НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
Сергій СТІРЕНКО
«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Комп'ютерні системи та мережі»
спеціальності 123 «Комп'ютерна інженерія»
на тему: «Система 3D моделювання»

Виконала:
студентка IV курсу, групи ІО-63
Марія САКОВИЧ _____

Керівник:
асистент
Павло РЕГІДА _____

Консультант з нормоконтролю:
Професор, доктор технічних наук
Валерій СІМОНЕНКО _____

Рецензент:

_____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Сергій СТІРЕНКО
«___»_____2020 р.

ЗАВДАННЯ

на дипломний проєкт студентці

Марії САКОВИЧ

1. Тема проєкту «Система 3D моделювання», керівник проєкту Регіда Павло Геннадійович, асистент, затверджені наказом по університету від «07» травня 2020 р. №1081-с

2. Термін подання студентом проєкту 04.06.2020

3. Вихідні дані до проєкту технічна документація

4. Зміст пояснювальної записки

Аналіз програмної частини існуючих систем 3D моделювання, Розробка систем 3D моделювання. Тестування розробленого програмного забезпечення.

5. Перелік графічного матеріалу

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультната	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Валерій СІМОНЕНКО		

7. Дата видачі завдання

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проекту	Примітка
1.	*Затвердження теми роботи*	01.09.2019	
2.	*Вивчення та аналіз завдання*	02.09.2020-13.03.2020	
3.	*Розробка архітектури та загальної структури системи*	13.03.2020-17.03.2020	
4.	*Реалізація системи*	18.03.2020-20.04.2020	
5.	*Тестування програмного продукту*	21.04.2020-27.04.2020	
6.	*Оформлення пояснювальної записки*	13.04.2020-25.05.2020	
7.	*Предзахист*	26.05.2020	
8.	*Захист*		

Студент Марія САКОВИЧ

Керівник Павло РЕГІДА

Анотація

Даний дипломний проект присвячений розробці програмного забезпечення для системи 3D моделювання. На початку дається неформальна постановка розв'язуваної задачі, а також короткий опис системи. Надається повний аналіз огляду існуючих рішень, а також методів реалізації. Під час виконання дипломного проекту було розроблене програмне забезпечення на мові C++ та представлені відомості про характеристики програми, реалізовані алгоритми, а також продемонстровано роботу кінцевого результату програми.

Abstract

This diploma project is devoted to the development of software for 3D modeling system. At the beginning the informal statement of the solved problem, also the short description of system is given. A full analysis of the review of existing solutions, as well as implementation methods is provided. During the implementation of the diploma project, software was developed in C ++ and presented information about the characteristics of the program as well as implemented algorithms, furthermore the work of the result of the program is demonstrated.

№ рядка	Формат	Позначення	Найменування	Кількість	Примітка
1	А4		Завдання на дипломний проект	2	
2	А4	ДП 4682.01.000 ВП	Відомість дипломного проекту	1	
3	А4	ДП 4682.02.000 ТЗ	Технічне завдання	3	
4	А4	ДП 4682.03.000 ПЗ	Пояснювальна записка	66	
5	А3	ДП 4682.04.000 Д1	Структура проекту	1	
6	А3	ДП 4682.05.000 Д2	Діаграма класів	1	
7	А4	ДП 4682.06.000 Д3	Алгоритм побудови BVH	1	
8	А4	ДП 4682. 07.000 Д4	Лістинг реалізації ідентифікації	33	

Зм.	Арк.	№ докум.	Підпис	Дата	4682.01.000 ВП		
Розробила		Сакович М.В.			Система 3D моделювання Відомість дипломного проєкту	Літ.	Аркуш 1 / Аркушів 1
Перевір.		Регіда П.Г.					
Н. контр.		Сімоненко В.П.				НТУУ “КПІ”, ФІОТ, ІО-63	
Затверд.							

Технічне завдання до дипломного проекту

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання розповсюджується на розробку програмного забезпечення для системи 3D моделювання.

Область застосування: перегляд та редагування 3D моделей з подальшим використанням.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки служить завдання на виконання розробки програмного забезпечення для системи 3D моделювання, затвердженою кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний Інститут».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка програмного забезпечення для системи 3D моделювання.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами для розробки служать науково-технічна література з комп’ютерних технологій, публікації в періодичних виданнях, довідники з програмованих логічних інтегральних схем, публікації в Інтернеті за даним питанням.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розроблюваного продукту

- Розробка засобів візуалізації моделювання
- Розробка алгоритмів для редагування 3D моделі
- Розробка зрозумілого графічного інтерфейсу

5.2. Вимоги до програмного забезпечення

- Операційна система на ядрі Linux
- C++11 і вище
- OpenGL
- Qt 5

5.3. Вимоги до апаратного забезпечення

- Процесор: 2.5 ghz або більше
- Оперативна пам’ять: 4 GB

Пояснювальна записка
до дипломного проєкту

на тему: «Система 3D моделювання»

Київ - 2020 року

ЗМІСТ

ВСТУП

З розвитком технологій та комп’ютерної техніки алгоритми їх роботи ставали набагато швидшими та більш працездатними. 3D-моделювання зараз широко використовуються в будь-якій галузі 3D-графіки. Зараз 3D-моделі використовують навіть на персональних комп’ютерах, а раніше багато комп'ютерних ігор використовували попередньо зображені 3D-моделі як Sprites, перш ніж комп'ютери змогли відобразити їх у режимі реального часу.

Сьогодні 3D-моделі використовуються в найрізноманітніших сферах.

Медична індустрія використовує детальні моделі органів, які створюються за допомогою декількох двовимірних фрагментів зображення за допомогою МРТ або КТ.

Кіно та телевізійна індустрія використовує їх як персонажі та об’єкти для анімаційних та реальних кінофільмів із кіно та телебачення (наприклад, Аватар, Зоряні війни та Гра Престолів).

Індустрія відеоігор використовує їх як засоби для комп'ютерних та відеоігор. Xbox, PlayStation 4 або Nintendo використовують тривимірні ресурси в іграх, незалежно від того, наскільки мультиплікаційними чи реальними вони є.

Сектор наукової галузі використовує їх як досить детальні моделі хімічних сполук, наприклад, проект геному людини.

Архітектура та будівельна галузь використовує їх замість традиційних, фізичних архітектурних моделей для демонстрації запропонованих будівель та ландшафтів. А деякі з цих 3D-моделей потім стають друкованими, щоб, наприклад, показати нову будівлю чи ландшафт у міському середовищі.

Інженерне співтовариство використовує їх для проектування нових пристроїв, транспортних засобів та конструкцій, а також для цілого ряду інших застосувань, таких як неруйнівна прототипізація.

3D-моделі також можуть бути основою для фізичних пристроїв, побудованих за допомогою 3D-принтерів.

Навчання 3D-моделюванню може бути досить непростим завданням. Багато програм складні для новачків, потребують часу і додатково матеріалу. Для швидкого редагування деякий функціонал є зайвим.

Метою дипломної роботи є розробка програмного забезпечення для Системи 3D моделювання, орієнтованого на швидке редагування 3D-моделей без зайвого функціоналу з доступним та зрозумілим інтерфейсом, для операційної системи Linux.

Для досягнення поставленої мети були поставлені наступні основні задачі:

Розгляд доцільності та практичності створення власного ПЗ.

Дослідження та аналіз можливих готових рішень.

Перегляд можливих реалізацій за допомогою існуючих алгоритмів.

Аналіз мов програмування та графічних бібліотек, що найкаще підходять для реалізації задачі.

Реалізація сисмеми 3D моделювання для потрібного програмного забезпечення.

Створення простого та інтуітивно зрозумілого графічного інтерфейсу для користувача.

Дипломна робота складається зі вступу, трьох розділів та висновків до проекту. У першому розділі проведено аналіз вже реалізованих інструментів для 3D моделювання. У другому розділі описуються існуючі алгоритми та доцільність їх використання для створення ПЗ для даної задачі. У третьому розділі описується реалізація всіх складових. У висновках до проекту підбиваються підсумки та робляться остаточні заключення щодо теми Проекту.

РОЗДІЛ 1

АНАЛІЗ ПРОБЛЕМИ

1.1 Вступ

Існує величезна кількість областей, де застосовується тривимірне моделювання та анімація. Наприклад, при випробуванні програми 3D Studio MAX користувачі виконали колосальну роботу, застосовуючи цю програму в різних галузях: від створення статичної реклами та динамічних заставок для телеканалів до моделювання катастроф та тривимірної анімації. [1]

Використовувані в таких галузях, як анімація, ігри, архітектура, виробництво, ітерація виробів та промисловий дизайн, 3D-моделі є найважливішими компонентами цифрового виробництва. Ось чому вибір правильного програмного забезпечення для 3D-моделювання є важливим - це допоможе реалізувати креативні ідеї з мінімальними витратами.

Пошук найкращого програмного забезпечення для моделювання - складне завдання. Існує безліч критеріїв, які допоможуть обрати між різним програмним забезпеченням, яке ідеально відповідає потребам.

Наприклад, у розробці ігор часто доводиться стикатися не тільки з грою і движком, але і з розробкою рівнів або логіки. Часто під такі завдання пишуться свої інструменти, за допомогою яких виконуються завдання, які не зручно або взагалі неможливо зробити в інших редакторах.

Щоб створювати контент для гри можна використовувати два варіанти. Перший варіант це використовувати вже готові рішення і конвертувати або використовувати результат їх роботи. Наприклад створивши модельку в існуючому редакторі її можна експортувати і використовувати в своїй грі. Якщо з якихось причин не можна реалізувати задумане - пишеться свій інструмент.

1.2 Критерії, які слід враховувати при виборі програмного забезпечення для 3d-моделювання

Існує широкий спектр програм для 3D-моделювання, призначених для різних сфер діяльності. Наприклад, існує програмне забезпечення, призначене для механічного проектування, інженерного проектування, цивільного будівництва, дизайну продукції, промислового дизайну або графічного дизайну [2]. Перше, що потрібно взяти до уваги, це обрати програмне забезпечення для розробки, орієнтоване на свій проект. Кожна сфера діяльності має різні потреби. Наприклад, проект зі створення ювелірних виробів не вимагає тих же інструментів 3D-дизайну, що і проект створення моделей літаків.

Якщо б треба було використовувати дизайнерське програмне забезпечення для 3D-друку, лазерного різання або просто для створення цифрового мистецтва, то завжди треба брати до уваги потреби технології, для якої це розробляється. Тоді можна обдумати: який бюджет і вирішити чи варто оплатити необхідну підписку. В якості альтернативи можна використовувати студентську ліцензію або освітню ліцензію, що надаються деякими програмами. В іншому випадку, існує різне програмне забезпечення для 3D-моделювання, яке доступно безкоштовно, але треба врахувати чи підходить воно для цих потреб.

