій тематиці можна перерахувати на пальцях, а більшість графічних бібліотек на Хаскелі працюють нестабільно. Тим не менше, вибір їх достатньо великий і специфічний.

1.1. GUI бібліотеки

Як відмово, GUI розшифровується як Graphical User Interface – тобто GUI бібліотеки – це ті бібліотеки, які відповідають за відображення графічного інтерфейсу програми, з яким користувач може взаємодіяти. Для Хаскелю існує безліч таких бібліотек, проте, на жаль, стандартної не існує і більшість з них неповні[1].

1.1.1. Низькорівневі бібліотеки

- *GLFW* та *GLUT*. Обидві бібліотеки є надбудовами над одноіменними C/C++ бібліотеками, які були створені для використання разом із графічною 3D бібліотекою OpenGL. Детальніше про те, як ними користуватись на практиці, буде описано в наступних розділах.
- *Win32*. Як зрозуміло з назви, це надбудова до так званого Win32 API, яке є набором функцій для створення різного виду вікон, програм, ігор під управлінням ОС Windows.

1.1.2. Високорівневі бібліотеки

На жаль, саме бібліотеки з високим рівнем абстракції в основному є експериментальними. Серед них:

- *Threepenny-gui* - це бібліотека, яка використовує для відображення браузер. Вона підтримує FRP (Functional Reactive Programming), про яке детальніше можна буде почитати в наступних розділах. Однак, недоліком є те, що ця бібліотека не має жодного стабільного релізу і її API час від часу сильно змінюється.
- *Webviewhs* - це хаскелівська надбудова над C++ бібліотекою *webview*. Як сказано в документації, ця бібліотека дозволяє створювати достатньо потужний і багатофункціональний UI. Як і попередня, вона також не має стабільного релізу(остання версія 0.1.0.0).
- *Fudgets* – бібліотека, яка активно розроблялась ще в середині 90-х, проте вона досі підтримується. Окрім створення GUI, її також можна використовувати для створення клієнт-серверних застосувань.
- *Fruit*. Також застосовує принцип FRP, проте уже деякий час не підтримується. Натомість існує альтернатива *wxFruit*, яка використовує ключові принципи *Fruit*, проте є надбудовою над ще однією бібліотекою *wxHaskell*.
- *SDL2* – надбудова над *SDL 2* (Simple DirectMedia Layer), до якого входить як низько-, так і високорівневе API. *SDL* надає доступ до аудіо, клавіатури, мишки, джойстика і графіки (через *OpenGL* або *Direct3D*).

1.2. Графічні 3D бібліотеки

Список почнемо із графічних 3D бібліотек:

- *LambdaCube 3D* – потужна бібліотека для програмування GPU, яка дозволяє створювати програми у чисто функціональному стилі.
- *HOpenGL* – надбудова над C/C++ реалізацією бібліотеки *OpenGL*. *HOpenGL* охоплює набір незалежний пакетів, серед яких:

    - *OpenGLRaw* – низькорівнева обгортка навколо *OpenGL*, яка практично повністю повторює API C бібліотеки.
    - *GLURaw* – схожа на *OpenGLRaw*, але є надбудовою лише для GLU (OpenGL Utility Library) частини *OpenGL*, яка використовує її функції для малювання складніших фігур.
    - *OpenGL* – високорівнена надбудова, яка перетворює низькорівневі функції *OpenGLRaw* у більш функціональний стиль, який відповідає Хаскелю. Саме її ми будемо використовувати в практичній частині цієї роботи.
    - *GLUT*(OpenGL Utility Toolkit) – про неї ужебуло згадано вище, це надбудова для *GLUT*, який, насправді, не підтримується уже протягом 20 років. Альтернативою для нього є *FreeGLUT*.
- *GPipe* – бібліотека, також створена для програмування GPU, і яка розроблялась як альтернатива до *OpenGL*. Перевагою бібліотеки є те, що вона створена в функціональному стилі і є Haskell орієнтованою, на відміну від *OpenGL*.

Інші бібліотеки:

- *Chart Library* – проста бібліотека для малювання 2D графіків.
- *Gloss* – з її допомогою можна малювати векторні зображення і навіть створювати анміації.
- *Diagrams* - фреймворк, який визначає спеціальну мову EDSL, за допомогою якої можна декларативно малювати різного роду діаграми і картинки на Haskell.
- *Grapefruit* – бібліотека для створення графічних анімацій в декларативному стилі. Також включає в себе GUI бібліотеку. Використовує принципи FRP.

І це далеко не повний список бібліотек. Детальніше про різноманіття вибору графічних бібліотек на Haskell можна прочитати на офіційному вікі (https://wiki.haskell.org/).

2. OPENGL I HASKELL

2.1. Основи OpenGL

2.1.1. OpenGL – не бібліотека

Варто почати з того, що OpenGL – це не зовсім повноцінна бібліотека, а швидше програмний інтерфейс для створення графіки. Це означає, що OpenGL – це в першу чергу специфікація, яка описує набір функцій і визначає їх точну поведінку, тобто це документ, який визначає вигляд API. Самою реалізацією цього API зазвичай займаються розробники апаратури (графічних карт), на яких цей OpenGL буде виконуватись. Перш, ніж певна реалізація може бути класифікована як OpenGL реалізація, вона має пройти ряд спеціальних тестів(conformance tests). Це так звані апаратні реалізації. Саме тому, якщо програміст помічає дивну поведінку OpenGL, зазвичай достатньо оновити драйвери графічної карти. В іншому випадку, потрібно зв’язатись з постачальником графічної апаратури.

Існують також програмні реалізації, які теоретично можуть запускатись на різних системах, і навіть якщо у комп’ютера відсутня графічна карта. Але такі реалізації виконуються довше, а деякі функції взагалі можуть бути недоступними. Апаратні реалізації набули широкого поширення і сьогодні доступні практично на будь-яких комп’ютерах з підтримкою 3D графіки.

OpenGL – процедурне API: щоб відобразити сцену з певним станом, програміст описує кроки, котрі необхідно виконати, щоб отримати таку сцену, замість того, щоб описувати, як ця сцена має виглядати.

2.1.1.1. Незалежність від платформи

OpenGL намагається максимально абстрагуватись від платформи, вона не містить ніяких GUI функцій, в тому числі функцій управління вікнами, вводом/виводом у файл, взаємодією користувача з програмою тощо. Адже кожна

така функція повністю залежить від платформи. Тому, не зважаючи на різноманіття функцій, які надає розробнику OpenGL, ви ніколи не знайдете там функцій операційних систем и графічних користувацьких інтерфейсів. Для реалізації таких дій ми можемо використовувати GUI бібліотеки, спеціально створені для роботи в прив'язці до OpenGL: GLUT та GLFW, SDL. Варто зазначити, що GLUT не пітримується уже довгий час, тому для нього існує альтернатива FreeGLUT, яку і рекомендовано використовувати замість GLUT. Загалом, вибір не обмежується цими бібліотеками, ми можемо використовувати будь-яку на наш розсуд.

2.1.2. Машина станів

Сам по собі OpenGL є великою машиною станів – набором змінних, які визначають як OpenGL повинен себе поводити в поточний момент часу. Цей стан часто називають *контекстом*.

Визначати всі змінні стану, коли ми хочемо щось намалювати, насправді було би дуже непрактично. Тому замість цього в OpenGL реалізована модель станів (або скінченний автомат), призначення якого відслідковувати всі змінні стану OpenGL. Встановлене значення стану залишається доти, доки якась функція його не змінить[2]. Використовуючи OpenGL, ми зазвичай змінюємо його стан, перевизначаючи деякі опції, заповнюючи буфери тощо і потім візуалізуємо це використовуючи контекст[3]. Більшість значень змінних - це просто мітки «включено» та «вимкнено». Наприклад, освітлення або включене, або вимкнене. Від цього залежить відображення інших об'єктів – геометрія, намальована без освітлення, малюється без застосування до кольорів розрахунків освітлення. Так само, кожного разу, коли ми хочемо відобразити одну фігуру замість іншої, наприклад, лінію замість трикутника, ми змінюємо стан OpenGL, перезаписуючи значення деяких контекстних змінних, які вказують OpenGL, що малювати.

2.1.2.1. Haskell та машина станів

Принцип машини станів насправді йде в розріз з ідеологією Хаскелю, який є «лінивою» мовою програмування. Це означає, що в ньому немає поняття змінного глобального стану, адже всі змінні є незмінними. До того ж всі функції та вирази в Хаскелі «чисті», тобто не можуть мати сторонніх ефектів. Вирази вираховуються лише тоді, коли це дійсно необхідно для подальшої роботи програми. Це означає, що може бути важко визначити порядок,в якому підвирази виконуються[4].

Як зазначалось вище, OpenGL зі свого боку працює на основі глобальної машини станів, значення якої використовуються для відображення. В програмах OpenGL функції виконуються одна за одною з метою зміни певних змінних стану.

Саме тому при практичному використанні OpenGL, нам доведеться постійно користуватись монадами, які дозволяють змоделювати зміни стану.

Зміна значення стану – базова операція OpenGL, в HOpenGL виконується за допомогою спеціального оператора $=, який фактично виконує функції оператора присвоєння. Визначення сеттера можна знайти на *лістингу 2.1*:

```
infixr 2 $=
class HasSetter s where
    ($=) :: s a -> a -> IO ()
```

Лістинг 2.1

Всі змінні HOpenGL, значення яких можна змінити, мають тип *SettableStateVar*[4].

Не менш важливою за встановлення значення змінної операцією є отримання значення. Це можливе за допомогою використання функції *get*, визначення якої можна знайти на *лістингу 2.2*:

```
class HasGetter g where
    get :: g a -> IO a
```

Лістинг 2.2

Аналогічно до сеттера, змінні, значення яких ми можемо отримати, повинні реалізовувати тип *GettableStateVar*[4].

2.1.3. Core Profile ma Immediate mode

В ранніх версіях OpenGL програмісти не мали іншого вибору, тільки як малювати в режимі Immediate, який надавав у доступ лише готові, легкі в використанні функції. Вся логіка була прихована всередині бібліотеки і програмісти не мали змоги впливати на ключові процеси рендерингу. Всі розрахунки робились «під капотом». Логічно припустити, що такі функції були надзвичайно неефективними. Тому з часом програмістам стало їх недостатньо і виникла потреба у більшій гнучкості OpenGL.

Починаючи з версій 3.x OpenGL, Immediate mode почав вважатись застарілим і заохочувалось використання програмістами Core Profile, який не містить функцій Immediate mode, натомість надаючи більш гнучкі альтернативи. Звичайно, Core Profile набагато складніший для вивчення, проте в якості винагороди ми отримуємо гнучкі та ефективні програми. Найбільш яскравою ознакою Core Profile є те, що для роботи в цьому режимі, нам потрібно визначити два шейдери: вершинний та фрагментний. Детальніше про те, що таке шейдери та як їх використовувати, можна знайти в наступний розділах.

Варто зазначити, що Immediate mode все ще можна використовувати, хоча це й не рекомендовано. Ця робота здебільшого орієнтована на використання Core Profile, проте будуть приклади і з Immediate mode.

2.1.4. Подвійна буферизація та анімація

Ми згадували про те, що OpenGL надає нам можливість малювати різного виду об’єкти, проте ми не говорили про те, що у нас також є можливість створювати анімаційні ефекти. Часто в 3D моделюванні потрібно переміщуватись по сцені,

повертати об’єкти, емулювати рухи тощо. Це досягається за допомогою нескінченного циклу, на кожній ітерації якого викликається функція рендерингу. Часто в реалізації таких функцій допомагають GUI бібліотеки: так в GLUT можна визначити функцію idle, яка викликатиметься кожного разу, коли наш додаток перебуває в режимі очікування:

- *C/C++*: glutIdleFunc(*func)
- *Haskell*: idleCallback $= Just func

В ній та інших подібних функціях ми можемо використовувати поточний час (який також можна отримати через API GUI бібліотек) для того, аби задавати значення змінних, які постійно змінюються (наприклад, положення падаючої кульки). До речі, зверніть увагу, що тут використовується оператор $= (*дивись розділ 2.1.2.1*), адже ми змінюємо значення змінної стану.

