НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.928

«До захисту допущено»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___»_____2020 р.

Магістерська дисертація

на здобуття ступеня магістра

освітньо-науковою програмою
«Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Алгоритмічно-програмний метод генерації LOD-моделей

Виконав:
студент ІІ курсу, групи КП-81мн
Лось Ігор Анатолійович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н., доцент,
Сулема Євгенія Станіславівна _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри СПСКС, к.т.н., доцент,
Боярінова Юлія Євгенівна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ

Науковий керівник кафедри

_____ Іван ДИЧКА

«___»_____2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Лосю Ігорю Анатолійовичу

1. Тема дисертації «Алгоритмічно-програмний метод генерації LOD-моделей», науковий керівник дисертації Сулема Євгенія Станіславівна, к.т.н., доцент, затверджена наказом по університету від «07» квітня 2020 р. №964-С

2. Термін подання студентом дисертації «15» травня 2020 р.

3. Об'єкт дослідження: процес створення тривимірних анімованих LOD моделей.

4. Предмет дослідження: методи та алгоритми спрощення тривимірних моделей.

5. Перелік завдань, які потрібно вирішити:
   - провести аналіз існуючих методів роботи з тривимірними моделями в контексті спрощення;
   - дослідити методи генерації LOD-моделей;
   - розробити реалізацію запропонованого методу;
   - обрати критерії оцінки методу, створити тестовий набір та провести тестування;
   - розробити користувацький інтерфейс для експлуатації технології користувачем;
   - провести аналіз отриманих результатів.

6. Орієнтовний перелік публікацій:
   - Тези доповіді " Алгоритм генерації LOD для анімованих моделей зі скелетом"
   - Стаття "Генерація LOD для моделей зі скелетом"

7. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., к.т.н., доцент		

8. Дата видачі завдання «11» жовтня 2018 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	17.12.2018	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.03.2019	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	16.05.2019	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення; підготовка матеріалів доповіді на конференції ПМК-2019	14.10.2019	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	15.12.2019	
6.	Проведення наукового дослідження; робота над третім та четвертим розділами магістерської дисертації	20.02.2020	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу	16.04.2020	
8.	Оформлення текстової і графічної частини магістерської дисертації	13.05.2020	

Студент Ігор ЛОСЬ

Науковий керівник Євгенія СУЛЕМА

РЕФЕРАТ

Актуальність теми. Тривимірні моделі для додатків, що працюють в реальному часі, часто створюються з надлишковою деталізацією. Високодеталізовані моделі використовуються в промоматеріалах, проте для швидкої роботи програми частіше за все потрібні меньш деталізовані моделі. В таких випадках моделі спрощуються вручну, або з допомогою спеціальних алгоритмів. Для анімованих моделей до цього додається також крок скінінгу моделі, що потребує роботи професіонала. Як наслідок, є необхідність спрощувати тривимірну анімовану модель, разом з її рескінінгом, та збереженням високої якості анімації. Для реалізації такої можливості пропонується розробити метод створення LOD для анімованих моделей. LOD називають сукупність методів, які полягають у заміні детальних моделей меньш детальними. LOD часто застосовують для підтримання високої працездатності системи при роботі в часі виконання.

Об'єктом дослідження є процес створення тривимірних анімованих LOD моделей.

Предметом дослідження є методи та алгоритми спрощення тривимірних моделей.

Мета роботи полягає у розробці ефективного методу спрощення анімованих тривимірних моделей без втрати якості анімації.

Методи дослідження: в роботі використовуються методи теоретичного дослідження: аналіз та синтез. Також застосовувалися емпіричні методи: експеримент, вимірювання та порівняння.

Наукова новизна роботи полягає у розробленні алгоритмічно-програмного методу генерації LOD-моделей для моделей зі скелетом, перевагогю якого є вища якість анімації (10-40%) за рахунок етапу перерахуванян ваг моделі. Розроблений метод може бути використаний для будь-якої моделі зі скелетом.

Також був запропонований модифікований критерій для вимірювання точності спрощених моделей та запропонований критерія для вимірювання якості топології полігональної сітки.

Практична цінність отриманих результатів роботи полягає в тому, що запропонований метод дає змогу підвищити точність анімації в LOD моделях зі скелетом, та краще зберігає топологію моделі.

Апробація роботи. Основні положення і результати роботи доповідалися та обговорювалися на XII науковій конференції магістрантів та аспірантів “Прикладна математика та комп’ютинг” ПМК-2019.

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень.

У першому розділі описані основні засади використання LOD моделей, оглянуті відомі рішення до створення LOD для моделей зі скелетом.

У другому розділі розглянуто основні принципи використання, створення та вимірювання помилки на LOD моделях. Запропонований модифікований метод на основі квадратичної помилки для генерації LOD для анімованих моделей.

У третьому розділі сформовані основні вимоги до додатку, що реалізує запропонований метод. Проведено аналіз інструментів для розробки, описано архітектуру реалізації.

У четвертому розділі визначено критерії оцінки ефективності, які застосовуються до розробленого методу; наведена інформація про дані, що використовувались при аналізі ефективності; проведений аналіз ефективності запропонованого методу створення LOD для анімованих моделей.

У висновках проаналізовано отримані результати роботи.

Робота виконана на 70 аркушах, містить 2 додатки та посилання на список використаних літературних джерел з 20 найменувань. У роботі наведено 33 рисунок та 3 таблиці.

Ключові слова: скелетна анімація, спрощення тривимірної моделі, квадратична помилка.

ABSTRACT

Urgency. 3D models, made for real-time applications, are often made with redundant level of detail. High detail models can be used in promotional material but for high performance during work of application less detailed models are usually used. In such cases models are often simplified by special algorithms. For animated model there is also added step of skinning the model, which requires a work of professional. As a result, there is a necessity for simplification of 3D model, simultaneously reskinning it, and also saving high animation quality. For this we have to develop a method for creation of LOD for animated models. LOD is a sum of methods for exchanging high detailed model with lower detailed one. LOD is often used for maintaining a high performance during runtime.

Object of research is process of creation of animated LOD models.

Subjects of research are methods and algorithms for simplification of 3D models.

Goal of the work is to develop an effective method of simplification of animated 3d models without compromising the animation quality.

Methods of research include methods of theoretical research: analysis and synthesis. Also there were used empirical methods: experiment, measurement and comparison.

Scientific novelty of the work is to develop a method of simplification of 3D models with skeleton which has better animation quality (10-40%) as a result of recalculating weights of resulting model. Developed method can be used on any animated model with a skeleton.

Also modified method for calculating error of simplified model and a method for calculating quality of topology where described.

Practical value of the received results of work is that the proposed method allows increase the accuracy of animation in LOD models with skeleton, and preserves the topology.

Approbation. The results of the research have been presented and discussed at the 12th scientific conference for graduate and postgraduate students “Applied mathematics and computing” PMK-2019.

Structure and content of the thesis. Master's thesis consists of an introduction, four chapters, conclusions and appendices.

The introduction provides a general description of the work, evaluated the current state of the problem, substantiated the relevance of the research direction, formulated the purpose and objectives of the study.

The first chapter describes the basic principles of LOD, evaluated existing methods for simplification of animated models.

The second chapter describes main principles of use of LOD, creation of simplified models and evaluating their errors. A method for simplification of animated models, based on quadric error metric, was proposed.

In the third section, the main requirements for an add-on, that implements discussed methods, are formed; the choice of the means used during development was substantiated. Conducted an analysis of instruments, for creation of such add-on, provided an architecture of this implementation.

The fourth chapter defines the criteria for assessing the effectiveness of the developed method; provides information on the data used in the analysis of efficiency; the analysis of the efficiency for the modified and basic methods of automated decision making on simplification of animated models

The conclusion contains brief overview of the results obtained in the work.

The work is done on 70 pages, contains 2 appendices and reference list of 20 titles. The work contains 33 pictures and 3 tables.

Keywords: skeletal animation, quadratic error, mesh simplification

ЗМІСТ

СПИСОК ТЕРМІНІВ СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Кістка – об’єкт, що включає набір можливих трансформацій, які впливають на дочірні кістки, та піддається впливу трансформацій батьківської кістки. Скелет – набір ієрархічно організованих кісток. Вага кістки – міра впливу конкретної кістки на дану вершину.

LOD – технологія рендерингу, що дозволяє зменшити кількість операцій, шляхом заміни складних деталей простішими.

Мультирезолюційна модель, або *модель з фіксованою роздільною здатністю* – набір моделей з різним рівнем деталізації, що репрезентують один об’єкт.

Вершина – координата точки, яка містить додаткові атрибути (колір, uv-координати, тощо).

Полігон, або *тригон* – набір трьох вершин, що описує зону простору, в площині цих вершин, та обмежену трикутником, утвореним цими вершинами.

Ребро – набір двох верши, що належать одному полігону.

UV-координати – координати, що задають співвідношення між 3-вимірними координатами вершин та 2-вимірними координатами текстур

Відкрита (non-manifold) модель – тривимірна модель що має будь-яку комбінацію наступних характеристик: ребро, що має більше трьох полігонів;два або більше ребер що з’єднані лише вершиною, але не ребром; суміжні полігони, чиї нормалі напрямлені в протилежні сторони.

Закрита (manifold) модель – модель, яка не є відкритою.

Тривимірний редактор – програмний продукт, створений для 3D моделювання.

3D моделювання – створення математичного опису будь-яких тривимірних поверхонь та об'єктів.

Тривимірний рушій – програмний рушій, ціллю якого є створення тривимірних додатків, у тому числі ігор.

CSF, або *функція чутливості до контрасту* – функція, що показує відношення мінімальної помітної зміни контрасту між об’єктом та фоном, до кутового розміру об’єку.

ВСТУП

Однією з основних переваг рендеру в реальному часі, у порівнянні з пре-рендереною графікою є інтерактиність. Можливість користувача взаємодіяти з об’єктами віртуального світу є невід’ємною частиною деяких програмних продуктів. Постійні покращення обчислювальної потужності промислових та комерційних комп’ютерів і нові методи рендеру створюють нові можливості для використання комп’ютерної графіки в сферах в навчанні та медицині. Однак, не зважаючи на всі вдосконалення, завжди існує потреба рендеру складнішої сцени.

Алгоритми рендеру в реальному часі призначені для обчислення властивостей елементів нового кадру, під час відображення попереднього. Через це, методи що реалізують ці алгоритми мають бати достатнь швидкі для ствоерння іллюзії руху. На практиці, високодетальні тривимірні моделі надзвичайно важко рендерити за такі короткі проміжки часу через обмеження в обчислювальних ресурсах. Це особливо актуально для мобільних платформ, які швидко поширюються та часно мають нестачу обчислювальних ресурсів.

Часто елементи сцени надлишково деталізовані – залежно від характеристик об’єкту, як-то розмір об’єкту на екрані, його віддаленість від камери, видимість, освітлення та інше маніпуляції над цим об’єктом не потребують проводити маніпуляції над об’єктом та його рендер у повній деталізації. Одним вирішенням цієї проблеми є створення декількох версій однієї моделі в різних рівнях деталізації. Ці моделі називають LOD-моделями. Ця методика була поширена в ранніх літальних симуляторах, де кількість використовуваної пам’яті і швидкість рендеру були важливішими за естетичний вигляд моделей і зараз LOD-моделі широко використовуються в рендері реального часу. Створення LOD-моделей є ресурсозатратним процесом, яке вимагає ретельної роботи спеціаліста. Існують автоматичні методи, які успішно зменшуються складність моделі

разом із збереженням її загальний вигляд. Один з таких методів – спрощення полігональної сітки – застосовується до полігональних моделей.

Нажаль більшість таких алгоритмів неможливо застосувати до анімованих скелетних моделей, оскільки вони не враховують додаткових елементів моделі, як-то ваги вершин. Це вимагає додаткових затрат часу та зусиль. Є потреба для створення автоматизованого методу спрощення тривимірних моделей.

1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1. Основні засади LOD

Для зменьшення часу рендерингу одного кадру в тривимірній графіці часто використовуються прийоми, призвані для усунення надлишкової інформації з кадру. Серед них можна виділити back-face culling, clipping, z-culling, image-based rendering occlusion culling та інші. LOD-рендеринг – один з таких прийомів. Він заснований на заміні моделей на меньш детальні при збільшенні відстані до неї.

Зазвичай тривимірні рушії відмальовують модель повністю, витрачаючи на це однаковий час, незалежно від її відстані до камери, навіть якщо кінцевий користувач не зможе розрізнити ці деталі через відносно невеликий кутовий розмір відмальовуваного елементу на екрані.

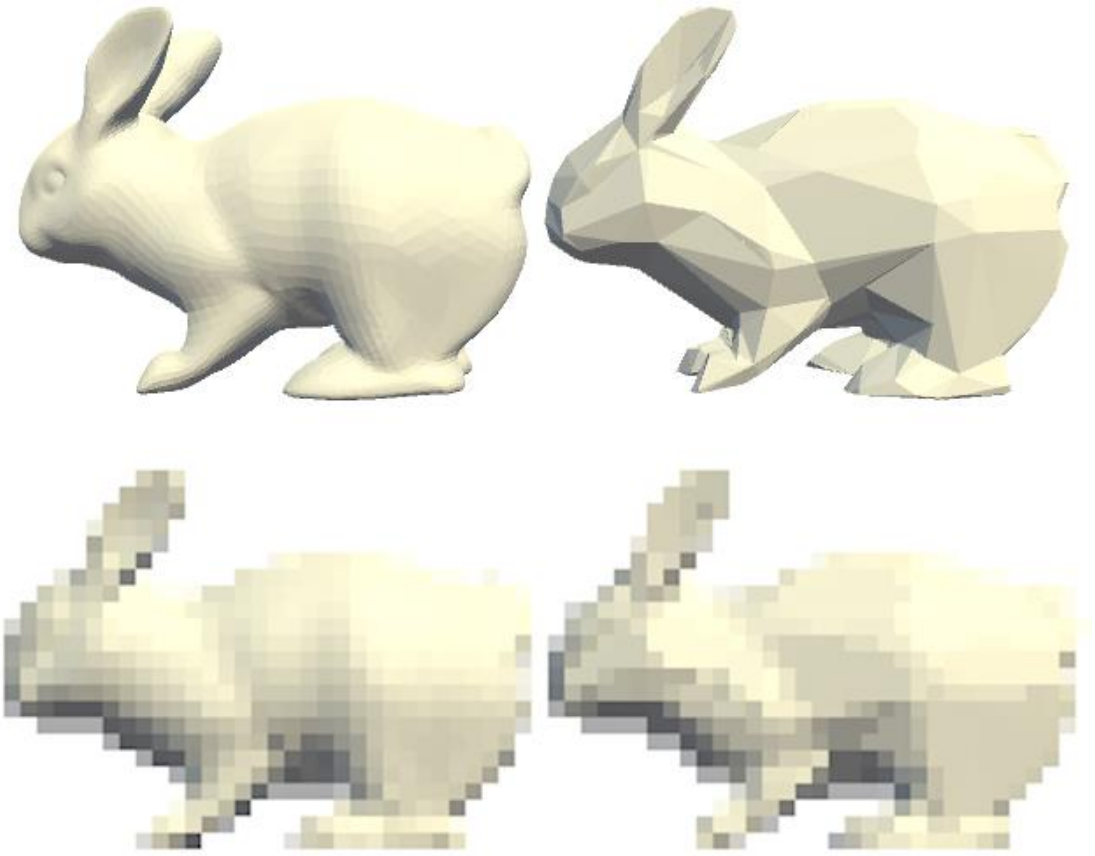


Рис. 1.1. Модель кролику, 14432 полігонів (зліва) та 856 (зправа) відрендерені висотою в 240 (зверху) та 24 (знизу) пікселі

Одним з рішень для покращення оптимізації в такому випадку є використання мультирезолюційних моделей [1,2], або LOD моделей.

Можливим використання LOD роблять 2 фактори. Перший – це скінченна роздільна здатність сучасних моніторів. При віддалені від камери, зменьшується кількість пікселів, що відображає цей об’єкт, що робить неможливим відображення всіх його деталей (рис. 1.1). Це дає можливісь зменьшити детальністть об’єкту, без суттєвого впливу на якість зображення.

Другий фактор – це особливості сприйняття зображень людиною. Окрім загальної гостроти зору, сприйняття людини також диктується можливістю розріняти об’єкт з його фоном. Можливість розрізення двох об’єктів з мінімальною різницею в яскравості називають чутливістю до контрасту.

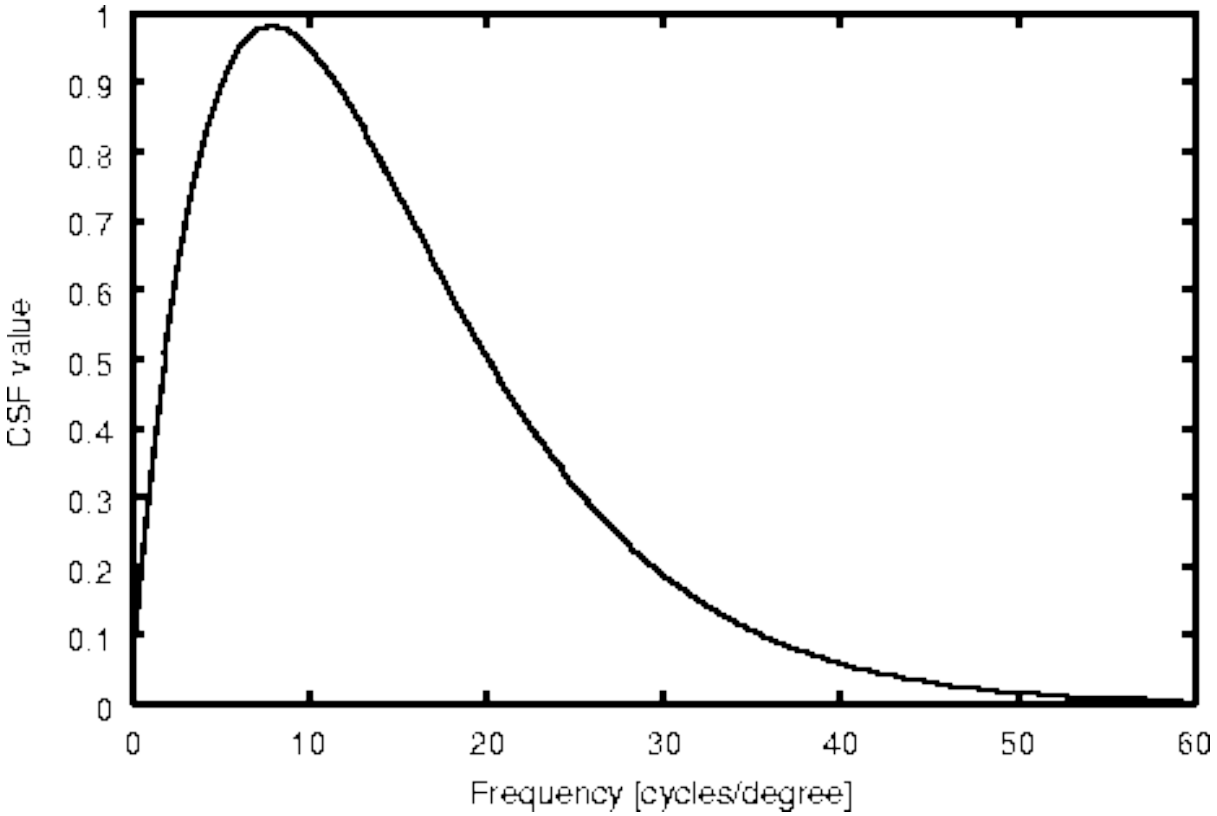


Рис. 1.2. Типовий графік чутливості до контрасту людського сприйняття

Ця метрика характеризуєтьс двома параметрами:

- різниця в яскравості різних частин зображення,
- просторова частота, що характеризує розмір окремих об’єктів.

Пікова контрастна чутливість людського ока спостергігається на об’єктах кутовим розміром 5-8 градусів. Зменьшення розміру об’єкту приводить до спаду можливості розрізняти об’єкти незалежно від їх контрасту, як видно на рис. 1.2.

1.2. Відомі рішення

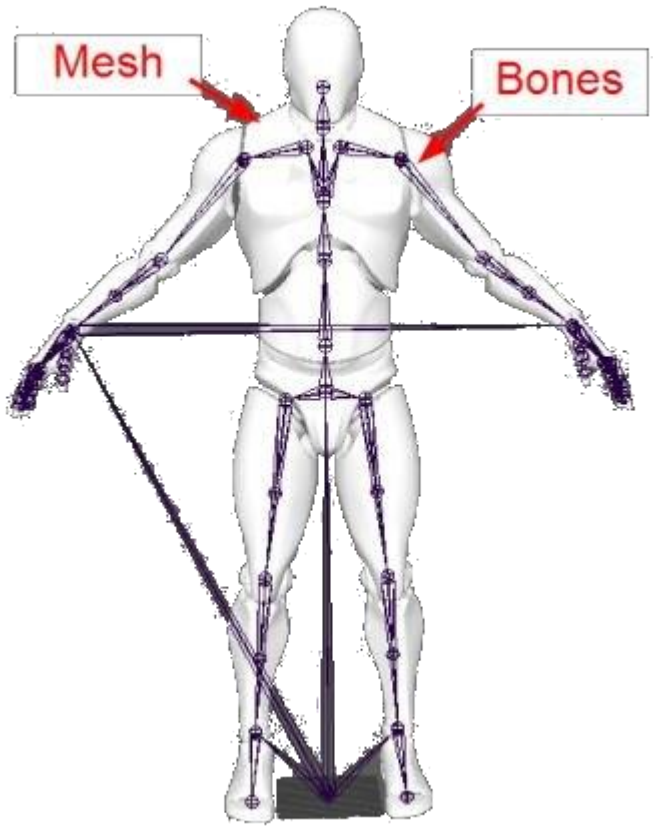


Рис 1.3. Приклад анімованої моделі зі скелетом

В анімованій моделі зі скелетом, коли рухається кістка, вершини асоційовані з нею слідують її трансформації. Такі моделі є найбільш поширеним методом анімації об’єктів в сучасному програмному забезпеченні для анімації (рис 1.3). Програмні продукти, що працюють в

реальному часі, використовують такі моделі для збережння пам’яті та використовувати такі прийому як інтерполяція кадрів, оптимізація моделей та інверсна

Як було сказано вище, тільки деякі методи можна використовувати для спрощення анімованих моделей. Результатом спроби спрощення моделі невідповідним алгоритмом може привести до значних дефектів анімації. У таких випадках, замість автоматизації процесу спрощення, модель спрощується як статична неанімована, після чого проводиться процес скіннінгу LOD-моделі, як показано на рис. 1.4

Основною складністю, під час спрощення анімованих скеледних моделей, є задача перерозподілу ваг вершин.

Серед існуючих методів створення анімованих LOD-моделей слід виділити наступні – метод спрощення за допомогою декомпозиції на під-моделі Фюрмана і Шмальштіка [3], метод коллапсу ребра в одну з вершин Хьюбнера [4], та спрощення коллапсом Хоула і Поліна [5].

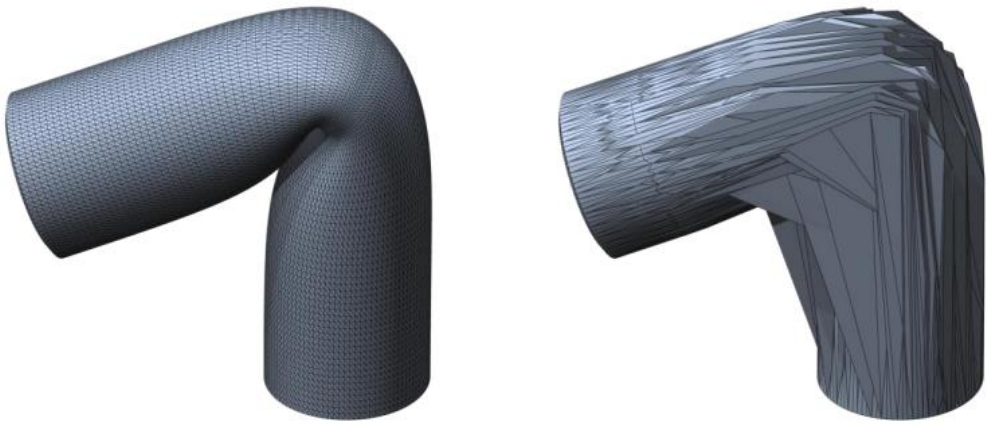


Рис. 1.4. Дефекти анімації спрощеної анімованої моделі

1.2.1. Метод прогресивних моделей

Для розуміння існуючих методів створення LOD-моделей необхідно розуміти метод прогресивних моделей, оскільки всі оглянуті в цьому розділі методи побудовані на його основі.

Метод прогресивних моделей – метод створення та використання LOD-моделей. Прогресивна модель – структура даних, яка створена з оригінальної моделі, яка являється найбільш деталізованою, для якої визначено операції колапсу ребра (edge collapse, далі ecol) та розділення вершини (vertex split, далі vsplit) [6,7,8].

Операція ecol колапсує ребро моделі, спрощуючи її. Ребро для колапсу обирається за допомогою заздалегідь обраного алгоритму.

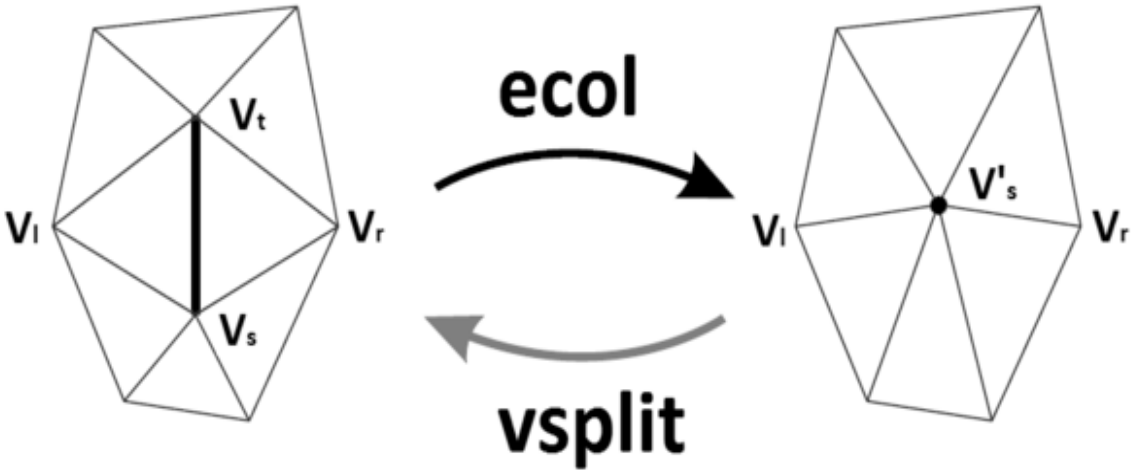


Рис. 1.5. Операції ecol та vsplit на прогресивній моделі

Операція vsplit є оберненою до операції ecol, та повертає модель в попередній стан. Даний процес зображено на рис. 1.5.

