

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування інституту, назва факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДІВ
РАСТЕРИЗАЦІЇ ТЕКСТУ В АПАРАТНО ПРИСКОРЕНИХ ЗАСТОСУНКАХ
RESEARCH AND SOFTWARE IMPLEMENTATION OF TEXT
RASTERIZATION METHODS IN THE HARDWARE-ACCELERATED
APPLICATIONS

Виконав: студент(ка) 2 курсу, групи КНТ-111м

Спеціальності 121 Інженерія програмного

забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

Дроздов О.М.

(прізвище та ініціали)

Керівник Гладкова О.М.

(прізвище та ініціали)

Рецензент Зеленьова І.Я.

(прізвище та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Інститут, факультет ФКНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
С.О. Субботін
“ ” 2022 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

Дроздова Олександра Миколайовича
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація методів
растеризації тексту в апаратно прискорених застосунках. Research and Software
Implementation of Text Rasterization Methods in the Hardware-Accelerated
Applications

керівник проєкту (роботи) Гладкова Ольга Миколаївна, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “4” листопада 2022 року № 363

2. Строк подання студентом проєкту (роботи) 1 грудня 2022 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне
завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань дослідження.
2. Матеріали та методи. 3. Розробка архітектури програми. 4. Основні рішення
щодо реалізації компонентів системи. 5. Експлуатація, тестування та
експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 розділ	Гладкова О.М., доцент		
Нормоконтроль	Белова А.В., асистент		

7. Дата видачі завдання “ 01 ” вересня 2022 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання
2	Аналіз предметної області.	2–3 тижні	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів вирішення задачі.	4–5 тижні	Розділ 2
4	Розробка архітектури програми.	6 тиждень	Розділ 3
5	Розробка програми.	7–8 тижні	Розділ 4
6	Тестування та експериментальне дослідження програмного забезпечення.	9 тиждень	Розділ 5
7	Оформлення пояснювальної записки та документів до неї.	10-13 тижні	Додатки
8	Нормоконтроль та рецензування.	14–15 тижні	
9	Захист роботи.	16 тиждень	

Студент

(підпис)

Дроздов О.М.

(прізвище та ініціали)

Керівник проєкту (роботи)

(підпис)

Гладкова О.М.

(прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
110 с., 8 табл., 35 рис., 2 дод., 35 джерел.

БАГАТОКАНАЛЬНЕ ПОЛЕ ДИСТАНЦІЙ ЗІ ЗНАКОМ, ВЕКТОРНА
ГРАФІКА, ВЕБЗАСТОСУНОК, РАСТЕРИЗАЦІЯ ГЛІФІВ, WEBASSEMBLY.

Об'єкт дослідження – процес растеризації тексту в апаратно
прискорених застосунках.

Предмет дослідження – методи растеризації тексту, які
використовуються в апаратно прискорених застосунках.

Мета роботи – створення апаратно-прискореного застосунку для
растеризації тексту із широкими можливостями його форматування.

Матеріали, методи та технічні засоби: структурне та імперативне
програмування, криві Безьє, векторна графіка, файли шрифтів, поля
дистанцій зі знаком, мова програмування Zig та JavaScript, мова розмітки
HTML, технологія WebAssembly, редактор VSCode, персональний комп'ютер
з процесором Intel Core i5 під управлінням операційної системи Microsoft
Windows 11.

Результати. Досліджено методи растеризації тексту в апаратно-
прискорених застосунках. Створено застосунок, який використовує метод
mSDF та надає широкі можливості по форматуванню тексту.

Висновки. Існуючі методи мають різні переваги та недоліки.
Створений застосунок дозволяє переконатись в інтерактивній формі, чи
задовольняє метод mSDF бажану якість растеризації тексту та можливості
його форматування.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 110 pages, 35 figures, 8 tables, 2 appendixes, 35 sources.

GLYPH RASTERIZATION, MULTI-CHANNEL SIGNED DISTANCE FIELD, VECTOR GRAPHICS, WEB APPLICATION, WEBASSEMBLY.

Object of study is text rasterization process in hardware accelerated applications.

Subject of study are text rasterization methods used in hardware accelerated applications.

The purpose of the work is to create a hardware-accelerated application for text rasterization with wide possibilities of text formatting.

Materials, methods, and tools: structured and imperative programming, Bezier curves, vector graphics, font formats, signed distance field, Zig and JavaScript programming languages, HTML markup language, WebAssembly, VSCode editor, personal computer with Intel Core i5 processor running Microsoft Windows 11 operating system.

Results. Investigated methods of text rasterization in hardware-accelerated applications. Developed an application that uses the mSDF method and provides a wide range of options for text formatting.

Conclusions. Existing methods have different advantages and disadvantages. The created application allows you to check in an interactive form whether the mSDF method satisfies the desired quality of text rasterization and formatting capabilities.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз проблеми та постановка завдань дослідження	10
1.1 Растеризація тексту	10
1.2 Визначення контурів гліфів у файлах шрифтів	13
1.3 Огляд та порівняння методів	16
1.3.1 Текстурні методи. Атлас растрованих гліфів	16
1.3.2 Текстурні методи. Signed Distance Fields та mSDF	18
1.3.3 Методи із використанням геометрії.....	22
1.3.4 Методи із безпосереднім використанням рівнянь кривих.....	25
1.3.5 Порівняння методів.....	29
1.4 Висновок за розділом 1.....	31
2 Матеріали та методи	32
2.1 Метод створення SDF на GPU	32
2.2 Модифікація методу для створення mSDF.....	34
2.3 Висновок за розділом 2.....	36
3 Розробка архітектури програми.....	37
3.1 Вибір мови програмування	37
3.2 Вибір графічного API	40
3.3 Опис архітектури застосунку.....	43
3.4 Висновок за розділом 3.....	47
4 Основні рішення щодо реалізації компонентів системи.....	48
4.1 Опис структури застосунку та головного циклу	48
4.2 Опис реалізації формування mSDF атласу.....	54
4.3 Висновок за розділом 4.....	58
5 Експлуатація, тестування та експериментальне дослідження програми	59
5.1 Призначення програми	59
5.2 Робота користувача із застосунком.....	59

5.3 Експериментальне дослідження програми.....	60
Висновки	65
Перелік джерел посилання	66
Додаток А Текст програми.....	70
Додаток Б Слайди презентації	103

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- 3D – трьохвимірний;
- API – Application Programming Interface;
- CPU – Central Processing Unit;
- GPU – Graphics Processing Unit;
- HTML – HyperText Markup Language;
- MSDF – Multi-Channel SDF;
- SDF – Signed Distance Field;
- WASM – WebAssembly.

ВСТУП

В апаратно-прискорених додатках, які виконують візуалізацію у реальному часі часто виникає необхідність відображення тексту всередині 3D-сцени. Такі потреби можуть бути розширені до більш складних як відображення рядків тексту на геометричних поверхнях та ефекти тексту як тінь та обведення. Простий метод растеризації окремих гліфів за певного розміру та їх подальша композиція у рядки має низьку візуальну якість. Це пояснюється тим, що позиція камери, з якої виконується подання сцени, майже завжди переміщується в просторі та обертається, тому гліфи з яких складається текст у сцені, постійно змінюють своє положення. Таким чином, розмір гліфів у кінцевому зображенні майже ніколи не буває однаковим від кадру до кадру. Крім того, гліфи часто малюються із врахуванням перспективи, оскільки камера не завжди спрямована перпендикулярно до умовної площини, на яку наноситься текст. Ці проблеми та вимоги створюють необхідність у методі для динамічного відтворення гліфів з високою якістю при довільних афінних і проекційних перетвореннях.

Велика кількість існуючих методів растеризації тексту, їх модифікацій та переваг та недоліків, робить вибір складним без існуючих робіт, які б порівнювали усі методи. Не існує інтерактивного застосунку який би дозволяв порівняти їх реалізацію у реальній 3D-сцені.

Об'єкт дослідження – процес растеризації тексту в апаратно прискорених застосунках.

Предмет дослідження – методи растеризації тексту, які використовуються в апаратно прискорених застосунках.

Мета роботи – створення апаратно-прискореного застосунку для растеризації тексту із широкими можливостями його форматування.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

У даному розділі буде розглянуто необхідні кроки, які передують раструванню тексту. Описано те як визначаються векторні гліфи у файлах шрифтів. Буде порівняно методи растрування гліфів та обрано один із них.

1.1 Растеризація тексту

Ми вже звикли та очікуємо, що застосунки підтримують різні стилі шрифтів, курсив, напівжирний, розмір, зміну кольору, градієнтну заливку, обведення тексту, свічення, відкидання тіні текстом. Дані опції форматування можуть визначатися у вигляді деревоподібної структури із наслідуванням властивостей для зручності введення, наприклад HTML, а при завантаженні таке представлення перетворюється у масив текстових прогонів.

Текстовий прогін (text run) – це послідовність символів, які мають однаковий набір властивостей форматування, однаковий напрямок написання, скрипт написання та мову [1]. Важливо відзначити різницю між скриптом і мовою: більшість скриптів використовується для написання різноманітних мов (латиниця, може використовуватися для написання десятків мов), і багато мов можуть бути написані більш ніж одним скриптом (японська мова використовує кандзі, хірагану та катакану) [2].

Розглянемо кроки, які необхідно виконати від представлення текстового прогону у пам'яті до його відображення на екрані.

По-перше необхідно визначитись із кодуванням тексту у пам'яті. На щастя, дні численних кодувань вже у минулому і кожна операційна система, база даних, вебсайт, текстовий файл, мова програмування використовує Юнікод у форматі UTF-8, UTF-16LE/BE, чи UTF-32.

Кожен текстовий прогін передається у рушій формування тексту (text-shaping engine). Формування тексту є невід'ємною частиною підготовки

тексту до відображення. Перш ніж послідовність Юнікод може бути відображена, кодові точки в рядку повинні бути зіставлені з відповідними гліфами, які знаходяться у шрифті, і ці гліфи повинні бути правильно розташовані один відносно одного. Для багатьох скриптів написання, що підтримуються в Юнікодi, ці кроки включають специфічні для скрипту правила компонування, включаючи складну поведінку з'єднання, переупорядкування та позиціонування. Реалізація цих правил покладена на рушій формування [3]. Багато шрифтів містять лігатури – комбінації символів, які відображаються як єдине ціле. Наприклад, послідовність літер "f, i" зазвичай відображається у друкованому вигляді як єдиний лігатурний гліф "fi" (рис. 1.1).

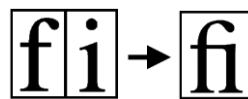


Рисунок 1.1 – Окремі літери та лігатурний гліф

Хоча такі лігатури, як "fi", є необов'язковими типографськими уточненнями, деякі мови вимагають певних замін для правильного відображення тексту. Кожна арабська літера має чотири різні варіанти, що відповідають різним позиціям, у яких вона може з'являтися у послідовності. Шрифт має окремі гліфи для початкової, середньої, кінцевої та ізольованої форм кожної літери, кожна з яких має свій ідентифікатор гліфа. Юнікод призначає тільки одну кодову точку для кожного символу, таким чином рядок Юнікоду не містить інформацію, який варіант гліфа треба використовувати для кожного символу. Щоб прийняти рішення, потрібно проаналізувати весь рядок і визначити відповідний гліф для кожного символу на основі його позиції. Тобто, рушій формування тексту вибирає правильну форму букви за її позицією та повертає правильний гліф зі шрифту [3].

Нарешті, необхідно виконати растеризацію гліфів на позиціях обчисленими рушієм формування. Растеризацією є процес перетворення

векторного зображення у растрове, тобто перетворення сукупності геометричних примітивів, що визначають гліф, у масив пікселів.

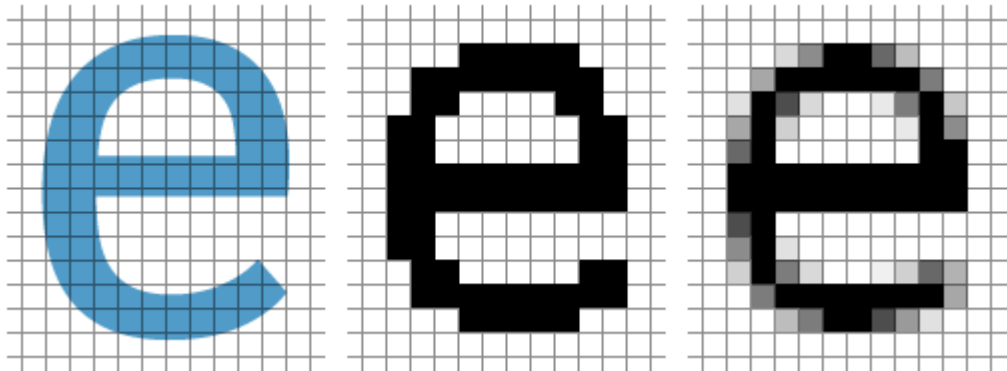


Рисунок 1.2 – Растеризація гліфу [4]

На рисунку 1.2 ліворуч зображено гліф у векторному вигляді поверх збільшеним у розмірі масиві 10x12 пікселів. По центру зображено растровий гліф, який було створено за умовою: зафарбувати суцільним кольором, якщо центр пікселю знаходиться всередині векторного представлення гліфу. Даний спосіб дискретизації простий, але неприємний для ока людини на екранах моніторів із стандартною щільністю пікселів у 96 PPI (pixels per inch). Меншої похибки дискретизації можна досягти аналітично підрахувавши область пікселя, яка всередині векторного гліфу та зафарбувати його пропорційно до цього значення, що зображено на рис. 1.2 праворуч. Варто зазначити, що збільшення розміру тексту не впливає на підвищення якості, що сприймається оком. Лише збільшення PPI екрану або більш якісний аналітичний метод допомагають сприймати результат більш згладженим.

Так як погрішності, так звані «зубці», при недостатній частоті дискретизації стають менш помітними лише на екранах із 300 PPI, які можна зустріти здебільшого лише на мобільних пристроях, то процес растеризації повинен виконувати і якісне згладжування гліфів.

1.2 Визначення контурів гліфів у файлах шрифтів

Більшість файлів шрифтів, які використовуються мають розширення ttf або otf. Контури гліфів можуть визначатися у двох форматах TrueType або Compact Font Format (CFF). Типове заблудження, що файл із розширенням ttf має TrueType контури, а otf – CFF контури.

TrueType – це стандарт контурних шрифтів, розроблений компанією Apple наприкінці 1980-х років як конкурент шрифтів Adobe Type 1, які використовувались у мові розмітки сторінок PostScript [5]. Щоб забезпечити його широке розповсюдження, компанія Apple безкоштовно надала Microsoft ліцензію на використання TrueType. Пізніше у 1990-х Microsoft спробувала ліцензувати технологію Apple GX Typography. Ці переговори не увінчалися успіхом, що спонукало Microsoft до розробки власної [6]. Adobe приєдналася до Microsoft у цих зусиллях, додавши підтримку контурних гліфів, що використовували у своїх шрифтах Type 1. Ці зусилля мали на меті витіснити як Apple TrueType так і Adobe Type 1 формати шрифтів. Потребуючи формату шрифту для обробки тонкої типографіки і складної поведінки різних систем письма у світі, дві компанії об'єднали основні технології обох форматів і додали нові розширення, призначені для усунення обмежень цих форматів. Для об'єднаних технологій було обрано назву OpenType [6].

Файл OpenType містить дані у форматі колекції таблиць, які включають TrueType або Compact Font Format (CFF) контури шрифту. Растеризатори використовують дані з таблиць, що містяться у шрифті, для відтворення TrueType або PostScript контурів гліфів. Деякі з допоміжних даних у таблицях використовуються незалежно від того, який формат контуру використовується; деякі допоміжні дані є специфічними для TrueType або PostScript контурів [7]. Формат OpenType був побудований на основі свого попередника TrueType, зберігши базову структуру таблиць TrueType і додавши багато нових таблиць, які використовуються рушієм формування, як було зазначено у попередньому підрозділі.

Таким чином, файли з розширенням ttf повинні бути у старішому стандарті TrueType, а файли otf – у новішому OpenType, який підтримує обидва типи контурів. Єдиним правильним способом визначення типу контурів, що знаходяться у файлі є перевірка наявності відповідних таблиць контурів.

Таблиця із назвою glyf містить інформацію, яка описує гліфи в шрифті із форматом контурів TrueType. Контур гліфа складається з набору контурів. Простий гліф може мати тільки один контур. Більш складні гліфи можуть мати два і більше контурів. Складні гліфи можуть бути побудовані шляхом поєднання двох або більше простих гліфів [8]. Контури складаються з прямих і кривих ліній. Криві визначаються набором точок, які описують сплайни Безьє другого порядку. Формат контурів TrueType використовує два типи точок для визначення кривих: ті, що знаходяться на кривій, і ті, що знаходяться поза нею. При визначенні кривої допустима будь-яка комбінація точок, що лежать і не лежать на кривій. Прямі лінії визначаються двома послідовними точками на кривій. Для визначення заповненості контурів, що утворюють гліф, має значення, чи є нумерація точок зростаючою або спадаючою. Напрямок кривих повинен бути таким, щоб, якщо слідувати за кривою в напрямку збільшення номерів точок, зафарбована область гліфу завжди знаходилась праворуч [8].

Таблиця із назвою CFF2, містить інформацію, яка описує гліфи в шрифті із форматом контурів Compact Font Format 2.0. В контексті растеризації він відрізняється від TrueType тим, що містить сплайни Безьє третього порядку.

Крива Безьє – параметрична крива, що використовується в комп'ютерній графіці та суміжних областях. Крива Безьє задається множиною контрольних точок від P_0 до P_n , де n називається порядком кривої ($n=1$ для лінійної, 2 для квадратичної, 3 для кубічної, як зображено на рис. 1.3). Перша та остання контрольні точки завжди є кінцевими точками

кривої, проте проміжні контрольні точки (якщо такі є), як правило, не лежать на кривій [9].

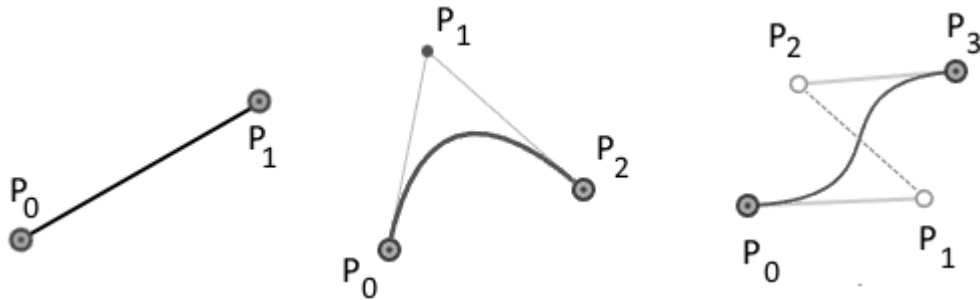


Рисунок 1.3 – Криві Безьє перших трьох порядків

Квадратична крива Безьє визначається трьома вершинами й описується рівнянням виду:

$$B(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2, t \in [0,1].$$

Кубічна крива Безьє визначається чотирма вершинами й описується рівнянням виду:

$$B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3, t \in [0,1].$$

Кожна квадратична крива Безьє є також кубічною кривою Безьє. Крива може бути розбита в будь-якій точці на дві підкриві або на довільну кількість підкривих, кожна з яких також є кривою Безьє. Крива Безьє порядку вище двох може перетинатися сама з собою або мати перегин при певному виборі контрольних точок [10].

1.3 Огляд та порівняння методів

Розмаїття методів растрування векторної графіки, пояснюється тим, що графічні процесори не надають для цього відповідні примітиви. Натомість розробники повинні використовувати трикутники, буфер глибини, буфер трафарету, текстури, шейдер програми тощо, для відтворення векторних контурів.

Відтворення 2D-графіки є складним, тому що мова йде про форми – точні межі контурів – а не про матеріали та освітлення об’ємних 3D-об’єктів, які в основному заповнюються суцільним кольором. Графічні процесори, як результат історії, вирішили не зосереджуватися на неявній геометрії в реальному часі, такий як криві, а натомість на всьому, що відбувається всередині трикутників, та їх великій кількості на екрані [11].

Методи можна класифікувати у ті, що використовують переваги швидкості GPU у її власних примітивах – текстурні та геометричні методи із невеликим шейдером; та ті, що намагаються виконати ефективно розпаралелювання класичних CPU алгоритмів на рівні окремого пікселю, реалізуючи його у велику шейдер програму.

1.3.1 Текстурні методи. Атлас растрованих гліфів

Найпростішим методом відображення тексту у 3D-сцені є попередня растеризація гліфів, що входять до складу шрифту, при одному або декількох певних розмірах шрифту та збереження результатів у вигляді растрового зображення, який називають атласом (рис. 1.4).



Рисунок 1.4 – Фрагмент атласу

Текст відтворюється у 3D-сцені шляхом створення геометрії одного прямокутника, який складається з двох трикутників, для кожного гліфу. Кожній вершині трикутника призначаються текстурні координати, які посилаються на попередньо растрований гліф у текстурному атласі.

Цей метод був популярним через його простоту реалізації та відмінну швидкодію, але якість зображення на виході вкрай обмежена, в першу чергу через розмиття викликаного білінійною інтерполяцією, коли відображувані розміри гліфів перевищують роздільну здатність з якою вони створювалися у текстурному атласі [12]. Даний недолік не є критичним, якщо застосунок може доповнювати атлас гліфами необхідних розмірів. Однак, у 3D-сцені гліфи з яких складається текст, постійно змінюють своє положення, розмір гліфів у кінцевому зображенні майже ніколи не буває однаковим від кадру до кадру, гліфи часто малюються із врахуванням перспективи. Така велика кількість динамічних параметрів робить постійне доповнення атласу неможливим.

Даний метод все ще використовується апаратно-прискореними застосунками, але лише за умови, що текст відображається перпендикулярно до користувача, тобто елементи 2D-інтерфейсу, та немає необхідності в анімації розміру тексту. Для деяких застосунків, як браузері, пріоритетним є відтворення стилю растеризації гліфів, який співпадає із операційною системою. Вони використовують API технології, яка надається ОС Windows – DirectWrite, Apple – Quartz, Linux та Android – бібліотека FreeType.

Яскравим виключенням з вище наведених умов є Google Maps [13], які у режимі перегляду вулиць використовують даний метод для відображення назв вулиць на дорозі, що виглядає дуже низької якості (рис. 1.5).

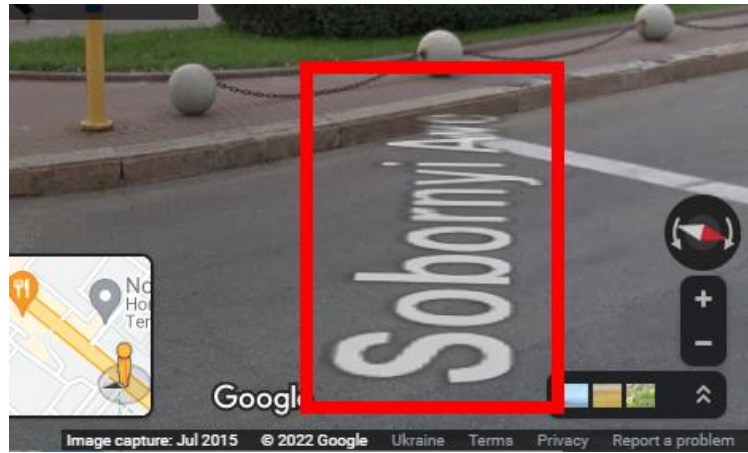


Рисунок 1.5 – Приклад низької якості растрованих гліфів при проекціях [13]

1.3.2 Текстурні методи. Signed Distance Fields та mSDF

Метод описаний у статті [14] здатен створювати чіткі контури гліфів у різних масштабах, без розмиття, без необхідності декількох розмірів растрованих гліфів.

В атласі замість того, щоб зберігати растрований вигляд кожного гліфа за певного розмірі тексту, зберігається найближча відстань до контуру зі знаком (рис. 1.6). Таким чином утворюється поле дистанцій зі знаком (signed distance field, SDF). Дистанції до контуру ззовні та всередині мають протилежні знаки. Дивлячись на знак, шейдер програма вирішує чи повинен піксель бути зафарбований.

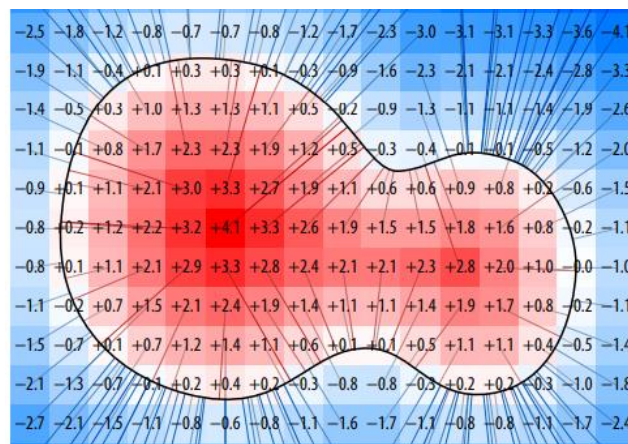


Рисунок 1.6 – Поле дистанцій зі знаком до контуру [15]

Значення дистанцій у такому полі можна інтерполювати, а отже використовуються переваги білінійної інтерполяції текселів у GPU. Атлас із SDF гліфами має характерний вигляд із нечіткими контурами (рис. 1.7).