Однією з найважливіших особливостей будь-якої графічної бібліотеки була і залишається підтримка подвійної буферизації, яка дозволяє виконувати операції рендерингу, візуалізуючи так званий закадровий буфер. Далі за допомогою команди заміни, малюнок моментально виводиться на екран[2].

Подвійна буферизація корисна у двох випадках: коли ми відображаємо складні об’єкти, рендеринг яких потребує багато часу та під час створення анімацій. В першому випадку, можливо ми не хочемо бачити кожен крок побудови зображення і за допомогою подвійної буферизації можна сформувати зображення і відобразити його лише після завершення побудови. Тобто користувач побачить зображення тільки після того, як воно повністю сформується. В другому, кожен кадр формується в закадровому буфері, і коли він буде готовий, швидко переключається на екран[2]. Це запобігає «мерехтінню» зображення під час анімації.

І GLUT і GLFW підтримують подвійну буферизацію. Для того, щоб включити підтримку подвійної буферизації в GLUT, необхідно зробити декілька кроків:

1. Додати в *initDisplayMode* значення Double Buffer:

initialDisplayMode $= [DoubleBuffered]

2. Наприкінці кожної ітерації вічного циклу викликати *swapBuffers*, яка також виконує операцію очистки буферу.

Вікна GLFW бібліотеки підтримують подвійну буферизацію по замовчуванню.

2.1.5. Проекція

2.1.5.1. Відсікання та поле перегляду

Усі об’єкти мають певні координати. Проте координати об’єкту у світі це не то саме, що й пікселі екрану, на якому цей об’єкт відображається. Як OpenGL переводить ці координати у піксельні координати? З якої сторони світу і під яким кутом ми спостерігаємо за сценою?

Для того, щоб вказати OpenGL як переводити вказані пари значень в пікселі, необхідно визначити область декартового простору, яку займає вікно. Ця область називається *областю відсікання*[2]. Область відсікання – це мінімальні та максимальні значення x та y, які належать вікну (*див рисунок 2.1*).

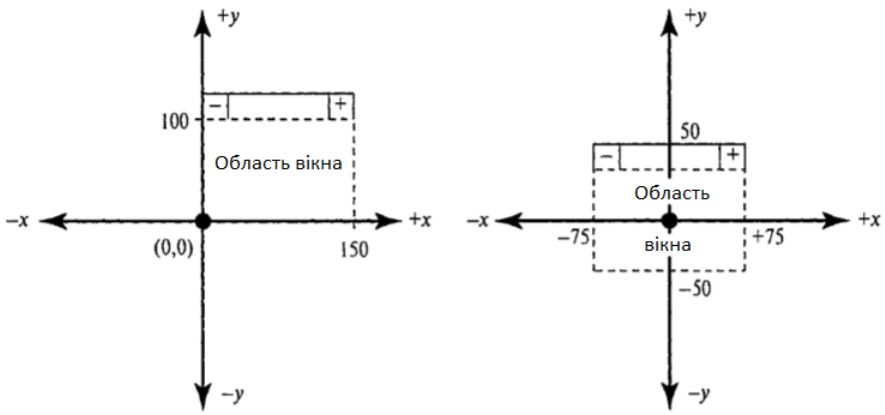

Рисунок 2.1[2]

Найчастіше розміри області відсікання не співпадають із розмірами вікна в пікселях. Тому необхідно задати правила відображення логічних координат у піксельні. Цей перехід задається за допомогою *поля перегляду* – області всередині вікна, яка використовується для відображення області відсікання (*див рисунок 2.2*). Зазвичай поле перегляду визначається на всю область вікна, але це не є обов'язковим правилом.

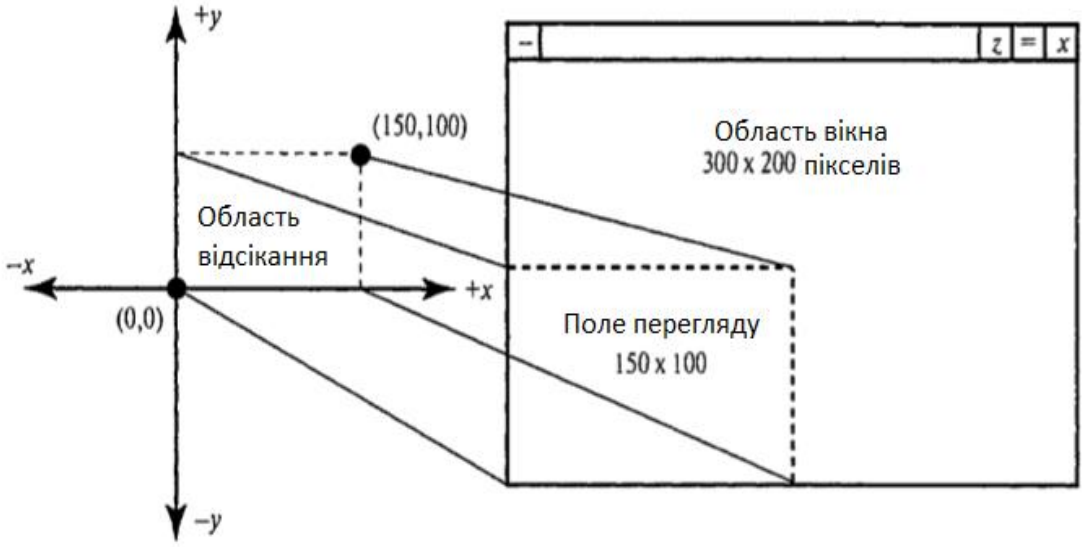

Рисунок 2.2[2]

2.1.5.2. Види проекцій

Проекція отримується в результаті «опускання» 3D координат об'єкта на двовимірну площину. Таким чином, проекція описує той об'єм, який ми можемо спостерігати з нашої точки зору.

В OpenGL є два основних види проекцій: ортографічна та перспективна. Ортографічну проекцію ще деколи називають паралельною, адже для неї ми задаємо прямокутний об'єм для спостерігання. Особливістю такої проекції є те, що всі об'єкти з однаковим розміром будуть в результаті мати однакову величину, незалежно від того, як далеко вони знаходяться[2] (*див рисунок 2.3*).

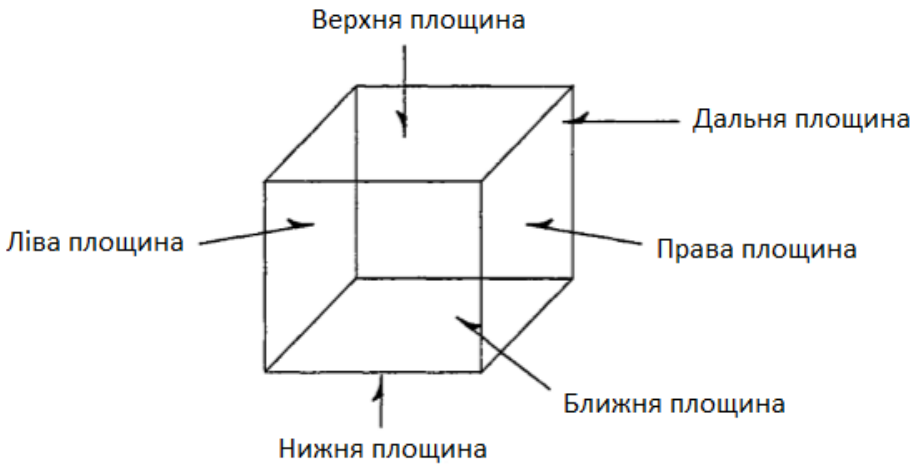

Рисунок 2.3[2]

Перспективна проекція додає ефект зменшення дальніх об’єктів. При цьому об’єм, який ми можемо спостерігати, по формі нагадує зрізану піраміду(*див рисунок 2.4*). Об’єкти, які знаходяться ближче до дальньої грані, стискаються при проектуванні на передню грань об’єму.

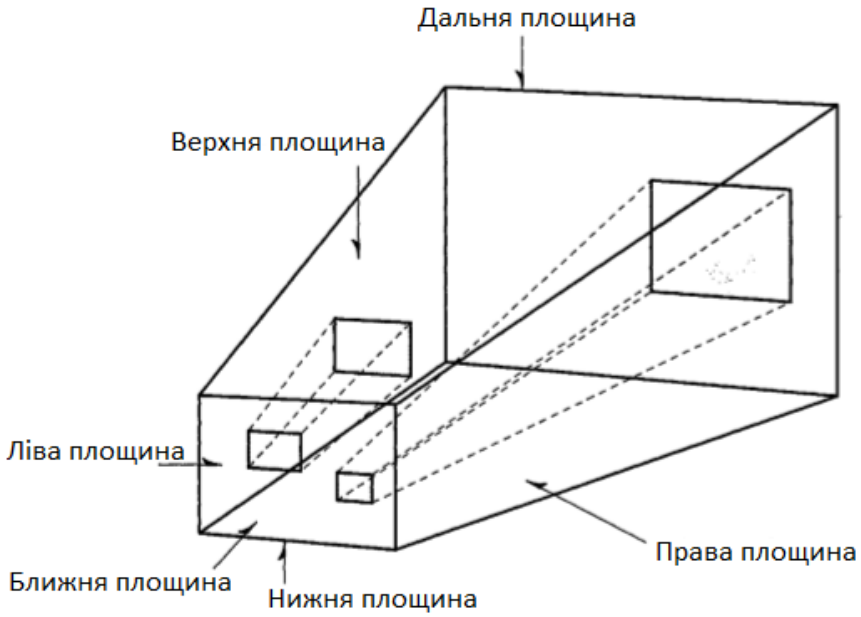

Рисунок 2.4[2]

Наприклад, на *рисунку 2.5* зображена одна і та ж сама фігура при різних проекціях (з використанням Immediate mode) . Зліва – перспективна проекція, справа – ортографічна.

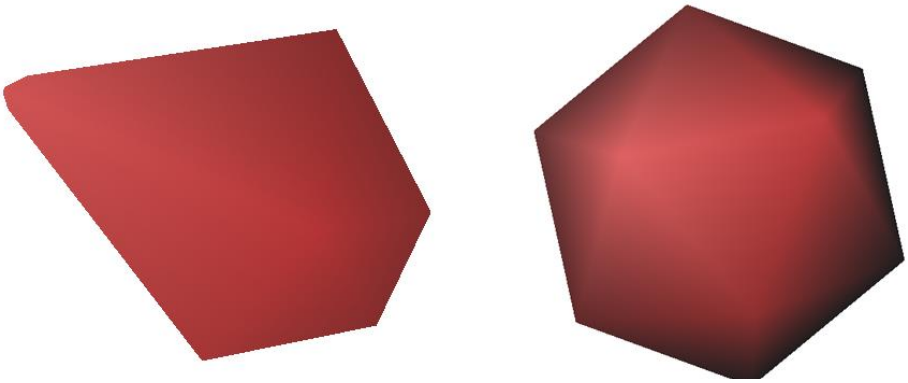

Рисунок 2.5

2.1.6. Шейдери

Процес перетворення 3D координат об’єкта у 2D координати реалізується графічним конвеєром OpenGL. Графічний конвеєр може бути розділений на дві великі частини: перша переводить 3D координати у 2D простір, а друга переводить отримані 2D координати у реальні зафарбовані пікселі[3].

Графічний конвеєр може бути поділений на декілька кроків і кожен наступний крок приймає на вхід результат роботи попереднього. Всі ці кроки вузькоспеціалізовані і виконують одну конкретну функцію. Саме тому їх легко розпаралелити. Тому сучасні графічні карти зазвичай складаються з тисяч маленьких ядер, які швидко обробляють дані всередині графічного конвеєра. Ці ядра запускають невеликі програми на GPU для кожного кроку конвеєра. Такі програми називаються *шейдерами*[3].

Деякі шейдери можуть бути перевизначені програмістом. Це дає нам більше контролю над цим, як OpenGL обробляє наші дані. Крім того, так як вони виконуються на GPU, це дозволяє нам зекономити цінні ресурси і час CPU. Шейдери визначаються використовуючи спеціальну мову програмування GLSL (OpenGL Shading Language), яка віддалено нагадує C[3].

