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 компенсації дефектів DQ-скінінгу»

Виконав:
студент ІІ курсу, групи КП-81мн
Руденко Костянтин Павлович _____

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 СПіСКС, к.т.н., доцент
Клятченко Ярослав Михайлович _____

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

Руденку Костянтину Павловичу

1. Тема дисертації «Алгоритмічно-програмний метод компенсації дефектів DQ-скінінгу», 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: скелетна анімація реалістичних тривимірних моделей.

4. Предмет дослідження: методи скінінгу реалистічних тривимірних моделей та дефекти скінінгу.

5. Перелік завдань, які потрібно вирішити:
   - провести аналіз існуючих методів скінінгу реалістичних тривимірних моделей;
   - дослідити дефекти скінінгу та способи боротьби з ними;
   - обрати метод скінінгу для компенсації дефектів;
   - розробити алгоритмічно-програмний метод пост-обробки моделі для компенсації дефектів скінінгу;
   - розробити реалізацію методу алгоритмічно-програмного методу пост-обробки моделі;
   - провести тестування та оптимізацію розробленої реалізації;
   - провести аналіз отриманих результатів.

6. Орієнтовний перелік публікацій:
   - Тези доповіді "Алгоритмічно-програмний метод пост-обробки моделі";
   - Стаття "Метод компенсації дефектів Dual quaternion скінінгу";
   - Стаття "A method of artifact compensation for dual quaternion skinning and its application in digital twin models".

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 Костянтин РУДЕНКО

Науковий керівник Євгенія СУЛЕМА

РЕФЕРАТ

Актуальність теми. Скелетна анімація - найбільш популярний, зручний та універсальний спосіб моделювання руху тривимірних персонажів та гнучких об’єктів. На сьогоднішній день для анімації в реальному часі використовуються методи скінінгу, що мають помітні візуальні дефекти, оскільки більш реалістичні методи скінінгу не мають достньої швидкодії, що особливо критично для мобільних платформ, а також платформ віртуальної та доповненої реальності. В даній магістерській роботі розроблено метод пост-обробки моделі після скінінгу дуальними кватерніонами, що дозволяє зменшити дефекти скінінгу за рахунок незначної втрати швидкодії.

Об’єктом дослідження є скелетна анімація реалістичних тривимірних моделей.

Предметом дослідження є методи скінінгу реалістичних тривимірних моделей та їх дефекти.

Мета роботи полягає у розробці ефективного методу компенсації дефектів скінінгу дуальними кватерніонами.

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 методу пост-обробки моделі після скінінгу з допомогою дуальних кватерніонів, що дозволяє зменшити дефект роздутого суглоба за рахунок незначного зниження швидкодії (на 8% у найгіршому випадку, згідно емпіричних вимірювань).

Практична цінність отриманих результатів роботи полягає в тому, що запропонований метод дозволяє значно покращити якість скелетної анімації без значної втрати швидкодії.

Також в рамках даного дослідження було розроблено програмну реалізацію запропонованого методу у вигляді плагіна для рушія Unity, що використовує розрахункові шейдери для підвищення швидкодії, підтримує роботу з цільовими формами, встановлення коефіцієнта компенсації як для моделі в цілому, так і для окремих вертексів, має швидкодію, близьку до вбудованого скінінгу (при використанні компілятора IL2CPP), автоматично попереджає розробника про розповсюджені помилки налаштування та сумісний з графічними API DirectX, OpenGL, Vulkan та Metal.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XI науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2019 та опубліковані у збірнику тез доповідей.

За результатами роботи була написана наукова стаття на тему «A method of artifact compensation for dual quaternion skinning» до наукового міжнародного журналу «Вісник Хмельницького національного університету» 2020 №1.

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 описано сучасні методи скінінгу, вимоги та критерії вибору методу скінінгу в залежності від області використання, їх переваги, особливості та недоліки, а також розглянуто відомі способи боротьби з цими недоліками.

У другому розділі надано теоретичне підґрутня для розробки методу компенсації дефектів скінінгу дуальними кватерніонами та описано запропоновану модель небажаних деформацій.

У третьому розділі проведено аналіз доступних засобів імплементації запропонованого методу компенсації дефектів скінінгу дуальними кватерніонами та надано опис розробленої імплементації запропонованого методу.

У четвертому розділі описано недоліки запропонованого методу та шляхи їх усунення, а також проведено аналіз швидкодії та оптимізацію розробленої імплементації.

У висновках проаналізовано отримані результати роботи.

Робота виконана на 73 аркушах, містить посилання на список використаних літературних джерел з 23 найменувань. У роботі наведено 68 рисунків.

Ключові слова: скелетна анімація, скінінг, дуальні кватерніони.

ABSTRACT

Actuality. Skeletal animation is the most popular, convenient and universal method of modeling the motion of 3-dimentional characters and flexible objects. The methods of skinning, that are nowadays being used for real-time skeletal animations have noticeable artifacts, as the more advanced skinning methods perform significantly slower, which is especially critical for mobile platforms, virtual reality and augmented reality platforms. This master's thesis contains a method of post-processing a model, animated with dual quaternion skinning, that significantly reduces visible artifacts of the animation with only a small performance drop.

Object of research is skeletal animation of realistic 3-dimensional.

Subjects of research are the methods of skinning 3-dimensional models and their artifacts.

Goal of the work is to develop an effective method of compensating skinning artifacts in 3-dimensional skeletal animations.

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 lies in the development of a method of post-processing a 3-dimensional model, animated using dual quaternion skinning, that decreases joint-bulging artifact at the cost of minimal performance decrease (according to empirical measurements, 8% in worst-case scenario).

Practical value of the results of work is that the proposed method significantly improves the quality skeletal animation with minimal performance impact.

Also a software implementation of the proposed was developed in a form of a plugin for Unity engine, that performs calculations in compute shaders for increased performance, supports blend shapes, allows setting the compensation coefficient both for the model as a whole and for separate vertices, displays performance speed comparable to that of built-in skinning (as long as IL2CPP

compiler is used), automatically detects and fixes common setup errors and is compatible with API DirectX, OpenGL, Vulkan and Metal.

Approbation. The main provisions and results of the work were presented and discussed at the XI scientific conference of masters and postgraduates "Applied Mathematics and Computing" PMK-2019 and published in the proceedings.

As a result of the work, a scientific article on the topic "A method of artifact compensation for dual quaternion skinning" was written to the international scientific journal "Herald of Khmelnytskyi national university" 2020 № 1.

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, an overview of the current state of the problem, outlines the relevance and goals of the study.

The first chapter describes existing methods of skinning, the choosing criteria and requirements for skinning methods based on the area of application, outlines their features, advantages and disadvantages and describes the known methods of dealing with these disadvantages.

The second chapter provides a theoretical basis for the development of an artifact compensation method for dual quaternion skinning and describes a simplified mathematical model of undesired deformations.

The third section contains an analysis of available means of implementing the proposed method of artifact compensation for dual quaternion skinning and the details of its implementation.

The fourth chapter describes the shortcomings of the developed method and the ways of mitigating them, as well as provides an analysis of the performance of the developed implementation and its optimization.

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

The work is done on 73 pages and contains a reference list of 23 titles. The work contains 68 pictures.

Keywords: skeletal animation, skinning, dual quaternions.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Скелетна анімація – це техніка комп'ютерної анімації, що використовує ієрархічну структуру об'єктів, що називається арматурою, або скелетом, для керування рухами моделі.

Скінінг – це процес визначення положень вертексів моделі в просторі в залежності від положення кісток арматури, до якої модель прив'язана.

Кватерніон – це гіперкомплексне число типу $a + bi + cj + dk$, де a, b, c, d – дійсні числа, i, j, k – уявні одиниці. Кватерніони, зокрема, можуть використовуватися для опису поворотів в тривимірному просторі.

Дуальні кватерніони – це гіперкомплексні числа типу $q_r + q_d\varepsilon$, де q_r, q_d – кватерніони, ε – число, таке, що $\varepsilon^2 = 0$, але $\varepsilon \neq 0$. Дуальні кватерніони можуть, зокрема, використовуватися для позначення комбінації зсуву та повороту в тривимірному просторі.

Воксель – елемент простору, позначає значення певної величини в клітинці рівномірної просторової ґратки. Аналог пікселю в двовимірних зображеннях.

ВСТУП

На сьогоднішній день використовується два методи анімації полігональних моделей: метод інтерполяції ключових форм та скелетна анімація.

Метод інтерполяції ключових форм полягає в інтерполяції положень вертексів та напрямів нормалей полігональної моделі між набором завчасно підготовлених станів. Цей метод не дозволяє відтворювати рухи, що включають в себе повороти, але дозволяє плавно змінювати форму моделі, не втручаючись в роботу алгоритмів скелетної анімації. Приклад інтерполяції ключових форм зображено на рис. 1.

Рис. 1. Інтерполяція ключових форм [1]

Метод скелетної анімації дозволяє керувати рухом об’єкта за допомогою допоміжної структури, що називається скелетом, або арматурою. Скелет являє собою ієрархію кісток, кожна з яких має положення в просторі, поворот та масштаб. При використанні скелетної анімації, аніматор керує

рухами кісток скелета, а модель деформується, повторюючи рухи скелета. Приклад анімації кисті руки за допомогою скелета зображено на рис. 2. Процес деформації моделі для відтворення рухів скелету називається скінінгом.

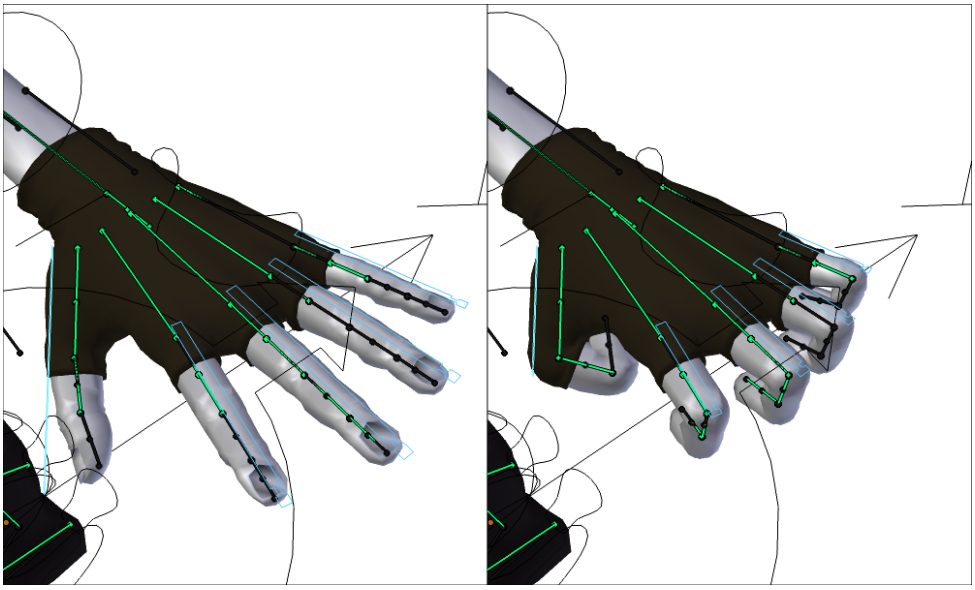


Рис. 2. Скелетна анімація руки [2]

Для виконання скінінгу кожен вертекс моделі має зберігати індекси кісток скелету, що на нього впливають, та коефіцієнти впливу, що називаються вагами. Для збереження об’єму моделі та попередження дефектів сума коефіцієнтів впливу кісток для кожного вертекса має дорівнювати одиниці. Вплив ваг кісток на деформацію моделі продемонстровано на рис. 3 (ваги кісток зображено червоним та синім кольорами). Ваги кісток можуть задаватися як вручну аніматором, так і автоматично.

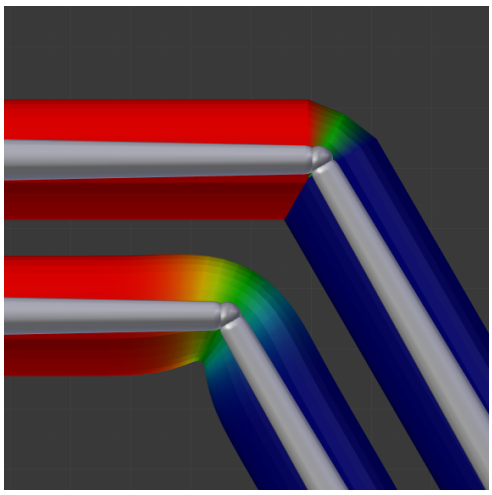


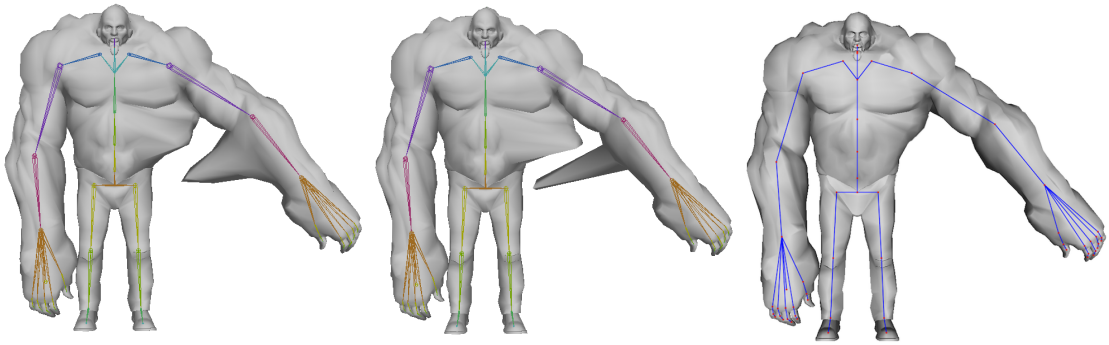
Рис. 3. Вплив ваг кісток на деформацію моделі

1. АНАЛІЗ ІСНУЮЧИХ АЛГОРИТМІЧНО-ПРОГРАМНИХ МЕТОДІВ СКІНІНГУ

1.1. Підготовка моделі до скінінгу

Автоматичне визначення ваг кісток для скелетної анімації може виконуватися на основі відстаней кісток до вертексів (ближча до вертекса кістка має більший вплив), сегментації моделі за зонами впливу кісток з подальшим згладжуванням переходів між зонами впливу, або шляхом вокселізації моделі та визначення геодезичних відстаней вертексів до кісток.

При визначенні ваг кісток за відстанню до вертексів можуть з’являтися небажані ваги, якщо різні кінцівки скелета знаходяться достатньо близько одна до одної. Приклад такого дефекту наведено на рис. 4 (а).



(а) (б) (в)

Рис. 4. Автоматична генерація ваг кісток на основі відстані до вертексів (а), на основі відстані до вертексів з урахуванням ієрархії (б) та на основі сегментації моделі (в) [3]

Частково позбавитись від небажаних ваг кісток можна шляхом використання ієрархії кісток скелета для визначення сусідніх кісток до тієї, що має найбільший вплив на даний вертекс. Це дозволяє попередити ситуацію, коли на один вертекс одночасно впливають декілька різних

кінцівок скелета. Проте у випадку, коли найближчою до вертекса є кістка, що не має на нього впливати, даний підхід також залишає дефект. Результат використання ієрархії кісток для зменшення дефектів автоматично визначених ваг наведено на рис. 4 (б).

Повністю позбавитись від небажаних ваг кісток дозволяє метод сегментації моделі з подальшою дифузією ваг [3]. Сегментація моделі виконується шляхом знаходження площин, що поділяють модель на окремі зони впливу. Для кожної пари з'єднаних між собою кісток J_{lower}, J_{upper} визначається площина Q така, що обидві кістки лежать в цій площині (рис. 5).

Після знаходження площини Q обирається випадковий одиничний вектор n_0, що виходить з суглоба J та лежить у площині Q. Повертаючи вектор n_0 навколо точки J в площині Q на фіксований кут до повного обороту, отримуємо набір потенційних нормалей n_i. Кожна нормаль n_i задає площину P_i. Для кожної площини P_i необхідно визначити її перетин з поверхнею моделі, що має являти собою замкнений контур (червоний пунктир на рис. 5). Площина P_i з найменшою довжиною такого контура обирається для розмежування сегментів впливу кісток. Приклад такого сегментування наведено на рис. 6.

Після виконання сегментування моделі всім вертексам у сегменті присвоюється максимальна вага відповідної цьому сегменту кістки. Для досягнення плавних деформацій виконується дифузія ваг через сусідні вертекси.

Визначення ваг кісток на основі сегментації моделі дозволяє попередити виникнення небажаних ваг, проте потребує моделі із замкнутою поверхнею для виконання дифузії ваг. Оскільки моделі з незамкненою поверхнею використовуються доволі часто, непридатність цього методу для роботи з моделями з незамкненою поверхнею є суттєвим недоліком.

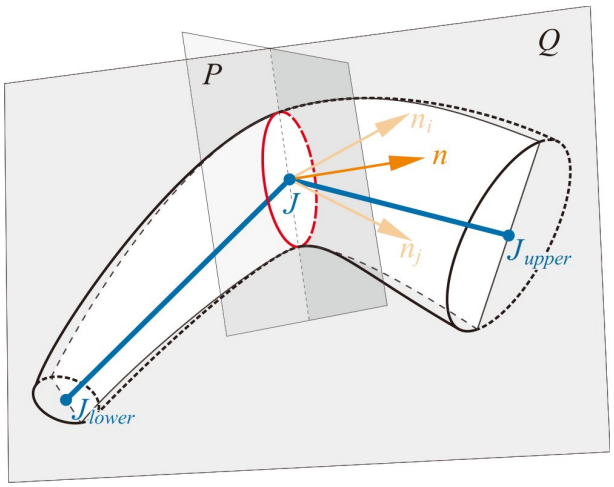


Рис. 5. Знаходження площини, що поділяє зони впливу кісток [4]

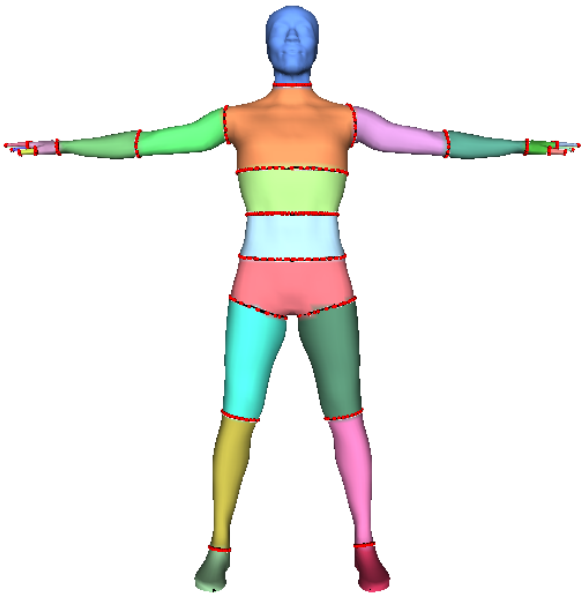


Рис. 6. Сегментування моделі [4]

Цього недоліку можливо позбавитись шляхом використання вокселізації моделі. Для вокселізації моделі замкнена поверхня не є обов'язковою, а дифузія ваг може бути виконана для набору суміжних вокселів за тим же принципом, що і для набору суміжних вертексів. Приклад використання вокселізації моделі для визначення ваг кісток

зображено на рис. 7.

Крім сегментації, для визначення ваг кісток вокселізованої моделі може бути використана геодезична відстань від вокселя, що містить даний вертекс, до кістки.

Для знаходження геодезичної відстані від вертексів до кістки, використовується модифікований алгоритм Дейкстри. При цьому на початку алгоритму вокселі, що перетинаються кісткою, додаються до робочої черги, а їх відстань встановлюється рівною нулю. На кожній ітерації алгоритму для кожного вокселя з робочої черги перевіряється значення відстані всіх сусідніх вокселів. Якщо значення відстані сусіднього вокселя більше, ніж значення відстані поточного вокселя плюс відстань між цими вокселями, значення відстані сусіднього вокселя оновлюється і він додається до робочої черги. Після огляду всіх сусідніх до поточного вокселів, поточний воксель видаляється з робочої черги. Робота алгоритму закінчується, коли в робочій черзі не залишається вокселів.

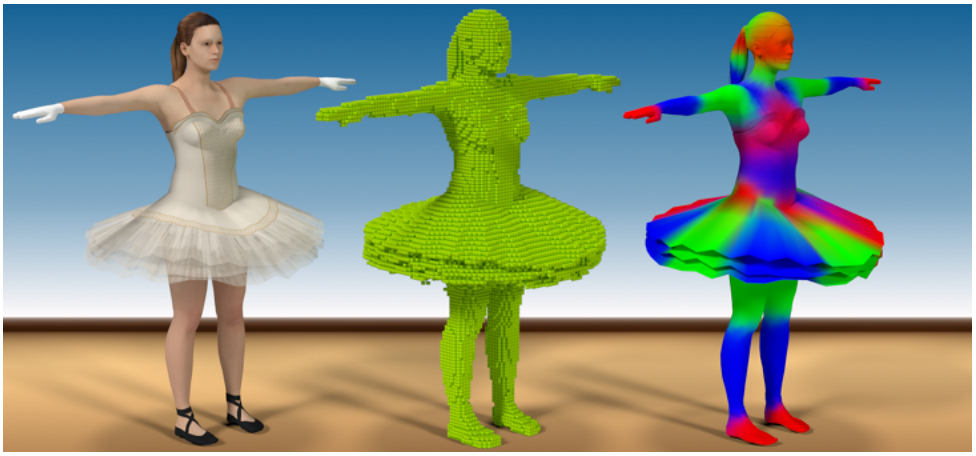


Рис. 7. Визначення ваг кісток моделі з допомогою вокселізації [5]

Після завершення роботи алгоритму, геодезична від відстань вертекса до кістки вважається рівною значенню відстані вокселя, що містить цей

вертекс. На рис. 8 зображено визначення відстані від кістки до вертекса. Білим кольором позначено вокселі, що перетинаються кісткою, червоним – вокселі, що знаходяться всередині моделі, блакитним – вокселі, що містять вертекси.

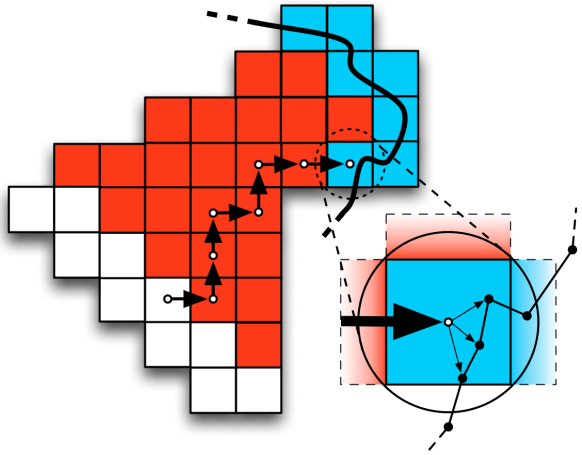


Рис. 8. Геодезична відстань від вертекса до кістки [3]

Розрахунки геодезичних відстаней до кісток можуть виконуватися паралельно на різних ядрах процесора, проте алгоритми скінінгу з використанням вокселізації все одно залишаються більш вимогливими до розрахункових ресурсів. Складність алгоритму пошуку геодезичних відстаней обернено пропорційна кубу довжини ребра вокселя, а при використанні занадто великих вокселів, розрахунки геодезичної відстані можуть мати дефекти через пропущені дрібні деталі моделі (рис. 9).

Окремим способом визначення початкових вагових коефіцієнтів кісток, що дозволяє аніматору частково керувати процесом, є використання оболонок кісток. При використанні оболонок, кожен суглоб представляється у вигляді сфери певного радіусу (радіус кожного суглоба визначається аніматором), а кістки представляються у вигляді зрізаних конусів, що поєднують сфери-суглоби (рис. 10).

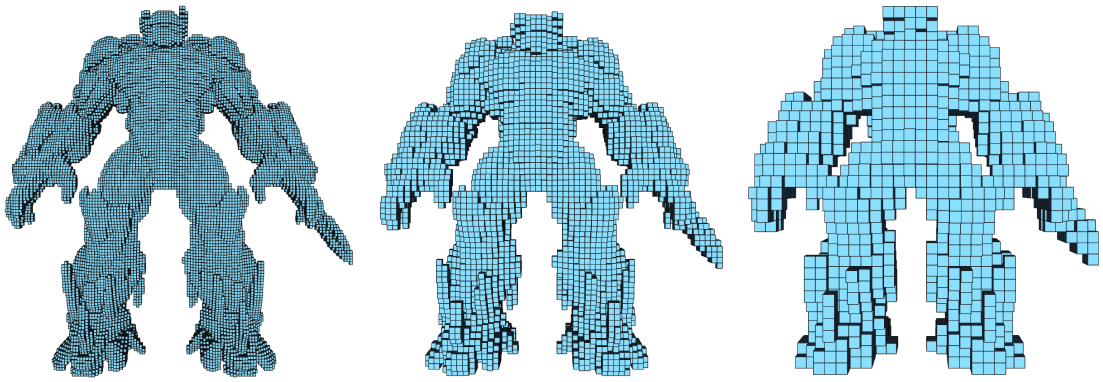


Рис. 9. Вокселізація моделі з різними розмірами вокселя [3]

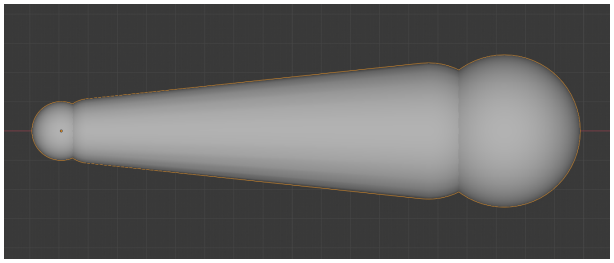


Рис. 10. Представлення кістки в пакеті програмного забезпечення Blender при використанні оболонок для визначення початкових вагових коефіцієнтів

Вертекси, що потрапляють у внутрішній об’єм кістки, мають коефіцієнт для цієї кістки, рівний одиниці. З віддаленням від поверхні кістки, її ваговий коефіцієнт спадає пропорційно квадрату відстані до поверхні кістки. Відстань під поверхні кістки, з досягненням якої її ваговий коефіцієнт спадає до нуля, називається товщиною оболонки та задається аніматором разом з діаметром суглоба (рис. 11).

Недоліком визначення початкових вагових коефіцієнтів кісток за допомогою оболонок є форма вигину суглобів, спричинена квадратичною функцією визначення вагового коефіцієнту на основі відстані до

кістки (рис. 12).

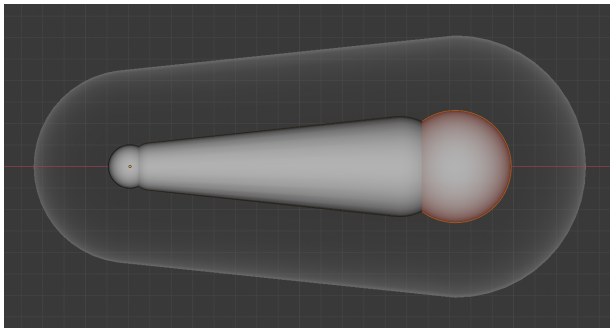


Рис. 11. Представлення оболонки кістки в пакеті програмного забезпечення Blender

Використання кількох ітерацій згладжування градієнту вагових коефіцієнтів дозволяє частково покращити форму зігнутого суглоба (рис. 13), але більш дієвим рішенням було б використання іншою функції для визначення вагового коефіцієнту на основі відстані до кістки.

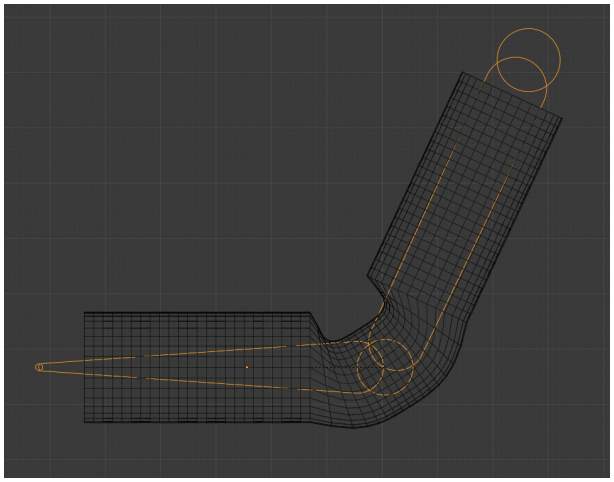


Рис. 12. Деформація згину суглоба при використанні оболонок кісток для визначення початкових вагових коефіцієнтів

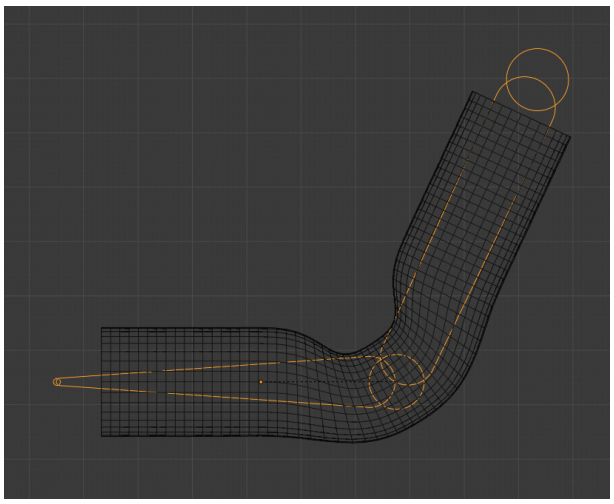


Рис. 13. Суглоб після згладжування градієнту вагових коефіцієнтів, отриманого за допомогою оболонок кісток

1.2. Вимоги до алгоритмічно-програмних методів скінінгу

Основні вимоги, що висуваються до методів скінінгу, включають в себе:

- високу швидкодію;
- реалістичність анімації;
- простоту налаштування;
- сумісність з сучасними інструментами створення анімації.

Залежно від мети застосування, ці вимоги можуть бути неважливими, бажаними, або необхідними. Так, для створення відео-контенту висока швидкодія не є критичною, в той час для створення інтерактивних анімацій без адекватної швидкодії неможливе. Внаслідок різноманіття застосувань та вимог існує цілий ряд методів скінінгу, кожен з яких спеціалізується на певному наборі бажаних властивостей.

Метод лінійного скінінгу та метод дуальних кватерніонів мають найвищу швидкодію та не потребують додаткового налаштування окрім задання ваг кісток, проте мають суттєві дефекти анімації, що детально описані пізніше в цьому розділі.

Метод лінійного скінінгу з оптимізованими центрами обертання дозволяють позбавитися від дефектів без значної втрати швидкодії, але потребує тривалої процедури визначення оптимальних центрів обертання, що змінюються при змінах коефіцієнтів ваги. Таким чином перегляд у реальному часі змін деформації моделі при редагуванні ваг не є можливим.

Методи на основі прикладу дозволяють позбавитися від дефектів без значної втрати швидкодії та відтворювати додаткові ефекти деформації, такі як скорочення м'язів та складки шкіри, проте для задання параметрів такого методу аніматор має надати приклади деформації моделі у різних позах, що значно ускладнює процес створення анімацій.

Неявний скінінг відтворює максимально реалістичну деформацію, але значно поступається всім попереднім методам за швидкодією.

1.3. Лінійний скінінг

При використанні лінійного скінінгу визначення положення вертекса деформованої моделі виконується за формулою (1):

$$v'_i = \sum_{j=1}^{m} w_{ij} T_j R_j^{-1} v_i, \tag{1}$$

де v'_i – координати і-го вертекса деформованої моделі; m – кількість кісток; w_{ij} – вага j-ї кістки для i-го вертекса; T_j – матриця переходу для поточного положення j-ї кістки; R_j – матриця переходу для початкового положення j-ї кістки (поза відпочинку); v_i – початкове положення i-го вертекса.

Лінійна інтерполяція матриць перетворення створює деформації, непридатні для використання. Замість цього в лінійному скінінгу інтерполюється положення вертекса вже після множення на матрицю перетворення (рис. 14). Така апроксимація добре працює для незначних рухів, але створює дефекти у більш екстремальних позах (рис. 15). Найбільш помітним цей дефект стає при скручуванні моделі, що робить її схожою на

обгортку цукерки (рис. 16). Частково цей дефект може бути компенсований за шляхом створення додаткових кісток, проте це значно ускладнює роботу аніматора.