Операція ecol повторюється доти, доки не буде отримана потрібна кількість полігонів на моделі. Оскільки при колапсі кожного ребра завжди знищується по 2 полігони, порахувати кількість операцій для цього є тривіальною задачою.

1.2.2. Метод Фюрмана і Шмальштіка

Даний метод оснований на декомпозиції моделі на під-моделі.розподіл відбувається за допомогою ваг вершин [3]. Ці під-моделі далі розділяються на жорсткі (rigid) та пластичні (deformable). Вершина що залежить тільки від однієї кістки є жорсткою, оскільки вона піддається тільки трансформаціям повороту і руху. Вершина що залежить від декількох кісток є пластичною, оскільки вона може піддаватись деформації (рис 1.6). Після розподілу спрощуються тільки жорсткі моделі, що дозволяє оминути перерозподіл ваг вершин. Кожна під-модель також будує свою окрему прогресивну модель, що дозволяє маніпулювати їми окремо. Це дозволяє зберегти загальну якість анімації, та використовувати деякі оптимізації рендеру, як display lists. В даного метода є деякі обмеження, оскільки він не працює з моделями, які не мають жорстких регіонів, а полігони на межі під-моделей неможливо знищувати, оскільки це приведе до невідповідності під-моделей.

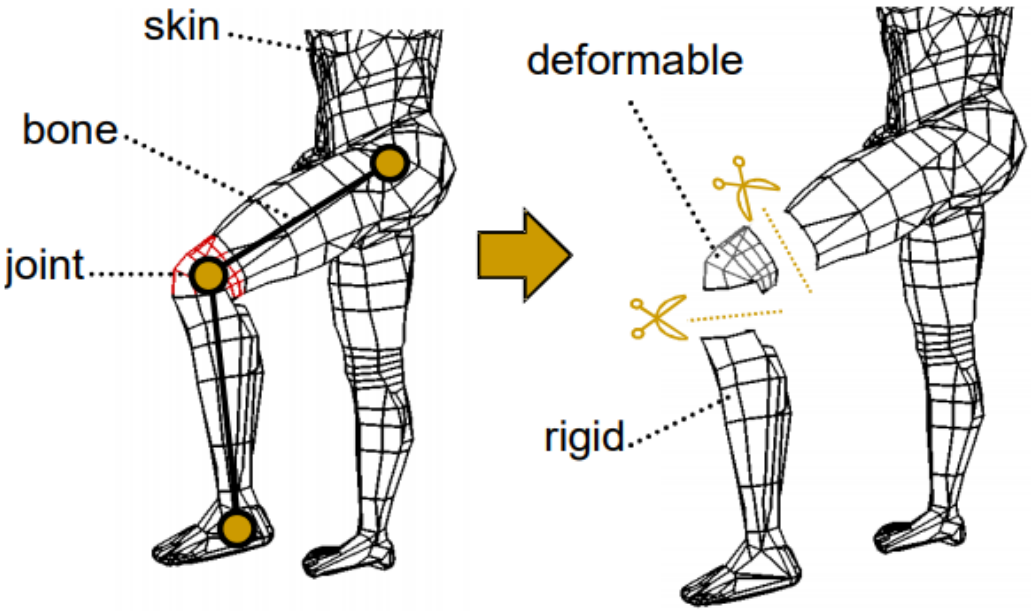


Рис. 1.6. Розподіл моделі на підмоделі

1.2.3. Метод Хьюбнера

Метод Хьюбнера будує прогресивну модель наступним чином – кожна операція ecol колапсує ребро в екстремальну точку, а саме – одну з вершин [4]. Таким чином спрощена модель є просто частиною оригінальної, що дозволяє не перерозподіляти ваги вершин та використовувати всі параметри оригінальної моделі, як-то uv-карту, текстурні координати і т.і. Даний метод також дозволяє проводити операції спрощення в реальному часі, оскільки всі дані необхідні для цього вже є в пам’яті, і всі зміни що необхідно провести стосуються лише управління

1.2.4. Метод Хоула і Поліна

В даному методі прогресивна модель з анімованої будується з оригінальної будь-яким методом, що можна пристосувати до статичної моделі [5]. Але при цьому, для того щоб не перераховувати ваги, створюється спеціальна система для анімації нової моделі. Оскільки при операції ecol нова вершина є точкою між двома попередньо-існуючої, можливо створити систему зворотнього поширення, при якій можна перетворювати оригінальну модель, але результат відображати на спрощеній.

Нехай P_1 і P_2 – вершини оригінальної моделі, P – результат операції ecol на ребрі P_1P_2. Тоді перетворення для вершини P можна обраховувати як інтерполяцію перетворення між P_1 і P_2. Таким чином для будь якої вершини, її перетворення при анімації можна роздивлятися як перетворення батьківських вершин (рис. 1.7). Даний прийом дозволяє обійти проблеми перерозподілу ваг вершин, та потребує спеціальну систему анімації та постійне збереження в пам’яті та маніпуляціїї оригінальної моделі.

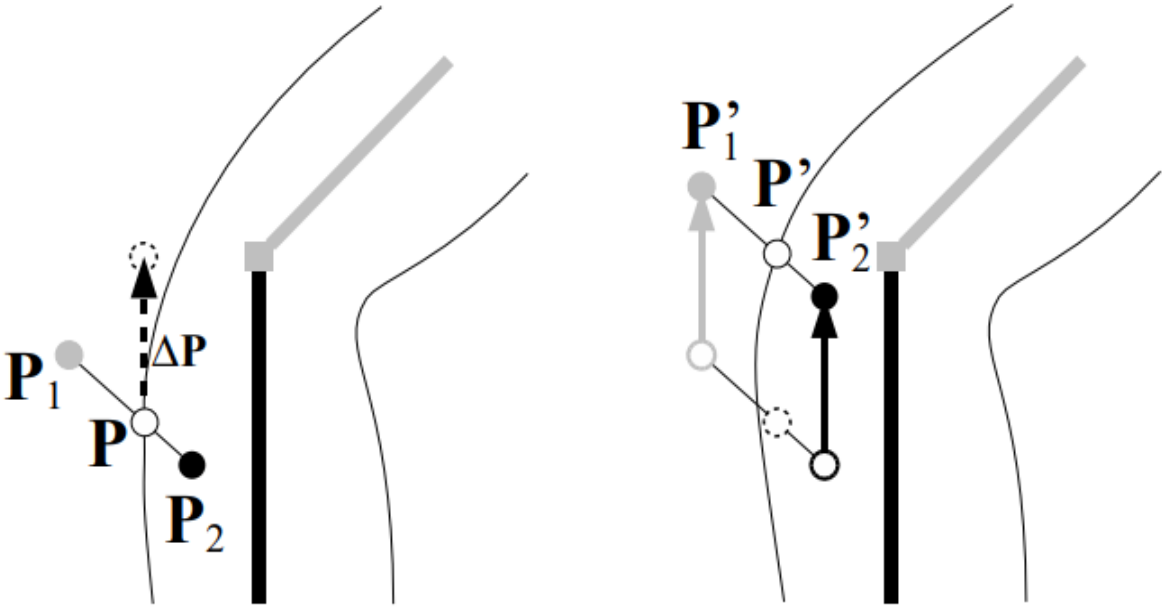


Рис. 1.7. Інтерполяція нової вершини

1.3. Порівняння існуючих рішень

Були обрані наступні критерії для порівняння існуючих методів спрощення анімованих скелетних моделей.

Чи зберігає метод оригінальну топологію. Втрата оригінальної топології робить неможливою оптимізацію позицій вершин під час спрощення що приводить до погіршення якості кінцевої модел і призводить до втрати оптимізації вже під час рендеру, оскільки такі прийоми як стрічка полігонів не можуть бути застосовані. Погана топологія також впливає на якість анімації і наявність артефактів рендеру. На рис. 1.8 показано приклад поганої та хорошої топології. На моделі з хорошою топологією легше проводити зміни художнику-аніматору.

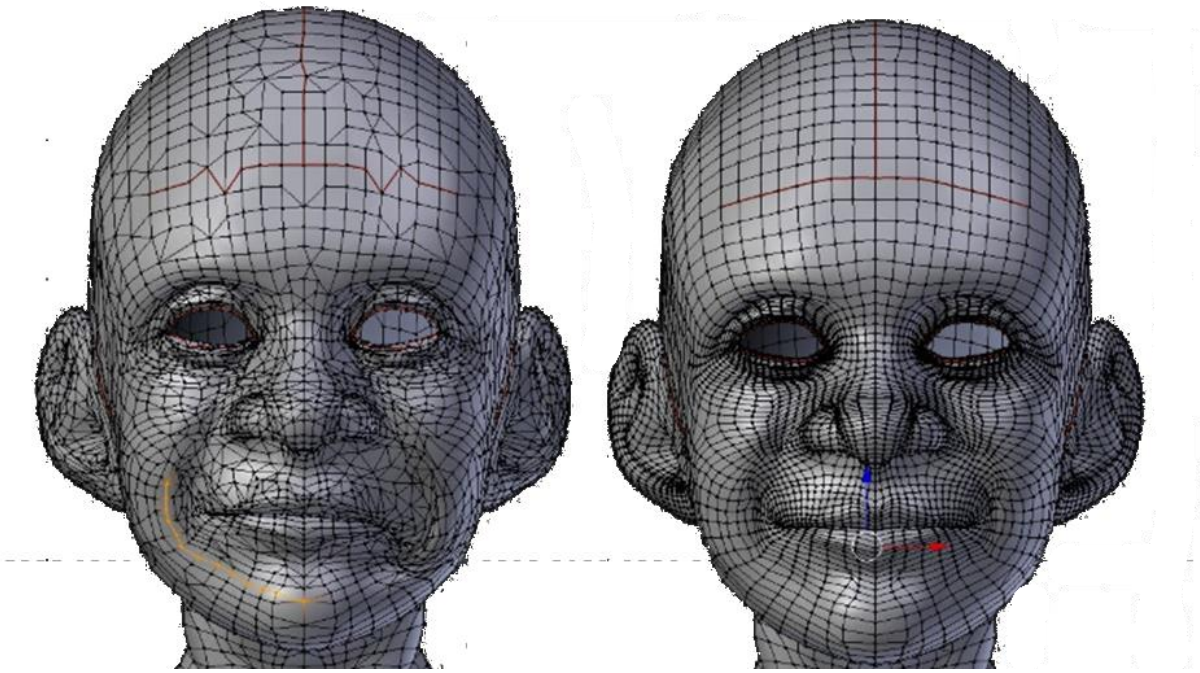


Рис. 1.8. Приклад поганої (зліва) та хорошої (зправа) топології

Чи перераховує метод ваги оригінальної моделі. Перерахунок ваг дозволяє уникнути використання прийомів які можуть підвищити час рендеру, та робить результуючу модель сумісною з більшою частиною програм для анімації.

Чи зберігає метод інші атрибути моделі. Збереження атрибутів допомагає зберегти атрибути оригінальної як UV-карти, колір вершини, тощо, що допомагає зменьшити кількість роботи художника-аніматора.

Чи зповільнює метод роботу рендеру в реальному часі. Оскільки деякі з описаних методів використовують додаткові обрахунки під час рендеру, вони можуть суттєво підвищити необхідний час для рендеру кадру, що призведе до втрати іллюзії руху.

На основі цих критеріїв була побудована табл. 1.1.

Таблиця 1.1

Порівняння існуючих методів спрощення скелетних моделей

Метод	Збереження топології	Перерахунок ваг	Збереження додаткових атрибутів	Швидкість роботи
Schmalstieg and Fuhrmann	–	Не потребує	–	+
Huebner	–	–	+	+
Houle and Pouline	–	–	+	–

1.4. Проблеми існуючих рішень

Як показано в табл 1.1, всі існуючі на даний момент методи для спрощення скелетних моделей мають суттєві недоліки.

Метод Шмальштіка і Фюрмана не зберігає додаткових атрибутів моделі, що призводить до потреби додаткової роботи художника-аніматора. Також цей метод має суттєві обмеження, оскільки його можна застосувати не до всіх анімованих моделей, а суттєві покращення в часі рендеру кадру будуть спостерігатися лише тоді, коли більша частина моделі є жорсткою.

Метод Хьюбнера використовує тільки частину оригінальних вершин, що, як і у випадку з попереднім методом, призводить до погіршення топології нової моделі. Це унеможливлює використання будь-яких оптимізацій позицій вершин під час процесу спрощення, що приводить до меньшої візуальної подібності між оригінальною і результуючою моделями. Окрім цього це робить неможливим техніки оптимізації під час рендеру як стрічки полігонів, які користуються перевагою полігонів об’єднаних в одну групу “стрічки” для зменьшення часу рендеру та економії пам’яті. Також це знову зменьшує потенційну якість нової моделі,

та робить важчою подальшу можливу роботу аніматора над результуючою моделлю. Обидва методи також не піддаються оптимізації face encoding.

Метод Хоула і Поліна потребує від тривимірного рушія існування спеціальної системи для обробки LOD-моделей через інтерполяцію вершин оригінальної моделі. Це не тільки збільшує час рендеру кадру, що частково переважає покращення в швидкості рендеру, а також потребує постійного тримання оригінальної моделі в оперативній пам’яті для постійних маніпуляцій над нею, що ще більше погіршує ситуацію. Окрім того метод також не є універсальним, що робить неможливим використання його результуючих прогресивних моделей з рушіями з закритим вихідним кодом.

1.5. Висновки до розділу 1

У даному розділі було проведено аналіз існуючих методів спрощення скелетних моделей.

Метод Шмальштіга і Фурмана заснований на розкладанні моделей на підмоделях на основі ваг, а потім спрощує їх.

Метод Х’юбнера спирається на згортання ребра моделі до однієї з вершин, щоб не змінювати атрибути, що належать моделі.

Метод Хоула і Поулін спирається на спрощення моделі, як якщо б вона була статичною, а потім обчислює позицію результуючих вершин через інтерполяцію його батьківських вершин.

Всі ці попередні методи мають певні недоліки. Перші два методи не можуть бути оптимізовані під час візуалізації і є поганими наближеннями оригінальної моделі. Метод Шмальштіка і Фюрмана можна застосувати не до всіх моделей, і не завжди його застосування доречне через малу кількість жорстких регіонів моделей. Метод Хьюбнера знижує потенційну якість нової моделі та ускладнює роботу аніматорів з результуючою моделлю. Метод Поуліна і Хоула потребують додаткових розрахунків під час візуалізації, що частково усуває мету створення моделі LOD.

Щоб позбутися всіх цих проблем, є необхідність створення нового методу спрощення скелетних моделей. Новий метод повинен перерозподіляти ваги вершин результуючої моделі щоб уникнути додаткових розрахунків під час рендеру, та для того щоб його результати роботи були сумісними з більшістю існуючих тривимірних рушіїв. Окрім того важливо щоб метод не змінював значно топологію оригінальної моделі для можливості використання оптимізацій під час рендеру та не ускладнював роботу художнику-аніматору.

2. РОЗРОБЛЕННЯ АЛГОРИТМІЧНО ПРОГРАМНОГО МЕТОДУ СПРОЩЕННЯ АНІМОВАНОЇ ТРИВИМІРНОЇ МОДЕЛІ

2.1. Постановка задачі

Задача полягає у розробці алгоритмічно-програмного мотоду, що дозволить створювати LOD-моделі з оригінальної високодетальної моделі, зберігаючи якість анімації, топологію моделі та додаткові атрибути.

Розв’язання задачі потребує наступного:

- огляду існуючих методик використа ння LOD;
- проведення аналізу методик створення LOD моделей;
- формування власного алгоритмічно-програмного методу ствоерення LOD моделей.

2.2. Класифікація LOD

Слід виділити 2 основні методи застосування LOD моделей [9].

2.2.1. Discrete Levels of detail

Discrete Levels of detail (DLOD) – включає в себе створення деякої кількості окремих моделей різного рівню деталізації. В даному методі, моделі заміняють одна одну за заздалегідь заданою евристикою, яка частіше всього пов’язана з візуальним розміром моделі на екрані (рис. 2.1) [10].

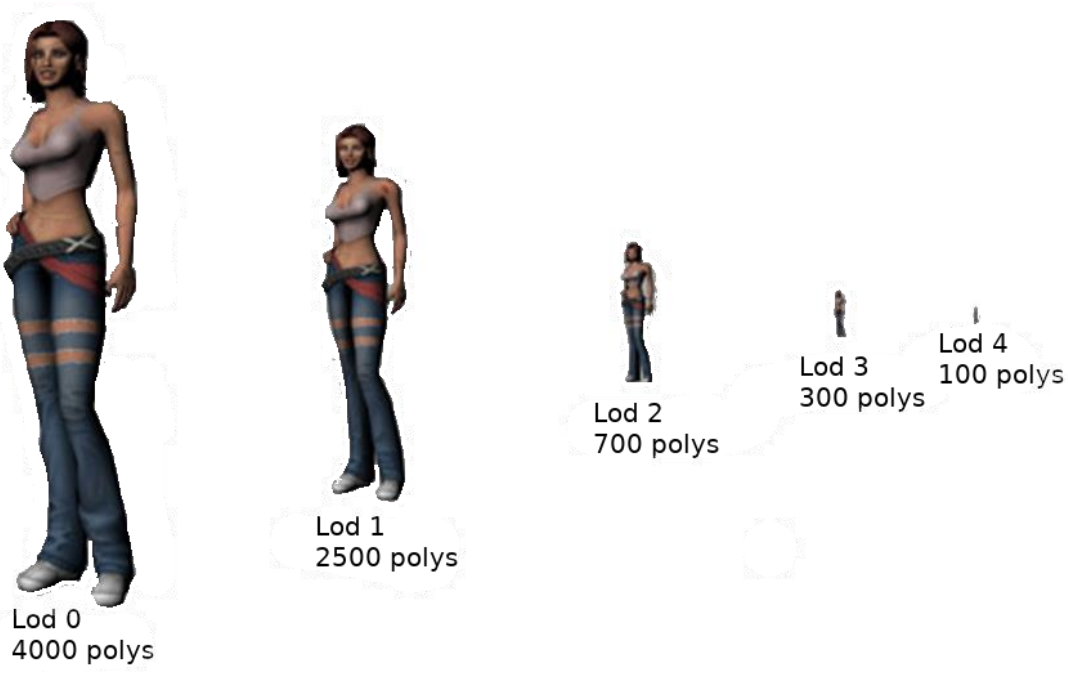


Рис. 2.1. Приклад моделей для використання в DLOD

В конвенціях сучасних тривимірних рушіїв, таких як Unity та Unreal Engine 4, оригінальну модель частіше всього обізначають як LOD 0, а спрощені моделі – LOD 1, LOD 2... LOD *N*, в порядку зменьшення їх деталізації (рис. 2.2). В деяких випадках місце найменьш детальних моделей можуть займати спрайти, для подальшої оптимізації [11].

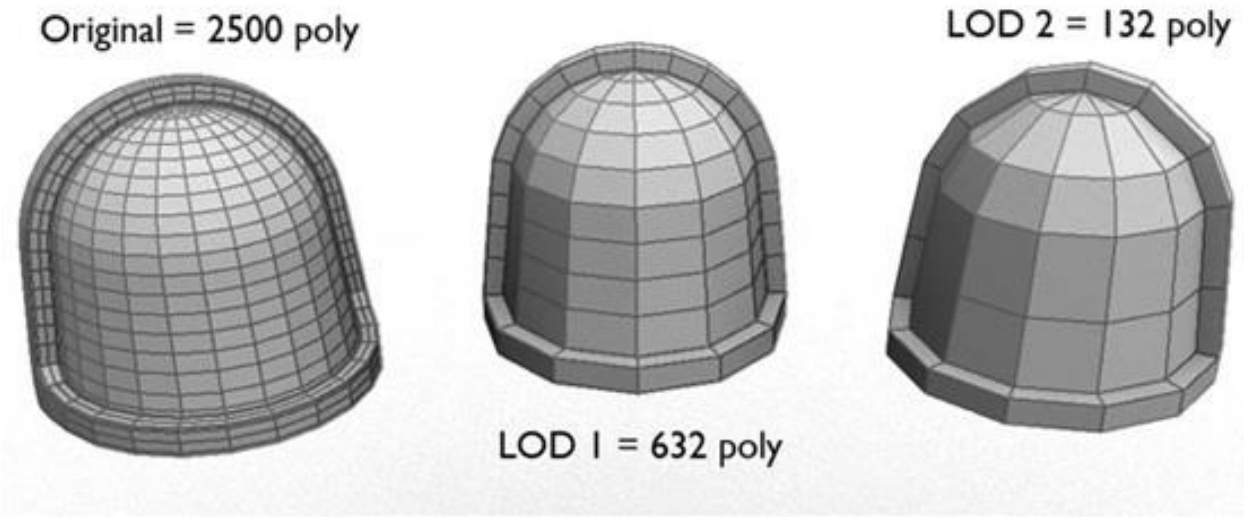


Рис. 2.2. Схема найменування LOD моделей

Hierarchical Levels Of Detail (HLOD) – один з видів застосунку DLOD, що дозволяє ще більше оптимізувати процес вибору моделей для заміни. В даному методі статичні об’єкти групуються в кластери, які використовуються для вибору LOD-моделей для всіх об’єктів у кластері.

Основною перевагою даного методу є відносна простота та швидкість роботи. Це дозволяє використовувати даний метод в додатках, які орієнтовані на високу продуктивність

До недоліків слід віднести складність роботи з дуже великими об’єктами, як ділянки місцевості. Іншим недоліком є можливі візуальні дефекти, як “popping” – різка зміна візуальної форми або текстури об’єкту при заміні однієї між LOD моделями. Це стається у випадках коли одна модель LOD заміняється іншою на відстані від камери, достатній для видимості факту підміни. Проте також існують методи для боротьби з цим ефектом як alpha blending та geomorphing [5,6], що дозволяють значно його послабити.

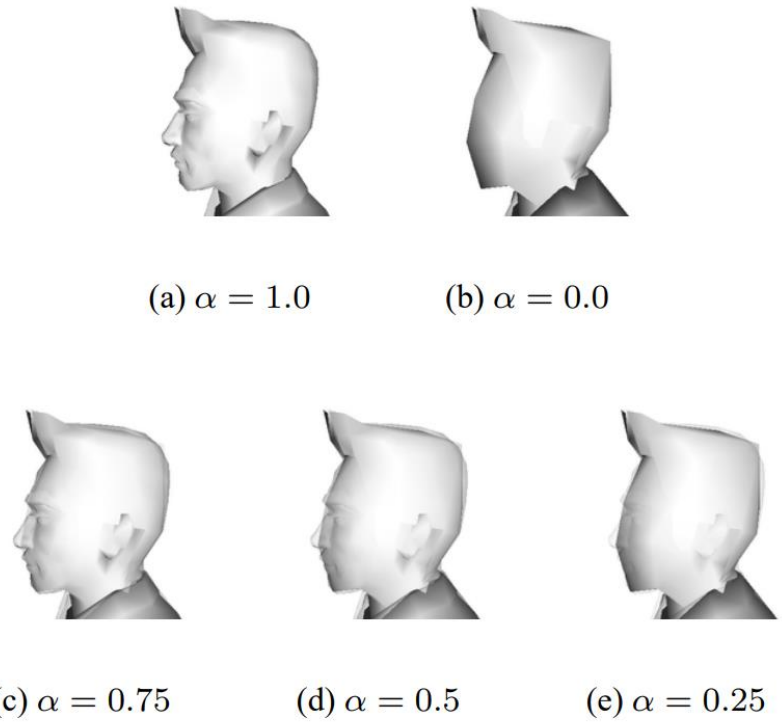


Рис 2.3. Приклад використання alpha blending для уникання артефакту “popping”.

Суть alpha blending, або LOD blending полягає у тому, що при заміні однієї моделі LOD на іншу, обидві моделі показуються одночасно та змішуються, шляхом зміни їх значення альфа каналу, за деякий короткий період переходу. При цьому модель яка заміняється швидко змінює значення альфа каналу від 1.0 (повністю непрозорий) до 0.0 (повністю прозорий), а модель на яку замінюють – навпаки (рис 2.3). Серед переваг alpha blending'у слід виділити простоту реалізації. Серед недоліків – вищу ціну розрахункві, оскільки обидві моделі мають бути присутні на екрані та високі ціну процесів що включають альфа канал, низьку ефективність на невеликих дистанціях від камер, та можливість ствоерення ефекту гостингу.

Геоморфінг полягає у доданні додаткових кроків, що апроксимують перехід між двома різними LOD моделями. Частіше всього це робиться за допомогою ствоерння прогресивної моделі, та операції колапсу ребер та розділення вершини. До переваг геоморфінгу слід віднести більш «природню» заміну LOD-моделей. До недоліків – неможливість зупинити процес геоморфінгу, після того як він почався.

Слід зауважити, що для обох прийомів боротьби з проблемою артефакта "popping", що лінія переходу має бути достатньо тонкою, щоб процес міг пройти швидко.

2.2.2. Continuous Levels of Detail

Continuous Levels of Detail (CLOD) – використовується для отримання рівня деталізації моделі, що може динамічно змінюватис свою складність, залежно від потреби.В основі цього методу лежить структура даних, яка включає в себе оригінальну модель, та дані, які дозволяють отримати потрібний рівень деталізації в режимі реального часу [12].

Основною перевагою даного методу є гладкий перехід між різними рівнями деталізації, можливість підлаштовувати рівень деталізації в

реальному часу, відсутність артефакту “popping”, метод краще справляється з великими об’єктами, ніж DLOD.

Основним недоліком даного методу є велика кількість операцій під час роботи в реальному часі.

2.2.3. View-dependent LOD

View-dependent LOD – використовується для створенні моделей, що можуть динамічно змінювати свою деталізаці, причому різні частини однієї моделі можуть мати різні рівні деталізації. Це може використовуватись для досягання наступних ефектів:

- відображення близьких частин об’єкту більш детальними, та далеких части об’єкту меньш детальними (рис. 2.4) [10];
- відображення регіонів, які відповідають за силует об’єкту з більшою детальністю, ніж інші регіони (рис. 2.5).
- відображення більшої кількості деталей коли користувач дивиться прямо на об’єкт, та меньшої, коли об’єкт в зонах периферійного зору користувача.