Також треба обрати програмне забезпечення для 3D-моделювання, сумісне з використовуваною операційною системою (ОС), оскільки не всі пакети призначені для використання всіма ОС: Windows, Mac, Linux. І останнє, але не менш важливе: обрати програму 3D-моделювання відповідно до рівня знань.

1.3 Огляд існуючих рішень

Отже, розглянемо список найпопулярнішого програмного забезпечення для 3D моделювання.

1. Autodesk Maya

Вартість передплати: 245 доларів США / місяць. Більшість провідних анімаційних студій використовують його (наприклад Pixar) завдяки потужним інструментам. Однак, Autodesk Maya - це недешево, і користувачу потрібно буде навчитися ним користуватися, перш ніж він зможете вправно користуватися додатком.

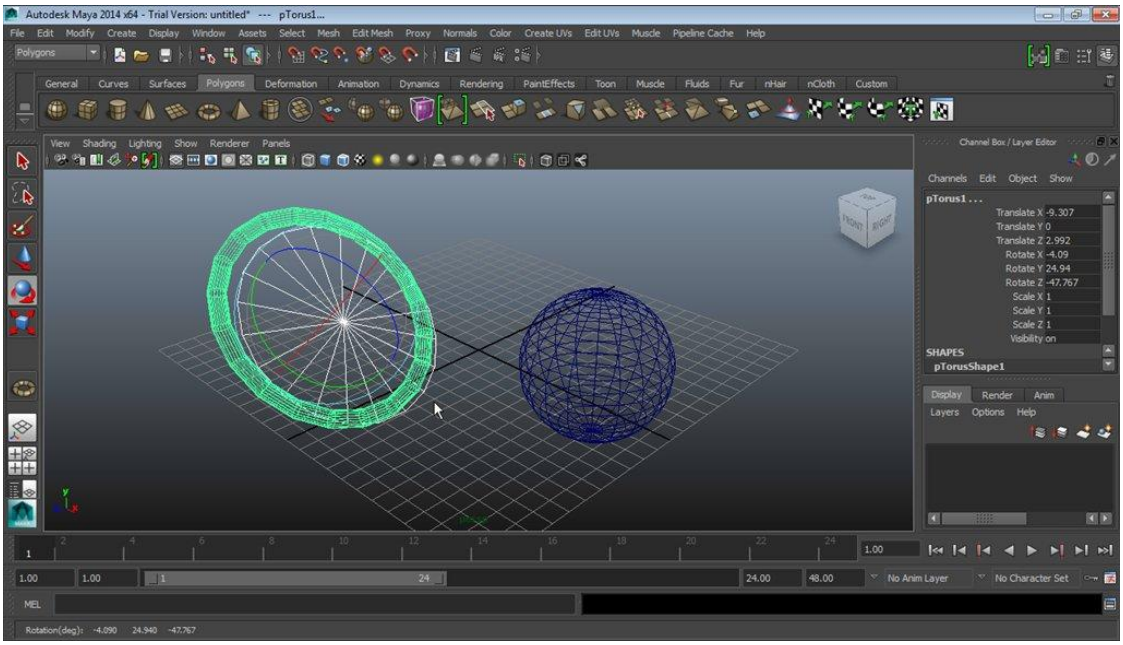


Рис. 1.1 – Autodesk Maya

2. Autodesk Mudbox

Вартість передплати: 245 доларів США / місяць. Окрім титану, відомого як Maya, Autodesk пропонує також Mudbox. Це один із найпростіших програмних пакетів для 3D-моделювання, але він більше орієнтується на

редагування та ліплення 3D-моделей за допомогою простого, інтуїтивного та тактильного набору інструментів, ніж він здатний виконувати складніші 3D завдання. Тут можна збільшувати кількість полігонів на льоту, встановлювати шари та поступово налаштовувати свої 3D-моделі, поки вони не стануть абсолютно ідеальними. Потім вбудовані функції створюють текстури, фарбують кольори, виправляють сітки та створюють звичайні карти. Також можна створювати речі з нуля, ліплячи все, що завгодно, але щоб виконати ці дії, знадобиться Maya або інше подібне програмне забезпечення для 3D моделювання.

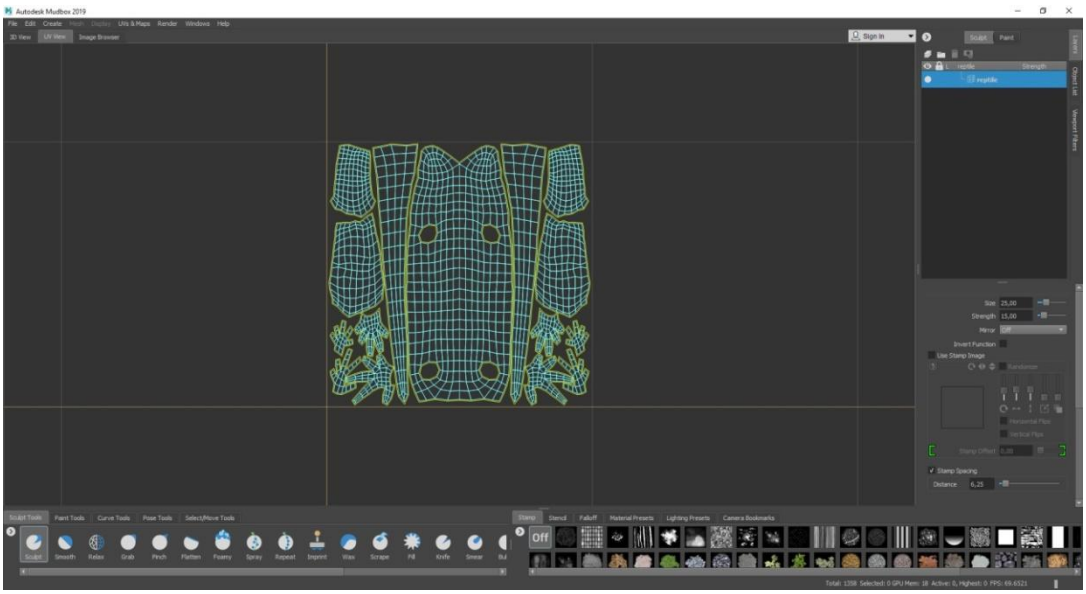


Рис. 1.2 – Autodesk Mudbox

3. Houdini

Ціна ліцензії: 1995, 4495 або 499 доларів на рік. Подібно до Autodesk Maya, Houdini - це ще один стандартний галузевий інструмент, який відноситься до числа найкращих програмних пакетів для 3D моделювання. Він використовує іншу методологію, ніж Autodesk Maya, використовуючи процедурний стиль виробництва на основі вузла, який надає художникам

величезну кількість контролю. Як і у Maya, потрібно багато часу для досягнення успіху з цим програмним забезпеченням для моделювання.

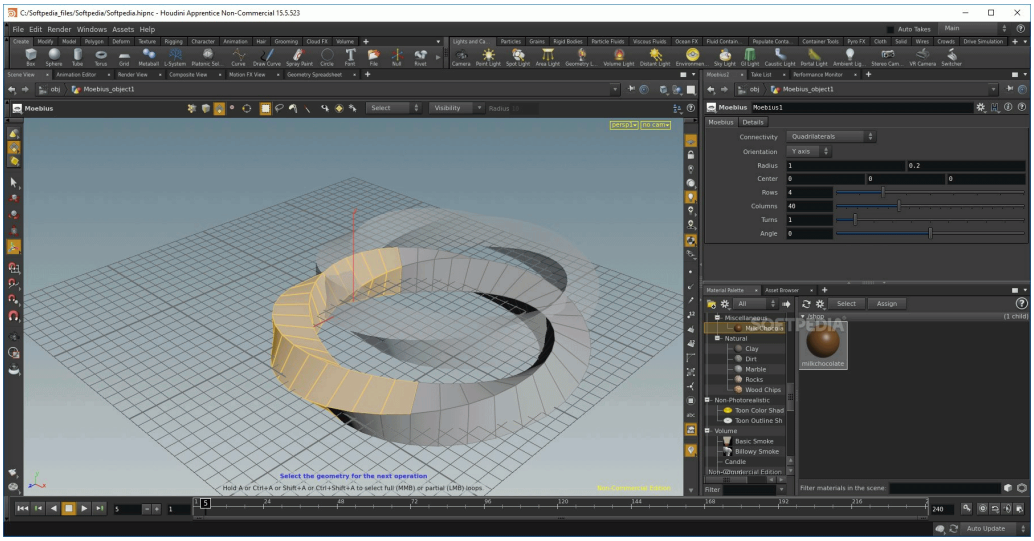


Рис. 1.3 – Houdini

4. Cinema 4D

Ціна ліцензії: $ 480 / рік до 2850 $. Cinema 4D також є серйозним конкурентом і легко входить до складу найкращих програм для 3D моделювання, які можна знайти. Цей потужний інструмент, призначений для створення ідеальної графіки руху, може конкурувати з вищезазначеними додатками. Найсильніша перемога над конкурентами є простою: навчитися користуватися ним набагато простіше.

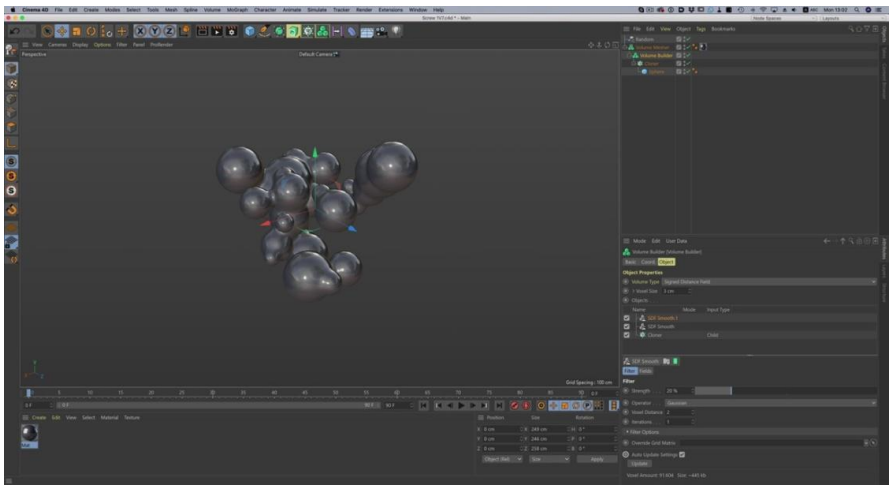


Рис. 1.4 – Cinema 4D

5. Modo

Вартість передплати чи ліцензії: від 399 доларів США на рік до 1799 доларів. Щоб виділитися з натовпу, Modo робить дещо інакше, ніж інші програми 3D-моделювання. Modo створений з урахуванням мистецтва, а не лише анімації, що призвело до того, що він пропонує досить надійний та цікавий вибір інструментів. Найбільш примітним є те, наскільки він зручний у користуванні. Хоча йому не вистачає інструментів вищого класу, пропонованих у таких програмах, як Autodesk Maya.