Нижче на *рисунку 2.6* зображено всі кроки графічного конвеєра. Зауважимо, що синім відмічені ті кроки, які ми маємо змогу перевизначати.

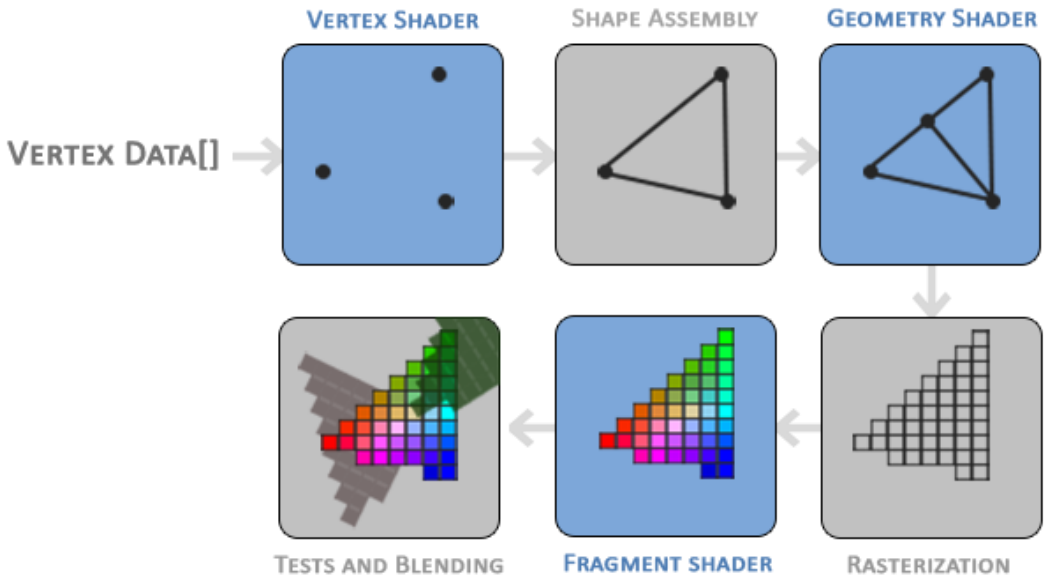

Рисунок 2.6[3]

На вхід графічному конвеєру ми передаємо список 3D координат, який на рисунку позначений як Vertex Data. Vertex не обов'язково має позначати набір з трьох значень – це може бути щозавгодно, що позначає одну вершину в нашій програмі. Сюди може входити колір, нормалі тощо.

Перша частина конвеєру – *vertex shader* – приймає на вхід одну вершину. Основна мета цього шейдеру застосувати певні перетворення до 3D координат. В результаті ми все ще матимемо 3D координати, але до яких уже застосовані потрібні нам розрахунки.

Другий крок – *shape assembly* – приймає на вхід стільки вершин з попереднього кроку, скільки треба для відображення потрібної фігури і збирає ці точки в задану примітивну фігуру. Як задаються типи фігур в OpenGL буде зазначено в наступних розділах.

Geometry shader приймає набір вершин, що формують фігуру, і має змогу створювати на їх основі нову фігуру, додаючи або видаляючи якісь точки. На рисунку він просто додає ще одну точку тим самим формуючи два трикутники.

Наступний крок – це *rasterization stage*, на якому відбувається процес співставлення точок фігури у відповідні пікселі екрану. В результаті формуються так звані фрагменти, які по факту є набором даних, що необхідні для відображення пікселю. Ці фрагменти слугують вхідними даними для *fragment shader*. Перед тим, як спрацює *fragment shader*, виконується ще одна операція – відсікання, яка видаляє всі фрагменти, які будуть невидимі для поточного поля зору, тим самим оптимізуючи програму.

Fragment shader обраховує фінальні кольори пікселя, часто виконуючи складні обрахунки. Зазвичай на цьому кроці стають можливими всі складні ефекти OpenGL. Часто цей шейдер містить чимало інформації про поточний стан сцени, яка може вплинути на результуючий колір: освітлення, тіні, туман, колір джерела світла тощо.

Останній крок - *alpha test and blending stage* – виконує перевірку відповідного значення глибини фрагменту і використовує це для перевірки чи результуючий фрагмент знаходиться спереду чи позаду інших об’єктів. Відповідно, якщо фрагмент знаходиться позаду, він буде відкинутий. Цей етап також виконує перевірку значень альфа (прозорість об’єкта) і змішує кольори об’єктів відповідно до цих значень.

В сучасних версіях OpenGL ми обов’язково повинні визначити хоча б *vertex shader* та *fragment shader*, адже для них не існує реалізацій по замовчуванню. А ось *geometry shader* можна залишити по замовчуванню, на практиці він рідко перевизначається.

3. ЗАСТОСОВУЮЧИ ТЕОРІЮ НА ПРАКТИЦІ

Тепер, маючи деякі теоретичні знання про те, як влаштований OpenGL, можна приступати до практичної частини. У цьому розділі ми будемо будувати сцену з двох об’єктів – ікосфери та квадрата, який буде виконувати роль джерела світла. Також у нас буде змога рухати камеру по сцені вперед/назад використовуючи клавіші WASD та повертати камеру за допомогою рухів мишки (*див рисунок 3.1*). Враховуючи розташування джерела світла, програма буде вираховувати потрібний колір грані.

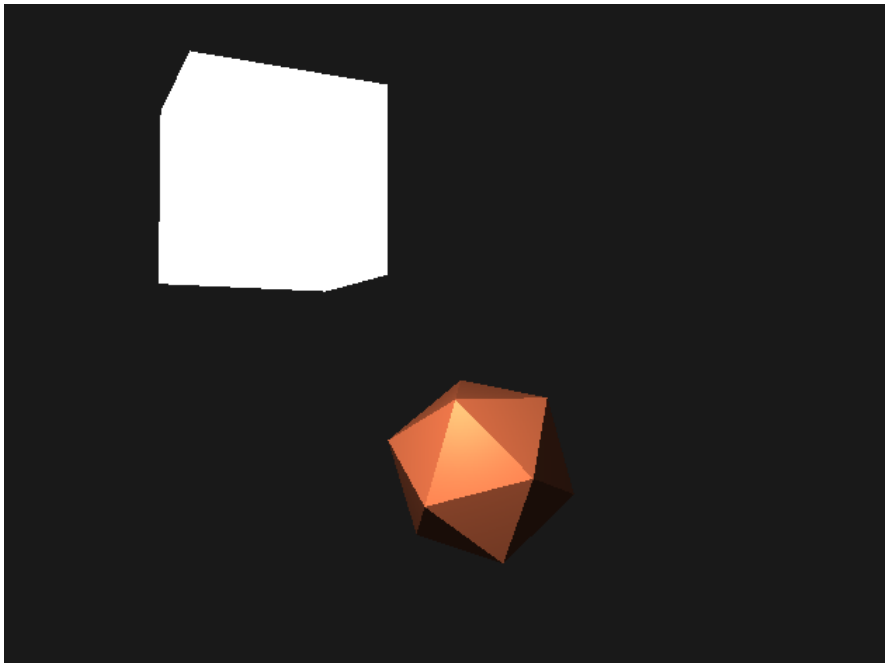

Рисунок 3.1

3.1. Крок 0. Налаштування середовища

Перед тим, як безпосередньо почати розробку, необхідно скачати допоміжні бібліотеки. Серед таких, наприклад, GLFW-b, Linear, OpenGL. Деякі модулі поставляються із Хаскель платформою, інші доведеться завантажити самостійно.

Скачуються бібліотеки за допомогою таких інструментів як cabal або stack. Наприклад, для встановлення GLFW-b пакету за допомогою cabal, необхідно виконати таку команду (звичайно, попередньо скачавши cabal):

cabal install GLFW-b

Аналогічно встановлюються й інші бібліотеки. Схожим чином замість cabal можна використати stack.

Програма розбита на окремі модулі, зібрати і запустити які разом, можна, запустивши відповідну команду:

ghc --make <шлях до кореневого модуля>

Коли GHC передається опція *--make*, він будує багатомодульну Хаскель програму, підвантажуючи відповідні залежності з кореневого модуля (*Main*), в результаті чого генерується .exe файл. Такий спосіб запуску програми є найпростішим.

Окрім цього, можна створити stack проект із відповідним package.yaml файлом, в якому вказуються усі залежності проекту. Будується і запускається проект командами *stack build* (який насправді теж використовує GHC) та *stack exec*. Детальніше про використання stack можна почитати в офіційній документації.

3.2.Крок 1: Задання фігури

В цьому розділі ми будемо малювати 3D зображення ікосфери (*рисунок 3.2*).

Ікосаедр є звичним многогранником з дванадцятьма вершинами та двадцятьма однаковими трикутниками (перша фігура на *рисунку 3.2*). Наступні фігури отримуються в результаті ділення кожного трикутника на чотири рівні підтрикутники.

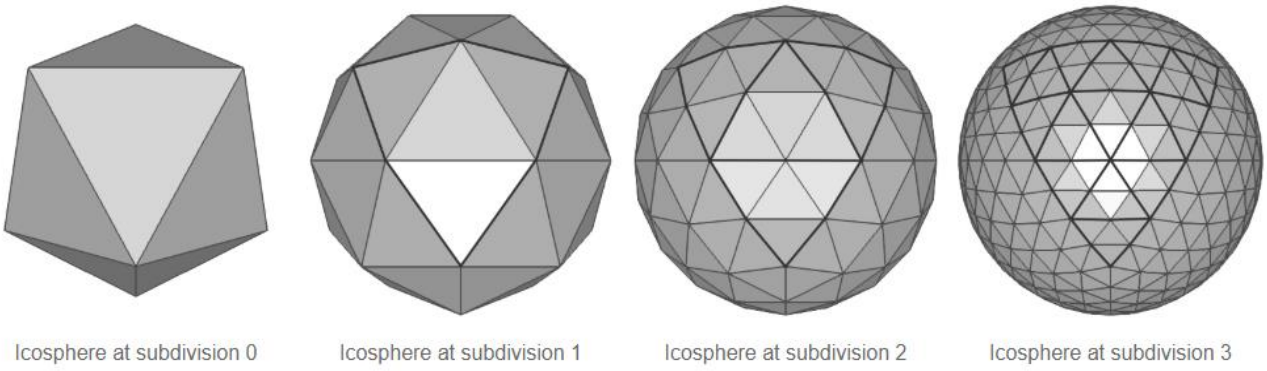

Рисунок 3.2[5]

Один зі способів, як можна задати ікосаедр, це використовуючи сферичні координати – дві вершини ми розташовуємо на півночі і півдні, а усі інші на широті $\pm \tan^{-1}(\frac{1}{2})$ градусів – по п’ять вершин на кожну широту. Точки на одній широті розташовуються через кожні 72 градуси (*рисунок 3.3*).