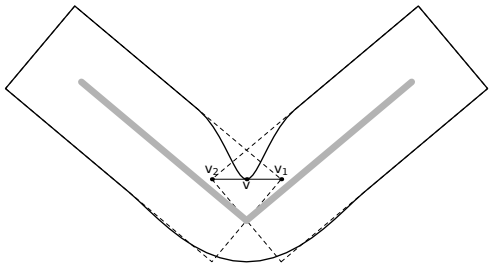


Рис. 14. Інтерполяція положення вертекса (лінійний скінінг) [14]

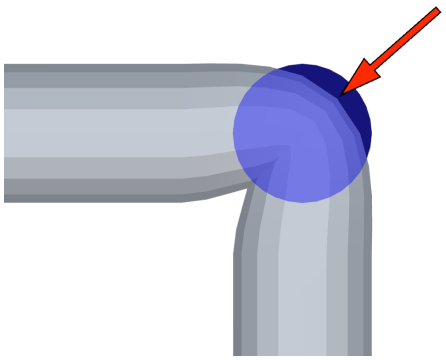


Рис. 15. Дефект лінійного скінінгу – втрата об’єму

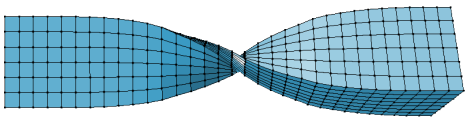


Рис. 16. Дефект лінійного скінінгу – ”обгортка цукерки” [6]

1.4. Скінінг дуальними кватерніонами

Метод скінінгу дуальними кватерніонами полягає у використанні дуальних кватерніонів для представлення положення кісток замість матриць перетворення [7–10]. На відміну від матриць перетворення, при інтерполяція дуальних кватерніонів результатом є плавне переміщення між заданими положеннями, при якому об'єкт обертається навколо осі, вздовж якої рухається. При цьому таке перетворення гарантовано зберігає об'єм моделі, що є основною проблемою лінійного скінінгу (рис. 17). Проте замість дефекту втрати об'єму скінінг дуальними кватерніонами створює новий дефект – "роздування" суглобів (рис. 18).

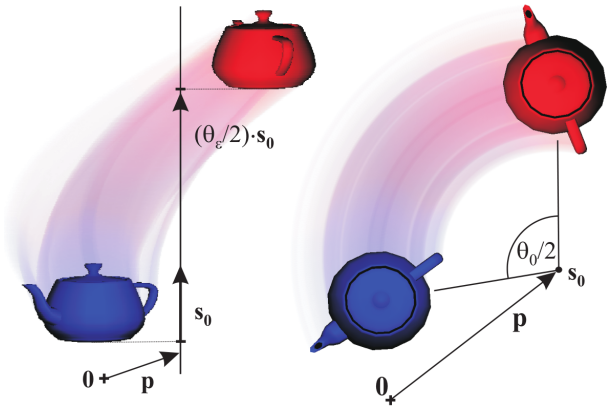


Рис. 17. Інтерполяція руху з допомогою дуальних кватерніонів [8]

Для запобіганню цьому дефекту, дослідники Kim YoungBeom та Han JungHyun пропонують підтримувати постійну відстань від вертекса моделі до найближчої кістки [11] (рис. 19). Проте такий метод призводить до створення розтягнутих полігонів в зонах, те сусідні вертекси зміщуються до різних кісток.

Варто зазначити, що дуальні кватерніони не передають масштабування, тому, для підтримки анімації з масштабуванням кісток, масштабування має бути реалізовано окремо [12].

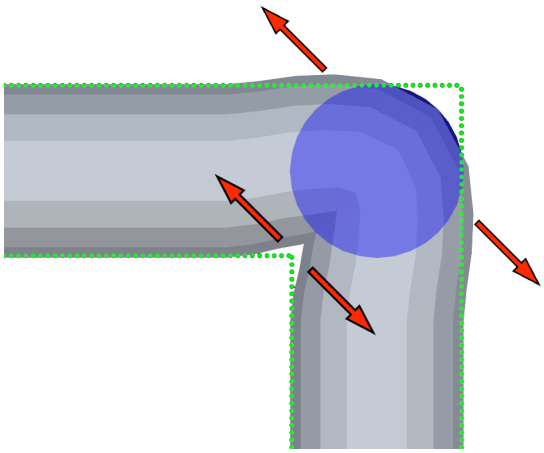


Рис. 18. Дефект скінінгу з допомогою дуальних кватерніонів – ”роздутий” суглоб

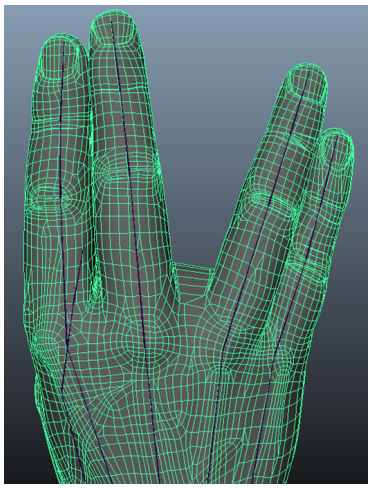


Рис. 19. Запобігання роздуванню суглоба шляхом підтримання ре-проектування вертексу до найближчої кістки [11]

1.5. Лінійний скінінг з оптимізованими центрами обертання

Лінійний скінінг з оптимізованими центрами обертання [13] дозволяє позбавитись від дефектів втрати об’єму завдяки використанню зміщених центрів обертання, що задаються окремо для кожного вертекса. При

цьому центри обертання задаються так, щоб вертекси з однаковими вагами виконували однаковий зсув. Це гарантує збереження об'єму моделі при деформаціях. Недоліком цього методу є необхідність виконувати ресурсовитратну процедуру пошуку оптимальних центрів обертання щоразу після зміні коефіцієнтів ваги кісток.

1.6. Скінінг в просторі анімації

Скінінг в просторі анімації [14, 15] є більш узагальненою формою лінійного скінінгу, що використовує окремі ваги для кожного елемента матриці перетворення. Визначення положення вертекса деформованої моделі відбувається за формулою (2).

$$v'_i = \sum_{j=1}^{m} \begin{pmatrix} w_{j,11}G_{j,11} & w_{j,12}G_{j,12} & w_{j,13}G_{j,13} & w_{j,14}G_{j,14} \\ w_{j,21}G_{j,21} & w_{j,22}G_{j,22} & w_{j,23}G_{j,23} & w_{j,24}G_{j,24} \\ w_{j,31}G_{j,31} & w_{j,32}G_{j,32} & w_{j,33}G_{j,33} & w_{j,34}G_{j,34} \\ 0 & 0 & 0 & 1 \end{pmatrix} v_i, \qquad (2)$$

де v'_i – координати i-го вертекса деформованої моделі; m – кількість кісток; w_{ij} – вага j-ї кістки для i-го вертекса; G_j – добуток матриць T_j та R_j з формули (1); v_i – початкове положення i-го вертекса.

Завдяки додатковим коефіцієнтам ваги положення вертекса деформованої моделі не обмежено відрізком між крайніми точками, проте для задання цих коефіцієнтів аніматор має надати приклади деформованої моделі у наборі різноманітних поз.

1.7. Скінінг за допомогою неявно заданих поверхонь

Скінінг за допомогою неявно заданих поверхонь [16] є більш ресурсовитратним методом за всі вищезазначені, але дозволяє досягти високої якості анімації без необхідності виконання додаткової роботи аніматором (такої, як створення набору прикладів деформованої моделі у

різних позах).

Для виконання скінігу за допомогою неявно заданих поверхонь, модель поділяється на набір замкнутих неявно заданих поверхонь, кожна з яких відповідає частині моделі, що прив’язана до однієї з кісток (рис. 20).

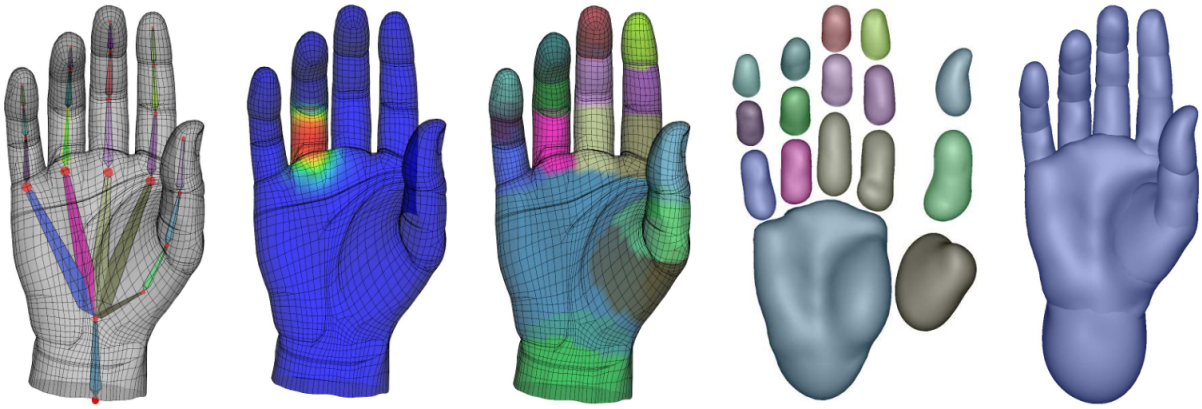


Рис. 20. Поділ моделі на набір неявно заданих поверхонь при використанні скінінгу за допомогою неявних поверхонь [16]

Неявно задані поверхні, що прив’язані до кісток, рухаються разом з ними без деформації. В процесі рендерингу кадру, неявно задані поверхні об’єднуються у одну, а вертекси моделі проектуються на отриману неявно задану поверхню (рис. 21).

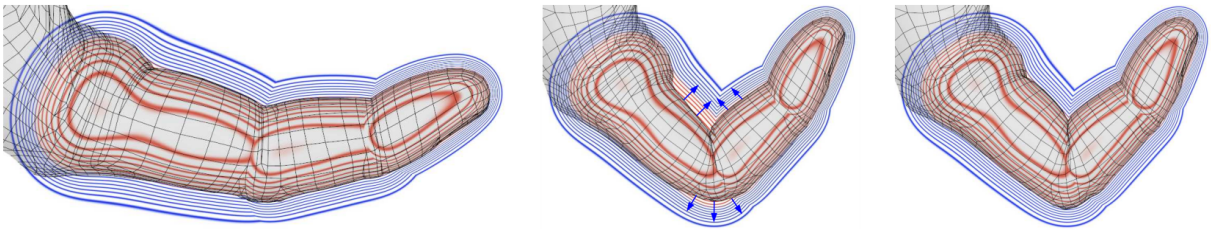
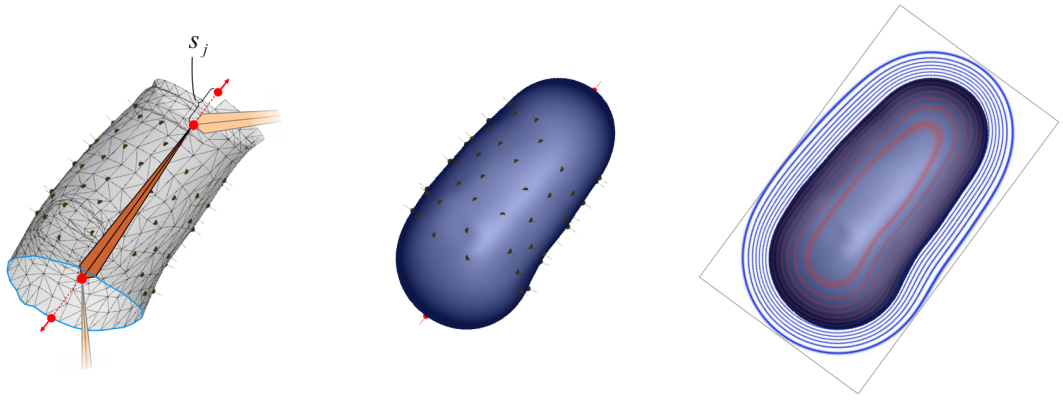


Рис. 21. Анімація за допомогою неявно заданих поверхонь [16]

Для генерації неявно заданої поверхні використовується радіальна

базисна функція Ерміта (рис. 22). Для генерації функції, що задає неявну поверхню, вузлові точки мають бути розташовані з приблизно рівномірною частотою, оскільки занадто висока частота частота вузлових точок може призвести до небажаних дрібних складок, згинів та дефектів, а недостатня частота вузлових точок може призвести до неточного передання результуючою функцією форми моделі.

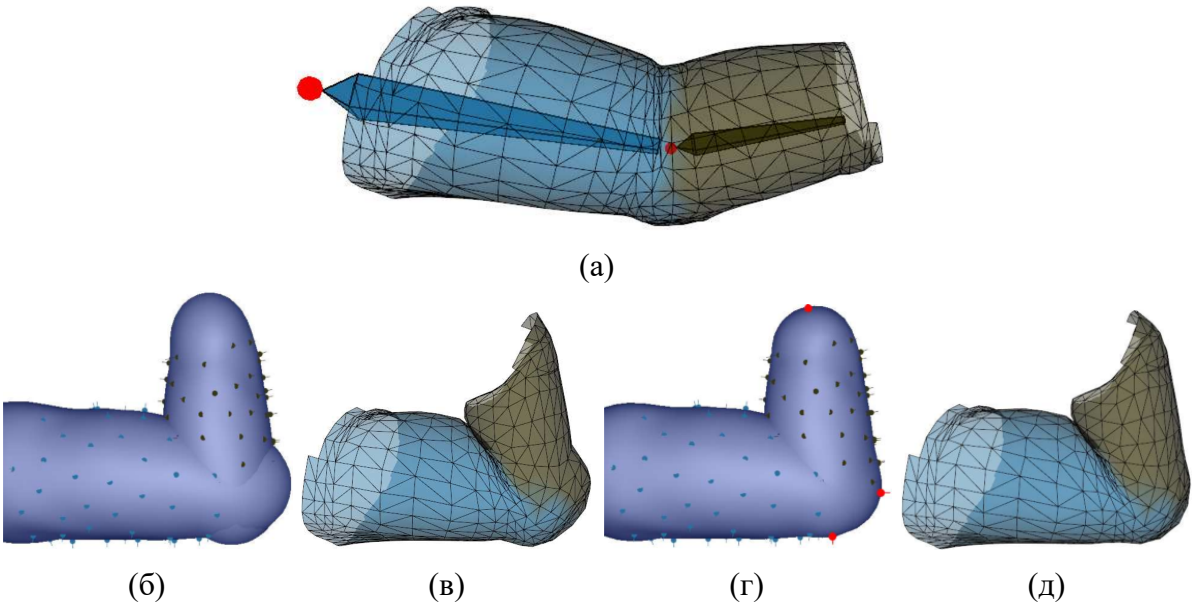
Для генерації набору вузлових точок рекомендується використовувати Poisson disk sampling [17], що дозволяє згенерувати псевдовипадковий набір точок, при цьому гарантуючи що відстань між будь-якою парою точок буде не менше заданого значення. Для генерації замкнутої поверхні до набору вузлових точок додаються точки, що лежать на одній лінії з кістками (відмічені червоним на рис. 22).



(а) (б) (в)

Рис. 22. Частина моделі (а), вузли для генерації неявно заданої поверхні (б), неявно задана поверхня (в) [16]

Без цих додаткових точок неявно задана поверхня все одно не матиме отворів, але відповідні зони матимуть дефекти інтерполяції через недостатню частоту вузлових точок для повноцінного опису форми моделі (рис. 23).



(а)

(б) (в) (г) (д)

Рис. 23. Частина моделі (а), неявно задана поверхня без використання додаткових вузлових точок (б), результуюча анімація без використання додаткових вузлових точок (в), неявно задана поверхня з використанням додаткових вузлових точок (г), результуюча анімація з використанням додаткових вузлових точок (д) [16]

1.8. Корегуючі ключові форми

Використання анімації за допомогою ключових форм у комбінації з методами скелетної анімації дозволяє аніматору більш детально контролювати деформацію моделі [18], але потребує значно більше часу та зусиль. Анімація ключовими формами дозволяє відтворювати рух м’язів, еластичність шкіри, зморшки, складки тканини та інші ефекти, які неможливо відтворити за допомогою методів скелетної анімації (рис. 24, 25).

Для використання анімації ключових форм у комбінації з методами скелетної анімації, ваги ключових форм можуть визначатися як вручну аніматором, так і визначатися автоматично на основі поточної пози моделі. Так, вага ключової форми, що відповідає скороченню біцепса, може бути прив’язана до кута згину руки у ліктьовому суглобі. У більш складних

випадках вага ключової форми може залежати одночасно від кількох кісток та проявлятися одночасно у кількох різних екстремальних позах. В таких випадках постає проблема вибору методу інтерполяції та оцінки близькості поточної пози до обраних ключових. [18]

Крім додаткових ефектів, корегуючі ключові форми можуть використовуватися для усунення дефектів, спричинених методами скелетної анімації, таких, як дефект обгортки цукерки та дефект втрати об’єму у випадку використання лінійного скінінгу, та дефект роздутого суглоба у випадку використання скінінгу дуальними кватерніонами. Для усунення дефекту аніматор обирає позу моделі, в якій дефект є максимально вираженим, та створює нову корегуючу ключову форму, що деформує модель, усуваючи цей дефект. При наближенні поточної пози моделі до обраної ключової пози, ваговий коефіцієнт корегуючої ключової форми наближується до одиниці, і, навпаки, спадає при віддаленні. Таким чином корегуюча ключова форма, що усуває дефект, активується одночасно з анімаціями, що цей дефект створюють (рис. 26).

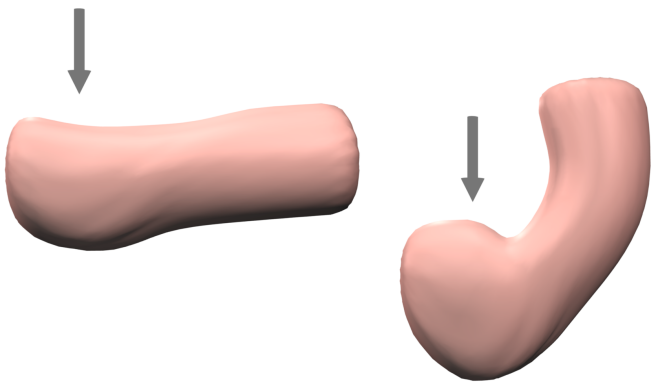


Рис. 24. Анімація скорочення м’яза при згинанні кінцівки за допомогою ключових форм [18]

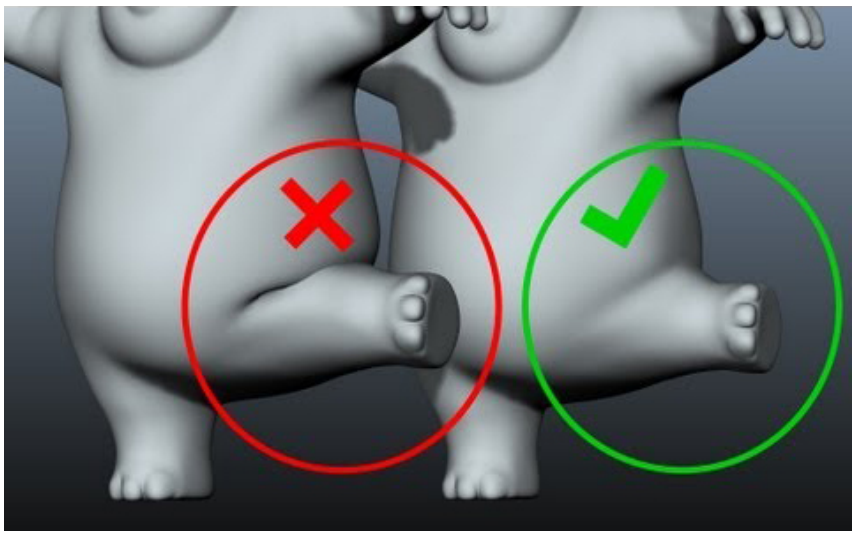


Рис. 25. Релістична анімація складки еластичної шкіри за допомогою ключових форм [19]

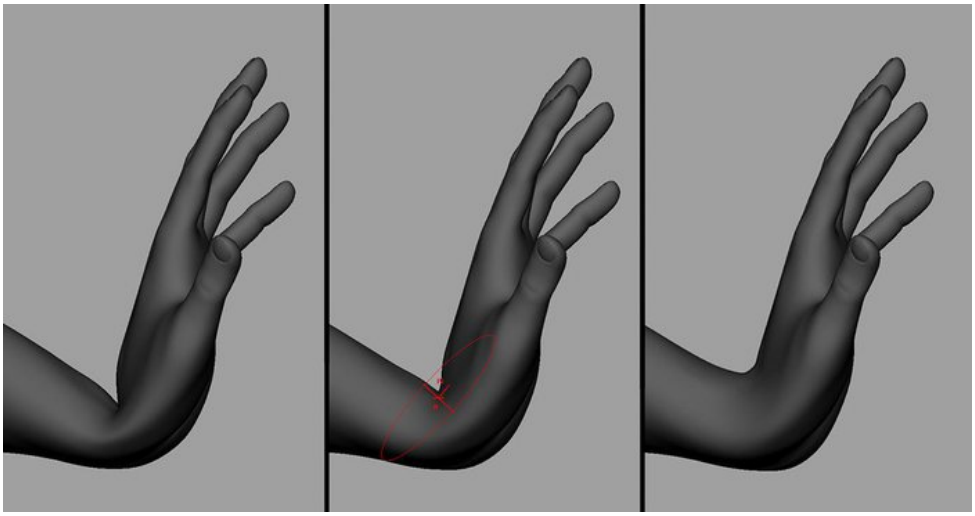


Рис. 26. Використання корегуючої ключової форми для усунення дефекту втрати об’єму [20]

Варто зазначити, що через складність підбору оптимального методу інтерполяції та оцінки віддаленості поточної пози моделі від обраної ключової пози, у проміжних положеннях ваговий коефіцієнт може бути занадто малим, що призводить до неповного усунення дефекту, або,

навпаки, занадто великим, що призводить до відхилення від бажаної форми, протилежного оригінальному дефекту. Задавання вагових коефіцієнтів корегуючих ключових поз вручну дозволяє позбутися від цих недоліків, але потребує значних затрат часу та зусиль, на додачу до створення самих корегуючих форм, що вже значно ускладнюють процес створення анімацій.

2. РОЗРОБЛЕННЯ АЛГОРИТМІЧНО-ПРОГРАМНОГО МЕТОДУ КОМПЕНСАЦІЇ ДЕФЕКТІВ DQ-СКІНІНГУ

2.1. Теоретичне підґрунтя

Основна ідея запропонованого методу полягає у приблизному визначенні трансформації, що відповідає небажаній деформації, для кожного окремого вертексу моделі, та застосуванні відповідних зворотних трансформацій після завершення роботи основного алгоритму скінінгу. Такий підхід подібний до використання корегуючих ключових форм, але, замість створення ключових форм вручну користувачем, вони генеруються автоматично в реальному часі, на основі параметрів, що доступні алгоритму скінінгу.

Повноцінна компенсація дефектів скінінгу у всіх можливих ситуаціях потребувала би занадто складних обчислень для використання у реальному часі, проте, зробивши певні припущення про цільову модель, можна досягнути компромісу між якістю та швидкодією.

В загальному випадку небажана деформація залежить від положення вертексу відносно кісток арматури та вагових коефіцієнтів кісток для цього вертекса. Проте, у переважній більшості реальних моделей, перехід між зонами впливу сусідніх кісток простягається на відстань, рівну одному-двом діаметрам кінцівки (рис. 27), а точка, де ваги сусідніх кісток рівні, є рівновіддаленою від цих кісток. Таким чином, лише одного з цих параметрів достатньо для приблизного визначення небажаної деформації. Оскільки коефіцієнти ваги доступні напряму у вигляді параметрів вертексу, доцільно в якості вхідного параметру обрати саме їх.

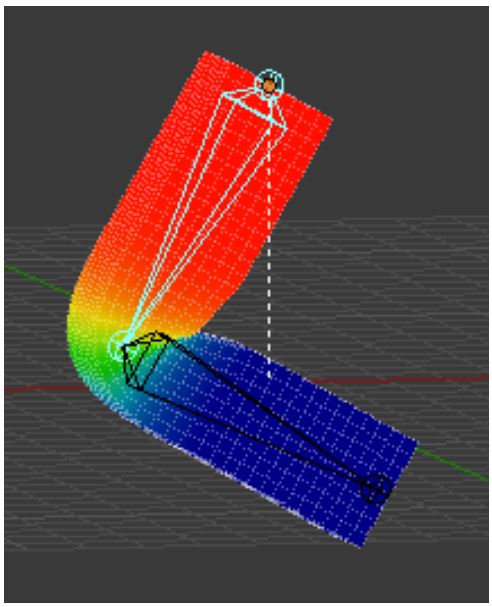


Рис. 27. Перехід між зонами впливу сусідніх кісток [21]

Для спрощення моделювання зігнутої кінцівки використовується двовимірна модель (рис. 28). При цьому така модель може відповідати будь-якому повздовжному зрізу тривимірної циліндричної кінцівки. Єдина різниця між окремими зрізами – радіус кінцівки. При цьому зміна радіусу кінцівки еквівалентна масштабуванню моделі, тому достатньо змоделювати лише одне значення радіусу кінцівки, а результуюче перетворення масштабувати пропорційно поточному значенню радіусу кінцівки.

Інтерполяція між дуальними кватерніонами результує у рух за гвинтовою траєкторією (рис. 29), тобто комбінацію зміщення вздовж певної осі та обертання навколо цієї ж осі. Зміщення сусідніх кісток одна відносно одної зустрічається рідко, тому з метою оптимізації розглядається лише випадок повороту. Розглядаючи суглоб в системі координат однієї з кінцівок, інтерполяція між позами двох кісток зводиться до повороту навколо суглоба. При використанні лінійної інтерполяції кватеорніонів кут повороту не буде пропорційним коефіцієнтам ваги, проте ця апроксимація є достатньо близькою.

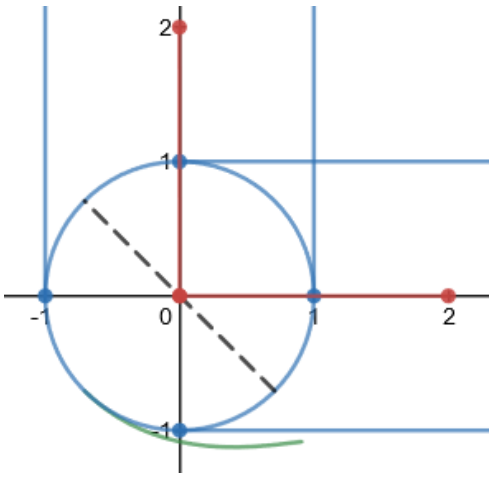


Рис. 28. Двовимірна модель зігнутого суглоба. Синій – поверхня моделі; червоний - кістки арматури; чорний пунктир – напрям небажаної деформації

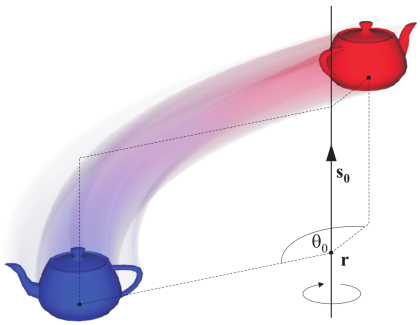


Рис. 29. Інтерполяція між дуальними кватерніонами результує у рух за гвинтовою траєкторією [9]

2.2. Моделювання небажаних деформацій

Залежність між коефіцієнтами ваги кісток та відстанню від вертекса до суглоба було змодельовано згорткою функції одиничного стрибка та функції Гауса. Така модель обрана на основі припущення, що вагові коефіцієнти були отримані шляхом привласненню кожного вертекса найближчій до

нього кістці (що моделюється функцією одиничного стрибка) та виконання кількох ітерацій згладжування вагових коефіцієнтів. Згладжування, зазвичай, виконується шляхом привласненню кожному вертексу усереднених значень ваги сусідніх вертексів (рис. 31), проте, згідно центральної граничної теореми, замість багатократного усереднення можна використовувати одну згортку з функцією Гауса (рис. 30).

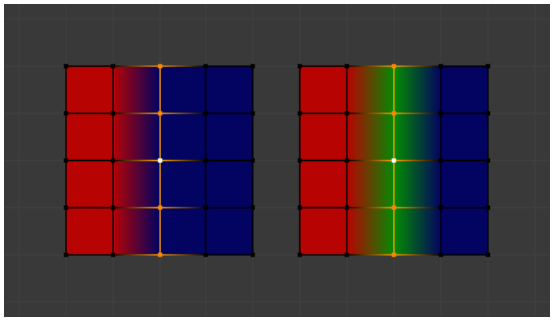


Рис. 30. Згладжування вагових коефіцієнтів кісток [22]

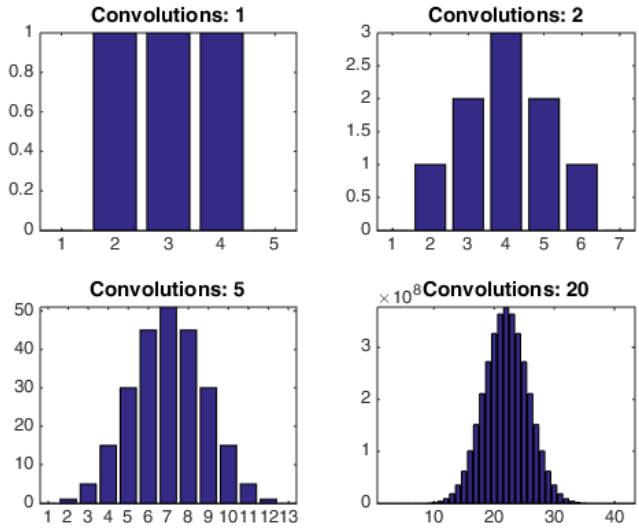


Рис. 31. Апроксимація багатократного усереднення ваги сусідніх точок згорткою з функцією Гауса [23]

Шляхом проб та помилок з використанням набору різноманітних анімованих моделей було встановлено, що функція Гауса з параметром c = 0.63 дозволяє найбільш точно змоделювати небажані деформації на тестовому наборі моделей (рис. 32). В якості альтернативного варіанту було протестовано використання лінійного спаду коефіцієнтів ваги по мірі віддалення від суглоба, але така модель створювала значно гірші деформації.

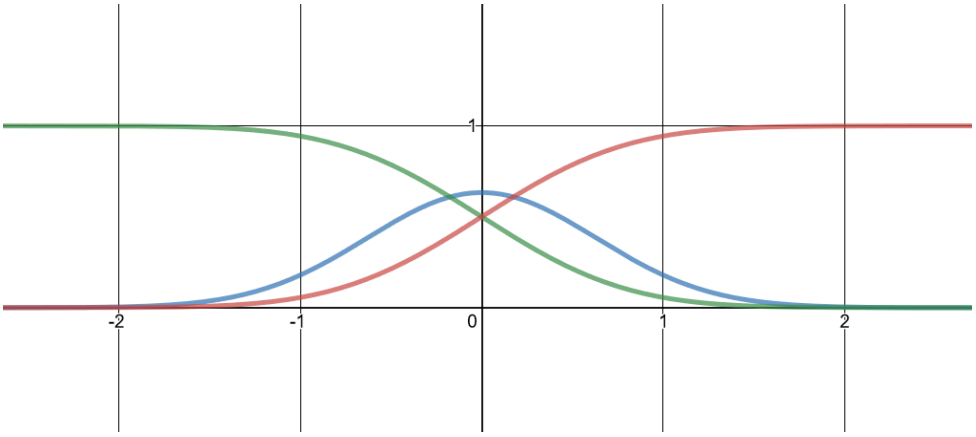


Рис. 32. Синій – функція Гауса, зелений та червоний – коефіцієнти ваги кісток

Для моделювання небажаних деформацій було зроблено припущення, що роздування суглоба відбувається шляхом зміщення вертексів вздовж бісектриси кута між напрямними векторами кісток (рис. 33).

Таким чином для моделювання небажаних деформацій має бути змодельована залежність між ваговими коефіцієнтами вертекса та довжиною зміщення вздовж бісектриси кута між напрямними векторами кісток. Точне визначення цієї залежності має занадто високу обчислювальну складність для розрахунків в реальному часі, але її можна з достатньою точністю апроксимувати з допомогою кубічного поліному.

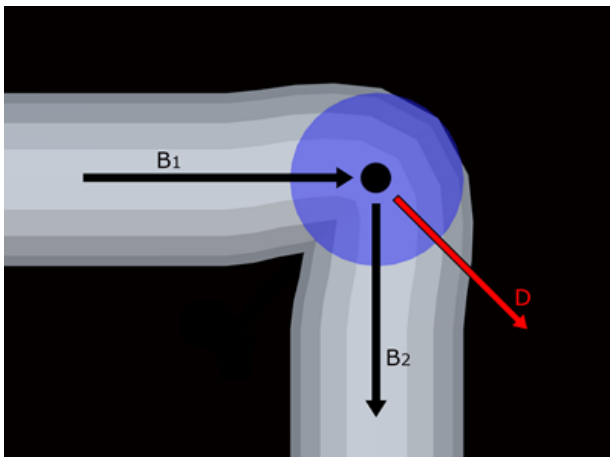


Рис. 33. B_1, B_2 – напрямні вектори кісток, D – бісектриса

При визначенні коефіцієнтів полінома було задано додаткові обмеження, що попереджують утворенню розривів:

- значення довжини зміщення має бути нулем на краях зони переходу: $f(0) = 0$, $f(1) = 0$;
- значення довжини зміщення має бути нулем в середині зони переходу: $f(0.5) = 0$.