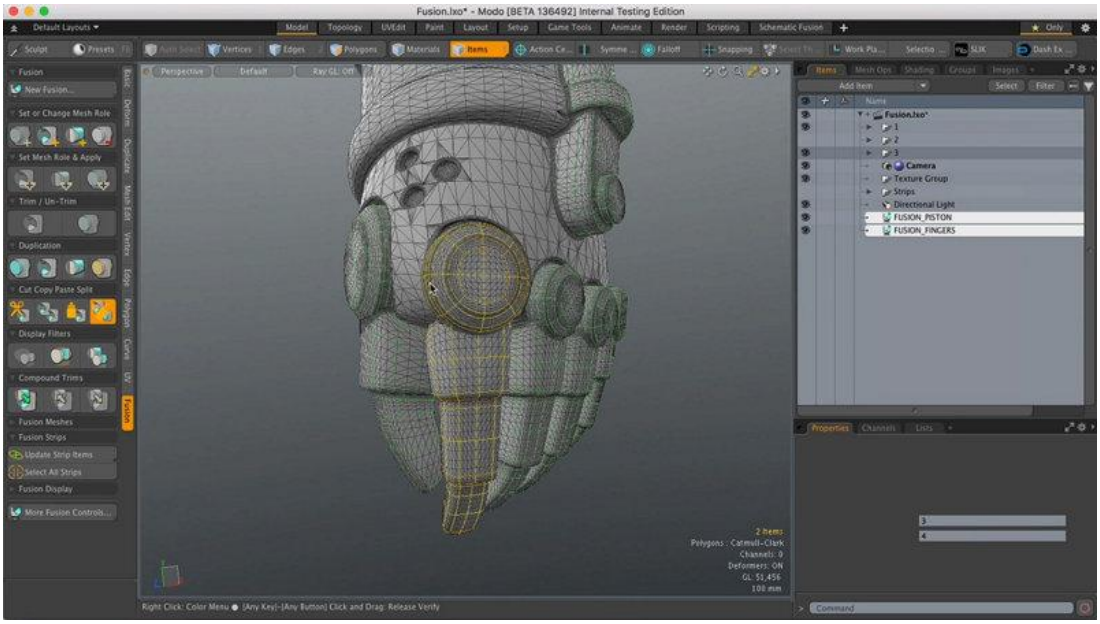


Рис. 1.5 – Modo

6. Autodesk 3Ds Max

Ціна підписки: $ 216 / місяць до $ 1740 / рік. 3Ds Max існує вже давно, що стосується програмного забезпечення для моделювання. Вона передує майже кожній іншій нинішній програмі протягом кількох років і в результаті має багато виправлень у виконанні. Це одна з найстабільніших програм 3D-моделювання і має гігантську бібліотеку.

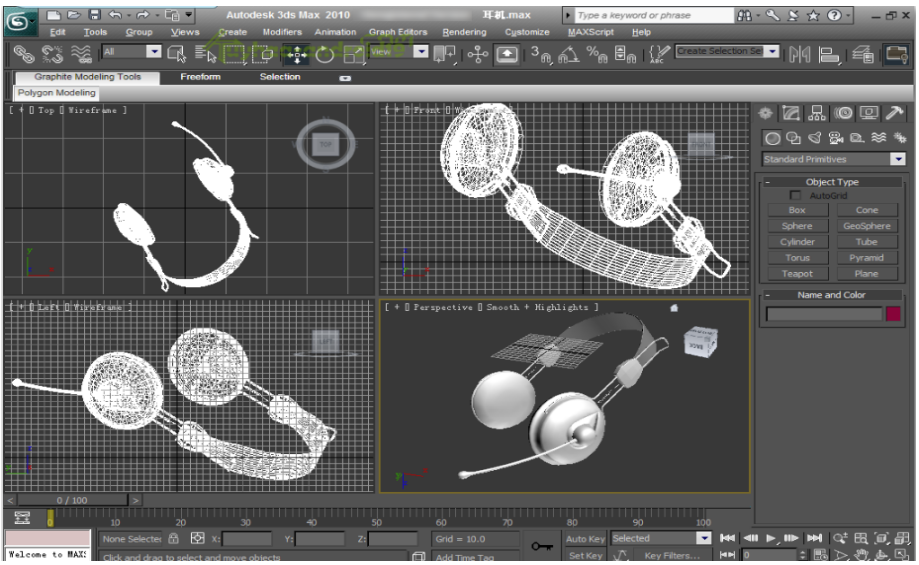


Рис. 1.6 – Autodesk 3Ds Max

7. Zbrush

Ціна ліцензії: $ 895. Що стосується скульптури дивовижних істот, людей та місць, то немає конкурентів, які б не були десь близькі до ZBrush. У кінофільмах і телебаченні - це основна програма для 3D моделювання. Ніщо не перемагає ZBrush для виготовлення вінілових іграшок чи фігур. Але знадобиться час на навчання різних інструментів та функцій.

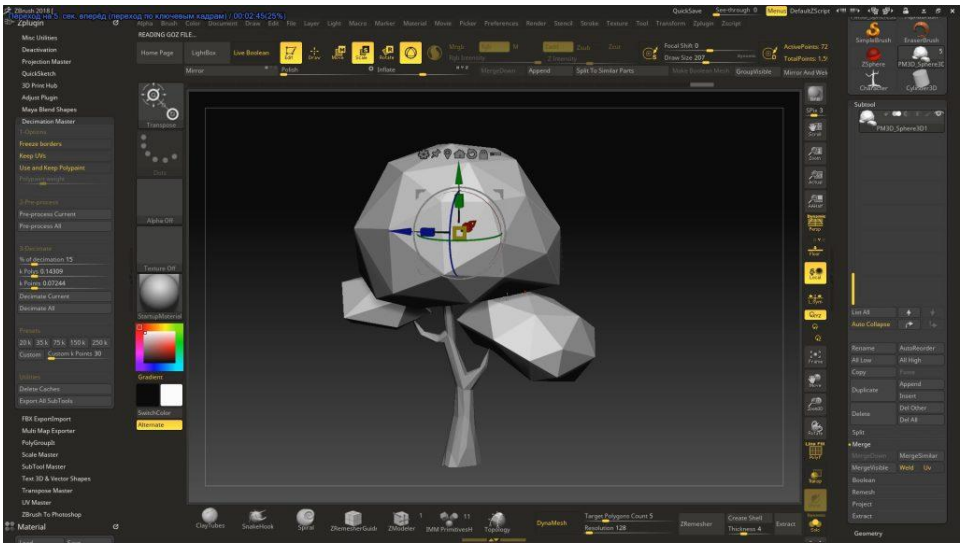


Рис. 1.7 – Zbrush

8. Rhinoceros

Ціна ліцензії: $ 995. Він сумісний практично з усім і має добре розроблений механізм візуалізації, який може обробляти навіть складні анімації без несподіваного уповільнення. Безкоштовно лише протягом 90 днів, після чого потрібно буде придбати ліцензію, щоб продовжувати користуватися можливостями та потужністю цієї програми.

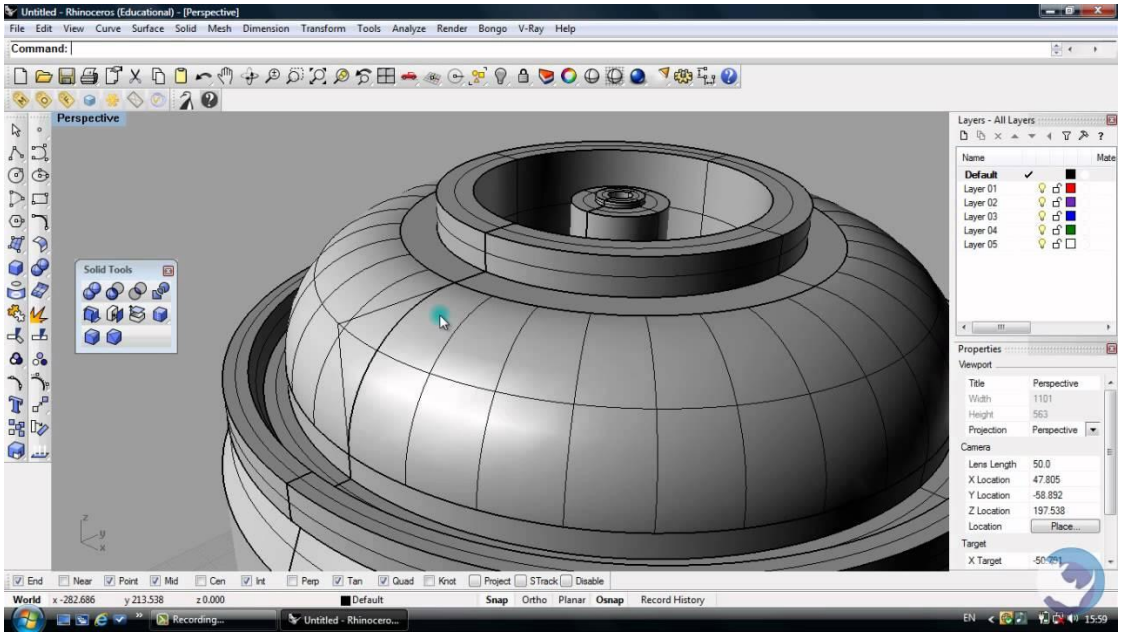


Рис. 1.8 – Zbrush

9. Substance Designer

Ціна ліцензії: $ 19,90 / місяць і вище. Існує багато кроків до створення 3D-моделі, і створення реалістичних текстур може зайняти багато часу та залучати їх. Substance Designer, одна з найпотужніших і приголомшливих програм для створення текстур. Вона може створити кілька по-справжньому дивовижних поверхонь для моделей.