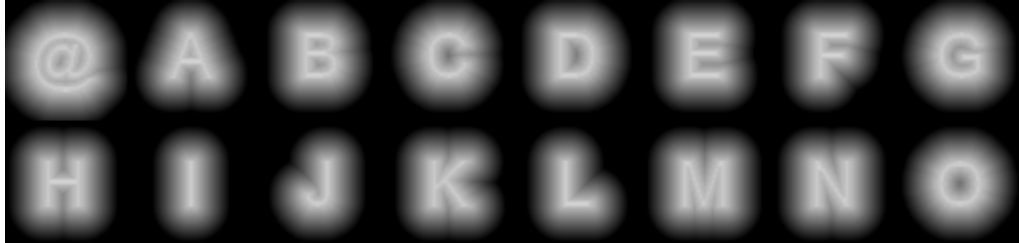


Рисунок 1.7 – Фрагмент SDF атласу

Радіус з яким створюється SDF зображення дозволяє використовувати додаткові ефекти до тексту: м'які тіні, свічення, обведення контуру. Це не сумнівно є перевагою, адже використовується тільки два трикутники для геометрії та дуже простий шейдер.

Оскільки метод покладається на інтерполяцію, яка забезпечує якісне наближення лише там, де швидкість зміни є більш-менш постійною, кути не можуть бути реконструйовані правильно, і замість цього будуть виглядати закругленими або сколотими, що зображено на рисунку 1.8 ліворуч [15].



Рисунок 1.8 – Округлені краї гліфів при одноканальному SDF

Для вирішення цієї проблеми у статті пропонується використання більше ніж одного каналу у текстурі, та виконувати декомпозицію контуру у них. Альтернативну спробу із комбінуванням SDF та векторного методу було виконано у роботі [16], але результат виявився у 5-12 разів повільніше.

Метод декомпозиції у непарну кількість каналів було описано пізніше у роботі [15]. При декомпозиції розглядається кожен контур гліфу, криві Безьє, що не утворюють кутів умовно об'єднуються у безперервні плавні сегменти контуру. Кожному сегменту призначається колір (рис. 1.9). Існує два правила для призначення кольорів. По-перше, кожен сегмент повинен мати принаймні щонайменше два канали, а по-друге, для того, щоб кут залишався гострим, два сусідні сегменти у контурі повинні мати рівно один спільний канал [15].



Рисунок 1.9 – Фарбування контурів у сегменти літери «е» [15]

Із мінімально необхідними змінами до коду генерації SDF створюється багатоканальний SDF (multi-channel SDF, mSDF) атлас (рис. 1.10). Фактично кожен піксель тепер зберігає відстань до двох найближчих контурів, а третій канал визначає яку булеву операцію між ними необхідно виконати. У шейдері також необхідно змінити лише одну лінію коду, після чого отримуємо гліфи із якісними кутами на рисунку 1.8 праворуч.



Рисунок 1.10 – Фрагмент mSDF атласу

Не дивлячись на значне підвищення якості в mSDF, із моїх власних спостережень, багато застосунків та ігор все ще використовують

одноканальний SDF. Популярний ігровий рушій Unity у плагіні для тексту за замовчуванням TextMeshPro, дозволяє створити тільки одноканальні SDF. Аналогічна ситуація у рушії Unreal Engine. Серед більш популярних тільки Godot Engine отримав підтримку mSDF у минулому 2021 році. Цікаво, що навіть кросплатформний фреймворк Qt використовує SDF замість атласу гліфів растрованих операційною системою [17].

Треба розуміти, що mSDF хоч і надає чіткі контури для будь-якого розміру тексту, він все ще є текстурним, а отже залежить від розміру текстури. Необхідно підбирати такий розмір поля, який зміг би передати найтонший елемент контуру у гліфі. Це у свою чергу призводить до даремного витрачання простору у більш широких частинах гліфу та необхідного розміру в цілому. Ця проблема актуальна для каліграфічних, курсивних шрифтів. На рисунку 1.11 зображено стилі шрифтів із такими особливостями стилю та порівняння із sans-serif стилем знизу, який має однорідну товщину у всьому гліфі.



Рисунок 1.11 – Тонкі елементи контурів у гліфів

Оптимізацію використаного простору текстури є спроби виконати за допомогою методу адаптивно дискретизованих полів відстаней (ADF) дослідженим у роботах [18] та [19].

Метод полягає у рекурсивному поділі текстури на комірки із бажаною точністю таким чином, щоб не виникало похибок білінійної інтерполяції (рис. 1.12). Для реконструкції гліфу із ADF на кожен комірку створюється

геометрія прямокутника із двох трикутників. Вершинам призначаються текстурні координати відповідних комірок.

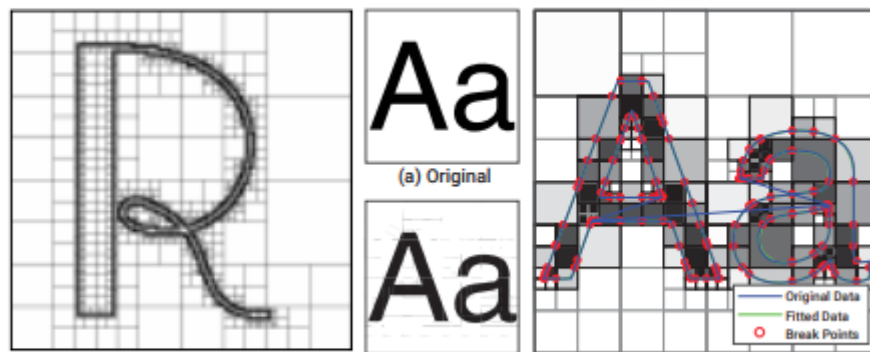


Рисунок 1.12 – Виконання поділу в ADF [18], [19]

Даний метод потребує як крок створення поля дистанцій зі знаком так і великої кількості геометрії для реконструкції гліфу, що робить ADF менш привабливим ніж використання чисто геометричного методу.

1.3.3 Методи із використанням геометрії

Растрезація гліфів на основі побудови сітки із трикутників була популяризована публікацією [20]. Над сукупністю контрольних точок, що визначають криві Безьє у контурах гліфу, виконується триангуляція Делоне. Видаляються ребра та трикутники які не належать гліфу, отримується сітка із трьох типів трикутників. Ті, що лежать всередині контуру векторного гліфу та ті, що частково заповнені – внутрішні, або зовнішні по відношенню до гліфу (рис. 1.13).

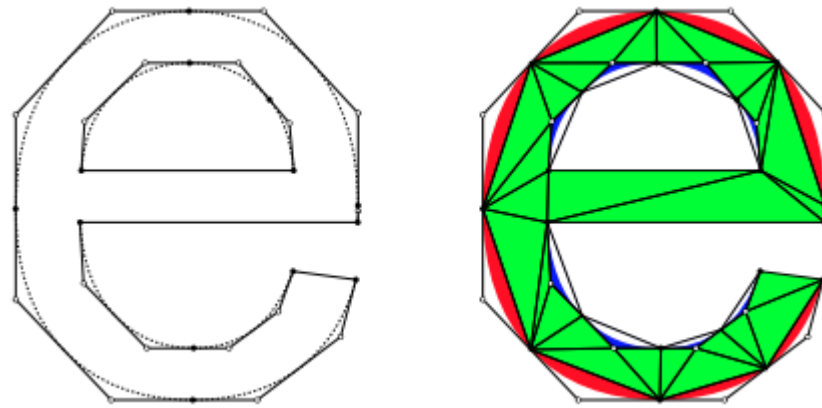


Рисунок 1.13 – Криві, що описують гліф та результат тріангуляції контрольних точок [20]

Трикутники, що є частково заповненими покривають щонайбільше одну криву Безьє. Незалежно від позицій контрольних точок квадратичної прямої, текстурні координати визначаються як на рисунку 1.14. Позиції вершин є контрольними точками.

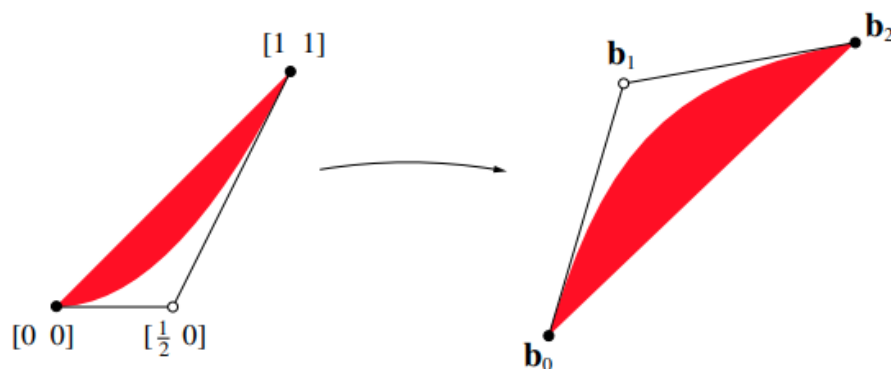


Рисунок 1.14 – Текстурні координати вершин (ліворуч), реальні позиції вершин на екрані (праворуч) [20]

Виконуючи три простих арифметичних операцій у шейдері над інтерпольованими текстурними координатами, визначається з якої сторони кривої Безьє лежить піксель, яким чином його зафарбовувати. Дана публікація також надає рівняння для кубічних кривих Безьє, однак для нього зареєстровано патент в США.

Даний метод якісно відтворює точні форми гліфів у всіх масштабах, але він вимагає дуже складного кроку триангуляції, в якому кількість вершин пропорційна кількості контрольних точок, що визначають кожен гліф. Після триангуляції також треба перевірити спеціальні випадки, наприклад трикутники, що можуть перетинатися, для уникнення артефактів.

Модифікацію методу без виконання кроку складної триангуляції було описано у [21]. Для цього використовується маска трафарету на GPU. Трикутники, що частково заповнені по контуру раструються без змін. Для заповнення внутрішньої частини гліфу створюються трикутники, одна вершина яких спільна – початок контуру, а інші дві визначаються контрольними точками кривої Безьє (рис. 1.15).

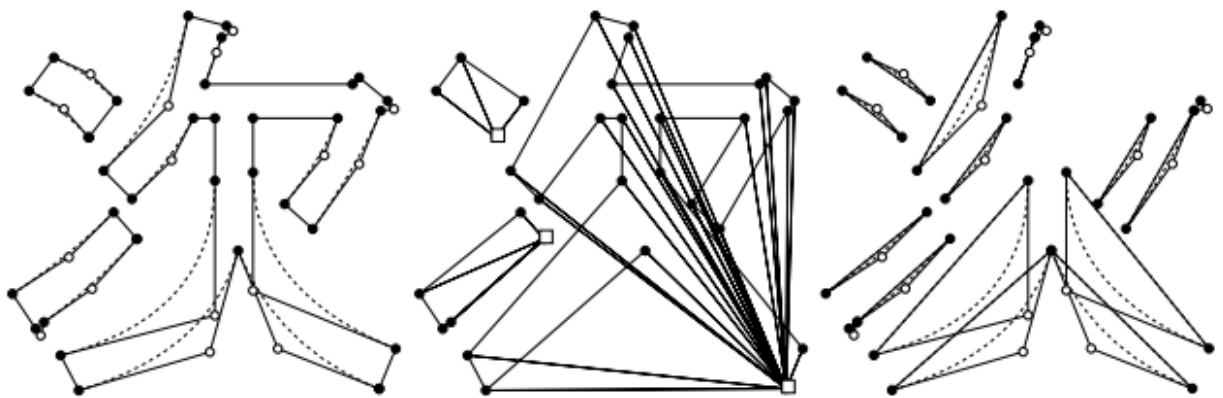


Рисунок 1.15 – Створення маски трафарету [21]

Оригінальний та модифікований методи, як правило, створюють багато невеликих, тонких трикутників, які перешкоджають оптимальній швидкодії графічного процесора [12]. Кожен гліф повторюється у тексті по декілька разів, робити декілька копій геометрії дуже витратно, тому використовується можливість GPU для створення декількох екземплярів у різних позиціях, що у свою чергу обмежує можливі ефекти, які можна застосувати на рівні окремого гліфу. Також відсутня можливість одночасного створення спеціальних ефектів до тексту без додаткових кроків.

1.3.4 Методи із безпосереднім використанням рівнянь кривих

Довгі роки виконувати растеризацію гліфів на GPU так само як на CPU, із безпосереднім використанням рівнянь кривих, у реальному часі, було не практично. Однак поліпшення архітектур сучасних GPU, швидкості доступу до пам'яті та банальне підвищення кількості транзисторів дозволяє це робити, не забираючи велику частку часу від растеризації безпосередньо 3D-об'єктів у сцені.

Дані методи відтворюють гліф, використовуючи лише два трикутники, аналогічно текстурним методам. На графічному процесорі шейдер, отримує доступ до контрольних точок кривих Безьє, що визначають контур гліфа. Щоб визначити, чи знаходиться піксель всередині або зовні гліфу, обчислюється індекс контуру відносно точки (winding number).

Індекс визначається шляхом проведення променя (рис. 1.16) від даної точки до нескінченності (напрямок, не має значення). Починаючи з нуля, ми віднімаємо одиницю кожного разу, коли контур гліфа перетинає промінь справа наліво або знизу вгору. Додаємо одиницю кожен раз, коли контур гліфа перетинає промінь у протилежних напрямках. Якщо отримане значення індексу не дорівнює нулю, то точка лежить всередині гліфу [8].

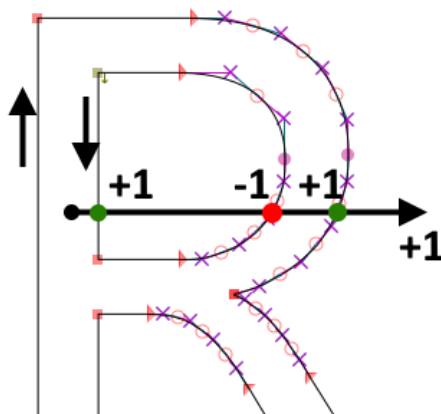


Рисунок 1.16 – Визначення індексу для точки

Для забезпечення практичності даного методу, необхідно зменшити кількість кривих Безьє, з якими промінь може потенційно перетинатися. Таким чином зменшується не тільки кількість обчислень, а й кількість даних, які необхідно завантажити при виконанні шейдеру на кожному пікселі. Для цього у статі [22] було запропоновано використовувати текстуру, перша частина якої зберігає сітку на яку розбивається гліф (рис. 1.17), а друга частина власне містить контрольні точки кривих. У частині із сіткою зберігається підмножина кривих, які перетинають кожен її клітинку, а саме – індекси до контрольних точок у другій частині.



Рисунок 1.17 – Сітка, що розбиває криві у контурах [22]

При реалізації методу автор знайшов та зазначив такі недоліки [22]:

- необхідно повернутися до використання текстурного атласу, коли текст досить малий так, що один піксель охоплював більше однієї клітини сітки, інакше спостерігаються ефекти аліасингу;
- можливі непередбачувані випадки, коли в одному місці міститься занадто багато кривих, переполюючи максимальну кількість кривих на клітинку сітки.

Схожий метод було використано у бібліотеці Glyphy [23] від відомого автора бібліотеки формування тексту Harfbuzz. Він апроксимує криві Безьє круговими дугами, поміщає їх у текстури, зберігає індекси до дуг у сітці, а відстань до них оцінює у піксельному шейдері.

Однак, головною проблемою, незалежно від методу із обчисленням індексу чи обчисленням відстані, є похибки точності округлення при виконанні обчислень із використанням 32-розрядних чисел із плаваючою

точкою. Похибки округлень призводять до неправильної класифікації розташування пікселів по відношенню до контуру гліфа. Це проявляється у вигляді окремих пікселів (рис. 1.18) або їх кластерів.

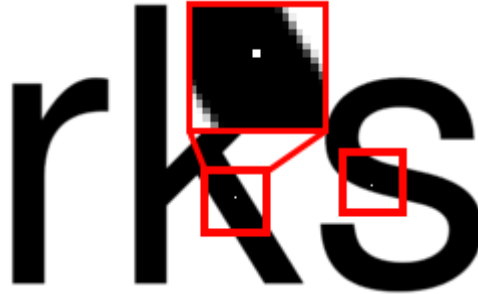


Рисунок 1.18 – Артефакти растеризації через недостатню точність чисел із плаваючою точкою

Дані похибки виникають під час обчислення коренів квадратичного рівняння у квадратичних кривих Безьє, коли деякі із коефіцієнтів наближуються до нуля. Цей недолік було усунуто у роботі [12] та реалізовано у бібліотеці автора під назвою Slug.

Числова стійкість у Slug досягається шляхом класифікації контрольних точок кожної кривої Безьє, щоб розбити безліч всіх можливих квадратичних кривих Безьє на вісім класів еквівалентності (рис. 1.19). Невелика таблиця двійкових значень у шейдері визначає, чи можуть корені рівняння кривої Безьє змінювати індекс контуру відносно точки у певному пікселі [12]. Цей метод запатентовано у США.

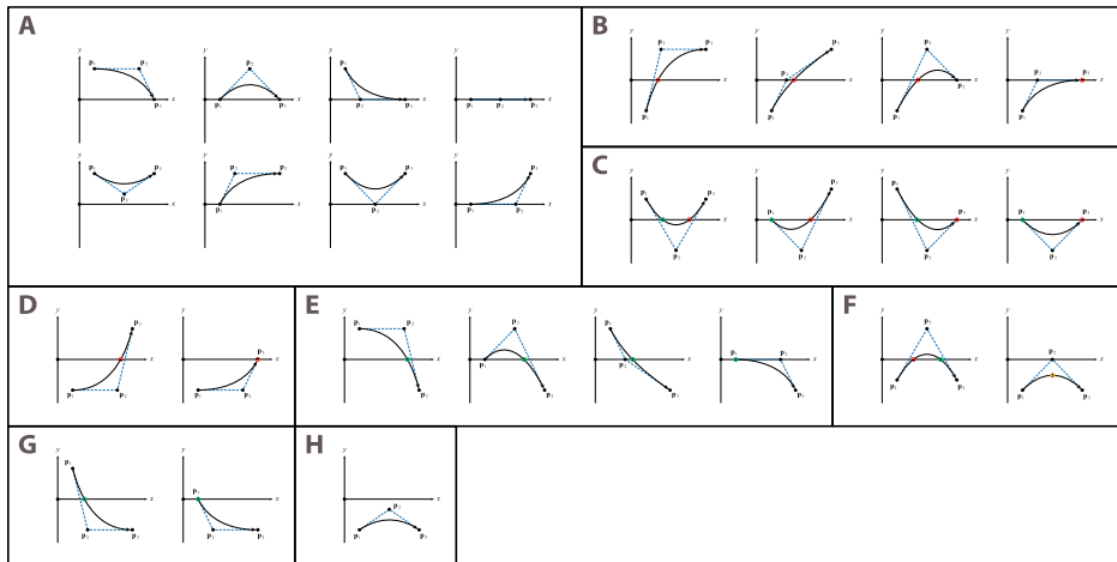


Рисунок 1.19 – Класифікація кривих відносно проміння [12]

Найважливішим фактором швидкої у методах із безпосереднім використанням рівнянь кривих є кількість кривих Безьє, що складають кожен гліф. Дані методи спираються на розбиття кривих у сітку (рис. 1.17), однак, при масштабуванні із збільшенням на весь екран такі гліфи потребують більше часу на растеризацію через те, що доводиться обраховувати багато кривих, які усі знаходяться в одній клітині, яка покриває увесь екран. У розглянутому гліфі на рисунку 1.20 із шрифту Royal цей час у 4 рази більше при збільшенні (на щастя, цей недолік майже не помітно на простих шрифтах як Arial).

Розробник може контролювати, якої складності шрифт використовується, але не положення користувача відносно тексту у 3D сцені, що змушує розробника робити припущення про найімовірніший розмір тексту та оптимізувати наповнення сцени відносно нього.

Slug version 6.4

NVIDIA GeForce GTX 1050 Ti

Resolution: 1920 × 1080

GPU Time: 5.938 ms



Рисунок 1.20 – Швидкодія методу Slug при збільшенні на весь екран

Бібліотека Slug дозволяє додати ефекти до тексту, однак, дублює для цього геометрію гліфів. Для ефекту обведення необхідно завчасно вказати це при імпортуванні шрифту, що призводить до подвоєння розміру даних із зчитаних гліфів. Також недоліком є неможливість використання м'якої тіні із її плавним зникненням та розміром більшим ніж сам гліф. У самій роботі [12] зазначається, що це можна зробити модифікувавши код шейдеру, але у бібліотеці розробленій автором такої можливості немає.

Бібліотека Slug вважається найостаннішим досягненням в області растеризації шрифтів на GPU. На її вебсторінці є список відомих компаній, ігрових студій, якими було придбано ліцензії: Ubisoft, Zenimax Online, Blizzard, id Software, Activision, Adobe, Wargaming тощо.

1.3.5 Порівняння методів

Усі розглянуті методи мають як сильні так і слабкі сторони.

Із огляду методів можемо побачити, що растрівання виконується у два кроки. При появі гліфу на екрані у перший раз, на CPU відбувається його підготовка із векторного представлення у деяке інше оптимізоване представлення, яке буде використовуватися методом на GPU. Крок підготовки виконується один раз та кешується, а крок растеризації виконується кожного фрейму. Тому доцільно віддати перевагу методам, що швидше виконуються у фрагментному шейдері на GPU.

Цікавими властивостями для порівняння є швидкість кроку підготовки на CPU, швидкість растеризації на GPU, чи відбувається відтворення гліфу із абсолютною точністю прямо із рівнянь кривих, популярність методу серед існуючих застосунків, ігрових рушіїв, головні недоліки наявні у методі, чи вільний метод для використання, якість антиаліасингу, легкість створення додаткових ефектів до тексту. Деякі із цих властивостей методів наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння методів растеризації кривих на графічному пристрої

Назва методу	Відносна швидк. (GPU)	Растрування прямо з кривих	Популярність	Головний недолік	Патенти (США)
Попередньо-растровані гліфи	1.0	Ні	+	Усі недоліки растрових зображень	—
SDF	1.2	Ні	+++	Кути контурів стають округленими	—
mSDF	1.6	Ні	++	Можливі артефакти у тонких контурів	—
Loop–Blinn геометричне представлення	2.0	Так		Потребує складний крок тріангуляції	до 2026 (кубічні криві)
Kokojima буфер трафарету	2.0	Так		Потребує буфер трафарету	—
Dobbie	32.0	Так		Числово нестійкий	—
Slug	14.0	Так	+	Відсутність м'яких тіней	до 2038

В результаті порівняння було обрано текстурний mSDF метод. Цей метод майже не поступається у швидкості звичайному атласу попередньо растрованих гліфів, лише треба приділяти увагу розміру поля щоб не втратити тонкі контури. Використовуючи два трикутники можна отримати усі бажані ефекти тіней, обведення, розмиття тощо. Даний метод має гарний компроміс між швидкістю та точністю відтворення гліфу, що підтверджується його популярністю використання в існуючих апаратно прискорених застосунках, які подають 3D сцену у реальному часі.

1.4 Висновок за розділом 1

В даному розділі було описано як перед растеризацію текстові прогони, що мають однаковий набір властивостей форматування, подаються у рушій формування тексту, який надає перелік гліфів та їх позиції. Описано процес растрування та необхідність антиаліасингу.

Визначено, що гліфи у файлах шрифтів складаються із контурів утворених кривими Безьє перших трьох порядків. Визначено правила заповненості контурів.

Було проведено огляд методів растеризації гліфів: текстурні методи, геометричні методи, із безпосереднім використанням рівнянь кривих. Виконано їх порівняно та обрано метод mSDF.

2 МАТЕРІАЛИ І МЕТОДИ

Підготовчим кроком відтворення гліфів за методом SDF є одноразове створення власне самого поля дистанцій зі знаком та збереження у текстуру на GPU.

Зазвичай створення поля виконується на CPU, що є дуже повільним. Так наприклад Qt наводить у статті [17], що для створення 50 полів гліфів може знадобитися 25мс, а це призведе до втрати кадру при 60 FPS. У такому випадку потрібно знати наперед усі можливі гліфи та згенерувати їх завчасно. Для цього Qt створила утиліту Distance Field Generator, яка використовує переваги табличної структури формату OpenType, модифікує файл шрифту додаючи нову таблицю із згенерованими полями.

Однак, коли ми бажаємо дати користувачу можливість встановити власний шрифт, або невідомій кількості конкретних гліфів при відображенні різноманітних CJK ієрогліфів, потрібна швидка генерація полів дистанцій під час виконання застосунку.

2.1 Метод створення SDF на GPU

Швидкої генерації полів можна досягти, створюючи їх прямо там де вони будуть використовуватися – на графічному процесорі. Для цього буде використовуватися паралелізм GPU у растеризації трикутників описаний у статті [24].

Спочатку створюється контур гліфу. Для кожної кривої Безьє у контурі створюються два трикутники, що повністю містять її та сторони яких віддалені від кінців кривої та точок екстремумів (якщо такі є) на заданий радіус створення поля дистанцій (рис. 2.1). Результатом виконання шейдеру є нормалізована відстань до кривої, інтервал якої: 0,5 – точка лежить прямо на кривій та 0,0 – точка поза радіусом створення поля дистанцій.