В якості вхідного параметру доцільно обрати ваговий коефіцієнт другої кістки, оскільки в такому випадку виконання обмеженням $f(0) = 0$ можна забезпечити, задавши нуль в якості коефіцієнта при першому степені змінної, а обмеженням $f(1) = 0$ можна знехтувати, бо значення ваги другої кістки не може бути більшим за $\frac{1}{2}$. Обмеження $f(0.5) = 0$ попереджає утворення розривів в точці, де змінюється напрям небажаної деформації (рис. 35). Враховуючи ці обмеження, отримуємо формулу (3) (рис. 34).

$$f(w_2) = 2.29w_2 - 9.14w_2^2 + 9.12w_2^3, \tag{3}$$

де $f(w_2)$ – відстань зміщення; w_2 – ваговий коефіцієнт другої кістки (в порядку зменшення ваги).

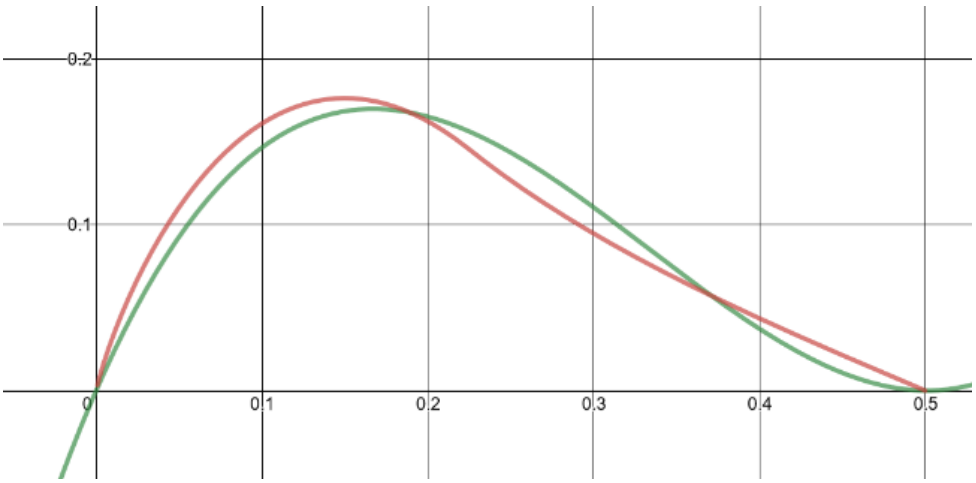


Рис. 34. Червоний – залежність між ваговим коефіцієнтом другої кістки (горизонтальна вісь) та довжиною зміщення вертекса (вертикальна вісь), зелений – апроксимація цієї залежності кубічним поліномом

Беручи до уваги вищезазначені обмеження, для кубічного полінома залишаються лише два не фіксовані коефіцієнти, що дозволяє вручну експериментувати з різними значеннями та емпіричним шляхом визначити оптимальний поліном для тестового набору анімованих моделей.

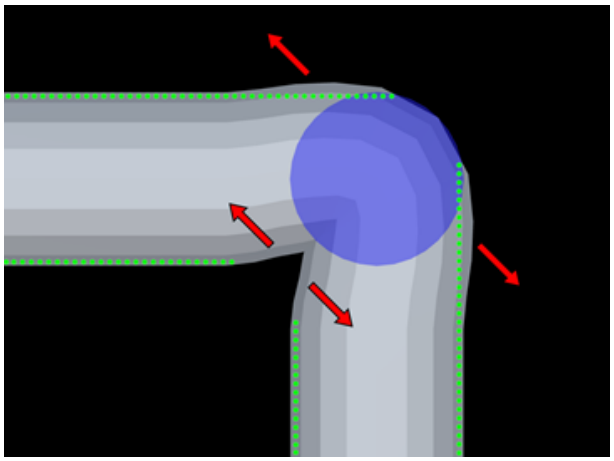


Рис. 35. Напрям небажаної деформації

Теоретично для визначення оптимальних значень коефіцієнтів

полінома можна було використовувати мінімізацію середньоквадратичного відхилення положення вертексів деформованої моделі від ідеалізованої форми деформованої моделі, проте мінімізація середньоквадратичного відхилення не гарантує досягнення найбільш візуально привабливих деформацій. Тому значення коефіцієнтів полінома підбиралися на основі візуальної оцінки деформацій спеціально сформованого набору моделей та анімацій.

Для порівняння якості компенсації дефектів анімації при використанні різних значень параметрів розробленого метода використовувалися наступні моделі:

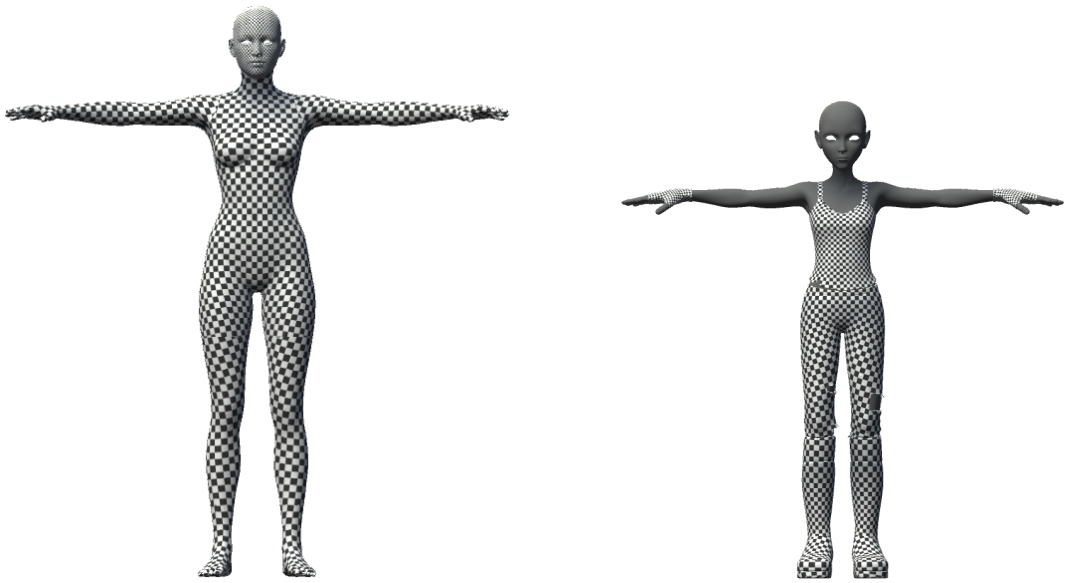
- реалістична модель людини Genesis 3 в складі програмного забезпечення для роботи з тривимірною графікою Daz Studio (рис. 36);
- стилізована модель людини Sintel з однойменного короткометражного анімаційного фільму (рис. 36);
- дві ідентичні моделі циліндру з одним суглобом, що відрізняються ступенем згладжування ваг кісток.

Моделі циліндру використовувалися для оцінки компенсації дефектів в ідеалізованих умовах, наближених до двовимірного випадку, розглянутого раніше, при цьому розглядаючи випадки як мінімального так і значного згладжування ваг кісток. Крім того на моделі циліндра найбільш помітні дефекти скручування. В той час, як моделі людини (реалістична та стилізована) використовувалися для пошуку проблематичних для розробленого методу ситуацій та його тестування у більш реалістичних умовах. Для пошуку недоліків розробленого методу використовувалося як ручне керування позами моделі, так а набір спеціалізованих анімацій range-of-motion, що спеціально включав велику кількість екстремальних поз та одночасних поворотів кісток навколо кількох осей. Особлива увага приділялася зонам плеча, стегна, щиколотки та кистей рук.

В результаті експериментів з набором тестових моделей було отримано формулу (4).

$$f(w_2) = 2.2w_2 - 9.6w_2^2 + 10.4w_2^3, \tag{4}$$

де $f(w_2)$ – відстань зміщення; w_2 – ваговий коефіцієнт другої кістки (в порядку зменшення ваги).



(а) (б)

Рис. 36. Реалістична модель людини Genesis 3 (зліва) та стилізована модель людини Sintel (справа)

Така апроксимація (рис. 37) дозволяє значно наблизити форму зігнутого суглоба до ідеалізованої моделі, як на двовимірній моделі (рис. 38), так і в реальній тривимірній анімації (рис. 39). Варто зазначити, що найкращої якості деформації було досягнуто при використанні полінома, що має похідну в точці 0.5, рівну нулю, таким чином запобігаючи розриву в перехідній зоні не тільки за значенням функції, але й за значенням похідної.

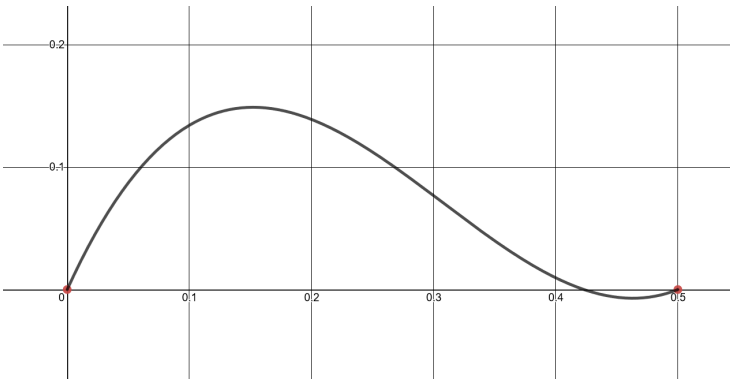


Рис. 37. Графік підібраного експериментальним шляхом полінома для апроксимації залежності відстані зміщення вертекса від вагового коефіцієнта другої кістки

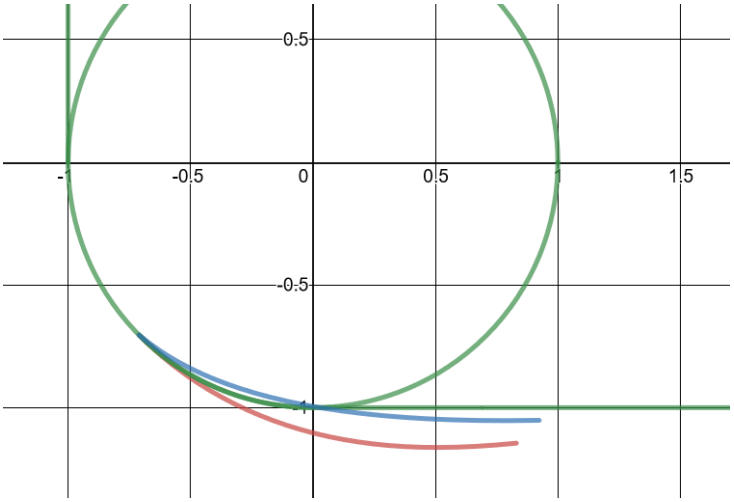


Рис. 38. Зелений – ідеалізована модель зігнутого суглоба; червоний – зігнутий суглоб з використанням dual quaternion скінінгу; синій – зігнутий суглоб з використанням dual quaternion скінінгу та компенсації роздування

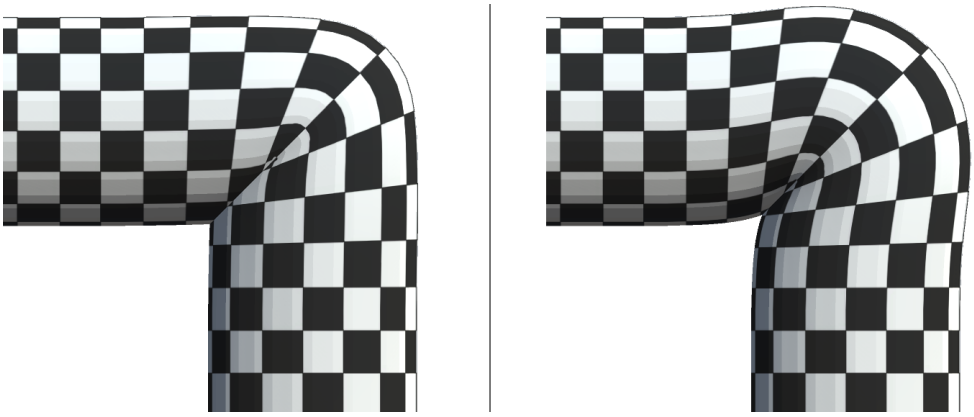


Рис. 39. Dual quaternion скінінг з компенсацією роздування (зліва) та без неї (справа)

Таким чином, пропонується наступний алгоритм компенсації дефекту (лістинг 1).

Лістинг 1. Базовий алгоритм компенсації дефекту роздутого суглоба

Ввід:

- кватерніони RQ_1 та RQ_2, що позначають повороти першої та другої кістки моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса) в глобальній системі координат;
- одиничні вектори V_{bone1} та V_{bone2}, що позначають напрям першої та другої кістки моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса) в глобальній системі координат;
- w_1, w_2 – коефіцієнти ваги першої та другої кістки моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса);
- s – коефіцієнт для ручного керування інтенсивністю компенсації дефекту;
- V_orig – координати розгляданого вертекса після DQ-скінінгу в глобальній системі координат.

Вивід: V – координати розгляданого вертекса після компенсації дефекту в глобальній системі координат.

Алгоритм компенсації дефекту роздутого суглоба:

1. $RQ = RQ_1 \cdot RQ_2^{-1}$
2. $V_{axis} = normalize(RQ.xyz)$
3. $V_{bisector} = normalize(V_{bone_1} + V_{bone_2})$
4. $if\ I_2 > I_1\ then$:
5. $\quad V_{bisector} = -V_{bisector}$
6. end
7. $l = 2.2w_2 - 9.6w_2^2 + 10.4w_2^3$
8. $l = l * s$
9. $V = V_{orig} + (V_{bisector} \cdot l)$

Домноженням кватерніона RQ_1 на обернений RQ_2 отримується кватерніон RQ, що позначає поворот між позиціями першої та другої кістки моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса).

Розглядаючи кватерніон з точки зору кута та осі, отримуємо вісь повороту між позиціями першої та другої кістки моделі шляхом нормалізації вектора, що складається з компонент x, y та z кватерніона RQ.

Бісектрису $V_{bisector}$ кута, утвореного першою та другою кісткою моделі, отримуємо шляхом нормалізації суми напрямних векторів цих кісток (V_{bone_1} та V_{bone_2}).

Для того, щоб стиснути суглоб в напрямку до центру, а не просто змістити всі вертекси в одному напрямку, необхідно визначити, з якої сторони суглоба знаходиться вертекс. Це можливо визначити за індексами кісток, якщо при імпорті моделі привласнити кісткам індекси таким чином, щоб для будь-якої пари кісток, кістка, що знаходиться нижче в ієрархії, завжди мала більший індекс. Тоді, порівнявши індекси першої та другої кістки моделі можна визначити, з якої сторони суглоба знаходиться розгляданий вертекс. Якщо індекс другої кістки є більшим, то напрям

деформації інвертуєтья.

В зоні, де вертекси з інвертованим та неінвертованим напрямком деформації знаходяться поруч, вага перших двух кісток моделі буде рівною 0.5 (за умови, що на вертекс вливають лише дві кістки). Оскільки значення полінома, що визначає відстань зміщення на основі ваги другої кістки моделі в точці 0.5 дорівнює 0, вертекси в цій зоні не будуть зміщуватися, а отже не утвориться розриву.

Після визначення напряму знаку напряму деформації, за допомогою вказаного вище полінома визначається відстань зміщення. Отримане значення множиться на коефіцієнт *s* для забезпечення можливості ручного керування інтенсивністю компенсації дефекту.

В результаті роботи алгоритму, положення вертекса V_{orig} зміщується у напрямку, заданому бісектрисою кута, утвореного першою та другою кісткою моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса) на відстань, визначену апроксимаційним поліномом та домножену на коефіцієнт ручного керування інтенсивністю компенсації дефекту *s*.

3. РЕАЛІЗАЦІЯ АЛГОРИТМІЧНО-ПРОГРАМНОГО МЕТОДУ КОМПЕНСАЦІЇ ДЕФЕКТІВ DQ-СКІНІНГУ

3.1. Обґрунтування вибору засобів реалізації

Для реалізації розробленого методу розглядалися варіанти створення додатка для одного з пакетів програмного забезпечення для створення тривимірних моделей та анімації (Autodesk Maya, Autodesk 3ds Max, Blender, Cinema4D, Modo та інші), створення додатка для одного з ігрових рушіїв (Unity, Unreal Engine, Cryengine) та створення окремого застосунку.

З наведених варіантів створення окремого застосунку є найбільш трудомістким та найменш універсальним, оскільки він потребує попередньої розробки середовища для роботи з тривимірною анімацією, що надається в готовому вигляді пакетами програмного забезпечення для роботи з тривимірною графікою та ігровими рушіями, а для використання такої реалізації розробленого методу у реальному проекті необхідно буде повністю її переписати для цільової платформи, використовуючи розроблений додаток лише як приклад.

Серед розроблення додатку для пакета програмного забезпечення для створення тривимірних моделей та анімації та для ігрового рушія було обрано розроблення додатку для ігрового рушія, оскільки однією з вимог до розроблюваного метода була висока швидкодія, що є набагато важливішою саме для ігрових рушіїв.

Серед рушіїв розглядалися Unity, Unreal Engine та Cryengine. Для реалізації розробленого методу було обрано рушій Unity, оскільки він має найбільш детальну документацію, зокрема щодо створення шейдерів. На відміну від Unreal Engine, код рушія Unity не є у вільному доступі, що дещо обмежує можливості по переоснащенню вбудованих функцій рушія, але для реалізації розробленого методу наданих рушієм Unity API цілком достатньо. Крім того, реалізація розробленого методу у вигляді додатка, без втручання

у вихідний код рушія, значно спрощує підключення розробленої реалізації до існуючих проектів.

Для створення додатків у рушії Unity використовується мова програмування C#, при чому доступні два бекенди: Mono та IL2CPP.

Mono – це відкрита крос-платформенна реалізація фреймворку .Net, що включає компілятор та середовище виконання. При використанні бекенду mono код C# додатків компілюється в байт-код, що виконується віртуальною машиною mono.

IL2CPP (Intermediate Language To C++) – це бекенд, розроблений для рушія Unity, як альтернатива для mono. При використанні бекенду IL2CPP байт-код додатків конвертується в код C++ та компілюється у машинний код, що використовує бібліотеку libil2cpp. При цьому проводиться ряд оптимізацій, зокрема оптимізація викликів віртуальних методів та видалення недосяжного коду.

Вибір бекенду не впливає на процес створення додатка та може бути змінений у налаштуваннях проекту в будь-який час (рис. 40).

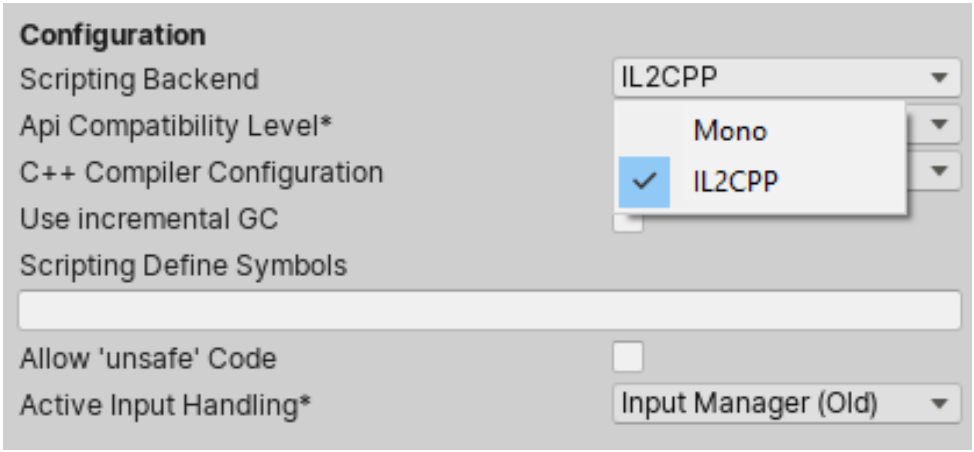


Рис. 40. Налаштування проекту Unity. Вибір бекенду

Для написання шейдерів у рушії Unity використовується мова CG, що є

сумісною з HLSL. Рушій Unity дозволяє використовувати компілятор HLSL від Microsoft для проектів DirectX, або автоматично перекладати код з HLSL на GLSL за допомогою компілятора HLSLcc для проектів OpenGL, OpenGL ES, Vulkan та Metal. При цьому немає потреби вносити зміни у вихідний код на HLSL, що є спільним для всіх налаштувань компіляції.

3.2. Особливості реалізації розробленого алгоритмічно-програмного методу

Розроблений метод було реалізовано у вигляді додатка для рушія Unity, що включає один шейдер матеріалу, три скрипти та два розрахункові шейдери (рис. 41).

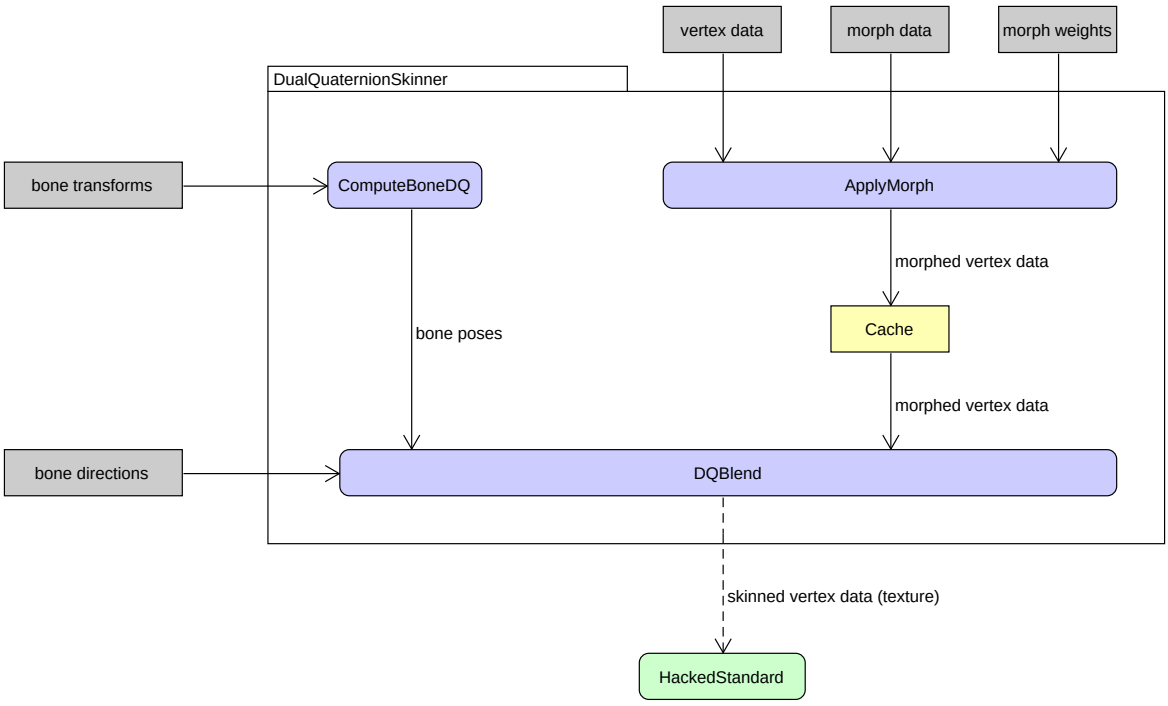


Рис. 41. Схема реалізації розробленого методу

Дані про положення вертексів, індекси та вагові коефіцієнти відповідних кісток арматури, напрямки нормалей та дотичних,

ключові форми та їх вагові коефіцієнти передаються до розрахункового шейдера DQBlend скриптом DualQuaternionSkinner (рис. 42). При цьому використовується кешування положень вертексів, напрямів нормалей та дотичних з застосованими ключовими формами. Повторний розрахунок ключових форм виконується лише у разі зміни вагових коефіцієнтів.

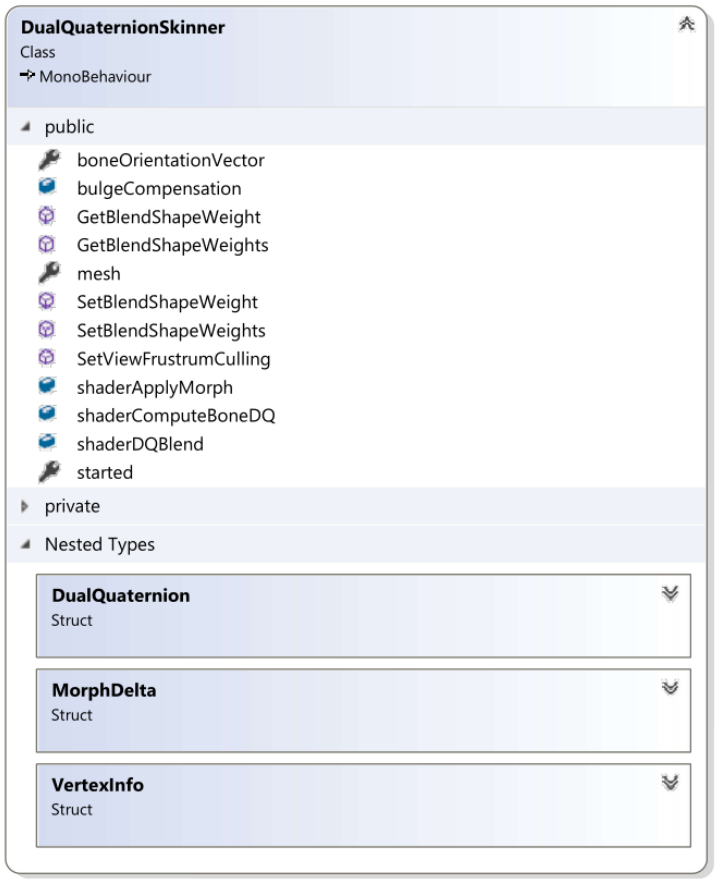


Рис. 42. Клас DualQuaternionSkinner

Функції GetBlendShapeWeight(int index) та SetBlendShapeWeight(int index, float weight) дозволяють відповідно отримати та встановити значення вагового коефіцієнта ключової форми за обраним індексом. Значення вагового коефіцієнти задається у відсотках (від 0 до 100), при чому скрипт коректно обробляє від’ємні значення вагового коефіцієнта та значення більше 100.

Функції GetBlendShapeWeights( ) та SetBlendShapeWeights(float[ ] weights) дозволяють відповідно отримати за встановити значення вагових коефіцієнтів одразу для всіх доступних цільових форм у вигляді масиву чисел з плаваючою точкою, індекси в якому відповідають індексам відповідних ключових форм.

Властивість boneOrientationVector визначає який з базисних векторів локальної системи координат кістки арматури має бути напрямленим вздовж відповідної кінцівки. Цей вектор має бути однаковим для всіх кісток арматури для коректної роботи скрипта (рис. 48). При цьому вектор може бути вказаний з від'ємним напрямом, що інвертує напрям деформації для компенсації дефектів анімації. Коректними значеннями для поля boneOrientationVector є: $(1,\ 0,\ 0)$, $(-1,\ 0,\ 0)$, $(0,\ 1,\ 0)$, $(0,\ -1,\ 0)$, $(0,\ 0,\ 1)$, $(0,\ 0,\ -1)$.

Властивість mesh надає доступ до об'єкту моделі, що в даний момент використовується скриптом та доступна тільки для читання. На даний момент скрипт DualQuaternionSkinner не підтримує заміну моделі під час роботи.

Функція SetViewFrustrumCulling(bool viewFrustrumCulling) дозволяє включати та відключати відсікання пірамідою огляду (рис. 43). Рушій Unity не прораховує обмежувальну коробку (рис. 44) моделі на кожному кадрі з метою оптимізації. Замість цього, при імпорті моделі з анімаціями, рушій генерує обмежувальну коробку, що вміщує всі імпортовані анімації. Негативним наслідком такого підходу є те, що при використанні згенерованих в реальному часі анімацій, або ручному редагуванні пози моделі, модель може виходити за рамки згенерованої обмежувальної коробки, призводячи до некоректної роботи відсікання пірамідою огляду. Відключення відсікання піраміди огляду для такої моделі попереджає ситуацію, коли модель не відображається через те, що її обмежувальна коробка знаходиться поза межами піраміди огляду, в той час, як полігональна

сітка моделі знаходиться в межах піраміди огляду.

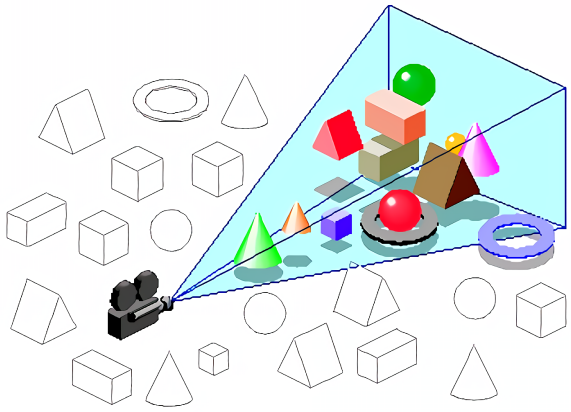


Рис. 43. Відсікання пірамідою огляду

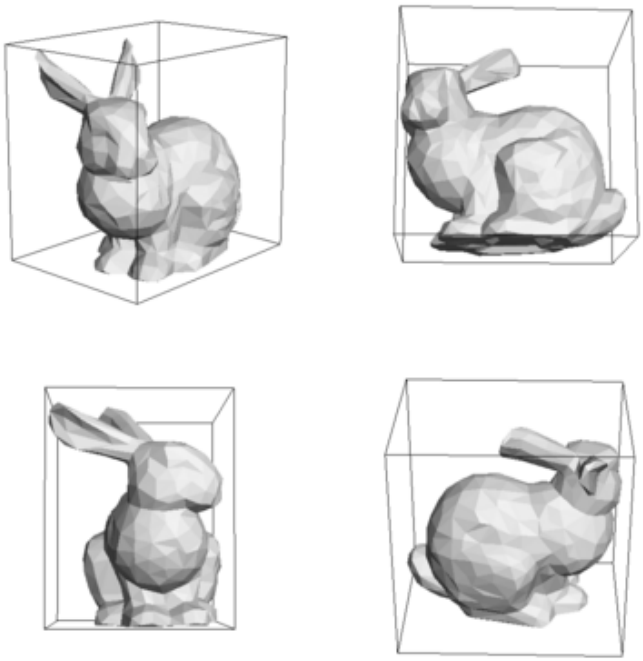


Рис. 44. Обмежувальна коробка моделі

Поля shaderApplyMorph, shaderComputeBoneDQ та shaderDQBlend

слугують для збереження посилань на відповідні шейдери. Ці поля мають доступ на запис, що теоретично дозволяє користувачу замінити шейдери власною реалізацією. У разі використання розробленої в рамках цієї роботи реалізації, ці поля мають залишатися незмінними.

Властивість started доступна лише на читання та повертає булеву змінну, що повідомляє про поточний стан скрипта (працює, або очікує старту).

Розрахунковий шейдер ComputeBoneDQ приймає матриці 4x4, що зберігають поточне положення в просторі кісток арматури та пози відпочинку для відповідних кісток та повертає дуальні кватерніони у вигляді структур DualQuaternion (рис. 45), що визначають поточне положення кісток арматури відносно їх поз відпочинку.

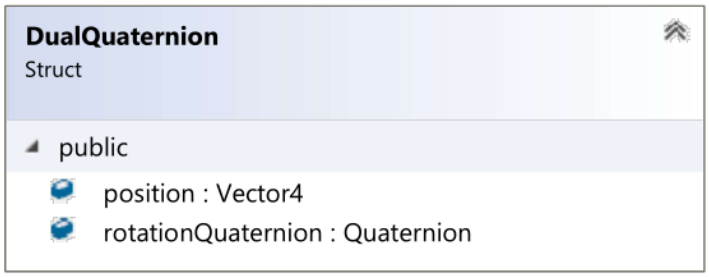


Рис. 45. Структура DualQuaternion

Розрахунковий шейдер DQBlend приймає глобальний коефіцієнт компенсації дефекту роздутого суглоба (від 0 до 1), дуальні кватерніони з шейдера ComputeBoneDQ, що зберігають поточне положення кісток арматури в просторі відносно пози відпочинку у вигляді структур DualQuaternion (рис. 45), та інформацію про вертекси моделі у вигляді структур VertexInfo (рис. 46), що включає індекси та ваги кісток арматури, до яких прив’язано даний вертекс, положення вертекса в просторі в позі відпочинку, вертексну нормаль в позі відпочинку, вертексну дотичну в позі відпочинку, та коефіцієнт компенсації дефекту роздутого суглоба для

даного вертексу, що дорівнює відстані від даного вертекса до кістки, що має для нього найбільший коефіцієнт (припускається, що ця кістка є для нього найближчою).