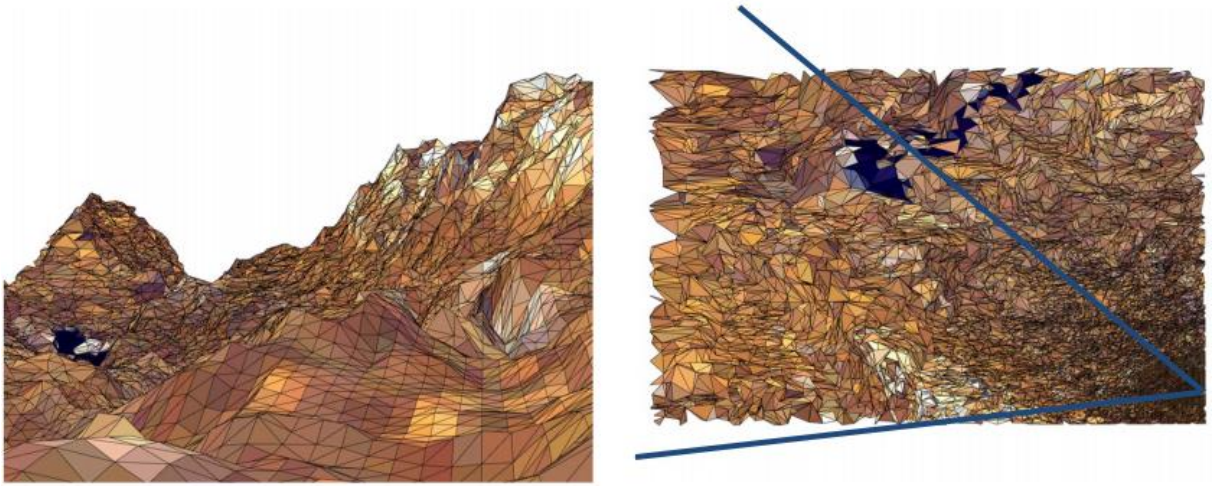


Рис. 2.4. Використання View-dependent LOD для відображення більшої кількості деталей біля камери

Це дає змогу відображати великі об’єкти (як ділянки місцевості), без суттєвої втрати якості на будь-яких дистанціях від камери, або зберігати силуети об’єктів одночасно отримуючи вищу продуктивність роботи програми, ніж з іншою моделлю з ідентичним силуетом.

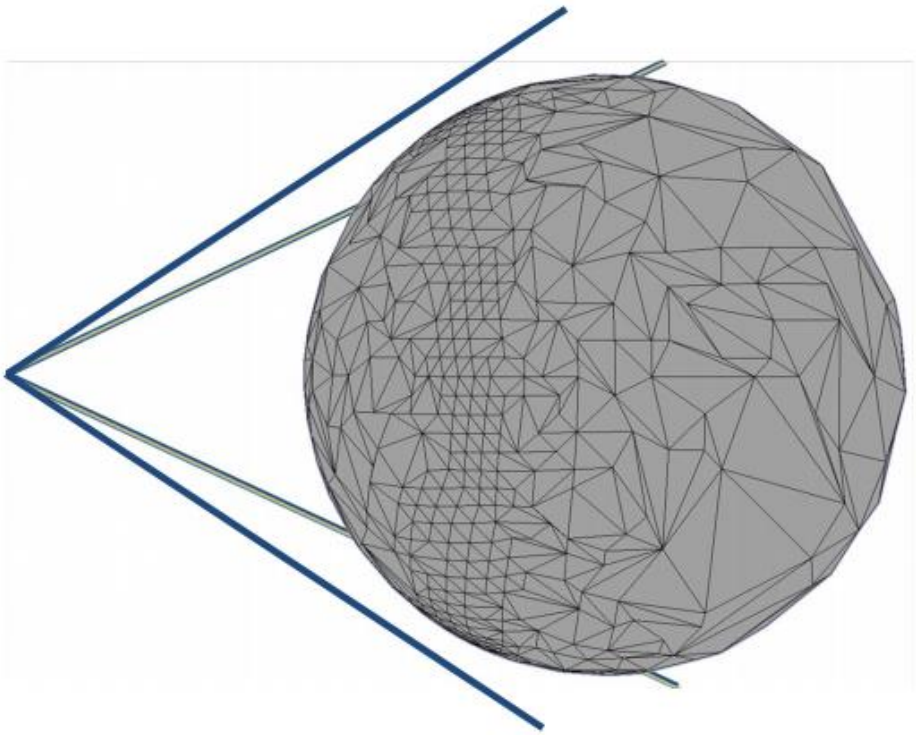


Рис. 2.5. Використання View-dependent LOD для збереження силуету об’єкту

До переваг даного методу можна віднести ще більш гладкий перехід між різними рівнями деталізації моделей, відсутність артефакту “popping”, метод найкраще справляється з великими об’єктами, та має додаткові застосуванняю.

До недоліків слід віднести ще більшу кількість операцій під час роботи додатку в реальному часі, та складну процедуру створення LOD моделей для використання.

2.3. Методи створення LOD моделей.

Методи створення LOD моделей можна поділити на локальні, або методи оператору спрощення, та глобальні, або видозалежні(view-dependant).

2.3.1. Локальні методи створення LOD

Локальними є методи, які вимірюють помилку одночасно тільки над однією частиною моделі – будь то полігон, вершина чи ребро. Серед них слід виділити видалення вершини та колапс ребра [14,15].

При видаленні вершин обирається одна вершина, знищення якої призведе модель з найменьшою помилкою, яка видаляється з моделі та виконується триангуляція(заповнення трикутниками) отриманого отвору (рис 2.6). Основними недоліками даного методу є значна зміна топології результуючої моделі, та зменьшення її об’єму та силуету.

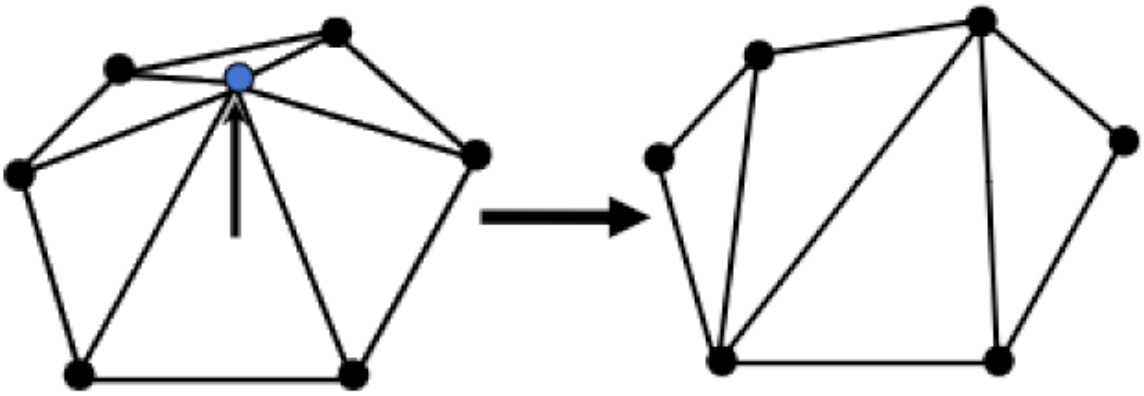


Рис. 2.6. Приклад операції знищення вершини

Згортання ребер оперує на парі вершин, з’єднаних ребром. Дані вершини згортаються в точку на ребрі, яка стає новою вершиною. Цей метод можна поділити на декілька під-методів, що класифікуютсья за способом обранням точки для новоутвореної вершини, кожен з яких має

свої переваги та недоліки. Загалом всі методи в цій категорії мають перевагу у виді можливості роботи на відкритих (non-manifold) моделях.

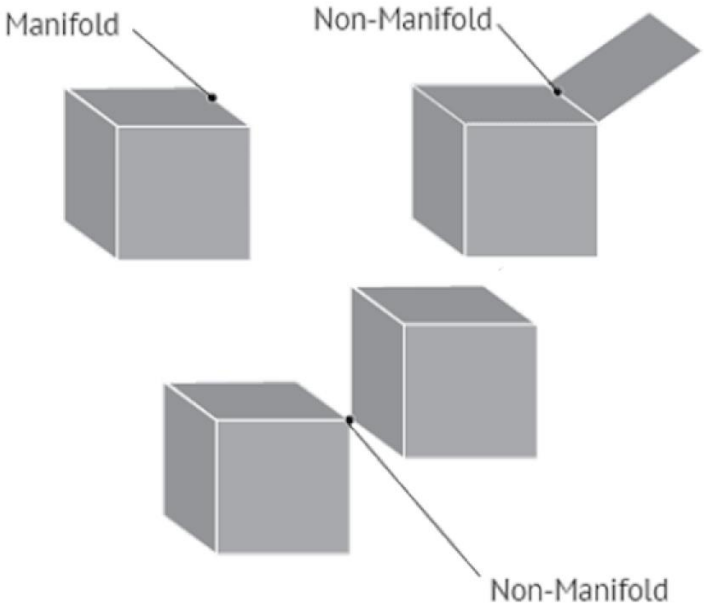


Рис. 2.7. Приклади зактритої(manifold) та відкритих (non-manifold) моделей

Перший з них це половинний колапс ребра (halfedge collapse), при якому точка новоутвероної вершини знаходиться в одній з кінцевих точок колапсованого ребра (рис 2.7). Основними перевагами цього методу є швидка робота через меньшу кількість розрахунків, та відсутність необхідності в перерозрахунках додаткових атрибутів моделі. До недоліків відосяться гірша, по відношенню до інших методів, якість спрощення та втрата об’єму та силуету.

Повний колапс ребра, виконується коли для новоутвореної вершини обирається точка, що мінімізує обрану метрику помилки на результуючій моделі. Цей метод загалом має кращу точність результуючої моделі, та краще зберігає її об’єм та силует, проте повільніше.

Згортання вершин, або згортання віртуального ребра – метод, при якому колапсувати можуть не тільки вершини, що лежать на одному ребрі,

а ще і вершини, які достатньо близько одна до одної. Він має ще вищу точність результуючої моделі, та краще збергіає силует, проте має серйозний недолів, в можливості об’єднання окремих частин моделі.

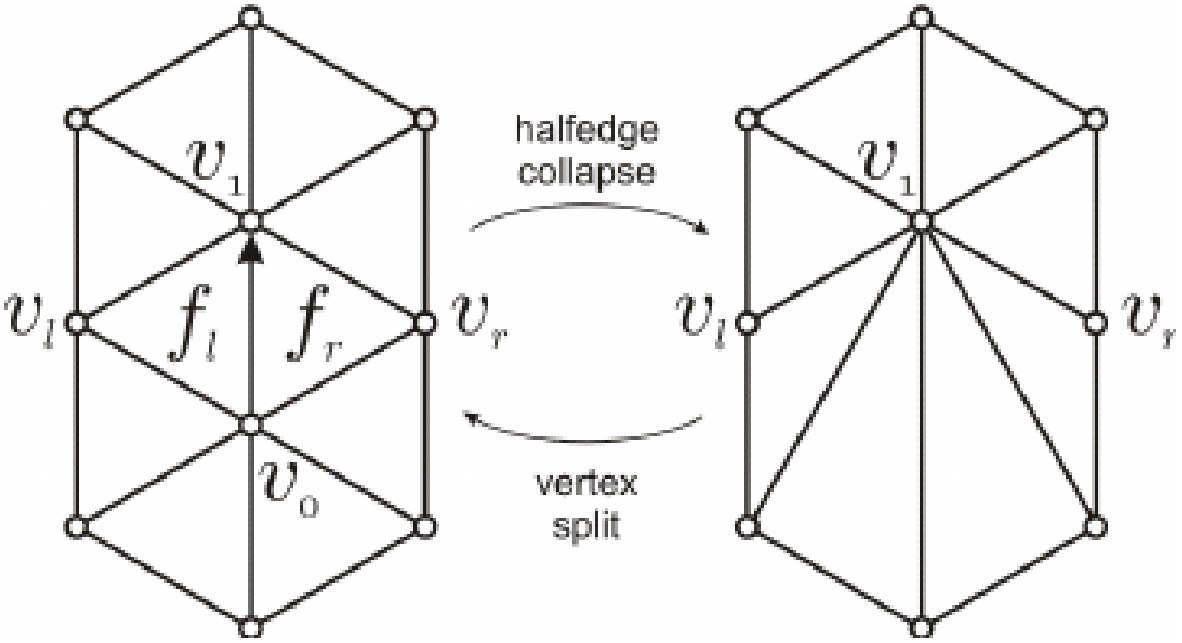


Рис. 2.8. Оперція половинного колапсу ребра, результуюча вершина опиняється в одній із батьківських

2.3.2. Глобальні методи створення LOD.

Дані методи контролюють глобальну помилку, на відміну від локальної. Загалом глобальні методи спочатку будують ієрархічну структуру кластерів вертексів, яку називаються ієрархією вертексів. Під час спрощення, в залежності від деякого шуканого парамеру, розмір кластерів може змінюватись[14,15]. Кластери що стають занадто малі, за зазделегідь вибраною метрикою, колапсують, спрощуючи модель. Ієрархія вертексів може мати 2 основні вигляди – дерево вертексів, та список активних трикутників(полігонів). Вертексне дерево описує повну модель, включаючи всі можливі операції спрощення. Список активних трикутників описує модель в теперішньому її вигляді. Глобальні добре методи підходяться для створення CLOD моделей. Деякі з існуючих глобальних

методів це метод прогресивних моделей, ієрархічне динамічне спрощення, мультитриангуляція та інші.

2.4. Методи вимірювання помилки на LOD моделях

Методи вимірювання помилок на згенерованих LOD моделях можна розділити на 2 групи – геометричні та візуальні.

2.4.1. Візуальні методи вимірювання помилки на LOD моделях

Ідея візуальних методів полягає у вимірюванні помилки, яку буде сприймати користувач при використанні додатку. Візуальні методи беруть до уваги такі фактори як силует нової моделі, тіні на цій моделі при рендерингу, кольори та нормалі вертексів, артефакти текстурування.

Для вимірювання візуальної помилки використовується вже згадана в розділі 1.1 функція чутливості до контрасту. Ця функція була визначена експерементально, та її апроксимація, запропонована Mannos і Sakrison знайшла своє застосування для стиснення зображень [16], та оцінки помилки на LOD моделях [13]:

$$A(f) = 2.6(0{,}0192 + 0{,}114f)e^{(0.114f)^{1.1}},$$

де f – кутовий розмір об’єкту.

Даний метод має перевагу в тому, що він близько апроксимує видиму помилку реального користувача, та має засади в реальному світі.

До недоліків можна віднести складність вимірювання цієї помилки, оскільки для цього потрібно створити набір сцен, що відповідає різноманітним умовам освітлення та ракурсу камери по відношенню до об’єкту вимірювання (рис 2.9), що залежить від середовища, в якому буде використовуватись модель.

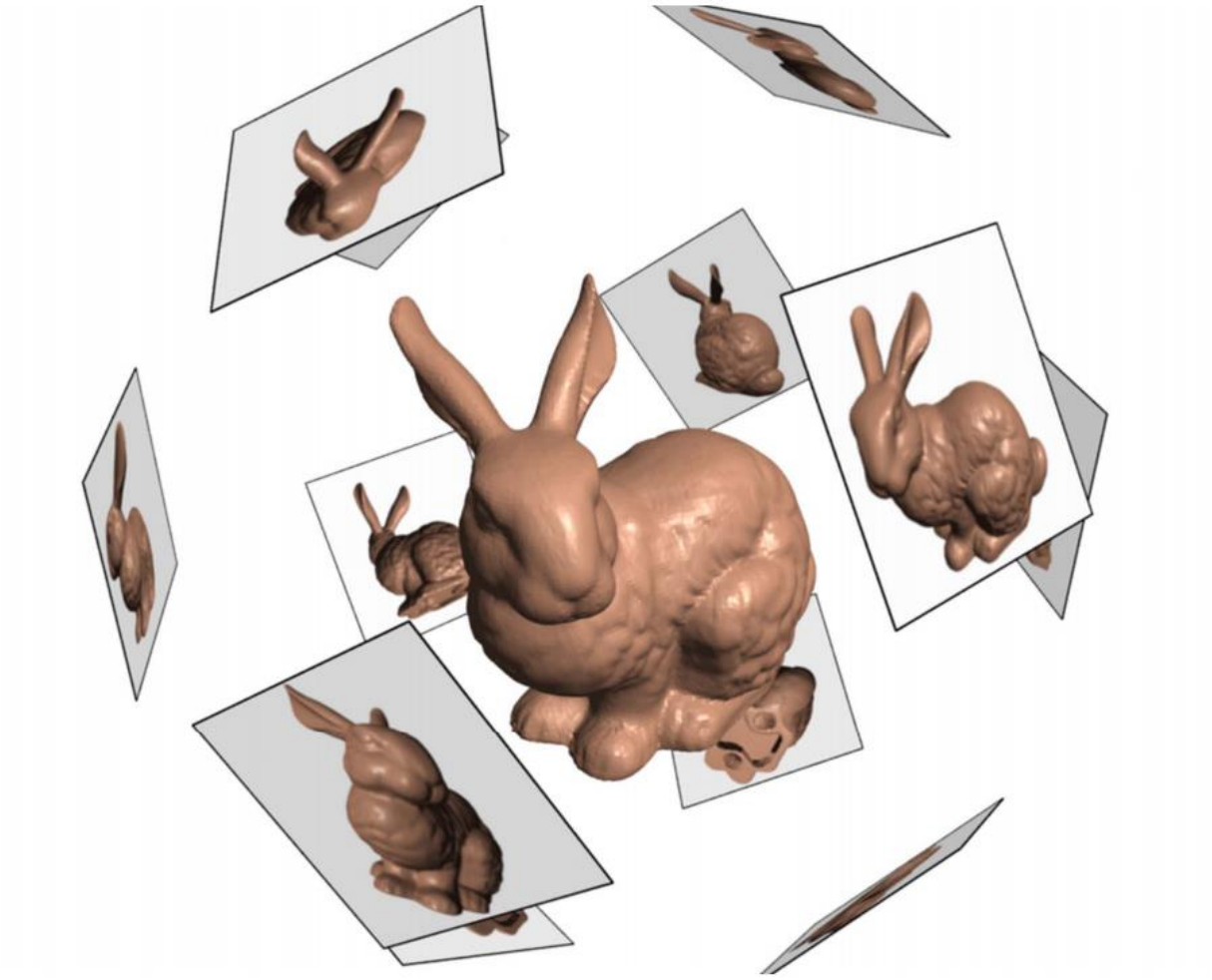


Рис. 2.9. Приклад створення сцени для вимірювання візуальної помилки на моделі.

2.4.2. Геометричні методи вимірювання помилки на LOD моделях

Дане сімейство методів апроксимують реальну візуальну помилку на LOD моделі, по відношенню до оригінальної, в основному за допомогою дистанцій між поверхнею оригінальної та поверхнею спрощеної моделі, або за допомогою порівняння ємності простору, що займають ці моделі.

Основним методом є визначення метрики Гаусдорфа між оригінальною та спрощеною моделями [9, 17]. Ця метрика є визначеною для замкнених скінченних підмонжинах метричного простору, але поверхня моделей також може бути описана такою множиною. Метрика Гаусдорфа, дає замірити максимальну з мінімальних дистанцій між двома

поверхнями, або, іншими словами ми визначаємо мінімальну дистанцію для кожної точки з одної підмножини до точок іншої підмножини, а потім беремо максимальну з цих дистанцій. Нехай існує 2 поверхні, *A* та *B*, то дистанція Гаусдорфа визначається наступним чином:

$$H(A,B) = \max\big(h(A,B), h(B,A) \big),$$

двостороння дистанція Гаусдорфа, де

$$h(A,B) = \max_{a \in A} \min_{b \in B} \big|\big| a-b \big|\big| ,$$

одностороння дистанція Гаусдорфа. Одностороння дистанція Гаусдорфа не є симетрично. Функцією, тобто $h(A,B) \neq h(B,A)$, тому вона присутня в двосторонній метриці двічі. Ця метрика є дуже точною для вирахування помилки, проте має низку недоліків, що роблять її складною у використанні. Проблемою є факт, що набори точок, визначені на двох тривимірних моделях, можуть не відповідати один одному, що погіршить точність вимірювань. Також, ці набори можуть містити цілі регіони точок, що не мають відповідних їм, або регіони де цих відповідних точок є декілька.

Інші геометричні метрики являють собою апроксимації дистанції Гаусдорфа. Серед поширених методів можна визначити метрики Гаусдорфа між наборами вертексів моделей, між вертексами та їх проекціями на площини, між вертексами та найближчими полігонами, METRO (метрика Гаусдорфа для набору випадкових точок на поверхнях моделей) та квадратична метрика. Остання є достатньо швидкою, щоб мати місце у вимірюванні помилки на кожному кроці створення LOD моделі, проте є меньш точною.

2.5. Опис запропонованого методу для створення LOD моделей

Пропонований метод базується на роботах Haule та Pauline по спрощенні анімованих моделей. Головними відмінностями є:

- ваги кісток не копіюються між моделями, а перераховуються;

- кілька поз використовуються для отримання вибірки для спрощення.

Для старту, модель ставиться в декілька поз, декілька її копій отримують екстремальні пози кісток моделі, та декілька поз між екстремальними позиціями. Біьша кількість проміжних позицій потенційно покращить якість анімації результуючої моделі. Для кожної пози, за квадратичною помилкою, вибираються найкращі ребра для коллапсу. При цьому коофіцієнт $t = 0$, для того щоб тільки вершини на одному ребрі могли коллапсувати. Вибірка з декількох поз, покращує результат анімації спрощеної моделі, усереднюючи помилку для всіх поз. Найкраще ребро колапсується на всіх моделях.

2.5.1. Вибір валідних пар для колапсу вершин

Важливим є вибір пари для колапсу вершин. Нехай пара вершин (v_1, v_2) є валідною вершиною для колапсу, якщо

1. (v_1, v_2) лежать на одному ребрі моделі, або
2. $\|v_1 - v_2\| < t$, де те t – деяке порогове значення

Даний алгоритм зображений у виді діаграми діяльності на рис. 2.10.

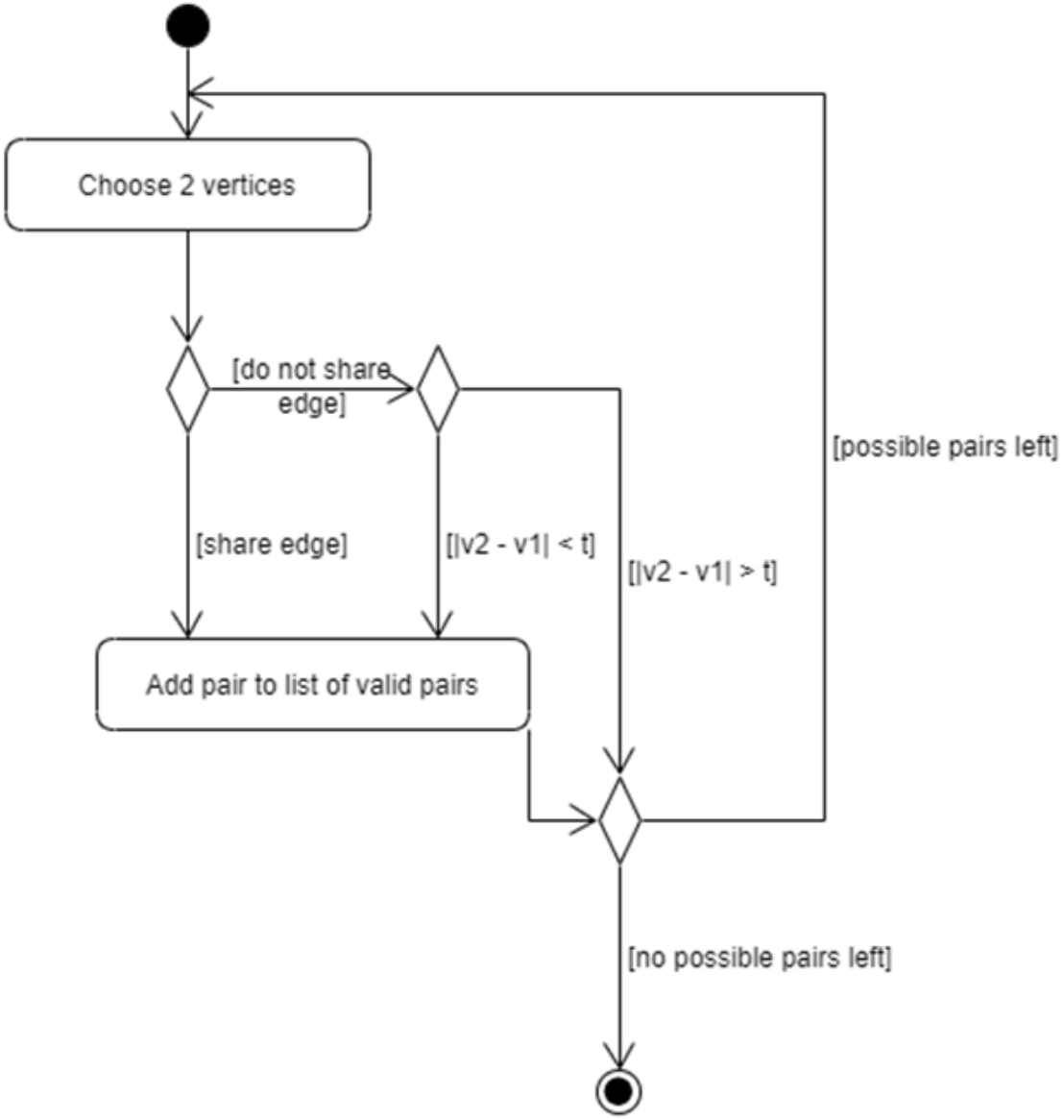


Рис. 2.10. Діаграма діяльності пошуку валідних верши для колапсу

Вищі значення *t* дають вершинам, що не є вершинами одного ребра розглядатись як кандидат для колапсу. Це дає загалом кращу форму результуючої моделі, проте може давати неочікувані артефакти, як «заростання» деяких відкритих областей на моделі. Значення в $t = 0$ дає звичпйний алгоритм колапсу ребер, та загалом дає кращу топологію результуючої моделі.

Звичайно значення t потрібно обирати обережно. Експерементальні досліди показують що оптимальними значеннями являються значення близько t = 0.1 [19].