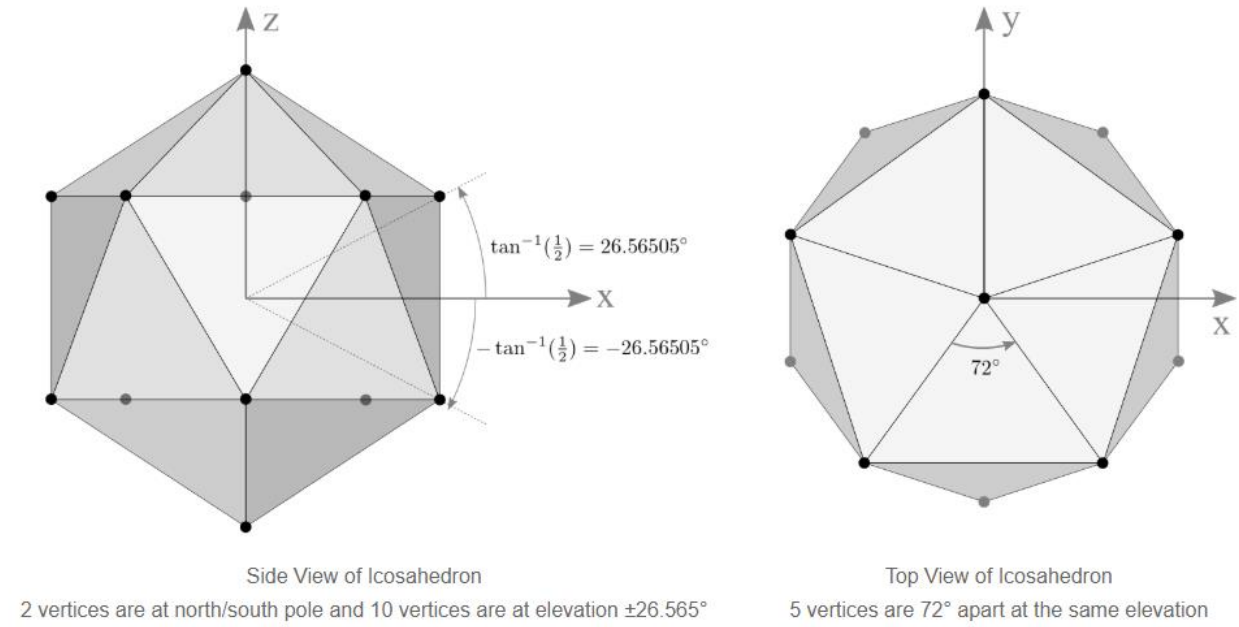

Рисунок 3.3 [5]

Маючи заданий радіус R, можна вирахувати координати точки, наприклад, на широті $\tan^{-1}(\frac{1}{2})$ (*формула 3.1-3.3*).

$$x = R * \cos(\tan^{-1}(\frac{1}{2})) * \cos(72 * n) \quad (3.1)$$

$$y = R * \cos(\tan^{-1}(\frac{1}{2})) * \sin(72 * n) \quad (3.2)$$

$$z = R * \sin(\tan^{-1}(\frac{1}{2})) \quad (3.3)$$

Ми не будемо зосереджуватись на тому, як реалізувати цей алгоритм на Хаскелі, лише зазначимо ключові моменти.

Для зручності створимо структуру даних Point, яка просто буде псевдонімом для кортежа з трьох дійсних значень:

type Point = (GLfloat, GLfloat, GLfloat)

Залежно від того, в якому режимі ми працюємо (*наведено в розділі 2.1.3*), OpenGL приймає точки в дещо різному вигляді. Якщо це Immediate mode, ми можемо використати функцію *renderPrimitive*, яка приймає на вхід тип фігури, яку ми хочемо відобразити за допомогою вершин, а також набір викликів до спеціальної функції *vertex*, яка додає передану їй точку на канвас. Маючи масив точок, можна визначити допоміжну функцію, яка до кожної точки застосує *vertex* (*лістинг 3.1*).

```
renderAs figure ps = renderPrimitive figure(makeVertx ps)

makeVertx :: [Point] -> IO ()
makeVertx = mapM_ (\(x,y,z)->vertex$Vertex3 x y z)
```

Лістинг 3.1

В Core Profile механізм передачі абсолютно інший, але про це згодом. Поки що зазначимо лише те, що в цьому режимі нам потрібно мати масив дійсних значень: *[Float]*, тому ми можемо визначити допоміжну функцію для переведення *[Point]* → *[Float]*(*лістинг 3.2*):

```
convertToFloarArr :: [Point] -> [Float]
convertToFloarArr = concat . map (\(x,y,z)-> [x, y, z])
```

Лістинг 3.2

На *рисунку 3.4* зображений результат реалізації ікосфери за допомогою засобів OpenGL в Immediate Mode.

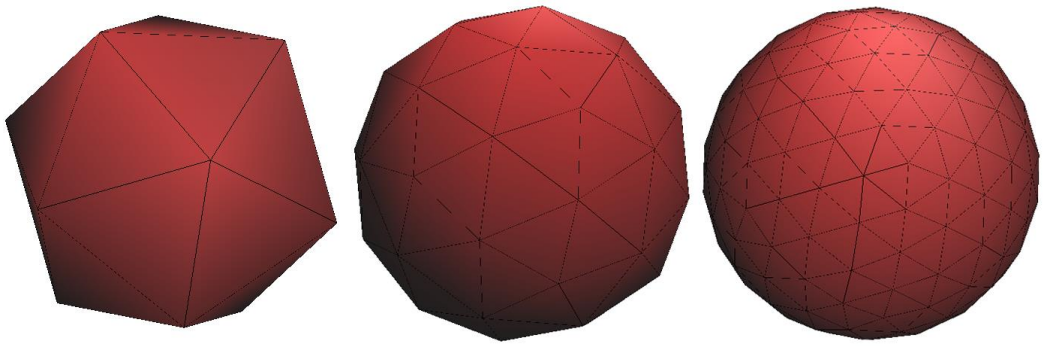

Рисунок 3.4

3.3. Крок 2: GUI

3.3.1. GLUT

Ми не будемо особливо зосереджуватись на цій бібліотеці, адже, як зазначалось, вона вважається застарілою. Розкажемо лише ро основні моменти. На *лістингу 3.3* проілюстроване базове використання функцій GLUT, на основі яких можна будувати складніші приклади.

```
main :: IO ()
main = do
  (_progName, _args) <- getArgsAndInitialize
  _window <- createWindow "Hello GLUT"
  displayCallback $= display
  mainLoop

display :: DisplayCallback
display = do
  clear [ ColorBuffer ]
  flush
```

Лістинг 3.3

Все, що робить цей код, це відкриває вікно з назвою «Hello GLUT!». Перша стрічка в main функції *getArgsAndInitialize* ініціалізує OpenGL систему. Потім створюється вікно за дпомогою функції *createWindow*. Далі визначається *displayCallback*, який є змінною стану і викликається кожного разу, коли створюється вікно. *displayCallback* відповідає за відображення, тому всі функції, які ми писатимемо для малювання фігур, будуть визначатись там. В цьому

прикладі функція *display* ще нічого значного не робить, лише очищує вікно від будь-якого кольору (буфер кольорів) і встановлює колір по замовчуванню (чорний або білий).

OpenGL теорія

> Буфер – це область зберігання інформації про зображення. Червоний, зелений та синій компоненти малюнку зазвичай об’єднуються під загальною назвою буфера кольорів або буфера пікселів[2].

Завжди, коли ми визначаємо якусь послідовність монадичних функцій в *display*, в кінці неї ми маємо зробити виклик *flush*. Виклик цієї функції забезпечує, що всі зазначені вирази остаточно були передані графічній системі, за допомогою якої відбувається відображення.

Функція *mainLoop* контролює всі деталі, пов’язані із взаємодією з операційною системою, оновлює вікно, забезпечує наступний виклик *displayCallback* для відображення графіки.

API GLUT цим не обмежується, в користування він надає чимало корисних функцій та змінних, серед яких:

- *windowSize* - задає розміри вікна.
- *idleCallback*, про який уже згадувалось і який викликається кожного разу, коли наша програма простоює.
- *reshapeCallback* – викликається при зміні розмірів вікна. Корисний для задання проекції (*проекції описані в розділі 2.1.5*). Він також викликається одразу після першого відображення вікна.
- *addTimerCallback* – задає функцію, яка буде виконана через певний проміжок часу.
- *positionCallback* – викликається кожного разу, коли змінюється положення вікна.
- *keyboardCallback* – викликається при натисканні на клавішу.

- *mouseCallback* – викликається при русі мишки або при натисканні на якусь із її клавіш.

Повний список функцій можна знайти в документації GLUT: https://hackage.haskell.org/package/GLUT.

3.3.2. GLFW

На *лістингу 3.4* можна знайти простий приклад ініціалізації GLFW вікна.

```
main :: IO ()
main = do
    G.setErrorCallback (Just errorCallback)
    successfulInit <- G.init
    G.windowHint (G.WindowHint'OpenGLProfile G.OpenGLProfile'Core)
    if' successfulInit onSuccessInit exitFailure
```

Лістинг 3.4

Більшість подій обробляються через колбеки: при натисканні клавіші, руху мишки або ж при виникненні помилки. Якщо в якійсь GLFW функції виникає помилка, її можна перехопити, визначивши *errorCallback*.

Перед тим, як використовувати GLFW, він має бути проініціалізований. Саме це і робить функція *init*, яка повертає булеве значення: true, якщо ініціалізація пройшла успішно і відповідно false в протилежному випадку. Після цього ми можемо налаштувати GLFW як нам потрібно, визначаючи *windowHints*. В цьому прикладі ми вказуємо GLFW, що збираємось використовувати Core Profile. Крім цього, ми можемо задати й інші параметри, наприклад, версію OpenGL. Далі робиться перевірка, чи ініціалізація пройшла успішно. Якщо так, то викликається допоміжна функція *onSuccessfulInit*, в якій ми уже визначаємо необхідні колбеки і задаємо вікно. Ми не будемо наводити увесь код функції *onSuccessfulInit*, лише наголосимо на ключових моментах.

Задати вікно можна як показано на *лістингу 3.5*.

```
mw <- G.createWindow 640 480 "Icosphere example" Nothing Nothing
```

Лістинг 3.5

Ми створюємо вікно розміром 640x480 пікселів з назвою «Icosphere example» і записуємо його у змінну *mw*. Ця функція поверне монадичне значення Maybe. Відповідно, якщо результат виконання Nothing, створення вікна не пройшло успішно. Перевірити це ми можемо як в *лістингу 3.6*.

```
maybe' mw (G.terminate >> exitFailure) $ \window -> do
```

Лістинг 3.6

maybe' – це допоміжна функція, яку ми визначили для зручності. Вона приймає 3 аргументи: монаду Maybe, функцію, яка має виконатись, якщо значення монади Nothing і функцію, яка виконається, якщо значення Just. Відповідно при невдалій ініціалізації ви викликаємо *terminate*, яка очистить всі ресурси GLFW, які ми виділили під час ініціалізації і завершить виконання програми. В протилежному випадку ми дістаємо «чисте» значення з монади і використовуємо його для подальших дій. Всі наступні операції визначаються в межах цього *do*.

Перш ніж ми зможемо викликати функції OpenGL, необхідно визначити поточний OpenGL контекст: *G.makeContextCurrent mw*. На відміну від GLUT, GLFW не надає функції типу *mainLoop*, яка зробить всю технічну роботу за нас. Тому ми маємо явно визначити рекурсивну функцію *mainLoop*, яка буде відповідати за виклик OpenGL функцій для відображення об'єктів на екрані. Після *mainLoop* важливо очистити ресурси і викликати функції *destroyWindow* та *terminate*.

На цьому завершуються основні налаштування середовища вікна і можна перейти до малювання.

GLFW, так само як і GLUT, дозволяє перевизначати колбеки для різних подій, проте зараз ми не будемо на них зосереджуватись.

Увесь наступний код робиться з розрахунку, що використовується GLFW.

3.4. Крок 3: Сцена та об'єкти

Подальші приклади і пояснення стосуватимуться Core Profile режиму.

3.4.1. Шейдери та Haskell

Ми не будемо детально зосереджуватись на синтаксисі, який використовують шейдери, лише зазначимо основні моменти. Найпростіший vertex шейдер виглядатиме як зображено на *лістингу 3.7*.

Спочатку вказується версія OpenGL (в даному випадку це «3.3»). Далі визначається вхідний vertex атрибут, якому вказується позиція (*location = 0*), тип (*vec3* – масив з трьох елементів) та назва (*aPos*).

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Лістинг 3.7[3]

Щоб визначити вихідний результат vertex shader, необхідно записати відповідне значення у визначену наперед змінну *gl_Position*, тип якої завжди *vec4*. Так як вхідні дані мають тип *vec3*, ми приводимо їх до типу *vec4*.

Для зручності шейдери визначаються в окремому файлі і тепер нам необхідно їх завантажити в програму і вказати OpenGL їх використовувати.