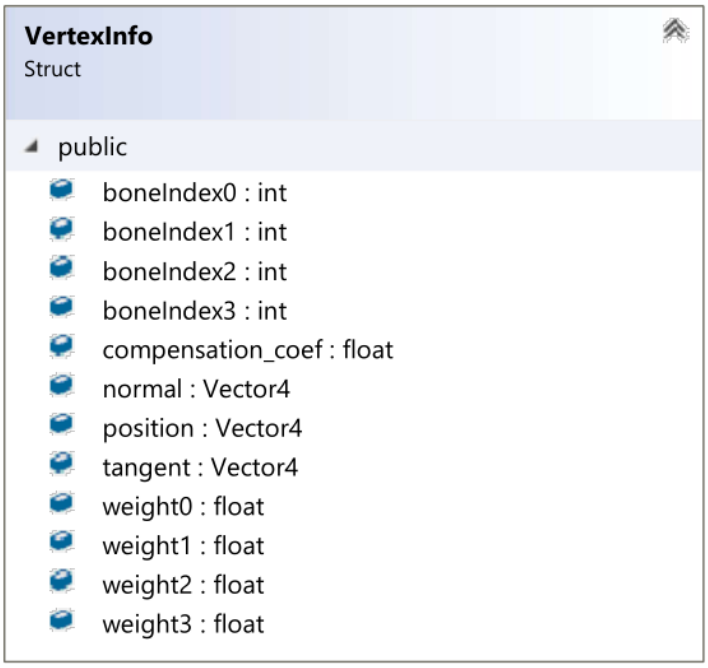


Рис. 46. Структура VertexInfo

На основі вхідних даних шейдер DQBlend розраховує положення в просторі, нормалі та дотичні для вертеків моделі методом скінінгу дуальними кватерніонами, після чого виконує компенсацію дефекту роздутих суглобів розробленим методом. Для цього шейдер потребує додаткові вхідні дані – напрямні вектори кісток арматури в глобальній системі координат. Ці вектори передаються напряму з шейдера ComputeBoneDQ через структурний буфер.

Оскільки на момент розробки з доступних у рушії Unity графічних бібліотек лише DirectX підтримує використання структурних буферів у вертексному шейдері, результат роботи шейдера DQBlend передається до вертексного шейдера за допомогою спеціально сформованих текстур. Для

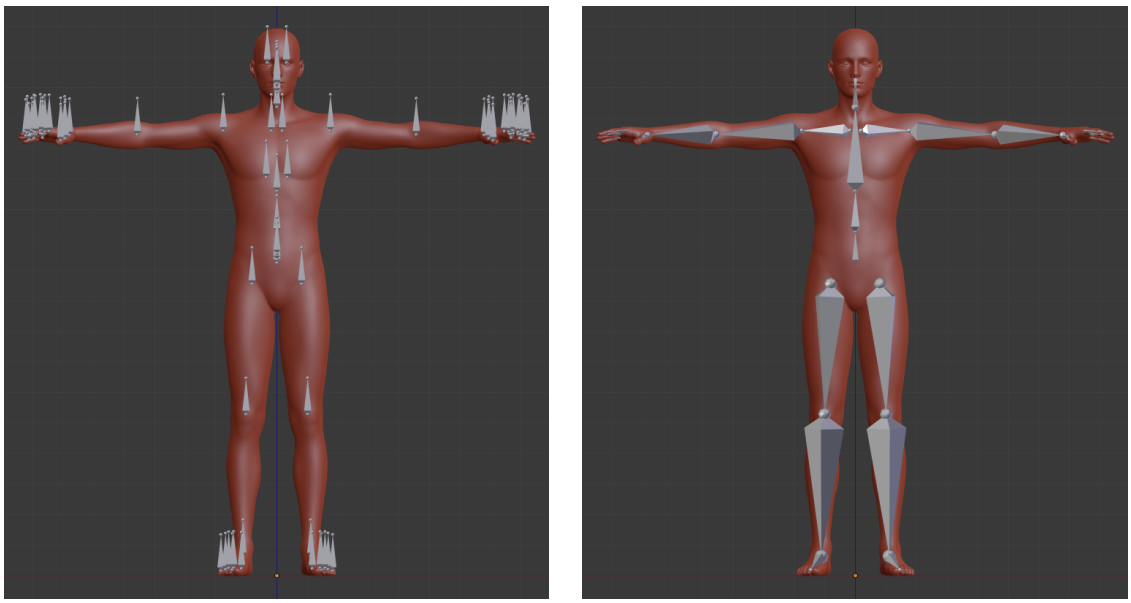
передачі даних використовуються дві текстури float4 та одна текстура float2 (рис. 47).

```
/*
    Vulkan and OpenGL only support ComputeBuffer in compute shaders
    passing data to the vertex and fragment shaders is done through RenderTextures

    using ComputeBuffers would improve the efficiency slightly but it would only work with Dx11

    layout is as such:
        rtSkinnedData_1         float4          vertex.xyz, normal.x
        rtSkinnedData_2         float4          normal.yz,  tangent.xy
        rtSkinnedData_3         float2          tangent.zw
*/
RenderTexture rtSkinnedData_1;
RenderTexture rtSkinnedData_2;
RenderTexture rtSkinnedData_3;
```

Рис. 47. Схема пакування даних про вертекси у текстурах

Для роботи розробленого методу компенсації дефектів dual quaternion скіннігу необхідно, щоб індекси кісток були відсортовані в порядку віддаленості по ієрархії від кореневої кістки. Ця умова забезпечується скриптом AssetPostProcessorReorderBones, що виконує сортування кісток при імпорті анімованих моделей. Для правильної роботи розробленого додатку після імпорту у існуючий проект необхідно виконати реімпорт анімованих моделей, щоб викликати скрип AssetPostProcessorReorderBones та виконати сортування кісток.

Крім необхідності сортування кісток розроблений метод накладає на використовувані анімовані моделі ще одну умову – локальні осі координат кісток арматури мають бути напрямлені вздовж відповідних кінцівок, при чому для всіх кісток арматури основна локальна вісь має бути однакова (X, Y або Z). Приклад допустимої та недопустимої арматури наведено на рис. 48.



(а) (б)

Рис. 48. Приклад допустимої (а) на недопустимої (б) конфігурації арматури для розробленого методу компенсації дефектів dual quaternion скінінгу

Для зручності користування додатком було розроблено скрипт DualQuaternionSkinnerEditor, що надає інтерфейс для роботи з компонентом DualQuaternionskinner (рис. 49).

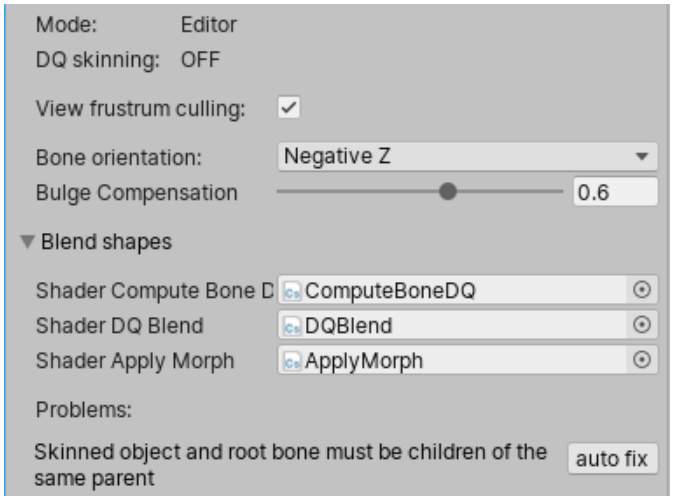


Рис. 49. Інтерфейс розробленого додатку

Розроблений додаток здатний самостійно виявляти розповсюджені помилки налаштування. В разі виявлення помилки в нижній секції інтерфейсу компонента DualQuaternionSkinner відображається короткий опис виявленої помилки та кнопка auto-fix, що дозволяє додатку самостійно виправити виявлену помилку. При цьому зміни в проекті, що виконуються для виправлення виявленої помилки, записуються в історію дій, що дозволяє покроково відміняти внесені зміни у разі виявлення користувачем небажаних побічних ефектів автоматичного виправлення. Помилки, налаштування, які розроблений додаток здатний автоматично виявляти, включають в себе:

- для використовуваної моделі відключено доступ на запис;
- коренева кістка та об’єкт з компонентом DualQuaternionSkinner не є прямими нащадками одного і того самого об’єкту;
- кістки арматури мають змінені розміри.

Відключений доступ на запис – єдина помилка, для якої розроблений додаток не надає автоматичного виправлення, замість цього користувач має поставити відповідну відмітку у налаштуваннях імпорту моделі (рис. 50).

Для використання розробленого додатку необхідно додати компонент DualQuaternionSkinner до об’єкту анімованої моделі, що містить вбудований компонент SkinnedMeshRenderer. При цьому компонент SkinnedMeshRenderer автоматично додає до об’єкта необхідний компонент MeshFilter та відключає компонент SkinnedMeshRenderer під час запуску режиму Play, перехоплюючи на себе керування анімацією (рис. 51).

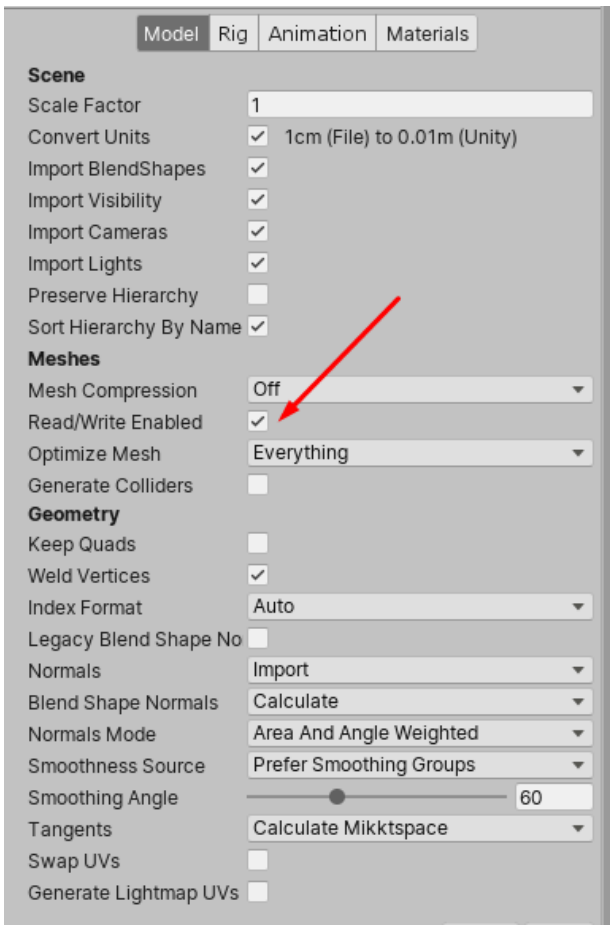


Рис. 50. Налаштування режиму читання-запису для імпортованої моделі в рушії Unity

Всі матеріали анімованої моделі у компоненті SkinnedMeshRenderer мають використовувати спеціальний шейдер, що дозволяє компоненту DualQuaternionSkinner керувати положенням вертексів та напрямами нормалей та дотичних анімованої моделі. У межах даної роботи було розроблено шейдер HackedStandard, що є копією вбудованого шейдера рушія Unity з додаванням підтримки компонента DualQuaternionSkinner.

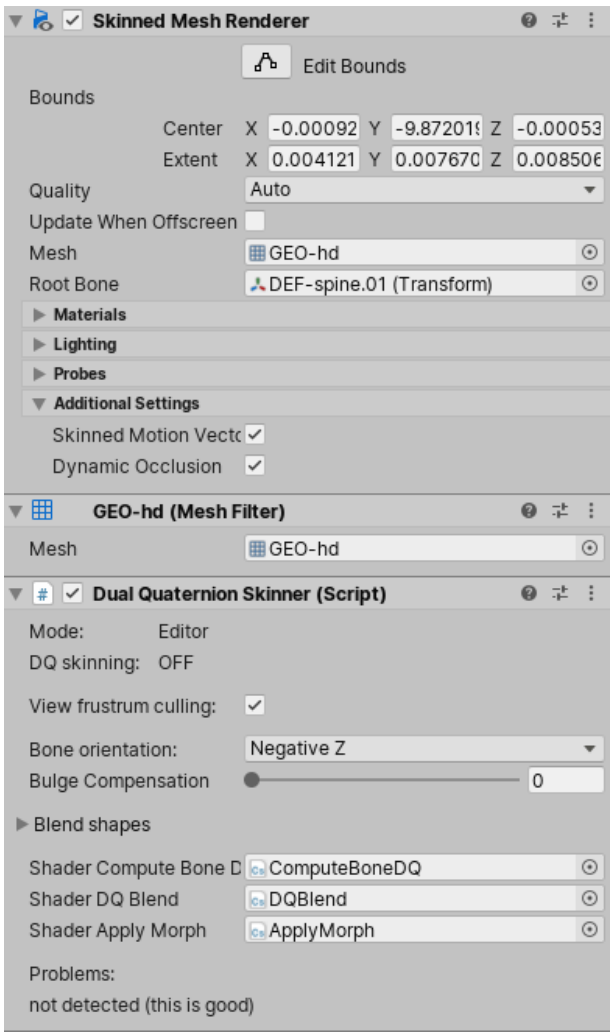


Рис. 51. Компоненти, необхідні для роботи розробленого додатка

Інтерфейс компонента SkinnedMeshRenderer (рис. 49) дозволяє контролювати інтерсивність компенсації дефектів dual quaternion скінінгу за допомогою повзунка ”Bulge Compensation” та (за необхідності) відключати відсічення пірамідою огляду. Рушій Unity будує межі анімованих об’єктів для відсічення з урахуванням усіх імпортованих анімацій. У випадку, коли розробник змінює положення кісток арматури вручну (наприклад, з метою тестування), може виникнути ситуація, коли відсічення спрацьовує для моделі, що видна на екрані. Для таких випадків передбачено можливість відключати відсічення пірамідою огляду.

4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ВДОСКОНАЛЕННЯ АЛГОРИТМІЧНО-ПРОГРАМНОГО МЕТОДУ КОМПЕНСАЦІЇ ДЕФЕКТІВ DQ-СКІНІНГУ

4.1. Залежність інтенсивності компенсації дефекту від кута згину суглоба

При малих кутах згину суглоба напрям апроксимованої деформації наближається до напрямних векторів кісток, що призводить до повздовжного стискання суглоба (рис. 52).

Рис. 52. Повздовжне стискання суглоба при малих кутах згину

Для уникнення такого дефекту інтенсивність компенсації роздування суглоба має зменшуватись при малих кутах згину суглоба. Цього можна досягти, домножуючи апроксимовану довжину зміщення на синус кута згину суглоба (рис. 53).

Окрему проблему становить скручування суглоба, оскільки при скручуванні збільшується кут повороту, але напрям апроксимованої деформації залишається незмінним, що також призводить до повздовжного стискання суглоба (рис. 54а). При використанні комбінації згину та

скручування ця невідповідність також створює дефект (рис. 55а)

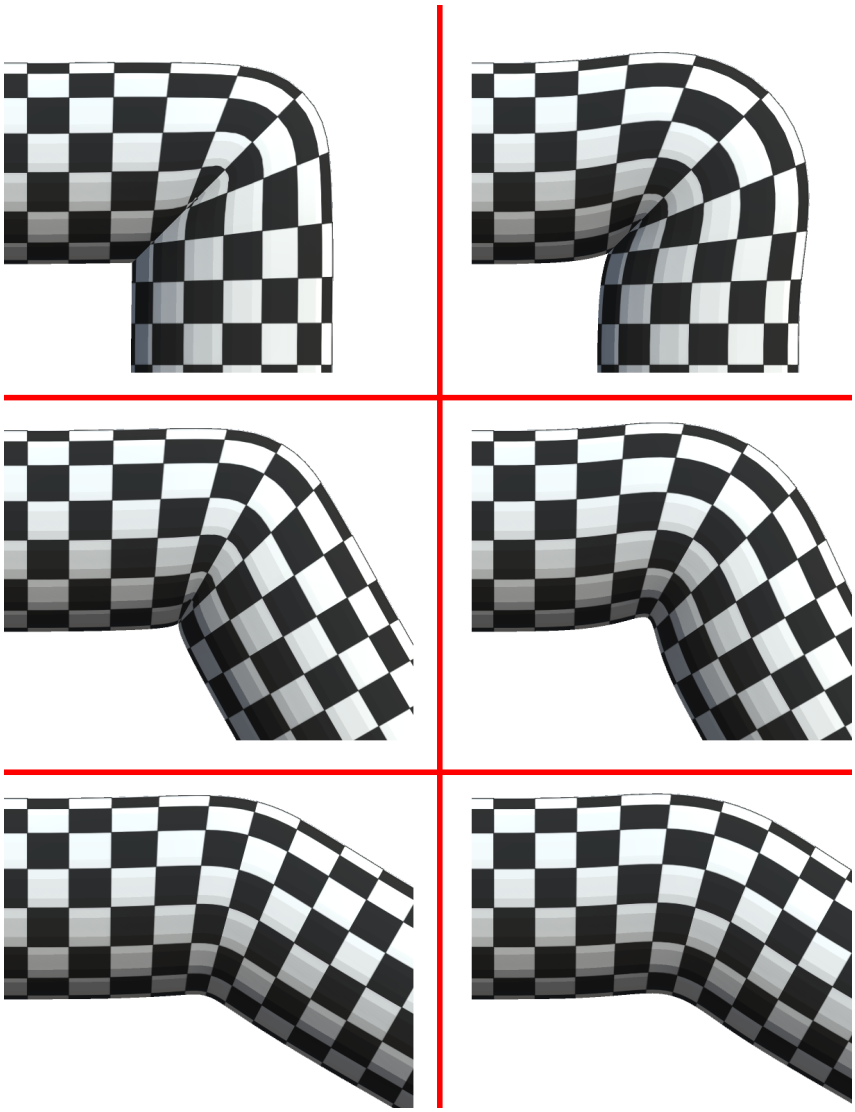
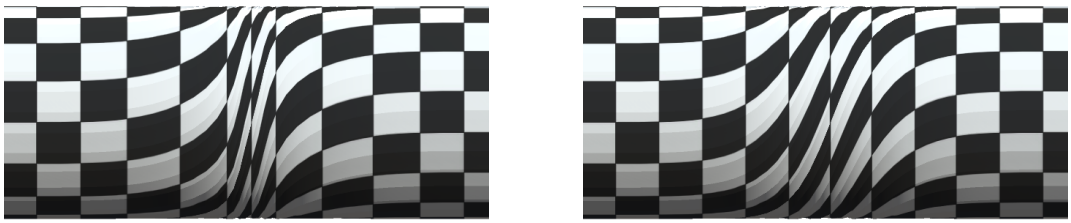


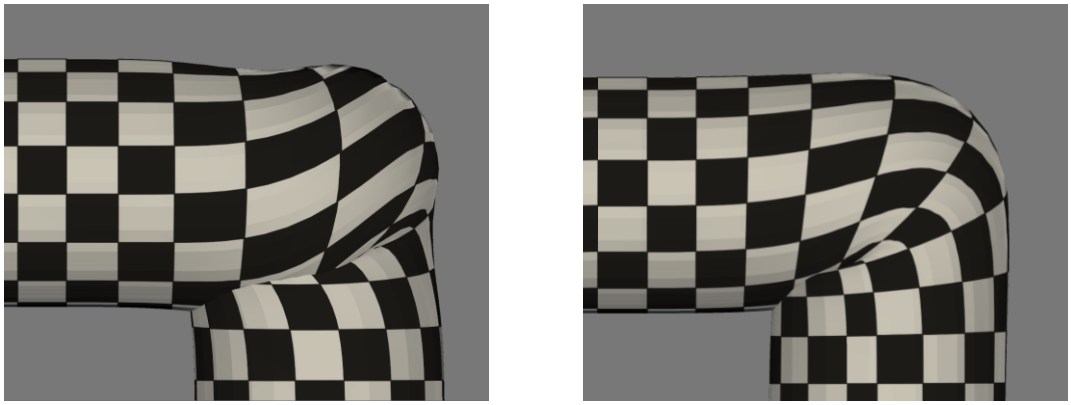
Рис. 53. Зміна інтенсивності компенсації дефекту в залежності від кута згину суглоба

Для уникнення цього дефекту інтенсивність компенсації роздування суглоба має зменшуватися пропорційно відношенню деформацій згину до скручування. Результат використання такого підходу продемонстровано на рис. 54б та рис. 55б.



(а) (б)

Рис. 54. Дефект повздовжного стискання суглоба при скручуванні (а); дефект усунуто завдяки зменшенню відстані зміщення на малих кутах згину (б)



(а) (б)

Рис. 55. Дефект при використанні комбінації скручування та згину (а), дефект усунуто завдяки зменшенню відстані зміщення пропорційно відношенню деформацій згину та скручування (б)

4.2. Зони впливу трьох і більше кісток

У зонах впливу більш ніж двох кісток запропонований метод призводить до розривів у місцях, де друга та третя кістка (в порядку зменшення відповідних коефіцієнтів ваги) міняються місцями (рис. 56а). Цей розрив може бути усунуто шляхом плавного зменшення інтенсивності

компенсації роздування суглоба навколо перехідної зони, де значення коефіцієнтів ваги другої та третьої кістки (в порядку спадання коефіцієнтів ваги) є близькими (рис. 56б).

(а) (б)

Рис. 56. Розрив у зоні впливу більш, ніж двох кісток (а), розрив усунуто завдяки плавному зменшенню інтенсивності компенсації роздування суглоба в зоні переходу (б)

З врахуванням залежності інтенсивності компенсації дефекту від куту згину суглоба та зон впливу більше, ніж трьох кісток, алгоритм компенсації дефекту роздутого суглоба приймає вигляд, представлений в лістингу 2.

Лістинг 2. Вдосконалений алгоритм компенсації дефекту роздутого суглоба

Ввід:

- кватерніони RQ_1 та RQ_2, що позначають повороти першої та другої кістки моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса) в глобальній системі координат;
- одиничні вектори V_{bone1} та V_{bone2}, що позначають напрям першої та другої кістки моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса) в глобальній системі координат;

- w_1, w_2 – коефіцієнти ваги першої та другої кістки моделі (в порядку спадання коефіцієнтів ваги для розгляданого вертекса);
- s – коефіцієнт для ручного керування інтенсивністю компенсації дефекту;
- V_orig – координати розгляданого вертекса після DQ-скінінгу в глобальній системі координат.

Вивід: V – координати розгляданого вертекса після компенсації дефекту в глобальній системі координат.

Вдосконалений алгоритм компенсації дефекту роздутого суглоба:

1. $RQ = RQ_1 \cdot RQ_2^{-1}$
2. $V_{axis} = normalize(RQ.xyz)$
3. $V_{bisector} = normalize(V_{bone_1} + V_{bone_2})$
4. $if\ I_2 > I_1\ then$:
5. $\quad V_{bisector} = -V_{bisector}$
6. end
7. $V_{offset} = V_{bisector} - V_{axis} \cdot (V_{axis} \cdot V_{bisector})$
8. $w = \dfrac{w_2}{w_1 + w_2}$
9. $l = 2.2w - 9.6w^2 + 10.4w^3$
10. $l = l * min(1, 2\sqrt{1 - RQ.W})$
11. $l = l * (w_1 + w_2)$
12. $l = l * (1 - \dfrac{w_3}{w_2})$
13. $l = l * s$
14. $V = V_{orig} + (V_{bisector} \cdot l)$

Різниця між вдосконаленим та базовим алгоритмом компенсації дефекту роздутого суглоба полягає в рядках з 7-го по 12-й.

В рядку 8 замість використання вагового коефіцієнта другої кістки

моделі (в порядку спадання вагового коефіцієнта для розгляданого вертекса) визначається коефіцієнт w. Оскільки сума всіх вагових коефіцієнтів для даного вертекса складає 1, а даний алгоритм приймає до уваги лише перші дві кістки, замість використання вагового коефіцієнта другої кістки напряму, цей коефіцієнт ділиться на суму вагових коефіцієнтів першої та другої кістки, таким чином ігноруючи всі інші коефіцієнти.

В рядку 9 отриманий раніше коефіцієнт w використовується замість w_2.

В рядку 10 відстань зміщення вертекса множиться на коефіцієнт, що залежить від кута згину суглоба. Компонент W кватерніона RQ дорівнює косинусу половинного кута згину суглоба.

В рядку 11 відстань зміщення вертекса множиться на суму вагових коефіцієнтів першої та другої кістки. У випадку, коли на вертекс впливають лише дві кістки, цей коефіцієнт буде рівний одиниці та не впливатиме на отриману деформацію. У випадку, коли на вертекс впливають три і більше кісток, цей коефіцієнт зменшить відстань деформації пропорційно долі впливу перший двох кісток на кістку.

В рядку 12 відстань зміщення вертекса множиться на коефіцієнт, що визначає наближення вагового коефіцієнта другої кістки до вагового коефіцієнта третьої кістки. Це необхідно, оскільки в зоні, де друга та третя кістка міняються позиціями, виникає розрив. Даний коефіцієнт прирівнює відстань зміщення до нуля в зоні розриву та запобігає дефекту.

4.3. Оптимізація реалізації розробленого алгоритмічно-програмного методу

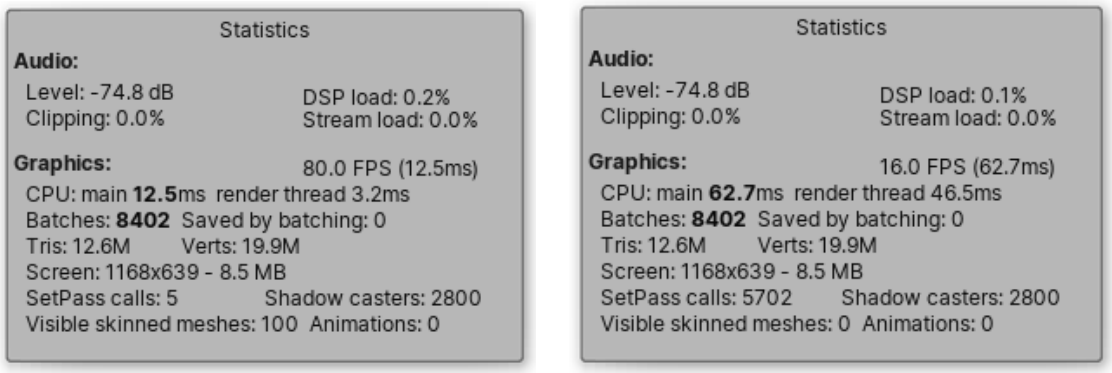
Однією з проблем виміру швидкодії скриптів за допомогою вбудованого профайлера Unity є те, що виклик збирача сміття значно впливає на швидкодію додатку, але, оскільки він виконується для додатка в цілому, для окремого скрипта можна виміряти лише кількість згенерованого ним сміття в байтах, що дає лише приблизне уявлення про реальний вплив даного

скрипта на швидкодію додатку.

Розроблена реалізація методу компенсації дефекту роздутого суглоба не генерує сміття, що дозволяє виключити необхідність оцінювати вплив виклику збирача сміття на швидкодію скрипта.

Для порівняння швидкодії розробленої реалізації з вбудованим скінінгом Unity використовувалися 100 копій моделі людини з пакету програмного забезпечення для роботи з 3D графікою Daz Studio (кожна модель має 126 тис. трикутників та 80 кісток).

Перша ітерація розробленої реалізації показала значно гіршу швидкодію за вбудований скінінг (рис. 57).



(а) (б)

Рис. 57. Статистика рендеру кадру з 100 копіями моделі людини використовуючи вбудований скінінг Unity (а) та розроблений скрипт (б)

Крім значної різниці у FPS на другому скріншоті (рис. 57б) значно більше викликів SetPass (5702 з розробленою реалізацією скінінгу дуальними кватерніонами з компенсацією дефекту роздутого суглоба та 5 з вбудованим скінінгом Unity) та у графі ”Visible skinned meshes” відображається нуль.

Нуль у графі ”Visible skinned meshes” спричинений тим, що розроблена реалізація скінінгу не використовує вбудований компонент для рендерингу

анімованих моделей SkinnedMeshRenderer, а замість цього додає компонент MeshRenderer (що використовується для моделей без скінінгу) та змінює положення вертексів моделі у вертексному шейдері. Як наслідок, рушій Unity не розпізнає моделі, анімовані за допомогою розробленого скрипта, як анімовані.

У глибокому профайлінгу (рис. 58) помітно, що найбільше часу під час обробки кадру виділяється на виклики Material.SetPassUncached. Це говорить про те, що збільшення кількості викликів SetPass і є причиною низької швидкодії розробленого скрипта.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	86.7%	0.1%	2	0 B	62.75	0.07
▼ Camera.Render	61.6%	0.0%	1	0 B	44.58	0.05
▼ Drawing	42.1%	0.0%	1	0 B	30.47	0.02
▼ Render.OpaqueGeometry	41.8%	0.0%	1	0 B	30.25	0.01
▼ RenderForwardOpaque.Render	39.9%	0.0%	1	0 B	28.88	0.02
▼ RenderForward.RenderLoopJob	37.6%	4.3%	1	0 B	27.23	3.15
Material.SetPassUncached	31.2%	31.2%	2800	0 B	22.57	22.57
▶ BatchRenderer.Flush	2.0%	1.2%	2800	0 B	1.50	0.92
▶ Shadows.RenderShadowMap	2.0%	0.0%	1	0 B	1.50	0.01
RenderLoop.CleanupNodeQueue	0.0%	0.0%	1	0 B	0.06	0.06
▶ RenderForwardOpaque.CollectShadows	0.0%	0.0%	1	0 B	0.04	0.01
Clear	0.0%	0.0%	1	0 B	0.00	0.00
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00
▶ Shadows.PrepareShadowmap	1.1%	0.0%	1	0 B	0.80	0.03
RenderForwardOpaque.Prepare	0.7%	0.7%	1	0 B	0.54	0.54
▶ Render.Prepare	0.2%	0.2%	1	0 B	0.16	0.16
▶ Render.TransparentGeometry	0.0%	0.0%	1	0 B	0.01	0.00
▶ UnityEngine.CoreModule.dll!UnityEngine::Cam	0.0%	0.0%	1	0 B	0.00	0.00
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00
Render.MotionVectors	0.0%	0.0%	1	0 B	0.00	0.00
RenderLoop.CleanupNodeQueue	0.0%	0.0%	1	0 B	0.00	0.00
Camera.ImageEffects	0.0%	0.0%	1	0 B	0.00	0.00
▼ UpdateDepthTexture	18.4%	0.1%	1	0 B	13.32	0.13
▼ DepthPass.Job	18.0%	2.2%	1	0 B	13.06	1.62
Material.SetPassUncached	14.0%	14.0%	2800	0 B	10.16	10.16
▶ BatchRenderer.Flush	1.7%	1.1%	2800	0 B	1.27	0.79
▶ WaitForJobGroupID	0.1%	0.1%	1	0 B	0.12	0.08
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00

Рис. 58. Глибокий профайлінг – 100 анімованих моделей з використанням розробленого скрипта для скінінгу

Методом виключення було виявлено, що додаткові виклики SetPass спричинені дуплікацією матеріалів. Модель, що використовувалася для тестування, має 28 окремих матеріалів. Скрипт SkinnedMeshRenderer

використовував посилання на один і той же об’єкт матеріалу для всіх 28 слотів, в той час, як розроблена реалізація скінінгу дуальними кватерніонами створювала окрему копію матеріалу для кожного слоту. Кожна зі 100 моделей виконувала окремий виклик SetPass для кожного з 28 матеріалів, що в сумі складає 2800 викликів, які і відображено на скріншоті профайлера (рис. 58).

Для запобігання копіюванню матеріалів, для отримання списку матеріалів моделі замість властивості materials компонента SkinnedMeshRenderer було використано властивість sharedMaterials. В документації рушія Unity вказано, що рекомендується не використовувати цю властивість, оскільки при зміні параметрів отриманих з неї матеріалів будуть змінені матеріали оригінальної моделі. В даному випадку оригінальний матеріал не використовується, оскільки компонент SkinnedMeshRenderer було відключено та замінено власною реалізацією скінінгу, тому використання sharedMaterials є виправданим.

Після переходу на використання SkinnedMeshRenderer.sharedMaterials замість SkinnedMeshRenderer.materials швидкодія розробленої реалізації суттєво зросла (рис. 59).

У вікні профайлера видно, що основну частину часу обробки кадру тепер займає виконання скриптів – 15.14 мс, а не безпосередньо рендер – 4.81 мс (рис. 60). Детальний профайлінг розробленого скрипта (рис. 61) вказує на те, що найбільше часу витрачається на виклики Transform.get_localToWorldMatrix( ) та String.memcpy( ).