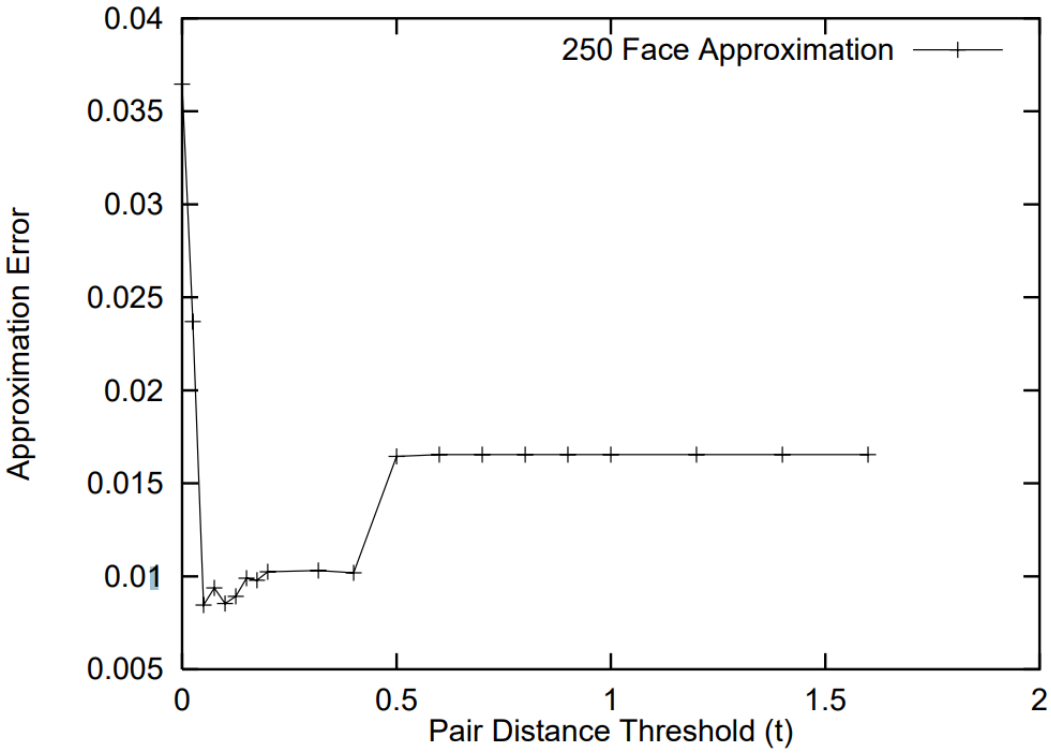


Рис. 2.11. Функція помилки на результуючій моделі відносно оригінальної залежно від значення t [19]

Також, як видно з рис. 2.11, значення t = 0 негативно впливають на загальну помилку результуючої моделі.

2.5.2. Апроксимація помилки моделі

Для вибору пари вершин з мінімальною помилкою, потрібно оцінити помилку результуючої моделі, утвореної після колапса цих вершин.Нехай для вертексу $v = \begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}^T$ помилка

$$\Delta(v) = v^T Q\, v.$$

Отримати матрицю Q можна наступним чином. Оскільки вершина на моделі – точка перетину кількох площин, можна асоціювати набір площин

з кожною вершиною та отримати помилку вершини як суму квадратичних дистанцій до її площин:

$$\Delta v = \Delta\left(\left[v_x\ v_y\ v_z\ 1\right]\right)^T = \sum_{p \in planes(v)} (p^T v)^2\ ,$$

де $p = [\ a\ b\ c\ d\ ]^T$, описує площину, що визначається рівнянням

$$ax+by+cz+d=0,$$

де $a^2 + b^2 + c^2 = 1$.

Ця помилка може бути переписана в наступному вигляді:

$$\begin{aligned}\Delta(v) &= \sum_{p \in planes(v)} (v^T p)(p^T v) \\ &= \sum_{p \in planes(v)} v^T (pp^T) v \\ &= v^T \Big(\sum_{p \in planes(v)} K_p \Big)\, v,\end{aligned}$$

де K_p – матриця наступного вигляду:

$$K_p = pp^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}.$$

Ця матриця може бути використана для визначення середньої квадратичної дистанції до площин p від будь-якої точки в просторі. Якщо просумувати ці матриці K_p для однієї вершини то отримаємо матрицю Q – яка буде представленням помилок всіх площин.

Для кожної можливої пари для колапсу $(v_1, v_2) \to v'$ матриця помилки $Q' = Q_1 + Q_2$. Оптимальною парою для для колапсу є пара вершин з мінімальним $\Delta(v)$.

Для обраненя нової позиції нової вершини v' знаходиться позиція на прямій (v_1, v_2), що мінімалізує помилку $\Delta(v')$. Для цього необхідно розв'язати рівняння $\frac{\partial \Delta}{\partial x} = \frac{\partial \Delta}{\partial y} = \frac{\partial \Delta}{\partial z} = 0$, що є еквівалентним до розв'язання наступного рівняння:

$$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}=\begin{bmatrix}q_{11} & q_{12} & q_{13} & q_{14}\\ q_{21} & q_{22} & q_{23} & q_{24}\\ q_{31} & q_{32} & q_{33} & q_{34}\\ 0 & 0 & 0 & 1\end{bmatrix} *v',$$

по відношенню до v'. Якщо дана матриця має обернену, то отримаємо

$$v'=\begin{bmatrix}q_{11} & q_{12} & q_{13} & q_{14}\\ q_{21} & q_{22} & q_{23} & q_{24}\\ q_{31} & q_{32} & q_{33} & q_{34}\\ 0 & 0 & 0 & 1\end{bmatrix}^{-1} *\begin{bmatrix}0\\0\\0\\1\end{bmatrix},$$

де v' – координати нової вершини. Матриця буде мати обернену, коли вона описує невироджений еліпсоїд. В такому разі v' буде знаходитися в центрі цього еліпсоїду.

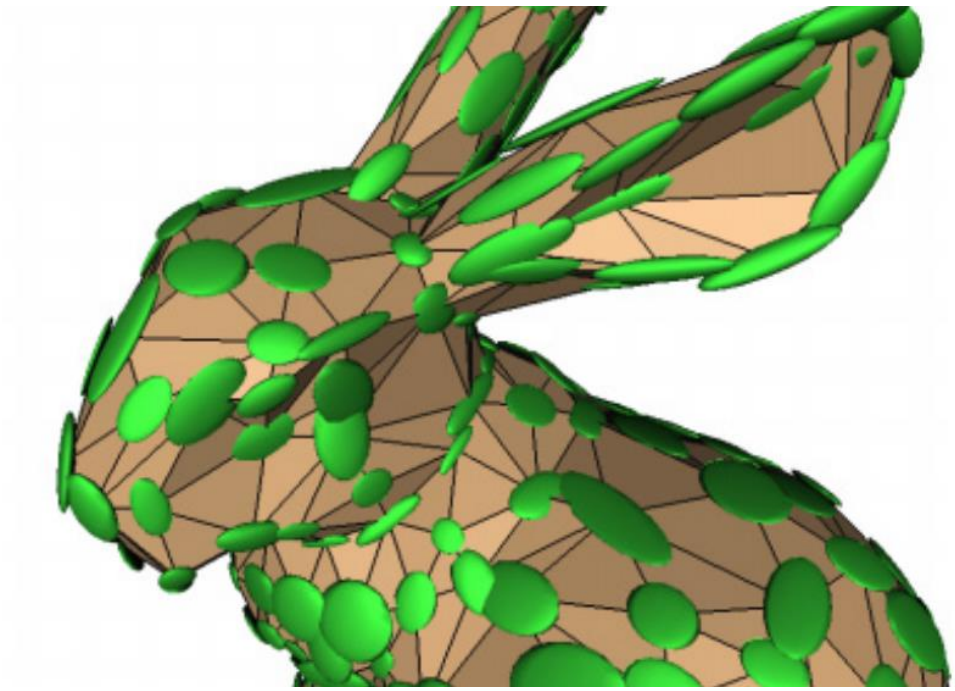


Рис. 2.12 візуалізація матриць *Q* на моделі стенфордського кролику[2]

Якщо ж ця матриця не має оберненої, то позицію для v' можна вибрати між кінцевими точками v_1, v_2 та точкою $(v_1+v_2)/2$, в залежності від того, яка з них буде мати меньшу результуючу помилку.

2.5.3. Розрахування ваг вершин результуючої моделі

Для знаходження ваги вершини на спрощеній моделі використується зважена сума. Припустимо v_1 та v_2 – вершини на одному ребрі, обрані для колапсу, v' – вершина, що є результатом колапсу вищеописаного ребра, w_{1i} – ваги вершини v_1 кістки i, w_{2i} – ваги вершини v_2 кістки i,w'_i – ваги результуючої вершини v' кістки i. Тоді результуючі ваги w'_i вершини v' можна описати наступним чином:

$$w'_i = w_{1i} * \frac{\text{distance}(v_1, v')}{\text{distance}(v_1, v_2)} + w_{2i} * \text{distance}(v_2, v') / \text{distance}(v_1, v_2)$$

для кожної кістки *i*.

Важливим є проведення цієї операції для кожної кістки, оскільки сумма ваг кожної вершини повинна бути рівна 1. Якщо існують ваги кісток, достатньо малі, щоб їми можна було знехтувати, важливо нормалізувати результуючий вектор вершин, після видалення цьої ваги.

Цей процес повторюється, поки не отримається модель, з потрібною кількістю полігонів. Кількість ітерації можна дуже просто розрахувати, адже кожну ітерацію знищується рівно 2 полігони.

Результуючий метод можна підсумувати так:

1. Анімована модель ставиться в основну позу, екстремальні пози для кісток, та деяку кількість поз між ними.
2. Обирається та коллапсується ребро з найменьшою помилкою для всіх поз.
3. Ваги кісток результуючої вершини розраховуються як зважена сума ваг батьківських вершин, де вага – відношення дистанцій від батьківської вершини до результуючої до дистанції між батьківськими вершинами.
4. Процесс повторюється до отримання моделі з потрібною складністю.

5. В результаті отрмується анімована модель, з меньшою кількістю полігонів ніж оригінальна, яка зберігає основні деталі моделі, та її форму.

Алгоритм запропонованого методу подано на рис. 2.13.

2.6. Висновки до розділу 2

В даному розділі сформульовано задачу по створенню алгоритмічно-програмного методу, проведено аналіз математичних моделей спрощення тривимірних моделей, та розглянуто основні підходи.

На основі теоретичних відомостей, наведених в поточному та попередніх розділах, запропоновано алгоритмічно програмний метод генерації LOD для моделей зі скелетом. Оригінальна модель спрощується за допомогою квадратичної метрики помилки, з вибіркою в декілька поз, та її нові ваги та інші атрибути перераховуються за допомогою позицій результуючих вершин, по відношенню до колапсованих. Більша вибірка поз дасть кращу точність анімації результуючих спрощених моделей.

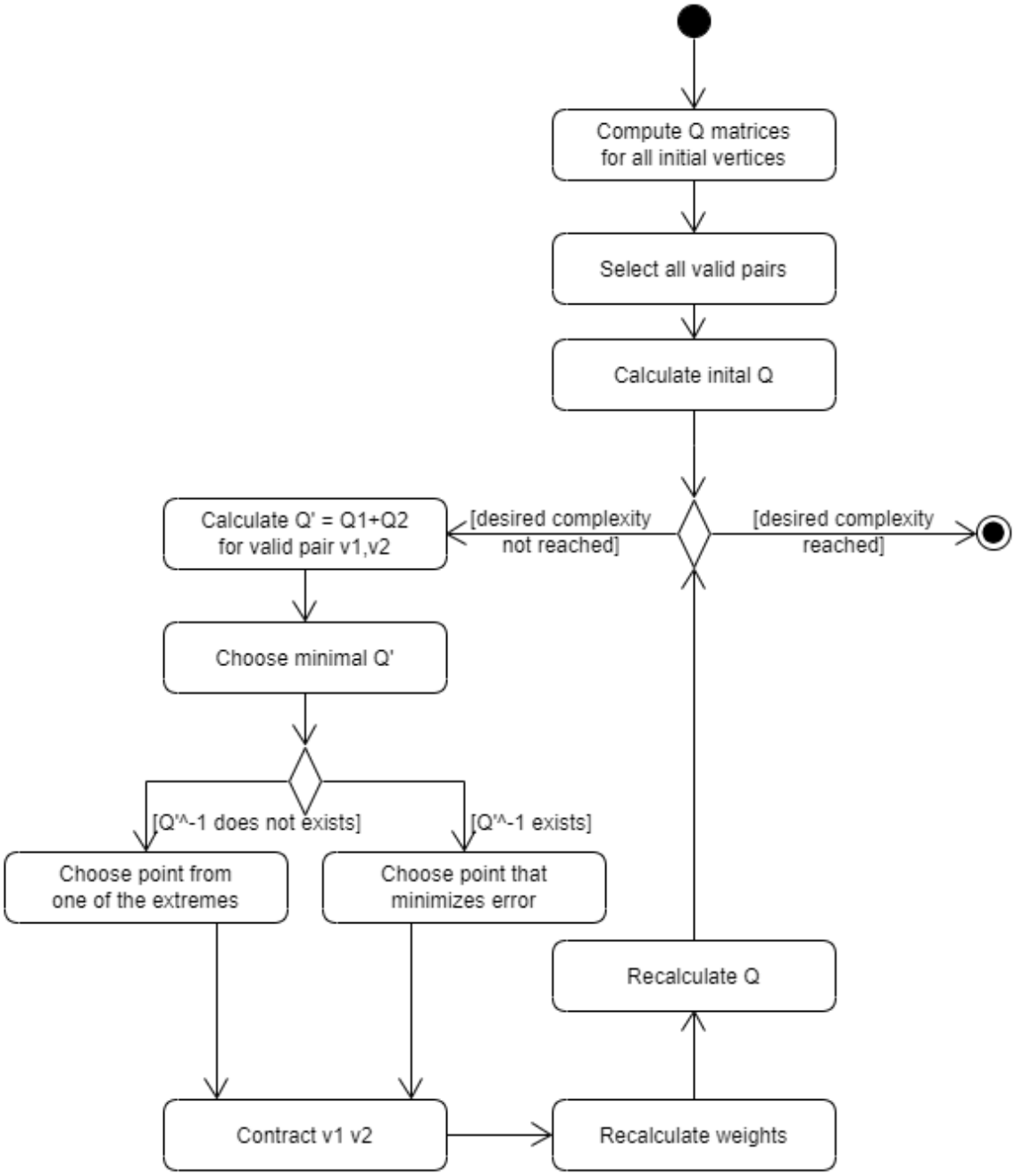


Рис. 2.13. Діаграма діяльності запропонованого алгоритму

3. РЕАЛІЗАЦІЯ ЗАПРОПОНОВАНОГО МЕТОДУ

3.1. Вибір засобів розроблення

Для розроблення програмної реалізації запропонованого методу необхіднийц редактор тривимірних моделей, що надає підтримку до користувацьких розширень.

Blender – безкоштовне (як для навчання, так і комерційно), відкрите програмне забезпечення для створення комп'ютерної графіки для моделювання, скульптингу, анімації, симуляції, рендерингу та композиції. Має відкриту екосистему розробників розширень. Розроблення розширень ведеться за допомогою мови Python з власним Blender Python API. Має обширну базу документації. Підтримує платформи Windows, Linux та Mac OS X.

3DS MAX – комерційне програмне забезпечення для 3D-моделювання і анімації, з ціною ліцензії в 1620 долларів на рік. 3DS MAX має API для розроблення додатків на мовах C#, Python та C++. Має закриту екосистему розробників розширень. Підтримує платформи Microsoft Windows.

ZBrush – комерційне програмне забезпечення для 3D-моделювання і анімації, з безстроковою ліцензією в розмірі 895 долларів на рік, або 39 долларів на місяць. Дозволяє розробляти плагіни за допомого влаштованої скриптової мови ZScript. Підтримує платформи Microsoft Windows та Mac OS X.

Cinema 4D – комерційне, крос-платформне програмне забезепечення для створення тривимірної графіки та анімації, з можливістю створення та редагування тривимірних об'єктів. Має безстрокову ліцензію ціною в 3495 долларів, або підписку ціною від 117 долларів на місяць. Має відкриту екосистему розробників розширень. Розроблення може вестись на мовах C++, за допомогою бібліотеки Cineware SDK, та Python. Підтримує

платформи Windows, Linux та Mac OS, потребує постійного підключення до інтернету.

Таблиця 3.1

Порівняння тривимірних редакторів

		Blender	3DS MAX	ZBrush	Cinema 4D
Початкова вартість створення продукту на їх основі		0	1680$	$39...895	$116…3495
Відкритість екосистеми розробників розширень		+	–	–	+
Підтримка ОС	Windows	+	+	+	+
	MacOS	+	–	–	+
	Linux	+	–	+	+
Повнота документації для розробників		+	+/–	+/–	+
Простота інтеграції в підсистему 3D-моделювання		+	+	+	+
Імпорт поширених форматів тривимірних моделей		+	+	+	+

Окрім створення додатку для редактору тривимірних моделей, можливою альтернативою є створення додатку для, або самостійної програми на основі одного з існуючих тривимірних рушіїв. Найпопулярнішими відкритими безкоштовними тривимірними рушіями на даний момент можна вважати Unreal Engine 4, Godot Engine та Unity Engine.

Unreal Engine 4 – безкоштовний, закритий, крос-платформенний, тривимірний рушій розроблений компанією Epic Games. Він підтримує розроблення додатків на C++. Серед шейдерних мов підтримується HLSL,

з кросс-компіляцією в GLSL, та шейдери типів global(compute) shaders, material/mesh shaders та vertex shader. Unreal Engine 4 підтримує імпорт тривимірних моделей у форматі FBX. Для підтримки інших форматів можливо встановити сторонні додатки. Unreal Engine 4 підтримує розроблення на платформах Windows, Linux, Mac OS X.

Godot Engine – безкоштовний, крос-платформенний, відкритий, тривимірний рушій для застосунків, ліцензований MIT. Він підтримує розроблення на C++, C#, Rust, Nim, D та власну високорівневу скриптову мову GDScript. Серед шейдерних мов він підтримує GLSL ES 3.0 і типи шейдерів для 2D, 3D та рендерингу чатинок, проте не підтримує технології compute shader. Godot Engine підтримує імпорт тривимірних моделей в форматах DAE (Collada), gITF 2.0 та OBJ. Підтримує розроблення на платформах Windows, Linux, Mac OS X та має серверну версію.

Unity Engine – безкоштовний, закритий, крос-платформенний, тривимірний рушій розроблений компанією Unity Technologies. Він підтримує розроблення на C#, Boo та власну скриптову мову UnityScript. Серед шейдерних мову Unity підтримує HLSL та GLSL, та типи surface shader, vertex/fragment shader та compute shader. Unity engine підтримує імпорт моделей в форматах FBX, DAE (Collada), DXF, OBJ, SKP та glTF 2.0. Окрім цього підтримуються імпорт власних форматів моделей таких тривимірних редакторів моделей як Max, Maya, Blender, Cinema4D, Modo, Lightwave та Cheetah3D. При імпорті файли цих форматів вони конвертуються в поширений формат FBX. Unity підтримує розроблення на платформах Windows, Linux та Mac OS.

Таблиця 3.2

Порівняння тривимірних редакторів

		Unity	Unreal Engine 4	Godot Engine
Початкова вартість створення продукту на їх основі		0	0	0
Відкритість екосистеми розробників розширень		+	+/–	+
Підтримка ОС	Windows	+	+	+
	MacOS	+	+	–
	Linux	+	+	+
Повнота документації для розробників		+	–	+
Простота інтеграції в підсистему 3D-відображення		+	+	+
Імпорт поширених форматів тривимірних моделей		+	–	–

Для реалізації запропонованого програмно-алгоритмічного методу генерації LOD моделей був обраний тривимірний рушій Unity. Unity має широку та детальну документацію своїх можливостей, що значно спрощує процес розроблення додатків на цій платформі, підтримує велику кількість розширень тривимірних моделей. Та має доступ до технології compute shader.

3.2. Платформа Unity

Unity – багатоплатформовий інструмент для розроблення дво- та тривимірних додатків та ігор, що працює на операційних системах

Windows і OS X. Створені за допомогою Unity застосунки працюють під системами Windows, OS X, Android, Apple iOS, Linux.

Технічні характеристики:

- Ігрова логіка пишеться за допомогою C#, доступні шейдерні мови GLSL та HDLS.
- Ігровий рушій повністю пов'язаний із середовищем розроблення. що дозволяє тестувати розроблюємий продукт прямо в редакторі.
- Робота з ресурсами можлива через звичайний Drag&Drop.
- Підтримка імпортування великої кількості форматів файлів.
- Вбудована підтримка мережі.
- Гнучкий інтферфейс редактору.
- Існує рішення для спільної розроблення (Asset server).

3.2.1. Особливості відображення тривимірних моделей на платформі Unity

Оскільки Unity позиціонує себе як ігровий рушій, в цьому рушії були введені деякі особливості, з метою оптимізації рендеру в режимі реального часу. Розглянемо один з найпростіших примітивів в рушії Unity – куб.

Куб можливо мінімально описати маючи 8 вершин, які об’єднуються в 12 ребер та утворюють 6 сторін. Якщо ж заглянути в редактор об’єкта Unity, можна побачити, що куб має 24 вершини, та складається з 12-ти трикутників.

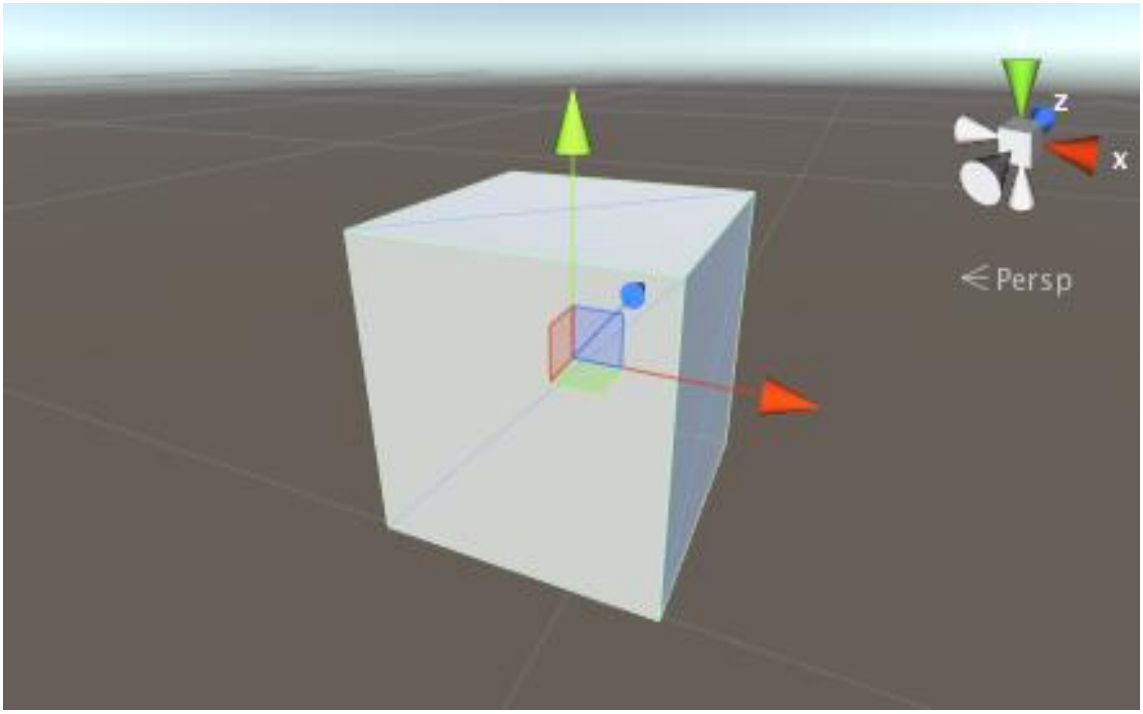


Рис. 3.1. примітив куба в рушії Unity.

Така кількість вершин пояснюється тим, що об’єкт вершини в рушії Unity зберігає в собі не тільки позицію окремої точки, а ще і додактові атрибути моделі, такі, як вектор нормалі. Оскільки кожна сторона куба має свій напрямок нормалі, кожна з вершин, що описує цю сторону має зберігати вектор нормалі для кожної з сторін куба. Якщо змінити позицію тільки однієї з вершин моделі, скоріше за все виникнуть візуальні дефекти.

Також Unity не працює з *n*-гонами, і всі сторони відмальовуж як трикутники, які являють собою набір трьох вершин. Ці вершині записані в масиві `triangles`[], розмір якого завжди повинен бути кратний трьом. Цей масив є масивом індексів, кожний елемент якого є індексом конкретної вершини в масиві `vertices`[], що містить об’єкти класу Vertex. Дуже важливо при роботі з цими масивами, щоб індекси в масиві `triangles`[] посилались тільки на існуючі елементи `vertices`[], оскільки інакше це може призвести до помилок.

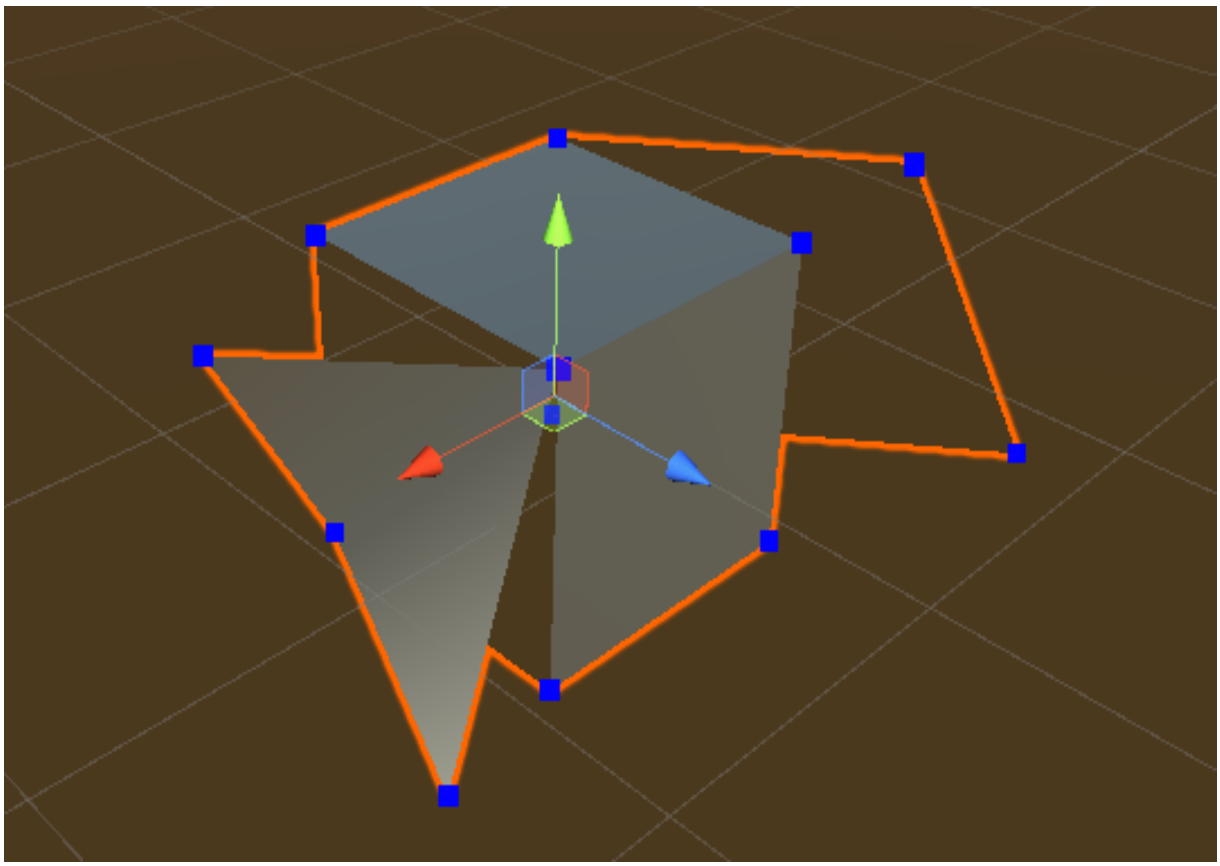


Рис. 3.2. Демонстрація рендеру окремих вершин об’єкту на програмному рушії Unity

Також, для оптимізації анімованих моделей, був створений спеціальний клас об’єктів BoneWeight. такий об’єкт містить в собі 4 статичних індекси – посилання на кістки моделі, та 4 ваги цих кісток. Таке обмеження в кількості ваг було зроблене задля пришвидшення рендерингу анімованих об’єктів, оскільки в переважній кількості випадків моделі не містять вершин що залежать від 5 та більше кісток. Це обмеження також можна обійти ввімкнувши спецальні опції в налаштуваннях анімованої моделі:

- Quality – в режимі Auto;
- Blend weights – режимі Unlimited.