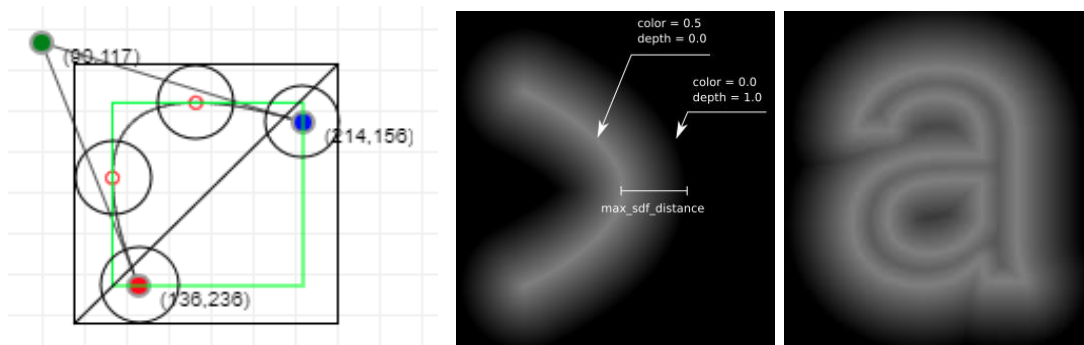


Рисунок 2.1 – Створення поля дистанцій для окремої кривої та кінцевий результат для всіх контурів [24]

Як можна здогадатися поля суміжних кривих перекриваються, тому додатково шейдер зберігає у буфер глибини також відстань. У графічному конвеєрі встановлюється тест глибини таким чином, щоб серед двох дистанцій було обрано найменшу із них (візуально обирається біліший піксель).

Завершальним кроком є коригування дистанцій, що знаходяться всередині гліфу. Для цього у буфері трафарету заповнюються пікселі, що лежать всередині гліфу використовуючи модифікований геометричний метод [21] описаний у підрозділі 1.3.3. Після чого створюється геометрія прямокутника, що покриває частину поля дистанцій, яка може бути всередині гліфу (рис. 2.2). В параметрах конвеєра вмикається тест із буфером трафарету та встановлюється функція змішування кольорів «одиниця мінус значення кольору у буфері».

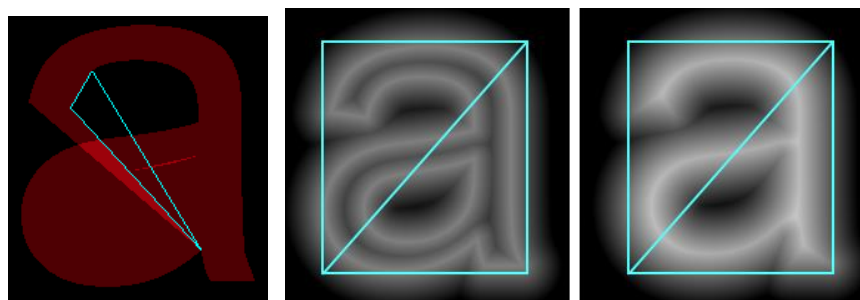


Рисунок 2.2 – Створення буферу трафарету та результат після його застосування [24]

2.2 Модифікація методу для створення mSDF

Метод mSDF виконує декомпозицію гліфу у три канали текстури.

Під час створення поля контуру гліфу, до даних про рівняння кривої, що буде використана шейдером, додається її колір. Вектор кольору складається із трьох компонентів, які можуть бути тільки 0 або 1. Результат шейдеру модифікується шляхом множення відстані на вектор кольору.

Так як ми повинні порівнювати відстані у кожному каналі окремо, то використання буферу глибини не є можливим. Замість цього у графічному конвеєрі встановлюється функція змішування кольорів «максимальне» із них.

Крок коригування дистанцій залишається без змін. Він просто буде виконуватися одночасно над трьома каналами текстури. На цьому моменті створення mSDF є завершеним, алгоритм створення зображено на рисунку 2.3.

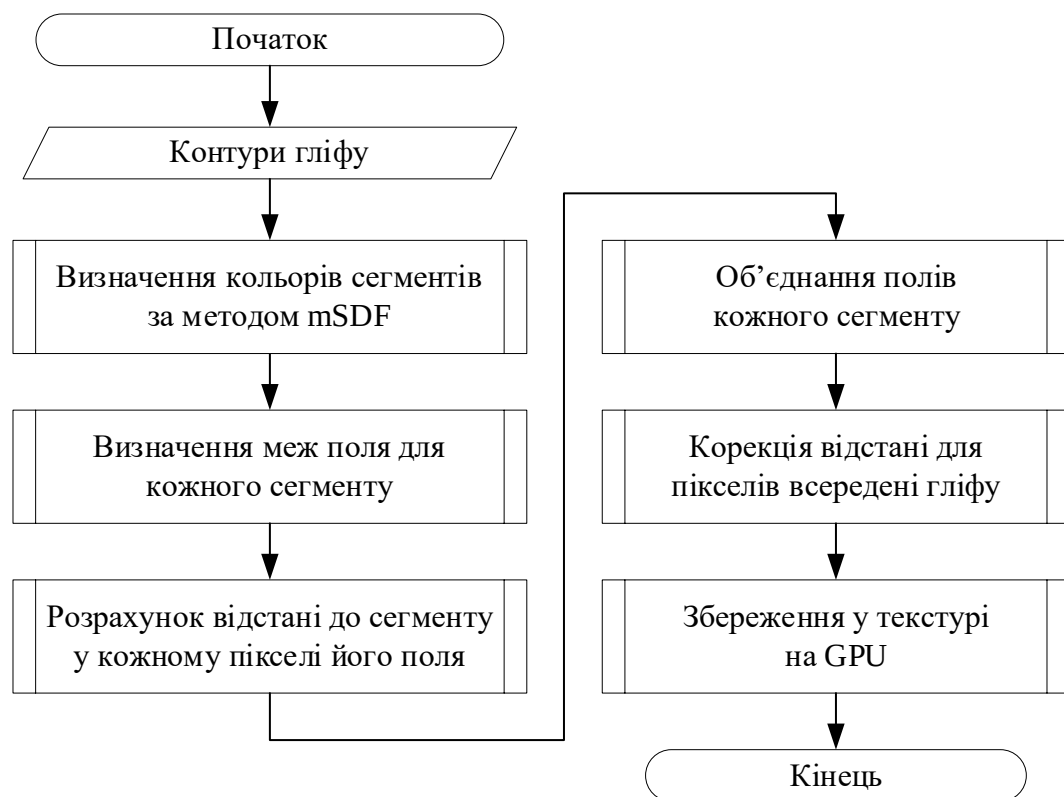


Рисунок 2.3 – Алгоритм створення mSDF поля гліфу

У роботі [15] також вводиться поняття псевдо-дистанцій, які дозволяють отримати ефект обведення тексту із гострими кутами не тільки внутрішніми, а й зовнішніми. Створення поля псевдо-дистанцій не є обов'язковим у методі mSDF для досягнення гострих контурів власне самого гліфу: «на перший погляд може здатися, що така зміна метрики відстані сприяє збереженню гострих кутів, але це не так» [15]. Фактично так звані псевдо-дистанції є кутовим видом стиків (рис. 2.4), а звичайний SDF – округленим, у специфікації SVG [25].



Рисунок 2.4 – Види стиків ліній у SVG [25]

Для створення псевдо-поля дистанцій окремого сегменту необхідно модифікувати визначення його розмірів. Якщо кінець з'єднується із іншою кривою Безьє плавно, то змін не потребується. Якщо кінець утворює із іншою кривою кут, то замість збільшення відстані на радіус, обирається найвіддаленіша точка перетину (точки D та E на рис. 2.5).

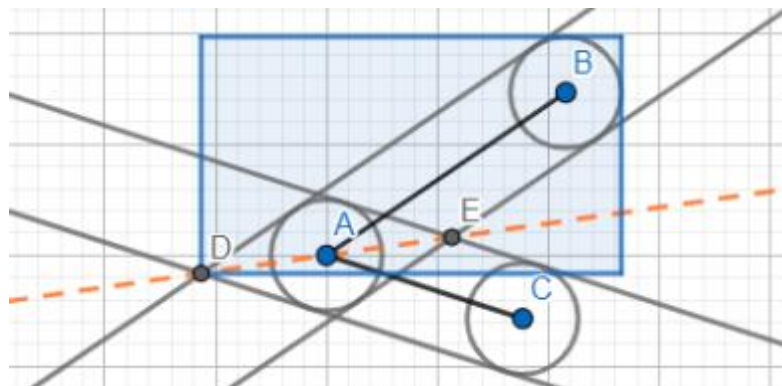


Рисунок 2.5 – Визначення меж поля для сегменту АВ

Результатом виконання шейдеру у межах прямокутника, є псевдо-дистанція до кривої Безьє, яка зберігається у буфер кольору. У роботі [15] описується, що якщо відстані однакові, то ми повинні порівняти їх ортогональність. Так як буфер глибини зберігає одне значення, необхідно знайти шлях збереження двох значень – відстані та ортогональності, а також їх порівняння. Це можна зробити шляхом відведення кожному значенню половину бітів буфера глибини. Старші біти для відстані, а молодші для ортогональності. Таким чином, коли старші біти будуть однакові, то молодші матимуть вплив і буде виконано правильний вибір псевдо-дистанції.

Якщо ми бажаємо створити поле псевдо-дистанцій для mSDF, то необхідно виконувати порівняння окремо для кожного каналу. На жаль, хитрість із встановленням функції змішування кольорів «максимальне» – не можлива. Доведеться виконувати шейдер створення контурів тричі, очищуючи буфер глибини між виконаннями. Можливо кращим рішенням є створення альтернативного методу, який би використовував обчислювальний тип шейдеру.

2.3 Висновок за розділом 2

В даному розділі було описано причини необхідності створення mSDF на GPU такі як дати користувачу можливість використання власних шрифтів, динамічний атлас mSDF, який кешує поля дистанцій гліфів за часом їх останнього використання.

Було описано метод створення SDF на GPU, виконано його модифікацію для створення mSDF, яка буде реалізована у даній роботі. Також було запропоновано метод для створення полів псевдо-дистанцій.

3 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

3.1 Вибір мови програмування

Для апаратно прискорених застосунків, що виконують подання 3D-сцени у реальному часі важливо не допускати пропуску кадрів. Мови програмування із автоматичним управлінням пам'яттю завжди будуть мати недолік недетермінованості збирача сміття. Тому для створення застосунку буде використано мову із ручним управлінням пам'яттю, інакше кажучи мову системного програмування. Із таких мов буде розглянуто C/C++, Rust та Zig.

Мови C та C++ існують довго, поступово еволюціонують відповідно до сучасних потреб. Вихідний код C сумісний із C++, а тому останній унаслідував і велику кількість підводних каменів, які очікують на кожному кроці. Часто можна почути вислів про C++, що мова та її специфікація настільки велика, що насправді ніхто не знає її повністю, навіть члени комітету стандартизації C++. Деякі із недоліків мови та інструментів [26]:

- існує багато варіантів компіляторів для C/C++ та інструментів збірки проекту, що робить обов'язковим їх аналіз та вибір ще до написання перших строк коду;
- вона використовується настільки різноманітно, що в ній набагато більше роздробленості спільноти розробників, стилю коду, ніж в інших мовах;
- мова довго існує і намагається бути максимально сумісною з попередніми версіями, має функції та можливості, яких слід уникати;
- компілятори мають багато опцій, але значення за замовчуванням не дуже гарні, наприклад необхідно встановлювати опції для отримання більшої кількості попереджень;
- арифметичні операції підпорядковуються багатьом неочевидним правилам, які можуть давати несподівані або непереносимі результати між платформами;

- система макросів та порядок підключення заголовних файлів можуть призвести до неочікуваних помилок, результатів виконання;
- незручна концепція визначення та оголошення типів вимагає дублювання коду, більшої кількості файлів та навігації між ними;
- компілятори виводять важко зрозумілі повідомлення для помилок, що використовують шаблони;
- C++ має стандартну бібліотеку, яку важку читати при відладці коду.

З іншого боку, за довгий час створена велика кількість бібліотек для різних структур даних і не тільки. Використання заголовних файлів дозволяє компілятору робити легке розпаралелювання компіляції аж до кроку збірки у виконавчий файл. Швидка компіляція дозволяє робити більше ітерацій тестів та експериментів у коді.

Мова Rust фокусується на безпечній роботі із пам'яттю. Мати впевненість у відсутності помилок через роботу із пам'яттю чудово, але якщо це потребує переписувати програму для задоволення перевірок у компіляторі, призводить до небажаного уповільнення виконання, то така допомога від компілятора вже не є такою привабливою. Окрім цього, перевірки заважають робити швидкі ітерації над чорновим варіантом коду. Застосункам, що подають 3D-сцену у реальному часі важливіше саме швидкість, в них немає критичної логіки як на сервері.

Взаємодія із зовнішнім «небезпечним» світом із нульовими показниками, інтерпретуванням пам'яті тощо, відбувається через інкапсуляцію такого коду в спеціальних «небезпечних блоках». Очікується, що такого коду у проекті буде невелика кількість, легко провести його тестування. Мова Rust відходить від стилю ООП як в C++, натомість має концепцію трейтів, що схожі на інтерфейси, надає перевагу композиції структур замість наслідування.

Інші недоліки мови Rust:

- код важкий для читання через велику кількість пунктуації у синтаксисі мови;
- стандартна бібліотека, яка не дозволяє використовувати власні алокатори пам'яті та викликає панічні ситуації без альтернативного способу їх обробки;
- занадто агресивна спільнота та рух у переписуванні існуючого коду [27];
- повільна компіляція.

Якщо Rust називають сучасною заміною C++, то Zig має на меті стати сучасним C. Zig не має автоматичного управління пам'яттю та безпеки як у Rust, але виправляє багато недоліків присутніх у C/C++. Стандартна бібліотека мови має фокус на контролі алокаторів для структур даних, що потребують виділення динамічної пам'яті. Надається готовий алокатор для відлагодження, який виявляє помилки типу «використання-після-вивільнення» та «повторне-вивільнення». Він автоматично друкує місця витоків пам'яті у командний рядок [28].

Компілятор може створювати фінальні збірки у двох режимах: швидкому та безпечному. При безпечному режимі компілятор додає перевірки доступу у межах масиву, а також не допускає: ділення на нуль, арифметичні дії із переповненням, спроби доступу до значення за нульовим вказівником тощо.

Zig намагається бути максимально простим: відсутні конструктори та деструктори (відсутній прихований потік виконання), відсутнє перевантаження операторів, немає інтерфейсів та наслідування, макросів. Водночас додані концепти, які роблять код меншим і безпечнішим: тип даних для зрізу масиву, обробка помилок у вигляді перерахувань, не обов'язковий тип даних. Компілятор Zig вже майже має завершену інкрементальну компіляцію та зв'язування, яка дозволить підвищити продуктивність розробки у великих проектах.

Недоліками Zig у поточному стані є відсутність офіційного менеджера залежностей (планується додати), можливі зміни у синтаксисі мови (фінальна версія планується не раніше 2024), відсутність інтересу з боку великих компаній (зосереджені на Rust), помилки у самій реалізації компілятора. У таблиці 3.1 наведено порівняння мов.

Таблиця 3.1 – Порівняння мов програмування

Мова	Швидкість ітерацій	Управління залежностями	Популярність, бібліотеки	Повідомлення, відладка	Головний недолік
C/C++	++	+	+++	+	Спадщина сумісності коду
Rust	+	+++	++	+++	Крута крива навчання
Zig	+++	++	+	++	Не випущено стабільної версії

Для розроблюваного застосунку було обрано мову Zig. Головними причинами для чого є крута крива навчання у Rust, яка на початку призводить до боротьби та задоволення вимог самого компілятора. C/C++ не підходять через свою спадщину сумісності, технічних боргів, підводних каменів.

3.2 Вибір графічного API

Для взаємодії із GPU необхідно використовувати графічне API, яке надається ОС та реалізується у відеодрайвері. Альтернативно можна використовувати бібліотеки, що надають єдину абстракцію над декількома API.

Багато років єдиними графічними API були Direct3D (до версії 12) та OpenGL (та різні його підмножини). Дані інтерфейси були зручні до використання, але уся складність покладалась на драйвер. Сучасні GPU дуже

швидкі, але зміна стану графічного конвеєру шляхом виклику великої кількості окремих функцій призводить до завантаженості CPU, але не повної на GPU. Таким чином з'явилося нове покоління графічних API: Vulkan, Metal, Direct3D12. Дані інтерфейси наближені до того, що відбувається на апаратному рівні у GPU, знижують розмір драйверу. Однак ціна, яка платиться за ці переваги, полягає в тому, що доводиться працювати зі значно складнішим API, фактично логіка із драйверу перекладається на розробників застосунків [29].

Виконання відлагодження коду у шейдері є складним, аналогічно багатопоточним застосункам. Найпримітивніший спосіб – вивід значення у командний рядок, не доступний на GPU. Тому часто застосовується спосіб кодування кольором результатів. Виконати огляд використовуваних ресурсів шейдером, перелік команд відмалювання та параметри графічного конвеєру можна у програмах RenderDoc (рис. 3.1), NVIDIA Nsight Graphics, PIX on Windows. На жаль, використати ці інструменти із браузером не можливо, через особливості безпеки останніх, ізоляції процесу драйвера, відмалювання сцени у позаекранний буфер.

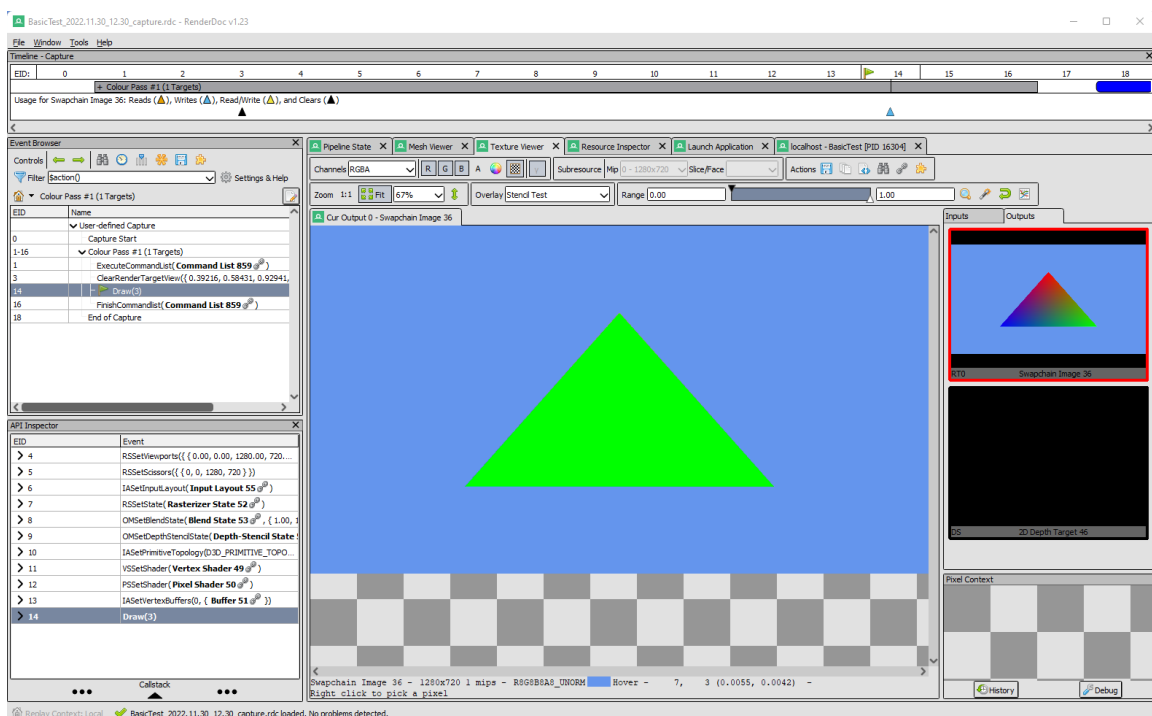


Рисунок 3.1 – Відладка кадру у RenderDoc [30]

У таблиці 3.2 наведено API, які доступні на різних платформах, їх мови шейдерів, легкість використання та те наскільки вони гнучкі, дозволяють використати усі можливості сучасних GPU.

Таблиця 3.2 – Порівняння графічних API

API	Мова шейдерів	Платформа	Легкість використ.	Можливості GPU
OpenGL	GLSL	Android, Linux, Windows	++++	++
Vulkan			+	++++
Metal	MSL	macOS, iOS	+++	+++
Direct3D11	HLSL	Windows	++++	++
Direct3D12			++	++++
WebGL2	GLSL	Браузер	++++	+
WebGPU	WGSL		+++	+++

Для розробки буде використано API WebGPU, яке доступне у браузерах. Дане API було розроблено з нуля на заміну WebGL та має на меті використовувати нове покоління графічних інтерфейсів на кожній із платформ підтримуваних браузером. WebGPU у порівнянні з WebGL вирішує проблеми дорогого виклику команд відмалювання та відсутність обчислювальних шейдерів [31].

WebGPU було обрано через те, що воно:

- має інтерфейс, який дозволяє краще розуміти як працюють сучасні GPU;
- є абстракцією, яка надає доступ лише к тим можливостям, що сумісні між доступними API на різних платформах;
- звільняє від написання великої кількості однотипного коду у новому поколінні графічних API.

Так як даний інтерфейс доступний лише у браузері, то застосунок на мові Zig буде скомпільовано у модуль WebAssembly. Модуль буде

підключено на вебсторінку. WebAssembly (WASM) – це двійковий формат інструкцій для стекової віртуальної машини [32]. Він надає компроміс між швидкістю та безпекою виконання невідомого коду, підтримкою декількох архітектур у браузері. Швидке виконання досягається шляхом ряду припущень щодо середовища виконання та сучасних CPU архітектур.

3.3 Опис архітектури застосунку

Застосунок повинен вміти працювати із файлами шрифтів надаваними користувачем, створювати та кешувати MSDF гліфи під час виконання, використовувати рушій формування тексту для позиціювання гліфів, надавати інтерфейс для зміни тексту та його властивостей оформлення. На рис. 3.2 зображено функціональний розподіл між компонентами.

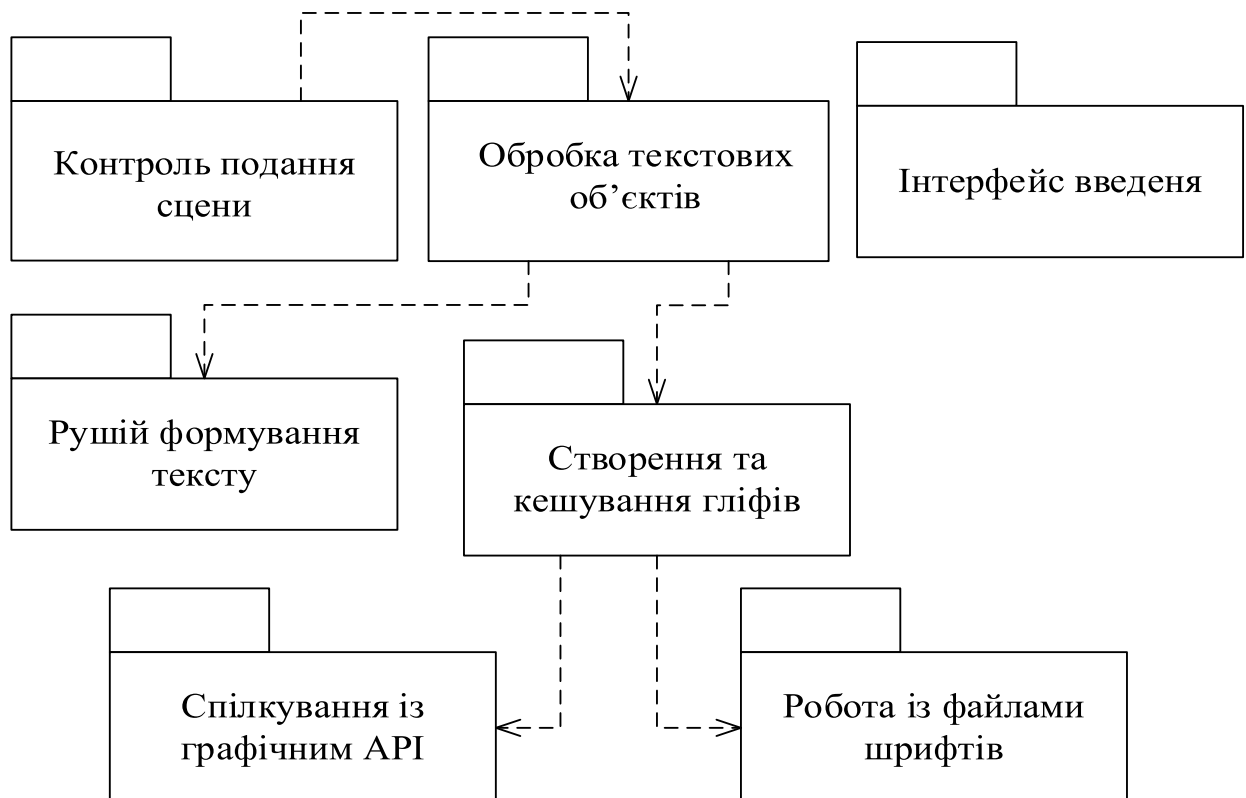


Рисунок 3.2 – Функціональний розподіл між компонентами

Класичним підходом до організації коду об'єктів сцени є наслідування класів. Усі об'єкти наслідують клас, що визначає віртуальний метод Update. У головному циклі застосунку виконується їх ітерація та виклик цього перевантаженого методу на кожному із них. Іншим шаблоном проектування, який набуває популярності є ECS (Entity-Component-System).