Швидкість отримання Transform.localToWorldMatrix залежить від реалізації API рушія, відповідно знаходження виклику Transform.get_localToWorldMatrix( ) на першому місці по затратам часу свідчить про те, що скрипт добре оптимізовано та його швидкодія залежить в основному від реалізації API рушія.

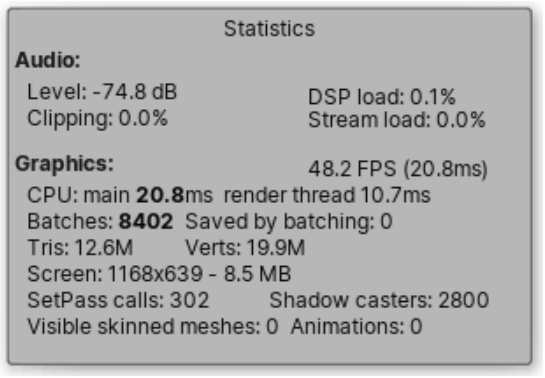


Рис. 59. Статистика рендеру кадру з 100 копіями моделі людини використовуючи розроблений скрипт без дублювання матеріалів

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	77.2%	0.2%	2	0 B	20.31	0.05
▶ Update.ScriptRunBehaviourUpdate	57.5%	0.0%	1	0 B	15.14	0.00
▶ Camera.Render	18.3%	0.1%	1	0 B	4.81	0.02
PostLateUpdate.MemoryFrameMaintenance	0.1%	0.1%	1	0 B	0.04	0.04
▶ PreUpdate.SendMouseEvents	0.1%	0.0%	1	0 B	0.04	0.00
UpdateScreenManagerAndInput	0.0%	0.0%	1	0 B	0.01	0.01
▶ PostLateUpdate.UpdateAudio	0.0%	0.0%	1	0 B	0.01	0.00
▶ UnityEngine.UIElementsModule.dll!UnityEngine.UIElemen	0.0%	0.0%	1	0 B	0.01	0.00
▶ UnityEngine.IMGUIModule.dll!UnityEngine::GUIUtility.Set	0.0%	0.0%	1	0 B	0.00	0.00
▶ Update.ScriptRunDelayedTasks	0.0%	0.0%	1	0 B	0.00	0.00
▶ GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
PostUpdateScreenManagerAndInput	0.0%	0.0%	1	0 B	0.00	0.00
▶ EarlyUpdate.ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
▶ PostLateUpdate.UpdateAllRenderers	0.0%	0.0%	1	0 B	0.00	0.00
▶ FrameEvents.NewInputBeforeRenderUpdate	0.0%	0.0%	1	0 B	0.00	0.00
▶ PlayerEndOfFrame	0.0%	0.0%	1	0 B	0.00	0.00
▶ EarlyUpdate.PlayerCleanupCachedData	0.0%	0.0%	1	0 B	0.00	0.00
▶ PreLateUpdate.ParticleSystemBeginUpdateAll	0.0%	0.0%	1	0 B	0.00	0.00
▶ FixedUpdate.PhysicsFixedUpdate	0.0%	0.0%	1	0 B	0.00	0.00
▶ FixedUpdate.NewInputFixedUpdate	0.0%	0.0%	1	0 B	0.00	0.00
Camera.FindStacks	0.0%	0.0%	2	0 B	0.00	0.00

Рис. 60. Глибокий профайлінг – 100 анімованих моделей з використанням розробленого скрипта для скінінгу без дублювання матеріалів

Виклик String.memcpy( ), що знаходиться на другому місці за затратами часу, є дещо неочікуваним, оскільки розроблений скрипт не використовує операції зі строками за винятком присвоєння значень параметрам матеріалів (рис. 62).

Методом виключення було встановлено, шо фреймворк Mono використовує метод String.memcpy( ) при копіюванні структур. Копіювання структур відбувається, коли в шейдер передаються матриці поз для кісток арматури (рис. 63).

Overview	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	2	0 B	20.27	0.06
▼ Update.ScriptRunBehaviourUpdate	1	0 B	15.12	0.00
▼ BehaviourUpdate	1	0 B	15.12	0.11
▼ Assembly-CSharp.dll!::DualQuaternionSkinner.Update()	100	0 B	15.00	0.07
▼ Assembly-CSharp.dll!::DualQuaternionSkinner.Update()	100	0 B	14.93	1.74
▶ UnityEngine.CoreModule.dll!UnityEngine::Transform.get_localToWorldMatrix()	8000	0 B	5.89	1.35
▶ mscorlib.dll!System::String.memcpy()	16100	0 B	4.39	1.66
▶ UnityEngine.CoreModule.dll!UnityEngine::ComputeBuffer.SetData()	100	0 B	0.74	0.03
▶ UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.Dispatch()	200	0 B	0.50	0.19
▶ Assembly-CSharp.dll!::DualQuaternionSkinner.get_mf()	400	0 B	0.27	0.04
▶ UnityEngine.CoreModule.dll!UnityEngine::MaterialPropertyBlock.SetTexture()	300	0 B	0.27	0.05
▶ UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.SetVector()	100	0 B	0.15	0.02
▶ Assembly-CSharp.dll!::DualQuaternionSkinner.get_mr()	200	0 B	0.14	0.02
UnityEngine.CoreModule.dll!UnityEngine::MeshFilter.get_mesh()	400	0 B	0.12	0.12
▶ UnityEngine.CoreModule.dll!UnityEngine::MaterialPropertyBlock.SetInt()	200	0 B	0.11	0.03
▶ UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.SetFloat()	100	0 B	0.10	0.02
▶ UnityEngine.CoreModule.dll!UnityEngine::Transform.get_worldToLocalMatrix()	100	0 B	0.10	0.02
▶ UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.SetMatrix()	100	0 B	0.09	0.02
▶ UnityEngine.CoreModule.dll!UnityEngine::Mesh.MarkDynamic()	100	0 B	0.06	0.02
▶ UnityEngine.CoreModule.dll!UnityEngine::Renderer.SetPropertyBlock()	100	0 B	0.04	0.01
UnityEngine.CoreModule.dll!UnityEngine::Component.get_transform()	100	0 B	0.04	0.04

Рис. 61. Глибокий профайлінг – 100 анімованих моделей з використанням розробленого скрипта для скінінгу без дублювання матеріалів (огляд розробленого скрипта)

```
this.shaderDQBlend.SetFloat("compensation_coef", this.bulgeCompensation);
```

Рис. 62. Присвоєння значення параметру матеріалу

Оскільки при передачі даних до шейдера не можна обійтись без копіювання структур, подальша оптимізація можлива лише за рахунок використання іншої реалізації фреймворка, що забезпечить більш високу швидкість для цієї операції. Використання компілятора IL2CPP дозволило

значно пришвидшити роботу скрипта (рис. 64).

Для максимально наближеного до реальних умов порівняння, було виконано вимір FPS скомпільованих додатків з відключеним налагодженням та профайлером (рис. 65). Для виміру FPS використовувалося програмне забезпечення RivaTuner Statistics Server. Було порівняно швидкодію розробленої реалізації скінінгу (рис. 65а), розробленої реалізації скінінгу з відключеною компенсацією дефекту роздутого суглоба (рис. 65б) та вбудованого лінійного скінінгу Unity (рис. 65в).

```
// Update is called once per frame
void Update()
{
    if (this.mr.isVisible == false)
    {
        return;
    }

    this.mf.mesh.MarkDynamic();

    for (int i = 0; i < this.bones.Length; i++)
    {
        this.poseMatrices[i] = this.bones[i].localToWorldMatrix;    // this calls String.memcpy()
    }

    this.bufPoseMatrices.SetData(this.poseMatrices);
```

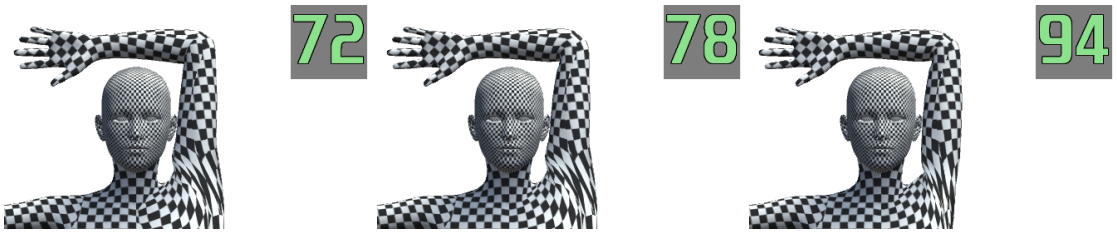
Рис. 63. Фрагмент коду, що викликає метод String.memcpy( ) через копіювання структур

За отриманими даними (рис. 65) власна реалізація скінінгу дуальними кватерніонами повільніша за вбудований лінійний скінінг на 20%, що є непоганим результатом – в [8] вказано час виконання лінійного скінінгу та скінінгу дуальними кватерніонами 0.32 мс та 0.46 мс відповідно (рис. 66), що дає різницю в 44%.

Включення компенсації дефекту роздутого суглоба сповільнює скрипт приблизно на 8%.

Overview	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	1	0 B	16.93	0.02
PostLateUpdate.FinishFrameRendering	1	0 B	9.42	0.01
Update.ScriptRunBehaviourUpdate	1	0 B	7.25	0.00
BehaviourUpdate	1	0 B	7.25	0.06
Assembly-CSharp.dll!::DualQuaternionSkinner.Update()	100	0 B	7.18	0.02
Assembly-CSharp.dll!::DualQuaternionSkinner.Update()	100	0 B	7.15	0.76
UnityEngine.CoreModule.dll!UnityEngine::Transform.get_localToWorldMatrix()	8000	0 B	3.75	0.94
UnityEngine.CoreModule.dll!UnityEngine::ComputeBuffer.SetData()	100	0 B	0.72	0.04
UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.Dispatch()	200	0 B	0.34	0.12
Assembly-CSharp.dll!::DualQuaternionSkinner.get_mf()	400	0 B	0.27	0.06
UnityEngine.CoreModule.dll!UnityEngine::MaterialPropertyBlock.SetTexture()	300	0 B	0.27	0.05
UnityEngine.CoreModule.dll!UnityEngine::MaterialPropertyBlock.SetInt()	200	0 B	0.17	0.03
UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.SetFloat()	100	0 B	0.15	0.01
UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.SetVector()	100	0 B	0.13	0.02
Assembly-CSharp.dll!::DualQuaternionSkinner.get_mr()	200	0 B	0.13	0.02
UnityEngine.CoreModule.dll!UnityEngine::MeshFilter.get_mesh()	400	0 B	0.09	0.09
UnityEngine.CoreModule.dll!UnityEngine::ComputeShader.SetMatrix()	100	0 B	0.08	0.01
UnityEngine.CoreModule.dll!UnityEngine::Transform.get_worldToLocalMatrix(	100	0 B	0.04	0.01
UnityEngine.CoreModule.dll!UnityEngine::Mesh.MarkDynamic()	100	0 B	0.04	0.01
UnityEngine.CoreModule.dll!UnityEngine::Mesh.get_vertexCount()	300	0 B	0.03	0.03
UnityEngine.CoreModule.dll!UnityEngine::Renderer.SetPropertyBlock()	100	0 B	0.03	0.00
UnityEngine.CoreModule.dll!UnityEngine::Component.get_transform()	100	0 B	0.03	0.03
UnityEngine.CoreModule.dll!UnityEngine::Renderer.get_isVisible()	100	0 B	0.02	0.02
UnityEngine.CoreModule.dll!UnityEngine::Vector4.op_Implicit()	100	0 B	0.01	0.01

Рис. 64. Глибокий профайлінг – 100 анімованих моделей з використанням розробленого скрипта (компілятор IL2CPP)



(а) (б) (в)

Рис. 65. Бенчмарк розробленого скрипта з компілятором IL2CPP (а), розробленого скрипта з компілятором IL2CPP та відключеною компенсацією дефекту роздутого суглоба (б), вбудованого скінінгу (в)

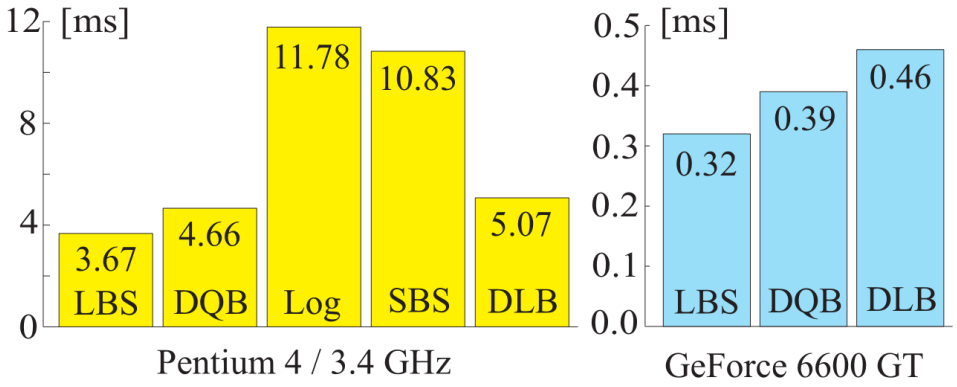


Рис. 66. Порівняння швидкодії методів скінінгу [8]. LBS — linear blend skinning, DQB — direct quaternion blending, Log — log-matrix blending, SBS — spherical blend skinning, DLB — dual quaternion linear blending

4.4. Аналіз отриманих деформацій

Розроблений метод значно зменшує дефект роздутого суглоба в ситуаціях, коли цей дефект є найбільш помітним, а саме – при згині трубоподібної частини моделі (руки, ноги, тощо) (рис. 67).

(а) (б) (в)

Рис. 67. Модель руки, зігнута в ліктьовому суглобі, з використанням лінійного скінінгу (а), скінінгу дуальними кватерніонами (б), та скінінгу дуальними кватерніонами з розробленим методом компенсації дефекту роздутого суглоба (в)

У зонах впливу трьох та більше кісток, а також у зонах, що підлягають одночасно деформації згину та скручування, розроблений метод плавно зменшує інтенсивність компенсації дефектів. Такий підхід запобігає утворенню розривів та різких перепадів (рис. 68).

(а) (б) (в)

Рис. 68. Модель циліндра, комбінована анімація скручування та згину, з використанням лінійного скінінгу (а), скінінгу дуальними кватерніонами (б), та скінінгу дуальними кватерніонами з розробленим методом компенсації дефекту роздутого суглоба (в)

ВИСНОВКИ

Головною метою дисертації була розроблення методу компенсації небажаних деформацій, що виникають при використанні DQ-скінінгу.

В ході роботи було розроблено метод пост-обробки моделі, що дозволяє компенсувати дефект роздутого суглоба за рахунок апроксимації небажаної деформації на основі вагових коефіцієнтів вертексів моделі та напрямних векторів кісток.

Розроблений метод значно покращує якість анімації у ситуаціях, де дефект роздутого суглоба є найбільш помітним, але ігнорує більш складні випадки, а саме – області моделі, що керуються більш, ніж двома кістками, та деформації, що є комбінацією згину та скручування.

Розроблений метод робить ряд припущень про анімовану модель, тому не є універсальним, проте у переважній більшості випадків ці припущення є достатньо точними, щоб використання запропонованого методу дозволило помітно покращити якість анімації.

Для демонстрації та тестування запропонованого методу було розроблено його реалізації у вигляді плагіну для рушія Unity. Даний плагін є сумісним з бібліотеками OpenGL, DirectX, Vulkan та Metal, підтримує використання ключових форм, виконує розрахунки з використанням розрахункових шейдерів, автоматично визначає та виправляє розповсюджені помилки налаштування.

Емпіричні виміри швидкодії розробленої реалізації показали, що без використання компенсації небажаних деформацій розроблена реалізація DQ-скінінгу є та 20% повільнішою за вбудований лінійний скінінг Unity, а використання компенсації небажаних деформацій сповільнює скінінг ще на 8%. Порівняно з альтернативними методами скінінгу, різниця швидкості виконання у 8% є незначною, а отже використання запропонованого методу компенсації небажаних деформацій є цілком виправданим.

В подальшому планується розроблення більш детальної апроксимації небажаних деформацій, включно з областями, що керуються більш, ніж двома кістками, та комбінованими деформаціями згину та скручування.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Morphing Serious to Smiling Face [Електронний ресурс] – Режим доступу: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.407.9073&rep=rep1&type=pdf#page=15. – Дата доступу: січень 2020. – Назва з екрана.
2. File:Sintel-hand.png [Електронний ресурс] – Режим доступу: https://commons.wikimedia.org/wiki/File:Sintel-hand.png. – Дата доступу: січень 2020. – Назва з екрана.
3. Dionne. Geodesic voxel binding for production character meshes. [Text] / Dionne, Olivier, and Martin de Lasa. – InProceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2013 Jul 19 – pp. 173-180.
4. Hétroy, F. Simple flexible skinning based on manifold modeling. / Franck Hétroy, Cédric Gérot, Lin Lu and Boris Thibert. – 2009.
5. Voxel Heat Diffuse Skinning [Електронний ресурс] – Режим доступу: https://blendermarket.com/products/voxel-heat-diffuse-skinning. – Дата доступу: січень 2020. – Назва з екрана.
6. Dual Quaternions skinning tutorial and C++ codes [Електронний ресурс] – Режим доступу: http://rodolphe-vaillant.fr/?e=29. – Дата доступу: січень 2020. – Назва з екрана.
7. A Beginners Guide to Dual-Quaternions [Електронний ресурс] – 2019. – Режим доступу: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.407.9073&rep=rep1&type=pdf#page=15 – Дата доступу: 25.2.2019 – Назва з екрану.
8. Kavan, L. Geometric skinning with approximate dual quaternion blending [Text] / Ladislay Kavan, Steven Collins, Jiří Žára, Carol O'Sullivan // New York, USA: ACM Transactions on Graphics (TOG), Volume 27 Issue 4, October 2008, Article No. 105.
9. Kavan, L. Skinning with dual quaternions [Text] / Ladislay Kavan, Steven

Collins, Jiří Žára, Carol O'Sullivan // Seattle, Washington: I3D '07 Proceedings of the 2007 symposium on Interactive 3D graphics and games, April 30 – May 02, 2007. – pp. 39–46.

10. Dual Quaternions for Rigid Transformation Blending [Електронний ресурс] – 2019. – Режим доступу: https://www.scss.tcd.ie/publications/tech-reports/reports.06/TCD-CS-2006-46.pdf – Дата доступу: 25.2.2019 – Назва з екрану.
11. YoungBeom, K. Bulging-free dual quaternion skinning [Text] / Kim YoungBeom, Han JungHyun // Computer Animation and Virtual Worlds, Volume 25, Issue 3–4, 16 May 2014. – pp. 321-329.
12. Enhanced Dual Quaternion Skinning for Production Use [Електронний ресурс] – 2019. – Режим доступу: https://www.researchgate.net/profile/Mark_Mclaughlin13/publication/262284169_Enhanced_dual_quaternion_skinning_for_production_use/links/581372b708aedc7d8961e1f6/Enhanced-dual-quaternion-skinning-for-production-use.pdf – Дата доступу: 25.2.2019 – Назва з екрану.
13. Binh Huy Le Real-time skeletal skinning with optimized centers of rotation [Text] / Binh Huy Le, Jessica K. Hodgins // New York, USA: ACM Transactions on Graphics (TOG), Volume 35 Issue 4, July 2016, Article No. 37.
14. Merry, B. Animation space: A truly linear framework for character animation. / Bruce Merry, Patrick Marais, and James Gain // ACM Transactions on Graphics (TOG) 25(4). – pp. 1400-23.
15. Jacka, D. A comparison of linear skinning techniques for character animation / David Jacka, Ashley Reid, Bruce Merry, James Gain // New York, USA: AFRIGRAPH '07 Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa. – pp. 177-186.
16. Vaillant, R. Implicit skinning: real-time skin deformation with contact

modeling [Text] / Rodolphe Vaillant, Loïc Barthe, Gaël Guennebaud, Marie-Paule Cani, Damien Rohmer, Brian Wyvill, Olivier Gourmel, Mathias Paulin // New York, USA: ACM Transactions on Graphics (TOG), Volume 27 Issue 4, October 2008, Article No. 105.

17. Lagae, A. A comparison of methods for generating Poisson disk distributions. / Ares Lagae, Philip Dutré // InComputer Graphics Forum 2008 Mar (Vol. 27, No. 1, pp. 114-129). Oxford, UK: Blackwell Publishing Ltd.
18. Galoppo, N. Controlling deformable material with dynamic morph targets. / Nico Galoppo, Miguel A. Otaduy, William Moss, Jason Sewall, Sean Curtis, and Ming C. Lin // InProceedings of the 2009 symposium on Interactive 3D graphics and games 2009 Feb 27. – pp. 39-47.
19. Corrective Blendshapes [Електронний ресурс] – Режим доступу: https: //skeletonsociety.wordpress.com/2016/11/18/corrective-blendshapes/. – Дата доступу: січень 2020. – Назва з екрана.
20. Introduction to rigging in Maya - Part 11 - Corrective blendshapes [Електронний ресурс] – Режим доступу: https://3dtotal.com/tutorials/t/maya-rigging-corrective-blendshapes-jahirul-amin-animation-blend-shapes. – Дата доступу: січень 2020. – Назва з екрана.
21. Weight Paint [Електронний ресурс] – Режим доступу: https://mmorley.hatenablog.com/entry/2016/03/28/234354. – Дата доступу: січень 2020. – Назва з екрана.
22. Editing Weight [Електронний ресурс] – Режим доступу: https://docs.blender.org/manual/en/latest/sculpt_paint/weight_paint/editing.html. – Дата доступу: січень 2020. – Назва з екрана.
23. Gaussian Blur - ARM / Aran Nolan [Електронний ресурс] – Режим доступу: https://www.arannolan.com/documents/gaussian-blur.pdf. – Дата доступу: січень 2020. – Назва з екрана.

ДОДАТКИ

Додаток 1

Лістинги коду програмної реалізації алгоритмічно-програмного методу компенсації дефектів DQ-скінінгу

Лістинг 3. Розширення класу Matrix4x4

```
using UnityEngine;

public static class MatrixExtensions
{
  public static Quaternion ExtractRotation(this Matrix4x4 matrix)
  {
    return Quaternion.LookRotation(matrix.GetColumn(2),
matrix.GetColumn(1));
  }

  public static Vector3 ExtractPosition(this Matrix4x4 matrix)
  {
    Vector4 position;
    position = matrix.GetColumn(3);
    position.w = 1;
    return position;
  }

  public static Vector3 ExtractScale(this Matrix4x4 matrix)
  {
    Vector3 scale;
    scale.x = new Vector4(matrix.m00, matrix.m10, matrix.m20,
matrix.m30).magnitude;
    scale.y = new Vector4(matrix.m01, matrix.m11, matrix.m21,
matrix.m31).magnitude;
    scale.z = new Vector4(matrix.m02, matrix.m12, matrix.m22,
matrix.m32).magnitude;
    return scale;
  }
}
```

Лістинг 4. Розширення класу Quaternion

```
using UnityEngine;

public static class QuaternionExtensions
{
  public static Vector4 ToVector4(this Quaternion quaternion)
  {
    return new Vector4(quaternion.x, quaternion.y, quaternion.z,
quaternion.w);
  }
}
```

Лістинг 5. Препроцесор імпортованих моделей

```
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
using System.Linq;

#if UNITY_EDITOR

/// <summary>
/// Sorts bone indexes in imported meshes.<br>
/// SkinnedMeshRenderer requires bone indexes to be sorted based on
hierarchy.
/// </summary>
public class AssetPostProcessorReorderBones : AssetPostprocessor
{
  void OnPostprocessModel(GameObject g)
  {
        this.Process(g);
  }

  void Process(GameObject g)
  {
    SkinnedMeshRenderer smr =
g.GetComponentInChildren<SkinnedMeshRenderer>();
```

```
    if (smr == null)
    {
      Debug.LogWarning("Unable to find Renderer" + smr.name);
      return;
    }

    //list of bones
    List<Transform> boneTransforms = smr.bones.ToList();

    //sort based on hierarchy
    boneTransforms.Sort(CompareTransform);

    //record bone index mappings (richardf advice)
    //build a Dictionary<int, int> that records the old bone index => new
bone index mappings,
    //then run through every vertex and just do boneIndexN =
dict[boneIndexN] for each weight on each vertex.
    var remap = new Dictionary<int, int>();
    for (int i = 0; i < smr.bones.Length; i++)
    {
      remap[i] = boneTransforms.IndexOf(smr.bones[i]);
    }

    //remap bone weight indexes
    BoneWeight[] bw = smr.sharedMesh.boneWeights;
    for (int i = 0; i < bw.Length; i++)
    {
      bw[i].boneIndex0 = remap[bw[i].boneIndex0];
      bw[i].boneIndex1 = remap[bw[i].boneIndex1];
      bw[i].boneIndex2 = remap[bw[i].boneIndex2];
      bw[i].boneIndex3 = remap[bw[i].boneIndex3];
    }

    //remap bindposes
    var bp = new Matrix4x4[smr.sharedMesh.bindposes.Length];
    for (int i = 0; i < bp.Length; i++)
    {
      bp[remap[i]] = smr.sharedMesh.bindposes[i];
```

```
    }

    //assign new data
    smr.bones = boneTransforms.ToArray();
    smr.sharedMesh.boneWeights = bw;
    smr.sharedMesh.bindposes = bp;
  }

  private static int CompareTransform(Transform A, Transform B)
  {
    if (B.IsChildOf(A))
        {
            return -1;
        }

        if (A.IsChildOf(B))
        {
            return -1;
        }

        return 0;
  }
}

#endif
```