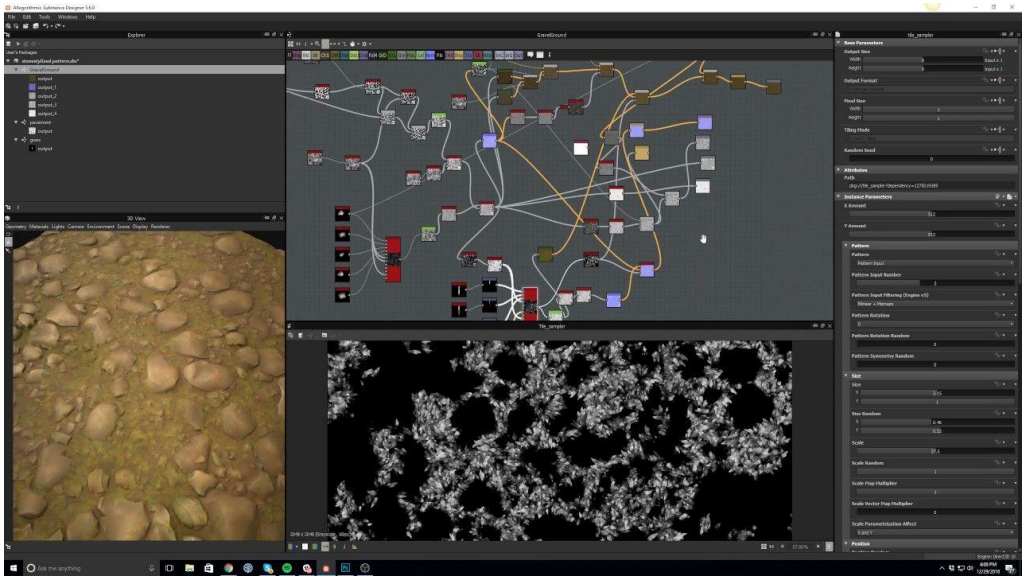


Рис. 1.9 – Substance Designer

Всі вищезгадані програми є платними, а деякі з них потребують ще й додаткових знань та часу на вивчення. Отже, є потреба в розгляді та аналізі безкоштовних рішень.

10. Blender

Король серед безкоштовного програмного забезпечення для 3D моделювання - це, без сумніву, Blender. Працюючи на всіх основних операційних системах, вона пропонує всі інструменти, які необхідні в моделюванні програмного забезпечення, включаючи такелаж, текстурування, ліплення та анімацію. Додаткова перевага це відкритий код. Це означає, що програма постійно вдосконалюються, а доступні додатки для нових функціональних можливостей не тільки поширені, але завжди безкоштовні. Але графічний інтерфейс цієї програми далекий від інтуїтивно зрозумілого, що означає, що початківцям буде трохи складно одразу зануритися і почати працювати в ній.

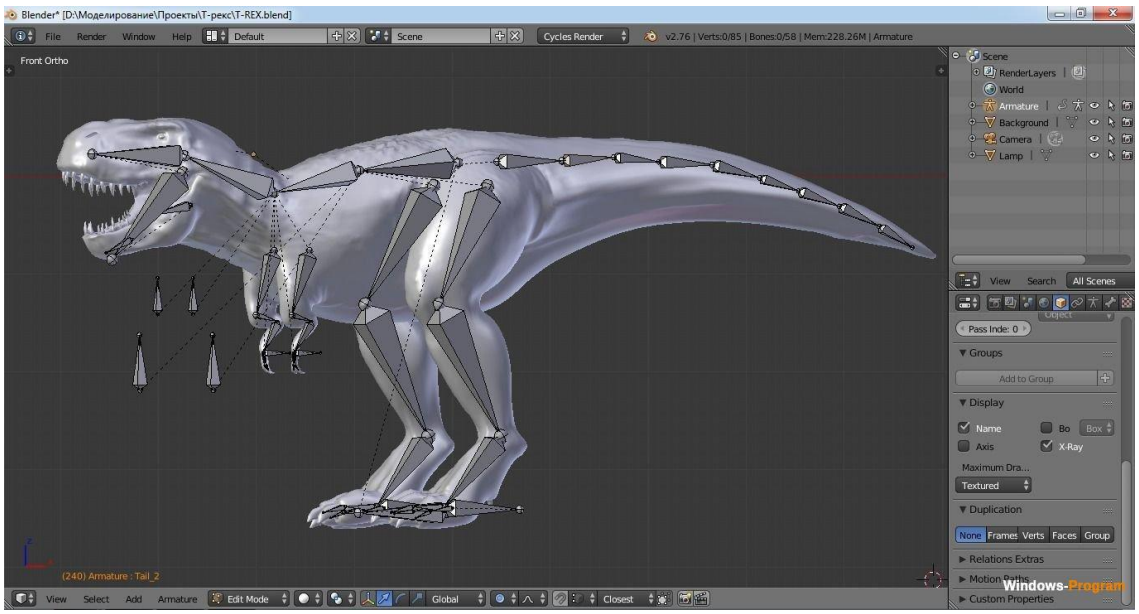


Рис. 1.10 – Blender

11. Daz Studio

Нещодавно зроблений додаток безкоштовний для всіх, Daz Studio доступний як для новачків, так і для досвідчених 3D-модельєрів, і зосереджений на створенні мистецтва, використовуючи людей, тварин та інші активи зі свого списку. На відміну від більшості інших, він більше схожий на інструмент для постановки, ніж для створення високоякісних 3D-моделей для виробництва. Незважаючи на те, що сама програма є абсолютно безкоштовною, багато ресурсів на їх ринку – платні.

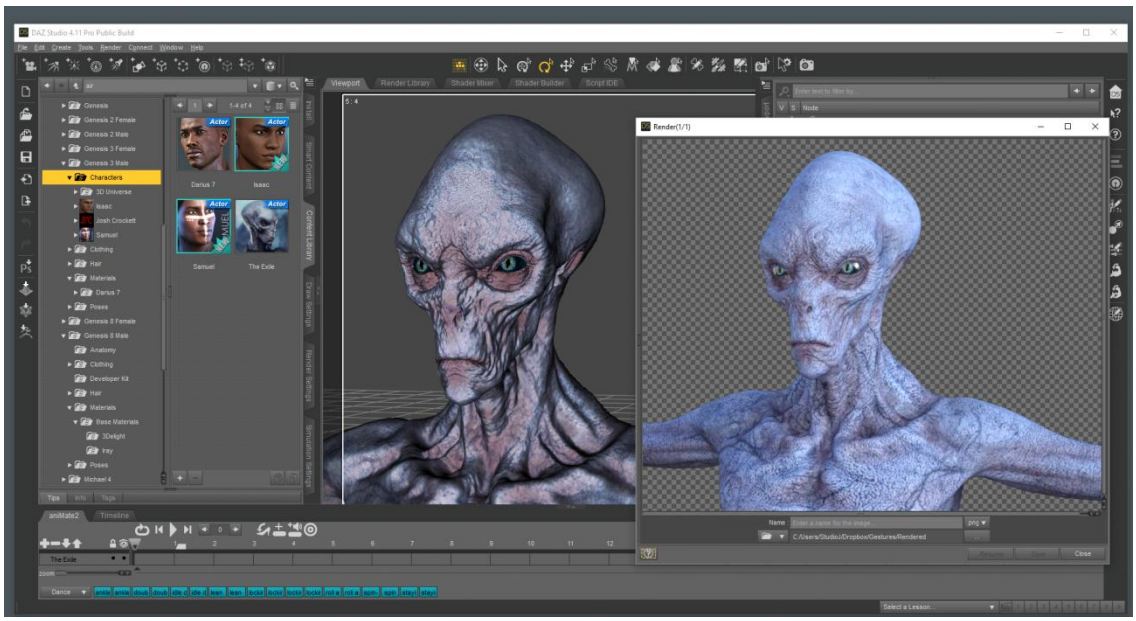


Рис. 1.11 – Daz Studio

12. SkethupFree

Напрочуд легкий варіант програмного забезпечення для 3D-моделювання, SketchUp працює у всіх основних операційних системах. В ньому можна малювати, встановлювати орбіти та комбінувати елементи для створення справжнього 3D-мистецтва з легкістю за допомогою природного підходу на основі ескізів. Він дуже зручно для тих, хто хоче стати дизайнером інтер’єру.

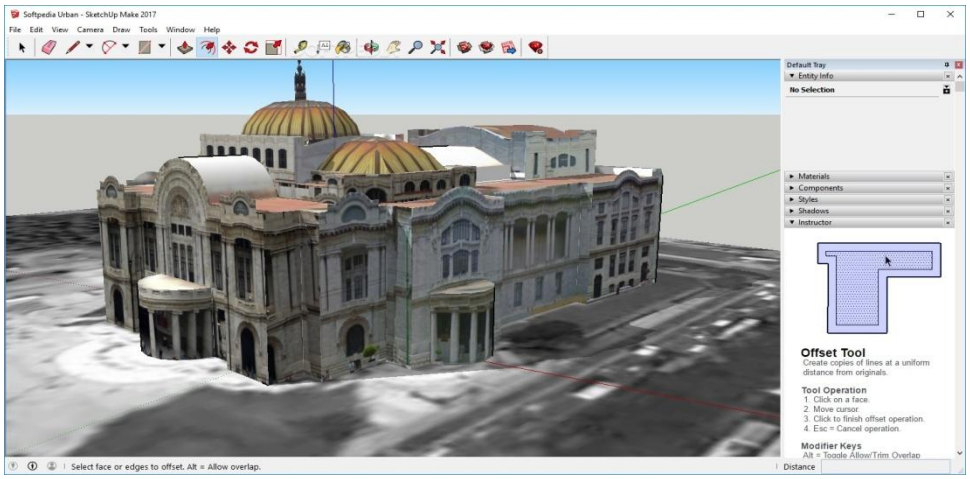


Рис. 1.12 – SkethupFree

13. Sculptris

Це абсолютно найкраще безкоштовне програмне забезпечення для ліплення. Протягом декількох коротких хвилин можна з'ясувати спосіб роботи елементів керування. Хоча він може створити кілька справді акуратних моделей, все одно знадобиться інша 3D програма, щоб максимально використати її.