ECS складається із розуміння його трьох складових. Сутності в основному використовуються для надання унікального ідентифікатора, є своєрідним кореневим об'єктом, який об'єднує в собі набір компонентів. Компоненти – це не що інше, як об'єкти-контейнери, які не володіють ніякою складною логікою. В ідеалі це прості об'єкти даних (POD). Кожен тип компонента може бути приєднаний до сутності, щоб забезпечити певну властивість. Системи є «закритими середовищами». Тобто, вони не володіють ні об'єктами, ні компонентами. Доступ до них здійснюється через незалежні об'єкти-менеджери, які, в свою чергу, опікуються життєвим циклом сутностей та компонентів [33].

Використання архітектури ECS має переваги:

- відсутність проблем наслідування об'єктів в ООП-стилі;
- дизайн орієнтований на данні, дружній до кешу CPU;
- перевикористання систем простим додаванням компоненту до сутності;
- ізоляція систем для тестування;
- більш легко виконати розпаралелювання;
- спрощене збереження та завантаження стану сцени.

У застосунку не буде реалізовано повноцінну ECS архітектуру, однак наступні системи будуть виконані для легкої інтеграції у нього:

- подання сцени із компоненту камери;
- подання текстових прогонів із відповідних компонентів.

Користувач бачить на екрані 3D-сцену, зверху якої зображено інтерфейс для встановлення властивостей тексту:

- розмір тексту, який визначається як висота великих літер в умовних одиницях 3D-простору;
- колір тексту;
- відстань між літерами;
- стиль шрифту;
- положення та нахил тексту у 3D-сцені;
- м'які тіні із радіусом та зміщенням відносно гліфу;
- обведення тексту, його колір та відстань;
- вирівнювання відносно базової точки положення.

Окрім кешування MSDF гліфів у атлас, система обробки текстових об'єктів не повинна виконувати формування тексту у кожному кадрі. На рис. 3.3 зображено алгоритм відмалювання тексту у кадрі, що виконується для кожного компоненту системою подання текстових прогонів.

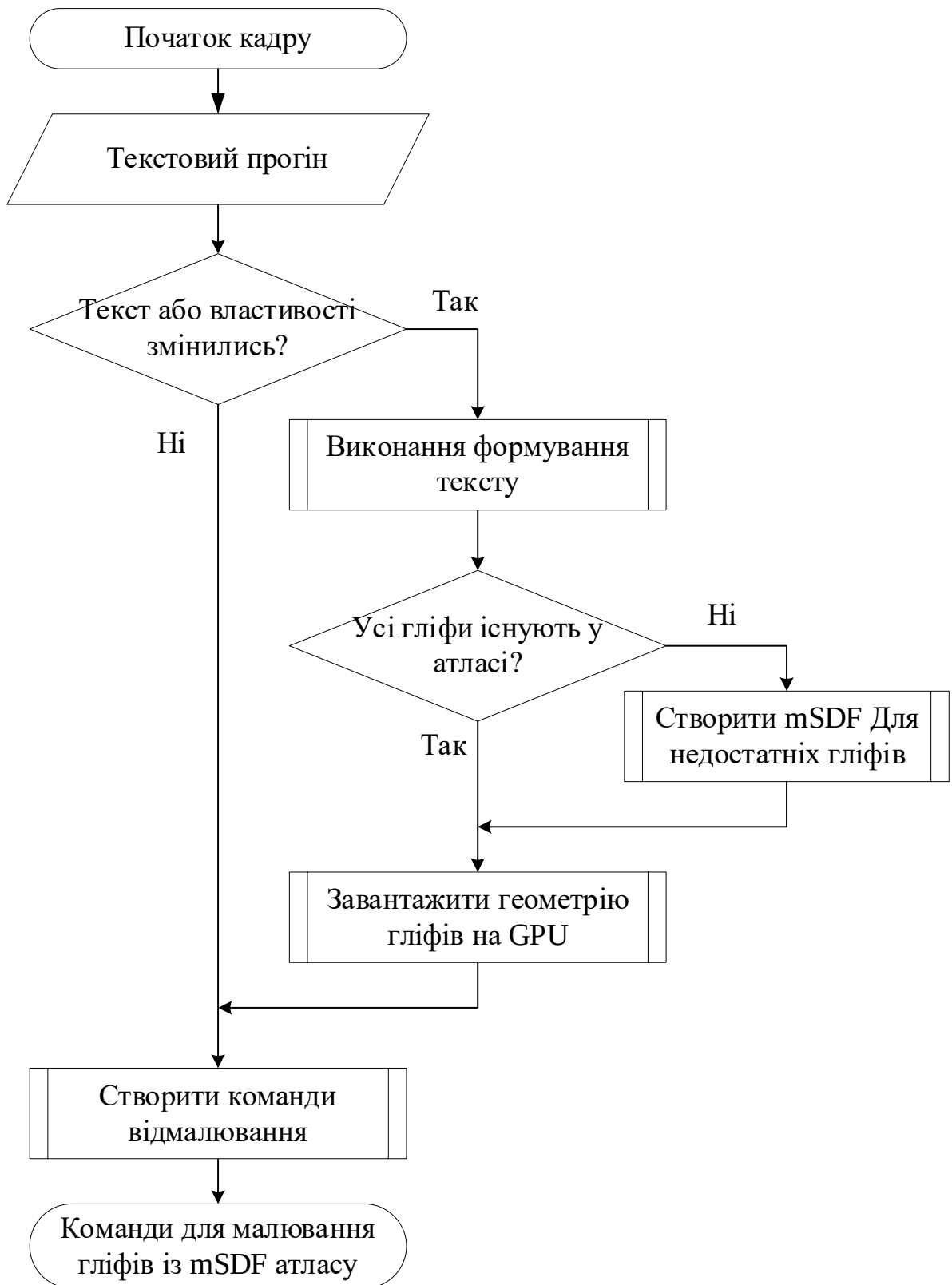


Рисунок 3.3 – Алгоритм відмалювання тексту у кадрі

3.4 Висновок за розділом 3

В даному розділі було розглянуто мови системного програмування, які підходять для створення застосунків у реальному часі. Було обрано та обґрунтовано вибір мови Zig. Також було розглянуто сучасні графічні API та обрано для цього WebGPU, яке доступно у браузері.

Окрім цього було визначено функціональні компоненти із яких складається застосунок, їх взаємозв'язки. Розглянуто архітектуру ECS на яку буде орієнтуватися створюваний код. Було наведено алгоритм для відмалювання тексту у кадрі, який виконується окремою системою для кожного компоненту текстових прогонів.

4 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

4.1 Опис структури застосунку та головного циклу

Файли застосунку згруповані у директорії відповідно до їх зв'язування та зображені на рис. 4.1. Проєкт було створено за допомогою команди `zig init-lib`. У файлі `build.zig` визначено інструкції побудови для компілятора, визначено шлях для підключення сторонніх бібліотек. Під час побудови створюється WASM модуль, який підключається на вебсторінку.

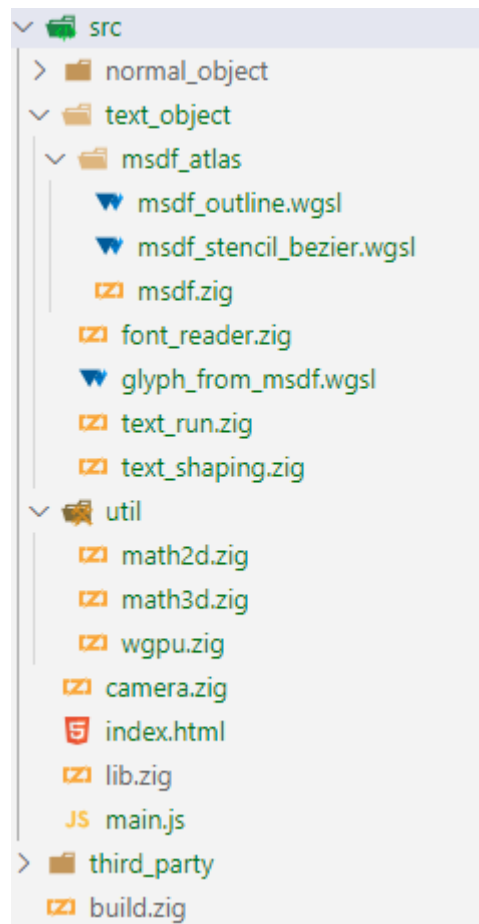


Рисунок 4.1 – Файлова структура застосунку у редакторі VS Code

У файлі `index.html` підключається файл стилів та ESM модуль `main.js`. Також у ньому міститься розмітка для полотна та форми зміни властивостей тексту. Форма достатньо проста, тому ніяких фреймворків використано не

було. Кожному елементу вводу встановлено атрибут `id`. Під час натиснення кнопки внесення змін збираються параметри форми та викликається функція у WASM модулі, яка змінює текстовий прогін у «забруднений» стан.

Для надання єдиного стилю інтерфейсу застосунку, єдиного ритму відступів використано CSS фреймворк `tailwindcss`. Цей фреймворк побудований на принципі: один клас визначає лише одну-дві CSS властивості. Декілька таких класів на одному елементі дозволяють отримати бажаний результат без написання власних стилів, швидше експериментувати, не витрачати час на вигадкування унікальних назв класів.

У файлі `main.js` міститься код, який використовує API `WebGPU` для вибору фізичного адаптеру `GPUAdapter` та логічного пристрою, зв'язування його із контекстом полотна у розмітці. Клас `IndexedJsObjects` зберігає об'єкти світу JavaScript аби їх можна було передати у модуль за допомогою `u32` індексу. Додано оброблювачі подій до зміни розміру сторінки, які необхідні для зміни розміру текстур у які виконується растеризація. Зміна розмірів також враховує монітори із щільним розміщенням пікселів завдяки `window.devicePixelRatio`. Під час ініціалізації виконується виклик `WebAssembly.instantiateStreaming` для створення модулю. У властивість `env` параметрів функції передається словник із назвами функціями та їх визначенням. Ці функції можна буде викликати із модулю:

- `pariGpuCreateBuffer` для створення різних типів буферів у виділеній пам'яті на GPU;
- `pariGpuCreateTexture` для створення текстур у пам'яті;
- `pariGpuCreateBindGroupLayout` для створення макету прив'язки;
- `pariGpuCreatePipelineLayout` для створення макету графічного конвеєру;
- `pariGpuCreateRenderPipeline` для створення графічного конвеєру;
- `pariGpuCreateBindGroup` для створення групи прив'язки;
- `pariGpuWriteBuffer` для копіювання даних до буферу на GPU;
- `pariGpuUnmap` для відв'язування буферу;

- `rapigpuSetPipeline` для встановлення поточного графічного конвеєру;
- `rapigpuSetVertexBuffer` для встановлення поточного буферу вершин;
- `rapigpuSetBindGroup` для встановлення поточної групи прив'язки;
- `rapigpuDraw` для відправлення команди малювання.

Як можна побачити усі надані функції зосереджені на роботі із API WebGPU, яке доступне тільки із JavaScript. Експортовані функції із модуля викликаються через `instance.exports`. Мова JavaScript не є типізованою, тому визначити типи цих функцій не потрібно. Функції, що експортуються знаходяться у різних `zig` файлах:

- `setFont` для зміни файлу шрифту;
- `initialize` для виконання єдиноразової ініціалізації;
- `handleResize` для виконання необхідних операцій під час зміну розміру;
- `handleMovement` для простого управління камерою у сцені;
- `tick` для відмалювання наступного кадру;
- `setText` для оновлення властивостей тексту.

У деякому сенсі оновлення даних є «лінивим». Під час виникнення через подію зміни тексту чи переміщення змінюється лише деякі дані, у той час як залежні дані від них обраховуються під час `tick` – відмалювання наступного кадру. Традиційний нескінчений головний цикл рушія реалізувати не можливо, тому він виконується шляхом зациклення функції у `requestAnimationFrame`. Відповідно до документації ця функція викликається після усіх функцій та перед наступним перемалюванням у браузері. Лістинг коду наведено нижче:

```
function mainLoop () {
    if (needResize) {
        handleResize(needResize.width, needResize.height);
```

```

    }
    const cmdBuffer = device.createCommandEncoder();
    const renderPass = cmdBuffer.beginRenderPass(...);
    instance.exports.tick();
    renderPass.end();
    device.queue.submit([cmdBuffer.finish()]);
    requestAnimationFrame(mainLoop);
  }
  requestAnimationFrame(mainLoop);

```

У випадках коли необхідно декодувати структури із пам'яті WASM, використовуються класи типізованої пам'яті Uint32Array та Uint8Array.

У файлі lib.zig знаходиться головна функція модулю – tick, яка викликає системи подання сцени з камери та систему обробки компонентів текстових прогонів.

У файлі camera.zig знаходиться реалізація простої камери подання 3D-сцени у перспективі, матриця трансформацій яка створюється із її властивостей. Також у ньому знаходиться експортована функція handleMovement для переміщення по сцені.

У файлі math3d.zig знаходяться класи для роботи із математичними примітивами: Vector3, Quaternion, Matrix4.

У файлі math2d.zig знаходяться класи для роботи із математичними примітивами: Vector2, Matrix3, BezierQuad, BoundingBox. Клас BezierQuad використовується для квадратичних кривих Безьє та кривих першого порядку, які представлені у вигляді однакових координат першої та другої точки квадратичної прямої. У класі BezierQuad визначено метод boundingBox для отримання меж кривої із урахуванням екстремумів.

У файлі wgpu.zig виконано об'явлення функцій із ключовим словом extern, які було надано модулю та описано вище у файлі main.js. Також визначено псевдонім типу JsRef для посилання на об'єкти, що знаходяться на стороні JavaScript.

У файлі font_reader.zig визначено клас Font, який має єдиний метод getGlyphById для отримання векторного гліфу за його номером. Він повертає

структуру типу Glyph, що має динамічний масив контурів, які у свою чергу мають масив кривих BezierQuad. Для зчитування контурів було використано таблиці head, loca, glyf у файлі шрифту, відповідно до стандарту OpenType [7]. Під час зчитування можуть виникати помилки, які винесено у перерахування FontError. У разі відсутності гліфу повертається спеціальний notdef гліф, який гарантовано повинен бути у шрифті.

У файлі text_shaping.zig визначено клас TextShaper. Виконання коректного формування тексту є окремою складною проблемою, тому було вирішено використовувати бібліотеку. Було розглянуто Uniscribe який використовується на Windows, CoreText який використовується на Apple пристроях та Harfbuzz який використовується на Linux та Android. Так як Uniscribe, CoreText прив'язані до ОС та доступні тільки із звичайних застосунків їх використання не є можливим. Так як браузер не мають API для формування тексту, то єдиним варіантом залишається Harfbuzz. Бібліотеку додано до директорії third_party. Клас TextShaper це невелика обгортка навколо цієї бібліотеки, яка ініціалізує необхідні буфери та структури. Взаємодіяти із кодом написаним на С дуже просто завдяки мові Zig. Немає необхідності дублювати визначення функцій та структур, а достатньо скористатися спеціальними конструкціями:

```
const hb = @cImport({
    @cInclude("hb.h");
});
```

Клас TextShaper має метод shape, який приймає рядок у кодуванні UTF-8, передає його у рушій формування тексту. Метод повертає структуру Glyph, яка містить номер векторного гліфу у шрифті, а також його розміри, позицію серед інших гліфів.

У файлі text_run.zig міститься компонент-структура TextRun, яка має властивості описані у таблиці 4.1.

Таблиця 4.1 – Опис структури текстового прогону (TextRun)

Назва поля	Тип	Опис
text	ArrayList(u8)	Октети рядку у UTF-8
isDirty	bool	Чи потребує формування тексту
font	*Font	Показчик на шрифт
color	f32[4]	Колір тексту
capSize	f32	Розмір тексту (велика літера)
align	TextAlign	Вирівнювання відносно якоря
outlineColor	f32[4]	Колір обведення
shadowColor	f32[4]	Колір м'якої тіні
outlineSize	f32	Розмір обведення
shadowSize	f32	Розмір тіні
shadowOffset	Vector2	Зміщення тіні
anchor	Vector3	Позиція якоря
rotation	Quaternion	Обертання рядку
vertCount	u32	Кількість вершин
vertBuffer	JsRef	Буфер із вершинами

Система обробки текстових прогонів TextSystem отримує масив усіх TextRun у сцені та створює команди для їх малювання. Якщо флаг isDirty неактивний, то функцією встановлюється графічний конвеєр та додається команда відмалювання трикутників, які не змінилися у буфері vertBuffer. Якщо ж флаг isDirty встановлено, то у рушій формування подається текстовий рядок. Для отриманих номерів гліфів відбувається пошук у кеші – mSDF атласі гліфів. Отримані координати гліфу у текстурі атласу та позиції, отримані від процесу формування, зберігаються у шістьох вершинах для кожного із гліфів. Ці вершини копіюються у буфер на GPU та їх кількість зберігається у поле vertCount. Вимикається флаг isDirty та виконується їх відмалювання.

Файл шейдеру glyph_from_msdf.wgsl містить функції для вершин та фрагментів, призначений для використання разом із графічним конвеєром малювання гліфів із mSDF атласу. У таблиці 4.2 наведено змінні, які використовуються у різних етапах виконання шейдеру.

Таблиця 4.2 – Опис змінних у шейдері (glyph_from_msdf.wgsl)

Назва поля	Тип	Збереження	Опис
viewProj	mat4x4<f32>	uniform	Матриця камери
bisampler	sampler	binding	Білінійний семплер
atlas	texture_2d<f32>	binding	Атлас mSDF
sdfRadius	f32	uniform	Радіус полів у атласі
color	vec4f	uniform	Колір тексту
effectColor	vec4f	uniform	Колір ефекту
effectDist	f32	uniform	Розмір ефекту
effectKind	u32	uniform	Тип ефекту
position	vec3f	vertex	Позиція вершини у сцені
uv	vec2f	vertex/frag.	Координати у атласі
position	vec4f	fragment	Позиція фрагменту

Нижче наведено лістинг коду, який виконується на етапі фрагментного шейдеру, він виконує семплювання багатоканального поля із текстури та перетворює у відстань до контуру відповідно до методу у роботі [15]:

```
var msd = textureSample(atlas, bisampler, in.uv).rgb;
var sd = median(msd.r, msd.g, msd.b);
```

Окрім цього у шейдері виконується антиаліасинг, застосовуються ефекти до тексту.

У директорії normal_object міститься реалізація управління графічним конвеєром та шейдер для відображення звичайних суцільних 3D-об'єктів у сцені, що не є цікавим у контексті даної роботи.

4.2 Опис реалізації формування mSDF атласу

Система обробки текстових прогонів звертається до кешу – mSDF атласу гліфів. Управління атласом реалізовано у файлі msdf.zig у директорії

msdf_atlas, в якому міститься клас MsdfGlyphCache. Поля класу описані у таблиці 4.3.

Таблиця 4.3 – Опис полів у класі (MsdfGlyphCache)

Назва поля	Тип	Опис
font	*Font	Показчик на шрифт
cache	ArrayHashMap(u32, BoundingBox)	Геш-таблиця номера гліфу у шрифті до його позиції у атласі
outlineBuffer	JsRef	Показчик на буфер для збереження геометрії контурів
segmentsBuffer	JsRef	Показчик на буфер для збереження рівнянь Безьє
outlinePipeline	JsRef	Показчик на графічний конвеєр для контурів
stencilBuffer	JsRef	Показчик на буфер для збереження геометрії трафарету
stencilPipeline	JsRef	Показчик на графічний конвеєр для трафарету

MsdfGlyphCache має один метод, який повертає позицію гліфу у текстурі атласа за його номером. Якщо mSDF для гліфу ще не було створено, то створюються команди для GPU. Спочатку отримуються контури гліфу із шрифту, потім створюється необхідна геометрія, відповідно методу описаному у розділі 2, для кожної кривої у контурах та для буферу трафарету. Радіус створення полів та значення допуску паралельності з'єднань кривих жорстко закодовані.

При створенні геометрії було приділено увагу можливим артефактам через правила растеризації на GPU. Якщо геометрія не покриває центр фрагменту, то він просто ігнорується (рис. 4.2). Аби не втрати дистанції, уникнути артефактів, розміри прямокутників було збільшено на 0.5 пікселя.

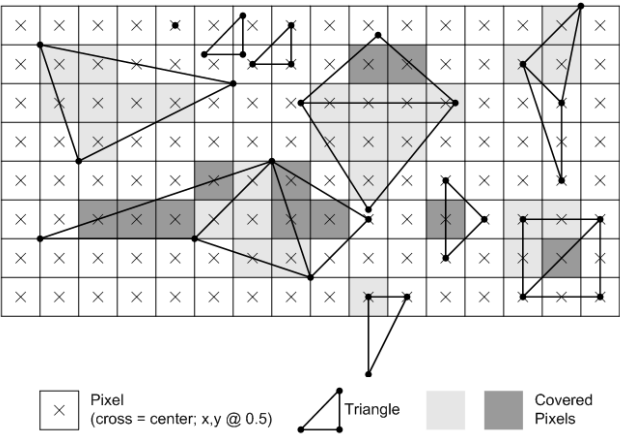


Рисунок 4.2 – Правила растеризації трикутників [34]

При створенні геометрії одночасно обраховується колір сегментів відповідно до методу mSDF:

```
var color = EdgeColor.init();
for (contour.segments) |segment, i| {
    if (i != 0 && !isSegmentsContinuous(
        contour.segments[i-1], segment)) {
        color.flip();
    }
}
```

Файл шейдеру msdf_outline.wgsl містить функції для вершин та фрагментів, призначений для обрахування відстані у фрагменті до кривих. У таблиці 4.4 наведено змінні, які використовуються у різних етапах виконання шейдеру.

Таблиця 4.4 – Опис змінних у шейдері (msdf_outline.wgsl)

Назва поля	Тип	Збереження	Опис
transform	mat3x3<f32>	uniform	Трансформація
sdfRadius	f32	uniform	Радіус полів у атласі
segments	array<BezSegment>	storage	Криві контурів
position	vec2f	vertex/frag.	Позиція вершини у атласі
uv	vec2f	vertex/frag.	Позиція у координатах гліфу
segIdx	u32	vertex/frag.	Індекс сегменту у буфері

Кожна структура BezSegment копіюється у пам'ять GPU та містить колір сегменту (vec3f) та контрольні точки кривої Безьє (array<vec2f, 3>). Для економії пам'яті, усі криві зберігається в окремому буфері, а до вершин додається індекс кривої у ньому. Відстань до кривої знаходиться відповідно до формул у роботі [15]. Для знаходження відстані також необхідно розв'язати кубічне рівняння. Функцію для цього реалізовано відповідно до статті [35]. Мови шейдерів дозволяють зручні операції над векторами, так наприклад випадок із трьома коренями має лаконічну форму:

```
var theta = acos(r / sqrt(-q3));
const k = float3(0, 1, 2) * TAU / 3;
roots.x = 2*sqrt(-q)*cos(theta/3 + k);
```

Однією із виконаних модифікацій кроків методу у розділі 2 є результат шейдеру:

```
var dist = distToBezier(...);
dist = min(dist / sdfRadius, 1.0);
var cov = 0.5 - dist / 2;
if (cov == 0.0) {
    discard;
}
return vec4f(in.color * cov, 1.0);
```

Іншим шейдером, який використовується для створення буферу трафарету є msdf_stencil_bezier.wgsl. У таблиці 4.5 наведено змінні, які використовуються у різних етапах виконання шейдеру.

Таблиця 4.5 – Опис змінних у шейдері (msdf_stencil_bezier.wgsl)

Назва поля	Тип	Збереження	Опис
transform	mat3x3<f32>	uniform	Трансформація
position	vec2f	vertex	Позиція вершини у атласі
uv	vec2f	vertex/frag.	Канонічні координати

Даний шейдер є найпростішим і слугує для тестування належності центру фрагменту одній зі сторін відносно кривої Безьє, за формулою у [20], лістинг коду:

```
if (in.uv.x * in.uv.x < in.uv.y) {  
    discard;  
} else {  
    return vec4f(1);  
}
```

Після створення команд малювання гліфу у атласі, до геш-таблиці заноситься його позиція. Ця ж сама позиція повертається як результат функції у класі MsdfGlyphCache.

4.3 Висновок за розділом 4

У даному розділі було наведено структуру застосунку, описано призначення кожного файлу. Описано функції, які імпортуються та експортуються із WASM модуля застосунку. Розглянуто реалізацію головного циклу програми. Обґрунтовано вибір рушія формування тексту. Описано структури для текстового прогону.

Виконано опис файлів для генерації mSDF атласу, описано використані змінні у шейдерах, буфери. Наведено лістинги коду та виконані модифікації цікаві у контексті даної роботи.

5 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

5.1 Призначення програми

Програмний продукт призначений для експериментування та дослідження можливостей растеризації тексту методом mSDF в апаратно прискорених застосунках. Продукт надає 3D-сцену та камеру для експериментів. Також користувач може змінювати текст та властивості його форматування в інтерфейсі застосунку.

Даний продукт автоматично завантажується при відкритті вебсторінки застосунку або може бути встановлений та використовувати системний компонент браузеру у ОС.

5.2 Робота користувача із застосунком

Після відкриття застосунку, одразу доступна форма редагування властивостей тексту.

Користувач може пересувати камеру за допомогою стрілок на клавіатурі, нахиляти її клавішами Q та E.

У формі доступні наступні поля:

- текст;
- колір тексту;
- розмір тексту (в умовних одиницях сцени);
- файл шрифту;
- вирівнювання відносно якоря;
- ефект тіні (радіус, колір, зміщення);
- ефект обведення (розмір, колір);
- позиція якоря тексту у сцені та його обертання.

5.3 Експериментальне дослідження програми

На рисунку 5.1 зображено простий текст без ефектів, розміром 1, вирівняним по центру. За замовчуванням використовується шрифт Arial.