Лістинг 6. Скрипт DualQuaternionSkinner

```
using UnityEngine;

/// <summary>
/// Replaces Unity's default linear skinning with DQ skinning
///
/// Add this component to a <a class="bold"
href="https://docs.unity3d.com/ScriptReference/GameObject.html">GameObject<
/a> that has <a class="bold"
href="https://docs.unity3d.com/ScriptReference/SkinnedMeshRenderer.html">Sk
innedMeshRenderer</a> attached.<br>
```

```
/// Do not remove <a class="bold"
href="https://docs.unity3d.com/ScriptReference/SkinnedMeshRenderer.html">Sk
innedMeshRenderer</a> component!<br>
/// Make sure that all materials of the animated object are using shader
\"<b>MadCake/Material/Standard hacked for DQ skinning</b>\"
/// </summary>
[RequireComponent(typeof(MeshFilter))]
public class DualQuaternionSkinner : MonoBehaviour
{
  /// <summary>
  /// Bone orientation is required for bulge-compensation.<br>
  /// Do not set directly, use custom editor instead.
  /// </summary>
  public Vector3 boneOrientationVector = Vector3.up;

  /// <summary>
  /// Do not edit directly, use SetViewFrustrumCulling() instead.
  /// </summary>
  public bool viewFrustrumCulling = true;

  struct VertexInfo
  {
    // could use float3 instead of float4 but NVidia says structures not
aligned to 128 bits are slow
    // https://developer.nvidia.com/content/understanding-structured-
buffer-performance

    public Vector4 position;
    public Vector4 normal;
    public Vector4 tangent;

    public int boneIndex0;
    public int boneIndex1;
    public int boneIndex2;
    public int boneIndex3;

    public float weight0;
    public float weight1;
```

```
    public float weight2;
    public float weight3;

    public float compensation_coef;
  }

  struct MorphDelta
  {
    // could use float3 instead of float4 but NVidia says structures not
aligned to 128 bits are slow
    // https://developer.nvidia.com/content/understanding-structured-
buffer-performance

    public Vector4 position;
    public Vector4 normal;
    public Vector4 tangent;
  }

  struct DualQuaternion
  {
    public Quaternion rotationQuaternion;
    public Vector4 position;
  }

  const int numthreads = 1024;    // must be same in compute shader code
  const int textureWidth = 1024;  // no need to adjust compute shaders

  /// <summary>
  /// Adjusts the amount of bulge-compensation.
  /// </summary>
  [Range(0,1)]
  public float bulgeCompensation  = 0;

  public ComputeShader shaderComputeBoneDQ;
  public ComputeShader shaderDQBlend;
  public ComputeShader shaderApplyMorph;

  /// <summary>
```

```
/// Indicates whether DualQuaternionSkinner is currently active.
/// </summary>
public bool started { get; private set; } = false;

Matrix4x4[] poseMatrices;

ComputeBuffer bufPoseMatrices;
ComputeBuffer bufSkinnedDq;
ComputeBuffer bufBindDq;

ComputeBuffer bufVertInfo;
ComputeBuffer bufMorphTemp_1;
ComputeBuffer bufMorphTemp_2;

ComputeBuffer bufBoneDirections;

ComputeBuffer[] arrBufMorphDeltas;

float[] morphWeights;

MeshFilter mf
{
  get
  {
    if (this._mf == null)
    {
      this._mf = this.GetComponent<MeshFilter>();
    }

    return this._mf;
  }
}
MeshFilter _mf;

MeshRenderer mr
{
  get
  {
```

```
      if (this._mr == null)
      {
        this._mr = this.GetComponent<MeshRenderer>();
        if (this._mr == null)
        {
          this._mr = this.gameObject.AddComponent<MeshRenderer>();
        }
      }

      return this._mr;
    }
  }
  MeshRenderer _mr;

  SkinnedMeshRenderer smr
  {
    get
    {
      if (this._smr == null)
      {
        this._smr = this.GetComponent<SkinnedMeshRenderer>();
      }

      return this._smr;
    }
  }
  SkinnedMeshRenderer _smr;

  MaterialPropertyBlock materialPropertyBlock;

  Transform[] bones;
  Matrix4x4[] bindPoses;

  /*
    Vulkan and OpenGL only support ComputeBuffer in compute shaders
    passing data to the vertex and fragment shaders is done through
RenderTextures
```

```
    using ComputeBuffers would improve the efficiency slightly but it would
only work with Dx11

    layout is as such:
      rtSkinnedData_1      float4      vertex.xyz,  normal.x
      rtSkinnedData_2      float4      normal.yz,  tangent.xy
      rtSkinnedData_3      float2      tangent.zw
  */
  RenderTexture rtSkinnedData_1;
  RenderTexture rtSkinnedData_2;
  RenderTexture rtSkinnedData_3;

  int kernelHandleComputeBoneDQ;
  int kernelHandleDQBlend;
  int kernelHandleApplyMorph;

  public void SetViewFrustrumCulling(bool viewFrustrumculling)
  {
    if (this.viewFrustrumCulling == viewFrustrumculling)
      return;

    this.viewFrustrumCulling = viewFrustrumculling;

    if (this.started == true)
      UpdateViewFrustrumCulling();
  }

  void UpdateViewFrustrumCulling()
  {
    if (this.viewFrustrumCulling)
      this.mf.mesh.bounds = this.smr.localBounds;
    else
      this.mf.mesh.bounds = new Bounds(Vector3.zero, Vector3.one *
100000000);
  }

  /// <summary>
  /// Returns an array of currently applied blend shape weights.<br>
```

```
  /// Default range is 0-100.<br>
  /// It is possible to apply negative weights or exceeding 100.
  /// </summary>
  /// <returns>Array of currently applied blend shape weights</returns>
  public float[] GetBlendShapeWeights()
  {
    float[] weights = new float[this.morphWeights.Length];
    for (int i = 0; i < weights.Length; i++)
        {
            weights[i] = this.morphWeights[i];
        }

        return weights;
  }

  /// <summary>
  /// Applies blend shape weights from the given array.<br>
  /// Default range is 0-100.<br>
  /// It is possible to apply negative weights or exceeding 100.
  /// </summary>
  /// <param name="weights">An array of weights to be applied</param>
  public void SetBlendShapeWeights(float[] weights)
  {
    if (weights.Length != this.morphWeights.Length)
        {
            throw new System.ArgumentException(
        "An array of weights must contain the number of elements " +
        $"equal to the number of available blendshapes. Currently " +
        $"{this.morphWeights.Length} blendshapes ara available but
{weights.Length} weights were passed."
      );
        }

    for (int i = 0; i < weights.Length; i++)
        {
            this.morphWeights[i] = weights[i];
        }
```

```
    this.ApplyMorphs();
    }

  /// <summary>
  /// Set weight for the blend shape with given index.<br>
  /// Default range is 0-100.<br>
  /// It is possible to apply negative weights or exceeding 100.
  /// </summary>
  /// <param name="index">Index of the blend shape</param>
  /// <param name="weight">Weight to be applied</param>
  public void SetBlendShapeWeight(int index, float weight)
  {
    if (this.started == false)
    {
      this.GetComponent<SkinnedMeshRenderer>().SetBlendShapeWeight(index,
weight);
      return;
    }

    if (index < 0 || index >= this.morphWeights.Length)
        {
            throw new System.IndexOutOfRangeException("Blend shape index
out of range");
        }

        this.morphWeights[index] = weight;

    this.ApplyMorphs();
  }

  /// <summary>
  /// Returns currently applied weight for the blend shape with given
index.<br>
  /// Default range is 0-100.<br>
  /// It is possible to apply negative weights or exceeding 100.
  /// </summary>
  /// <param name="index">Index of the blend shape</param>
  /// <returns>Currently applied weight</returns>
```

```
  public float GetBlendShapeWeight(int index)
  {
    if (this.started == false)
        {
            return
this.GetComponent<SkinnedMeshRenderer>().GetBlendShapeWeight(index);
        }

        if (index < 0 || index >= this.morphWeights.Length)
        {
            throw new System.IndexOutOfRangeException("Blend shape index
out of range");
        }

        return this.morphWeights[index];
  }

  /// <summary>
  /// UnityEngine.<a class="bold"
href="https://docs.unity3d.com/ScriptReference/Mesh.html">Mesh</a> that is
currently being rendered.
  /// @see <a class="bold"
href="https://docs.unity3d.com/ScriptReference/Mesh.GetBlendShapeName.html"
>Mesh.GetBlendShapeName(int shapeIndex)</a>
  /// @see <a class="bold"
href="https://docs.unity3d.com/ScriptReference/Mesh.GetBlendShapeIndex.html
">Mesh.GetBlendShapeIndex(string blendShapeName)</a>
  /// @see <a class="bold"
href="https://docs.unity3d.com/ScriptReference/Mesh-
blendShapeCount.html">Mesh.blendShapeCount</a>
  /// </summary>
  public Mesh mesh
  {
    get
    {
      if (this.started == false)
            {
                return this.smr.sharedMesh;
```

```
            }

            return this.mf.mesh;
        }
    }

  /// <summary>
  /// If the value of boneOrientationVector was changed while
DualQuaternionSkinner is active (started == true),
UpdatePerVertexCompensationCoef() must be called in order for the change to
take effect.
  /// </summary>
  public void UpdatePerVertexCompensationCoef()
  {
    var vertInfos = new VertexInfo[this.mf.mesh.vertexCount];
    this.bufVertInfo.GetData(vertInfos);

    for (int i = 0; i < vertInfos.Length; i++)
    {
      Matrix4x4 bindPose = this.bindPoses[vertInfos[i].boneIndex0].inverse;
            Quaternion   boneBindRotation = bindPose.ExtractRotation();
      Vector3 boneDirection = boneBindRotation *
this.boneOrientationVector;  // ToDo figure out bone orientation
      Vector3 bonePosition = bindPose.ExtractPosition();
      Vector3 toBone = bonePosition - (Vector3)vertInfos[i].position;

      vertInfos[i].compensation_coef = Vector3.Cross(toBone,
boneDirection).magnitude;
    }

    this.bufVertInfo.SetData(vertInfos);
    this.ApplyMorphs();
  }

  void GrabMeshFromSkinnedMeshRenderer()
  {
    this.ReleaseBuffers();
```

```
    this.mf.mesh = this.smr.sharedMesh;
    this.bindPoses = this.mf.mesh.bindposes;

    this.arrBufMorphDeltas = new
ComputeBuffer[this.mf.mesh.blendShapeCount];

    this.morphWeights = new float[this.mf.mesh.blendShapeCount];

    var deltaVertices = new Vector3[this.mf.mesh.vertexCount];
    var deltaNormals = new Vector3[this.mf.mesh.vertexCount];
    var deltaTangents = new Vector3[this.mf.mesh.vertexCount];

    var deltaVertInfos = new MorphDelta[this.mf.mesh.vertexCount];

    for (int i = 0; i < this.mf.mesh.blendShapeCount; i++)
    {
      this.mf.mesh.GetBlendShapeFrameVertices(i, 0, deltaVertices,
deltaNormals, deltaTangents);

      this.arrBufMorphDeltas[i] = new
ComputeBuffer(this.mf.mesh.vertexCount, sizeof(float) * 12);

      for (int k = 0; k < this.mf.mesh.vertexCount; k++)
      {
        deltaVertInfos[k].position  = deltaVertices  != null ?
deltaVertices[k]  : Vector3.zero;
        deltaVertInfos[k].normal  = deltaNormals  != null ? deltaNormals[k]
: Vector3.zero;
        deltaVertInfos[k].tangent  = deltaTangents  != null ?
deltaTangents[k]  : Vector3.zero;
      }

      this.arrBufMorphDeltas[i].SetData(deltaVertInfos);
    }

    Material[] materials = this.smr.sharedMaterials;
    for (int i = 0; i < materials.Length; i++)
    {
```

```
      materials[i].SetInt("_DoSkinning", 1);
    }
    this.mr.materials = materials;

    this.shaderDQBlend.SetInt("textureWidth", textureWidth);

    this.poseMatrices = new Matrix4x4[this.mf.mesh.bindposes.Length];

    // initiate textures and buffers

    int textureHeight = this.mf.mesh.vertexCount / textureWidth;
    if (this.mf.mesh.vertexCount % textureWidth != 0)
    {
      textureHeight++;
    }

        this.rtSkinnedData_1 = new RenderTexture(textureWidth,
textureHeight, 0, RenderTextureFormat.ARGBFloat)
        {
            filterMode = FilterMode.Point,
            enableRandomWrite = true
        };
        this.rtSkinnedData_1.Create();
    this.shaderDQBlend.SetTexture(this.kernelHandleDQBlend,
"skinned_data_1", this.rtSkinnedData_1);

        this.rtSkinnedData_2 = new RenderTexture(textureWidth,
textureHeight, 0, RenderTextureFormat.ARGBFloat)
        {
            filterMode = FilterMode.Point,
            enableRandomWrite = true
        };
        this.rtSkinnedData_2.Create();
    this.shaderDQBlend.SetTexture(this.kernelHandleDQBlend,
"skinned_data_2", this.rtSkinnedData_2);

        this.rtSkinnedData_3 = new RenderTexture(textureWidth,
textureHeight, 0, RenderTextureFormat.RGFloat)
```

```
        {
            filterMode = FilterMode.Point,
            enableRandomWrite = true
        };
        this.rtSkinnedData_3.Create();
    this.shaderDQBlend.SetTexture(this.kernelHandleDQBlend,
"skinned_data_3", this.rtSkinnedData_3);

    this.bufPoseMatrices = new ComputeBuffer(this.mf.mesh.bindposes.Length,
sizeof(float) * 16);
    this.shaderComputeBoneDQ.SetBuffer(this.kernelHandleComputeBoneDQ,
"pose_matrices", this.bufPoseMatrices);

    this.bufSkinnedDq = new ComputeBuffer(this.mf.mesh.bindposes.Length,
sizeof(float) * 8);
    this.shaderComputeBoneDQ.SetBuffer(this.kernelHandleComputeBoneDQ,
"skinned_dual_quaternions", this.bufSkinnedDq);
    this.shaderDQBlend.SetBuffer(this.kernelHandleDQBlend,
"skinned_dual_quaternions", this.bufSkinnedDq);

    this.bufBoneDirections = new
ComputeBuffer(this.mf.mesh.bindposes.Length, sizeof(float) * 4);
    this.shaderComputeBoneDQ.SetBuffer(this.kernelHandleComputeBoneDQ,
"bone_directions", this.bufBoneDirections);
    this.shaderDQBlend.SetBuffer(this.kernelHandleDQBlend,
"bone_directions", this.bufBoneDirections);

    this.bufVertInfo = new ComputeBuffer(this.mf.mesh.vertexCount,
sizeof(float) * 16 + sizeof(int) * 4 + sizeof(float));
    var vertInfos = new VertexInfo[this.mf.mesh.vertexCount];
    Vector3[] vertices = this.mf.mesh.vertices;
    Vector3[] normals = this.mf.mesh.normals;
    Vector4[] tangents = this.mf.mesh.tangents;
    BoneWeight[] boneWeights = this.mf.mesh.boneWeights;
    for (int i = 0; i < vertInfos.Length; i++)
    {
      vertInfos[i].position = vertices[i];
```

```
      vertInfos[i].boneIndex0 = boneWeights[i].boneIndex0;
      vertInfos[i].boneIndex1 = boneWeights[i].boneIndex1;
      vertInfos[i].boneIndex2 = boneWeights[i].boneIndex2;
      vertInfos[i].boneIndex3 = boneWeights[i].boneIndex3;

      vertInfos[i].weight0 = boneWeights[i].weight0;
      vertInfos[i].weight1 = boneWeights[i].weight1;
      vertInfos[i].weight2 = boneWeights[i].weight2;
      vertInfos[i].weight3 = boneWeights[i].weight3;

      // determine per-vertex compensation coef

      Matrix4x4 bindPose = this.bindPoses[vertInfos[i].boneIndex0].inverse;
            Quaternion   boneBindRotation = bindPose.ExtractRotation();
      Vector3 boneDirection = boneBindRotation *
this.boneOrientationVector;  // ToDo figure out bone orientation
      Vector3 bonePosition = bindPose.ExtractPosition();
      Vector3 toBone = bonePosition - (Vector3)vertInfos[i].position;

      vertInfos[i].compensation_coef = Vector3.Cross(toBone,
boneDirection).magnitude;
    }

    if (normals.Length > 0)
    {
      for (int i = 0; i < vertInfos.Length; i++)
      {
        vertInfos[i].normal = normals[i];
      }
    }

    if (tangents.Length > 0)
    {
      for (int i = 0; i < vertInfos.Length; i++)
      {
        vertInfos[i].tangent = tangents[i];
      }
    }
```

```
    this.bufVertInfo.SetData(vertInfos);
    this.shaderDQBlend.SetBuffer(this.kernelHandleDQBlend, "vertex_infos",
this.bufVertInfo);

    this.bufMorphTemp_1 = new ComputeBuffer(this.mf.mesh.vertexCount,
sizeof(float) * 16 + sizeof(int) * 4);
    this.bufMorphTemp_2 = new ComputeBuffer(this.mf.mesh.vertexCount,
sizeof(float) * 16 + sizeof(int) * 4);

    // bind DQ buffer

    Matrix4x4[] bindPoses = this.mf.mesh.bindposes;
    var bindDqs = new DualQuaternion[bindPoses.Length];
    for (int i = 0; i < bindPoses.Length; i++)
    {
      bindDqs[i].rotationQuaternion  = bindPoses[i].ExtractRotation();
      bindDqs[i].position        = bindPoses[i].ExtractPosition();
    }

    this.bufBindDq = new ComputeBuffer(bindDqs.Length, sizeof(float) * 8);
    this.bufBindDq.SetData(bindDqs);
    this.shaderComputeBoneDQ.SetBuffer(this.kernelHandleComputeBoneDQ,
"bind_dual_quaternions", this.bufBindDq);

    this.UpdateViewFrustrumCulling();
    this.ApplyMorphs();
  }

  void ApplyMorphs()
  {
    ComputeBuffer bufMorphedVertexInfos = this.GetMorphedVertexInfos(
      this.bufVertInfo,
      ref this.bufMorphTemp_1,
      ref this.bufMorphTemp_2,
      this.arrBufMorphDeltas,
      this.morphWeights
    );
```

```
    this.shaderDQBlend.SetBuffer(this.kernelHandleDQBlend, "vertex_infos",
bufMorphedVertexInfos);
  }

  ComputeBuffer GetMorphedVertexInfos(ComputeBuffer bufOriginal, ref
ComputeBuffer bufTemp_1, ref ComputeBuffer bufTemp_2, ComputeBuffer[]
arrBufDelta, float[] weights)
  {
    ComputeBuffer bufSource = bufOriginal;

    for (int i = 0; i < weights.Length; i++)
    {
      if (weights[i] == 0)
            {
                continue;
            }

            if (arrBufDelta[i] == null)
            {
                throw new System.NullReferenceException();
            }

            this.shaderApplyMorph.SetBuffer(this.kernelHandleApplyMorph,
"source", bufSource);
      this.shaderApplyMorph.SetBuffer(this.kernelHandleApplyMorph,
"target", bufTemp_1);
      this.shaderApplyMorph.SetBuffer(this.kernelHandleApplyMorph, "delta",
arrBufDelta[i]);
      this.shaderApplyMorph.SetFloat("weight", weights[i] / 100f);

      int numThreadGroups = bufSource.count / numthreads;
      if (bufSource.count % numthreads != 0)
      {
        numThreadGroups++;
      }
```

```
      this.shaderApplyMorph.Dispatch(this.kernelHandleApplyMorph,
numThreadGroups, 1, 1);

      bufSource = bufTemp_1;
      bufTemp_1 = bufTemp_2;
      bufTemp_2 = bufSource;
    }

    return bufSource;
  }

  void ReleaseBuffers()
  {
    this.bufBindDq?.Release();
    this.bufPoseMatrices?.Release();
    this.bufSkinnedDq?.Release();

    this.bufVertInfo?.Release();
    this.bufMorphTemp_1?.Release();
    this.bufMorphTemp_2?.Release();

    this.bufBoneDirections?.Release();

    if (this.arrBufMorphDeltas != null)
    {
      for (int i = 0; i < this.arrBufMorphDeltas.Length; i++)
      {
        this.arrBufMorphDeltas[i]?.Release();
      }
    }
  }
  void OnDestroy()
  {
    this.ReleaseBuffers();
  }

  // Use this for initialization
  void Start()
```

```
  {
    this.materialPropertyBlock = new MaterialPropertyBlock();

    this.shaderComputeBoneDQ =
(ComputeShader)Instantiate(this.shaderComputeBoneDQ);    // bug workaround
    this.shaderDQBlend = (ComputeShader)Instantiate(this.shaderDQBlend);
// bug workaround
    this.shaderApplyMorph =
(ComputeShader)Instantiate(this.shaderApplyMorph);          // bug
workaround

    this.kernelHandleComputeBoneDQ =
this.shaderComputeBoneDQ.FindKernel("CSMain");
    this.kernelHandleDQBlend = this.shaderDQBlend.FindKernel("CSMain");
    this.kernelHandleApplyMorph =
this.shaderApplyMorph.FindKernel("CSMain");

    this.bones = this.smr.bones;

    this.started = true;
    this.GrabMeshFromSkinnedMeshRenderer();

    for (int i = 0; i < this.morphWeights.Length; i++)
    {
      this.morphWeights[i] = this.smr.GetBlendShapeWeight(i);
    }

    this.smr.enabled = false;
  }

  // Update is called once per frame
  void Update()
  {
    if (this.mr.isVisible == false)
    {
      return;
    }
```

```
    this.mf.mesh.MarkDynamic();    // once or every frame? idk.
                    // at least it does not affect performance

    for (int i = 0; i < this.bones.Length; i++)
    {
      this.poseMatrices[i] = this.bones[i].localToWorldMatrix;
    }

    this.bufPoseMatrices.SetData(this.poseMatrices);

    // Calculate blended quaternions

    int numThreadGroups = this.bones.Length / numthreads;
    numThreadGroups += this.bones.Length % numthreads == 0 ? 0 : 1;

    this.shaderComputeBoneDQ.SetVector("boneOrientation",
this.boneOrientationVector);
    this.shaderComputeBoneDQ.SetMatrix(
      "self_matrix",
      this.transform.worldToLocalMatrix
    );
    this.shaderComputeBoneDQ.Dispatch(this.kernelHandleComputeBoneDQ,
numThreadGroups, 1, 1);

    numThreadGroups = this.mf.mesh.vertexCount / numthreads;
    numThreadGroups += this.mf.mesh.vertexCount % numthreads == 0 ? 0 : 1;

    this.shaderDQBlend.SetFloat("compensation_coef",
this.bulgeCompensation);
    this.shaderDQBlend.Dispatch(this.kernelHandleDQBlend, numThreadGroups,
1, 1);

    this.materialPropertyBlock.SetTexture("skinned_data_1",
this.rtSkinnedData_1);
    this.materialPropertyBlock.SetTexture("skinned_data_2",
this.rtSkinnedData_2);
    this.materialPropertyBlock.SetTexture("skinned_data_3",
this.rtSkinnedData_3);
```

```
    this.materialPropertyBlock.SetInt("skinned_tex_height",
this.mf.mesh.vertexCount / textureWidth);
    this.materialPropertyBlock.SetInt("skinned_tex_width", textureWidth);

    this.mr.SetPropertyBlock(this.materialPropertyBlock);
  }
}
```

Лістинг 7. Набір функцій для роботи з дуальними кватерніонами (файл DQ.cginc)

```
struct dual_quaternion
{
  float4 rotation_quaternion;
  float4 translation_quaternion;
};

float4 QuaternionInvert(float4 q)
{
  q.xyz *= -1;
  return q;
}

float4 QuaternionMultiply(float4 q1, float4 q2)
{
  float w = q1.w * q2.w - dot(q1.xyz, q2.xyz);
  q1.xyz = q2.xyz * q1.w + q1.xyz * q2.w + cross(q1.xyz, q2.xyz);
  q1.w = w;
  return q1;
}

struct dual_quaternion DualQuaternionMultiply(struct dual_quaternion dq1,
struct dual_quaternion dq2)
{
  struct dual_quaternion result;

  result.translation_quaternion =
QuaternionMultiply(dq1.rotation_quaternion,    dq2.translation_quaternion)
+
```

```
                  QuaternionMultiply(dq1.translation_quaternion,
dq2.rotation_quaternion);

  result.rotation_quaternion = QuaternionMultiply(dq1.rotation_quaternion,
dq2.rotation_quaternion);

  float mag = length(result.rotation_quaternion);
  result.rotation_quaternion /= mag;
  result.translation_quaternion /= mag;

  return result;
}

struct dual_quaternion DualQuaternionShortestPath(struct dual_quaternion
dq1, struct dual_quaternion dq2)
{
  bool isBadPath = dot(dq1.rotation_quaternion, dq2.rotation_quaternion) <
0;
  dq1.rotation_quaternion    = isBadPath ? -dq1.rotation_quaternion    :
dq1.rotation_quaternion;
  dq1.translation_quaternion  = isBadPath ? -dq1.translation_quaternion  :
dq1.translation_quaternion;
  return dq1;
}

float4 QuaternionApplyRotation(float4 v, float4 rotQ)
{
  v = QuaternionMultiply(rotQ, v);
  return QuaternionMultiply(v, QuaternionInvert(rotQ));
}

inline float signNoZero(float x)
{
  float s = sign(x);
  if (s)
    return s;
  return 1;
}
```

```
struct dual_quaternion DualQuaternionFromMatrix4x4(float4x4 m)
{
  struct  dual_quaternion dq;

  //
http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm
  // Alternative Method by Christian
  dq.rotation_quaternion.w = sqrt(max(0, 1.0 + m[0][0] + m[1][1] + m[2][2])) / 2.0;
  dq.rotation_quaternion.x = sqrt(max(0, 1.0 + m[0][0] - m[1][1] - m[2][2])) / 2.0;
  dq.rotation_quaternion.y = sqrt(max(0, 1.0 - m[0][0] + m[1][1] - m[2][2])) / 2.0;
  dq.rotation_quaternion.z = sqrt(max(0, 1.0 - m[0][0] - m[1][1] + m[2][2])) / 2.0;
  dq.rotation_quaternion.x *= signNoZero(m[2][1] - m[1][2]);
  dq.rotation_quaternion.y *= signNoZero(m[0][2] - m[2][0]);
  dq.rotation_quaternion.z *= signNoZero(m[1][0] - m[0][1]);

  dq.rotation_quaternion = normalize(dq.rotation_quaternion);  // ensure unit quaternion

  dq.translation_quaternion = float4(m[0][3], m[1][3], m[2][3], 0);
  dq.translation_quaternion = QuaternionMultiply(dq.translation_quaternion, dq.rotation_quaternion) * 0.5;

  return dq;
}
```

Лістинг 8. Розрахунковий шейдер (ComputeBoneDQ.compute), що генерує вхідні дані для шейдеру скінінгу (DQBlend.compute)

```
// Each #kernel tells which function to compile; you can have many kernels
#pragma kernel CSMain

#include "DQ.cginc"

struct boneWeight
{
  int boneIndex0;
  int boneIndex1;
  int boneIndex2;
  int boneIndex3;

  float boneWeight0;
  float boneWeight1;
  float boneWeight2;
  float boneWeight3;
};

RWStructuredBuffer<float4x4> pose_matrices;
float4x4 self_matrix;

RWStructuredBuffer<dual_quaternion> bind_dual_quaternions;
RWStructuredBuffer<dual_quaternion> skinned_dual_quaternions;

RWStructuredBuffer<float4> bone_directions;

float4 boneOrientation;

[numthreads(1024,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
  struct dual_quaternion dq_bind;
  dq_bind  = bind_dual_quaternions.Load(id.x);
```

```
  dq_bind.translation_quaternion =
QuaternionMultiply(dq_bind.translation_quaternion,
dq_bind.rotation_quaternion) * 0.5;

  float4x4 pose_matrix = transpose(pose_matrices.Load(id.x));
  pose_matrix = mul(self_matrix, pose_matrix);

  struct dual_quaternion dq_pose =
DualQuaternionFromMatrix4x4(pose_matrix);
  struct dual_quaternion dq_skinned = DualQuaternionMultiply(dq_pose,
dq_bind);

  skinned_dual_quaternions[id.x].rotation_quaternion    =
dq_skinned.rotation_quaternion;
  skinned_dual_quaternions[id.x].translation_quaternion  =
dq_skinned.translation_quaternion;

  bone_directions[id.x] = QuaternionApplyRotation(boneOrientation,
dq_pose.rotation_quaternion);
}
```

Лістинг 9. Розрахунковий шейдер, що виконує скінінг (DQBlend.compute)

```
// Each #kernel tells which function to compile; you can have many kernels
#pragma kernel CSMain
#pragma multi_compile_local __ TWIST_COMPENSATION_EXPERIMENTAL

#include "DQ.cginc"

struct bone_weight_info
{
  int4 bone_indexes;
  float4 bone_weights;
};

struct vertex_info
{
  float4 position;
```

```
  float4 normal;
  float4 tangent;

  int4 bone_indexes;
  float4 bone_weights;

  float compensation_coef;
};

float compensation_coef;

RWStructuredBuffer<dual_quaternion> skinned_dual_quaternions;
RWStructuredBuffer<vertex_info> vertex_infos;

uint textureWidth;
RWTexture2D<float4> skinned_data_1;
RWTexture2D<float4> skinned_data_2;
RWTexture2D<float2> skinned_data_3;

RWStructuredBuffer<float4> bone_directions;

struct vertex_info SkinVertex(struct vertex_info vertinfo)
{
  int4 bone_indexes  = vertinfo.bone_indexes;
  float4 bone_weights  = vertinfo.bone_weights;

  struct dual_quaternion dq0 =
skinned_dual_quaternions.Load(bone_indexes[0]);
  struct dual_quaternion dq1 =
skinned_dual_quaternions.Load(bone_indexes[1]);
  struct dual_quaternion dq2 =
skinned_dual_quaternions.Load(bone_indexes[2]);
  struct dual_quaternion dq3 =
skinned_dual_quaternions.Load(bone_indexes[3]);

  dq1 = DualQuaternionShortestPath(dq1, dq0);
  dq2 = DualQuaternionShortestPath(dq2, dq0);
  dq3 = DualQuaternionShortestPath(dq3, dq0);
```

```
  struct dual_quaternion skinned_dq;
  skinned_dq.rotation_quaternion =  dq0.rotation_quaternion *
bone_weights[0];
  skinned_dq.rotation_quaternion +=  dq1.rotation_quaternion *
bone_weights[1];
  skinned_dq.rotation_quaternion +=  dq2.rotation_quaternion *
bone_weights[2];
  skinned_dq.rotation_quaternion +=  dq3.rotation_quaternion *
bone_weights[3];

  skinned_dq.translation_quaternion =    dq0.translation_quaternion *
bone_weights[0];
  skinned_dq.translation_quaternion +=  dq1.translation_quaternion *
bone_weights[1];
  skinned_dq.translation_quaternion +=  dq2.translation_quaternion *
bone_weights[2];
  skinned_dq.translation_quaternion +=  dq3.translation_quaternion *
bone_weights[3];

  float mag = length(skinned_dq.rotation_quaternion);
  skinned_dq.rotation_quaternion    /= mag;
  skinned_dq.translation_quaternion  /= mag;

  vertinfo.position  = QuaternionApplyRotation(vertinfo.position,
skinned_dq.rotation_quaternion);
  vertinfo.normal    = QuaternionApplyRotation(vertinfo.normal,
skinned_dq.rotation_quaternion);
  vertinfo.tangent  = QuaternionApplyRotation(vertinfo.tangent,
skinned_dq.rotation_quaternion);

  vertinfo.position += QuaternionMultiply(skinned_dq.translation_quaternion
* 2, QuaternionInvert(skinned_dq.rotation_quaternion));
  vertinfo.normal = normalize(vertinfo.normal);
  vertinfo.tangent = normalize(vertinfo.tangent);

  // experimental
```

```
  float4 rq = QuaternionMultiply(dq0.rotation_quaternion,
QuaternionInvert(dq1.rotation_quaternion));

  // branching is removed by the compiler optimization
  if((bone_weights[1] != 0.0 && length(rq.xyz) > 0.001))
  {
    float4 boneDir0 = bone_directions.Load(bone_indexes[0]);
    float4 boneDir1 = bone_directions.Load(bone_indexes[1]);

    float3 axis = normalize(rq.xyz);
    float3 bisector = normalize(boneDir0.xyz + boneDir1.xyz);
    bisector = bone_indexes[0] > bone_indexes[1] ? bisector : -bisector;

    float3 offset = bisector - axis * dot(axis, bisector);

    float angleCoef = saturate(2.0*sqrt(1.0-rq.w));

    float x = bone_weights[1] / (bone_weights[0] + bone_weights[1]);
    float compensation = 2.2*x-9.6*x*x+10.4*x*x*x;

    compensation *= vertinfo.compensation_coef;
    compensation *= compensation_coef;
    compensation *= angleCoef;
    compensation *= 1.0 - bone_weights[2] / bone_weights[1];
    compensation *= bone_weights[0] + bone_weights[1];

    vertinfo.position.xyz += offset * compensation;
  }

  return vertinfo;
}

[numthreads(1024,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
  int2 pos;
  pos.x = id.x % textureWidth;
  pos.y = id.x / textureWidth;
```

```
  struct vertex_info vertinfo = vertex_infos.Load(id.x);

  vertinfo = SkinVertex(vertinfo);

  skinned_data_1[pos] = float4(vertinfo.position.xyz, vertinfo.normal.x);
  skinned_data_2[pos] = float4(vertinfo.normal.yz, vertinfo.tangent.xy);
  skinned_data_3[pos] = vertinfo.tangent.zw;
}
```

Лістинг 10. Розрахунковий шейдер, що застосовує ключові форми до моделі (ApplyMorph.compute)

```
// Each #kernel tells which function to compile; you can have many kernels
#pragma kernel CSMain

#include "DQ.cginc"

struct vertex_info
{
  float4 position;
  float4 normal;
  float4 tangent;

  int4 bone_indexes;
  float4 bone_weights;

  float compensation_coef;
};

struct morph_delta
{
  float4 position;
  float4 normal;
  float4 tangent;
};

RWStructuredBuffer<vertex_info> source;
```

```
RWStructuredBuffer<vertex_info> target;
RWStructuredBuffer<morph_delta> delta;

float weight;

[numthreads(1024, 1, 1)]
void CSMain(uint3 id : SV_DispatchThreadID)
{
  struct vertex_info vertinfo = source.Load(id.x);
  struct morph_delta morphinfo = delta.Load(id.x);

  vertinfo.position  += morphinfo.position  * weight;
  vertinfo.normal    += morphinfo.normal    * weight;
  vertinfo.tangent  += morphinfo.tangent  * weight;

  target[id.x] = vertinfo;
}
```