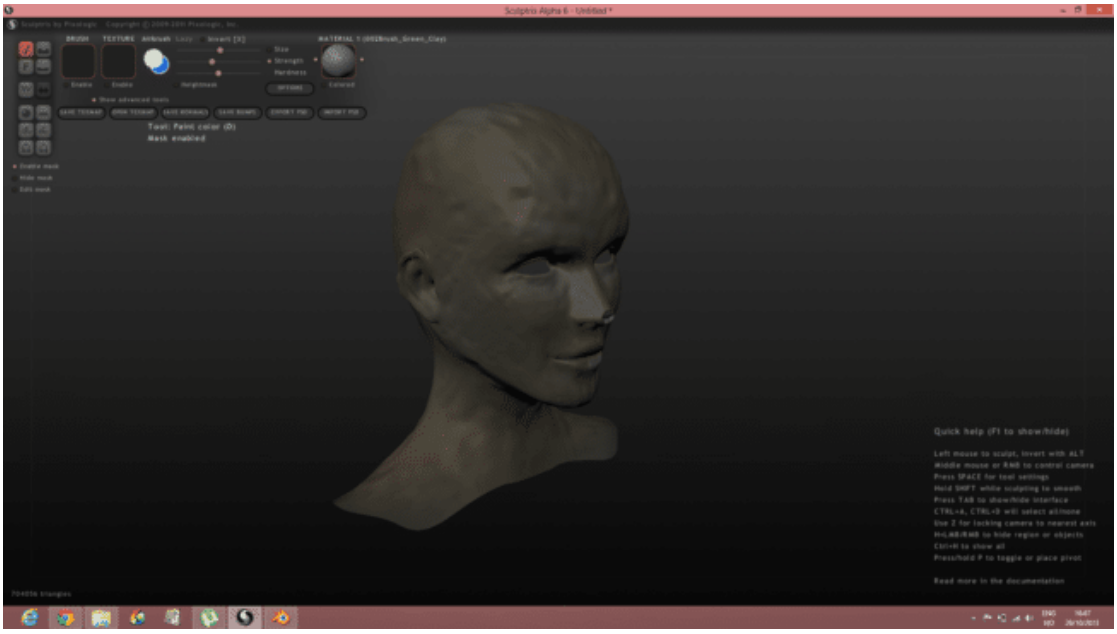


Рис. 1.13 – Sculptris

14. Houdini Apprentice

Houdini - насправді одне з найкращих програмних забезпечень для 3D-моделювання. Оснащений усією потужністю та гнучкістю, пропонованою платною версією програми, Houdini Apprentice дозволить навчитися використовувати всі її функції, не витрачаючи 2 000 доларів, необхідних для професійної версії, але без ліцензії все одно потрібно буде придбати повну версію, щоб використовувати її в комерційних цілях.

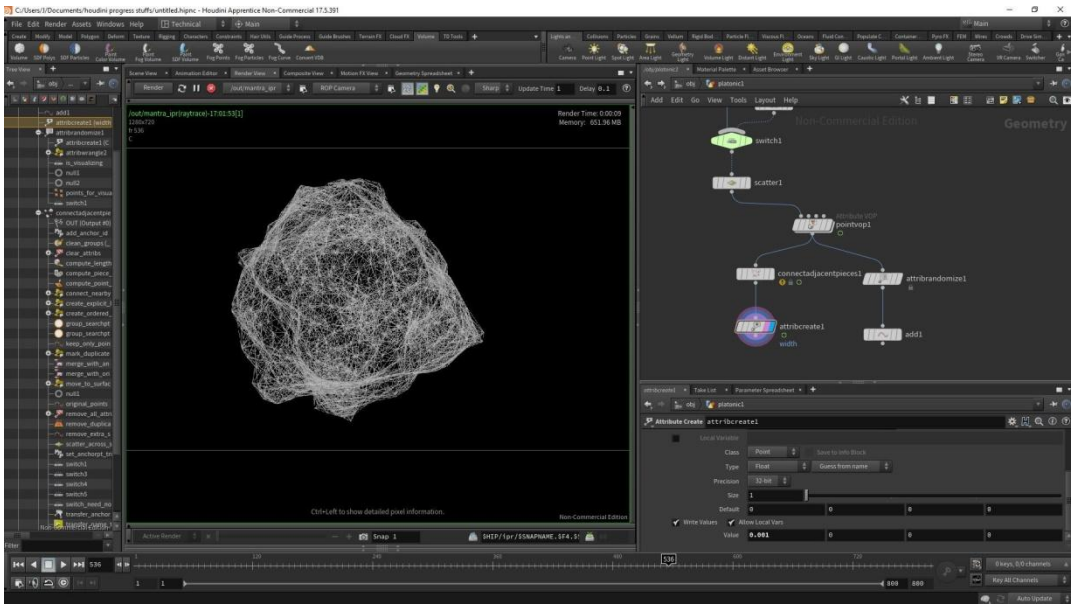


Рис. 1.13 – Houdini Apprentice

1.4 Результати аналізу

Проведений аналіз існуючих рішень систем редагування показав, що, незважаючи на досить широкий спектр сучасних типових засобів редагування, представлених на ринку та їх потужні функціональні можливості, не один продукт не задовольняє поставлених вимог.

В деяких випадках, ціна занадто висока, в інших немає можливості використовувати продукт на операційний системі Linux.

Особливу увагу потрібно приділити інтерфейсу більшості розглянутих рішень. Для користувача, котрий немає досвіду у 3D моделюванні або у використанні схожих додатків, буде досить складно одразу розпочати редагування та зрозуміти принцип роботи продукту.

У зв'язку з результатами, було прийнято рішення про розробку власного програмного забезпечення для системи 3D моделювання, що має простий графічний інтерфейс, а також можливість використання на Linux.

Висновки до розділу 1

Аналіз існуючих програмних забезпечень з реалізацією системи 3D моделювання показав, що на даний момент серед них немає такого, що відповідає всім вимогам. Також після аналізу сформулювалося більш чітке розуміння потрібного функціоналу та інтерфейсу для користувачів.

Отже, можна зазначити, що для розробки необхідного програмного забезпечення для системи 3D моделювання треба реалізувати наступний функціонал:

1. Відображення, редагування та збереження 3D моделі.
2. Зрозумілий графічний інтерфейс.
3. Сумісність з оперативною системою Linux.

РОЗДІЛ 2

МЕТОД ТА ПРОГРАМНІ ЗАСОБИ ПОБУДОВИ

2.1 Вступ

Інтерес до тривимірного пошуку форми в даний час зростає, що обумовлено двома важливими причинами - швидким збільшенням обсягу мультимедійних даних і помітним прогресом в області комп'ютерного обладнання та програмного забезпечення в останні роки.

В даний час можливо отримати складні тривимірні моделі за розумний проміжок часу завдяки використанню складних алгоритмів опису тривимірних фігур, що було немислимо кілька років тому. Основним питанням є ефективність підходів, які повинні працювати як швидко, так і надійно.

Тривимірні об'єкти можуть бути згенеровані автоматично або створені вручну шляхом деформації, наприклад, шляхом маніпулювання вершинами. В процесі тривимірного моделювання створюється цифровий об'єкт, здатний повністю бути анімованим, що призводить до його використання для анімації персонажів і спеціальних ефектів.

Ядром моделі є сітка, яку найкраще описати як сукупність точок у просторі. Ці точки відображаються в тривимірній сітці і об'єднуються у вигляді багатокутників, зазвичай трикутників або квадратів.

Кожна точка або вершина має свою позицію на сітці, і шляхом об'єднання цих точок в форми створюється поверхня об'єкта.

Моделі часто експортуються в інше програмне забезпечення для використання в іграх або фільмах. Але деякі програми 3D-моделювання дозволяють створювати 2D-зображення, використовуючи процес, званий 3D-рендерингом. Ця техніка відмінно підходить для створення гіперреалістичність сцен з використанням складних алгоритмів освітлення.

Отже, необхідно оцінити та порівняти алгоритми опису тривимірної моделі, щоб визначити, який з них найбільш ефективний.

2.2 Опис алгоритмів для моделювання

3D моделювання використовується для створення 3D-моделей для секторів, починаючи від інженерії та виготовлення до цифрової анімації для фільмів та відеоігор. Перші використання комп'ютерної графіки були на початку 1960-х для наукових та інженерних цілей, а художнє вираження CGI почалося в кінці 1960-х. Перша комерційно доступна програма для твердого моделювання під назвою Syntha Vision була випущена в 1969 році. Лише через 20 років на сцені з'явилися NURBS і параметричне моделювання, остання - народження Pro / ENGINEER. 3D моделювання зростало у популярності та корисності, із застосуванням 3D моделювання, починаючи від кіно та відеоігор, закінчуючи всіма аспектами комерційного дизайну та виготовлення.

Перша технологія 3D-друку, стереолітографія (SLA), з'явилася в 1986 році і надає свою назву популярному тепер файлу STL. 3D-моделювання - це процес створення 3D-об'єкта за допомогою програм 3D-моделювання. Існує декілька загальних методів 3D моделювання, які перераховані нижче.

2.3 Основні алгоритми

2.3.1 Полігональне моделювання

Полігони складаються з геометрії на основі вершин, країв і граней, які можна використовувати для створення тривимірних моделей. Полігони корисні для побудови багатьох типів 3D-моделей і широко використовуються при розробці 3D-вмісту для анімаційних ефектів у фільмі, інтерактивних

відеоіграх та Інтернеті. Полігони - це прямобічні форми (3 і більше сторін), визначені тривимірними точками (вершинами) та прямими лініями, що їх з'єднують (ребра) [3]. Внутрішня область багатокутника називається обличчям. Вершини, краї та грані є основними складовими багатокутників. Ви вибирають та змінюють багатокутники за допомогою цих основних компонентів.

При моделюванні з багатокутниками зазвичай використовують тристоронні багатокутники - трикутники, або чотиристоронні багатокутники, які називаються чотирикутниками (квадратиками). Також можливе створення багатокутників з більш ніж чотирма сторонами, але вони не так часто використовуються для моделювання.

Окремий багатокутник зазвичай називається грань і визначається як область, обмежена трьома або більше вершинами та пов'язаними з ними ребрами. Коли багато граней з'єднані разом, вони створюють мережу граней, званих багатокутною сіткою (також її називають полісетом або полігональним об'єктом). Отже, створюються 3D моделі, використовуючи багатокутні сітки.

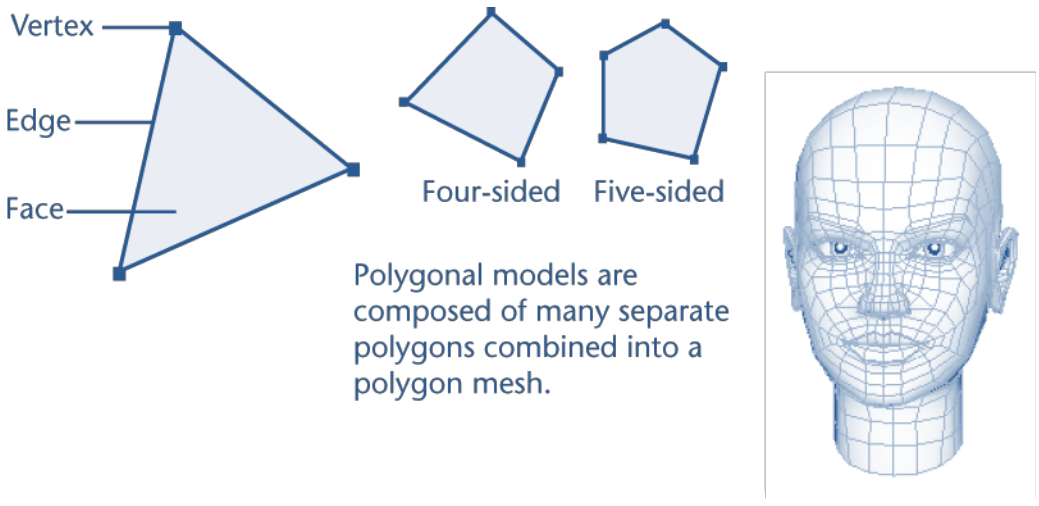


Рис. 2.1 – Полігональне моделювання

2.3.2 Моделювання кривих nurbs

NURBS – Non-Uniform Rational B-Splines (Неоднорідний раціональний B-сплайн) - це один тип геометрії, який можна використовувати для створення 3D-кривих та поверхонь [4].