Програма шейдерів – це об'єкт, який по факту є послідовністю зв'язаних між собою шейдерів. Щоб використовувати шейдери в програмі, необхідно прив'язати їх до програми і потім активувати цю програму кожного разу, коли ми хочемо використати цей набір шейдерів (*лістинг 3.8*).

```
program <- loadShaders [
    ShaderInfo VertexShader (FileSource "shaders/triangles.vert"),
    ShaderInfo FragmentShader (FileSource "shaders/triangles.frac")]
currentProgram $= Just program
```

Лістинг 3.8[6]

Тут використовується допоміжна функція *loadShaders*, яку ми розглянемо трохи пізніше. Вона приймає на вхід масив об'єктів типу *ShaderInfo* (*лістинг 3.9*), який містить інформацію про тип шейдера і розташування відповідного шейдера.

```
data ShaderInfo = ShaderInfo ShaderType ShaderSource
   deriving ( Eq, Ord, Show )
```

Лістинг 3.9[6]

Далі ми активуємо цю зв'язку шейдерів, перевизначаючи глобальну змінну *currentProgram*.

Функція *loadShaders* зображена на *лістингу 3.10*.

```
loadShaders :: [ShaderInfo] -> IO Program
loadShaders infos =
   createProgram `bracketOnError` deleteObjectName $ \program -> do
      loadCompileAttach program infos
      linkAndCheck program
      return program
```

Лістинг 3.10[6]

Тут створюється програма шейдерів через функцію OpenGL *createProgram*. *bracketOnError* – це допоміжна функція з модуля *Control.Exception*, яка викличе *deleteObjectName* у випадку, якщо наступний аргумент – функція – завершиться аварійно. *loadCompileAttach* та *linkAndCheck* - це ще одні допоміжні функції, на деталях яких ми не будемо зупинятись, лише зазначимо ключові OpenGL, які вони викликають:

- createShader – створює об'єкт шейдера.
- shaderSourceBS – встановлює джерело шейдера для даного шейдер об'єкта. В нашому випадку ми маємо зчитати відповідні файли (*readFile* з модуля *Data.ByteString*), перевести їх у формат UTF8 і передати в цю

функцію. Зробити це можна через функцію *encodeUtf8* з модуля *Data.Text.Encoding*. Усі ці дії зображені на *лістингу 3.11*.

- compileShader – компілює відповідний шейдер
- attachShader – прив'язує відповідний шейдер до вказаної програми.
- linkProgram – зв'язує всі вказані для цієї програми шейдери в одну фінальну шейдер програму.

```
getSource (FileSource path) = B.readFile path

packUtf8C :: String -> B.ByteString
packUtf8C = TE.encodeUtf8 . T.pack
```

Лістинг 3.11[6]

Вказані функції викликаються в тому порядку, в якому зазначені тут. Повний код можна знайти в практиці.

Важливо згадати про ще один аспект, який знадобиться нам в *розділі 3.3.5* - уніформи. Уніформи – це ще один спосіб передачі даних з програми, яка працює на CPU до шейдера на GPU. Найголовніше, що відрізняє їх від атрибутів, це те, що вони глобальні, тобто доступні в будь-якому шейдері.

Визначити уніформу в шейдері можна таким чином:

uniform mat4 view;

Спочатку вказується спеціальне слово *uniform*, потім тим і назва.

Щоб записати в цю змінну значення, в програмі ми повинні зробити декілька дій:

- задати поточну програму шейдерів через *currentProgram*
- отримати відповідну змінну через *uniformLocation*, що повертає відповідну глобальну змінну. Значення з неї ми можемо дістати за допомогою функції get.
- Встановити нове значення цій змінній через функцію *uniform*.

Приклад використання продемонстрований на *лістингу 3.12*.

```
currentProgram $= Just program
viewLocation <- get $ uniformLocation program "view"
uniform viewLocation $= (viewMatrix)
```

Лістинг 3.12

Де «view» - назва уніформи в шейдері.

3.4.2. VBO

Це напевно одна з найскладніших тем для тих, хто лише починає вивчати OpenGL, адже вона передбачає засвоєння великої кількості неочевидної інформації.

Повертаючись до теми шейдерів (*наведено в розділ 2.1.6*), щоб відобразити щось на екрані, ми повинні передати у *vertex shader* набір точок. Для цього необхідно створити місце в пам'яті GPU, де ми будемо зберігати точки, налаштувати OpenGL, щоб він міг правильно зчитувати інформацію з цієї пам'яті і визначити, як відправляти дані до графічної карти.

Управляти цією пам'яттю ми можемо через VBO, що розшифровується як *vertex buffer object*, який може зберігати велику кількість точок в пам'яті GPU. Завдяки таким об'єктам ми можемо передати до GPU велику кількість даних один раз замість того, щоб передавати по одній вершині. Це економить досить багато часу, адже передача даних до GPU відбувається дуже повільно.

Створити VBO ми можемо за допомогою функції *genObjectName (лістинг 3.13)*. OpenGL містить чималий набір типів для буферних об'єктів. Тип VBO - це *Array*

```
let numVertices = length vertices
vertexBuffer <- genObjectName
bindBuffer ArrayBuffer $= Just vertexBuffer

withArray vertices $ \ptr -> do
    let size = fromIntegral (numVertices * sizeOf (head vertices))
    bufferData ArrayBuffer $= (size, ptr, StaticDraw)
```

Лістинг 3.13

Buffer. OpenGL зв'язувати об'єкти з декількома буферами, якщо вони різних типів. Щоб зв'язати об'єкт з буфером, треба скористатись змінною стану *bindBuffer* (*лістинг 3.13*). Далі необхідно скопіювати масив точки у цей буфер (використовуючи *bufferData*).

При виклику *bufferData* вказується тип буферу, далі передається розмір масиву точок в байтах, вказівник на початок масиву та спеціальна змінна OpenGL, яка визначає, як ми хочемо, щоби графічна карта працювала з нашими даними. Значення StaticDraw означає, що дані встановлюються лише один раз і використовуються багато разів. Так як ми відображатимемо нерухомий об'єкт, нам підходить StaticDraw. Якби об'єкт часто змінювався, ми могли б використати DynamicDraw.

В цьому прикладі також використовується допоміжна функція *withArray* з бібліотеки *Foreign.Marshal*, за допомогою якого ми можемо виконувати маршалинг. В даному випадку нам треба зберегти в пам'яті масив.

Ще одна допоміжна функція – це *sizeOf* з бібліотеки *Foreign.Storable*, яка обраховує пам'ять, яку займає аргумент, в байтах.

Припустимо, що наш масив точок виглядає приблизно таким чином, як зображено на *рисунку 3.5*. Так як ми можемо задати будь-який тип даних на вхід vertex shader, необхідно вказати OpenGL, як інтерпретувати масив вершин, щоб vertex shader міг його зчитати. Для цього потрібно використати функцію *vertexAttribPointer (лістинг 3.14).*

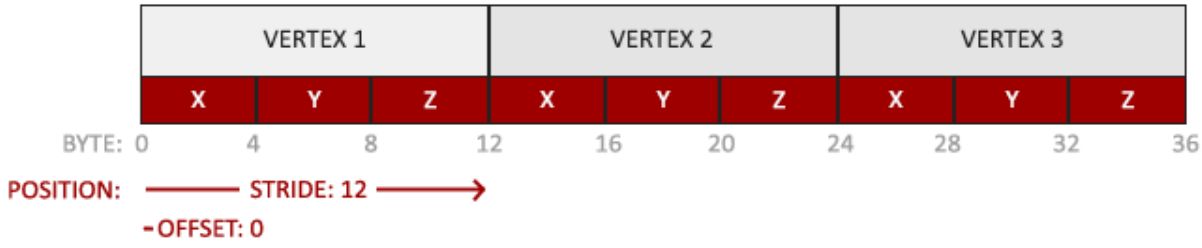

Рисунок 3.5[3]

```
let sizeF = fromIntegral (6 * sizeOf (head vertices))

let firstIndex = 0
    vPosition = AttribLocation 0
vertexAttribPointer vPosition $=
    (ToFloat, VertexArrayDescriptor 3 Float sizeF (bufferOffset firstIndex))
vertexAttribArray vPosition $= Enabled
```

Лістинг 3.14

В *vertexAttribPointer* вказується доволі багато аргументів:

- vPosition – визначає, який vertex атрибут ми хочемо визначити. Ці атрибути визначаються в vertex shader і фактично означають вхідні дані для vertex шейдера. Кожен vertex атрибут має унікальне ID.
- ToFloat – значення типу IntegerHandling.
- VertexArrayDescriptor з набором своїх параметрів:
    - 3 – задає розмір vertex атрибута. Так як даний атрибут має тип vec3, ми вказуємо число «3».
    - Float – тип даних, з яким ми працюємо.
    - sizeF – це крок, який вказує на розмір простору між двома атрибутами. Як видно з коду, кожного разу ми «переступаємо» через шість чисел. Це потрібне для враховування кольору і детальніше розглянеться в *розділі 3.4.4*)
    - Останній параметр – це зсув, який позначає, звідки починаються дані в буфері. В нашому випадку це 0. *bufferOffset* – допоміжна функція.

Після того, як ми визначили правила переходу від даних до vertex атрибута, ми повинні цей атрибут активувати. Робиться це через функцію *vertexAttribArray*, якій передається позиція атрибута та встановлюється значення Enabled.

Після цього ми можемо викликати функції для відображення наших вершин. Проте кожного разу, коли ми захочемо їх зобразити, доведеться робити всі вище наведені операції. Якщо у нас багато різних об’єктів, це може перетворитись в доволі громіздкий процес.

3.4.3. VAO

Полегшити створення об’єктів може допомогти VAO (*див рисунок 3.6*). VAO – це *vertex array object*. Вони створюються і зв’язуються по аналогії з VBO, і зберігають всі налаштування vertex атрибутів, які були зроблені після їх

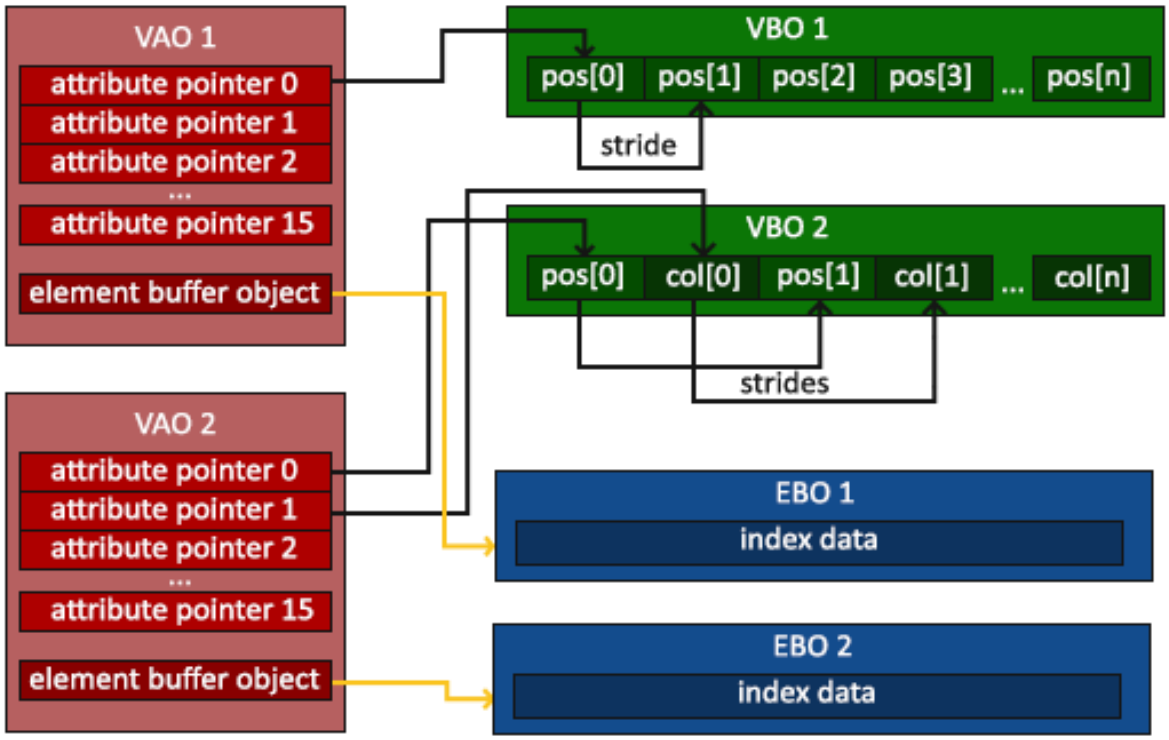

Рисунок 3.6[3]

ініціалізації. Це означає, що завжди, коли нам необхідно щось намалювати, достатньо просто викликати відповідний VAO.