Лістинг 11. Шейдер матеріалу HackedStandard

```
// Unity built-in shader source. Copyright (c) 2016 Unity Technologies. MIT
license (see license.txt)

/*
  modified to use MadCake's dual quaternion skinning script

  modifications marked by  comments:
    // ----- DQ modification start -----
    ---inserted code---
    // ----- DQ modification end -----

  modifications added to following passes:
    - Forward
    - ShadowCaster
    - Deferred
*/

Shader "MadCake/Material/Standard hacked for DQ skinning"
```

```
{
    Properties
    {
        _Color("Color", Color) = (1,1,1,1)
        _MainTex("Albedo", 2D) = "white" {}

        _Cutoff("Alpha Cutoff", Range(0.0, 1.0)) = 0.5

        _Glossiness("Smoothness", Range(0.0, 1.0)) = 0.5
        _GlossMapScale("Smoothness Scale", Range(0.0, 1.0)) = 1.0
        [Enum(Metallic Alpha,0,Albedo Alpha,1)] _SmoothnessTextureChannel
("Smoothness texture channel", Float) = 0

        [Gamma] _Metallic("Metallic", Range(0.0, 1.0)) = 0.0
        _MetallicGlossMap("Metallic", 2D) = "white" {}

        [ToggleOff] _SpecularHighlights("Specular Highlights", Float) = 1.0
        [ToggleOff] _GlossyReflections("Glossy Reflections", Float) = 1.0

        _BumpScale("Scale", Float) = 1.0
        _BumpMap("Normal Map", 2D) = "bump" {}

        _Parallax ("Height Scale", Range (0.005, 0.08)) = 0.02
        _ParallaxMap ("Height Map", 2D) = "black" {}

        _OcclusionStrength("Strength", Range(0.0, 1.0)) = 1.0
        _OcclusionMap("Occlusion", 2D) = "white" {}

        _EmissionColor("Color", Color) = (0,0,0)
        _EmissionMap("Emission", 2D) = "white" {}

        _DetailMask("Detail Mask", 2D) = "white" {}

        _DetailAlbedoMap("Detail Albedo x2", 2D) = "grey" {}
        _DetailNormalMapScale("Scale", Float) = 1.0
        _DetailNormalMap("Normal Map", 2D) = "bump" {}
```

```
        [Enum(UV0,0,UV1,1)] _UVSec ("UV Set for secondary textures", Float)
= 0


        // Blending state
        [HideInInspector] _Mode ("__mode", Float) = 0.0
        [HideInInspector] _SrcBlend ("__src", Float) = 1.0
        [HideInInspector] _DstBlend ("__dst", Float) = 0.0
        [HideInInspector] _ZWrite ("__zw", Float) = 1.0
    }

    CGINCLUDE
        #define UNITY_SETUP_BRDF_INPUT MetallicSetup
    ENDCG

    SubShader
    {
        Tags { "RenderType"="Opaque" "PerformanceChecks"="False" }
        LOD 300


        // ------------------------------------------------------------------
--
        //  Base forward pass (directional light, emission, lightmaps, ...)
        Pass
        {
            Name "FORWARD"
            Tags { "LightMode" = "ForwardBase" }

            Blend [_SrcBlend] [_DstBlend]
            ZWrite [_ZWrite]

            CGPROGRAM
            #pragma target 3.0

            // -------------------------------------

            #pragma shader_feature _NORMALMAP
```

```
            #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON
_ALPHAPREMULTIPLY_ON
            #pragma shader_feature _EMISSION
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature ___ _DETAIL_MULX2
            #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
            #pragma shader_feature _ _GLOSSYREFLECTIONS_OFF
            #pragma shader_feature _PARALLAXMAP

            #pragma multi_compile_fwdbase
            #pragma multi_compile_fog
            #pragma multi_compile_instancing
            // Uncomment the following line to enable dithering LOD
crossfade. Note: there are more in the file to uncomment for other passes.
            //#pragma multi_compile _ LOD_FADE_CROSSFADE

      //#pragma vertex vertForwardBase
            #pragma fragment fragBase
            #include "UnityStandardCoreForward.cginc"

      // ----- DQ modification start -----

      #pragma vertex vertSkinnedForward // original #pragma vertex (above)
was commented

      // expanded version of original vertex input structure
      struct VertexInputSkinningForward
      {
        float4 vertex   : POSITION;
        half3 normal    : NORMAL;
        float2 uv0      : TEXCOORD0;
        float2 uv1      : TEXCOORD1;

        // this was added, everything else remains unchanged
        uint id : SV_VertexID;

        #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
```

```
        float2 uv2      : TEXCOORD2;
      #endif

      #ifdef _TANGENT_TO_WORLD
        half4 tangent   : TANGENT;
      #endif

      UNITY_VERTEX_INPUT_INSTANCE_ID
    };

    // variables used for skining, always the same for every pass
    sampler2D skinned_data_1;
    sampler2D skinned_data_2;
    sampler2D skinned_data_3;
    uint skinned_tex_height;
    uint skinned_tex_width;
    bool _DoSkinning;

    // the actual skinning function
    // don't change the code but change the argument type to the name of
vertex input structure used in current pass
    // for this pass it is VertexInputSkinningForward
    void vert(inout VertexInputSkinningForward v) {
      if (_DoSkinning) {
        float2 skinned_tex_uv;

        skinned_tex_uv.x = (float(v.id % skinned_tex_width)) /
skinned_tex_width;
        skinned_tex_uv.y = (float(v.id / skinned_tex_width)) /
skinned_tex_height;

        float4 data_1 = tex2Dlod(skinned_data_1, float4(skinned_tex_uv,
0, 0));
        float4 data_2 = tex2Dlod(skinned_data_2, float4(skinned_tex_uv,
0, 0));

        #ifdef _TANGENT_TO_WORLD
```

```
            float2 data_3 = tex2Dlod(skinned_data_3, float4(skinned_tex_uv,
0, 0)).xy;
          #endif

          v.vertex.xyz = data_1.xyz;
          v.vertex.w = 1;

          v.normal.x = data_1.w;
          v.normal.yz = data_2.xy;

          #ifdef _TANGENT_TO_WORLD
            v.tangent.xy = data_2.zw;
            v.tangent.zw = data_3.xy;
          #endif
        }
      }

      // this function will replace the original vertex function
(vertForwardBase)
      // the return type is the same as in the original function
      // the argument type is our expanded structure
      VertexOutputForwardBase vertSkinnedForward(VertexInputSkinningForward
vs)
      {
        // first we apply skinning
        vert(vs);

        // then we create the original vertex structure (VertexInput for
this pass)
        // and fill it with the data from our expanded version
        VertexInput v;
        v.vertex = vs.vertex;
        v.normal = vs.normal;
        v.uv0 = vs.uv0;
        v.uv1 = vs.uv1;

        // this variable is inside an "if defined" block in original
structure
```

```
        // so accessing it should be enclosed in identical "if defined"
block
        #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
          v.uv2 = vs.uv2;
        #endif

        // same here
        #ifdef _TANGENT_TO_WORLD
          v.tangent = vs.tangent;
        #endif

        // finally we pass the original vertex structure to the original
vertex function
        // and return the result
        return vertForwardBase(v);
      }

      // ----- DQ modification end -----

          ENDCG
        }
        // ------------------------------------------------------------------
--
        //  Additive forward pass (one light per pass)
        Pass
        {
            Name "FORWARD_DELTA"
            Tags { "LightMode" = "ForwardAdd" }
            Blend [_SrcBlend] One
            Fog { Color (0,0,0,0) } // in additive pass fog should be black
            ZWrite Off
            ZTest LEqual

            CGPROGRAM
            #pragma target 3.0

            // --------------------------------------
```

```
            #pragma shader_feature _NORMALMAP
            #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON
_ALPHAPREMULTIPLY_ON
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
            #pragma shader_feature ___ _DETAIL_MULX2
            #pragma shader_feature _PARALLAXMAP

            #pragma multi_compile_fwdadd_fullshadows
            #pragma multi_compile_fog
            // Uncomment the following line to enable dithering LOD
crossfade. Note: there are more in the file to uncomment for other passes.
            //#pragma multi_compile _ LOD_FADE_CROSSFADE

            //#pragma vertex vertAdd
            #pragma fragment fragAdd
            #include "UnityStandardCoreForward.cginc"

      // ----- DQ modification start -----

      #pragma vertex vertSkinnedForwardAdd // original #pragma vertex
(above) was commented

      struct VertexInputSkinningForwardAdd
      {
        float4 vertex   : POSITION;
        half3 normal    : NORMAL;
        float2 uv0      : TEXCOORD0;
        float2 uv1      : TEXCOORD1;
        uint id : SV_VertexID;

        #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
          float2 uv2      : TEXCOORD2;
        #endif

        #ifdef _TANGENT_TO_WORLD
```

```
        half4 tangent   : TANGENT;
      #endif

      UNITY_VERTEX_INPUT_INSTANCE_ID
    };

    sampler2D skinned_data_1;
    sampler2D skinned_data_2;
    sampler2D skinned_data_3;
    uint skinned_tex_height;
    uint skinned_tex_width;
    bool _DoSkinning;

    void vert(inout VertexInputSkinningForwardAdd v) {
      if (_DoSkinning) {
        float2 skinned_tex_uv;

        skinned_tex_uv.x = (float(v.id % skinned_tex_width)) /
skinned_tex_width;
        skinned_tex_uv.y = (float(v.id / skinned_tex_width)) /
skinned_tex_height;

        float4 data_1 = tex2Dlod(skinned_data_1, float4(skinned_tex_uv,
0, 0));
        float4 data_2 = tex2Dlod(skinned_data_2, float4(skinned_tex_uv,
0, 0));

        #ifdef _TANGENT_TO_WORLD
          float2 data_3 = tex2Dlod(skinned_data_3, float4(skinned_tex_uv,
0, 0)).xy;
        #endif

        v.vertex.xyz = data_1.xyz;
        v.vertex.w = 1;

        v.normal.x = data_1.w;
        v.normal.yz = data_2.xy;
```

```
            #ifdef _TANGENT_TO_WORLD
              v.tangent.xy = data_2.zw;
              v.tangent.zw = data_3.xy;
            #endif
          }
        }

        VertexOutputForwardAdd
vertSkinnedForwardAdd(VertexInputSkinningForwardAdd vs)
        {
          vert(vs);

          VertexInput v;
          v.vertex = vs.vertex;
          v.normal = vs.normal;
          v.uv0 = vs.uv0;
          v.uv1 = vs.uv1;

          #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
            v.uv2 = vs.uv2;
          #endif

          #ifdef _TANGENT_TO_WORLD
            v.tangent = vs.tangent;
          #endif

          return vertAdd(v);
        }

        // ----- DQ modification end -----

            ENDCG
          }
          // ------------------------------------------------------------------
--
          //  Shadow rendering pass
          Pass {
              Name "ShadowCaster"
```

```
            Tags { "LightMode" = "ShadowCaster" }

            ZWrite On ZTest LEqual

            CGPROGRAM
            #pragma target 3.0

            // -------------------------------------


            #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON
_ALPHAPREMULTIPLY_ON
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma shader_feature _PARALLAXMAP
            #pragma multi_compile_shadowcaster
            #pragma multi_compile_instancing
            // Uncomment the following line to enable dithering LOD
crossfade. Note: there are more in the file to uncomment for other passes.
            //#pragma multi_compile _ LOD_FADE_CROSSFADE

      //#pragma vertex verthadowCaster
            #pragma fragment fragShadowCaster

            #include "UnityStandardShadow.cginc"

      // ----- DQ modification start -----

      #pragma vertex vertSkinnedShadowCaster // original #pragma vertex
(above) was commented

      struct VertexInputSkinningShadowCaster
      {
        float4 vertex   : POSITION;
        half3 normal    : NORMAL;
        float2 uv0      : TEXCOORD0;
        uint id : SV_VertexID;
```

```
        #if defined(UNITY_STANDARD_USE_SHADOW_UVS) && defined(_PARALLAXMAP)
          half4 tangent   : TANGENT;
        #endif

        UNITY_VERTEX_INPUT_INSTANCE_ID
      };

      sampler2D skinned_data_1;
      sampler2D skinned_data_2;
      sampler2D skinned_data_3;
      uint skinned_tex_height;
      uint skinned_tex_width;
      bool _DoSkinning;

      void vert(inout VertexInputSkinningShadowCaster v) {
        if (_DoSkinning) {
          float2 skinned_tex_uv;

          skinned_tex_uv.x = (float(v.id % skinned_tex_width)) /
skinned_tex_width;
          skinned_tex_uv.y = (float(v.id / skinned_tex_width)) /
skinned_tex_height;

          float4 data_1 = tex2Dlod(skinned_data_1, float4(skinned_tex_uv,
0, 0));
          float4 data_2 = tex2Dlod(skinned_data_2, float4(skinned_tex_uv,
0, 0));

          #if defined(UNITY_STANDARD_USE_SHADOW_UVS) &&
defined(_PARALLAXMAP)
            float2 data_3 = tex2Dlod(skinned_data_3, float4(skinned_tex_uv,
0, 0)).xy;
          #endif

          v.vertex.xyz = data_1.xyz;
          v.vertex.w = 1;

          v.normal.x = data_1.w;
```

```
            v.normal.yz = data_2.xy;

            #if defined(UNITY_STANDARD_USE_SHADOW_UVS) &&
defined(_PARALLAXMAP)
              v.tangent.xy = data_2.zw;
              v.tangent.zw = data_3.xy;
            #endif
          }
        }

        void vertSkinnedShadowCaster(
          VertexInputSkinningShadowCaster vs,
          out float4 opos : SV_POSITION

          #ifdef UNITY_STANDARD_USE_SHADOW_OUTPUT_STRUCT
            , out VertexOutputShadowCaster o
          #endif

          #ifdef UNITY_STANDARD_USE_STEREO_SHADOW_OUTPUT_STRUCT
            , out VertexOutputStereoShadowCaster os
          #endif
        )
        {
          vert(vs);

          VertexInput v;
          v.vertex = vs.vertex;
          v.normal = vs.normal;
          v.uv0 = vs.uv0;

          #if defined(UNITY_STANDARD_USE_SHADOW_UVS) && defined(_PARALLAXMAP)
            v.tangent = vs.tangent;
          #endif

          vertShadowCaster(
            v,
            opos
```

```
            #ifdef UNITY_STANDARD_USE_SHADOW_OUTPUT_STRUCT
              , o
            #endif

            #ifdef UNITY_STANDARD_USE_STEREO_SHADOW_OUTPUT_STRUCT
              , os
            #endif
          );
        }

        // ----- DQ modification end -----

        ENDCG
          }
          // ------------------------------------------------------------------
--
          //  Deferred pass
          Pass
          {
              Name "DEFERRED"
              Tags { "LightMode" = "Deferred" }

              CGPROGRAM
              #pragma target 3.0
              #pragma exclude_renderers nomrt


              // -------------------------------------

              #pragma shader_feature _NORMALMAP
              #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON
_ALPHAPREMULTIPLY_ON
              #pragma shader_feature _EMISSION
              #pragma shader_feature _METALLICGLOSSMAP
              #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
              #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
              #pragma shader_feature ___ _DETAIL_MULX2
              #pragma shader_feature _PARALLAXMAP
```

```
            #pragma multi_compile_prepassfinal
            #pragma multi_compile_instancing
            // Uncomment the following line to enable dithering LOD
crossfade. Note: there are more in the file to uncomment for other passes.
            //#pragma multi_compile _ LOD_FADE_CROSSFADE

            //#pragma vertex vertDeferred
            #pragma fragment fragDeferred

            #include "UnityStandardCore.cginc"

      // ----- DQ modification start -----

      #pragma vertex vertSkinnedDeferred // original #pragma vertex (above)
was commented

      struct VertexInputSkinningDeferred
        {
          float4 vertex   : POSITION;
          half3 normal    : NORMAL;
          float2 uv0      : TEXCOORD0;
          float2 uv1      : TEXCOORD1;
          uint id : SV_VertexID;

          #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
            float2 uv2      : TEXCOORD2;
          #endif

          #ifdef _TANGENT_TO_WORLD
            half4 tangent   : TANGENT;
          #endif


          UNITY_VERTEX_INPUT_INSTANCE_ID
        };

      sampler2D skinned_data_1;
```

```
      sampler2D skinned_data_2;
      sampler2D skinned_data_3;
      uint skinned_tex_height;
      uint skinned_tex_width;
      bool _DoSkinning;

      void vert(inout VertexInputSkinningDeferred v) {
        if (_DoSkinning) {
          float2 skinned_tex_uv;

          skinned_tex_uv.x = (float(v.id % skinned_tex_width)) /
skinned_tex_width;
          skinned_tex_uv.y = (float(v.id / skinned_tex_width)) /
skinned_tex_height;

          float4 data_1 = tex2Dlod(skinned_data_1, float4(skinned_tex_uv,
0, 0));
          float4 data_2 = tex2Dlod(skinned_data_2, float4(skinned_tex_uv,
0, 0));

          #ifdef _TANGENT_TO_WORLD
            float2 data_3 = tex2Dlod(skinned_data_3, float4(skinned_tex_uv,
0, 0)).xy;
          #endif

          v.vertex.xyz = data_1.xyz;
          v.vertex.w = 1;

          v.normal.x = data_1.w;
          v.normal.yz = data_2.xy;

          #ifdef _TANGENT_TO_WORLD
            v.tangent.xy = data_2.zw;
            v.tangent.zw = data_3.xy;
          #endif
        }
      }
```

```
      VertexOutputDeferred vertSkinnedDeferred(VertexInputSkinningDeferred
vs)
      {
        vert(vs);

        VertexInput v;
        v.vertex = vs.vertex;
        v.normal = vs.normal;
        v.uv0 = vs.uv0;
        v.uv1 = vs.uv1;

        #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
          v.uv2 = vs.uv2;
        #endif

        #ifdef _TANGENT_TO_WORLD
          v.tangent = vs.tangent;
        #endif

        return vertDeferred(v);
      }

      // ----- DQ modification end -----

            ENDCG
        }

        // ------------------------------------------------------------------
--
        // Extracts information for lightmapping, GI (emission, albedo,
...)
        // This pass it not used during regular rendering.
        Pass
        {
            Name "META"
            Tags { "LightMode"="Meta" }

            Cull Off
```

```
            CGPROGRAM
            //#pragma vertex vert_meta
            #pragma fragment frag_meta

            #pragma shader_feature _EMISSION
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma shader_feature ___ _DETAIL_MULX2
            #pragma shader_feature EDITOR_VISUALIZATION

            #include "UnityStandardMeta.cginc"

        // ----- DQ modification start -----

        #pragma vertex vertSkinnedMeta // original #pragma vertex (above) was
commented

        struct VertexInputSkinningMeta
          {
            float4 vertex   : POSITION;
            half3 normal    : NORMAL;
            float2 uv0      : TEXCOORD0;
            float2 uv1      : TEXCOORD1;
            uint id : SV_VertexID;

            #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
              float2 uv2      : TEXCOORD2;
            #endif

            #ifdef _TANGENT_TO_WORLD
              half4 tangent   : TANGENT;
            #endif


            UNITY_VERTEX_INPUT_INSTANCE_ID
          };
```

```
      sampler2D skinned_data_1;
      sampler2D skinned_data_2;
      sampler2D skinned_data_3;
      uint skinned_tex_height;
      uint skinned_tex_width;
      bool _DoSkinning;

      void vert(inout VertexInputSkinningMeta v) {
        if (_DoSkinning) {
          float2 skinned_tex_uv;

          skinned_tex_uv.x = (float(v.id % skinned_tex_width)) /
skinned_tex_width;
          skinned_tex_uv.y = (float(v.id / skinned_tex_width)) /
skinned_tex_height;

          float4 data_1 = tex2Dlod(skinned_data_1, float4(skinned_tex_uv,
0, 0));
          float4 data_2 = tex2Dlod(skinned_data_2, float4(skinned_tex_uv,
0, 0));

          #ifdef _TANGENT_TO_WORLD
            float2 data_3 = tex2Dlod(skinned_data_3, float4(skinned_tex_uv,
0, 0)).xy;
          #endif

          v.vertex.xyz = data_1.xyz;
          v.vertex.w = 1;

          v.normal.x = data_1.w;
          v.normal.yz = data_2.xy;

          #ifdef _TANGENT_TO_WORLD
            v.tangent.xy = data_2.zw;
            v.tangent.zw = data_3.xy;
          #endif
        }
      }
```

```
      v2f_meta vertSkinnedMeta(VertexInputSkinningMeta vs)
      {
        vert(vs);

        VertexInput v;
        v.vertex = vs.vertex;
        v.normal = vs.normal;
        v.uv0 = vs.uv0;
        v.uv1 = vs.uv1;

        #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
          v.uv2 = vs.uv2;
        #endif

        #ifdef _TANGENT_TO_WORLD
          v.tangent = vs.tangent;
        #endif

        return vert_meta(v);
      }

      // ----- DQ modification end -----


          ENDCG
      }
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" "PerformanceChecks"="False" }
        LOD 150

        // ------------------------------------------------------------------
--
        //  Base forward pass (directional light, emission, lightmaps, ...)
        Pass
```

```
        {
            Name "FORWARD"
            Tags { "LightMode" = "ForwardBase" }

            Blend [_SrcBlend] [_DstBlend]
            ZWrite [_ZWrite]

            CGPROGRAM
            #pragma target 2.0

            #pragma shader_feature _NORMALMAP
            #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON
_ALPHAPREMULTIPLY_ON
            #pragma shader_feature _EMISSION
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
            #pragma shader_feature _ _GLOSSYREFLECTIONS_OFF
            // SM2.0: NOT SUPPORTED shader_feature ___ _DETAIL_MULX2
            // SM2.0: NOT SUPPORTED shader_feature _PARALLAXMAP

            #pragma skip_variants SHADOWS_SOFT DIRLIGHTMAP_COMBINED

            #pragma multi_compile_fwdbase
            #pragma multi_compile_fog

            //#pragma vertex vertBase
            #pragma fragment fragBase
            #include "UnityStandardCoreForward.cginc"

      // ----- DQ modification start -----

      #pragma vertex vertSkinnedForward // original #pragma vertex (above)
was commented

      struct VertexInputSkinningForward
      {
        float4 vertex   : POSITION;
```

```
        half3 normal    : NORMAL;
        float2 uv0      : TEXCOORD0;
        float2 uv1      : TEXCOORD1;
        uint id : SV_VertexID;

        #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
          float2 uv2      : TEXCOORD2;
        #endif

        #ifdef _TANGENT_TO_WORLD
          half4 tangent   : TANGENT;
        #endif


        UNITY_VERTEX_INPUT_INSTANCE_ID
      };

      sampler2D skinned_data_1;
      sampler2D skinned_data_2;
      sampler2D skinned_data_3;
      uint skinned_tex_height;
      uint skinned_tex_width;
      bool _DoSkinning;

      void vert(inout VertexInputSkinningForward v) {
        if (_DoSkinning) {
          float2 skinned_tex_uv;

          skinned_tex_uv.x = (float(v.id % skinned_tex_width)) /
skinned_tex_width;
          skinned_tex_uv.y = (float(v.id / skinned_tex_width)) /
skinned_tex_height;

          float4 data_1 = tex2Dlod(skinned_data_1, float4(skinned_tex_uv,
0, 0));
          float4 data_2 = tex2Dlod(skinned_data_2, float4(skinned_tex_uv,
0, 0));
```

```
          #ifdef _TANGENT_TO_WORLD
            float2 data_3 = tex2Dlod(skinned_data_3, float4(skinned_tex_uv,
0, 0)).xy;
          #endif

          v.vertex.xyz = data_1.xyz;
          v.vertex.w = 1;

          v.normal.x = data_1.w;
          v.normal.yz = data_2.xy;

          #ifdef _TANGENT_TO_WORLD
            v.tangent.xy = data_2.zw;
            v.tangent.zw = data_3.xy;
          #endif
        }
      }

      VertexOutputForwardBase vertSkinnedForward(VertexInputSkinningForward
vs)
      {
        vert(vs);

        VertexInput v;
        v.vertex = vs.vertex;
        v.normal = vs.normal;
        v.uv0 = vs.uv0;
        v.uv1 = vs.uv1;
      #if defined(DYNAMICLIGHTMAP_ON) || defined(UNITY_PASS_META)
        v.uv2 = vs.uv2;
      #endif

      #ifdef _TANGENT_TO_WORLD
        v.tangent = vs.tangent;
      #endif
        return vertForwardBase(v);
      }
```

```
      // ----- DQ modification end -----

            ENDCG
        }
        // ------------------------------------------------------------------
--
        //  Additive forward pass (one light per pass)
        Pass
        {
            Name "FORWARD_DELTA"
            Tags { "LightMode" = "ForwardAdd" }
            Blend [_SrcBlend] One
            Fog { Color (0,0,0,0) } // in additive pass fog should be black
            ZWrite Off
            ZTest LEqual

            CGPROGRAM
            #pragma target 2.0

            #pragma shader_feature _NORMALMAP
            #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON
_ALPHAPREMULTIPLY_ON
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
            #pragma shader_feature ___ _DETAIL_MULX2
            // SM2.0: NOT SUPPORTED shader_feature _PARALLAXMAP
            #pragma skip_variants SHADOWS_SOFT

            #pragma multi_compile_fwdadd_fullshadows
            #pragma multi_compile_fog

            #pragma vertex vertAdd
            #pragma fragment fragAdd
            #include "UnityStandardCoreForward.cginc"

            ENDCG
        }
```

```
        // ------------------------------------------------------------------
--
        //  Shadow rendering pass
        Pass {
            Name "ShadowCaster"
            Tags { "LightMode" = "ShadowCaster" }

            ZWrite On ZTest LEqual

            CGPROGRAM
            #pragma target 2.0

            #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON
_ALPHAPREMULTIPLY_ON
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma skip_variants SHADOWS_SOFT
            #pragma multi_compile_shadowcaster

            //#pragma vertex vertShadowCaster
            #pragma fragment fragShadowCaster

            #include "UnityStandardShadow.cginc"

      // ----- DQ modification start -----

      #pragma vertex vertSkinnedShadowCaster // original #pragma vertex
(above) was commented

      struct VertexInputSkinningShadowCaster
      {
        float4 vertex   : POSITION;
        half3 normal    : NORMAL;
        float2 uv0      : TEXCOORD0;
        uint id : SV_VertexID;

        #if defined(UNITY_STANDARD_USE_SHADOW_UVS) && defined(_PARALLAXMAP)
          half4 tangent   : TANGENT;
```

```
        #endif

        UNITY_VERTEX_INPUT_INSTANCE_ID
      };

      sampler2D skinned_data_1;
      sampler2D skinned_data_2;
      sampler2D skinned_data_3;
      uint skinned_tex_height;
      uint skinned_tex_width;
      bool _DoSkinning;

      void vert(inout VertexInputSkinningShadowCaster v) {
        if (_DoSkinning) {
          float2 skinned_tex_uv;

          skinned_tex_uv.x = (float(v.id % skinned_tex_width)) /
skinned_tex_width;
          skinned_tex_uv.y = (float(v.id / skinned_tex_width)) /
skinned_tex_height;

          float4 data_1 = tex2Dlod(skinned_data_1, float4(skinned_tex_uv,
0, 0));
          float4 data_2 = tex2Dlod(skinned_data_2, float4(skinned_tex_uv,
0, 0));

          #if defined(UNITY_STANDARD_USE_SHADOW_UVS) &&
defined(_PARALLAXMAP)
            float2 data_3 = tex2Dlod(skinned_data_3, float4(skinned_tex_uv,
0, 0)).xy;
          #endif

          v.vertex.xyz = data_1.xyz;
          v.vertex.w = 1;

          v.normal.x = data_1.w;
          v.normal.yz = data_2.xy;
```

```
          #if defined(UNITY_STANDARD_USE_SHADOW_UVS) &&
defined(_PARALLAXMAP)
            v.tangent.xy = data_2.zw;
            v.tangent.zw = data_3.xy;
          #endif
        }
      }

      void vertSkinnedShadowCaster(
        VertexInputSkinningShadowCaster vs,
        out float4 opos : SV_POSITION

        #ifdef UNITY_STANDARD_USE_SHADOW_OUTPUT_STRUCT
          , out VertexOutputShadowCaster o
        #endif

        #ifdef UNITY_STANDARD_USE_STEREO_SHADOW_OUTPUT_STRUCT
          , out VertexOutputStereoShadowCaster os
        #endif
      )
      {
        vert(vs);

        VertexInput v;
        v.vertex = vs.vertex;
        v.normal = vs.normal;
        v.uv0 = vs.uv0;

        #if defined(UNITY_STANDARD_USE_SHADOW_UVS) && defined(_PARALLAXMAP)
          v.tangent = vs.tangent;
        #endif

        vertShadowCaster(
          v,
          opos

          #ifdef UNITY_STANDARD_USE_SHADOW_OUTPUT_STRUCT
            , o
```

```
            #endif

            #ifdef UNITY_STANDARD_USE_STEREO_SHADOW_OUTPUT_STRUCT
              , os
            #endif
          );
        }

        // ----- DQ modification end -----

            ENDCG
          }

          // ------------------------------------------------------------------
--
          // Extracts information for lightmapping, GI (emission, albedo,
...)
          // This pass it not used during regular rendering.
          Pass
          {
              Name "META"
              Tags { "LightMode"="Meta" }

              Cull Off

              CGPROGRAM
              #pragma vertex vert_meta
              #pragma fragment frag_meta

              #pragma shader_feature _EMISSION
              #pragma shader_feature _METALLICGLOSSMAP
              #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
              #pragma shader_feature ___ _DETAIL_MULX2
              #pragma shader_feature EDITOR_VISUALIZATION

              #include "UnityStandardMeta.cginc"
              ENDCG
          }
```

```
    }


    FallBack "VertexLit"
    CustomEditor "StandardShaderGUI"
}
```

Лістинг 12. Кастомізований інспектор для скрипта DualQuaternionSkinner

```
using UnityEngine;
using UnityEditor;

using System.Collections.Generic;

#if UNITY_EDITOR

/// <summary>
/// Provides custom inspector for {@link DualQuaternionSkinner}
/// </summary>
[CustomEditor(typeof(DualQuaternionSkinner))]
public class DualQuaternionSkinnerEditor : Editor
{
  SerializedProperty shaderComputeBoneDQ;
  SerializedProperty shaderDQBlend;
  SerializedProperty shaderApplyMorph;
  SerializedProperty bulgeCompensation;

  DualQuaternionSkinner dqs;
  SkinnedMeshRenderer smr
  {
    get
    {
      if (this._smr == null)
      {
        this._smr =
((DualQuaternionSkinner)this.target).gameObject.GetComponent<SkinnedMeshRen
derer>();
      }
```

```
      return this._smr;
    }
  }
  SkinnedMeshRenderer _smr;

  bool showBlendShapes = false;

  enum BoneOrientation {X, Y, Z, negativeX, negativeY, negativeZ}
  readonly Dictionary<BoneOrientation, Vector3> boneOrientationVectors =
new Dictionary<BoneOrientation, Vector3>();

  private void OnEnable()
  {
    this.shaderComputeBoneDQ  =
this.serializedObject.FindProperty("shaderComputeBoneDQ");
    this.shaderDQBlend      =
this.serializedObject.FindProperty("shaderDQBlend");
    this.shaderApplyMorph    =
this.serializedObject.FindProperty("shaderApplyMorph");
    this.bulgeCompensation    =
this.serializedObject.FindProperty("bulgeCompensation");

    this.boneOrientationVectors.Add(BoneOrientation.X, new Vector3(1,0,0));
    this.boneOrientationVectors.Add(BoneOrientation.Y, new Vector3(0,1,0));
    this.boneOrientationVectors.Add(BoneOrientation.Z, new Vector3(0,0,1));
    this.boneOrientationVectors.Add(BoneOrientation.negativeX, new
Vector3(-1,0,0));
    this.boneOrientationVectors.Add(BoneOrientation.negativeY, new
Vector3(0,-1,0));
    this.boneOrientationVectors.Add(BoneOrientation.negativeZ, new
Vector3(0,0,-1));
  }

  public override void OnInspectorGUI()
  {
    this.serializedObject.Update();
```

```
    this.dqs = (DualQuaternionSkinner)this.target;

    EditorGUILayout.BeginHorizontal();
      EditorGUILayout.LabelField("Mode: ", GUILayout.Width(80));
      EditorGUILayout.LabelField(Application.isPlaying ? "Play" : "Editor",
GUILayout.Width(80));
    EditorGUILayout.EndHorizontal();

    EditorGUILayout.BeginHorizontal();
      EditorGUILayout.LabelField("DQ skinning: ", GUILayout.Width(80));
      EditorGUILayout.LabelField(this.dqs.started ? "ON" : "OFF",
GUILayout.Width(80));
    EditorGUILayout.EndHorizontal();

    EditorGUILayout.Space();
    this.dqs.SetViewFrustrumCulling(EditorGUILayout.Toggle("View frustrum
culling: ", this.dqs.viewFrustrumCulling));
    EditorGUILayout.Space();

    BoneOrientation currentOrientation = BoneOrientation.X;
    foreach(BoneOrientation orientation in
this.boneOrientationVectors.Keys)
    {
      if (this.dqs.boneOrientationVector ==
this.boneOrientationVectors[orientation])
      {
        currentOrientation = orientation;
        break;
      }
    }
    var newOrientation = (BoneOrientation)EditorGUILayout.EnumPopup("Bone
orientation: ", currentOrientation);
    if (this.dqs.boneOrientationVector !=
this.boneOrientationVectors[newOrientation])
    {
      this.dqs.boneOrientationVector =
this.boneOrientationVectors[newOrientation];
      this.dqs.UpdatePerVertexCompensationCoef();
```

```
    }

    EditorGUILayout.PropertyField(this.bulgeCompensation);
    EditorGUILayout.Space();

    this.showBlendShapes = EditorGUILayout.Foldout(this.showBlendShapes,
"Blend shapes");

    if (this.showBlendShapes)
    {
      if (this.dqs.started == false)
      {
        EditorGUI.BeginChangeCheck();

Undo.RecordObject(this.dqs.gameObject.GetComponent<SkinnedMeshRenderer>(),
"changed blendshape weights by DualQuaternionSkinner component");
      }

      for (int i = 0; i < this.dqs.mesh.blendShapeCount; i++)
      {
        EditorGUILayout.BeginHorizontal();
        EditorGUILayout.LabelField("   " +
this.dqs.mesh.GetBlendShapeName(i),
GUILayout.Width(EditorGUIUtility.labelWidth - 10));
        float weight =
EditorGUILayout.Slider(this.dqs.GetBlendShapeWeight(i), 0, 100);
        EditorGUILayout.EndHorizontal();
        this.dqs.SetBlendShapeWeight(i, weight);
      }
    }

    EditorGUILayout.Space();

    EditorGUILayout.PropertyField(this.shaderComputeBoneDQ);
    EditorGUILayout.PropertyField(this.shaderDQBlend);
    EditorGUILayout.PropertyField(this.shaderApplyMorph);

    EditorGUILayout.Space();
```

```
    EditorGUILayout.LabelField("Problems: ");

    if (this.CheckProblems() == false)
    {
      EditorGUILayout.LabelField("not detected (this is good)");
    }

    this.serializedObject.ApplyModifiedProperties();
  }

  bool CheckProblems()
  {
    var wrapStyle = new GUIStyle() { wordWrap = true };

    if (this.smr.sharedMesh.isReadable == false)
    {
      EditorGUILayout.Space();
      EditorGUILayout.LabelField("Skinned mesh must be read/write enabled
(check import settings)", wrapStyle);
      return true;
    }

    if (this.smr.rootBone.parent != this.dqs.gameObject.transform.parent)
    {
      EditorGUILayout.Space();

      EditorGUILayout.BeginHorizontal();
        EditorGUILayout.LabelField("Skinned object and root bone must be
children of the same parent", wrapStyle);
        if (GUILayout.Button("auto fix"))
        {
          Undo.SetTransformParent(
            this.smr.rootBone,
            this.dqs.gameObject.transform.parent,
            "Changed root bone's parent by Dual Quaternion Skinner (auto
fix)"
          );
          EditorApplication.RepaintHierarchyWindow();
```

```
        }
      EditorGUILayout.EndHorizontal();

      return true;
    }

    foreach(Transform bone in this.smr.bones)
    {
      if (bone.localScale != Vector3.one)
      {
        EditorGUILayout.Space();

        EditorGUILayout.BeginHorizontal();
          EditorGUILayout.LabelField(string.Format("Bone scaling not
supported: {0}", bone.name), wrapStyle);
          if (GUILayout.Button("auto fix"))
          {
            Undo.RecordObject(bone, "Set bone scale to (1,1,1) by Dual
Quaternion Skinner (auto fix)");
            bone.localScale = Vector3.one;
          }
        EditorGUILayout.EndHorizontal();

        return true;
      }
    }

    return false;
  }
}

#endif
```

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 КОМПЕНСАЦІЇ ДЕФЕКТІВ DQ-СКІНІНГУ

Виконав: Руденко Костянтин Павлович

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

Київ – 2020

Актуальність

Для використання у інтерактивних застосунках, особливо для мобільних платформ, VR та AR, швидкодія методу скінінгу є критичною. Сучасні методи скінінгу, що забезпечують високу швидкодію, створюють значні дефекти анімації.

Об’єкт дослідження: скелетна анімація реалістичних тривимірних моделей.

Предмет дослідження: методи скінінгу реалистічних тривимірних моделей та дефекти скінінгу.

Наукове завдання:
розроблення алгоритмічно-програмного методу пост-обробки моделі, що мінімізує дефекти скінінгу без значного зниження швидкодії.

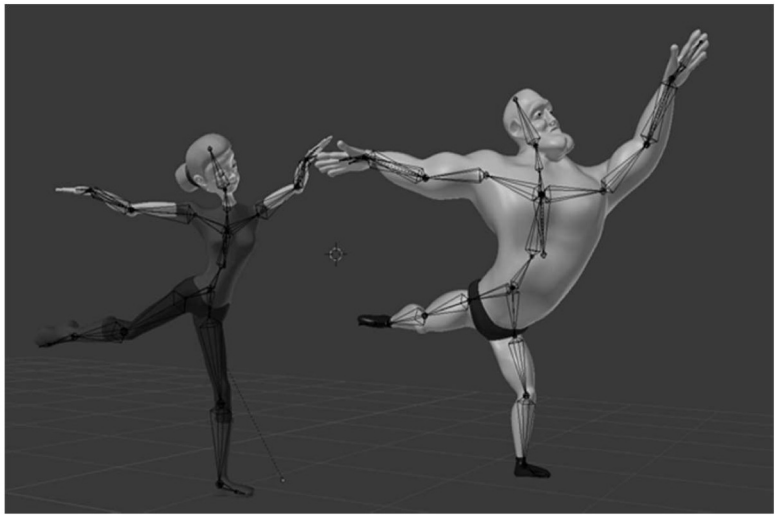
Мета дослідження:
розробити програмне забезпечення, що забезпечує компенсацію небажаних деформацій при скінінгу тривимірних моделей.