Non-Uniform (неоднорідний) відноситься до параметризації кривої. Неоднорідні криві дозволяють наявність багатовузлів, які потрібні для представлення кривих Безьє.

Rational (раціональний) відноситься до основного математичного представлення. Ця властивість дозволяє NURBS представляти точні канонічні перерізи (такі як параболічні криві, кола та еліпси) на додаток до кривих вільної форми.

B-сплайни - це кускові поліноміальні криві, які мають параметричне зображення.

NURBS корисні для побудови багатьох видів органічних 3D-форм через плавний та мінімальний характер кривих, які вони використовують для побудови поверхонь. Типи поверхні NURBS широко використовуються в галузі анімації, ігор, наукової візуалізації та промислового дизайну.

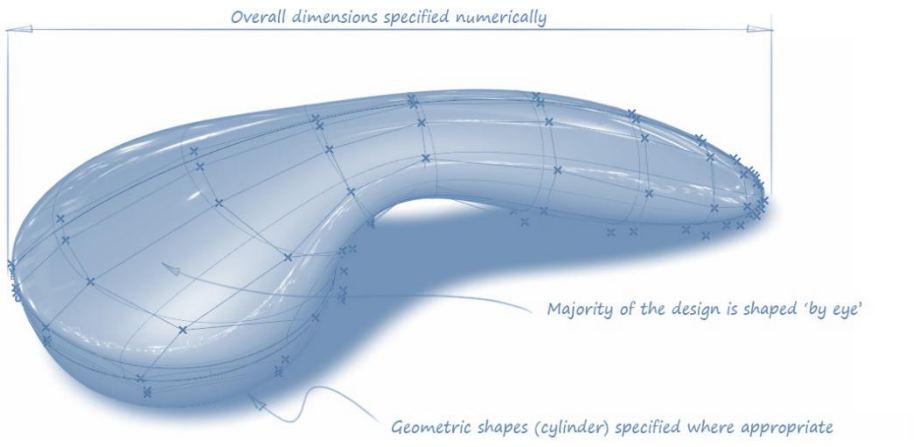


Рис. 2.2 – Моделювання кривих NURBS

2.3.3 Процедурне моделювання

Процедурне моделювання буває різних форм і розмірів, є два типи моделювання. Перший - на інструментальній основі. Ми або хтось інший створили інструмент, призначений для процедурного генерування купи подібних об’єктів. Наприклад, у нас може бути генератор будівлі. Тоді ми могли б ввести купу таких параметрів, як кількість поверхів, високу стелю і яку форму має мати дах. Потім ми кілька разів запускаємо програму, і щоразу з’являється нова модель, яка відповідає нашим критеріям.

Наступний вид процедурного моделювання тісно пов'язаний із затіненням. Шейдер може мати зсув результату, і завдяки цьому зсуву беручи такі прості примітиви, як сфера чи площина, і використовуючи математичні формули для деформації поверхні, вони можуть стати складним об'єктом чи поверхнею. Це тенденція, яка зростає, коли все більше і більше інструментів стали доступними для витіснення геометрії за допомогою затінення.

Доступні як традиційні переміщення, що працюють на осі вгору і вниз, так і переміщення вектора. Вектор зміщення може зміщувати геометрію в усіх напрямках, створюючи дуже прогресивні об'єкти з простої геометрії.

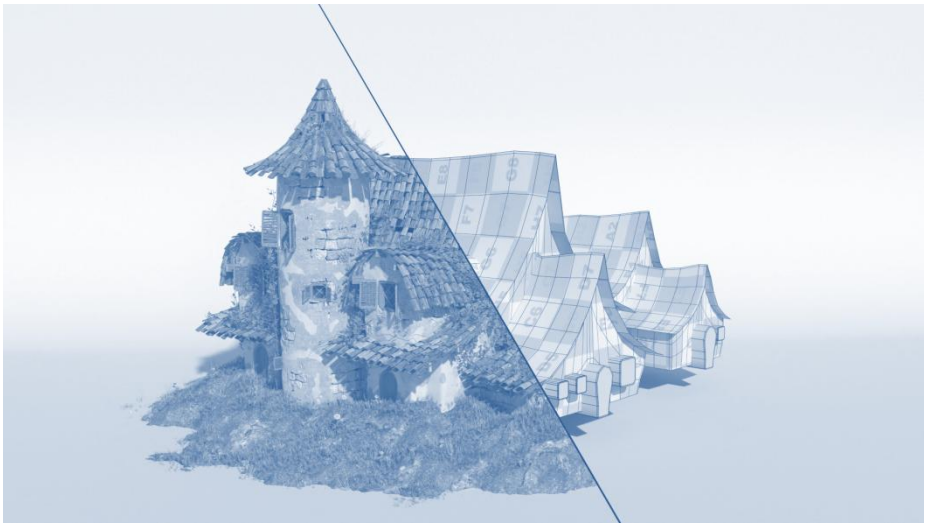


Рис. 2.3 – Процедурне моделювання

2.3.4 Сплайнове моделювання

Сплайни (Spline - кусочно-поліноміальна функція) - це двовимірні геометричні об'єкти. З них можуть бути створені більш складні тримірні тіла, але також вони можуть бути абсолютно самостійними. Сплайнами є різноманітні лінії, форма лінії визначається типом вершин, через які вона проходить [5].

Сплайни це як і найпростіші геометричні фігури (прямокутники, зірки, еліпси і ін.), так і складні ламані або криві, а також контури текстових символів. Спочатку будується сплайновий каркас, для створення моделі за допомогою тривимірних кривих, а далі на основі цього каркасу огинаюча тривимірна геометрична поверхня. Гладкість кривої визначає набір тривимірних контрольних точок, які задають тримірні криві. Крім того, в сплайновому моделюванні використовуються сплайнові примітиви (лінії, дуги, спіралі, кола, кільця, еліпси, прямокутники, багатокутники). Об’єкти, створені сплайновим моделюванням, є гнучкі до редагування та зміни їх форми.

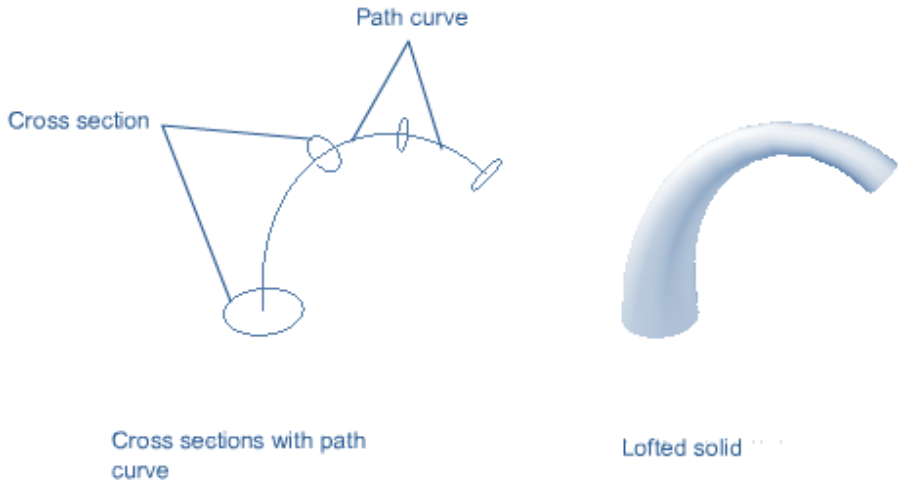


Рис. 2.4 – Сплайнове моделювання

2.3.5 Точне моделювання в сапрах (система автоматизова-нного проектування)

Це моделювання ґрунтоване на операціях, що по черзі здійснюються над тілом або поверхнею. Модель задається не полігонами, а математичними формулами . Відмінність між полігональним моделюванням і моделюванням в САПРах аналогічно відмінності растрової графіки від векторної. Щоб отримати гладку модель полігонами, треба збільшувати кількість полігонів - і все одно на якомусь рівні модель матиме нерівність. У САПРах же будь-яка криволінійна поверхня абсолютно гладка при будь-якому наближенні, оскільки задається математично.

Через те, що модель задається математично, можна моделювати з точністю до міліметра.

Ще одна особливість САПРов - можливість створювати параметричні моделі. Це означає, що модель створюється послідовністю дій. У будь-який момент можна змінити параметри, які були задані (наприклад, висота і діаметр циліндра). При правильно побудованій моделі можна міняти будь-який параметр і модель автоматично перебудується.

По суті, при побудові створюється алгоритм моделі, який можна змінити у будь-якому місці. Це дуже зручно і при простому проектуванні, коли треба виправити свої помилки, так і у тому випадку, коли потрібно змоделювати багато подібних один до одного об'єктів.

Проте САПРи не призначені для створення складних органічних моделей, як, наприклад, людське тіло. Теоретично можливо створити в них простого персонажа, але це займе величезну кількість часу і зусиль, незрівняну з полігональним моделюванням чи скульптингом.

САПР часто використовується як допоміжний інструмент у створенні, модифікації, аналізі або оптимізації конструкцій, що додатково виготовляються. Однак базове програмне забезпечення для САПР часто є недостатнім для проектування складних об'єктів.

Нинішня система САПР розроблена насамперед для використання у звичайних технологіях виготовлення, де достатньо простих кіл та прямих ліній.