При роботі з вагами, важливо пам’ятати, що сума вагів одної вершини завжди має давати в сумі 1.

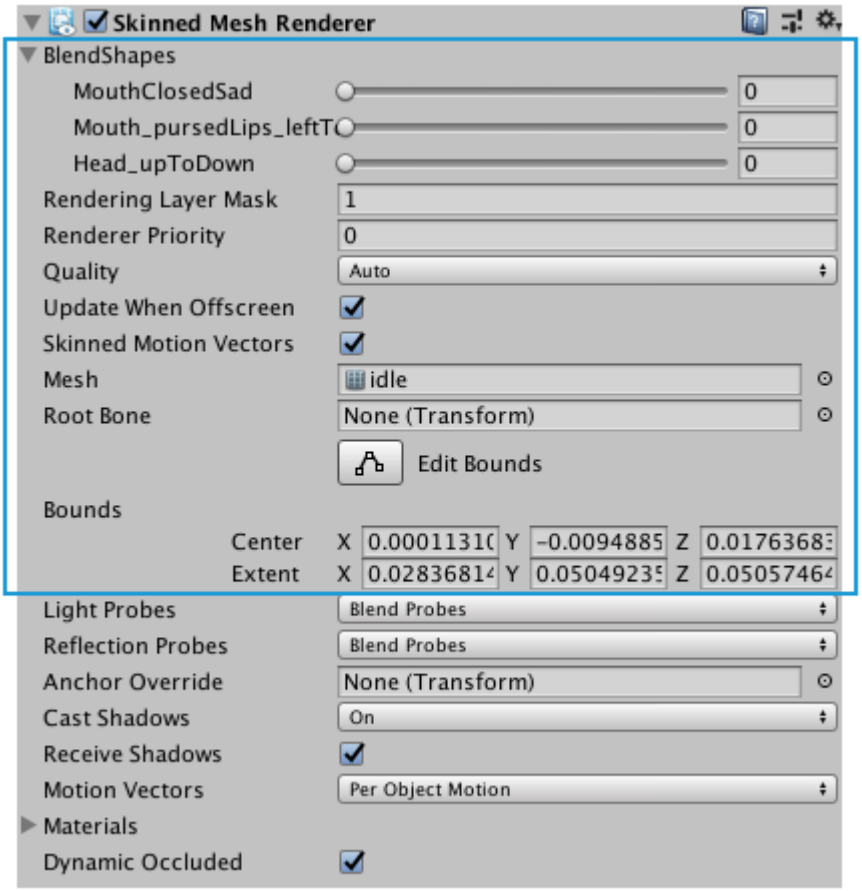


Рис. 3.3. Налаштування моделі зі скелетом в рушії Unity

Також Unity дає можливість працювати з LOD моделями, налаштовувати коли моделі замінюються LOD-моделями, налаштовувати максимальну якість моделі, та інше.

3.2.2. Технологія compute shader

Compute shader – це програми що виконуються на графічній карті, ззовні звичайного процесу рендерингу. Вони можуть бути використані для програмування масивно паралелізованих алгоритмів, або прискорення окремих частин рендерингу. Для ефективного їх використання потрімдні знання алогритмів паралелізації, а також знання DirectCompute, OpenGL Compute, CUDA або OpenCL.

В рушії Unity compute shader близько збігаються з технологією DircetCompute DircetX 11 або 12.

3.3. Особливості реалізації розробленого методу

На рис. 3.4 наведено архітектуру реалізації запропонованого методу. Аддон що слугує для спрощення моделі, додаєтться скриптом на саму модель. Таким чином кожна окрема модель в редакторі може мати свої унікальні налаштування.

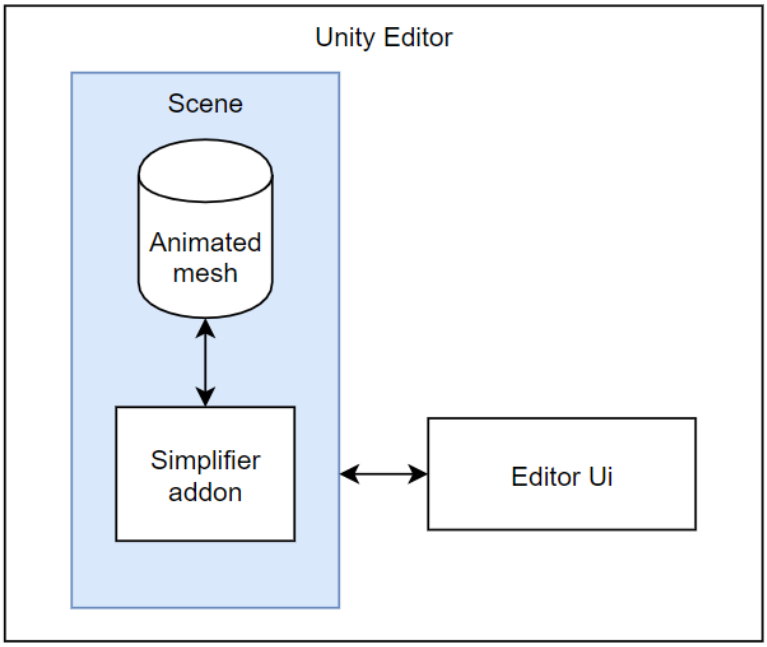


Рис. 3.4. Архітектура реалізації запропонованого методу

3.3.1. QuadricErrorMetrics

Основний клас, що реалізує запропонований алгоритм. Наслідує базовий клас `MonoBehaviour`, від якого мають наслідуватись усі скрипти.

Функція `Init()` ініціалізує початкові параметри роботи додатку та викликає функцію `InitVertexData()`.

Функція `InitVertexData()` обчислює початкові матриці K_p, описані у підрозділі 2.5, пункті 2, для всіх вершин модел (лістинг 3.1).

Лістинг 3.1. Обрахунок матриць *K в реалізації*

```
            for (int i = 0; i < vertices.Count; i++)
            {
                VertexData data = new VertexData();
                data.edges = GetEdges(i);
                vertexDatas.Add(data);
            }

            //int error metrics
            for (int i = 0; i < triangles.Count; i += 3)
            {
                int v0 = triangles[i];
                int v1 = triangles[i + 1];
                int v2 = triangles[i + 2];
                Vector3 p0 = vertices[v0];
                Vector3 p1 = vertices[v1];
                Vector3 p2 = vertices[v2];
                Vector3 normal = Vector3.Cross(p1 - p0, p2 -
p1).normalized;
                float a = normal.x;
                float b = normal.y;
                float c = normal.z;
                float d = -Vector3.Dot(normal, p0);
                Matrix4x4 q = new Matrix4x4();
                q.SetRow(0, new Vector4(a * a, a * b, a * c, a *
d));
                q.SetRow(1, new Vector4(b * a, b * b, b * c, b *
d));
                q.SetRow(2, new Vector4(c * a, c * b, c * c, c *
d));
                q.SetRow(3, new Vector4(d * a, d * b, d * c, d *
d));
                vertexDatas[v0].errorMatrix                    =
MathUtil.AddMatrix(vertexDatas[v0].errorMatrix, q);
                vertexDatas[v1].errorMatrix                    =
MathUtil.AddMatrix(vertexDatas[v1].errorMatrix, q);
                vertexDatas[v2].errorMatrix                    =
MathUtil.AddMatrix(vertexDatas[v2].errorMatrix, q);
            }

            //Init Errors
            for (int i = 0; i < vertexDatas.Count; i++)
            {
                CalcError(i);
            }
```

Функція `GetEdges()` повертає всі валідні ребра моделі.

Функція `FindBestContract()` обирає найкращу пару вершин для колапсу.

Функція `Simplify()` викликає корутину, що колапсує ребра моделі, доки не буде досягнена бажана деталізація моделі, встановлена користувачем.

Функція `mergeBoneWeights()` приймає пару об’єктів класу BoneWeight, коофіцієнт, та повертає валідно розподіленів ваги, як це описано у підрозділі 2.5, пункті 3 (лістинг 3.2).

Лістинг 3.2. Перерахунок ваг вершин у реалізації

```
            if (weight1.Equals(weight2)) return weight1;
            List<Weight> w1 = new List<Weight>();
            w1.Add(new                    Weight(weight1.boneIndex0,
weight1.boneIndex0));
            w1.Add(new                    Weight(weight1.boneIndex1,
weight1.boneIndex1));
            w1.Add(new                    Weight(weight1.boneIndex2,
weight1.boneIndex2));
            w1.Add(new                    Weight(weight1.boneIndex3,
weight1.boneIndex3));
            List<Weight> w2 = new List<Weight>();
            w2.Add(new                    Weight(weight2.boneIndex0,
weight2.boneIndex0));
            w2.Add(new                    Weight(weight2.boneIndex1,
weight2.boneIndex1));
            w2.Add(new                    Weight(weight2.boneIndex2,
weight2.boneIndex2));
            w2.Add(new                    Weight(weight2.boneIndex3,
weight2.boneIndex3));
            List<Weight> result = new List<Weight>();
            int   index   =   w2.FindIndex(w   =>   w.boneIndex   ==
weight1.boneIndex0);
            if       (index       !=       -1)       result.Add(new
Weight(weight1.boneIndex0,

weight1.weight0*coef + w2[index].weight*(1-coef)));
            else      result.Add(new      Weight(weight1.boneIndex0,
weight1.weight0*coef));

            index    =    w2.FindIndex(w    =>    w.boneIndex    ==
weight1.boneIndex1);
            if       (index       !=       -1)       result.Add(new
Weight(weight1.boneIndex1,

weight1.weight1*coef + w2[index].weight*(1-coef)));
            else      result.Add(new      Weight(weight1.boneIndex1,
weight1.weight1*coef));
            index    =    w2.FindIndex(w    =>    w.boneIndex    ==
weight1.boneIndex2);
            if       (index       !=       -1)       result.Add(new
Weight(weight1.boneIndex2,
```

Продовження лістингу 3.2

```
weight1.weight2*coef + w2[index].weight*(1-coef)));
            else      result.Add(new      Weight(weight1.boneIndex2,
weight1.weight2*coef));
            index    =    w2.FindIndex(w    =>    w.boneIndex    ==
weight1.boneIndex3);
            if       (index       !=       -1)       result.Add(new
Weight(weight1.boneIndex3,

weight1.weight3*coef + w2[index].weight*(1-coef)));
            else      result.Add(new      Weight(weight1.boneIndex3,
weight1.weight3*coef));
            result.Sort((a,b) => b.weight.CompareTo(a.weight));
            float summ = result.Sum(x => x.weight);
            BoneWeight newBoneWeight = new BoneWeight();
            _w0 = result[0].weight/summ;
            _w1 = result[1].weight/summ;
            _w2 = result[2].weight/summ;
            _w3 = result[3].weight/summ;
            newBoneWeight.boneIndex0 = result[0].boneIndex;
            newBoneWeight.weight0 = result[0].weight/summ;
            newBoneWeight.boneIndex1 = result[1].boneIndex;
            newBoneWeight.weight1 = result[1].weight/summ;
            newBoneWeight.boneIndex2 = result[2].boneIndex;
            newBoneWeight.weight2 = result[2].weight/summ;
            newBoneWeight.boneIndex3 = result[3].boneIndex;
            newBoneWeight.weight3 = result[3].weight/summ;
            return newBoneWeight;
```

3.3.2. VertexData

Структура даних, що зберігає в собі матриці Q окремої вершини, викоирстовуючи для цього оптимізовану структуру Matrix4x4.

3.3.3. BoneWeight

Цей клас слугує оберткою навколо базового класу boneWeight Unity Engine, та використовується для полегшення роботи з ними.

3.3.4. QuadricErrorEditor

Даний клас наслідується від класу Editor ядра рушія Unity, що слугує для створення та редагування елементів інтерфейсу типу Inspector. Це дозволяє створювати детальні інтерфейси користувача, та влаштвовувати їх прпямо в інтерфейс Unity. Даний клас має елемент `CustomEditor` типу

`QuadricErrorMetrics`, що дозволяє йому напряму переглядати або змінювати публічні елементи окремого об’єкту класу `QuadricErrorMetrics`.

Відповідає за елементи інтерфейсу. Слайдер Complexity надає доступ до установлення деталізації результуючої LOD моделі. Елементи Vertice Count та Triangle Count показують число вершин та полігонів на результуючій моделі відповідно.

Кнопка Simplify викликає підпрограму, що виконує спрощення моделі, доки не досягнеться бажана деталізація.

Кнопка Reset Mesh повертає модель до оригінального рівня деталізації.

Кнопка Calculate Error слугує для обрахунку помилки на результуючій моделі, за допомогою формули, описаної в підрозділі 4.2.

Кнопки Update Collider будує новий колайдер для моделі, для можливості фізичної взаємодії з іншими об’єктами.

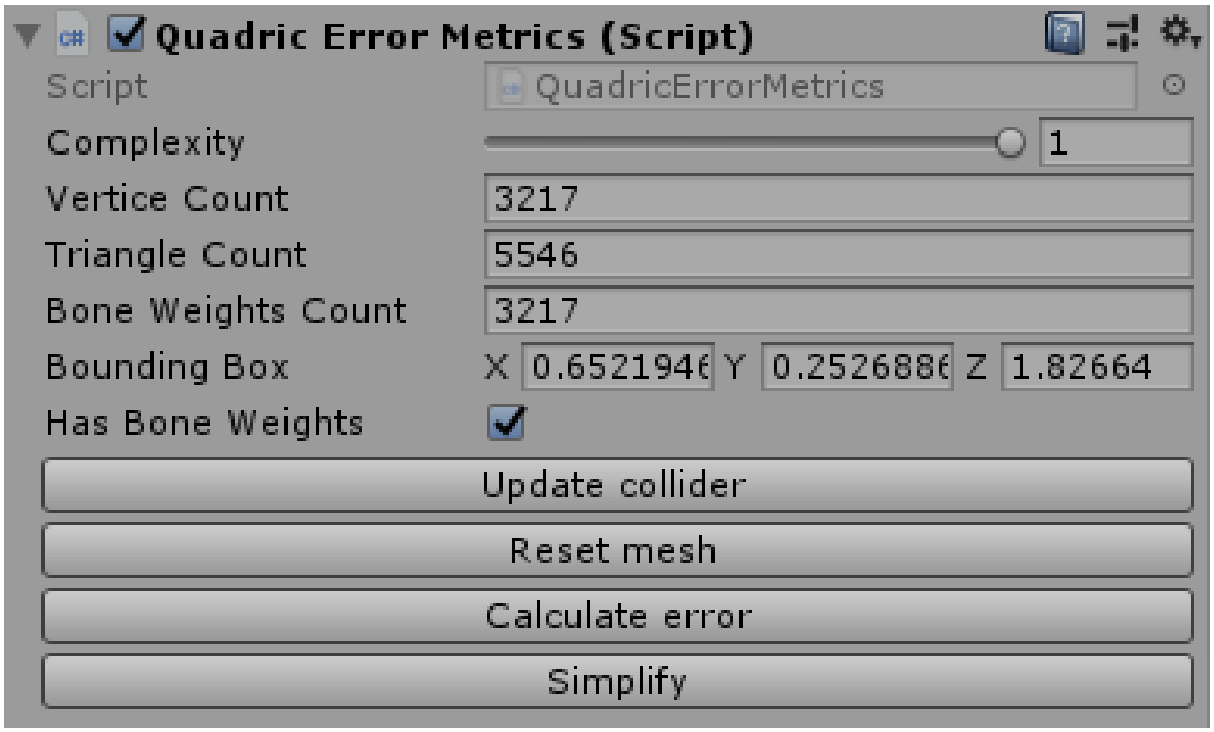


Рис. 3.5. Інтерфейс додатку

3.4. Алгоритм пошуку валідних пар для колапсу

Відповідно до описаного способу в підрозділі 2.5 для колапсу вершин потрібно знайти всі валідні пари для колапсу. З урахуваннями особливостей рушія Unity, таких як подання вершин та трикутників у пам’яті, був розроблений алгоритм для пошуку валідних вершин для колапсу, показаний на рис. 3.6, а код його реалізації – на лістингу 3.2.

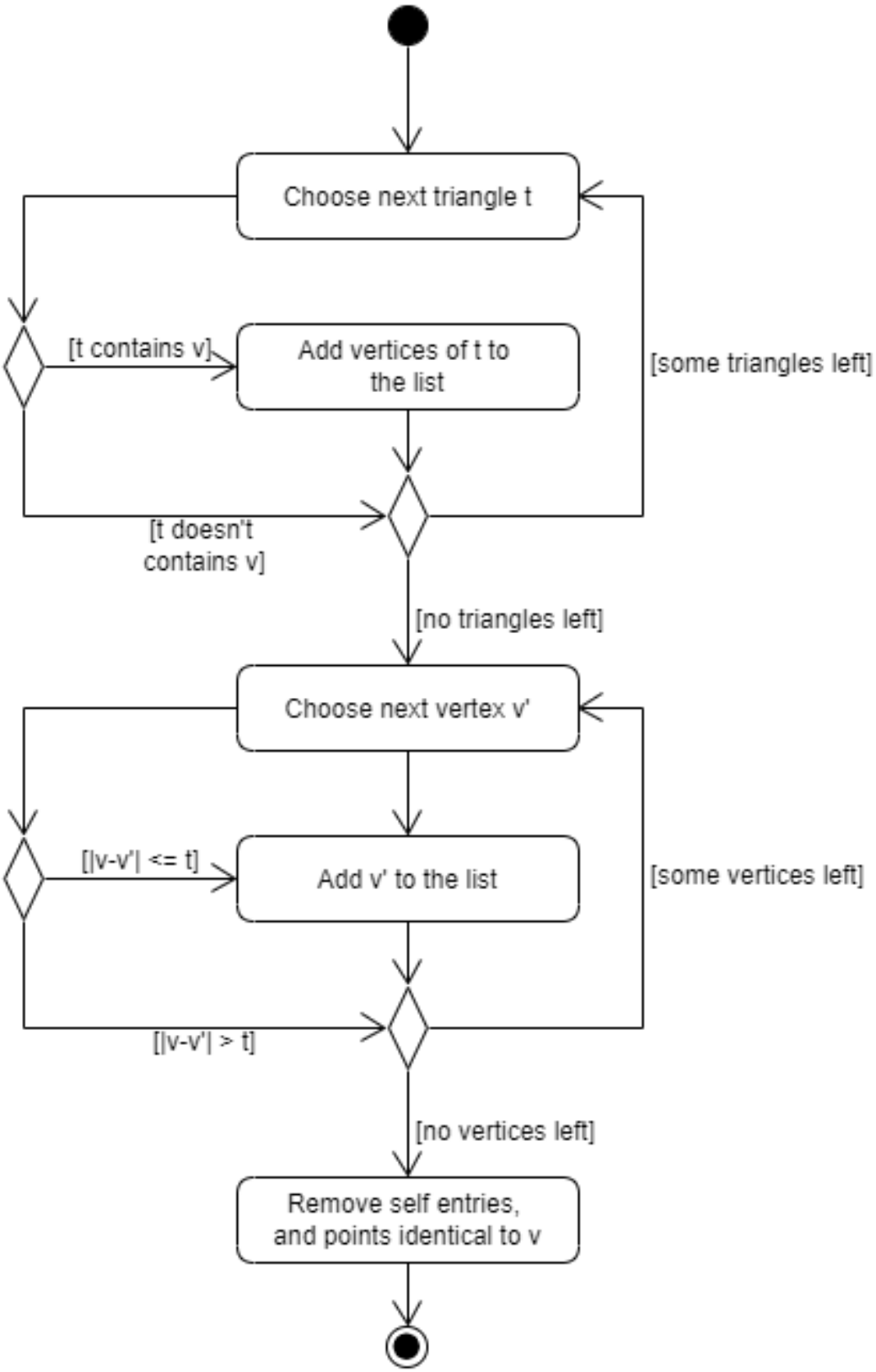


Рис. 3.6. Діаграма діяльності алгоритму знаходження валідних пар вершин для колапсу

Лістинг 3.2. Код алгоритму знаходження валідних пар вершин для колапсу.

```
List<int> ret = new List<int>();
for (int i = 0; i < triangles.Count; i += 3)
{
      for (int j = 0; j < 3; j++)
      {
            int index = i + j;
            if (triangles[index] == v)
            {
                  ret.Add(triangles[i]);
                  ret.Add(triangles[i + 1]);
                  ret.Add(triangles[i + 2]);
            }
      }
}
Vector3 pos = vertices[v];
for (int i = 0; i < vertices.Count; i++)
{
      if (Vector3.Distance(vertices[i], pos) < this.t)
      {
            ret.Add(i);
      }
}
ret.RemoveAll(vertex => vertex == v);
```

3.5. Висновки до розділу 3

В даному розділі сформульовано основні вимоги до розроблення реалізаціїї алгоритмічно-програмного методу генерації LOD, описаного в підрозділі 2.5.

Було розглянуто 2 варіанти виконання реалізації – як розширення до одного з популярних редакторів тривимірних моделей, або як розширення до одного з рушіїїв тривимірної графіки. Було проведено аналіз можливих продуктів для створення розширення та обрано тривимірний рушій Unity Engine.

Наведено розроблену архітектуру реалізаціїї алгоритмічно-програмного методу, та лістинги коду його ключових алгоритмів на мові C#.

4. РЕЗУЛЬТАТИ ЕКСПЕРЕМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1. Визначення об’єктивних характеристик роботи

Серед основних критеріїв роботи програми можна виділити якість результуючої моделі у порівнянні з оригінальною, швидкість роботи так кількість використаної оперативної пам’яті. Заміри проводились на ноутбуці Clevo W370SS, з наступними характеристиками: процесор Intel Core i7 4810mq, з частотою 2.8ГГц, відеокарта Nvidia Geforce GTX 860m, 16 гігайбайт оперативної пам’яті DDR3-1600.

4.2. Критерії вимірюванян якості спрощення

Для якісної оцінки результатів роботи пропонованого алгоритму, потрібно виміряти різницю між оригінальною та результуючою моделлю. Для цього була обрана метрика, що показуэ середнє квадратичне дистанції між апроксимованою моделлю та оригіналом, описана в [19]. Ця метрика є схожою на модель енергії описаною Норрем [20]. Помилка описується $E_i = E(\, M_n, M_i\,)$, спрощеної моделі M_i як:

$$E_i \frac{1}{(\,|X_n|+|X_i|\,)\,D}\left(\sum_{v\epsilon X_n} d^2(v,\, M_i) + \sum_{v\epsilon X_i} d^2(v,\, M_i)\right),$$

де X_n та X_i – набори точок, випадково розкижаних на поверхнях моделей M_n та M_i відповідно, D – розмір діагоналі мінімальної коробки, що повністю охоплює оригінальну модель. Додання D у рівняння слугує для нормалізації реузльтатів помилки моделей різного розміру. Дистанція $d(v, M) = \min(p\epsilon M)||v - p||$ мінімальна дистанція від v до найблищчого полігона M (||*|| – оператор евклідової довжини вектора). Ця метрика використовується лише для оцінки результатів роботи, та не грає ролів в самому алгоритмі. Реалізація даного методу вимірювання помилки на спрощеній моделі надана в лістингу 4.1

Лістинг 4.1. Код алгоритму знаходження валідних пар вершин для колапсу.

```
foreach(Vector3 vertex in vertices)
{
float min = float.MaxValue;
for (int i = 0; i < triangleCount; i+=3)
{
triangle = new Plane(original.vertices[original.triangles[i]],
                     original.vertices[original.triangles[i+1]],
                    original.vertices[original.triangles[i+2]]);
tmp = triangle.GetDistanceToPoint(vertex);
if (Mathf.Abs(tmp) < Mathf.Abs(min)) min = tmp;
}
summ += min*min;
}
foreach(Vector3 vertex in original.vertices)
{
float min = float.MaxValue;
for (int i = 0; i < triangleCount; i+=3)
{
triangle = new Plane(vertices[triangles[i]],
                      vertices[triangles[i+1]],
                      vertices[triangles[i+2]]);
tmp = triangle.GetDistanceToPoint(vertex);
if (Mathf.Abs(tmp) < Mathf.Abs(min)) min = tmp;
}
summ += min*min;
}
```

4.3. Оцінка топології результуючої моделі.

Як показано на рис. 4.1, топологія результуючої моделі суттєво не змінюється. Також слід відмітити відсутність диспропорційних полігонів.

Для якісної оцінки топології була використана наступна формула:

$$E = \frac{1}{|T_i|}\sum_{e\epsilon T_i}\frac{\left(\frac{\max(e_{i\,1},e_{i\,2})}{\min(e_{i\,1},e_{i\,2})}\right)^2+\left(\frac{\max(e_{i\,1},e_{i\,3})}{\min(e_{i\,1},e_{i\,3})}\right)^2+\left(\frac{\max(e_{i\,2},e_{i3})}{\min(e_{i\,2},e_{i\,3})}\right)^2}{3},$$

де T_i – набір трикутників моделі, e_{ij} – сторона j трикутнка i. Ця величина показує квадратичне відношення між сторонами трикутника, та дає якісну оціну топології моделі. Код можна побачити в лістнигу 4.2.