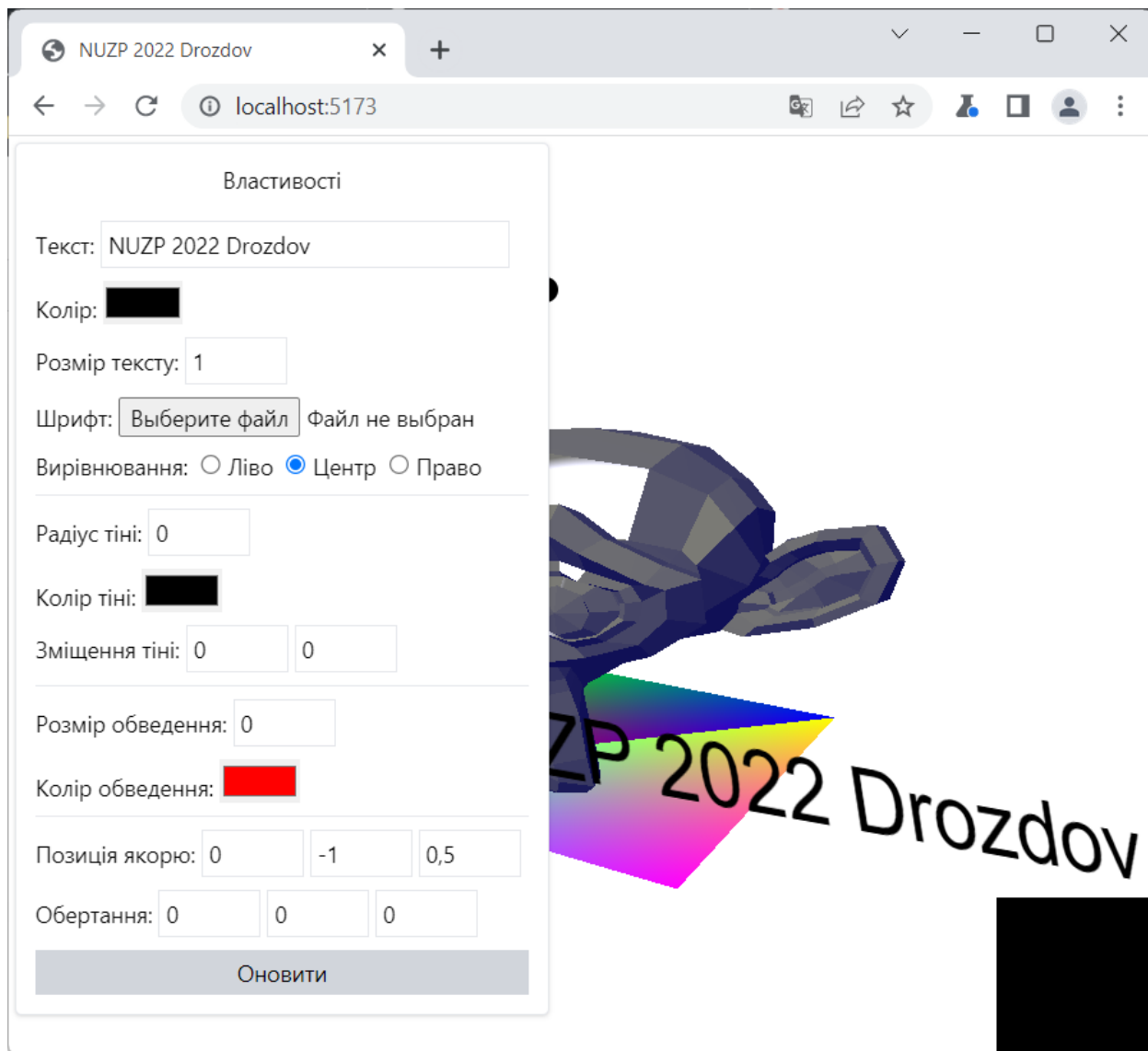


Рисунок 5.1 – Відображення звичайного тексту

На рисунку 5.2 зображено текст із ефектом тіні жовтого кольору, радіусом 0,5, зміщенням по X та Y на 0,2.

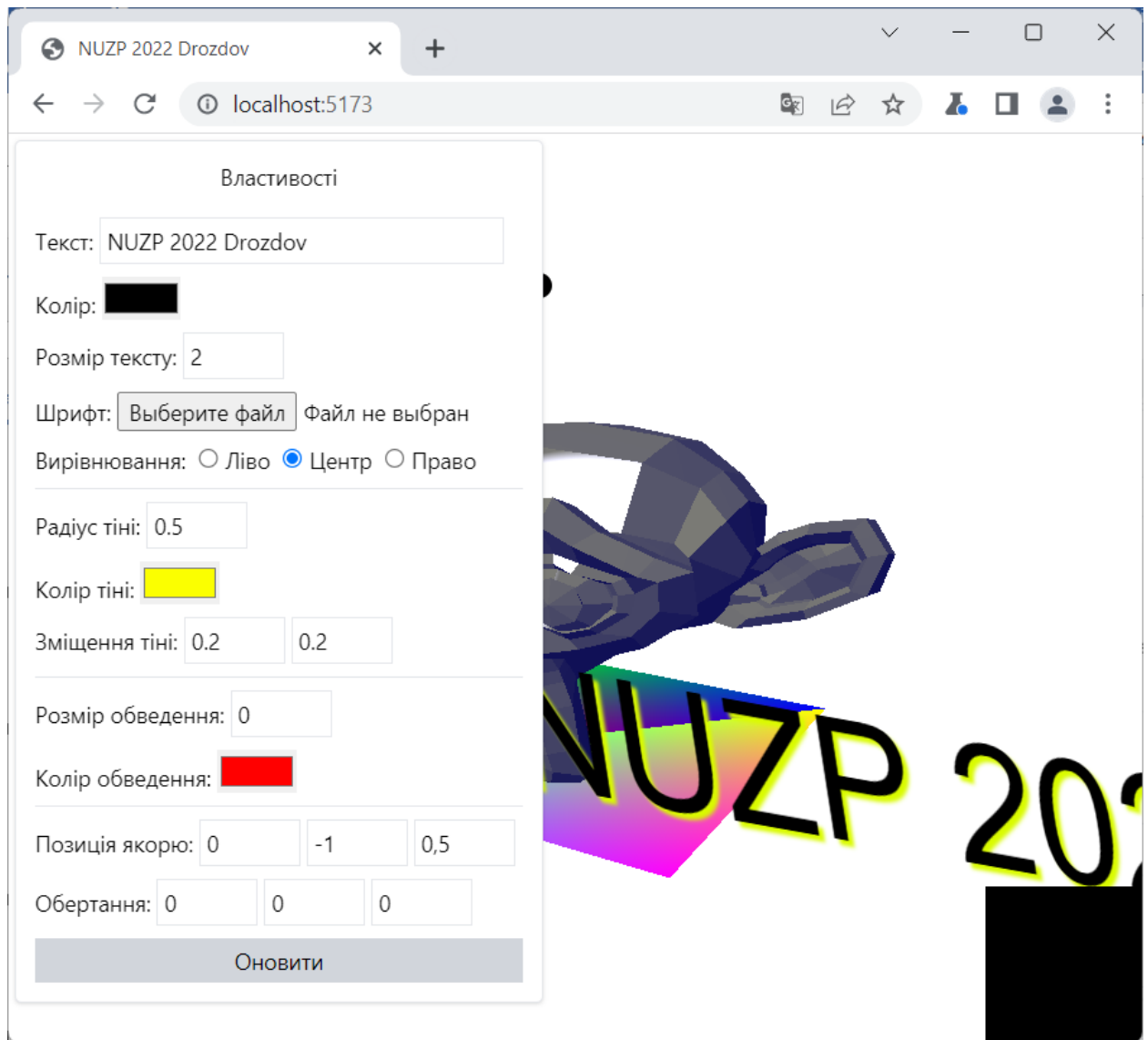


Рисунок 5.2 – Текст із ефектом тіні

На рисунку 5.3 зображено текст із ефектом обведення червоного кольору, розміром 0,4.

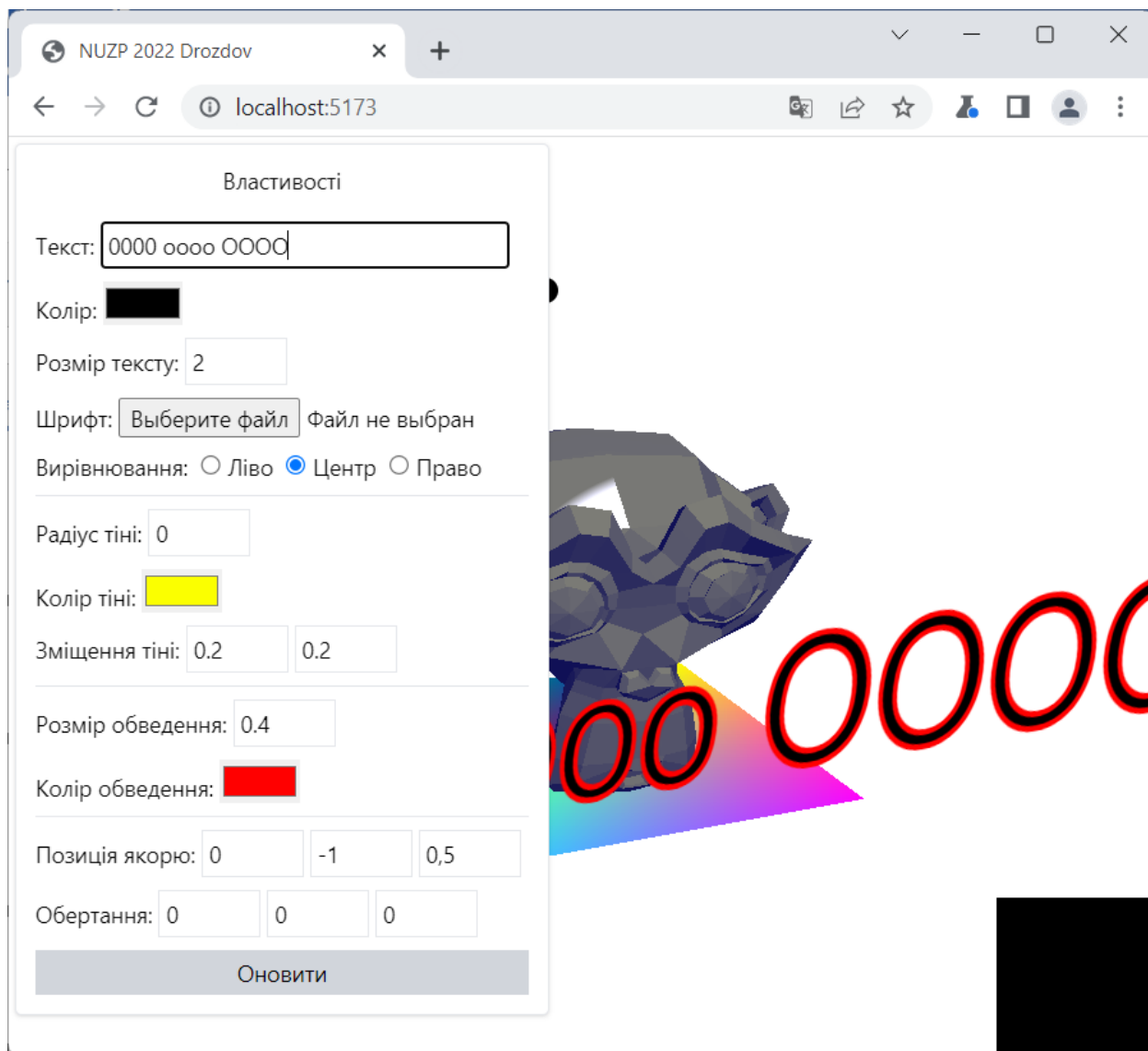


Рисунок 5.3 – Текст із ефектом обведення

Завдяки використанню методу mSDF растеризовані гліфи мають гострі кути контурів. Окрім цього у шейдері виконується антиаліасинг, який можемо побачити на збільшеному зображенні на рис. 5.4.



Рисунок 5.4 – Растрований текст із антиаліасингом

Модифікація методу створення SDF для створення mSDF має аналогічну швидкодію, адже було змінено лише декілька параметрів. На жаль, API WebGPU все ще є експериментальним і не надає можливостей чіткого визначення витраченого часу команд відмалювання на GPU. Тому, аби хоч якось упевнитись у швидкодії використано звичайний профайлер в інструментах розробника. На рис. 5.5 можемо побачити, що на відмалювання витрачається 0,6 мс на відеоадаптері GeForce GTX 1050 Ti.

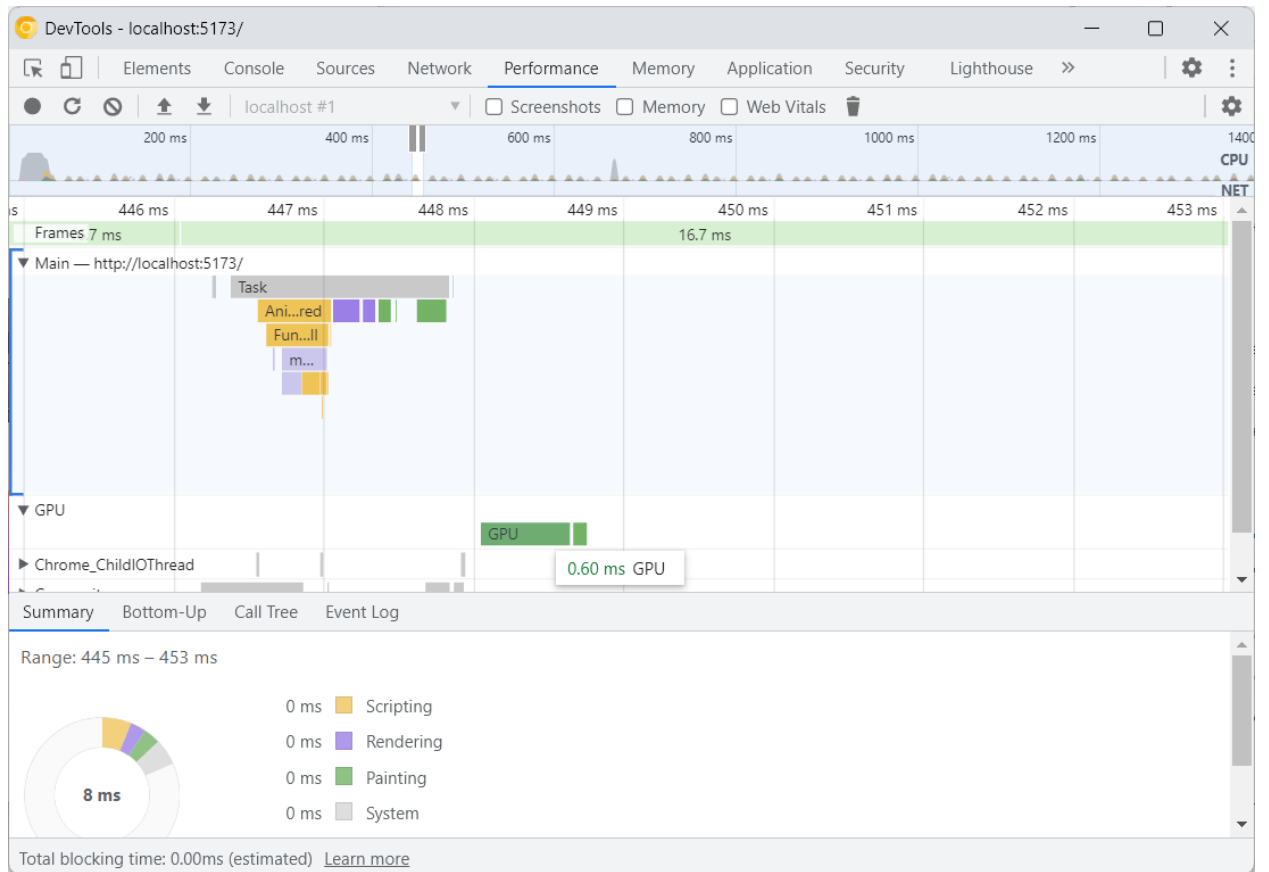


Рисунок 5.5 – Результат профілювання

ВИСНОВКИ

У роботі було досліджено поняття текстових прогонів, формування тексту, растеризації гліфів, антиаліасингу. Було проведено огляд методів растеризації гліфів, їх порівняння та вибір методу mSDF. Розглянуто причини необхідності створення SDF на GPU, запропоновано модифікацію методу для створення полів покращеним способом.

В результаті роботи створено програмний продукт призначений для експериментування та дослідження можливостей растеризації тексту методом mSDF в апаратно прискорених застосунках. Застосунок надає 3D-сцену та камеру для експериментів. Застосунок створено із використанням мови програмування Zig, графічного API WebGPU та інтерфейсу користувача на мові розмітки HTML.

У роботі експериментально перевірено швидкодію застосунку із використанням модифікованого методу, та не було виявлено її зниження.

Науковою новизною роботи є порівняння методів растеризації тексту, запропонований модифікований метод для створення mSDF на GPU, а також полів псевдо-дистанцій.

Практична цінність роботи полягає в розробленому застосунку, який дозволяє переконатись в інтерактивній формі, чи задовольняє метод mSDF бажану якість растеризації тексту і якщо так, то переконатись у можливості швидкого створення mSDF атласу під час виконання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Text runs [Electronic resource] // HarfBuzz Manual. – Access mode: <https://harfbuzz.github.io/text-runs.html>.
2. Supported Scripts [Electronic resource] // The Unicode Standard. – Access mode: <https://unicode.org/standard/supported.html>.
3. Why do I need a shaping engine? [Electronic resource] // HarfBuzz Manual. – Access mode: <https://harfbuzz.github.io/why-do-i-need-a-shaping-engine.html>.
4. Ahrens T. A Closer Look At Font Rendering [Electronic resource] / T. Ahrens. – Access mode: <https://www.smashingmagazine.com/2012/04/a-closer-look-at-font-rendering/>.
5. TrueType [Electronic resource] // Wikipedia, The Free Encyclopedia. – Access mode: <https://en.wikipedia.org/wiki/TrueType>.
6. OpenType [Electronic resource] // Wikipedia, The Free Encyclopedia. – Access mode: <https://en.wikipedia.org/wiki/OpenType>.
7. Information technology. Coding of audio-visual objects. Part 22. Open Font Format : ISO/IEC 14496-22:2019. – [Effective from 2019-01]. – Switzerland: ISO, 2019. – 628 p.
8. TrueType Fundamentals [Electronic resource]. – Access mode: <https://learn.microsoft.com/en-us/typography/opentype/spec/ttch01>.
9. Маценко, В.Г. Комп'ютерна графіка : навчальний посібник [Текст] / В.Г. Маценко. – Чернівці : Рута, 2009. – 343 с.
10. Bézier curve [Electronic resource] // Wikipedia, The Free Encyclopedia. – Access mode: https://en.wikipedia.org/wiki/B%C3%A9zier_curve.
11. Pierre J. Why are 2D vector graphics so much harder than 3D? [Electronic resource] / J. Pierre. – Access mode: <https://blog.mecheye.net/2019/05/why-is-2d-graphics-is-harder-than-3d-graphics/>.

12. Lengyel E. GPU-Centered Font Rendering Directly from Glyph Outlines [Text] / E. Lengyel // Journal of Computer Graphics Techniques (JCGT). – 2017. – Vol. 6, no. 2. – P. 31–47.
13. Google Maps [Electronic resource]. – Access mode: <https://www.google.com/maps>.
14. Green, C. Improved Alpha-Tested Magnification for Vector Textures and Special Effects [Text] / C. Green // ACM SIGGRAPH 2007 courses, August 2007 : proceedings. – New York : Association for Computing Machinery, 2007. – P. 9–18.
15. Chlumsky, V. Shape Decomposition for Multi-channel Distance Fields : thesis ... master of science / V. Chlumsky. – Prague : Czech Technical University in Prague, 2015. – 75 p.
16. Alvin, O. Rendering Resolution Independent Fonts in Games and 3D Applications : thesis ... master of science / O. Alvin. – Lund : Lund University, 2020. – 56 p.
17. Blomfeldt E. Introducing the Distance Field Generator [Electronic resource] / E. Blomfeldt. – Access mode: <https://www.qt.io/blog/2018/10/10/introducing-distance-field-generator>.
18. Velzen, J.-W. Texture-based Rendering of Vector-based Shapes : thesis ... master of science / Jan-Willem Velzen. – Delft : Delft University of Technology, 2018. – 44 p.
19. Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics [Text] / [S. Frisken, R. Perry, A. Rockwood et al.] // SIGGRAPH 2000, July 2000 : proceedings. – USA : ACM Press/Addison-Wesley Publishing Co., 2000. – P. 249–254.
20. Loop C. Resolution Independent Curve Rendering Using Programmable Graphics Hardware [Text] / C. Loop, J. Blinn // ACM Transactions on Graphics. – 2005. – Vol. 24, no. 3. – P. 1000–1009.
21. Resolution Independent Rendering of Deformable Vector Objects Using Graphics Hardware [Text] / [Y. Kokojima, K. Sugita, T. Saito et al.] // ACM

SIGGRAPH 2006 Sketches : Boston, July 2006 : proceedings. – New York, 2006.
– P. 118.

22. Dobbie W. GPU text rendering with vector textures [Electronic resource] / W. Dobbie. – Access mode: <http://wdobbie.com/post/gpu-text-rendering-with-vector-textures/>.

23. Esfahbod B. Glyphy [Electronic resource] / B. Esfahbod. – Access mode: http://behdad.org/glyphy_slides.pdf.

24. Stiopin A. Efficient 2D Signed Distance Field Generation on GPU [Electronic resource] / A. Stiopin. – Access mode: <https://astiopin.github.io/2019/01/06/sdf-on-gpu.html>.

25. SVG Strokes [Electronic resource]. – Access mode: <https://www.w3.org/TR/svg-strokes/>.

26. Tom M. Everything I wish I knew when learning C [Electronic resource] / M. Tom. – Access mode: <https://tmewett.com/c-tips/>.

27. Done C. My negative views on Rust [Electronic resource] / C. Done. – Access mode: <https://chrisdone.com/posts/rust/>.

28. Why Zig When There is Already C++, D, and Rust? [Electronic resource]. – Access mode: https://ziglang.org/learn/why_zig_rust_d_cpp/.

29. Overvoorde A. Vulkan Introduction [Electronic resource] / A. Overvoorde. – Access mode: <https://vulkan-tutorial.com/Introduction>.

30. RenderDoc [Electronic resource]. – Access mode: <https://renderdoc.org/>.

31. WebGPU Explainer [Electronic resource]. – Access mode: <https://gpuweb.github.io/gpuweb/explainer/>.

32. WebAssembly Introduction [Electronic resource]. – Access mode: <https://webassembly.github.io/spec/core/intro/introduction.html>.

33. Stein T. The Entity-Component-System - An awesome game-design pattern in C++ [Electronic resource] / T. Stein. – Access mode: <https://www.gamedeveloper.com/design/the-entity-component-system---an-awesome-game-design-pattern-in-c-part-1->.

34. Rasterization Rules [Electronic resource]. – Access mode: <https://learn.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-rasterizer-stage-rules>.

35. Weisstein E. Cubic Formula [Electronic resource] / E. Weisstein. – Access mode: <https://mathworld.wolfram.com/CubicFormula.html>.