Лістинг 3.15 показує, як можна створити VAO. Аналогічно до VBO, спочатку ми створюємо об’єкт за допомогою функції *genObjectName* і потім зв’язуємо його через функцію *bindVertexArrayObject*.

```
triangles <- genObjectName
bindVertexArrayObject $= Just triangles
```

Лістинг 3.15

Тепер кожного разу, коли ми захочемо відобразити цей набір вершин, ми можемо зв’язувати перед малюванням triangles.

Визначимо допоміжну функцію для малювання об’єктів (*лістинг 3.16*)

```
renderObjects :: VertexArrayObject -> NumArrayIndices -> Program -> IO()
renderObjects vao nVerts program =  do
    currentProgram $= Just program

    bindVertexArrayObject $= Just vao
    drawArrays Triangles 0 nVerts
```

Лістинг 3.16

На вхід вона приймає три аргументи - VAO, кількість вершин, які ми хочемо відобразити з нашого масиву, та програму, до якої при’язані шейдери. Спочатку потрібно активувати шейдери, тому тут перевизначається глобальна змінна стану *currentProgram*. Потім активується відповідний VAO і викликається функція *drawArrays*. Вона приймає 3 аргументи:

- Triangles – значення, яке позначає примітив, який ми хочемо відобразити. В нашому випадку на основі переданих вершин ми завжди відображаємо трикутники.
- 0 – початковий індекс масиву вершин, з якого ми хочемо почати малювання
- nVerts – відповідно кількість вершин

Ми можемо викликати цю функцію кожного разу коли необхідно щось намалювати.

3.4.4. Освітлення

Однією з реалістичних моделей оствілення є модель відображення Фонга, яка складається з трьох частин(*див рисунок 3.7*):

- Навколишнє освітлення (Ambient lighting) – симулює світло навколишнього середовища. Навіть якщо темно і джерело світла

«вимкнене», все одно ми можемо розрізняти об’єкти, адже невелика кількість світла завжди присутня у світі.

- Дифузне освітлення (Diffuse lighting) – симулює вплив, який джерело світла має на об’єкт, враховуючи напрямок. Чим більше площина повернена до світла, тим вона яскравіша.
- Дзеркальне освітлення (Specular lighting) – симулює яскраву точку світла, яка з’являється на блискучих об’єктах.

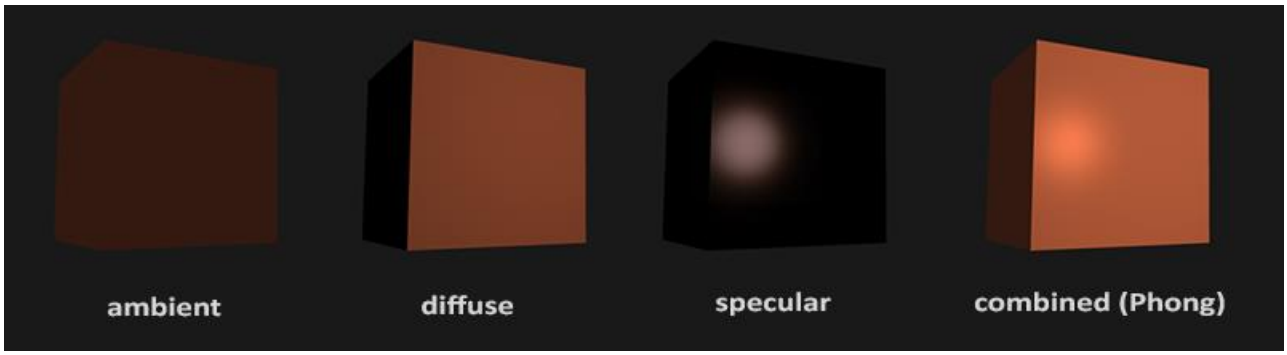

Рисунок 3.7[3]

Основна логіка, яка реалізує подібне освітлення, виконується в fragment шейдері, тому ми не будемо на ній зосереджуватись.

Щоби реалізувати дифузне освітлення, необхідно знайти нормалі кожної вершини та позицію джерела світла (*див рисунок 3.8*).

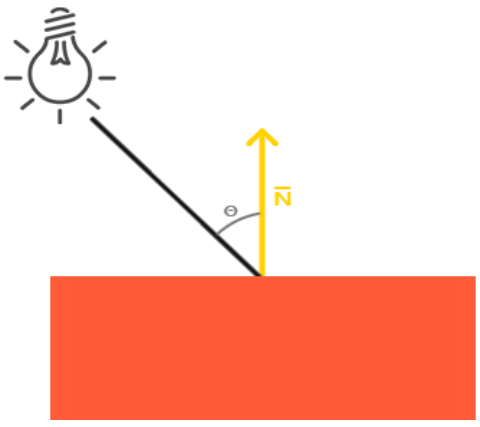

Рисунок 3.8[3]

Джерело світла – це звичайний куб білого кольору, для якого ми задамо свій список вершин та свої шейдери. По аналогії із будь-яким іншим об’єктом, ми

створюємо VAO і шейдерну програму, які викликаємо кожен раз, коли хочемо відобразити джерело світла.

Нормаль – це вектор, перпендикулярний до площини вершини. Так як вершина не має поверхні, ми вирахуємо нормаль, використовуючи навколишні вершини, щоб визначити, якій площині належить вершина. Ми можемо створити функцію, яка буде до кожної точки додавати додаткові 3 дійсні значення, які описують нормаль цієї точки (*лістинг 3.17*).

```
addNormals :: [Point] -> [NormalsPoint]
addNormals [] = []
addNormals verts@(a:b:c:ps) = do
    let res = addNormals ps
    let aVec = pointToV3 a
    let bVec = pointToV3 b
    let cVec = pointToV3 c
    let diff1 = bVec ^-^ aVec
    let diff2 = cVec ^-^ bVec
    let normal = L.cross diff1 diff2
    let normal' = normal ^/ (L.quadrance normal)
    (([appendPoints aVec normal'] ++ [appendPoints bVec normal']) ++ [appendPoints cVec normal']) ++ res
```

Лістинг 3.17

Також в vertex shader ми маємо визначити ще один атрибут, який буде описувати нормаль (*лістинг 3.18*)

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
...
```

Лістинг 3.18[3]

Цю нормаль ми потім передамо до fragment shader, де вона уже буде враховуватись при розрахунках кольору.

Так як одна точка містить не три значення, а шість, то крок при заданні *vertexAttribPointer* (*лістинг 3.14*) відповідно також «6». Також, передаючи

```
let nFirstIndex = 3 * sizeOf (head vertices)
    nPosition = AttribLocation 1
vertexAttribPointer nPosition $=
    (ToFloat, VertexArrayDescriptor 3 Float sizeF (bufferOffset nFirstIndex))
vertexAttribArray nPosition $= Enabled
```

Лістинг 3.19

нормалі до шейдера, ми вказуємо початковий індекс «3», а розташування атрибуту «1» (так, як ми визначили в шейдері) (*лістинг 3.19*).

На *рисунку 3.9* проілюстрований результат реалізації подібного освітлення. Як видно з рисунку, пляма дзеркального світла змінює положення і розмір залежно від кута зору.

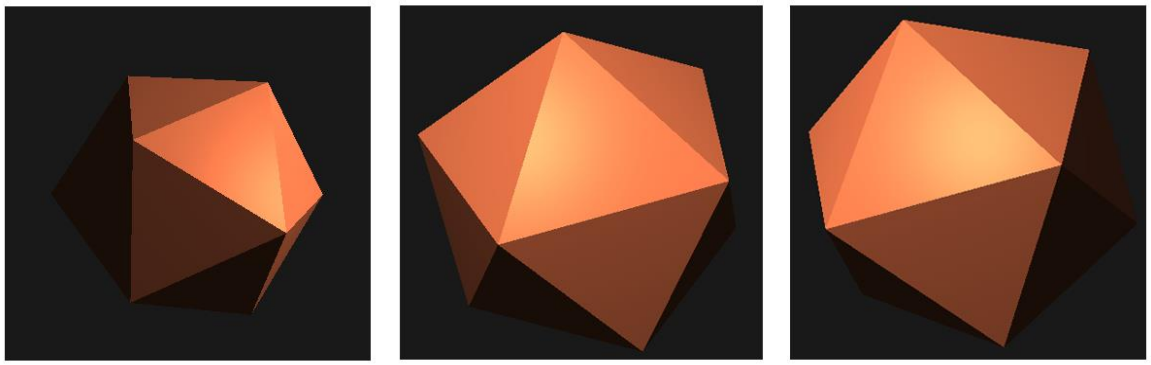

Рисунок 3.9

3.4.5. Камера

Тепер, коли ми можемо задати об’єкти на сцені, нам потрібна камера, яка дозволятиме нам спостерігати за сценою з різних позицій.

Як відомо, будь-яке перетворення можна задати через відповідну матрицю. Ми не будемо зосереджуватись на цій темі, адже вона має бути відносно відома кожному. Варто зазначити, що рух камери це поняття відносне, адже по факту такого об’єкта на нашій сцені немає. Ми просто рухаємо відповідні існуючи

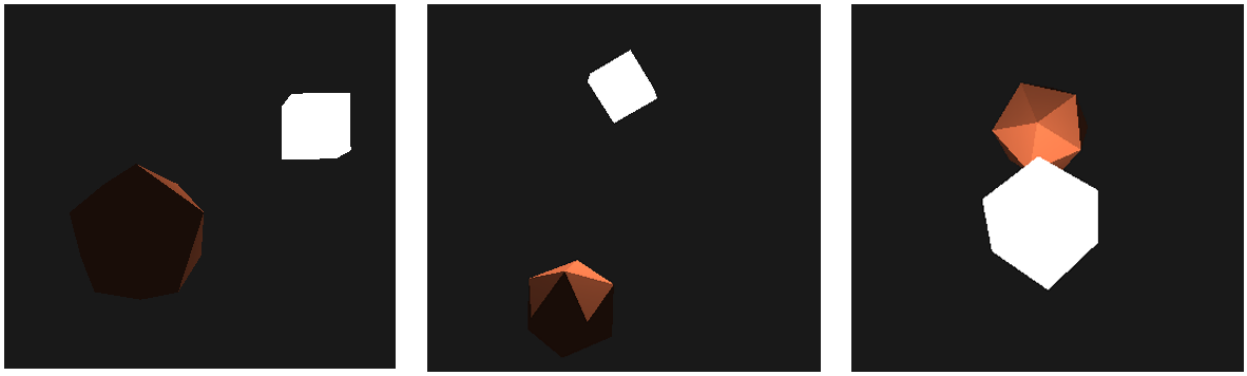

Рисунок 3.10

об’єкти, створюючи ефект наближення/віддалення. На *рисунку 3.10* зображена одна і та ж сцена з різних точок зору.

Для того, щоб ефективно задати положення камери, нам потрібно декілька значень: позиція камер в межах світу, напрямок, в якому вона дивиться, вектор, який вказує направо від камери і вектор, який вказує вгору (*див рисунок 3.11*).

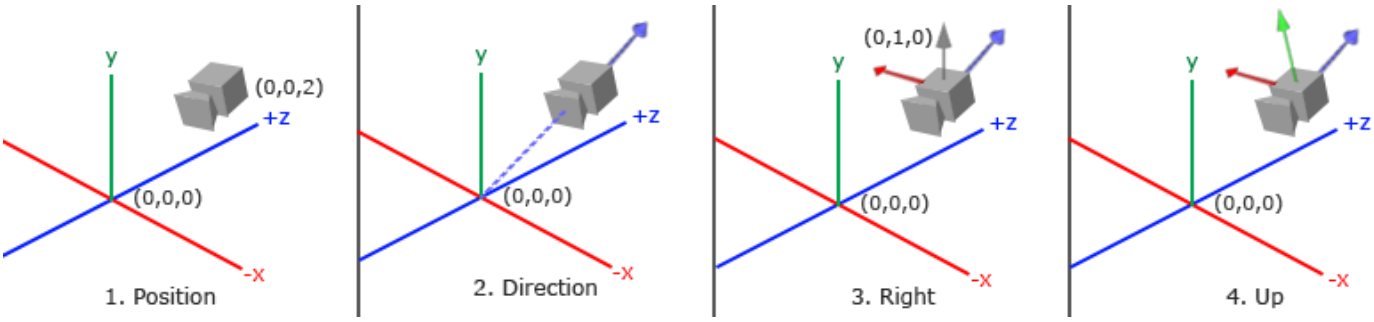

Рисунок 3.11[3]

Задати камеру в Хаскелі ми можемо як зображено на *лістингу 3.20*.