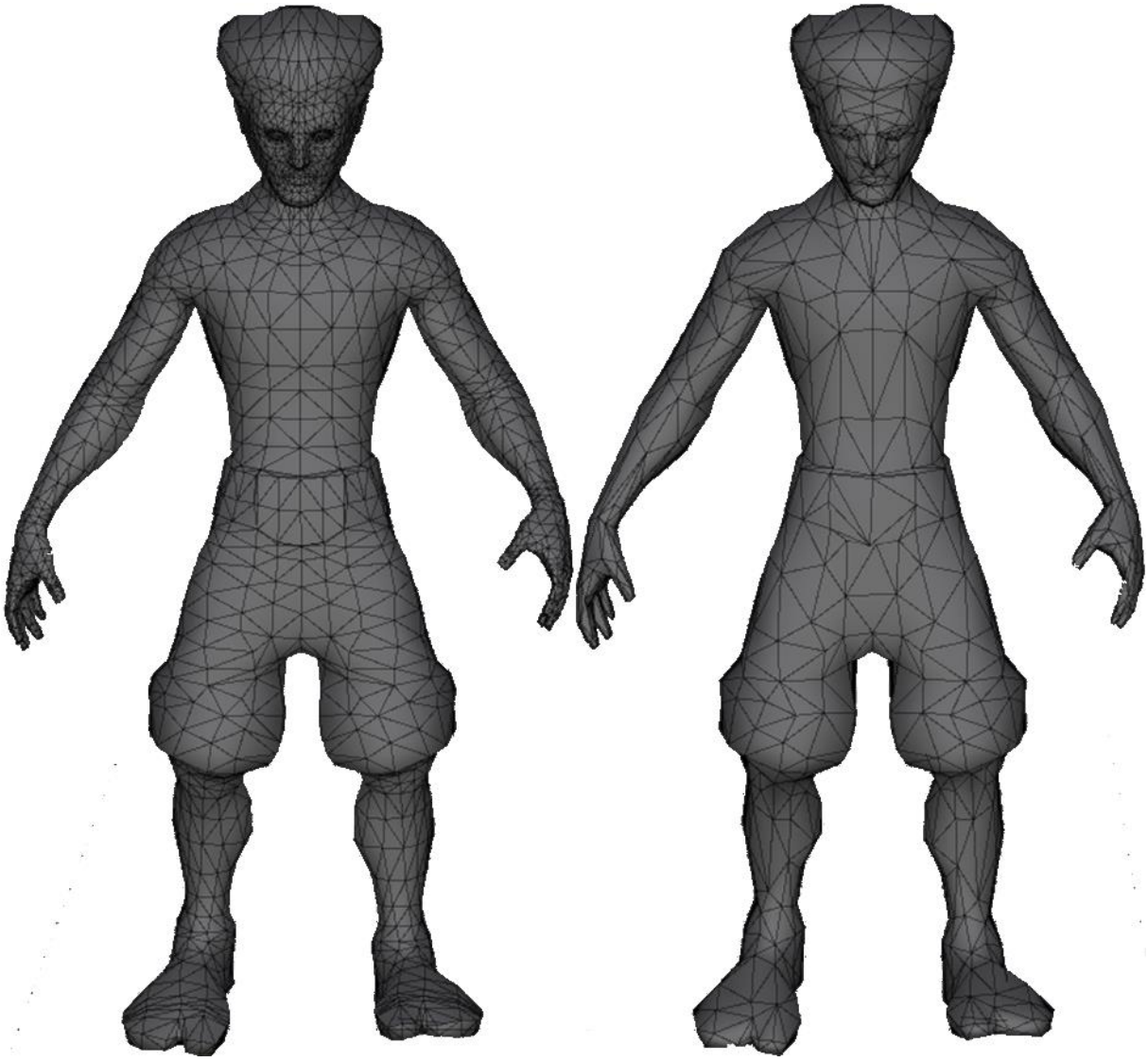


Рис. 4.1. Демонстарція топології оригінальної (зліва) та результуючої моделі (зправа), 5546, та 1906 полігонів відповідно

На рис 4.2 показано графік зміни метрики топології від кількості полігонів на спрощеній моделі. Як видно, метрика на спрощеній моделі

відзиляється від оригінального лиш незначно, навіть на моделях з дуже низькою деталізацією.

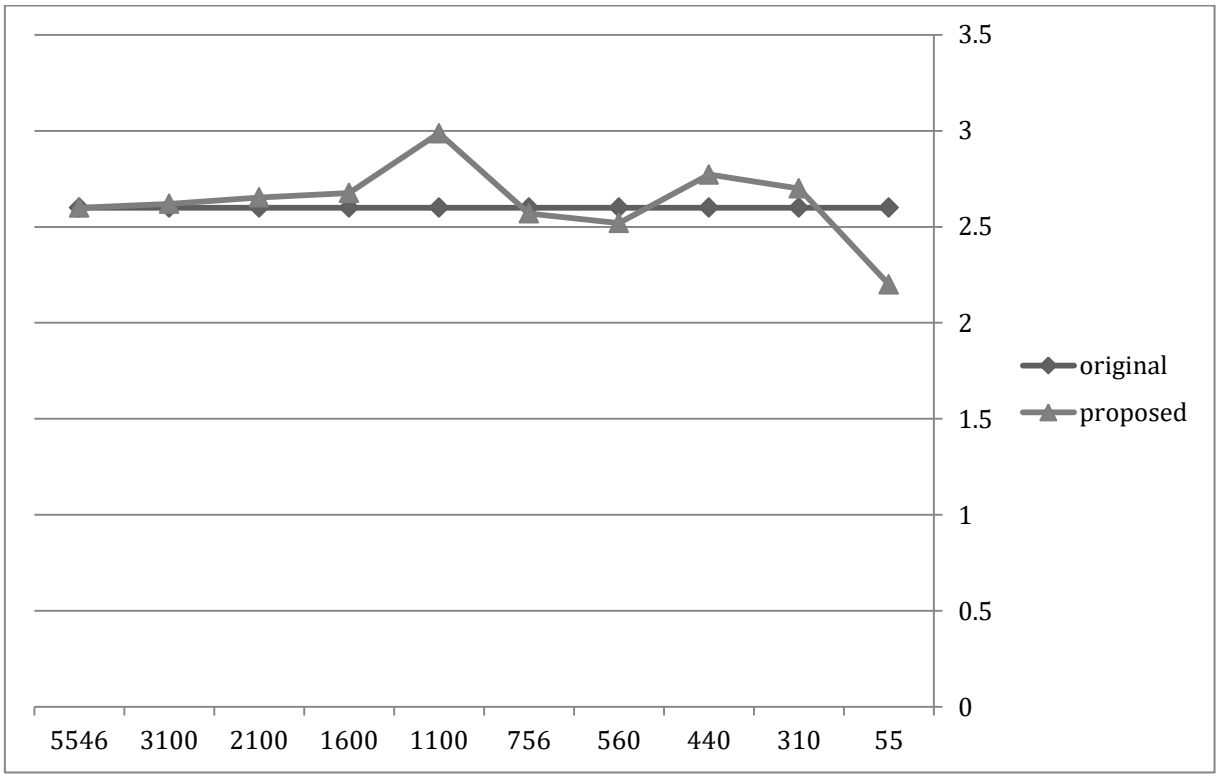


Рис. 4.2. Результат роботи алгоритму на анімованій моделі. Кількість полігонів на моделях з ліво на право: 5546, 4700, 2800, 1500, 550

В лістингу 4.2 наведено код реалізації вимірювання цієї метрики.

Лістинг 4.2. Вимірювання метрики топології

```
avg = 0;
        float edge1;
        float edge2;
        float edge3;
        Vector3 tmp;
        for (int i = 0; i < triangleCount; i += 3)
        {
            tmp = vertices[triangles[i]] - vertices[triangles[i
+ 1]];
            edge1 = tmp.magnitude;
```

Продовження Лістингу 4.2

```
                tmp = vertices[triangles[i]] - vertices[triangles[i
+ 2]];
                edge2 = tmp.magnitude;
                tmp       =       vertices[triangles[i+1]]       -
vertices[triangles[i + 2]];
                edge3 = tmp.magnitude;
                avg   +=   Mathf.Pow(Mathf.Max(edge1,   edge2)   /
Mathf.Min(edge1,edge2),2);
                avg   +=   Mathf.Pow(Mathf.Max(edge2,   edge2)   /
Mathf.Min(edge2, edge2),2);
                avg   +=   Mathf.Pow(Mathf.Max(edge1,   edge3)   /
Mathf.Min(edge1, edge3),2);
            }
            avg /= triangleCount;
```

4.4. Оцінка якості анімації результуючої моделі

Для оцінки якості анімації, був підготовлений спеціальний набір анімацій, що представляє модель у більшості можливих позицій скелету. Результати можна побачити на рис. 4.4. Окремі дрібні деталі, як індивідуальні пальці, залишаються навіть на моделях з дуже низькою кількістю полігонів.

На рис. 4.4 показано графік помилки на результуючій моделі, залежно від кількості полігонів, розрахованої за попередньо описаною формулою. Слід відмітити низьку помилку (<10^-3) навіть при малій кількості полігонів

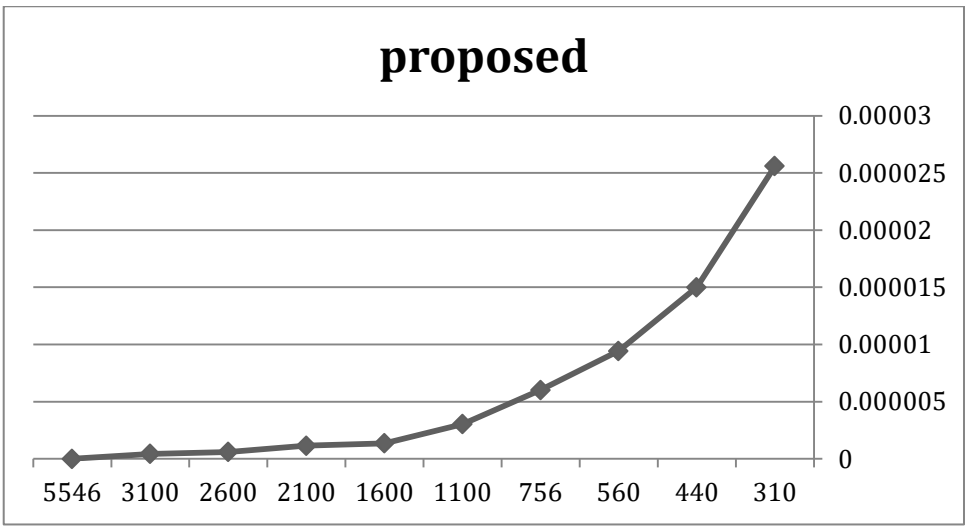


Рис. 4.3. Графік розрахованої помилки на результуючій моделі від кількості полігонів

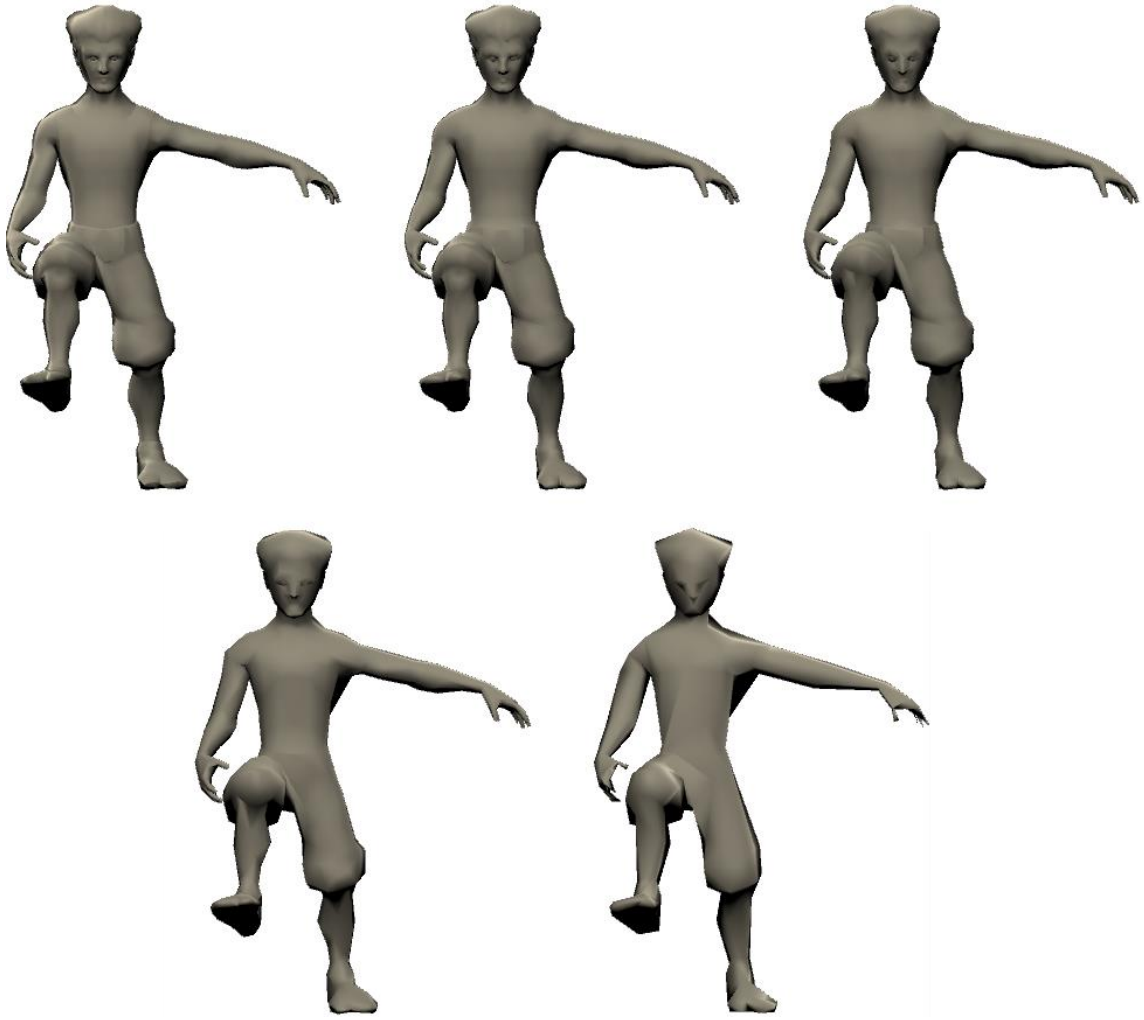


Рис. 4.4. Результат роботи алгоритму на анімованій моделі. Кількість полігонів на моделях з ліво на право: 5546, 4700, 2800, 1500, 550

4.5. Порівняння з існуючими методами

Для порівняння з існуючими методами були обрані параметри швидкості роботи, запропонованої метрики помилки та запропонованої метрики топології. На рис 4.5 показано графік помилки спрощених моделей за допомогою методів половинного колапсу ребра, методу Pauline та запропонованого методу. Можна побачити що запропонований метод в має нижчу помилку на 10..40% у більшості випадків.

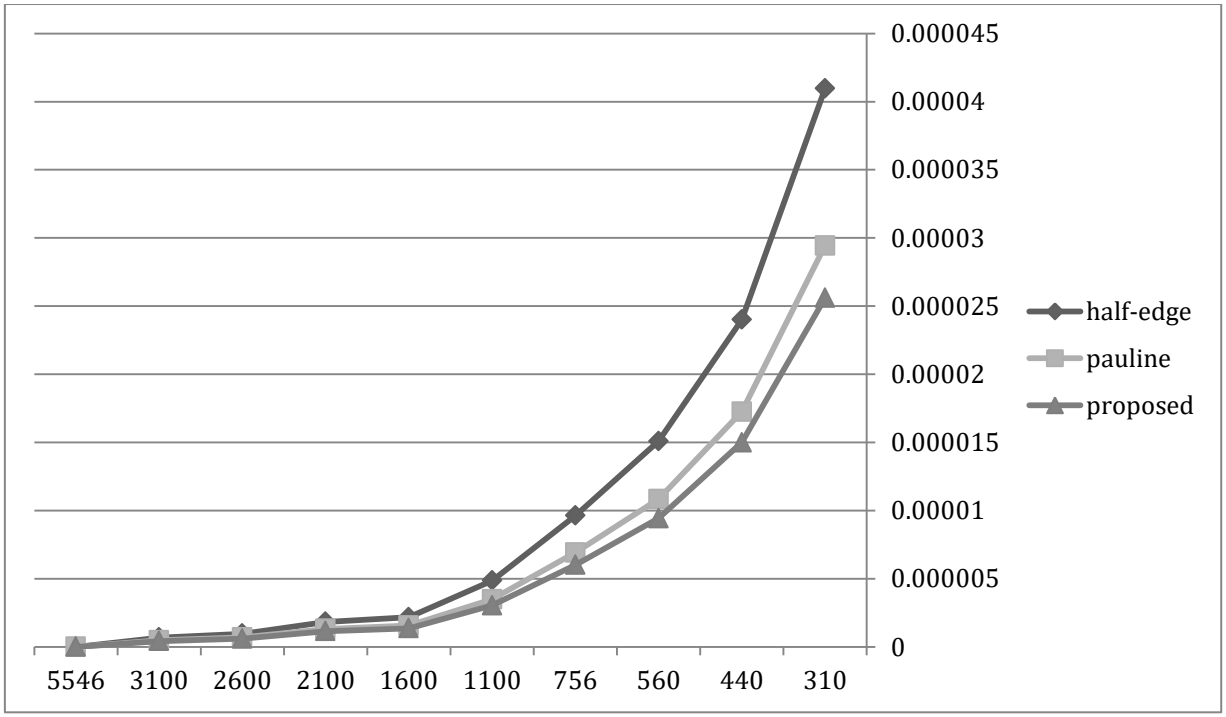


Рис. 4.5. Порівнялний графік розрахованої помилки на результуючій моделі залежно від кількості полігонів

Також для порівняння була виміряна метрика топології для оригінальної моделі, методів що використовують половинний колапс та запропонованого (метод Pauline дає близьку топологію до запропонованого). Результати можна побачити на рис. 4.6. Слід відмітити що метрика топології від

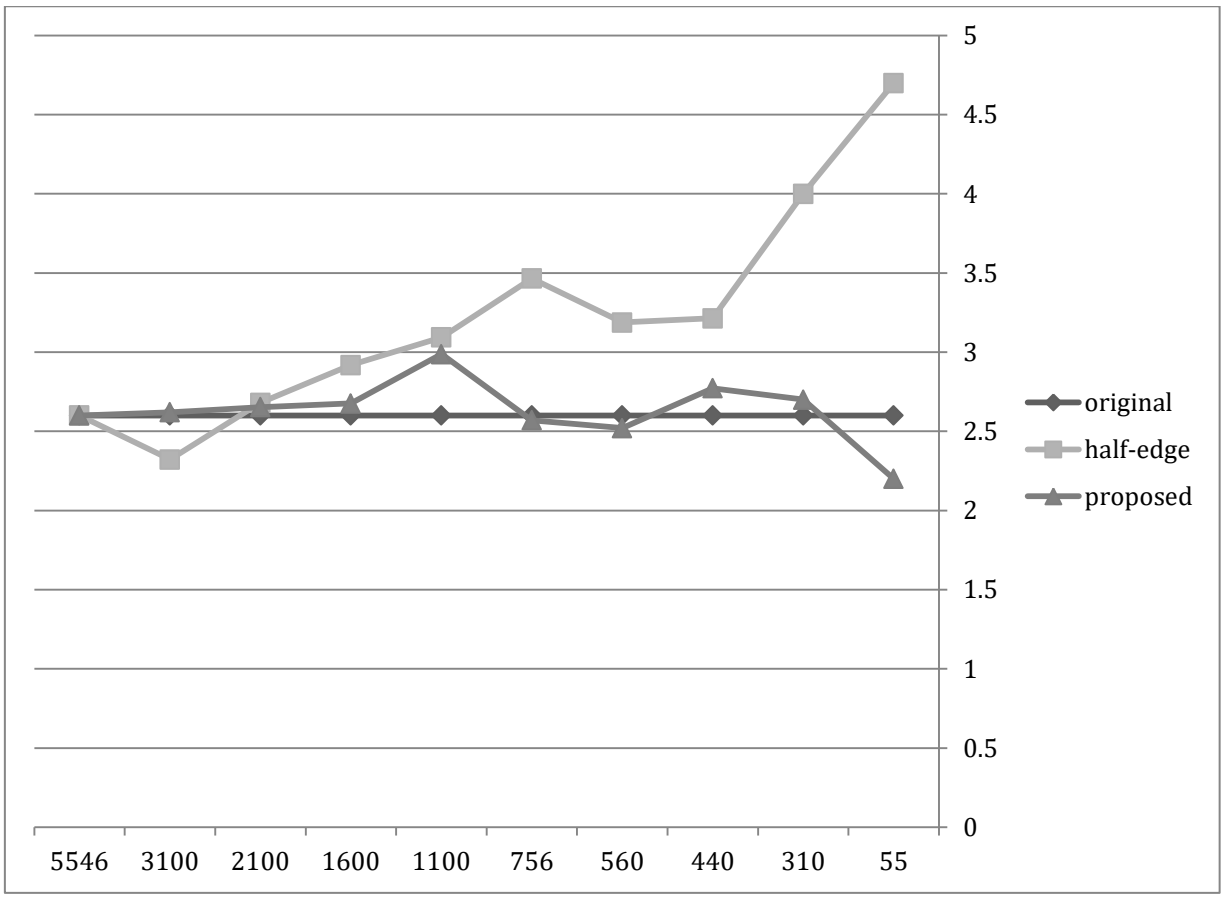


Рис. 4.6. Порівнялний графік розрахованої метрики топології на результуючих моделях та оригінальній від кількості полігонів

4.6. Аналіз отриманих результатів

Дослідження показали що перерахунок ваг дає позитивний вплив на якість анімації моделі. Також запропонований метод дає загалом кращу топологію моделі.

Слід виділити, що запропонований метод є повільнішим його попередників. Для зменьшення цього недоліку слід створити реалізацію запропонованого методу, що використовує відеокарту, оскільки цей метод достатньо легко піддається паралелізації.

4.7. Висновки до розділу 4

Було сформульовано та наведено основні тестові сценарії. Також було створено та наведено набір моделей та анімації для тестування якості спрощення результуючих моделей.

Окрім цього, описано критерії для якісної оцінки точності результуючих моделей. Було наведеном метрику для оцінки топології на тривимірних моделях, та лістинги коду, що містять реалізацію обчислення цих критеріїв.

Досліди показали, що запропонований метод має на 10..40% нижчу помилку від інших методів спрощення анімованої моделі, а також набагато краще зберігає топологію оригінальної моделі. Запропонований метод загалом повільніше від попередніх.

ВИСНОВКИ

Метою даної магістрської дисертації є створення програмного забезпечення, яке надаватиме можливість спрощувати високодетальні анімовані моделі до довільної деталізації, без суттєвої втрати якості анімації, та uv-mapping'у.

На початковому етапі роботи над дисертацією було виконано наступні роботи:

- аналіз засад до використання LOD;
- аналіз існуючих підходів та засобів до спрощення тривимірних моделей;
- аналіз існуючих методів до спрощення анімованих моделей, виявлення їх переваг та недоліків;
- формування науково-інноваційної задачі;

На основі результатів виконання цих робіт було сформульовано алгоритмічно-програмний метод створення LOD-моделей. Для цього було обрано колапс вершин за допомогою квадратичної метрики Гоппса, після чого перераховуються ваги на результуючій вершині.

Було проведено виявлення та аналіз вимог до програмної реалізації запропонованого методу Розглянуто два варіанти виконання програмного продукту: як розширення для редакторів тривимірних моделей або як окремої прикладної програми, побудованої з використанням програмних рушіїв з підтримкою 3D графіки та систем доповненої реальності. На основі порівняльного аналізу засобів розроблення, було обрано програмний рушій Unity Engine, та мова C#. Наведено архітектуру реалізації запропонованого методу та її ключові алгоритми.

Для забезпечення реалізації вимогам було розроблено методику тестування, створено набір тестових моделей, запропоновано якісні критерії для оцінки тривимірних моделей, та їх топології. Виконано аналіз

отриманих результатів, та наведено приклади напрямків вдосконалення реалізації запропонованого методу.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Garland. M. Multiresolution modeling: Survey & future opportunities. / Garland Michale [text] // State of the art report (STAR), Eurographics ’99, 1999.
2. Garland. M. “Multiresolution Modeling for Fast Rendering.” / Michael Garland // Computer Science Department Carnegie Mellon University Pittsburgh, Pennsylvania (1999).
3. Schmalstieg D. , Fuhrmann, A.: Coarse view-dependent levels of detail for hierarchical and deformable models / Dieter Schmalstieg , Anton Fuhrmann // Technical Report TR-186-2- 99-20, Vienna University of Technology, 1999.
4. Robert Huebner. Application of continuous level-of-detail for 3D games // SIGGRAPH 2000 Course Notes, number 39, July 2000.
5. Houle, J.. Simplification and Real-time Smooth Transitions of Articulated Meshes / Jocelyn Houle, Pierre Poulin // Proceedings of Graphics Interface 2001: Ottawa, Ontario, Canada, 7 - 9 2001. – С. 55-60.
6. Hugues, H. Progressive meshes / Hoppe Hugues // SIGGRAPH ’96 Conference Proceedings, Annual Conference Series, 1996. – С. 99-108.
7. Hugues, H. Efficient implementation of progressive meshes. / Hoppe Hugues // Computers & Graphics, 22(1):27-36, February 1998.
8. Hugues, H.. View-dependent refinement of progressive meshes. / Hoppe Hugues // In Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH ’97).
9. Scopigno, Roberto & Cnuce, Istituto & Ricerche, Consiglio. (1998). Simplification, LOD and Multiresolution - Principles and Applications.
10. Clark, J.. Hierarchical Geometric Model for Visible Surface / James Clark // Algorithms. communications of the ACM. October, 1976. vol. 19: pp. 547-554.