Окремі завдання

1. Провести аналіз існуючих методів скінінгу реалістичних тривимірних моделей.
2. Дослідити дефекти скінінгу та способи боротьби з ними.
3. Обрати метод скінінгу для компенсації дефектів.
4. Розробити алгоритмічно-програмний метод пост-обробки моделі для компенсації дефектів скінінгу.
5. Розробити реалізацію методу алгоритмічно-програмного методу пост-обробки моделі.
6. Провести аналіз розробленої реалізації відносто швидкодії та якості анімації в порівнянні з існуючими методами скінінгу.

Список термінів, скорочень та позначень

Скелетна анімація – цетехніка комп’ютерної анімації, що використовує ієрархічну структуру об’єктів, що називається арматурою, або скелетом, для керування рухами моделі.



Список термінів, скорочень та позначень

Скінінг – це процес визначення положень вертексів моделі в просторі в залежності від положення кісток арматури, до якої модель прив’язана.

Воксель – елемент простору, позначає значення певної величини в клітинці рівномірної просторової ґратки. Аналог пікселю в двовимірних зображеннях.

Вертекс – це структура даних, що опусує точку в просторі та первний набір атрибутів (колір, вектор нормалі, тощо).

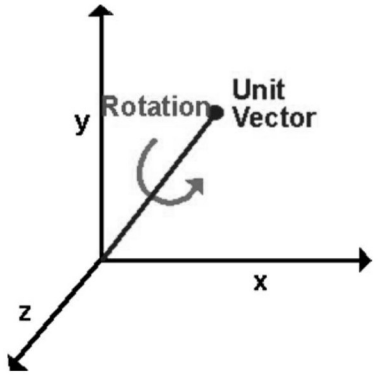
Список термінів, скорочень та позначень

Кватерніон – це гіперкомплексне число типу a + bi + cj + dk, де a,b,c,d – дійсні числа, i,j,k – уявні одиниці. Кватерніони, зокрема, можуть використовуватися для опису поворотів в тривимірному просторі.

$$a + i\,b + j\,c + k\,d$$

.

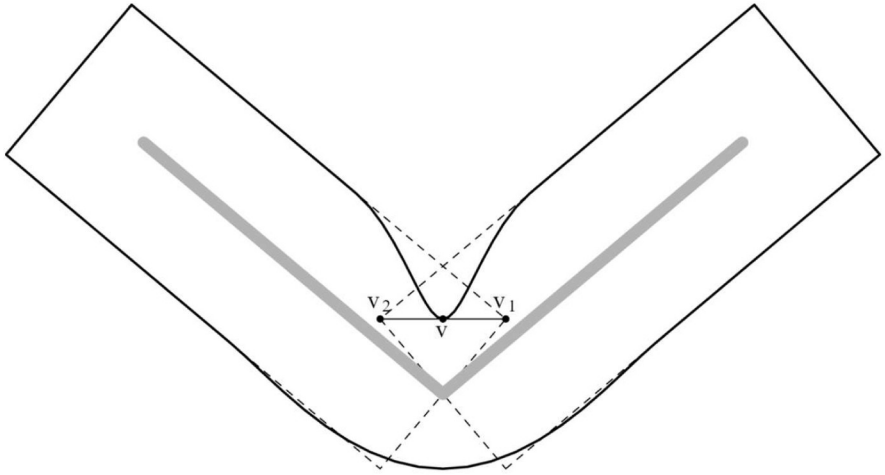
$$ij = k, \qquad ji = -k,$$
$$jk = i, \qquad kj = -i,$$
$$ki = j, \qquad ik = -j.$$



Список термінів, скорочень та позначень

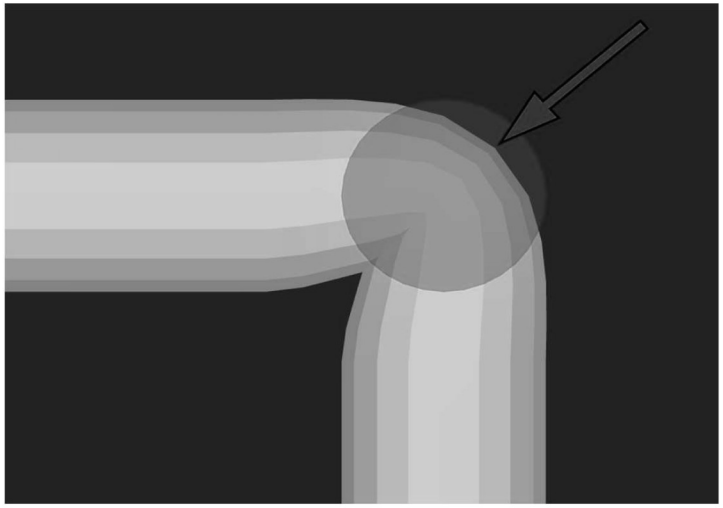
Дуальні кватерніони – це гіперкомплексні числа типу q r + q d ε, де q r , q d –кватерніони, ε – число, таке, що ε 2 = 0, але ε / = 0. Дуальні кватерніони можуть, зокрема, використовуватися для позначення комбінації зсуву та повороту в тривимірному просторі.

Дефект лінійного скінінгу – втрата об’єму



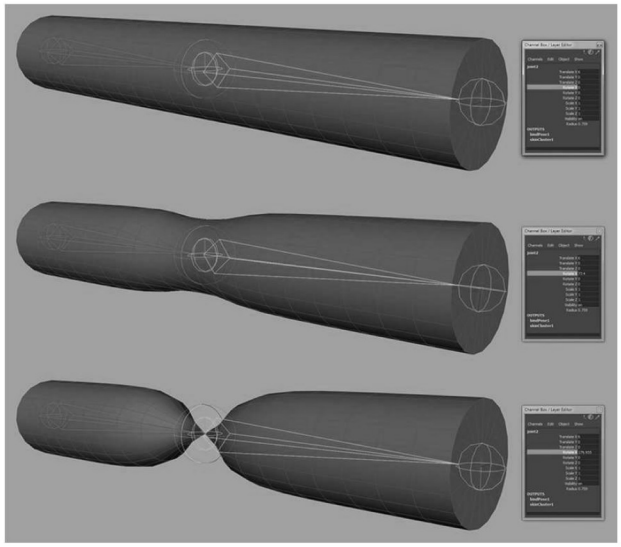
При використанні лінійного скінінгу, положення вертекса під контролем кількох кісток визначається, як зважена сума його положень під повним контролем кожної кістки окремо. В результаті вертекс наближається до кістки та модель втрачає об’єм.

Дефект лінійного скінінгу – втрата об’єму



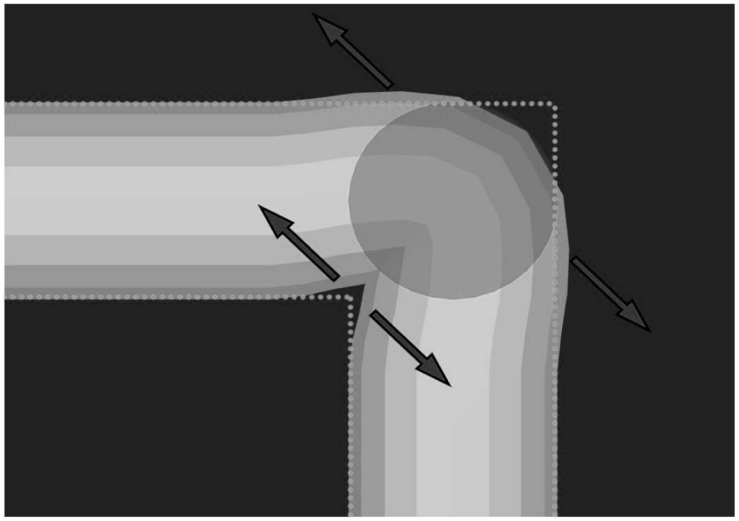
Синій диск відображає ідеалізовану форму зігнутого суглоба. Червона стрілка показує напрям небажаної деформації, що призводить до втрати о’єму

Дефект лінійного скінінгу – втрата об’єму



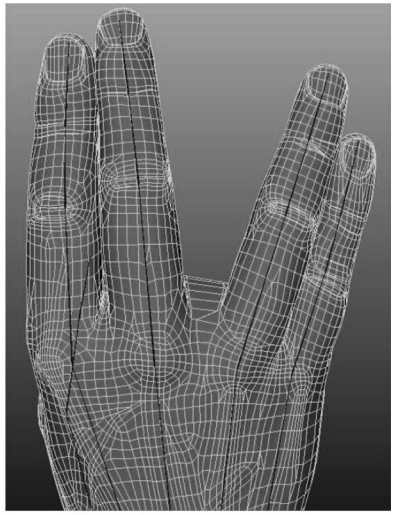
При скручуванні моделі дефект втрати об’єму є більш помітним. Скручування на 180 градусів призводить до стискання області моделі у точку.

Дефект DQ-скінінгу – роздування суглоба



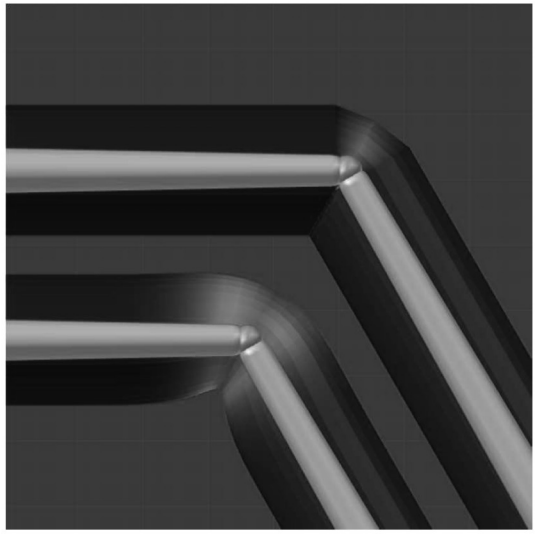
DQ-скінінг має протилежний дефект – роздування суглоба. Червоні стрілки показують напрям небажаного дефекту. Синій диск та зелений пунктир показують ідеалізовану форму зігнутого суглоба.

Bulging-free dual quaternion skinning



Метод bulging-free DQ-скінінгу попереджає роздування моделі, але створює дефекти у зонах, де сісідні вертекси зміщуються в напрямку до різних кісток (наприклад, між пальцями руки).

Припущення щодо оброблюваної моделі



Припускається, що:

- градієнт вагових коефіцієнтів не має різких стрибків;
- зона, не вагові коефіцієнти сусідніх кісток є рівними, лежить на середині суглоба (зелена на рисунку);
- форма градієнту вагових коефіцієнтів є прилизно однакова для всіх моделей;
- ширина перехідної зони пропорційна радіусу кінцівки (відстані від вертекса до кістки).

Градієнт вагових коефіцієнтів

$$H(x) = \frac{d}{dx}\max\{x, 0\} \quad (1)$$

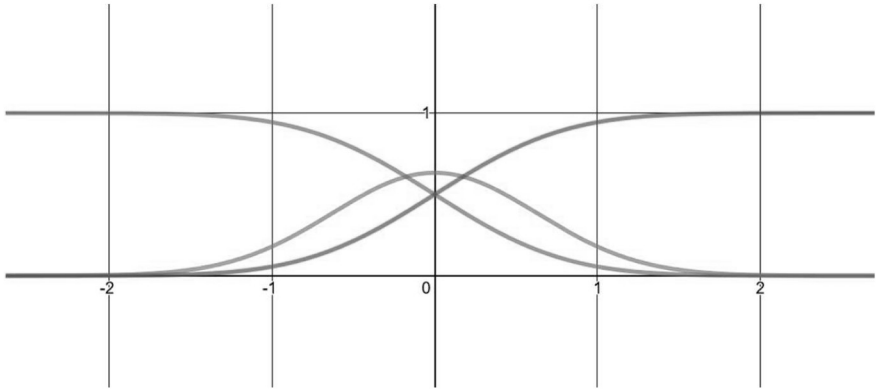
$$g(x) = \frac{e^{-\frac{x^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}} \quad (2)$$

$$w_1(x) = \int_{-\infty}^{\infty} g(t)H(t-x)dt \quad (3)$$

$$w_2(x) = \int_{-\infty}^{\infty} g(t)H(x-t)dt \quad (4)$$

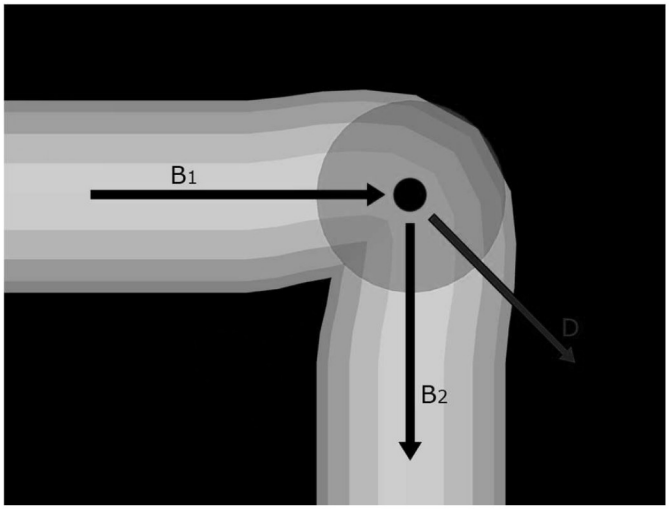
Спочатку ваговий коефіцієнт вертксів визначається, як одиничний стрибок (1).

Після багатократного згладжування, що апроксимується згорткою (3), (4) з функцією Гауса (2)

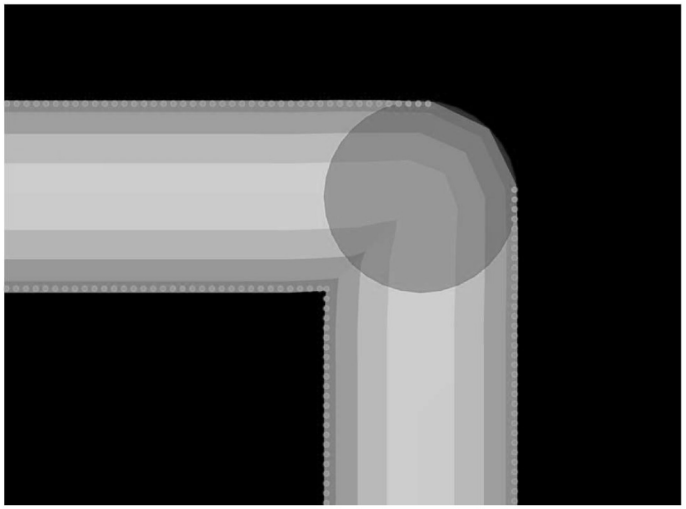


- Горизонтальна вісь – положення вертекса відносно суглоба;
- зелений та червоний – вагові коефіцієнти кісток;
- Синій – функція Гауса

Метод компенсації роздутого суглоба



Напрям деформації D визначається, як бісектриса кута, утвореного кістками кінцівки B_1 та B_2



Модель після цомпенсації дефекту.

Відстань зміщення

$$d = f(w_2)g(\alpha)r$$

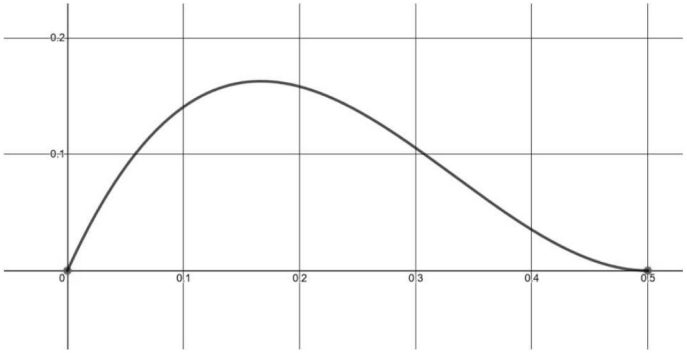
Відстань зміщення *d* визначається на основі вагового коефіцієнта другої кістки w_2, куту згину суглоба α та радіусу кінцівки *r*

Відстань зміщення

$$f(w_2) = aw_2 + bw_2{}^2 + cw_2{}^3 + k \qquad (1)$$

$$\begin{matrix} f(0) = 0 \\ f(0.5) = 0 \end{matrix} \Rightarrow \begin{matrix} k = 0 \\ c = -4a - 2b \end{matrix} \qquad (2)$$

$$f'(0) = 0 \Rightarrow \begin{matrix} \mathrm{b} = -4\mathrm{a} \\ c = -4a - 2b \end{matrix} \Rightarrow \begin{matrix} \mathrm{b} = -4\mathrm{a} \\ c = 4\mathrm{a} \end{matrix} \qquad (3)$$

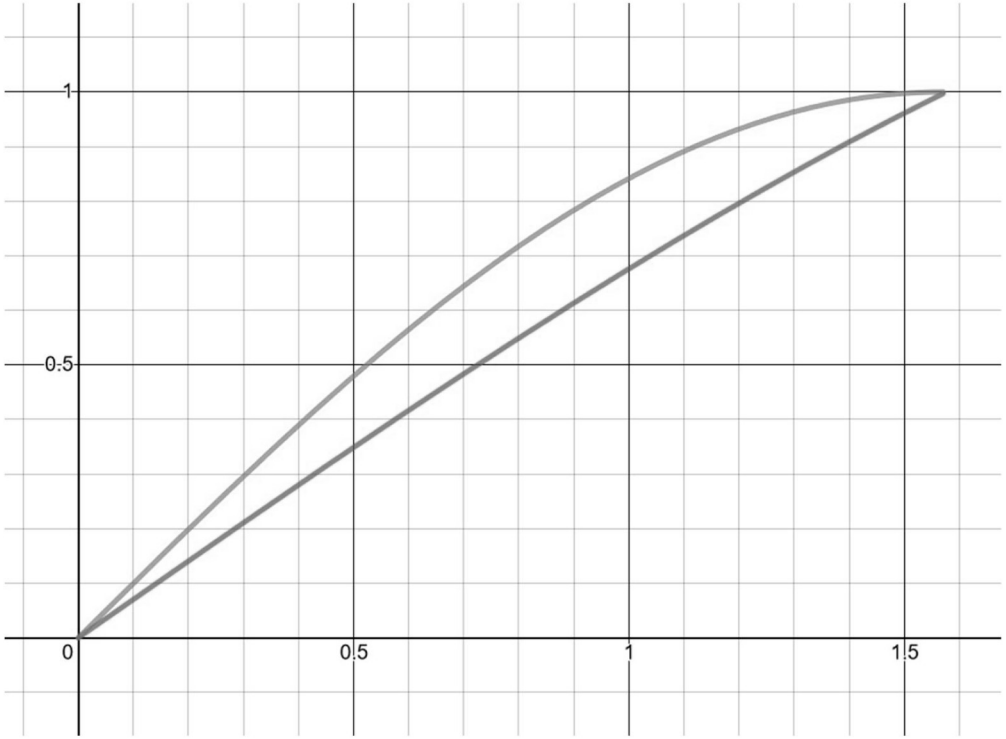


Функцію $f(w_2)$ апроксимовано кубічним поліномом (1) з накладанням обмежень (2) та (3).

Обмеження (2) гарантує відсутність компенсації дефекту на границях перехідної зони та в центрі.

Обмеження (3) гарантує, що в центрі перехужної зони похідна функції $f(w_2)$ дорівнює нулю.

Відстань зміщення



1. $1.41 \sin\left(\frac{x}{2}\right)$
2. $\sin(x)$

Лінійна залежність від кута згину забезпечує найкращу деформацію.

Проте синус половинного кута достатньо точно апроксимує лінійну залежність та потребує менше розрахунків.

$$g(\alpha) = min\left(1.41 \cdot sin\left(\frac{\alpha}{2}\right), 1\right)$$

Алгоритм компенсації дефекту DQ-скінінгу

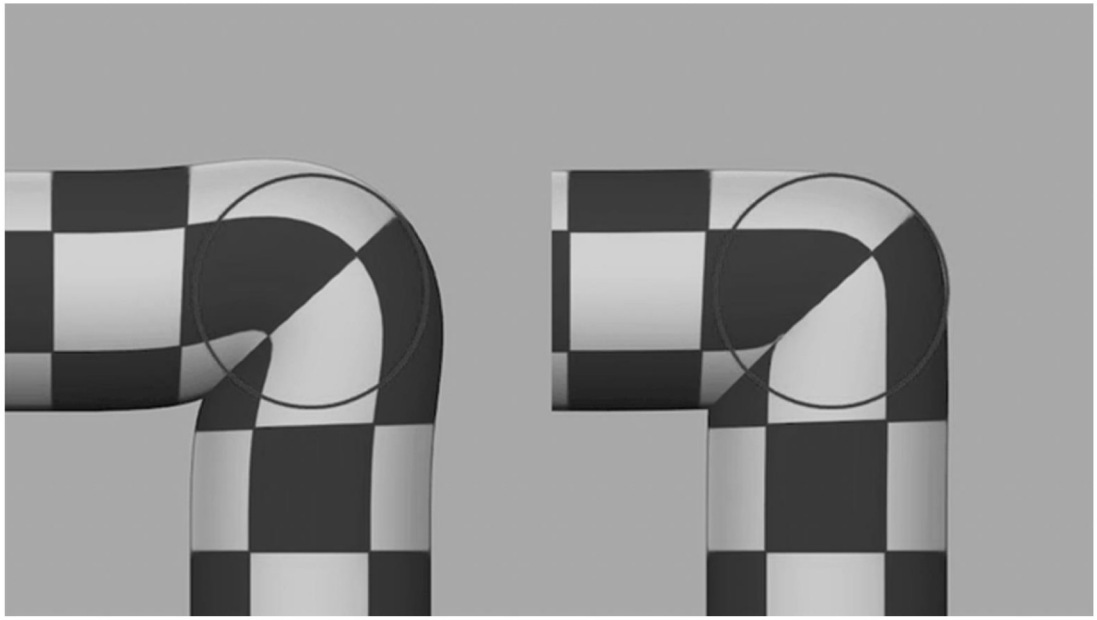
Input:

- rotation quaternions RQ_1 and RQ_2, representing world space rotations of first and second bone, respectively;
- unit vectors V_{bone1} and V_{bone2}, representing world-space directions of the first and second bone, respectively;
- I_1 and I_2 — indices of the first and second bone, respectively
- w_1, w_2, w_3 — weight coefficients for first, second and third bone, respectively;
- s — coefficient for manual adjustment of bulging compensation strength;
- V_{orig} — original position of the vertex V after dual-quaternion skinning.

Output: bulge-compensated vertex position V

1. $RQ = RQ_1 \cdot {RQ_2}^{-1}$
2. $V_{axis} = normalize(RQ.xyz)$
3. $V_{bisector} = normalize(V_{bone1} + V_{bone2})$
4. $if\ (I_2 > I_1)\ then$:
5. $\quad V_{bisector} = -V_{bisector}$
6. $end\ if$
7. $V_{offset} = V_{bisector} - V_{axis} \cdot (V_{axis} \cdot V_{bisector})$
8. $w = \frac{w_2}{w_1 + w_2}$
9. $l = 2.2w - 8.1w^2 + 7.4w^3$
10. $l = l \cdot \min(1, 2\sqrt{1 - RQ.W})$
11. $l = l \cdot (w_1 + w_2)$
12. $l = l \cdot (1 - \frac{w_3}{w_2})$
13. $l = l \cdot s$
14. $V = V_{orig} + (V_{offset} \cdot l)$

Результат компенсації дефекту

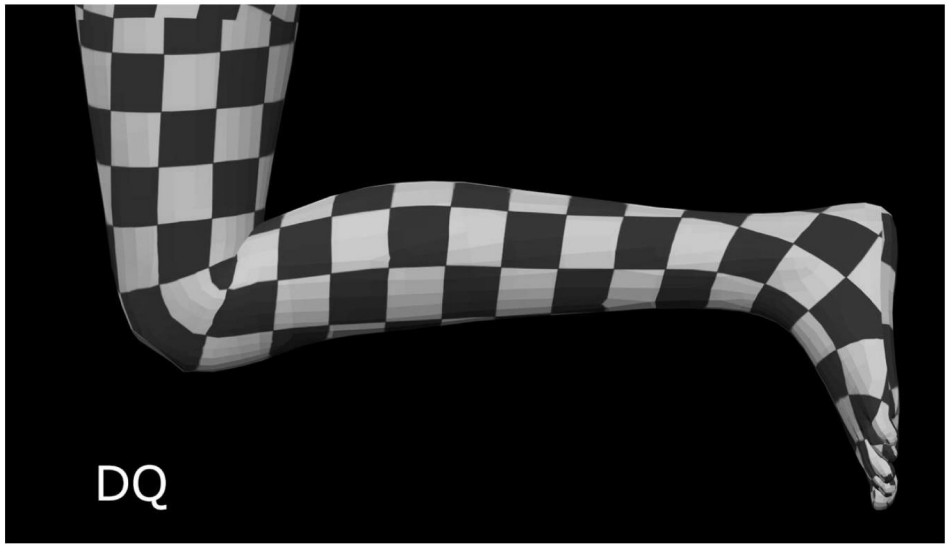


Червоне коло відображає ідеалізовану форму зігнутої моделі.

При згинанні за допомогою звичайного DQ-скінінгу (зліва), модель відходить від ідеалізованої форми.

При використанні компенсації дефекту, модель повторює ідеалізовану форму (справа).

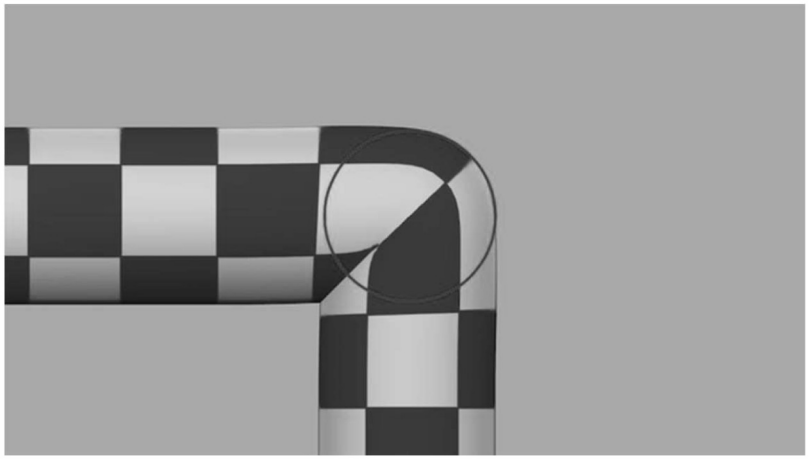
Результат компенсації дефекту



Анімація демонструє порівняння DQ-скінінгу з використанням компенсації дефекту та без неї.

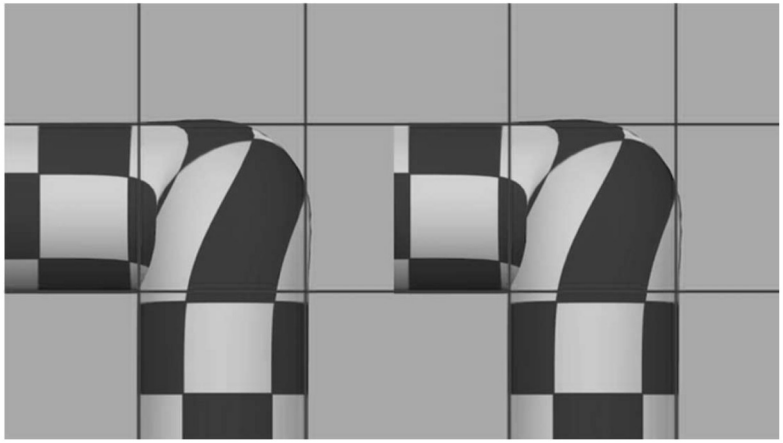
Крім роздування суглоба, полігон на зовнійній стороні коліна є сильно розтягнутим без використання компенсації дефекту.

Результат компенсації дефекту



Анімація демонструє можливість зміни інтенсивності компенсації дефекту безпосередньо під час анімації.

Результат компенсації дефекту

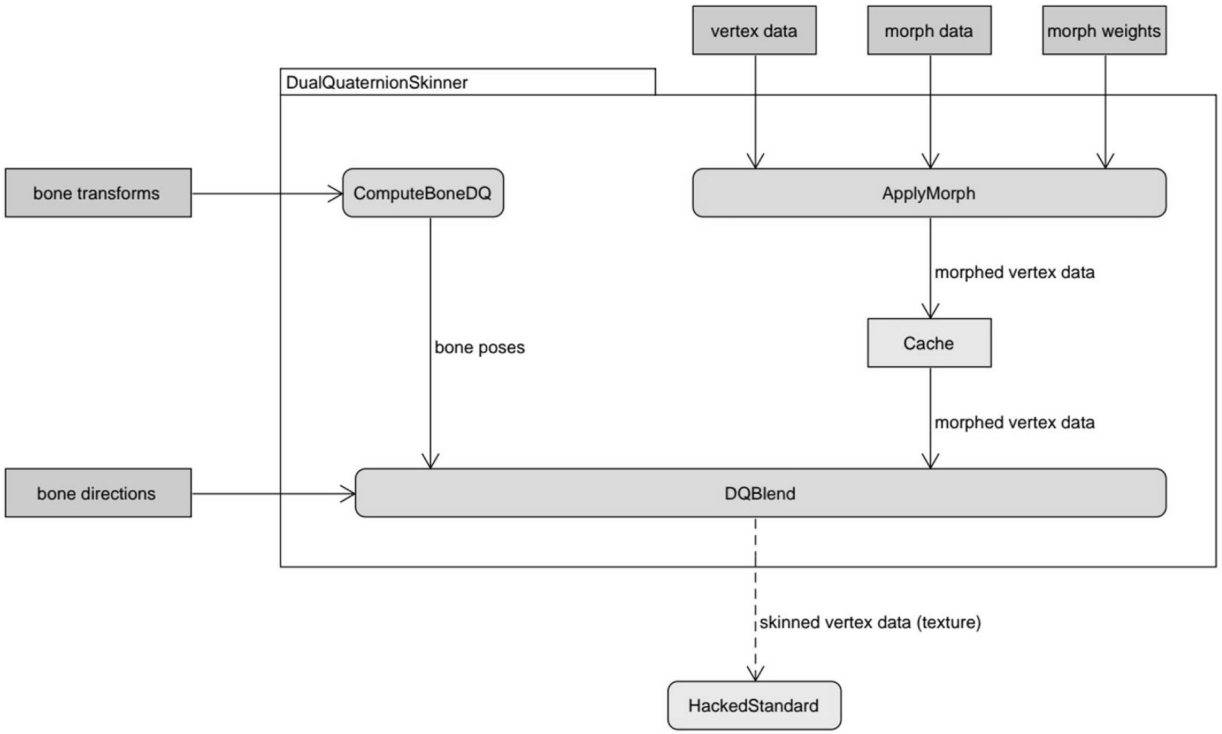


Анімація демонструє комбінацію згину та скручування при використанні DQ-скінінгу без (зліва) за з (справа) компенсацією дефекту.

Запропонований метод не вносить значних змін в такі анімації.

Реалізація розробленого методу

Схема роботи реалізованого плагіна



НАУКОВО-ІНОВАЦІЙНА НОВИЗНА

- Розроблено метод пост-обробки тривимірних моделей, що дозволяє значно відвищити якість анімації з використанням DQ-скінінгу при мінімальних втратах швидкодії.

ВИСНОВКИ

1. В результаті аналізу існуючих методів скінінгу було обрано метод DQ-скінінгу для розробки методу компенсації дефектів
2. Було розроблено метод пост-обробки моделі після DQ-скінінгу, що значною мірою зменшує дефект роздутого суглоба без значної втрати швидкодії.
3. Було розроблено реалізацію запропонованого методу у вигляді плагіну для рушія Unity та проведено його оптимізацію. Фінальна версія плагіну виконує скінінг приблизно на 20% повільніше, ніж вбудована система лінійного скінінгу рушія. Реалізація запропонованого методу знижує швидкодію ще на 8%, що, є незначною різницею, порівняно з альтернативними методами скінінгу.

АПРОБУВАННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

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

2. Стаття на тему «A method of artifact compensation for dual quaternion skinning» до наукового міжнародного журналу «Вісник Хмельницького національного університету» 2020 №1

Дякую за увагу!