```
data Camera = Camera {
    cameraPos :: V3 GLfloat,
    cameraFront :: V3 GLfloat,
    cameraUp :: V3 GLfloat
} deriving Show
```

Лістинг 3.20

Ми навмисно не задали тут напрямок, адже це буде враховано згодом в обрахунках.

Згадуючи лінійну алгебру, за допомогою матриць ми можемо задати тривимірний координатний простір з будь-яких трьох перпендикулярних векторів. Потім, помноживши будь-який вектор на утворену матрицю, ми можемо отримати цей вектор в межах заданого координатного простору. Для цього можна використати спеціальну матрицю, відому в лінійній алгебрі, *lookAt (див рисунок 3.12)*. Вона робить саме те, що означає її назва – створює матрицю, яка «дивиться» на задану ціль. Фактично за допомогою такої матриці ми можемо

ефективно переводити всі координати світу у простір перегляду, який ми визначили.

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

Рисунок 3.12[3]

На малюнку ми бачимо, що матриця містить вектор в напрямку «Право» (R), вектора в напрямку «Вгору» (U), напрямок, в якому дивиться камера (D) та позицію камери (P).

В хаскелі подіну матрицю ми можемо задати використовуючи бібліотеку *Linear*. У ній визначено функцію lookAt, яка приймає три аргументи: позицію камери, розташування цілі, та вектор «Вгору». Ми можемо визначити допоміжну функцію, яка створюватиме таку матрицю (*лістинг 3.21*).

```
lookAtMatrix :: Camera -> M44 GLfloat
lookAtMatrix (Camera pos front up) = L.lookAt pos (pos ^+^ front) up
```

Лістинг 3.21

Потім отриману матрицю ми можемо передати в шейдер і помножити її на позицію нашого об’єкта (*лістинг 3.22*).

```
setViewUniform ::  M44 GLfloat -> Program -> IO()
setViewUniform lookAtMat program = do
    viewMatrix <- newMatrix RowMajor $ toArray lookAtMat :: IO (GLmatrix GLfloat)

    currentProgram $= Just program
    viewLocation <- get $ uniformLocation program "view"
    uniform viewLocation $= (viewMatrix)
```

Лістинг 3.22

Функція приймає на вхід *lookAt* матрицю та шейдерну програму і встановлює відповідне значення уніформу в шейдерах (*див в розділі 3.3.1*). Перша стрічка в функції просто переводить матрицю у той вигляд, в якому її сприйматиме *uniform* (останній рядок).

3.4.6. Main Loop

Ми вже говорили про те, що основна логіка OpenGL будується на зміні деякого глобального стану. Усі програми шейдерів, VAO, розташування камери – усе це та інформація, яка має бути відома протягом усієї програми і яку ми повинні мати змогу змінювати. Для цього ми можемо визначити спеціальний об'єкт State, який зберігатиме всю необхідну нам інформацію (*лістинг 3.23*):

```
data Descriptor = Descriptor VertexArrayObject Program

data State = State {
    descriptors :: [Descriptor],
    camera :: Camera,
    nVerts :: NumArrayIndices,
    lastFrameTime :: Float,
    keyRefs :: IORef (Set G.Key),
    mouseRef :: IORef MouseInfo
}
```

Лістинг 3.23

- descriptors – список дескрипторів - об'єктів, які містять інформацію, необхідну для відображення певного набору вершин: VAO та шейдерну програму. В нашому випадку це дескриптори ікосфери та джерела світла.
- camera – камера
- nVerts – кількість вершин для відображення
- lastFrameTime – час попереднього фрейму. Поточний час можна отримати через функцію *getTime*.
- keyRefs – список клавіш, які зараз натиснуті (*детальніше про це в розділі 3.4.1*)
- mouseRef – інформація про поточне положення мишки (*детальніше в розділі 3.4.2*)

Перед початком безкінечного циклу ми можемо задати початкові значення стану і потім передавати цей стан від однієї ітерації *mainLoop* до іншої (*лістинг 3.24*).

```
let state = State {
    descriptors = [descriptor, descriptorLamp],
    camera = initCamera,
    nVerts = fromIntegral $ length verts,
    lastFrameTime = _lastFrameTime,
    keyRefs = initKeyRefs,
    mouseRef = initMouseRef
}

mainLoop window state
```

Лістинг 3.24

Основні події, які необхідно визначити в *mainLoop*:

- Встановити проекцію
- Очистити буфер кольорів, встановити колір вікна
- Відобразити ікосферу та джерело світла, використовуючи дескриптори, які ми передали в змінній state
    - На цьому крокі також необхідно передати уніформи в vertex шейдер, щоб розташувати об’єкт в правильному місці: положення кожного об’єкту та матрицю lookAt (використовуючи *setViewUniform*)
- Оновити стан, а саме положення камери, адже воно буде змінюватись, та час останнього фрейму (*лістинг 3.25*)
- Перемкнути буфер через команду *swapBuffers*
- Забезпечити реагування на події(потрібно для руху камери) через команду *pollEvents*, яка дістає події з черги і обробляє їх. Без неї анімація працювати не буде.
- Повторно викликати *mainLoop* з оновленим станом.

```
let newState = state{camera = newCamera, lastFrameTime = currTime}
```

Лістинг 3.25

3.4.7. Проекція

Ми уже говорили про проекції в *розділі 2.1.5*. Тепер застосуємо знання на практиці. Як уже згадувалось, так само як і в GLUT, в GLFW ми можемо задати

колбек, який буде виконуватись кожного разу, коли змінюється розмір вікна. Ми задамо його через *FramebufferSizeCallback*, який викликатиметься кожного разу, коли змінюється розмір буфера кадру (*лістинг 3.26*). Усі GLFW колбеки встановлюються перед викликом *mainLoop*.

```
G.setFramebufferSizeCallback window (Just $ setProjection state)
```

Лістинг 3.26

setProjection – це допоміжна функція, яку ми визначили для встановлення проекції (*лістинг 3.27*).

```
setProjection :: State -> G.FramebufferSizeCallback
setProjection state window width height = do
    let program = getProgramFromState state 0
    let programLamp = getProgramFromState state 1

    let ratio = fromIntegral width / fromIntegral height
    let projMat = L.perspective (toRadians 45) ratio 0.1 100.0

    projectionMatrix <- newMatrix RowMajor $ toArray projMat :: IO (GLmatrix GLfloat)

    projLocation <- get $ uniformLocation program "projection"
    uniform projLocation $= (projectionMatrix)

    projLocationLight <- get $ uniformLocation programLamp "projection"
    uniform projLocationLight $= (projectionMatrix)
```

Лістинг 3.27

В цьому прикладі ми використовуємо перспективну проекцію. Таку проекцію можна отримати також через матрицю, за допомогою якої ми можемо отримати ефект перспективи. Створити таку матрицю ми можемо використовуючи уже знайому бібліотеку *Linear*, яка містить функцію *perspective*.

Ця функція створює зрізаний конус видимого простору. Все, що опиняється поза конусом, буде обрізане. Перший параметр функції це – *fov (field of view)*, або поле зору. Воно визначає, наскільки великим є простір спостереження. Значення має бути в радіанах. Наступний параметр – співвідношення ширини та висоти вікна, або *aspect ratio*. Третій і четвертий параметри встановлюють ближню і дальню площини зрізаного конуса. Фактично отримана матриця задає конус, показаний на *рисунку 3.13*.

Отримавши потрібну матрицю, ми переводимо її в потрібний вигляд і передаємо в шейдер, де вона перемножується з розташуванням об’єкта. Таку дію ми повторюємо як для ікосфери, так і для джерела світла.

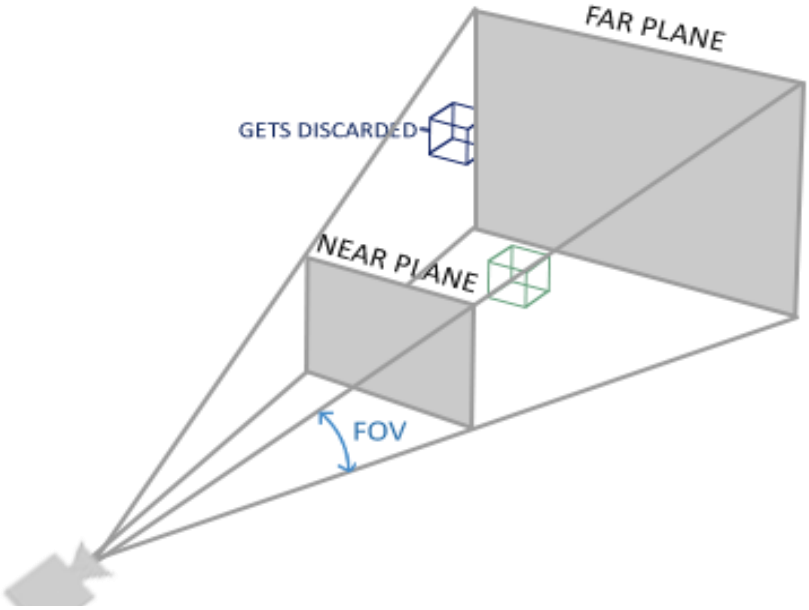

Рисунок 3.13[3]

3.5. Крок 4: Події

Ми відобразили ікосферу, джерело світла і задали положення камери. Тепер цю камеру потрібно «зрушити з місця».

3.5.1. Натискання клавіш

GLFW по аналогії з GLUT, дозволяє визначити колбек, який викликатиметься кожного разу при натисканні клавіші, за допомогою функції *setKeyCallback*.

Щоб мати змогу натискати декілька клавіш одночасно, ми будемо зберігати всі натиснуті клавіші в структуру даних *Set*, визначення якої знаходиться в модулі *Data.Set*. Перед *mainLoop* ми визначаємо пустий сет і встановлюємо колбек (*лістинг 3.28*). Так як сет буде постійно змінюватись протягом роботи програми,

нам потрібно огорнути його в спеціальну монаду. Це робить функція *newIORef* (модуль *Data.IORef*), яка визначає гетер і сетер для значення.

```
initKeyRefs <- newIORef S.empty
G.setKeyCallback window (Just $ keyCallback initKeyRefs)
```

Лістинг 3.28

keyCallback – функція, яка приймає набір натиснутих клавіш і відповідно додає або видаляє з сету значення (*лістинг 3.29*).

```
keyCallback :: IORef (Set G.Key) -> G.KeyCallback
keyCallback ref window key scanCode keyState modKeys = do
    case keyState of
        G.KeyState'Pressed -> do
            modifyIORef ref (S.insert key)
        G.KeyState'Released -> modifyIORef ref (S.delete key)
        _ -> return ()
    when (key == G.Key'Escape && keyState == G.KeyState'Pressed)
        (G.setWindowShouldClose window True)
```

Лістинг 3.29

У кожної події, спричиненої натисненням клавіші, є декілька станів: Pressed, Released, Repeating. Нас цікавлять перші два – коли клавіша натиснута(Pressed), ми додаємо її в сет, якщо ж клавіша була «звільнена» (Released), ми видаляємо її з сету. В кінці функції також перевіряється чи була натиснута клавіша Esc або Enter і якщо так, вікно закривається.

На кожній ітерації *mainLoop* ми будемо перевизначати позицію камери, використовуючи інформацію про те, які клавіші зараз натиснуті (*лістинг 3.30*).