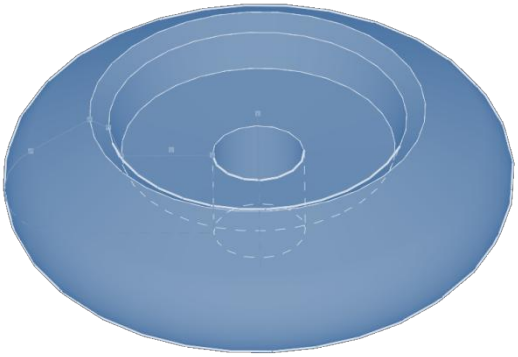


Рис. 2.5 – Точне моделювання в САПРах

2.4 Використання 3d моделей у сучасному програмному забезпеченні

Оскільки для зручностей 3D моделі якось потрібно підігнати під загальноприйняті стандарти, на певному етапі їх переводять в полігональні моделі, а далі все обчислюється стандартними методами, які підтримуються як програмно, так і апаратно відеоадаптерами.

Для прикладу сучасного конвеєра візуалізації можна взяти наступну стандартну послідовність дій.

Полігональний конвеєр візуалізації:

Локальний простір. Моделювання кожного об'єкта в його системі координат.

Світовий простір. Збірка безлічі моделей в рамках світової системи координат.

Перетворення в простір виду. Перехід до системи координат, пов'язаної з видом проекції (розташуванням віртуальної камери, яка стає початком координат, причому вісь Z стає прямо спрямованою від глядача).

Видалення невидимих поверхонь.

Освітлення.

Відсікання в рамках видимої області екрану. Тобто, прибираються ті трикутники, які виходять за межі поля видимості віртуальної камери, а ті, через які проходить межа зони видимості, розбиваються на частини (видиму і невидиму).

Перехід від 3D-представлення до 2D. Ця операція називається "проекцією".

Перетворення порту перегляду, а саме - перехід до системи координат виведення на екран. Наприклад, якщо не повноекранний режим, а віконний, або передбачена спеціальна клієнтська область.

Растеризация, тобто, зафарбовування двомірних трикутників.

Тепер потрібно зупинитися на пункті 4. Справа в тому, що відсікання невидимого може відбуватися декількома принципово відмінними методами. Одні засновані на відсікання невидимих граней за принципом лицьова / лицьові (робиться або по порядку обходу вершин полігонів, або з обчисленням нормалей граней). Інші варіанти передбачають растеризування і визначення найближчих пікселів до глядача (z-буферизація, порядкове сканування). Іноді використовують і те, і інше в гібриді.

Виходячи з полігонального подання розраховується і освітлення. Все здається досить простим. Але якщо замість полігонів залишатиметься наприклад NURBS-уявлення, то потрібна нова, особлива алгоритмічна база. У цьому полягає основна складність.

Отже, можна зробити висновок, що полігональне моделювання підійде, якщо: моделюється чисто художня річ не для серійного виробництва і не потрібні точні розміри; модель не для виробництва, а, наприклад, для ігор, анімації або рендеру картинки; потрібна якісна художня обробка моделі. Наприклад, потрібно зробити виразний дизайн ігрового персонажа.

2.5 Принципи полігонального моделювання

2.5.1 Основні елементи полігональних моделей

Суть полігонального моделювання дуже проста. Моделі такого типу складаються з трьох основних елементів (примітивів): вершин, ребер і багатокутників. Для маніпулювання даними елементами розроблено кілька методів, які детально розглядаються нижче.

1. Вершини

Вершина - це одновимірний об'єкт, точка, розташована в просторі (рис. 2.6). Якщо з'єднати дві вершини, вийде ребро (рис. 2.7).

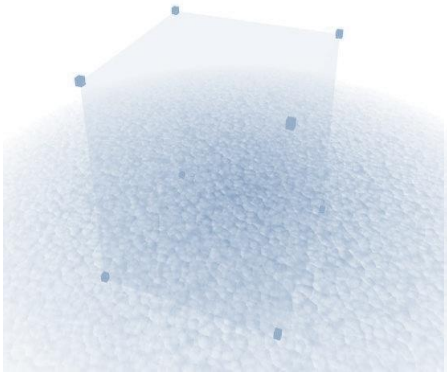


Рис. 2.6 – Вершина

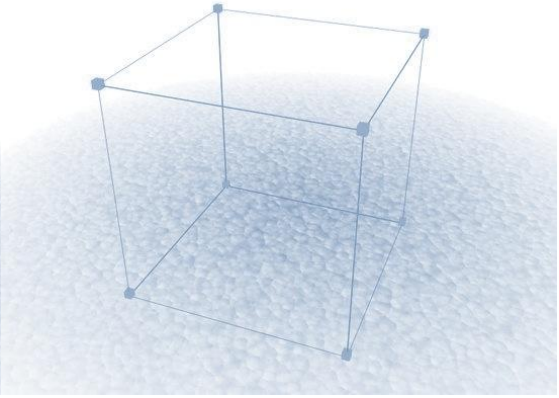


Рис. 2.7 – Ребро

Деякі програми моделювання пропонують додаткові операції над вершинами, наприклад Extrude (Екструдування) і Bevel (Формування скоса), показані на рис. 2.8 і 2.9 відповідно; в обох випадках створюються додаткові ребра і багатокутники.

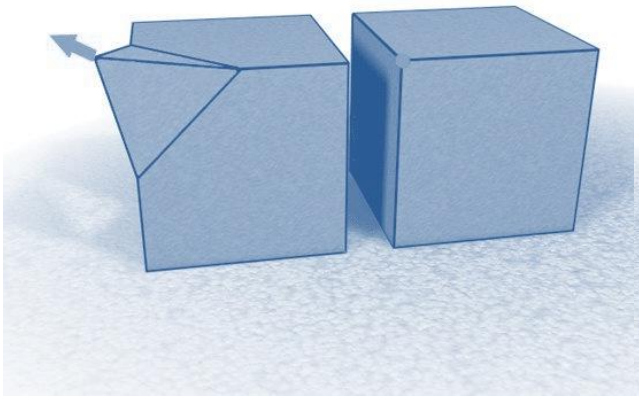


Рис. 2.8 – Екструдування вершини

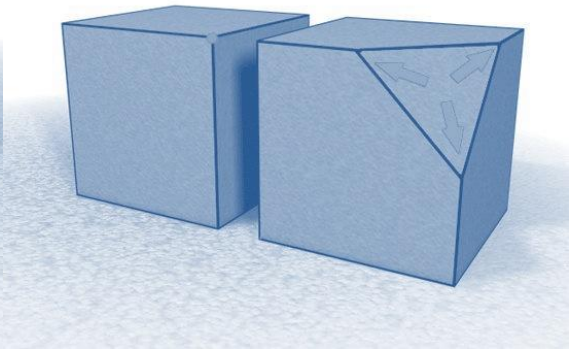


Рис. 2.9 – Формування скоса вершини

2. Ребра

Ребро визначається двома вершинами. Коли об'єкт зображується у вигляді дротяного каркаса, всі його ребра видно. По суті, вони являють собою лінії і є двовимірними об'єктами. Три ребра або більш утворюють багатокутник. Над ребрами можна виконувати ряд операцій. В їх число входять вже названі Extrude і Bevel, операція Collapse (Стягування), яка видаляє ребро, перетворюючи його в вершину. Крім того, для ребер передбачена операція Cut / Connect (Розрізування / З'єднання): існуючі ребра розрізаються навпіл, а точки розрізу з'єднуються новим ребром. Результати виконання цих операцій показані на рис. 2.10-2.13.

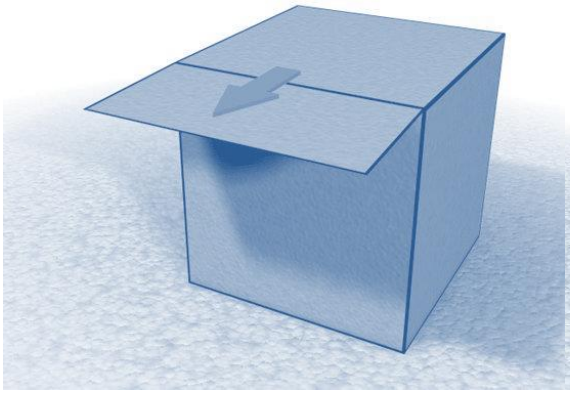


Рис. 2.10 – Екструдування ребра

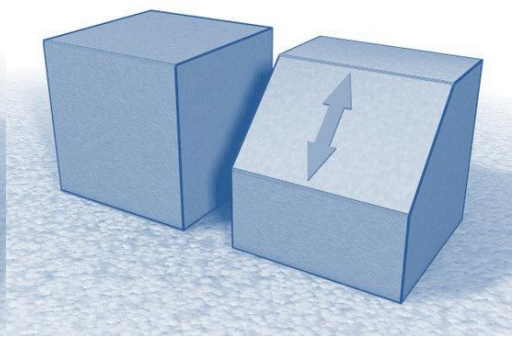


Рис. 2.11 – Формування скоса ребра

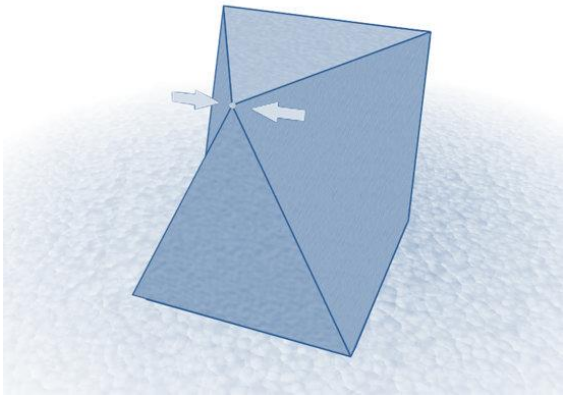


Рис. 2.12 – Стягування ребра

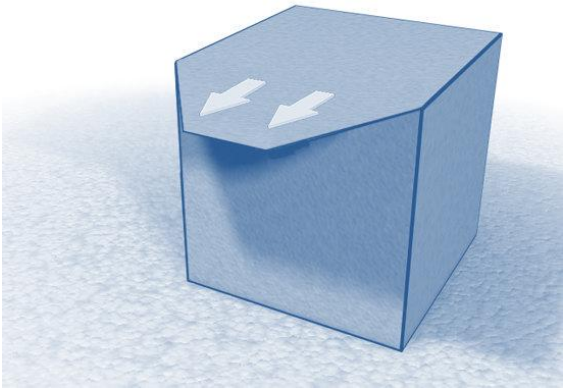


Рис. 2.13 – Розрізування / З'єднання ребра

3. Багатокутники

Багатокутник визначається трьома або більше ребрами. По суті він є плоским тривимірним об'єктом. Для виконання рендеринга поверхня повинна складатися з багатокутників.

Більшість пакетів дозволяє працювати з багатокутниками, що мають більше трьох ребер, проте зазвичай в процесі візуалізації такі примітиви все одно розбиваються на трикутники.