ДОДАТОК А
Текст програми

A.1. Файл index.html

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>NUZP 2022 Drozdov</title>
    <script src="https://cdn.tailwindcss.com"></script>
  </head>
  <body>
    <script type="module" src="/src/main.js"></script>
    <form id="props" onsubmit="handleSubmit" class="absolute top-1 left-1 bg-white border p-3 rounded flex flex-col gap-2 shadow text-sm">
      <div class="text-center mb-2">Властивості</div>
      <div><label for="text">Текст:</label>
        <input id="text" type="text" class="border p-1 w-64" value="NUZP 2022 Drozdov"></div>
      <div><label for="color">Колір:</label>
        <input id="color" type="color" value="#000000"></div>
      <div><label for="size">Розмір тексту:</label>
        <input id="size" type="number" class="border p-1 w-16" value="1"></div>
      <div><label for="font">Шрифт:</label>
        <input id="font" type="file"></div>
      <fieldset class="flex gap-2"><legend>Вирівнювання:</legend>
        <div class="flex gap-2">
          <div><input type="radio" id="alignLeft" name="align" value="0" checked>
            <label for="alignLeft">Ліво</label></div>
          <div><input type="radio" id="alignCenter" name="align" value="1">
            <label for="alignCenter">Центр</label></div>
          <div><input type="radio" id="alignRight" name="align" value="2">
            <label for="alignRight">Право</label></div>
        </div>
      </fieldset>
      <hr>
      <div><label for="shadowDist">Радіус тіні:</label>
        <input id="shadowDist" type="number" class="border p-1 w-16" value="0"></div>
      <div><label for="shadowColor">Колір тіні:</label>
        <input id="shadowColor" type="color" value="#000000"></div>
      <div><label for="shadowX">Зміщення тіні:</label>
        <input id="shadowX" type="number" class="border p-1 w-16" value="0">
        <input id="shadowY" type="number" class="border p-1 w-16" value="0"></div>
      <hr>
      <div><label for="outlineDist">Розмір обведення:</label>
        <input id="outlineDist" type="number" class="border p-1 w-16" value="0"></div>
      <div><label for="outlineColor">Колір обведення:</label>
        <input id="outlineColor" type="color" value="#ff0000"></div>
      <hr>
      <div><label for="anchorX">Позиція якоря:</label>
        <input id="anchorX" type="number" class="border p-1 w-16" value="0">
        <input id="anchorY" type="number" class="border p-1 w-16" value="-1">
        <input id="anchorZ" type="number" class="border p-1 w-16" value="0.5"></div>
      <div><label for="rotRoll">Обертання:</label>
        <input id="rotRoll" type="number" class="border p-1 w-16" value="0">

```

```

        <input id="rotPitch" type="number" class="border p-1 w-16" value="0">
        <input id="rotYaw" type="number" class="border p-1 w-16"
value="0"></div>
        <button type="submit" class="bg-gray-300 py-1">Оновити</button>
    </form>
</body>
</html>

```

A.2. Файл main.js

```

import wasmURL from './assets/parent.wasm?url'

class IndexedJsObjects<T extends object> {
    data: Array<T | undefined> = []
    private _freeList: Array<number> = []
    private _next: number = -1

    store (obj: T): number {
        if (this._next === -1) {
            this.data.push(obj)
            this._freeList.push(-1)
            return this.data.length - 1
        } else {
            const prev = this._next
            this.data[prev] = obj
            this._next = this._freeList[prev]
            // this is not necessary
            // this._freeList[prev] = -1
            return prev
        }
    }

    free (idx: number): void {
        this.data[idx] = undefined
        this._freeList[idx] = this._next
        this._next = idx
    }

    get (idx: number): T {
        return this.data[idx] as T
    }
}

const gpuDevices = new IndexedJsObjects<GPUDevice>()
const gpuShaderModules = new IndexedJsObjects<GPUShaderModule>()
const gpuBuffers = new IndexedJsObjects<GPUBuffer>()
const gpuSamplers = new IndexedJsObjects<GPUSampler>()
const gpuTextureViews = new IndexedJsObjects<GPUTextureView>()
const gpuBindGroupLayouts = new IndexedJsObjects<GPUBindGroupLayout>()
const gpuPipelineLayouts = new IndexedJsObjects<GPUPipelineLayout>()
const gpuRenderPipelines = new IndexedJsObjects<GPURenderPipeline>()
const gpuBindGroups = new IndexedJsObjects<GPUBindGroup>()

const wasmMem = new WebAssembly.Memory({ initial: 17, maximum: 40, shared:
false })
let memU32 = new Uint32Array(wasmMem.buffer)
let memU8 = new Uint8Array(wasmMem.buffer)

const BUFFER_BINDING_TYPE: GPUBufferBindingType[] = [

```

```

    'uniform',
    'storage',
    'read-only-storage'
]

const VERTEX_FORMAT: GPUVertexFormat[] = [
  'float32x3',
];
const VERTEX_FORMAT_SIZE: number[] = [
  12
];

const { instance, module } = await WebAssembly.instantiateStreaming(
  fetch(wasmURL), {
    console: {
      logv: window.console.log,
      logFont: window.console.log,
      logstr (ptr: number, length: number) {
        const decoder = new TextDecoder()
        const str = decoder.decode(memU8.subarray(ptr, ptr + length))
        console.log(str)
      },
    },
    env: {
      memory: wasmMem,
      papiGpuCreateBuffer: (device: number, size: number, usage:
GPUBufferUsageFlags): number => {
        const mappedAtCreation = Boolean((usage & GPUBufferUsage.VERTEX) ||
(usage & GPUBufferUsage.INDEX))
        const buffer = gpuDevices.get(device)
          .createBuffer({ size, usage, mappedAtCreation })
        return gpuBuffers.store(buffer)
      },
      papiGpuCreateBindGroupLayout: (device: number, length: number, wLayout:
number): number => {
        wLayout /= memU32.BYTES_PER_ELEMENT
        const descEntries: GPUBindGroupLayoutEntry[] = []
        while (length--) {
          const kind = memU32[wLayout + 0]
          descEntries.push({
            binding: memU32[wLayout + 1],
            visibility: memU32[wLayout + 2],
            buffer: (kind === 0)
              ? {
                  type: BUFFER_BINDING_TYPE[memU32[wLayout + 3]],
                  hasDynamicOffset: memU32[wLayout + 4] === 1,
                  minBindingSize: memU32[wLayout + 5],
                } : undefined,
            sampler: (kind === 1)
              ? {
                  type: 'filtering',
                } : undefined,
            texture: (kind === 2)
              ? {
                  sampleType: 'float'
                } : undefined,
            storageTexture: undefined,
            externalTexture: undefined
          })
          wLayout += 6
        }
        const bindGroupLayout = gpuDevices.get(device)
          .createBindGroupLayout({ label: undefined, entries: descEntries })
        return gpuBindGroupLayouts.store(bindGroupLayout)
      }
    }
  }
);

```

```

    },
    papiGpuCreatePipelineLayout: (device: number, length: number, wLayout:
number): number => {
        wLayout /= memU32.BYTES_PER_ELEMENT
        const bindGroupLayouts: GPUBindGroupLayout[] = []
        while (length--) {
            bindGroupLayouts.push(gpuBindGroupLayouts.get(memU32[wLayout]))
            wLayout += 1
        }
        const pipelineLayout = gpuDevices.get(device)
            .createPipelineLayout({
                label: undefined,
                bindGroupLayouts: bindGroupLayouts
            })
        return gpuPipelineLayouts.store(pipelineLayout)
    },
    papiGpuCreateRenderPipeline: (device: number, wPipeline: number): number
=> {
        wPipeline /= memU32.BYTES_PER_ELEMENT;
        const pipelineLayout = gpuPipelineLayouts.get(memU32[wPipeline + 0]);
        const vertexShader = gpuShaderModules.get(memU32[wPipeline + 1]);

        const bufferLayouts: GPUVertexBufferLayout[] = [];
        let wBuffers = (wPipeline + 2) * memU32.BYTES_PER_ELEMENT;
        let wAttr = memU32[wPipeline + 3];
        for (let i = 0; i < 4; ++i) {
            const serialized = memU8[wBuffers + i];
            if (serialized === 0) break;
            const attrCount = 0x7F & serialized;
            const stepMode = (0x80 & serialized) ? 'instance' : 'vertex';
            const bufferDesc: GPUVertexBufferLayout = {
                arrayStride: 0,
                stepMode: stepMode,
                attributes: []
            };
            for (let i = 0; i < attrCount; ++i, wAttr += 2) {
                const formatEnum = memU8[wAttr + 0];
                const shaderLocation = memU8[wAttr + 1];
                (bufferDesc.attributes as GPUVertexAttribute[]).push({
                    format: VERTEX_FORMAT[formatEnum],
                    offset: bufferDesc.arrayStride,
                    shaderLocation
                });
                bufferDesc.arrayStride += VERTEX_FORMAT_SIZE[formatEnum];
            }
            bufferLayouts.push(bufferDesc);
        }

        const wFrag = memU32[wPipeline + 4] / memU32.BYTES_PER_ELEMENT
        const fragState: GPUFragmentState | undefined = (wFrag !== 0)
        ? {
            module: gpuShaderModules.get(memU32[wFrag + 0]),
            entryPoint: 'fragment_main',
            targets: [
                {
                    format: canvasFormat,
                    blend: undefined,
                    writeMask: undefined
                }
            ]
        }
        : undefined

        const cull = (memU32[wPipeline + 5] !== 0) ? 'back' : 'none'
        const psoDesc: GPURenderPipelineDescriptor = {

```

```

    label: undefined,
    layout: pipelineLayout,
    vertex: {
      module: vertexShader,
      entryPoint: 'vertex_main',
      buffers: bufferLayouts
    },
    fragment: fragState,
    primitive: {
      topology: 'triangle-list',
      cullMode: cull,
      frontFace: 'ccw'
    },
    depthStencil: {
      depthWriteEnabled: true,
      depthCompare: 'less',
      format: 'depth24plus',
    },
    multisample: undefined
  }
  console.log(JSON.stringify(psoDesc.vertex.buffers, null, 2));
  const pipeline = gpuDevices.get(device).createRenderPipeline(psoDesc)
  return gpuRenderPipelines.store(pipeline)
},
papiGpuCreateBindGroup: (device: number, layout: number, length: number,
wEntries: number): number => {
  wEntries /= memU32.BYTES_PER_ELEMENT
  const entries: GPUBindGroupEntry[] = []
  while (length--) {
    const kind = memU32[wEntries + 0]
    let resource: GPUBindingResource
    if (kind === 0) {
      resource = {
        buffer: gpuBuffers.get(memU32[wEntries + 2]),
        offset: undefined,
        size: undefined
      }
    } else if (kind === 1) {
      resource = gpuSamplers.get(memU32[wEntries + 2])
    } else if (kind === 2) {
      resource = gpuTextureViews.get(memU32[wEntries + 2])
    }
    entries.push({ binding: memU32[wEntries + 1], resource: resource! })
    wEntries += 3
  }
  const bindGroup = gpuDevices.get(device).createBindGroup({
    label: undefined,
    layout: gpuBindGroupLayouts.get(layout),
    entries: entries
  })
  return gpuBindGroups.store(bindGroup)
},
papiGpuWriteBuffer: (device: number, buffer: number, bufferOffset:
number, offset: number, size: number): void => {
  gpuDevices.get(device).queue.writeBuffer(
    gpuBuffers.get(buffer),
    bufferOffset,
    wasmMem.buffer,
    offset,
    size
  )
},
papiGpuUpload: (buffer: number, bufferOffset: number, source: number,
offset: number, size: number): void => {

```

```

    const mapped = new Uint8Array(
      gpuBuffers.get(buffer).getMappedRange(bufferOffset, size));

    mapped.set(new Uint8Array(PRELOADED[source], offset, size));
  },
  papiGpuUnmap: (buffer: number): void => {
    gpuBuffers.get(buffer).unmap();
  },
  papiGpuCreateShader: (device: number, source: number): number => {
    const utf8reader = new TextDecoder();
    const shaderCode = utf8reader.decode(PRELOADED[source]);
    const shaderModule = gpuDevices.get(device).createShaderModule({ code:
shaderCode });
    return gpuShaderModules.store(shaderModule);
  },
  papiGpuSetPipeline: (pipeline: number): void => {
    renderPassGlobal!.setPipeline(gpuRenderPipelines.get(pipeline));
  },
  papiGpuSetVertexBuffer: (slot: number, buffer: number, offset: number):
void => {
    renderPassGlobal!.setVertexBuffer(slot,          gpuBuffers.get(buffer),
offset);
  },
  papiGpuSetIndexBuffer: (buffer: number, offset: number): void => {
    renderPassGlobal!.setIndexBuffer(gpuBuffers.get(buffer),          'uint16',
offset);
  },
  papiGpuDraw: (vertexCount: number): void => {
    renderPassGlobal!.draw(vertexCount);
  },
  papiGpuDrawIndexed: (indexCount: number): void => {
    renderPassGlobal!.drawIndexed(indexCount);
  },
  papiGpuSetBindGroup: (index: number, bindGroup: number): void => {
    renderPassGlobal!.setBindGroup(index, gpuBindGroups.get(bindGroup));
  },
}
})

const engine = instance.exports

const adapter = await navigator.gpu.requestAdapter({
  powerPreference: 'low-power'
})
if (!adapter) throw new Error()

const device = await adapter.requestDevice()
device.lost.then(() => {
  console.log('GPU lost.')
})

gpuDevices.store(device)

function createCanvas () {
  const canvas = document.createElement('canvas')
  canvas.id = 'fullscreen'

  // https://web.dev/device-pixel-content-box/
  const observer = new ResizeObserver(([canvasEntry]) => {
    const deviceSize = canvasEntry.devicePixelContentSize[0]
    needResize = { width: deviceSize.inlineSize, height: deviceSize.blockSize
  }
  engine.handleResize(deviceSize.inlineSize, deviceSize.blockSize)
}

```

```

    })
    observer.observe(canvas, { box: 'device-pixel-content-box' })

    return canvas
  }

let depthTexture: GPUTexture | null = null;
let needResize: { width: number, height: number } | null = null

function handleResize (width: number, height: number) {
  canvas.width = width;
  canvas.height = height;

  if (depthTexture) {
    depthTexture.destroy();
  }

  depthTexture = device.createTexture({
    size: { width: width, height: height },
    format: 'depth24plus',
    usage: GPUTextureUsage.RENDER_ATTACHMENT,
  });
}

const canvas = createCanvas()
document.body.appendChild(canvas)

const swapChain = canvas.getContext('webgpu')
if (!swapChain) throw new Error()
console.log(swapChain)

const canvasFormat = navigator.gpu.getPreferredCanvasFormat()
swapChain.configure({
  device: device,
  usage: GPUTextureUsage.RENDER_ATTACHMENT | GPUTextureUsage.COPY_SRC /* copy
for zoom */,
  format: canvasFormat,
  alphaMode: 'opaque'
})

const nearestSampler = device.createSampler({
  magFilter: 'nearest',
  minFilter: 'nearest',
});
gpuSamplers.store(nearestSampler)

let simpleTexture: GPUTexture
{
  simpleTexture = device.createTexture({
    size: { width: 4, height: 4 },
    format: 'bgra8unorm-srgb',
    viewFormats: ['bgra8unorm'],
    usage:
      GPUTextureUsage.TEXTURE_BINDING |
      GPUTextureUsage.COPY_DST |
      GPUTextureUsage.RENDER_ATTACHMENT,
  });
  device.queue.writeTexture(
    { texture: simpleTexture },
    new Uint8Array(PRELOADED[4], 0x96),
    { bytesPerRow: 4 * 1 * 4 },
    { width: 4, height: 4 },
  );
}

```

```

const simpleTextureView = simpleTexture.createView({ format: 'bgra8unorm' });
gpuTextureViews.store(simpleTextureView); // idx: 0

let rtCopy: GPUTexture
{
  rtCopy = device.createTexture({
    size: { width: 20, height: 20 },
    format: swapChain!.getCurrentTexture().format,
    usage:
      GPUTextureUsage.TEXTURE_BINDING |
      GPUTextureUsage.COPY_DST |
      GPUTextureUsage.RENDER_ATTACHMENT,
  });
  gpuTextureViews.store(rtCopy.createView()); // idx: 1
}

await new Promise(resolve => setTimeout(resolve, 100));
handleResize(needResize!.width, needResize!.height)
engine.initialize()

let renderPassGlobal: GPURenderPassEncoder | null = null

const zoomOrigin = { x: 0, y: 0 };

function mainLoop () {
  if (needResize) {
    handleResize(needResize.width, needResize.height);
  }

  const cmdBuffer = device.createCommandEncoder()

  const renderPass = cmdBuffer.beginRenderPass({
    colorAttachments: [{
      view: swapChain!.getCurrentTexture().createView(),
      clearValue: { r: 1.0, g: 1.0, b: 1.0, a: 1.0 },
      // clearValue: { r: 0.0, g: 0.0, b: 0.0, a: 1.0 },
      // clearValue: { r: 0.05, g: 0.05, b: 0.15, a: 1.0 },
      loadOp: 'clear',
      storeOp: 'store'
    }],
    depthStencilAttachment: {
      view: depthTexture!.createView(),
      depthClearValue: 1.0,
      depthLoadOp: 'clear',
      depthStoreOp: 'store',
    }
  })

  renderPassGlobal = renderPass
  engine.tick()
  renderPassGlobal = null
  renderPass.end()

  cmdBuffer.copyTextureToTexture(
    { texture: swapChain!.getCurrentTexture(), origin: zoomOrigin },
    { texture: rtCopy },
    { width: 20, height: 20 }
  );
  // TODO: Here we should add next renderpass, but in current impl
  //        zoomed part of current frame is rendered in the next one
  //        (also can't show zoom what's rendered under zoomed area).

  device.queue.submit([cmdBuffer.finish()])
  requestAnimationFrame(mainLoop)
}

```

```

}
requestAnimationFrame(mainLoop)

const KeyCodesMap = new Map([
  ['KeyW', 1],
  ['KeyA', 2],
  ['KeyS', 3],
  ['KeyD', 4],
  ['ArrowUp', 5],
  ['ArrowLeft', 6],
  ['ArrowRight', 7],
  ['ArrowDown', 8],
  ['KeyQ', 9],
  ['KeyE', 10],
])

window.addEventListener('keyup', (e) => {
  if (e.code === 'KeyP') {
    canvas.toBlob((blob) => {
      navigator.clipboard.write([
        new ClipboardItem({ [blob!.type]: blob! }),
      ]);
    }, 'image/png');
  }
  const num = KeyCodesMap.get(e.code)
  if (num !== null) {
    engine.handleMovement(num)
  }
})

canvas.addEventListener('mousemove', (e) => {
  const radius = 10
  zoomOrigin.x = Math.min(Math.max(0, Math.floor(e.x *
window.devicePixelRatio - radius)), canvas.width - radius * 2)
  zoomOrigin.y = Math.min(Math.max(0, Math.floor(e.y *
window.devicePixelRatio - radius)), canvas.height - radius * 2)
})

window.document.body.addEventListener('click', async () => {
  const fonts = await window.queryLocalFonts({ postscriptNames: ['SegoeUI']
});
  const segoeUI = fonts[0];
  const blob = await segoeUI.blob();
  const data = await blob.arrayBuffer();

  const fontPtr = engine.malloc(data.byteLength);
  memU32 = new Uint32Array(wasmMem.buffer)
  memU8 = new Uint8Array(wasmMem.buffer)
  new Uint8Array(wasmMem.buffer, fontPtr).set(new Uint8Array(data));
  console.time('setFont')
  engine.setFont(fontPtr, data.byteLength);
  console.timeEnd('setFont')
});

```

A.3. Файл lib.zig

```

const assert = @import("std").debug.assert;
const Math = @import("std").math;
const std = @import("std");
const font = @import("../text_object/font_reader.zig");

```

```

const textRun = @import("../text_object/text_run.zig");
const camera = @import("../camera.zig");

var gpa = std.heap.GeneralPurposeAllocator(.{}){};
// var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
const alloc = gpa.allocator();

export fn malloc (size: usize) [*]u8 {
    const slice = alloc.alloc(u8, size) catch unreachable;
    return slice.ptr;
}

export fn free (ptr: [*]u8, size: usize) void {
    alloc.free(ptr[0..size]);
}

var fontStorage: []u8 = undefined;
export fn setFont (bytes: [*]u8, size: u32) void {
    fontStorage = bytes[0..size];
    font.initialize(@alignCast(4, fontStorage)) catch unreachable;
}

// buffer={index(u16),position(vec3<f32>),data(normal,color...)}
const Primitive = struct {
    buffer: JsRef,
    size: u32,
    vertexCount: u16,
    indexCount: u16,

    fn positionOffset (self: Primitive) u32 {
        return self.indexCount * @sizeof(u16);
    }

    fn dataOffset (self: Primitive) u32 {
        return self.positionOffset() + self.vertexCount * @sizeof([3]f32);
    }
};

const Window = struct {
    width: u32,
    height: u32,
};

var window = Window{
    .width = 1,
    .height = 1,
};

const deviceRef: JsRef = 0;

// #region 3D path
// TODO: should it be called per viewport?

var unlitShader: JsRef = undefined;
var unlitPipeline: JsRef = undefined;
var litShader: JsRef = undefined;
var litPipeline: JsRef = undefined;
var plane: Primitive = undefined;
var monkey: Primitive = undefined;
// #endregion

const Box2D = struct {
    width: f32,
    height: f32,
    marginTop: f32,

```

```

marginLeft: f32,

fn writeToBuffer (self: Box2D, primitive: *Primitive) void {
    const rectData = [_]f32{
        self.marginLeft + self.width, self.marginTop + 0,          1, 0,
        self.marginLeft + 0, self.marginTop + 0,                  0, 0,
        self.marginLeft + self.width, self.marginTop + self.height, 1, 1,
        //
        self.marginLeft + 0, self.marginTop + 0,                  0, 0,
        self.marginLeft + 0, self.marginTop + self.height,        0, 1,
        self.marginLeft + self.width, self.marginTop + self.height, 1, 1,
    };
    papiGpuWriteBuffer(deviceRef, primitive.buffer, 0, @ptrCast([]*const u8,
&rectData), primitive.size);
}
};

var msdfCurveShader: JsRef = undefined;
var msdfCurvePipeline: JsRef = undefined;

export fn initialize () void {
    unlitShader = papiGpuCreateShader(deviceRef, (1));
    litShader = papiGpuCreateShader(deviceRef, (2));
    sdfCurveShader = papiGpuCreateShader(deviceRef, (7));

    {
        const meshesBuf = papiGpuCreateBuffer(deviceRef,
            4096,
            GPUBufferUsage.Vertex | GPUBufferUsage.Index
        );
        plane = Primitive{
            .size = 156,
            .vertexCount = 6,
            .indexCount = 6,
            .buffer = meshesBuf,
        };
        papiGpuUpload(meshesBuf, 0, (0), 0, plane.size);
        papiGpuUnmap(meshesBuf);
    }

    {
        const meshesBuf = papiGpuCreateBuffer(deviceRef,
            64000,
            GPUBufferUsage.Vertex | GPUBufferUsage.Index
        );
        monkey = Primitive{
            .size = 52992,
            .vertexCount = 1966,
            .indexCount = 2904,
            .buffer = meshesBuf,
        };
        papiGpuUpload(meshesBuf, 0, (6), 0, monkey.size);
        papiGpuUnmap(meshesBuf);
    }

    {
        const meshesBuf = papiGpuCreateBuffer(deviceRef,
            4096,
            GPUBufferUsage.Vertex | GPUBufferUsage.CopyDst
        );
        zoomVertData = Primitive{
            .size = (2 + 2) * 6 * 4,
            .vertexCount = 6,
            .indexCount = 0,
        };
    }
}

```

```

        .buffer = meshesBuf,
    };
    papiGpuUnmap(meshesBuf);
}

{
    const meshesBuf = papiGpuCreateBuffer(deviceRef,
        4096,
        GPUBufferUsage.Vertex | GPUBufferUsage.CopyDst
    );
    btnVertData = Primitive{
        .size = (2 + 2) * 6 * 4,
        .vertexCount = 6,
        .indexCount = 0,
        .buffer = meshesBuf,
    };
    papiGpuUnmap(meshesBuf);
}

{
    perCamBuffer = papiGpuCreateBuffer(deviceRef,
        @sizeof(PerCamUniform),
        GPUBufferUsage.Uniform | GPUBufferUsage.CopyDst
    );
    const layoutEntries = [_]GPUBindGroupLayoutEntry{
        .{
            .kind = GPUBindGroupLayoutEntry.Kind.Buffer,
            .binding = 0,
            .visibility = GPUShaderStage.Vertex | GPUShaderStage.Fragment,
            .data = .{
                .buffer = .{
                    .type = GPUBufferBindingType.Uniform,
                    .hasDynamicOffset = false,
                    .minBindingSize = @sizeof(PerCamUniform),
                },
            },
        },
    };
    perCamBindGroupLayout = papiGpuCreateBindGroupLayout(deviceRef,
        layoutEntries.len, &layoutEntries);
    const bindGroupLayouts = [_]JsRef{perCamBindGroupLayout};
    perCamPipelineLayout = papiGpuCreatePipelineLayout(deviceRef,
        bindGroupLayouts.len, &bindGroupLayouts);
}

{
    const bufferLayouts = [_]u8{
        GPUVertexBufferLayout.serialize(1, GPUVertexStepMode.Vertex),
        GPUVertexBufferLayout.serialize(1, GPUVertexStepMode.Vertex),
        0, undefined,
    };
    var vertexAttributes = [_]GPUVertexAttribute{
        .{ .format = GPUVertexFormat.Float32x3, .shaderLocation = 0 },
        .{ .format = GPUVertexFormat.Float32x3, .shaderLocation = 1 },
    };

    var fragState = GPUFragmentState{
        .module = unlitShader,
    };

    const psoDesc = GPURenderPipelineDescriptor{
        .layout = perCamPipelineLayout,
        .vertex = .{
            .module = unlitShader,

```

```

        .buffers = bufferLayouts,
        .attributes = &vertexAttributes,
    },
    .fragment = &fragState,
    .cullBackface = false,
};
unlitPipeline = papiGpuCreateRenderPipeline(deviceRef, &psoDesc);
}

{
    const bufferLayouts = [_]u8{
        GPUVertexBufferLayout.serialize(1, GPUVertexStepMode.Vertex),
        GPUVertexBufferLayout.serialize(1, GPUVertexStepMode.Vertex),
        0, undefined,
    };
    var vertexAttributes = [_]GPUVertexAttribute{
        .{ .format = GPUVertexFormat.Float32x3, .shaderLocation = 0 },
        .{ .format = GPUVertexFormat.Float32x3, .shaderLocation = 1 },
    };

    var fragState = GPUFragmentState{
        .module = litShader,
    };

    const psDesc = GPURenderPipelineDescriptor{
        .layout = perCamPipelineLayout,
        .vertex = .{
            .module = litShader,
            .buffers = bufferLayouts,
            .attributes = &vertexAttributes,
        },
        .fragment = &fragState,
        .cullBackface = true,
    };
    litPipeline = papiGpuCreateRenderPipeline(deviceRef, &psDesc);
}