11. Chen, B. & Edward, J. & Ii, LOD-Sprite Technique for Accelerated Terrain Rendering / Baoquan Chen, J. Edward // Proceedings Visualization '99 (Cat. No.99CB37067), March 2009
12. Taylor, D., Barrett, W. An algorithm for continuous resolution polygonalizations of a discrete surface./ David Taylor, William Barret // Proceedings of Graphics Interface '94: Banff, Alberta, Canada, 18 - 20 May 1994, 33-42
13. Luebke, H.& Hallen, B. Perceptually Driven Simplification for Interactive Rendering. / David Luebke, Benjamin Hallen // EGSR 2001: Rendering Techniques 2001 pp 223-234
14. Cignoni, P., Montani, C., Scopigno, R.. A comparison of mesh simplification algorithms / Paolo Cignoni, Claudio Montani, Roberto Scopigno // Computers & Graphics, 22(1): P. 37–54, September 1997.
15. Erikson, C. Polygonal simplification: An overview. Technical Report / Carl Erikson // University of North Carolina, Department of Computer Science, 1996.
16. Mannos, J. Sakrison, D. The effects of a visual fidelity criterion of the encoding of images,/ James Manos, David Sakrison // IEEE Transactions on Information Theory, vol. 20, no. 4, pp. 525-536, July 1974, doi: 10.1109/TIT.1974.1055250.
17. Guthe, M. & Borodin, P. & Klein, Reinhard. (2005). Fast and Accurate Hausdorff Distance Calculation between Meshes./ Michale Guthe, Pavel Borodin //. Journal of WSCG. 13. P. 41-48.
18. Hugues, H... New quadric metric for simplifying meshes with appearance attributes. / Hoppe Hugues //In IEEE Visualization 1999. October 1999. – P. 59-66.
19. Garland, M. Heckbert, P. Surface simplification using quadric error metrics./ Michael Garland, Paul Heckbert // SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques August 1997 P. 209–216

20. Hugues, H... Quadric metric for simplifying meshes with appearance attributes / Hoppe Hugues // Microsoft Research, USA 1999.

ДОДАТКИ

Додаток 1

Лістинги коду програмної реалізації

Лістинг 1. Модуль спрощення анімованої моделі

```
public class QuadricErrorMetrics : MonoBehaviour
{
    //[SerializeField]
    private Mesh input;
    private Mesh original;
    private MeshCollider originalCollider;
    SkinnedMeshRenderer meshRenderer;
    //MeshFilter meshFilter;
    [HideInInspector]
    public List<Vector3> vertices;
    [HideInInspector]
    public List<int> triangles;
    [HideInInspector]
    List<VertexData> vertexDatas = new List<VertexData>();
    [HideInInspector]
    public List<BoneWeight> boneWeights;

    [Range(0.01f, 1.0f)]
    public float complexity;
    public int verticeCount;
    public int triangleCount;
    public int boneWeightsCount;
    private int maxVertices;
    [HideInInspector]
    public float coef;
    [HideInInspector]
    public float _w0,_w1,_w2,_w3;
    public Vector3 boundingBox = new Vector3(0,0,0);
    private bool hasBoneWeights;
    public bool halfCollapse;
    public float avg;
```

```
    private void Start()
    {
        Init();
    }

    private void Init()
    {
        meshRenderer = GetComponent<SkinnedMeshRenderer>();
//meshFilter = GetComponent<MeshFilter>();
        input = meshRenderer.sharedMesh;
        original = meshRenderer.sharedMesh;
        vertices = new List<Vector3>(input.vertices);
        triangles = new List<int>(input.triangles);
        boneWeights = new List<BoneWeight>(input.boneWeights);
        maxVertices = input.vertexCount;
        verticeCount = vertices.Count();
        triangleCount = triangles.Count()/3;
        boneWeightsCount = boneWeights.Count();
        halfCollapse = false;
        avg = 0;
        UpdateCollider();
        originalCollider = GetComponent<MeshCollider>();

        boundingBox.x = vertices.Max(v => Mathf.Abs(v.x));
        boundingBox.y = vertices.Max(v => Mathf.Abs(v.y));
        boundingBox.z = vertices.Max(v => Mathf.Abs(v.z));

        complexity=1.0f;
        hasBoneWeights = input.boneWeights.Count() > 0;
        InitVertexData();
        UpdateMesh();
```

```
        //InvokeRepeating("ContractCoroutine", 0 , 0.001f);
    }

    void InitVertexData()
    {
        for (int i = 0; i < vertices.Count; i++)
        {
            VertexData data = new VertexData();
            data.edges = GetEdges(i);
            vertexDatas.Add(data);
        }

        //int error metrics
        for (int i = 0; i < triangles.Count; i += 3)
        {
            int v0 = triangles[i];
            int v1 = triangles[i + 1];
            int v2 = triangles[i + 2];
            Vector3 p0 = vertices[v0];
            Vector3 p1 = vertices[v1];
            Vector3 p2 = vertices[v2];
            Vector3 normal = Vector3.Cross(p1 - p0, p2 -
p1).normalized;
            float a = normal.x;
            float b = normal.y;
            float c = normal.z;
            float d = -Vector3.Dot(normal, p0);
            Matrix4x4 q = new Matrix4x4();
            q.SetRow(0, new Vector4(a * a, a * b, a * c, a * d));
            q.SetRow(1, new Vector4(b * a, b * b, b * c, b * d));
            q.SetRow(2, new Vector4(c * a, c * b, c * c, c * d));
            q.SetRow(3, new Vector4(d * a, d * b, d * c, d * d));
```

```
            vertexDatas[v0].errorMatrix =
MathUtil.AddMatrix(vertexDatas[v0].errorMatrix, q);

            vertexDatas[v1].errorMatrix =
MathUtil.AddMatrix(vertexDatas[v1].errorMatrix, q);

            vertexDatas[v2].errorMatrix =
MathUtil.AddMatrix(vertexDatas[v2].errorMatrix, q);

        }


        //Init Errors

        for (int i = 0; i < vertexDatas.Count; i++)

        {

            CalcError(i);

        }

    }


    void CalcError(int i)

    {

        var vertexData = vertexDatas[i];

        vertexData.errors.Clear();

        vertexData.vBest.Clear();


        for (int j = 0; j < vertexData.edges.Count; j++)

        {

            int edge0 = i;

            int edge1 = vertexData.edges[j];


            if
(Mathf.Approximately(Vector3.Distance(vertices[edge0],
vertices[edge1]), 0))

            {

                vertexData.errors.Add(-1);

                vertexData.vBest.Add(vertices[edge0]);

                continue;

            }
```

```
            Matrix4x4 errorCombine =
MathUtil.AddMatrix(vertexDatas[edge0].errorMatrix,
vertexDatas[edge1].errorMatrix);

            Matrix4x4 differentialMatrix = new Matrix4x4();

            differentialMatrix.SetRow(0, new
Vector4(errorCombine.m00, errorCombine.m01, errorCombine.m02,
errorCombine.m03));

            differentialMatrix.SetRow(1, new
Vector4(errorCombine.m01, errorCombine.m11, errorCombine.m12,
errorCombine.m13));

            differentialMatrix.SetRow(2, new
Vector4(errorCombine.m02, errorCombine.m12, errorCombine.m22,
errorCombine.m23));

            differentialMatrix.SetRow(3, new Vector4(0, 0, 0, 1));

            Vector4 vbest = differentialMatrix.inverse * new
Vector4(0, 0, 0, 1);

            float error = Vector4.Dot(vbest, errorCombine * vbest);

            if (differentialMatrix.determinant == 0)

            {

                Vector3 p0 = vertices[edge0];

                Vector3 p1 = vertices[edge1];

                Vector3 mid = (p0 + p1) / 2;

                float errorP0 = Vector4.Dot(p0, errorCombine * p0);

                float errorP1 = Vector4.Dot(p1, errorCombine * p1);

                float errorMid = Vector4.Dot(mid, errorCombine *
mid);

                if (errorP0 <= errorP1 && errorP0 <= errorMid)

                {

                    vbest = p0;

                    error = errorP0;

                }

                else if (errorP1 <= errorP0 && errorP1 <= errorMid)

                {

                    vbest = p1;

                    error = errorP1;

                }

                else
```

```
            {
                vbest = mid;
                error = errorMid;
            }
        }

        //bounding box
        if (Mathf.Abs(vbest.x) > boundingBox.x+0.1f ||
            Mathf.Abs(vbest.y) > boundingBox.y+0.1f ||
            Mathf.Abs(vbest.z) > boundingBox.z+0.1f  )
        {
            vertexData.errors.Add(float.MaxValue);
            vertexData.vBest.Add(Vector3.zero);
            continue;
        }

        if (error < 0)
            error = 0;

        vertexData.errors.Add(error);
        vertexData.vBest.Add(vbest);
    }
}

List<int> GetEdges(int v)
{
    List<int> ret = new List<int>();
    //add real edge
    for (int i = 0; i < triangles.Count; i += 3)
    {
        for (int j = 0; j < 3; j++)
        {
```

```
                int index = i + j;
                if (triangles[index] == v)
                {
                    ret.Add(triangles[i]);
                    ret.Add(triangles[i + 1]);
                    ret.Add(triangles[i + 2]);
                }
            }
        }
        //close vertices can be a edge
        Vector3 pos = vertices[v];
        for (int i = 0; i < vertices.Count; i++)
        {
            if (Vector3.Distance(vertices[i], pos) < 0.01f)
            {
                ret.Add(i);
            }
        }
        ret.RemoveAll(vertex => vertex == v);
        return ret.Distinct().ToList();
    }


    void FindBestContract(out int v0, out int v1, out Vector3
newPos)
    {
        float minError = float.MaxValue;
        VertexData target = null;
        int targetIndex = 0;
        foreach (var vd in vertexDatas)
        {
            float min = float.MaxValue;
            foreach (var e in vd.errors)
            {
```

```
                if (e < min)
                {
                    min = e;
                }
            }

            if (min <= minError)
            {
                minError = min;
                target = vd;
                targetIndex = vd.errors.IndexOf(min);
            }
        }
        v0 = vertexDatas.IndexOf(target);
        v1 = target.edges[targetIndex];
        newPos = target.vBest[targetIndex];
    }
    public void Simplify()
    {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        do {
            ContractCoroutine();
        } while (vertices.Count > (int)(maxVertices*complexity));
        stopwatch.Stop();
        UnityEngine.Debug.Log("Time in ms: " +
stopwatch.Elapsed.Milliseconds.ToString());
    }
    void ContractCoroutine()
    {
        // if (vertices.Count > (int)(maxVertices*complexity))
        // {
            int v0, v1;
```

```
            Vector3 newPos;
            FindBestContract(out v0, out v1, out newPos);
            Contract(v0, v1, newPos);
            UpdateMesh();
        // }
    }

    BoneWeight mergeBoneWeights(BoneWeight weight1, BoneWeight
weight2, float coef)
    {
        //need revision
        if (weight1.Equals(weight2)) return weight1;
        List<Weight> w1 = new List<Weight>();
        w1.Add(new Weight(weight1.boneIndex0, weight1.boneIndex0));
        w1.Add(new Weight(weight1.boneIndex1, weight1.boneIndex1));
        w1.Add(new Weight(weight1.boneIndex2, weight1.boneIndex2));
        w1.Add(new Weight(weight1.boneIndex3, weight1.boneIndex3));
        List<Weight> w2 = new List<Weight>();
        w2.Add(new Weight(weight2.boneIndex0, weight2.boneIndex0));
        w2.Add(new Weight(weight2.boneIndex1, weight2.boneIndex1));
        w2.Add(new Weight(weight2.boneIndex2, weight2.boneIndex2));
        w2.Add(new Weight(weight2.boneIndex3, weight2.boneIndex3));
        List<Weight> result = new List<Weight>();
        int index = w2.FindIndex(w => w.boneIndex ==
weight1.boneIndex0);
        if (index != -1) result.Add(new Weight(weight1.boneIndex0,
                                               weight1.weight0*coef
+ w2[index].weight*(1-coef)));
        else result.Add(new Weight(weight1.boneIndex0,
weight1.weight0*coef));

        index = w2.FindIndex(w => w.boneIndex ==
weight1.boneIndex1);
        if (index != -1) result.Add(new Weight(weight1.boneIndex1,
```

```
                                                    weight1.weight1*coef
+ w2[index].weight*(1-coef)));

        else result.Add(new Weight(weight1.boneIndex1,
weight1.weight1*coef));



        index = w2.FindIndex(w => w.boneIndex ==
weight1.boneIndex2);

        if (index != -1) result.Add(new Weight(weight1.boneIndex2,

                                                    weight1.weight2*coef
+ w2[index].weight*(1-coef)));

        else result.Add(new Weight(weight1.boneIndex2,
weight1.weight2*coef));



        index = w2.FindIndex(w => w.boneIndex ==
weight1.boneIndex3);

        if (index != -1) result.Add(new Weight(weight1.boneIndex3,

                                                    weight1.weight3*coef
+ w2[index].weight*(1-coef)));

        else result.Add(new Weight(weight1.boneIndex3,
weight1.weight3*coef));



        result.Sort((a,b) => b.weight.CompareTo(a.weight));

        //result.Reverse();



        float summ = result.Sum(x => x.weight);



        BoneWeight newBoneWeight = new BoneWeight();

        _w0 = result[0].weight/summ;

        _w1 = result[1].weight/summ;

        _w2 = result[2].weight/summ;

        _w3 = result[3].weight/summ;

        newBoneWeight.boneIndex0 = result[0].boneIndex;

        newBoneWeight.weight0 = result[0].weight/summ;

        newBoneWeight.boneIndex1 = result[1].boneIndex;

        newBoneWeight.weight1 = result[1].weight/summ;

        newBoneWeight.boneIndex2 = result[2].boneIndex;
```

```
        newBoneWeight.weight2 = result[2].weight/summ;
        newBoneWeight.boneIndex3 = result[3].boneIndex;
        newBoneWeight.weight3 = result[3].weight/summ;

        return newBoneWeight;
    }

    void Contract(int v0, int v1, Vector3 newPos)
    {
        int newVertexId = vertices.Count;
        if (halfCollapse)
        {
            vertices.Add(new Vector3(vertices[v0].x, vertices[v0].y,
vertices[v0].z)); //half-collapse
            boneWeights.Add(boneWeights[v0]); //half collapse
        }
        else
        {
            //merge v0 and v1 into end of vertexDatas
            vertices.Add(newPos);

            float dist0 = Vector3.Distance(vertices[v0], newPos);
            float dist1 = Vector3.Distance(vertices[v1], newPos);

            coef = dist0 / (dist0 + dist1);
            //add boneWeight fo resulting vertex
            BoneWeight newBoneWeight =
mergeBoneWeights(boneWeights[v0], boneWeights[v1],
coef);//boneWeights[v0];
            boneWeights.Add(newBoneWeight);
        }

        VertexData newVertexData = new VertexData();
```

```
newVertexData.Combine(vertexDatas[v0]);
newVertexData.Combine(vertexDatas[v1]);
vertexDatas.Add(newVertexData);

CalcError(newVertexId);
if (v0 < v1)
{
    int vt = v0;
    v0 = v1;
    v1 = vt;
}
vertices.RemoveAt(v0);
vertexDatas.RemoveAt(v0);
boneWeights.RemoveAt(v0);
vertices.RemoveAt(v1);
vertexDatas.RemoveAt(v1);
boneWeights.RemoveAt(v1);
foreach (var vertexData in vertexDatas)
{
    vertexData.RemoveEdge(v0);
    vertexData.RemoveEdge(v1);
}

//create edge from newVertexData (a, b) to (b, a)
int finalId = vertexDatas.Count - 1;
for (int i = 0; i < newVertexData.edges.Count; i++)
{
    int v = newVertexData.edges[i];
    vertexDatas[v].edges.Add(finalId);
    vertexDatas[v].errors.Add(newVertexData.errors[i]);
    vertexDatas[v].vBest.Add(newVertexData.vBest[i]);
}
```

```
// update triangles vertex id
for (int i = triangles.Count - 1; i >= 2; i -= 3)
{
    int hitCount = 0;
    for (int j = 0; j < 3; j++)
    {
        int index = i - j;
        int vertex = triangles[index];
        bool hit = false;
        if (vertex == v0 || vertex == v1)
        {
            hit = true;
            hitCount++;
        }
        if (hit)
        {
            if (hitCount == 1)
            {
                triangles[index] = newVertexId;
            }
            else if (hitCount == 2)
            {
                triangles.RemoveAt(i);
                triangles.RemoveAt(i - 1);
                triangles.RemoveAt(i - 2);
                break;
            }
        }
    }
}
```

```
    for (int i = 0; i < triangles.Count; i++)
    {
        int vertex = triangles[i];
        int sub = 0;
        if (vertex > v0)
            sub += 1;
        if (vertex > v1)
            sub += 1;
        triangles[i] = vertex - sub;
    }
}

void UpdateMesh()
{
    Mesh output = new Mesh();
    output.vertices = vertices.ToArray();
    output.triangles = triangles.ToArray();
    output.boneWeights = boneWeights.ToArray();
    output.bindposes= input.bindposes;

    verticeCount = vertices.Count();
    triangleCount = triangles.Count()/3;
    boneWeightsCount = boneWeights.Count();
    output.RecalculateNormals();
    output.RecalculateTangents();
    //meshFilter.mesh = output;

    meshRenderer.sharedMesh = output;
}

public void CalcApproximationError()
{
```

```
        //UpdateCollider();
        float summ = 0;
        int divisor = verticeCount;
        float tmp;
        Plane triangle = new Plane();
        Vector3 vertex;
        //int iteration = 0;

        for (int j = 0; j < triangleCount; j+=3)
        {
            float min = float.MaxValue;

            vertex =
(vertices[triangles[j]]+vertices[triangles[j+1]]+vertices[triangles[
j+2]])/3;
            for (int i = 0; i < original.triangles.Length; i += 3)
            {
                triangle = new
Plane(original.vertices[original.triangles[i]],

original.vertices[original.triangles[i + 1]],

original.vertices[original.triangles[i + 2]]);
                tmp = triangle.GetDistanceToPoint(vertex);
                if (Mathf.Abs(tmp) < Mathf.Abs(min)) min = tmp;
            }
            summ += min * min;

        }

        float error = summ/divisor;
        //error /= Vector3.Distance(Vector3.zero, boundingBox);
        UnityEngine.Debug.Log("Error = " + error.ToString());
    }
```

```
    public void CalcTopologyError()
    {
        avg = 0;
        float edge1;
        float edge2;
        float edge3;
        Vector3 tmp;
        for (int i = 0; i < triangleCount; i += 3)
        {
            tmp = vertices[triangles[i]] - vertices[triangles[i +
1]];
            edge1 = tmp.magnitude;
            tmp = vertices[triangles[i]] - vertices[triangles[i +
2]];
            edge2 = tmp.magnitude;
            tmp = vertices[triangles[i+1]] - vertices[triangles[i +
2]];
            edge3 = tmp.magnitude;
            avg += Mathf.Pow(Mathf.Max(edge1, edge2) /
Mathf.Min(edge1,edge2),2);
            avg += Mathf.Pow(Mathf.Max(edge2, edge2) /
Mathf.Min(edge2, edge2),2);
            avg += Mathf.Pow(Mathf.Max(edge1, edge3) /
Mathf.Min(edge1, edge3),2);
        }
        avg /= triangleCount;
    }


    public void UpdateCollider()
    {
        MeshCollider collider = GetComponent<MeshCollider>();


        Mesh output = new Mesh();
        output.vertices = vertices.ToArray();
        output.triangles = triangles.ToArray();
```

```
            output.boneWeights = boneWeights.ToArray();
            output.bindposes= input.bindposes;

            meshRenderer.BakeMesh(output);
            collider.sharedMesh = null;
            collider.sharedMesh = output;

        }

        public void Reset()
        {
            meshRenderer.sharedMesh = original;
            complexity = 1.0f;
            vertices = new List<Vector3>(original.vertices);
            triangles = new List<int>(original.triangles);
            boneWeights = new List<BoneWeight>(original.boneWeights);
            maxVertices = original.vertexCount;
            verticeCount = vertices.Count();
            triangleCount = triangles.Count()/3;
            boneWeightsCount = boneWeights.Count();

            vertexDatas = new List<VertexData>();

            InitVertexData();
            UpdateMesh();
        }
    }
```

Додаток 2

Копія слайдів презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

Алгоритмічно-програмний метод генерації LOD моделей

Виконав: Лось Ігор Анатолійович

Науковий керівник: доцент, к.т.н., доцент Сулема Євгенія Станіславівна

Київ – 2018

АКТУАЛЬНІСТЬ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВИЯВЛЕННЯ SQL-ІН'ЄКЦІЙ

Спрощені моделі постійно використовуються у тривимірній графіці для пришвидшення рендерингу.

Існуючі методи, що не потребують людського втручання, мають суттєві недоліки.

Об'єктом дослідження є процес створення тривимірних анімованих моделей.

Предметом дослідження є спосіб спрощення тривимірних анімованих моделей без втрати якості анімації

ПОСТАНОВКА НАУКОВО-ТЕХНІЧНОЇ ЗАДАЧІ

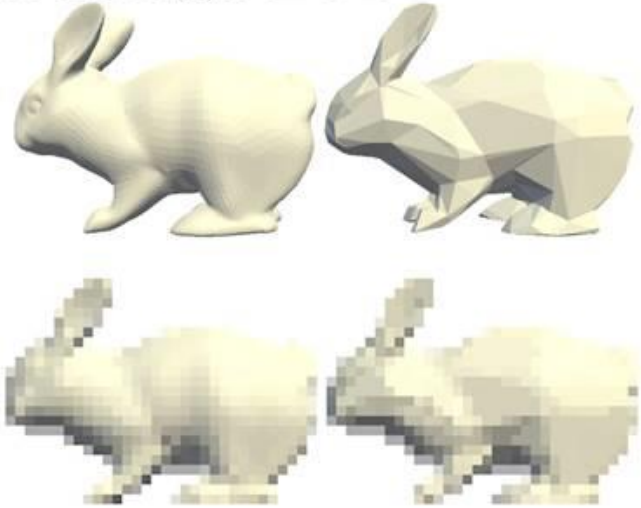
.

Наукове завдання – розробити алгоритмічно-програмний метод спрощення моделі, який зберігає топологію, параметри вершин та перерозподіляє ваги вершин результуючої моделі.

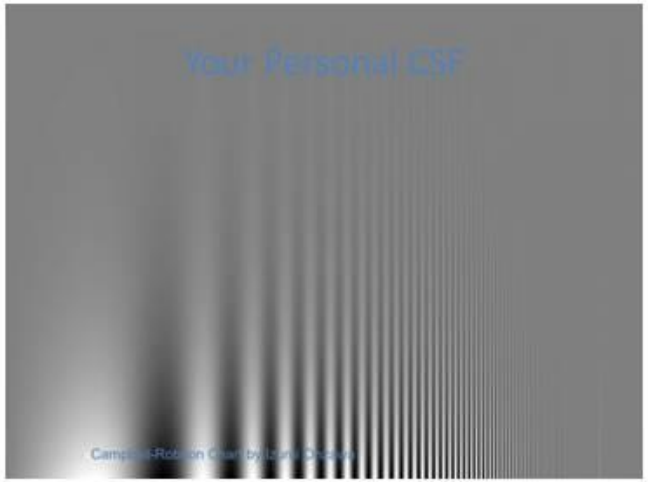
Мета дослідження – створення програмного забезпечення, яке надаватиме можливість спрощувати високодетальні анімовані моделі до довільної деталізації, без суттєвої втрати якості анімації, та uv-mapping'у.

Основні засади LOD

Обмеження сучасних дисплеїв



Особливості сприйняття зображень людиною



КЛАСИФІКАЦІЯ МЕТОДІВ LOD

За оцінкою помилки

- Візуальні
- Геометричні

За методом використання

- Discrete LOD
- Continuous LOD
- View-Dependant LOD

За метрикою оцінки

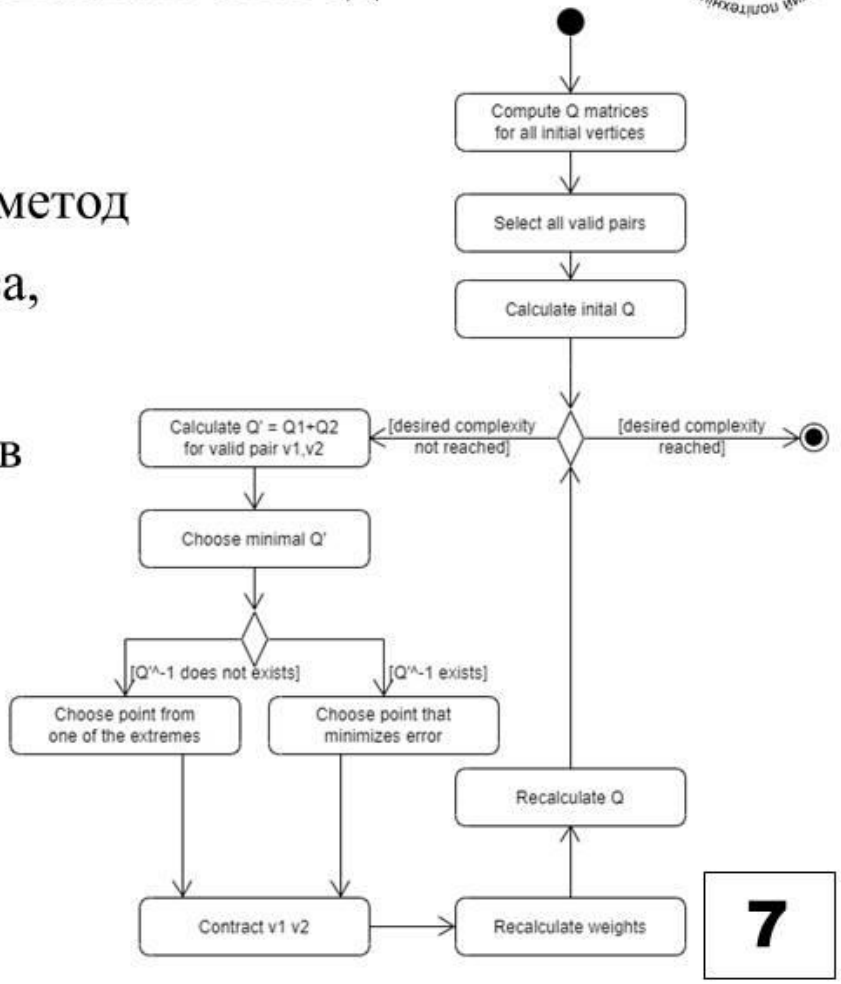
- Локальні
- Глобальні

Порівняння існуючих методів спрощення анімованих моделей

Метод	Збереження топології	Перерахунок ваг	Збереження інших атрибутів вершин	Швидкість генерації
Schmalstieg	-	No need	-	+
Huebner	-	No need	+	+
Poulin	+	-	+	+
Proposed	+	+	+	+/-

Запропонований метод

- Колапс вершин, локальний метод
- Квадратична метрика Гоппса, геометрична оцінка
- Можна використовувати як в DLOD так і CLOD
- Перерахунок ваг.