```
updateCamera :: Set G.Key -> GLfloat -> Camera -> Camera
updateCamera keySet speed cam@(Camera pos front up) = let
    moveVector = S.foldr (\key vec -> case key of
            G.Key'W -> vec ^+^ front
            G.Key'S -> vec ^-^ front
            G.Key'A -> vec ^-^ L.normalize (cross front up)
            G.Key'D -> vec ^+^ L.normalize (cross front up)
            _ -> vec
            ) (V3 0 0 0) keySet
    in cam {cameraPos = pos ^+^ (speed *^ L.normalize moveVector)}
```

Лістинг 3.30

В функцію *updateCamera* передається список натиснутих клавіш, швидкість, з якою ми хочемо, щоби рухалась камера та сам об’єкт камери. Використовуючи

foldr, функція проходиться по кожній клавіші і послідовно додає до нульового вектора відповідні значення. Після чого отриманий вектор додається до поточної позиції камери. *Foldr* дозволяє реалізувати реагування одночасно на декілька натиснутих клавіш.

3.5.2. Рухи мишки

За допомогою мишки реалізуються повороти камери. Щоби мати змогу роздивлятися сцену, необхідно змінювати «Front» вектор камери. Для реалізації використовуються кути Ейлера (*див рисунок 3.14*).

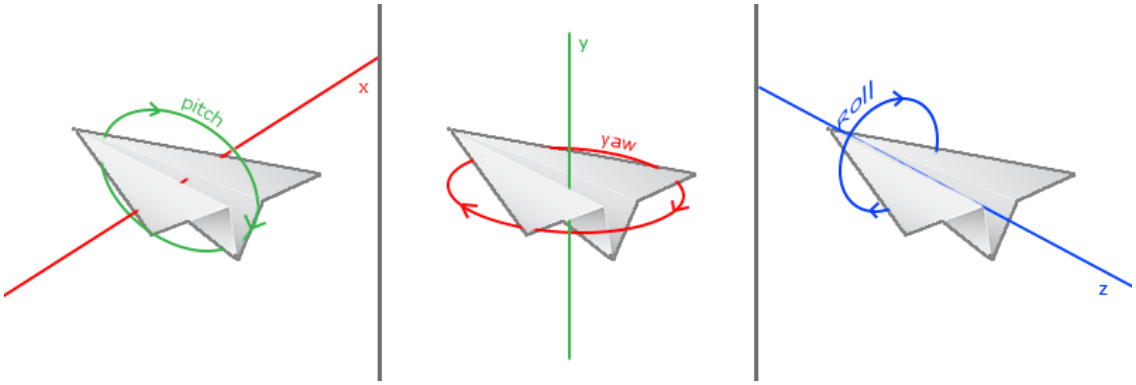

Рисунок 3.14[3]

Перший кут (pitch) описує, наскільки сильно об’єкт дивиться вгору або вниз. Другий (yaw) показує магнітуду, наскільки сильно об’єкт відхилений вліво або право. І останній (roll) описує, наскільки сильно об’єкт «перекочений».

Для камери використовуються лише перші два значення. Маючи значення цих кутів, можна перевести їх у 3D вектор і таким чином застосувати до «Front» вектора камери. Знаючи, як проектуються ці кути на відповідні осі (pitch на осі x та y, yaw на осі x та z), ми можемо поєднати відповідні проекції і отримати значення нових координат «Front» вектора (*формули 3.4 – 3.6*).

$$x = \cos(yaw) * \cos(pitch) \quad (3.4)$$

$$y = \sin(pitch) \quad (3.5)$$

$$z = \sin(yaw) * \cos(pitch) \quad (3.6)$$

Значення pitch та yaw обраховуються з врахуванням попереднього їхнього значення та зсуву, який залежить від положення мишки.

Для того, щоб обрахувати вектор, необхідно зберегти інформацію про попередні значення кутів та розташування мишки (*лістинг 3.31*).

```
data MouseInfo = MouseInfo {
    prevPos :: Maybe (Double,Double),
    prevPitch :: Double,
    prevYaw :: Double,
    frontVec :: V3 GLfloat
} deriving Show
```

Лістинг 3.31

Цей об’єкт ми зберігаємо глобально в State (*лістинг 3.24*), задаємо колбек для мишки за допомогою GLFW функції *setCursorPosCallback* та вимикаємо відображення курсору через *setCursorInputMode* (*лістинг 3.32*).

```
initMouseRef <- newIORef $ MouseInfo Nothing 0 (-90) (V3 0 0 (-1))
G.setCursorInputMode window G.CursorInputMode'Disabled
G.setCursorPosCallback window (Just $ cursorPosCallback initMouseRef)
```

Лістинг 3.32

cursorPosCallback - це допоміжна функція, яка робить всі вищенаведені розрахунки (*лістинг 3.33*).

```
cursorPosCallback :: IORef MouseInfo -> G.CursorPosCallback
cursorPosCallback ref window xpos ypos = do
    modifyIORef ref $ \oldInfo -> let
        (lastX, lastY) = case prevPos oldInfo of
            Nothing -> (xpos,ypos)
            (Just (lastX,lastY)) -> (lastX,lastY)
        sensitivity = 0.02
        xoffset = (xpos - lastX) * sensitivity
        yoffset = (lastY - ypos) * sensitivity
        lastX' = xpos
        lastY' = ypos
        oldPitch = prevPitch oldInfo
        oldYaw = prevYaw oldInfo
        newYaw = (oldYaw + xoffset) `mod'` 360.0
        newPitch = min (max (oldPitch + yoffset) (-89)) 89
        pitchR = toRadians newPitch
        yawR = toRadians newYaw
        front = L.normalize $ V3 (cos yawR * cos pitchR) (sin pitchR) (sin yawR * cos pitchR)
        in MouseInfo (Just (lastX',lastY')) newPitch newYaw front
```

Лістинг 3.33

4. HASKELL I GAMEDEV

На власному досвіді ми переконались, що фактично обмежень для задання цікавих і складних сцен на Хаскелі немає. Все залежить від бібліотеки і у випадку OpenGL у нас дійсно є доволі широкий і гнучкий набір інструментів. Проте можна піти ще далі і, використовуючи спеціальні бібліотеки, розробляти невеликі ігри.

4.1. Yampa

Yampa – це предметно-орієнтована мова, вбудована в Хаскель, яка використовує принципи FRP (Функціональне Реактивне Програмування). Синтаксис Yampa побудований на основі Arrows, які фактично є узагальненнями монад.

Як можна зрозуміти з назви, FRP є концепцією реактивного програмування, основні будівельні блоки якої засновані на принципах функціонального програмування. Дуже часто FRP використовується саме для програмування графічних інтерфейсів та ігор. Використовуючи FRP, можна визначати події у вигляді потоку. Програміст може розташовувати частини коду (сигнали) в цьому потоці, що дозволяє зв’язати їх з ціллю та відреагувати на відповідні події.

Функціональне реактивне програмування інтегрує концепцію часового потоку у чисто функціональний стиль. З його допомогою додаток стає явним та надійним.

Yampa базується на двох основний концепціях – це сигнали та сигнальні функції. Сигнал – це функція, яка позначає перехід від часу до значення[6]:

$$Signal\ \alpha \approx Time \rightarrow \alpha$$

Час є безперервним і представляється у вигляді невід’ємного дійсного числа. Параметр α позначає значення, яке передається сигналом[6].

Сигнальна функція є функцією від одного сигналу до іншого[6]:

$$SF\alpha\beta \approx Signal\ \alpha \rightarrow Signal\ \beta$$

Коли функція типу SFαβ застосовується до вхідного сигналу Signal α, на виході отримується сигнал типу Signal β.

Програмування з Yampa полягає у визначенні сигнальних функцій за допомогою набору комбінаторів та базових примітивних сигналів, які безпосередньо надає бібліотека Yampa. Найбільш базовими комбінаторами є *arr*, який перетворює звичайну функцію в сигнальну, та два композиційні комбінатори >>> та <<<[6] (*лістинг 4.1*). *Рисунок 4.1* зображає ті ж комбінатори, але у діаграмному вигляді.

```
arr   :: (a -> b) -> SF a b
(<<<) :: SF b c -> SF a b -> SF a c
(>>>) :: SF a b -> SF b c -> SF a c
```

Лістинг 4.1[6]

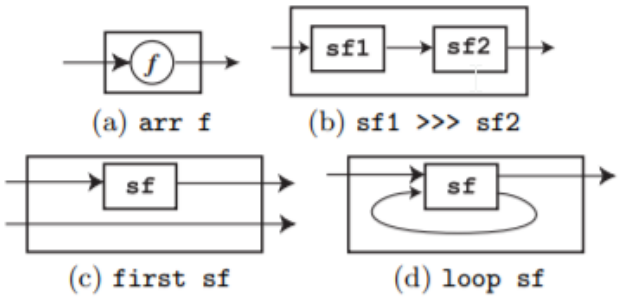

Рисунок 4.1[6]

Щоб запустити Yampa програму, потрібно якось зв'язати вхідні та вихідні сигнали із навколишнім світом[6]. Це можна зробити за допомогою функції *reactimate*, яка фактично є циклом виду «вхідні дані – процес – вихідні дані». Вона формує інтерфейс між сигнальними функціями Yampa та навколишнім світом. Як зазначалось, час вважається безперервним, але для того, щоб обробити сигнал, необхідно вибрати його в певний момент часу[7].

Ми не будемо детально зупинятись на інших комбінаторах Yampa, адже застосування Yampa на практиці є темою для окремої роботи.

Застосовуючи принципи Yampa, можна визначати логіку динамічних об’єктів, стан яких змінюється з часом (наприклад, гравець, ворог, можливо деякі елементи середовища).

Висновки

Хаскель – суто математична мова програмування. Цей міф існує уже не один рік серед тих, хто не знайомий з цією мовою програмування.

В рамках цієї курсової роботи було доведено, що можливості Хаскелю не обмежуються програмуванням математичних формул. Навпаки, він пропонує широкий спектр засобів для створення цікавих та складних програм, в тому числі і графічних, які, більше того, можна розвинути і використовувати як основу для розробки ігор.

Було реалізовано побудову сцени з ікосаедром та кубом, який виконує роль джерела світла та земульовано ефект камери, яку можна рухати і повертати. Фактично подібна програма надає базовий фундамент для створення набагато складніших сцен, адже було вирішено найголовнішу проблему - поєднання Хаскелю, де немає поняття змінного глобального стану, з графічною бібліотекою, логіка якої якраз заснована на подібному стані.

Звичайно, графічна розробка та gamedev – не ті задачі, для яких Хаскель створювався. Тому можливості використання графіки все ще обмежені через недостатню кількість навчальних матеріалів та стабільно працюючих бібліотек. Проте задачі маленької та середньої складності абсолютно реально реалізувати (і деколи навіть зручніше) засобами Хаскелю.

Список використаної літератури

1. Applications and libraries/GUI libraries [Електронний ресурс]. – Режим доступу: https://wiki.haskell.org/Applications_and_libraries/GUI_libraries
2. Ричард С. Райт-мл. OpenGL Суперкнига (Третье издание) / Ричард С. Райт-мл, Бенджамин Липчак – М. : Издательский дом «Вильямс», 2006 – 53-59с, 61-74с, 85-87с, 93-95с
3. Learn OpenGL [Електронний ресурс]. – Режим доступу: https://learnopengl.com/Getting-started/OpenGL
4. Sven Eric Panitz. HOpenGL – 3D Graphics with Haskell – 1-2с, 11-13с, 37с, 51с
5. OpenGL Sphere [Електронний ресурс]. – Режим доступу: http://www.songho.ca/opengl/gl_sphere.html
6. Antony Courtney. The Yampa Arcade / Antony Courtney, Henrik Nilsson, John Peterson – [Електронний ресурс]. – Режим доступу: https://www.antonycourtney.com/pubs/hw03.pdf
7. Yampa/reactimate [Електронний ресурс]. – Режим доступу: https://wiki.haskell.org/Yampa/reactimate