{
    const uniformBindGroupEntries = [_]GPUBindGroupEntry{
        .{
            .kind = GPUBindGroupLayoutEntry.Kind.Buffer,
            .binding = 0,
            .resource = .{ .ref = perCamBuffer },
        },
    };
    perCamBindGroup = papiGpuCreateBindGroup(deviceRef, perCamPipelineLayout,
uniformBindGroupEntries.len, &uniformBindGroupEntries);
}
}

export fn tick () void {
    camera.transform();
    papiGpuWriteBuffer(deviceRef, camera.perCamBuffer, 0, @ptrCast([*]const u8,
&camera.firstCamUniform), @sizeof(@TypeOf(camera.firstCamUniform)));

    papiGpuSetBindGroup(0, perCamBindGroup);
    // plane
    papiGpuSetPipeline(unlitPipeline);
    papiGpuSetIndexBuffer(plane.buffer, 0);
    papiGpuSetVertexBuffer(0, plane.buffer, plane.positionOffset());
    papiGpuSetVertexBuffer(1, plane.buffer, plane.dataOffset());
    papiGpuDrawIndexed(plane.indexCount);
    // monkey
    papiGpuSetPipeline(litPipeline);

```

```

    papiGpuSetIndexBuffer(monkey.buffer, 0);
    papiGpuSetVertexBuffer(0, monkey.buffer, monkey.positionOffset());
    papiGpuSetVertexBuffer(1, monkey.buffer, monkey.dataOffset());
    papiGpuDrawIndexed(monkey.indexCount);

    textRun.draw();
}

export fn handleResize (width: u32, height: u32) void {
    window.width = width;
    window.height = height;
}

```

A.4. Файл camera.zig

```

const assert = @import("std").debug.assert;
const Math = @import("std").math;
const std = @import("std");
const math3d = @import("../math3d.zig");
const Vector3 = math3d.Vector3;
const Vector4 = math3d.Vector4;
const Quaternion = math3d.Quaternion;
const Matrix4 = math3d.Matrix4;

var perCamBindGroupLayout: JsRef = undefined;
var perCamPipelineLayout: JsRef = undefined;
var perCamBuffer: JsRef = undefined;
var perCamBindGroup: JsRef = undefined;

const PerCamUniform = extern struct {
    viewProj: Matrix4,
    camPos: Vector4,
    lightPos: Vector4,
};

var firstCamUniform = PerCamUniform{
    .viewProj = Matrix4.Identity,
    .camPos = Vector4.new(0, 0, 0, 1),
    .lightPos = Vector4.new(0, -15, 15, 1),
};

// from RH z-up to LH y-up
const ndcMatrix = Matrix4.new(.{
    1, 0, 0, 0,
    0, 0, 1, 0,
    0, 1, 0, 0,
    0, 0, 0, 1,
});

/// For NDC with LH y-up.
fn perspective (
    left: f32, bottom: f32, near: f32,
    right: f32, top: f32, far: f32
) Matrix4 {
    const x = 2 * near / (right - left);
    const y = 2 * near / (top - bottom);

    const a = -(right + left) / (right - left);
    const b = -(top + bottom) / (top - bottom);
    const c = far / (far - near);

```

```

const d = -far * near / (far - near);

return Matrix4.new({
  x, 0, a, 0,
  0, y, b, 0,
  0, 0, c, d,
  0, 0, 1, 0
}).mul(ndcMatrix);
}

/// For NDC with LH y-up.
fn orthographic (
  left: f32, bottom: f32, near: f32,
  right: f32, top: f32, far: f32
) Matrix4 {
  const sx = 2 / (right - left);
  const sy = 2 / (top - bottom);
  const sz = 1 / (far - near);

  const tx = -(right + left) / (right - left);
  const ty = -(top + bottom) / (top - bottom);
  const tz = -near / (far - near);

  return Matrix4.new({
    sx, 0, 0, tx,
    0, sy, 0, ty,
    0, 0, sz, tz,
    0, 0, 0, 1
  }).mul(ndcMatrix);
}

var camCenter = Vector3.Zero;
var camPos = Vector3.Zero;
var camRot = Quaternion.Identity;
var camYaw: f32 = 0;
var camPitch: f32 = 0;
var camRoll: f32 = 0;

export fn transform () void {
  const view =
    Matrix4.fromQuaternion(camRot.normalize().invert()).mul(
      Matrix4.fromTranslation(camPos.negate()));

  const aspect: f32 = 1;
  const near: f32 = 0.1;
  const far: f32 = 100;
  const fovy: f32 = 1; // rad
  const angle = fovy / 2;
  const ymax = near * @tan(angle);
  const xmax = ymax * aspect;

  firstCamUniform.viewProj = perspective(
    -xmax, -ymax, near,
    xmax, ymax, far
  ).mul(view);
}

const KeyCode = enum(u32) {
  KeyW = 1,
  KeyA,
  KeyS,
  KeyD,
  ArrowUp,
  ArrowLeft,

```

```

    ArrowRight,
    ArrowDown,
    KeyQ,
    KeyE,
};

export fn handleMovement (code: KeyCode) void {
    camCenter = switch (code) {
        .KeyW => Vector3.new(camCenter.x, camCenter.y + 1, camCenter.z),
        .KeyS => Vector3.new(camCenter.x, camCenter.y - 1, camCenter.z),
        .KeyA => Vector3.new(camCenter.x - 1, camCenter.y, camCenter.z),
        .KeyD => Vector3.new(camCenter.x + 1, camCenter.y, camCenter.z),
        else => camCenter
    };

    const deltaRad = Math.pi / 18.0;
    switch (code) {
        .ArrowUp => { camPitch -= deltaRad; },
        .ArrowDown => { camPitch += deltaRad; },
        .ArrowLeft => { camYaw += deltaRad; },
        .ArrowRight => { camYaw -= deltaRad; },
        .KeyQ => { camRoll -= deltaRad; },
        .KeyE => { camRoll += deltaRad; },
        else => {},
    }

    camRot = Quaternion.fromEuler(camPitch, camRoll, camYaw);

    const lookDirection = Vector3.Forward.rotate(camRot);

    const radius: f32 = 5;
    camPos = lookDirection.negate().scale(radius);
    firstCamUniform.camPos = Vector4.fromVec3(camPos, 1.0);
}

```

A.5. Файл text_run.zig

```

const std = @import("std");
const Vector2 = @import("../util/math2d.zig").Vector2;
const Vector3 = @import("../util/math3d.zig").Vector3;
const Quaternion = @import("../util/math3d.zig").Quaternion;
const Font = @import("../font_reader.zig").Font;

const TextAlign = enum {
    Left,
    Center,
    Right,
};

const TextRun = struct {
    text: std.ArrayList(u8),
    isDirty: bool,
    font: *Font,
    capSize: f32,
    color: f32[4],
    origin: Vector3,
    rotation: Quaternion,
    textAlign: TextAlign,
    outlineColor: f32[4],
    shadowColor: f32[4],
}

```

```

    outlineSize: f32,
    shadowSize: f32,
};

const TextSystem = struct {
    textPipeline: JsRef,

```

A.6. Файл text_shaping.zig

```

const std = @import("std");
const hb = @cImport({
    @cInclude("hb.h");
});
const math2d = @import("../util/math2d.zig");
const Vector2 = math2d.Vector2;
const BoundingBox = math2d.BoundingBox;

pub const ShapedGlyph = struct {
    id: u16,
    bbox: BoundingBox,
};

pub const TextShaper = struct {
    buffer: *hb.hb_buffer_t,
    font: *hb.hb_font_t,
    face: *hb.hb_face_t,
    shaper: hb.hb_script_t,

    pub fn init () TextShaper {
        var buffer = hb.hb_buffer_create();
        hb.hb_buffer_set_direction(buffer, hb.HB_DIRECTION_LTR);
        hb.hb_buffer_set_script(buffer, hb.HB_SCRIPT_LATIN);
        hb.hb_buffer_set_language(buffer, hb.hb_language_from_string("en", -1));

        return TextShaper {
            .buffer = buffer,
            .font = hb.hb_font_create(),
            .face = hb.hb_face_create(),
            .shaper = hb.hb_script_from_string("en", -1),
        };
    }

    pub fn shape (self: *TextShaper, text: []const u8)
std.ArrayList(ShapedGlyph) {
    hb.hb_buffer_add_utf8(self.buffer, text.ptr, text.len, 0, text.len);
    hb.hb_shape(self.font, self.buffer, null, 0);

    var glyph_count: u32 = undefined;
    var glyphs = hb.hb_buffer_get_glyph_infos(self.buffer, &glyph_count);
    var positions = hb.hb_buffer_get_glyph_positions(self.buffer,
&glyph_count);

    var i = 0;
    var out = std.ArrayList(ShapedGlyph).init(std.heap.page_allocator);
    while (i < glyph_count) : (i += 1) {
        var glyph = glyphs[i];
        var position = positions[i];
        var shaped = ShapedGlyph {
            .id = @intCast(u16, glyph.codepoint),

```

```

        .bbox = BoundingBox {
            .min = Vector2 {
                .x = @intToFloat(f32, position.x_offset),
                .y = @intToFloat(f32, position.y_offset),
            },
            .max = Vector2 {
                .x = @intToFloat(f32, position.x_offset + position.x_advance),
                .y = @intToFloat(f32, position.y_offset + position.y_advance),
            },
        },
    };
    out.append(shaped) catch unreachable;
}
return out;
}

pub fn deinit (self: *TextShaper) void {
    hb.hb_buffer_destroy(self.buffer);
    hb.hb_font_destroy(self.font);
    hb.hb_face_destroy(self.face);
}
};

```

A.7. Файл font_reader.zig

```

const std = @import("std");
const mem = @import("std").mem;
const assert = std.debug.assert;
const read = mem.readIntSliceForeign;
const Vector2 = @import("../util/math2d.zig").Vector2;
const BezierQuad = @import("../util/math2d.zig").BezierQuad;

pub const Glyph = struct {
    scale: Vector2,
    contours: std.ArrayList(Contour),
};

pub const Contour = struct {
    segments: std.ArrayList(BezierQuad),
};

const Tag = extern union {
    str: [4]u8,
    num: u32,

    pub fn new (str: *const [4]u8) Tag {
        return Tag{ .str = str.* };
    }

    pub fn equal (self: Tag, other: Tag) bool {
        return self.num == other.num;
    }
};

const TableBytes = []align(4) const u8;

const TableDirectory = extern struct {
    sfntVersion: Tag,
    numTables: u16,
    _searchRange: u16,

```

```

    _entrySelector: u16,
    _rangeShift: u16,

    pub fn fromBytes (bytes: TableBytes) FontError!*const TableDirectory {
        if (bytes.len < @sizeof(TableDirectory)) {
            return error.InvalidFont;
        }
        return @ptrCast(*const TableDirectory, bytes.ptr);
    }

    pub fn tableRecords (self: TableDirectory, bytes: TableBytes)
FontError![]const TableRecord {
    const numTables = @byteSwap(self.numTables);
    if (bytes.len - @sizeof(TableDirectory) < numTables *
@sizeof(TableRecord)) {
        return error.InvalidFont;
    }
    return @ptrCast(
        [*]const TableRecord,
        bytes[@sizeof(TableDirectory)..].ptr
    )[0..numTables];
}

    pub fn findTableBytes (
        comptime TableT: type,
        records: []const TableRecord,
        bytes: TableBytes
    ) FontError!TableBytes {
        var tblRecord: ?*const TableRecord = null;
        for (records) |*record| {
            if (record.tableTag.equal(TableT.TAG)) {
                tblRecord = record;
                break;
            }
        }
        if (tblRecord == null) return error.InvalidFont;

        const offset = @byteSwap(tblRecord.?.offset);
        const length = @byteSwap(tblRecord.?.length);

        if (!mem.isAligned(offset, 4) or
            bytes.len < offset + length or
            length < @bitSizeOf(TableT) / 8
        ) {
            return error.InvalidFont;
        }

        return @alignCast(4, bytes[offset..offset + length]);
    }

    pub fn findTable (
        comptime TableT: type,
        records: []const TableRecord,
        bytes: TableBytes
    ) FontError!*const TableT {
        const tableBytes = try TableDirectory.findTableBytes(TableT, records,
bytes);
        return @ptrCast(*const TableT, tableBytes.ptr);
    }
};

const TableRecord = extern struct {
    tableTag: Tag,
    checksum: u32,

```

```

    offset: u32,
    length: u32,
};

const LocaTable = union(enum) {
    const TAG = Tag.new("loca");
    Short: ArrayReader(u16),
    Long: ArrayReader(u32),

    pub fn fromBytes (bytes: []const u8, isLong: bool) LocaTable {
        var stream = Stream.new(bytes);
        return if (isLong) .{
            .Long = stream.readArrayToEnd(u32)
        } else .{
            .Short = stream.readArrayToEnd(u16)
        };
    }

    pub fn getGlyphBytes (self: LocaTable, glyphId: u16, glyf: []const u8)
    FontError![]const u8 {
        var start: u32 = 0;
        var end: u32 = 0;
        if (glyphId == std.math.maxInt(u16)) {
            return error.InvalidFont;
        }
        switch (self) {
            .Short => |offsets| {
                start = 2 * @as(u32, try offsets.get(glyphId));
                end = 2 * @as(u32, try offsets.get(glyphId + 1));
            },
            .Long => |offsets| {
                start = try offsets.get(glyphId);
                end = try offsets.get(glyphId + 1);
            },
        }
        if (start > end) return error.InvalidFont;
        if (glyf.len < end) return error.InvalidFont;
        return glyf[start..end];
    }
};

const MaxpTable = extern struct {
    const TAG = Tag.new("maxp");
    version: u32,
    numGlyphs: u16,
};

const GlyphPoint = struct {
    x: i16,
    y: i16,
    onCurve: bool,
    endOfCurve: bool,
};

const GlyfTable = extern struct {
    const TAG = Tag.new("glyf");

    fn parseGlyph (bytes: []const u8) FontError!Glyph {
        var stream = Stream.new(bytes);
        const glyphType = try stream.read(i16);
        const xMin = @intToFloat(f32, try stream.read(i16));
        const yMin = @intToFloat(f32, try stream.read(i16));
        const xMax = @intToFloat(f32, try stream.read(i16));
        const yMax = @intToFloat(f32, try stream.read(i16));
    }
};

```

```

    if (glyphType < 0) {
        // composite glyph
        return error.UnsupportedFont;
    }
    if (glyphType == 0) {
        return error.InvalidFont;
    }
    // simple glyph
    const numberOfContours = @intCast(u16, glyphType);

    const endPtsOfContours = try stream.readSoaField(u16, numberOfContours);

    // skip instructions
    const instructionLength = try stream.read(u16);
    stream.skipArray(u8, instructionLength);

    const numPts = endPtsOfContours.get(numberOfContours - 1) +% 1;
    if (numPts == 0) {
        return error.InvalidFont;
    }

    var glyphFlags: CheckedStream = undefined;
    var xCoordinates: CheckedStream = undefined;
    var yCoordinates: CheckedStream = undefined;
    {
        var tmpStream = Stream.new(stream.bytes);
        var pointIdx: u32 = 0;
        var xLen: u32 = 0;
        var yLen: u32 = 0;
        while (pointIdx < numPts) {
            const flags = try tmpStream.read(u8);
            var repeats: u32 = if (flags & 0x08 != 0) try tmpStream.read(u8) else
0;
            repeats += 1;

            // if (flags & 0x02 != 0)
            xLen += @boolToInt(flags & 0x02 != 0) * repeats * @sizeof(u8);
            // else if (flags & 0x10 == 0)
            xLen += @boolToInt(flags & (0x02 | 0x10) == 0) * repeats *
@sizeof(i16);

            // if (flags & 0x04 != 0)
            yLen += @boolToInt(flags & 0x04 != 0) * repeats * @sizeof(u8);
            // else if (flags & 0x20 == 0)
            yLen += @boolToInt(flags & (0x04 | 0x20) == 0) * repeats *
@sizeof(i16);

            pointIdx += repeats;
        }

        const fLen = @ptrToInt(tmpStream.bytes.ptr) -
@ptrToInt(stream.bytes.ptr);
        glyphFlags = try stream.readStream(fLen);
        xCoordinates = try stream.readStream(xLen);
        yCoordinates = try stream.readStream(yLen);
    }

    var contours = std.ArrayList(Contour).init(std.heap.page_allocator);
    var currContour = try contours.addOne();
    var contourIdx: u16 = 0;
    var lastPointIdx: u16 = endPtsOfContours.get(contourIdx);
    var pointIdx: u16 = 0;
    var xPrev: i16 = 0;

```

```

var yPrev: i16 = 0;
var repeats: u8 = 0;
var flags: u8 = 0;

while (pointIdx < numPts) : (pointIdx += 1) {
    if (repeats == 0) {
        flags = glyphFlags.read(u8);
        repeats = if (flags & 0x08 != 0) glyphFlags.read(u8) else 0;
    } else {
        repeats -= 1;
    }

    if (flags & 0x02 != 0) {
        var abs: i16 = xCoordinates.read(u8);
        xPrev += if (flags & 0x10 != 0) abs else -abs;
    } else if (flags & 0x10 == 0) {
        xPrev += xCoordinates.read(i16);
    }

    if (flags & 0x04 != 0) {
        var abs: i16 = yCoordinates.read(u8);
        yPrev += if (flags & 0x20 != 0) abs else -abs;
    } else if (flags & 0x20 == 0) {
        yPrev += yCoordinates.read(i16);
    }

    const point = GlyphPoint{
        .x = xPrev,
        .y = yPrev,
        .onCurve = (flags & 0x01 != 0),
        .endOfCurve = (pointIdx == lastPointIdx),
    };
    try currContour.segments.addOne(point);

    if (pointIdx == lastPointIdx) {
        contourIdx += 1;
        if (contourIdx < numberOfContours) {
            lastPointIdx = endPtsOfContours.get(contourIdx);
            currContour = try contours.addOne();
        } else break;
    }
}

return Glyph{
    .scale = Vector2.new(xMax - xMin, yMax - yMin),
    .contours = contours,
};
}
};

const Stream = struct {
    bytes: []const u8,
    pub fn new (bytes: []const u8) Stream {
        return .{ .bytes = bytes };
    }
    pub fn read (self: *Stream, comptime T: type) FontError!T {
        if (self.bytes.len < @sizeof(T)) {
            return error.InvalidFont;
        }
        const value = mem.readIntForeign(T, &self.bytes[0]);
        self.bytes = self.bytes[@sizeof(T)..];
        return value;
    }
}

```

```

    pub fn readSoaField (self: *Stream, comptime T: type, soaLen: u32)
FontError!SoaReader(T) {
    if (self.bytes.len < soaLen * @sizeof(T)) {
        return error.InvalidFont;
    }
    const reader = SoaReader(T).new(self.bytes.ptr);
    self.bytes = self.bytes[soaLen * @sizeof(T)..];
    return reader;
}
    pub fn readStream (self: *Stream, streamLen: usize) FontError!CheckedStream
{
    if (self.bytes.len < streamLen) {
        return error.InvalidFont;
    }
    const reader = CheckedStream.new(self.bytes.ptr);
    self.bytes = self.bytes[streamLen..];
    return reader;
}
    pub fn readArrayToEnd (self: *Stream, comptime T: type) ArrayReader(T) {
    const reader = ArrayReader(T).fromBytes(self.bytes);
    self.bytes = self.bytes[self.bytes.len..];
    return reader;
}
    pub fn skip (self: *Stream, comptime T: type) void {
    const advanceBy = @minimum(self.bytes.len, @sizeof(T));
    self.bytes = self.bytes[advanceBy..];
}
    pub fn skipArray (self: *Stream, comptime T: type, len: u16) void {
    const advanceBy = @minimum(self.bytes.len, len * @sizeof(T));
    self.bytes = self.bytes[advanceBy..];
}
};

const FontError = error {
    InvalidFont,
    UnsupportedFont,
};

pub const Font = struct {
    bytes: TableBytes,
    locaTbl: LocaTable,
    glyfBytes: TableBytes,

    pub fn new (bytes: TableBytes) FontError!Font {
        const tableDir = try TableDirectory.fromBytes(bytes);
        if (!tableDir.sfntVersion.equal(Tag.new("\x00\x01\x00\x00"))) {
            return error.UnsupportedFont;
        }
        const tblRecords = try tableDir.tableRecords(bytes);

        const headTbl = try TableDirectory.findTable(HeadTable, tblRecords,
bytes);
        const indexToLocFormat = @byteSwap(headTbl.indexToLocFormat);
        if (indexToLocFormat > 1) {
            return error.UnsupportedFont;
        }
        const glyfBytes = try TableDirectory.findTableBytes(GlyfTable,
tblRecords, bytes);
        const locaBytes = try TableDirectory.findTableBytes(LocaTable,
tblRecords, bytes);
        const locaTbl = LocaTable.fromBytes(locaBytes, indexToLocFormat != 0);
        const maxpTbl = try TableDirectory.findTable(MaxpTable, tblRecords,
bytes);
        const numGlyphs = @byteSwap(maxpTbl.numGlyphs);

```

```

    if (numGlyphs < 2) {
        return error.InvalidFont;
    }
    return .{ .bytes = bytes, .locaTbl = locaTbl, .glyphBytes = glyphBytes };
}

pub fn getGlyphById (self: Font, id: u16) FontError!Glyph {
    const glyphBytes = try self.locaTbl.getGlyphBytes(id, self.glyphBytes);
    return try GlyphTable.parseGlyph(glyphBytes);
}
};

```

A.8. Файл `glyph_from_msdf.wgsl`

```

struct Uniforms {
    viewProj: mat4x4<f32>,
    color: vec4f<f32>,
    effectColor: vec4f<f32>,
    effectDistance: vec2f<f32>,
    effectKind: u32,
    sdfRadius: f32,
};

@group(0) @binding(0) var<uniform> uniforms : Uniforms;

@group(1) @binding(0) var bisampler: sampler;
@group(1) @binding(1) var atlas: texture_2d<f32>;

struct VertexInput {
    @location(0) position: vec3f,
    @location(1) uv: vec2f,
};

struct VertexOutput {
    @builtin(position) position: vec4f,
    @location(0) uv: vec2f,
};

@vertex
fn vertex_main(ver: VertexInput) -> VertexOutput {
    var out: VertexOutput;
    out.position = uniforms.viewProj * vec4f(ver.position, 1.0);
    out.uv = ver.uv;
    return out;
}

fn median(r: f32, g: f32, b: f32) -> f32 {
    return max(min(r, g), min(max(r, g), b));
}

fn screenPxRange(uv: vec2f) -> f32 {
    var unitRange = vec2f(uniforms.sdfRadius) /
    vec2f(textureDimensions(atlas));
    var screenTexSize = vec2f(1.0) / fwidth(uv);
    return max(0.5*dot(unitRange, screenTexSize), 1.0);
}

@fragment
fn fragment_main(in: VertexOutput) -> @location(0) vec4f {
    var msd = textureSample(atlas, bisampler, in.uv).rgb;
    var sd = median(msd.r, msd.g, msd.b);
}

```

```

var dist = abs(sd - 0.5) * 2;
var outColor = uniforms.color.rgb;
var opacity = 1.0;
if (uniforms.effectKind == 1) {
    // outline effect
    if (dist > uniforms.effectDistance.x && dist < uniforms.effectDistance.y)
    {
        outColor = uniforms.effectColor.rgb;
    }
} else if (uniforms.effectKind == 2) {
    // shadow effect
    if (dist > uniforms.effectDistance.x) {
        opacity = smoothstep(
            uniforms.effectDistance.x, uniforms.effectDistance.y, dist);
    }
}

// antialiasing
var screenPxDistance = screenPxRange(in.uv) * (sd - 0.5);
opacity *= clamp(screenPxDistance + 0.5, 0.0, 1.0);
if (opacity == 0) {
    discard;
}
return vec4f(outColor, uniforms.color.a * opacity);
}

```

A.9. Файл msdf.zig

```

const std = @import("std");
const math2d = @import("../util/math2d.zig");
const Vector2 = math2d.Vector2;
const BoundingBox = math2d.BoundingBox;
const BezierQuad = math2d.BezierQuad;
const math3d = @import("../util/math3d.zig");
const Vector3 = math3d.Vector3;
const Font = @import("../font_reader.zig").Font;

const GLYPH_SIZE = 64;
const DIST_RADIUS = 8;
const SHARP_TOLERANCE = 0.02;

const MsdfCache = struct {
    glyphMap: std.AutoArrayHashMap(u16, BoundingBox),
    font: Font,
    usedSlots: u32,
    renderer: GlyphMsdfRenderer,

    pub fn init(font: Font) MsdfCache {
        return MsdfCache{
            .glyphMap = std.AutoArrayHashMap(u16,
BoundingBox).init(std.heap.page_allocator),
            .font = font,
            .renderer = GlyphMsdfRenderer.init(),
        };
    }

    pub fn getGlyphFromAtlas (self: *MsdfCache, id: u16) BoundingBox {
        if (self.glyphMap.contains(id)) {
            return self.glyphMap.get(id);
        }
    }
}

```