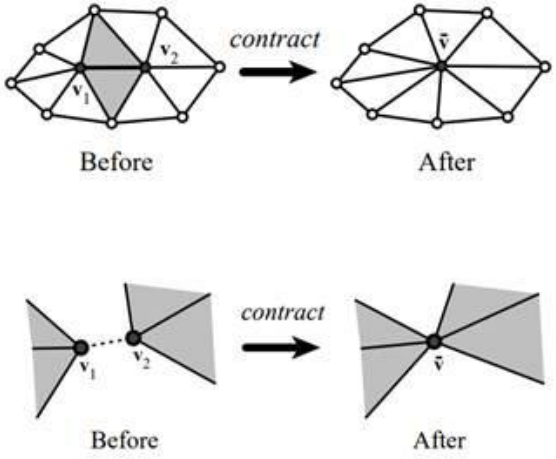


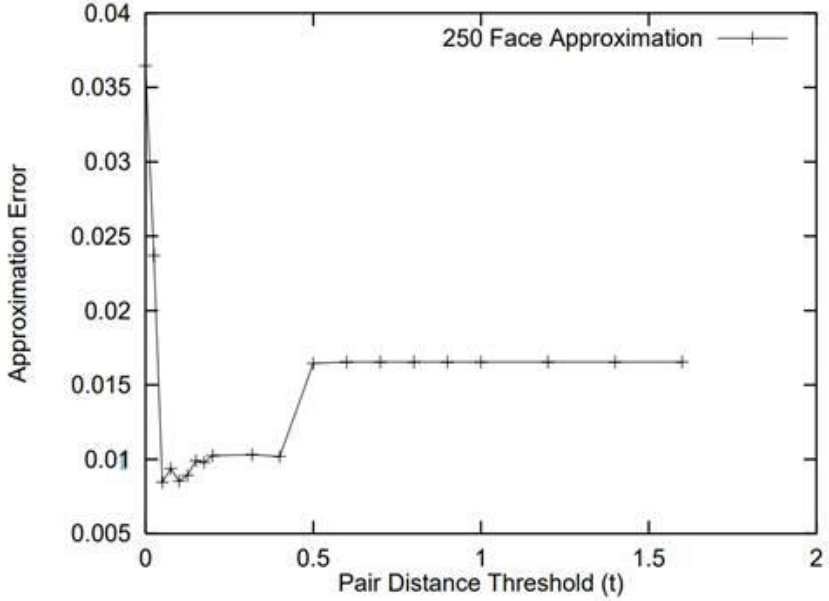
Вибір вершин для колапсу

(v1, v2) лежать на одному ребрі моделі, або

$\|v1-v2\| < t$, де те t – деяке порогове значення,

де v1, v2 – дві довільні вершини на моделі





Квадратична метрика Гоппса

Переваги: низька середня помилка, висока швидкість ітерації, економія пам’яті

Недоліки: висока максимальна помилка

$\Delta(v) = v^T Q\, v \Delta(v) = v^T Q\, v,$

$\boldsymbol{Q' = Q_1 + Q_2},$

$$v' = \begin{bmatrix} q'_{11} & q'_{12} & q'_{13} & q'_{14} \\ q'_{21} & q'_{22} & q'_{23} & q'_{24} \\ q'_{31} & q'_{32} & q'_{33} & q'_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} * \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

, де вершина

$v = \begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}^T$,

Q – матриця квадратичних помилок

v

q – елемент матриці Q

Перерозподіл ваг

- Визначити квадратичну помилку для набору екстремальних поз моделі, та якоїсь кількості поз між ними для цього ребра за квадратичною метрикою Гоппса.
- Порахувати середнє зважене квадратичних помилок для цього ребра з кожної пози:

$\Delta' = \frac{\sum_{i=1}^{n} k_i \Delta_i}{\sum_{i=1}^{n} k_i}$, де Δ_i - помилка вершини i, k_i - кількість входжень вершини i, n – кількість поз у вибірці.

- Застосувати операцію ecol для розрахованої помилки для ребра з найменьшою помилкою:

$w' = w_1 \frac{(v_1 - v`)}{(v_1 - v_2)} + w_2 \frac{(v_2 - v`)}{(v_1 - v_2)}$, де v_1 i v_2 - батьківські вершини, w_1 i w_2 - їх ваги, $v`$ та w' - результуюча вершина та її вага відповідно.

- Повторити до бажаної кількості полігонів

Вимоги до реалізації: функціональні

1. Імпортувати та експортувати 3d моделі у найбільш популярних форматах
2. Спрощувати високодетальні анімовані тривимірні моделі до довільної деталізації, для використання в LOD
3. У реальному часі переглядати модель, що є результатом спрощення
4. Обраховувати помилку моделі, що є результатом спрощення
5. Активно представляти користувачу всю релевантну інформацію про модель – кількість вершин, трикутників, розміри, помилку.
6. Можливість швидко протестувати результати роботи системи на придатність

Вимоги до реалізації: нефункціональні

1. Графічний інтерфейс.
2. Робота в популярних операційних системах.
3. Інтеграція з наявними редакторами трривимірних моделей

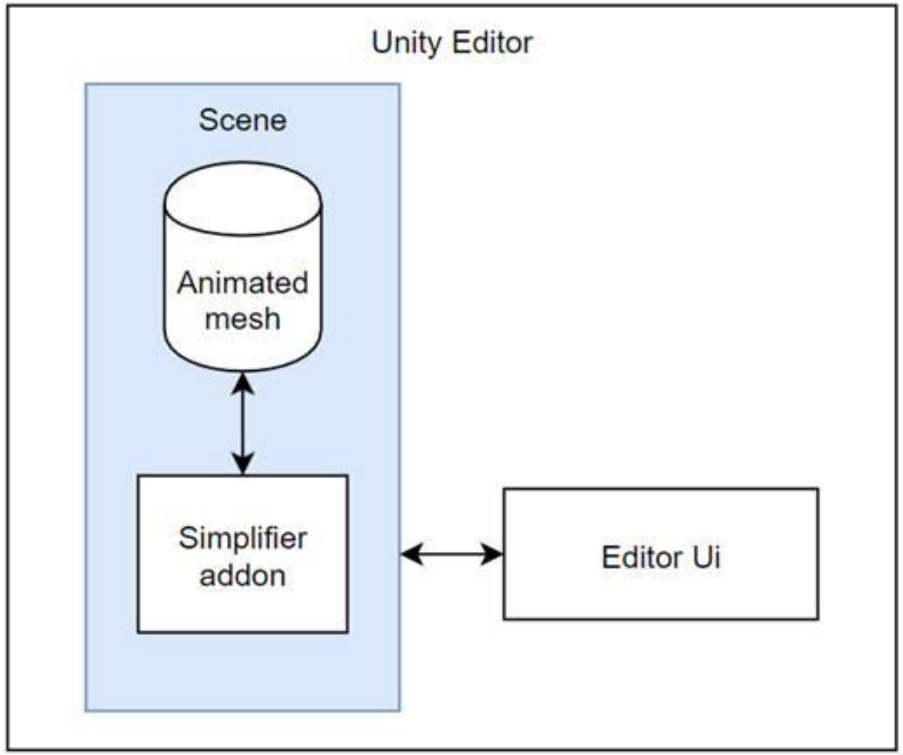
Вибір засобів розроблення: редактори тривимірних моделей

		Blender	ZBrush	Cinema 4D
Початкова вартість створення продукту на їх основі		0	$39...895	$116...3495
Відкритість екосистеми розробників розширень		+	+/-	-
Підтримка ОС	Windows	+	+	+
	MacOS	+	+	+
	Linux	+	+	+
Повнота документації для розробників		+/-	+/-	+
Імпорт поширених форматів тривимірних моделей		+	+/-	+

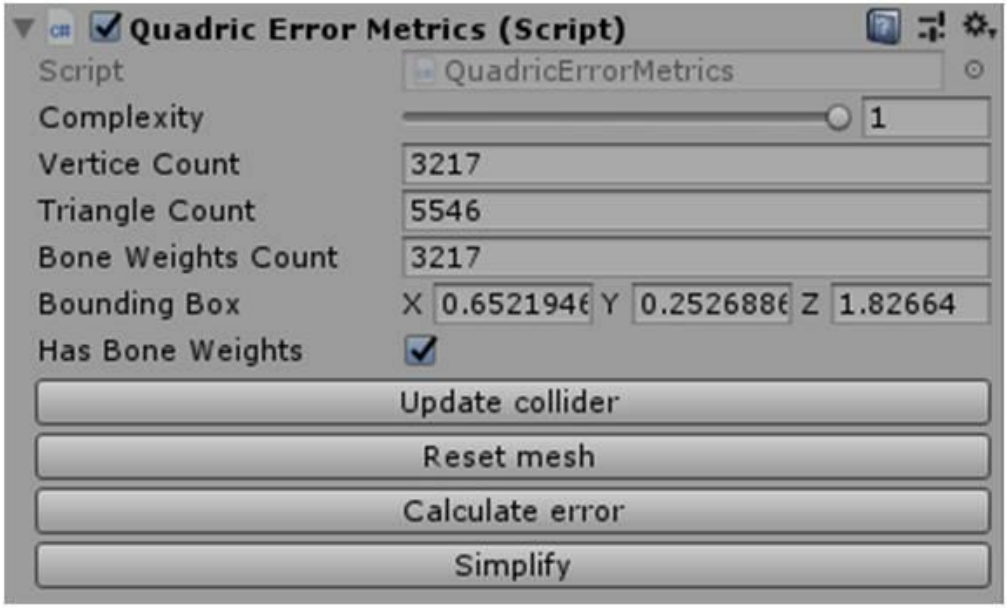
Вибір засобів розроблення: програмні рушії

		Unity	UE 4
Початкова вартість створення продукту на їх основі		0	0
Відкритість екосистеми розробників розширень		+	+/-
Підтримка ОС	Windows	+	+
	MacOS	+	+
	Linux	+	+
Повнота документації для розробників		+	-
Імпорт поширених форматів тривимірних моделей		+	-

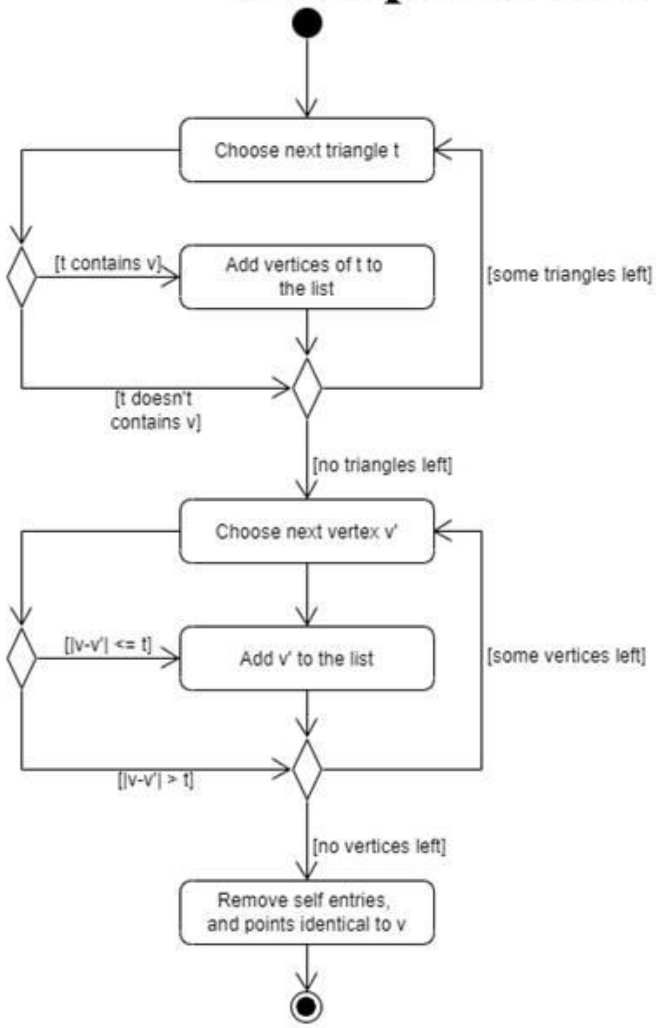
Архітектура імплементації



Інтерфейс додатку



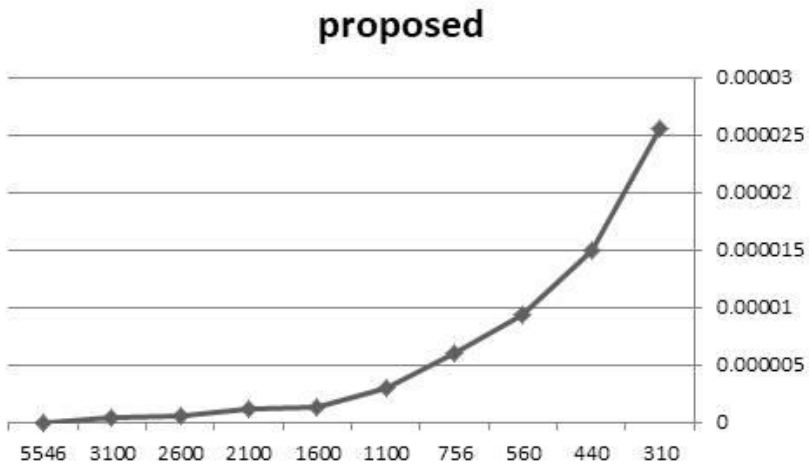
Алгоритм вибору валідних пар




```
List<int> ret = new List<int>();
for (int i = 0; i < triangles.Count; i += 3)
{
   for (int j = 0; j < 3; j++)
   {
      int index = i + j;
      if (triangles[index] == v)
      {
         ret.Add(triangles[i]);
         ret.Add(triangles[i + 1]);
         ret.Add(triangles[i + 2]);
      }
   }
}
Vector3 pos = vertices[v];
for (int i = 0; i < vertices.Count; i++)
{
   if (Vector3.Distance(vertices[i], pos) < this.t)
   {
      ret.Add(i);
   }
}
ret.RemoveAll(vertex => vertex == v);
```

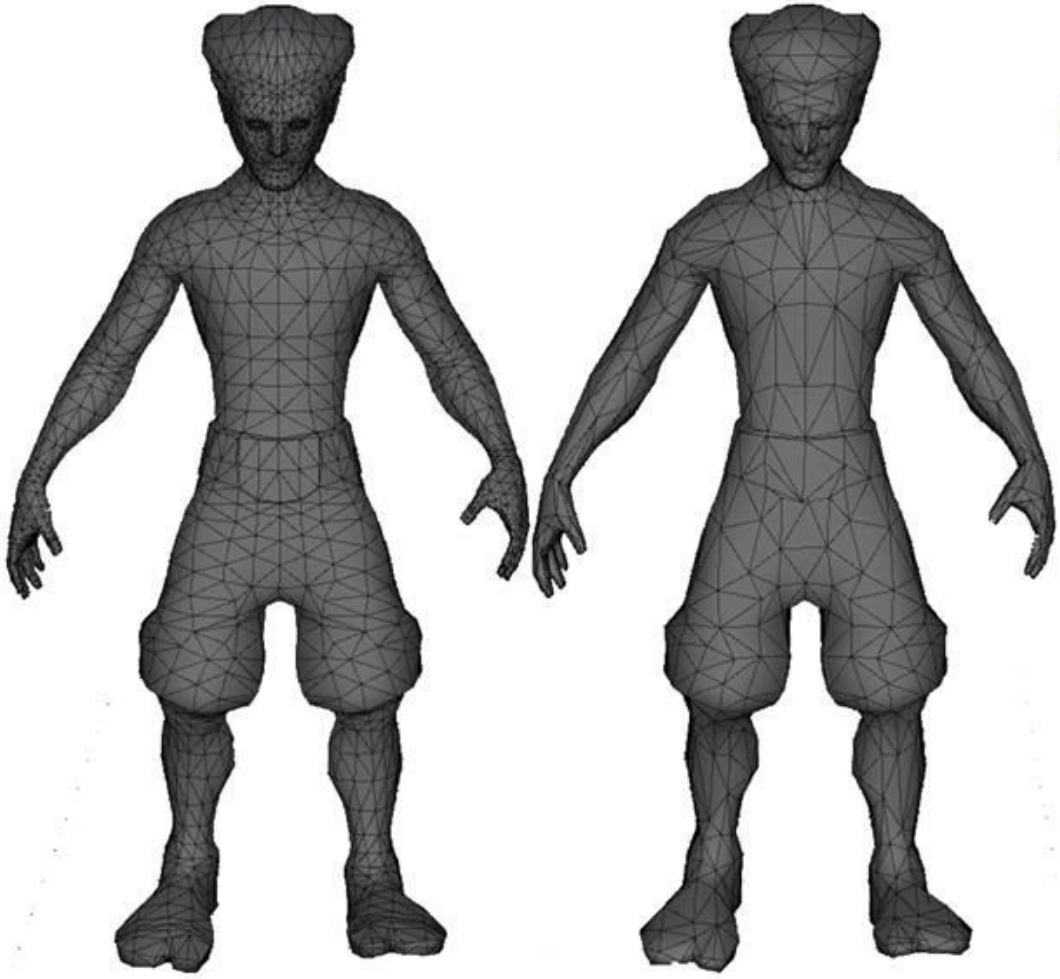
Обчислення помилки

$$E_i \frac{1}{(|X_n|+|X_i|)D}\left(\sum_{v\epsilon X_n} d^2(v, M_i) + \sum_{v\epsilon X_i} d^2(v, M_i)\right)$$

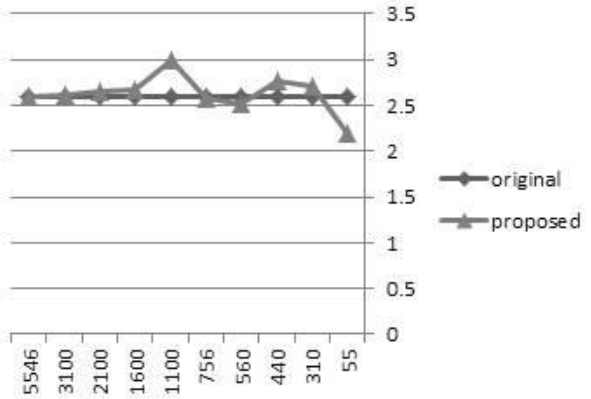



```
foreach(Vector3 vertex in vertices)
{
        float min = float.MaxValue;
        for (int i = 0; i < triangleCount; i+=3)
        {
                triangle = new
        Plane(original.vertices[original.triangles[i]],

        original.vertices[original.triangles[i+1]],

        original.vertices[original.triangles[i+2]]);
                tmp = triangle.GetDistanceToPoint(vertex);
                if (Mathf.Abs(tmp) < Mathf.Abs(min)) min = tmp;
        }
        summ += min*min;
}
foreach(Vector3 vertex in original.vertices)
{
        float min = float.MaxValue;
        for (int i = 0; i < triangleCount; i+=3)
        {
                triangle = new Plane(vertices[triangles[i]],
                                     vertices[triangles[i+1]],
                                     vertices[triangles[i+2]]);
                tmp = triangle.GetDistanceToPoint(vertex);
                if (Mathf.Abs(tmp) < Mathf.Abs(min)) min = tmp;
        }
        summ += min*min;
}
```

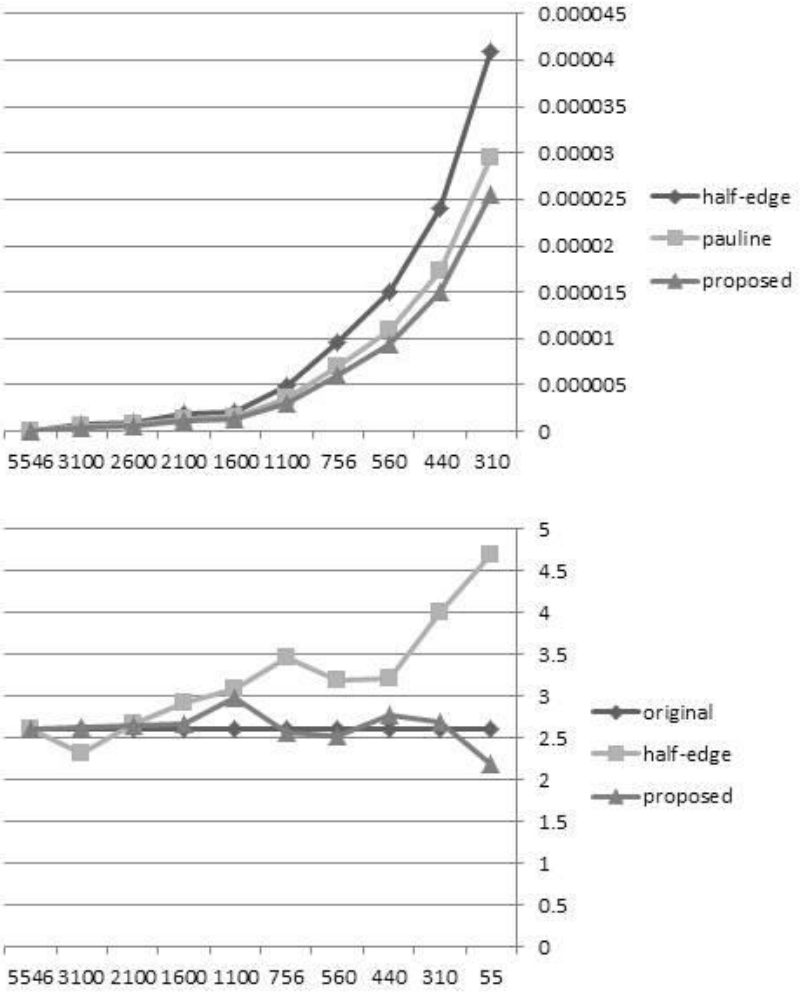
Демонстрація топології, якісна оцінка



$$\frac{\left(\frac{\max(e_{i1},e_{i2})}{\min(e_{i1},e_{i2})}\right)^2+\left(\frac{\max(e_{i1},e_{i3})}{\min(e_{i1},e_{i3})}\right)^2+\left(\frac{\max(e_{i2},e_{i3})}{\min(e_{i2},e_{i3})}\right)^2}{3}$$



Порівняння з існуючими методами

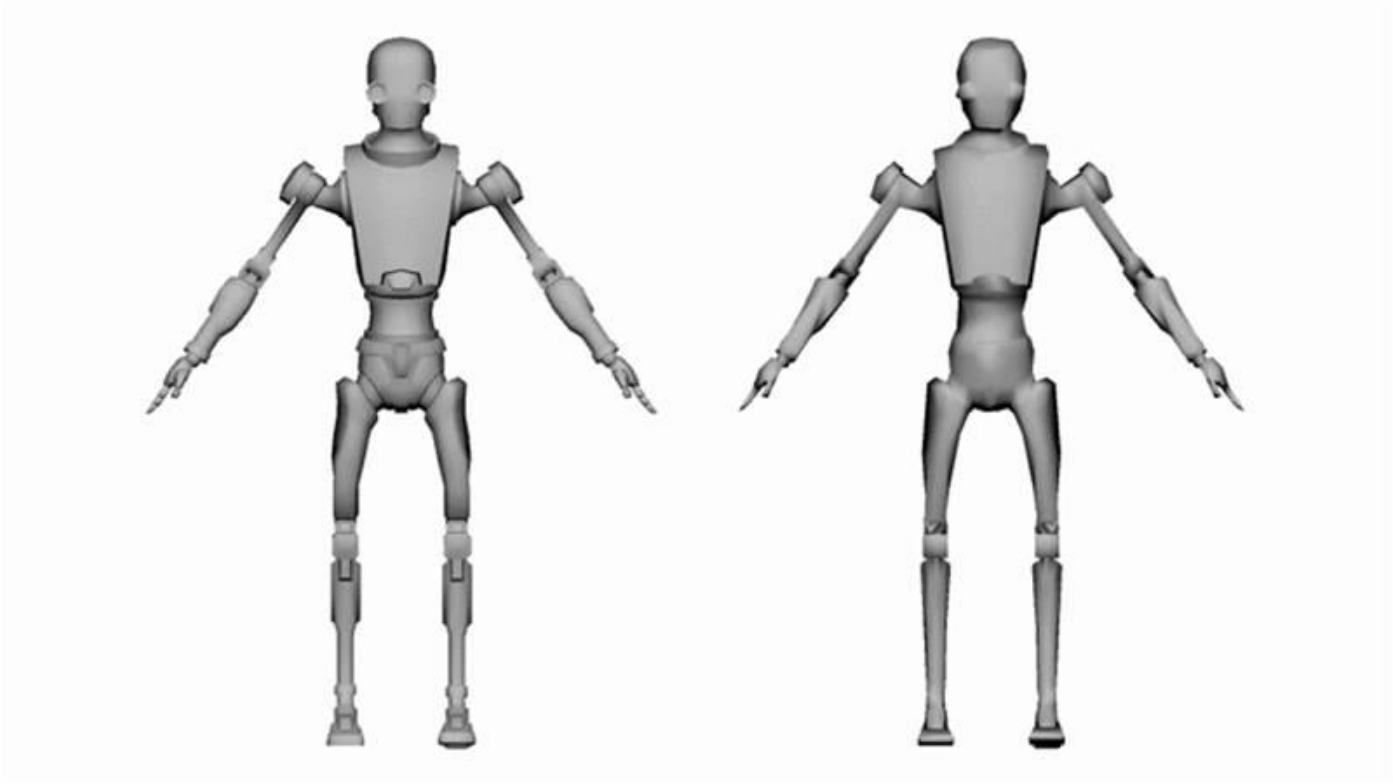


- На 8..40% менша помилка під час анімації.
- На 60% менша різниця метрики топології.

Спрощення з 6200 полігонів до 55 відбулося за:

- 0.4 с. для половинних
- 0.65 с. для Pauline
- 0.75 с. для запропонованого

Демонстрації моделі в русі (6132, 1285 полігонів)



Вимоги до апаратного забезпечення

1. Процесор – X64 архітектура з підтримкою SSE2.
2. Відеокарта – DX10+, Metal-capable, або Vulcan .
3. Обсяг оперативної пам’яті – 1024 Мб+.
4. Операційна система – Windows 7+, Sierra 10.12.6+, Ubuntu 16.04+

Наукова новизна

1. Запропоновано алгоритмічно-програмний метод, який, на відміну від існуючих методів створення LOD-моделей, дозволяє зменшити спотворення топології моделі на 65% та форми анімованої тривимірної моделі на 8-40% за рахунок перерахування ваг моделі.
2. Запропоновано модифікований критерій для вимірювання точності LOD-моделей.
3. Запропоновано критерій для вимірювання якості топології полігональної сітки

Висновки

1. В результаті аналізу існуючих методів було обрано квадратичну метрику оцінки помилки та спрощення за допомогою колапсу вершин.
2. Було розроблено метод спрощення анімованих моделей, що краще зберігає топологію моделі (на 65%) та має меньшу помилку під час анімації (8-40%).
3. Було розроблено реалізацію запропонованого методу у вигляді плагіну для рушія Unity. Процес спрощення моделі відбувається на 60..20% повільніше, проте спрощена модель має вищу якість анімації та топології сітки.

Апробування отриманих результатів

1. XII наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2019.

Дякую за увагу!