```

    }

    const glyph = glyphlbl: {
        self.font.getGlyph(id) catch {
            if (self.glyphMap.contains(0)) {
                return self.glyphMap.get(0);
            }
            break :glyphlbl self.font.getGlyph(0);
        };
    };

    const bbox = BoundingBox.new(
        Vector2.new(GLYPH_SIZE * usedSlots, 0),
        Vector2.new(GLYPH_SIZE * usedSlots + GLYPH_SIZE, GLYPH_SIZE),
    );
    self.usedSlots += 1;
    self.glyphMap.put(id, bbox);
    return bbox;
}
};

const BezSegment = extern struct {
    points: Vector2[3],
    color: Vector3,

    fn writeToBuffer (self: BezSegment, buffer: JsRef, n: u32) void {
        papiGpuWriteBuffer(deviceRef, buffer, n * @sizeof(BezSegment),
        @ptrCast([*]const u8, &self), @sizeof(BezSegment));
    }
};

const GlyphMsdfRenderer = struct {
    uniformBuffer: JsRef,
    segmentsBuffer: JsRef,
    outlineBuffer: JsRef,
    outlinePipeline: JsRef,
    stencilBuffer: JsRef,
    stencilPipeline: JsRef,

    pub fn init () GlyphMsdfRenderer {
        {
            uniformBuffer = papiGpuCreateBuffer(deviceRef,
                36,
                GPUBufferUsage.Uniform | GPUBufferUsage.CopyDst
            );
            const layoutEntries = [_]GPUBindGroupLayoutEntry{
                .{
                    .kind = GPUBindGroupLayoutEntry.Kind.Buffer,
                    .binding = 0,
                    .visibility = GPUShaderStage.Vertex | GPUShaderStage.Fragment,
                    .data = .{
                        .buffer = .{
                            .type = GPUBufferBindingType.Uniform,
                            .hasDynamicOffset = false,
                            .minBindingSize = 36,
                        },
                    },
                },
            };
            var bindGroupLayout = papiGpuCreateBindGroupLayout(deviceRef,
                layoutEntries.len, &layoutEntries);
            const bindGroupLayouts = [_]JsRef{bindGroupLayout};
            stencilPipelineLayout = papiGpuCreatePipelineLayout(deviceRef,
                bindGroupLayouts.len, &bindGroupLayouts);
        }
    }
};

```

```

{
    const bufferLayouts = [_]u8{
        GPUVertexBufferLayout.serialize(1, GPUVertexStepMode.Vertex),
        GPUVertexBufferLayout.serialize(1, GPUVertexStepMode.Vertex),
        0, undefined,
    };
    var vertexAttributes = [_]GPUVertexAttribute{
        {.format = GPUVertexFormat.Float32x2, .shaderLocation = 0 },
        {.format = GPUVertexFormat.Float32x2, .shaderLocation = 1 },
        {.format = GPUVertexFormat.Uint32, .shaderLocation = 3 },
    };

    var fragState = GPUFragmentState{
        .module = outlineShader,
    };

    const psoDesc = GPURenderPipelineDescriptor{
        .layout = perCamPipelineLayout,
        .vertex = {.
            .module = outlineShader,
            .buffers = bufferLayouts,
            .attributes = &vertexAttributes,
        },
        .fragment = &fragState,
    };
    outlinePipeline = papiGpuCreateRenderPipeline(deviceRef, &psoDesc);
}

pub fn render (self: *GlyphMsdfRenderer, glyph: Glyph, pos: BoundingBox)
void {
    var segIdx = 0;
    for (glyph.contours) |contour| {
        var color = EdggleColor.new();
        var startPoint = contour.segments[0].p0;
        for (contour.segments) |segment, i| {
            if (i != 0 and
                !isSegmentsContinuos(contour.segments[i - 1], segment)) {
                color.flip();
            }

            const seg = BezSegment{
                .points = segment.points,
                .color = color.color,
            };
            seg.writeToBuffer(self.segmentsBuffer, segIdx);
            segIdx += 1;

            const segBbox = segment.boundingBox()
                .expandedBy(DIST_RADIUS);

            OutlineVertex.writeToBuffer(self.outlineBuffer, segIdx, segBbox);
        }
    }

    papiGpuSetBindGroup(0, outlineBindGroup);
    papiGpuSetPipeline(outlinePipeline);
    papiGpuSetVertexBuffer(0, outlineBuffer, 0);
    papiGpuDraw(segIdx * 6);
    papiGpuSetPipeline(stencilPipeline);
    papiGpuSetVertexBuffer(0, stencilBuffer, 0);
    papiGpuDraw(segIdx * 6 + 6);
}
};

```

```

const OutlineVertex = extern struct {
    position: Vector2,
    uv: Vector2,
    segIdx: u32,

    fn new (pos: Vector2, uv: Vector2, segIdx: u32) OutlineVertex {
        return OutlineVertex{
            .position = pos,
            .uv = uv,
            .segIdx = segIdx,
        };
    }

    fn writeToBuffer (buffer: JsRef, n: u32, bbox: Vector2) void {
        const tri2 = [_]OutlineVertex{
            // tri1
            OutlineVertex.new(Vector2.new(bbox.max.x,    bbox.min.y),    Vector2.new(1,
0), n),
            OutlineVertex.new(Vector2.new(bbox.min.x,    bbox.min.y),    Vector2.new(0,
0), n),
            OutlineVertex.new(Vector2.new(bbox.max.x,    bbox.max.y),    Vector2.new(1,
1), n),
            // tri2
            OutlineVertex.new(Vector2.new(bbox.min.x,    bbox.min.y),    Vector2.new(0,
0), n),
            OutlineVertex.new(Vector2.new(bbox.min.x,    bbox.max.y),    Vector2.new(0,
1), n),
            OutlineVertex.new(Vector2.new(bbox.max.x,    bbox.max.y),    Vector2.new(1,
1), n),
        };
        papiGpuWriteBuffer(deviceRef,    buffer,    n    *    @sizeof(OutlineVertex),
@ptrCast([]*]const u8, &self), @sizeof(OutlineVertex));
    }
};

const EdggleColor = struct {
    color: Vector3,

    pub fn new () EdggleColor {
        return EdggleColor{ .color = Vector3.new(1, 0, 1), };
    }

    pub fn flip (self: *EdggleColor) void {
        if (self.color.x == 1 and self.color.y == 1 and self.color.z == 0) {
            self.color = Vector3.new(0, 1, 1);
        } else {
            self.color = Vector3.new(1, 1, 0);
        }
    }
};

fn isSegmentsContinuos (prev: BezierQuad, next: BezierQuad) bool {
    var vec1 = Vector2(prev.p2, prev.p1).normalize();
    var    vec2    =    Vector2(next.p0,    if    (next.isLine())    next.p2    or else
next.p1).normalize();
    const dot = @fabs(vec1.dot(vec2));
    return (1 - dot < SHARP_TOLERANCE);
}

```

A.10. Файл msdf_outline.wgsl

```

const TAU = 6.2831853071795862;

fn cbrt (e: f32) -> f32 {
    return sign(e) * pow(abs(e), 1.0/3);
}

fn cross2 (e1: vec2f, e2: vec2f) -> f32 {
    return e1.x * e2.y - e2.x * e1.y;
}

struct Uniforms {
    transform: mat3x3<f32>,
    sdfRadius: f32,
};
@group(0) @binding(0) var<uniform> uniforms : Uniforms;

struct BezSegment {
    points: array<vec2f, 3>,
    color: vec3f,
};
@group(1) @binding(0) var<storage> segments : array<BezSegment>;

struct VertexInput {
    @location(0) position: vec2f,
    @location(1) uv: vec2f,
    @location(2) segIdx: u32,
};

struct VertexOutput {
    @builtin(position) position: vec4f,
    @location(0) uv: vec2f,
    @location(1) @interpolate(flat) segIdx: u32,
};

@vertex
fn vertex_main(vert: VertexInput) -> VertexOutput {
    var out: VertexOutput;
    out.position = (uniforms.transform * vec3f(vert.position, 1.0)).xy;
    out.uv = vert.uv;
    out.segIdx = vert.segIdx;
    return out;
}

fn bezierLinear (t: f32, p0: vec2f, p1: vec2f) -> vec2f {
    // p0 + t * (p1 - p0);
    return mix(p0, p1, t);
}

fn bezierLinearD1 (p0: vec2f, p1: vec2f) -> vec2f {
    return p1 - p0;
}

fn bezierQuadratic (t: f32, p0: vec2f, p1: vec2f, p2: vec2f) -> vec2f {
    return p0 + 2*t * (p1 - p0) + t*t * (p2 - 2*p1 + p0);
}

fn bezierQuadraticD1 (t: f32, p0: vec2f, p1: vec2f, p2: vec2f) -> vec2f {
    return 2 * (p1 - p0) + 2*t * (p2 - 2*p1 + p0);
}

fn distToLine (p: vec2f, a: vec2f, b: vec2f) -> f32 {
    var ab = b - a;
    var pa = a - p;
    return abs(cross2(ab, pa)) / distance(a, b);
}

```

```

}

// Signed distance
// p1 ^
//   |//
// (-) |// (+)
//   |//
// p0 |//

fn distToLineSeg (p: vec2f, a: vec2f, b: vec2f) -> f32 {
    var ab = b - a;
    var ap = p - a;
    var t = dot(ap, ab) / dot(ab, ab);
    t = clamp(t, 0, 1);
    var closestP = bezierLinear(t, a, b);
    // if (sdist) {
    //     return dist * sign(cross2(ab, closestP - p));
    // }
    return distance(p, closestP);
}

struct CubicRoots { n: u32, x: vec3f };

// x^3 + a2*x^2 + a1*x + a0 = 0
fn solveCubicNorm (a2: f32, a1: f32, a0: f32) -> CubicRoots {
    var roots: CubicRoots;
    var q = (3*a1 - a2*a2) / 9;
    var q3 = q*q*q;
    var r = (9*a2*a1 - 27*a0 - 2*a2*a2*a2) / 54;
    var r2 = r*r;
    var d = q3 + r2;

    if (d < 0.0) {
        roots.n = 3;
        var theta = acos(r / sqrt(-q3));
        const k = vec3f(0, 1, 2) * TAU / 3;
        roots.x = 2*sqrt(-q)*cos(theta/3 + k);
    } else if (d > 0.0) {
        roots.n = 1;
        var sqrtD = sqrt(d);
        var s = cbrt(r + sqrtD);
        var t = cbrt(r - sqrtD);
        // can use when p1 is always inside of p0,p2 rectangle
        // var t = -q/s;
        roots.x = vec3f(s + t);
    } else {
        roots.n = 2;
        var cbrtR = cbrt(r);
        roots.x = vec3f(2*cbrtR, cbrtR, cbrtR);
    }
    return roots;
}

fn solveCubic (a: f32, b: f32, c: f32, d: f32) -> CubicRoots {
    var roots = solveCubicNorm(b/a, c/a, d/a);
    roots.x -= b/(3*a);
    return roots;
}

fn distToQuadBez (p: vec2f, p0: vec2f, p1: vec2f, p2: vec2f) -> f32 {
    // finding t from: dot(bez(t) - p, bezD1(t)) = 0
    var v0 = p0 - p;
    var v1 = p1 - p0;
    var v2 = p2 - 2*p1 + p0;

```

```

var a = dot(v2, v2);
var b = 3 * dot(v1, v2);
var c = 2 * dot(v1, v1) + dot(v0, v2);
var d = dot(v0, v1);
var roots = solveCubic(max(a, 1e-6), b, c, d);

var minDist = distance(p, bezierQuadratic(0.0, p0, p1, p2));
{
    var testP = bezierQuadratic(1.0, p0, p1, p2);
    var dist = distance(p, testP);
    if (dist < minDist) {
        minDist = dist;
    }
}

for (var i: u32 = 0; i < roots.n; i++) {
    var testT = roots.x[i];
    if (testT <= 0 || testT >= 1) { continue; }
    var testP = bezierQuadratic(testT, p0, p1, p2);
    var dist = distance(p, testP);
    if (dist < minDist) {
        minDist = dist;
    }
}

return minDist;
}

@fragment
fn fragment_main(in: VertexOutput) -> @location(0) vec4f {
    var seg = segments[in.segIdx];
    var dist: f32;
    if (seg.points[0] == seg.points[1]) {
        dist = distToLineSeg(in.uv, seg.points[0], seg.points[2]);
    } else {
        dist = distToQuadBez(in.uv, seg.points[0], seg.points[1], seg.points[2]);
    }
    dist = min(dist / uniforms.sdfRadius, 1.0);
    var out = 0.5 - dist / 2;
    if (out == 0.0) {
        discard;
    }
    return vec4f(seg.color * out, 1.0);
}

```

A.11. Файл msdf_stencil_bezier.wgsl

```

struct Uniforms {
    transform: mat3x3<f32>,
};
@group(0) @binding(0) var<uniform> uniforms : Uniforms;

struct VertexInput {
    @location(0) position: vec2f,
    @location(1) uv: vec2f,
};

struct VertexOutput {
    @builtin(position) position: vec4f,
    @location(0) uv: vec2f,
};

```

```
};

@vertex
fn vertex_main(vert: VertexInput) -> VertexOutput {
    var out: VertexOutput;
    out.position = (uniforms.transform * vec3f(vert.position, 1.0)).xy;
    out.uv = vert.uv;
    return out;
}

@fragment
fn fragment_main(in: VertexOutput) -> @location(0) vec4f {
    if (in.uv.x * in.uv.x - in.uv.y < 0) {
        return vec4f(1.0);
    } else {
        discard;
    }
}
```

ДОДАТОК Б
Слайди презентації

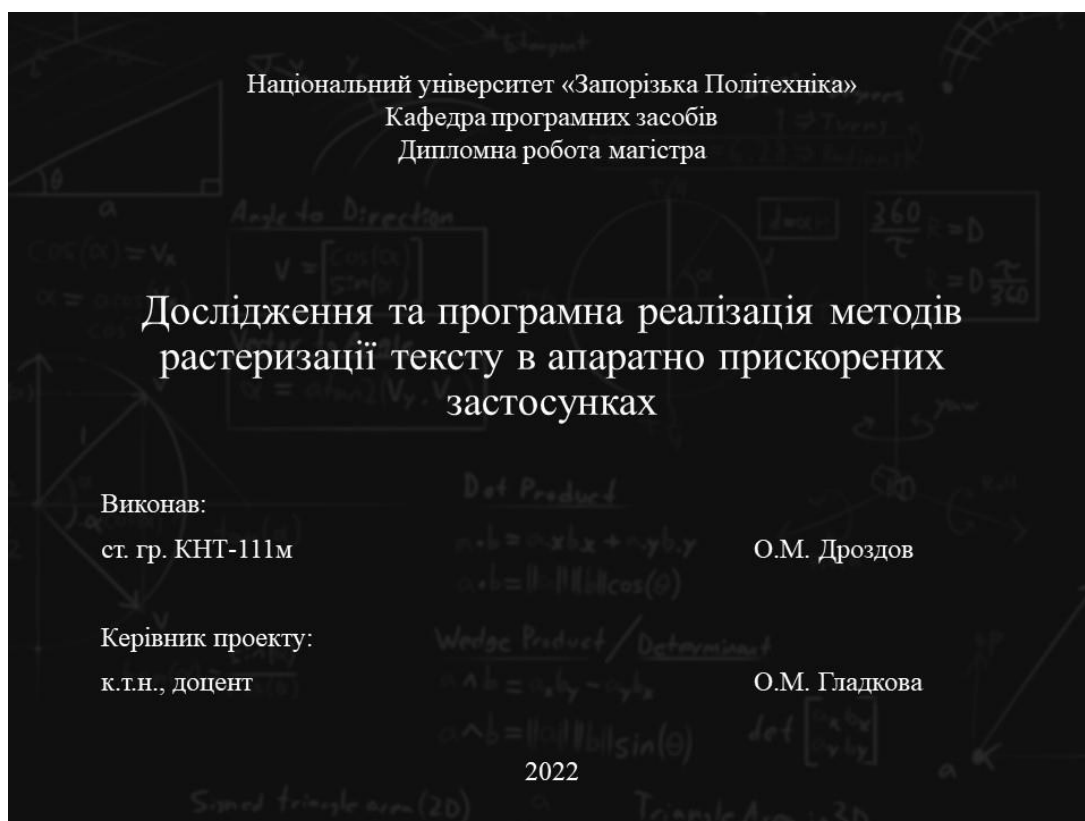


Рисунок Б.1 – Слайд 1

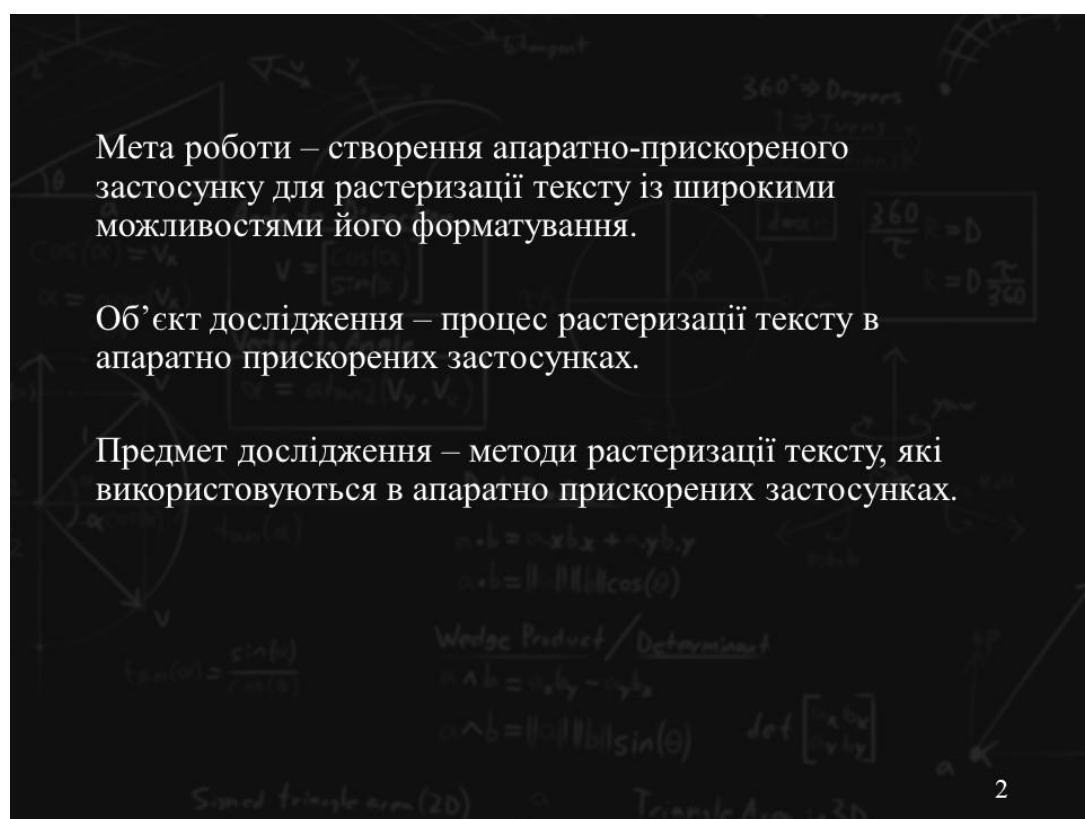


Рисунок Б.2 – Слайд 2

Завдання

- аналіз вимог до програмного забезпечення для растеризації тексту із використанням апаратного прискорення;
- аналіз методів растеризації тексту в апаратно прискорених застосунках;
- вибір засобів розробки;
- проєктування програмного забезпечення;
- розроблення програмного забезпечення;
- тестування програмного забезпечення.

3

Рисунок Б.3 – Слайд 3

Класифікація методів



4

Рисунок Б.4 – Слайд 4

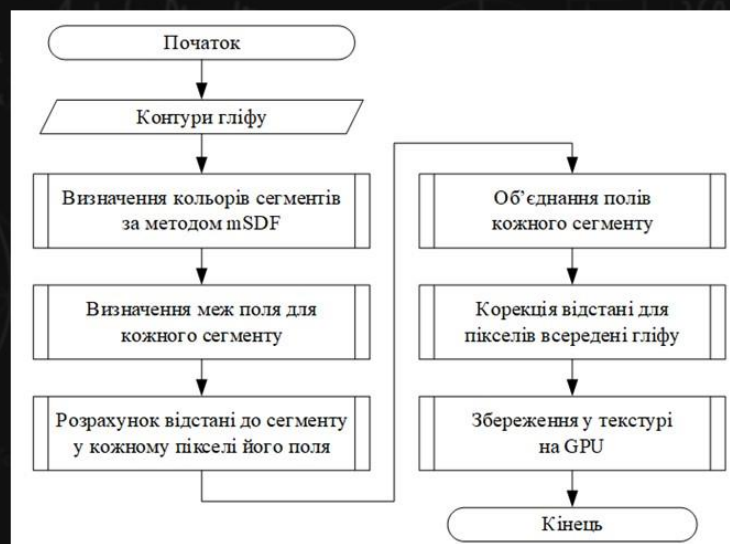
Порівняння існуючих методів

Назва методу	Відносна швидк. (GPU)	Растрівання прямо з кривих	Популярність	Головний недолік	Патенти (США)
Попередньо-растровані гліфи	1.0	Ні	+	Усі недоліки растрових зображень	–
SDF	1.2	Ні	+++	Кути контурів стають округленими	–
mSDF	1.6	Ні	++	Можливі артефакти у тонких контурах	–
Loop-Blinn геометричне представлення	2.0	Так		Потребує складний крок триангуляції	до 2026 (кубічні криві)
Kokojima буфер трафарету	2.0	Так		Потребує буфер трафарету	–
Dobbie	32.0	Так		Числово нестійкий	–
Slug	14.0	Так	+	Відсутність м'яких тіней	до 2038

5

Рисунок Б.5 – Слайд 5

Модифікований метод створення mSDF на GPU



6

Рисунок Б.6 – Слайд 6

Функціональний розподіл між компонентами



7

Рисунок Б.7 – Слайд 7

Вибір мови програмування

Мова	Швидкість ітерацій	Управління залежностями	Популярність, бібліотеки	Повідомлення, відладка	Головний недолік
C/C++	++	+	+++	+	Спадщина сумісності коду
Rust	+	+++	++	+++	Крута крива навчання
Zig	+++	++	+	++	Не випущено стабільної версії

8

Рисунок Б.8 – Слайд 8

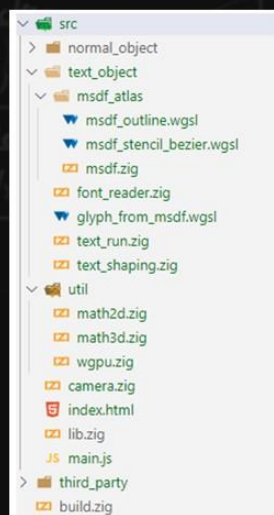
Вибір графічного API

API	Мова шейдерів	Платформа	Легкість використ.	Можливості GPU
OpenGL	GLSL	Android, Linux, Windows	++++	++
Vulkan			+	++++
Metal	MSL	macOS, iOS	+++	+++
Direct3D11	HLSL	Windows	++++	++
Direct3D12			++	++++
WebGL2	GLSL	Браузер	++++	+
WebGPU	WGSL		+++	+++

9

Рисунок Б.9 – Слайд 9

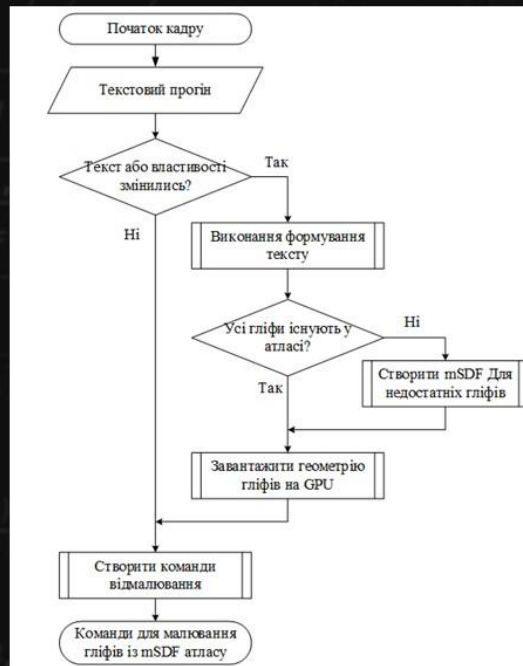
Структура програмної реалізації



10

Рисунок Б.10 – Слайд 10

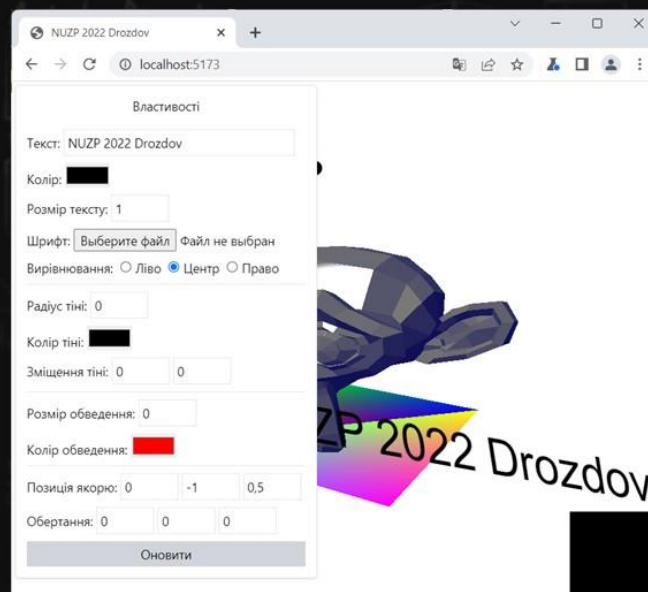
Алгоритм відмалювання тексту у кадрі



11

Рисунок Б.11 – Слайд 11

Інтерфейс форми властивостей тексту



12

Рисунок Б.12 – Слайд 12

Висновки

Проведено огляд методів растеризації гліфів, виконано їх порівняння та обрано метод mSDF. Розглянуто причини необхідності створення SDF на GPU. Для реалізації застосунку використано мову програмування Zig, графічне API WebGPU.

Науковою новизною роботи є порівняння методів растеризації тексту, запропонований модифікований метод для створення mSDF на GPU, а також полів псевдо-дистанцій.

Практична цінність роботи полягає в розробленому застосунку, який дозволяє переконавшись в інтерактивній формі, чи задовольняє метод mSDF бажану якість растеризації тексту і якщо так, то переконавшись у можливості швидкого створення mSDF атласу під час виконання.

13

Рисунок Б.13 – Слайд 13