Над багатокутниками можна виконувати наступні операції. Проілюстровані на рис. 2.14-2.16 операції Extrude (Екструдування), Inset (Вставка) і Bevel (Формування скоса) є варіантами однієї і тієї ж операції - створення додаткового багатокутника на основі того чи іншого вихідного елемента.

Відмінність полягає в тому, що відбувається з вихідним багатокутником. При екструдуванні частина його висувається назовні або переміщається всередину зазвичай вздовж нормалі до поверхні (тобто уздовж перпендикуляра до межі).

Коли виконується вставка, масштаб вихідного багатокутника, як правило, зменшується, і отримана фігура розташовується в площині вихідної.

Операція формування скоса по суті є комбінацією перших двох, оскільки вихідний елемент масштабується і переміщується. В результаті з'являються нові грані.

Крім того, до багатокутників застосовується операція Collapse (Стягування), яка видаляє багатокутник, стягуючи його в одну вершину (рис. 2.17).

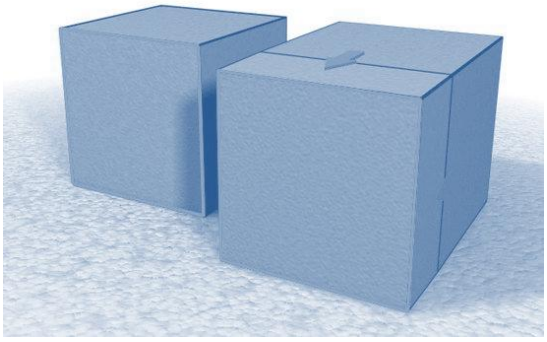


Рис. 2.14 – Екструдування багатокутника

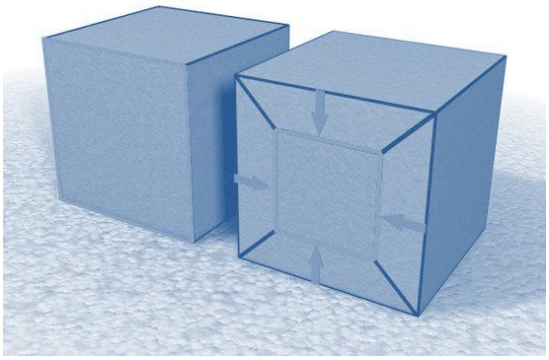


Рис. 2.15 – Операція Вставка

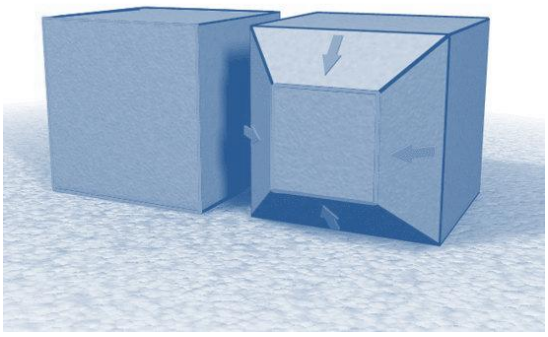


Рис. 2.16 – Формування скоса багатокутника

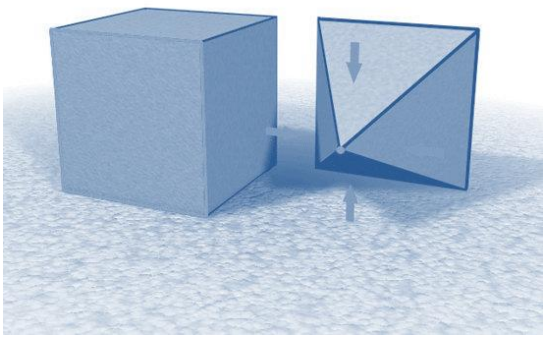


Рис. 2.17 – Стягування багатокутника

Зазвичай створюється будь-який базовий об'єкт, наприклад паралелепіпед, циліндр або сфера. Потім він модифікується за допомогою

ряду операцій і набуває більш складну форму, відповідну контурах персонажа.

При конструюванні полігональних поверхонь потрібно починати з невеликої кількості елементів і потім додавати їх в міру необхідності. Багато розробників починають з примітивів: паралелепіпеда або сфери, - а потім надають поверхні бажану форму, вводячи нові деталі.

2.5.2 Булеві операції

Булеві полігональні операції не користуються популярністю серед 3D-моделерів, оскільки вони переконфігуровують сітку і часто ця конфігурація не є топологічно правильною. Це означає, що коли робиться згладжування сітки, керованої булем, вона, ймовірно, матиме деякі недоліки.

Іноді навіть не потрібно згладжувати сітку, щоб побачити її вади. Однак, за деяких обставин, булеві операції можуть бути корисними (інакше їх би не включали до більшості 3D-пакетів, як інструмент). У моделерів є просте правило з цього приводу: «Якщо ви можете уникнути використання булевих операцій, уникайте їх, але якщо ви не можете, не забудьте зробити все можливе, щоб перевірити та виправити всі вади вашої сітки».

Булева операція - це така операція, яка, задаючи об'єкт A і B (як показано на рис. 2.18), створює нову форму. Булеві операції включають віднімання, об'єднання та перетин.

Рис. 2.18 – Булеві операції

2.5.3 Операція subdivide для поверхонь. Згладжування

Більшість розробників персонажів для кіно або відеофільмів прагнуть до того, щоб їх моделі мали згладжену поверхню і виглядали якомога природніше. На перший погляд здається, що цього важко досягти, оскільки додаткові елементи ускладнюють маніпулювання поверхнею. На щастя, є спеціальні інструменти, які дозволяють майже автоматично перетворити модель з низьким дозволом в модель з високою роздільною здатністю, якщо вихідний зразок має впорядковану конструкцію. Цей метод поділу багатокутників (або ущільнення каркаса) називається smoothing (згладжування).

В процесі дроблення полігональної поверхні створюються нові елементи, які необхідно правильно розмістити. Формування додаткових багатокутників регулюється низкою правил.

В теорії все виглядає просто: три або чотири точки, що не лежать на одній прямій, інтерполюються кривої більш високого порядку, ніж пряма, і в результаті виходить така ж апроксимація поверхні, як і при використанні патчів.

Найпростіший спосіб обчислити місцеположення додаткових елементів - відсікти певні кути. Якщо за допомогою математичних формул округлювати різкі виступи на моделі віртуального героя, її поверхня стане більш гладкою. Метод, що дозволяє це зробити, називається відсіканням кутів.

Беруться серединні точки або центри багатокутників, що утворюють поверхню об'єкта, і з'єднуються шляхом створення проміжних багатокутників: одна фігура перетворюється в чотири, чотири - в шістнадцять і т.д. На кожному кроці відбувається відсікання кутів, в результаті чого об'єкт стає більш гладким. Одна з цікавих властивостей даної операції полягає в тому, що якщо виконувати її нескінченно довго, вийде така ж поверхня, як при використанні патчів на основі B-сплайнів.

Звісно, нескінченну кількість операцій не може виконати навіть найпотужніший комп'ютер, так що потрібно обмежуватися двома або трьома проходами. Зазвичай цього достатньо, щоб поверхня виглядала на екрані абсолютно гладкою.

Єдина проблема, що виникає в результаті дроблення полігональної поверхні, полягає в тому, що додаткові деталі «обтяжують» модель персонажа. Імітація швидких рухів і проведення деформацій стають практично неможливими.

Для вирішення цієї проблеми придумано кілька прийомів. Більшість з них ґрунтується на тому, що поверхня персонажа згладжується після створення анімації, але до візуалізації.

Деякі об'єктно-орієнтовані пакети, дозволяють згладжувати поверхню в будь-який момент, але зазвичай дизайнери роблять це після деформування моделі і створення анімації. Багато пакетів дають можливість управляти модифікатором згладжування. Якщо модифікатор вимкнений, модель персонажа має низький дозвіл, і анімація проводяться швидко. При включеному модифікаторі дозвіл моделі стає вищим, і її можна візуалізувати. Оскільки каркас ущільняється вже після деформації моделі, розтягнень і складок при згладжуванні не виникає.

Щоб анімація проходила в правильному темпі, використовуються моделі з низьким дозволом. Для отримання моделі, що має реалістичний вигляд, після деформування і перед візуалізацією проводиться згладжування.

Ще один спосіб вирішення даної проблеми - одночасно працювати з двома моделями, що мають низьку якість і високу роздільну здатність. Перша з них піддається анімації.

Потім криві анімації переносяться на другу модель, і виконується візуалізація. Єдина проблема полягає в тому, що деформації моделі з низьким дозволом трохи відрізняються від деформацій згладженої моделі, і в результаті можуть виникнути розтягування і інші дрібні неприємності.

Висновки до розділу 2

Аналіз можливих алгоритмів для 3D моделювання показав, шо полігональне моделювання є більш підходящим для поставлених цілей. В другому розділі також детально розглянутий цей алгоритм та його особливості. Зважаючи, на даний аналіз можна вважати, що розробка програмного забезпечення для моделювання на основі полігонального моделювання є доцільною та немає вагомих аргументів щодо неможливості створення такого програмного забезпечення на базі цих технологій.

РОЗДІЛ 3

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Загальний огляд проекту

Для реалізації системи було вирішено використовувати мову програмування C++, оскільки для вирішення поставлених задач вона є однією з найпопулярніших, так як існує достатня кількість бібліотек та фрейм ворків для взаємодії як напряму з графічним процесором, так і через додаткові модулі.

Як показано у Додатку А, умовно проект можна поділити на три частини:

I. Розробка інструментів для зчитування, запису та збереження 3D моделей.
II. Розробка модулів для відображення та редагування вже завантажених моделей.
III. Розробка інтуїтивно зрозумілого інтерфейсу.

Розробка системи 3D моделювання виконувалася саме в такому порядку. На першому та другому етапі використовувався текстовий редактор Sublime Text та система збірки Makefile. На третьому етапі виконувалася розробка графічного інтерфейсу за допомогою QtCreator та для збірки використовувався Qmake. Також проект знаходиться у системі контролю версій Git. В систему моделювання вбудована робота з вводом та виводом даних, логування, візуалізація та взаємодія з користувачем.

Архітектура проекту представлена у Додатку Б. Кожен модуль є незалежним від модулів інших рівней та кожен виконує свою чітко поставлену задачу. Модулі, що реалізують усю логіку проекту написані на C++, а модулі, що реалізують інтерфейс користувача використовують

бібліотеку Qt. Qt багатоплатформовий фреймворк для розробки програмного забезпечення на мові програмування C++. Є також «прив'язки» до багатьох інших мов програмування: Python - PyQt, PySide; Ruby - QtRuby; Java - Qt Jambi; PHP - PHP-Qt і інші [6].

Для візуалізації використовується OpenGL. OpenGL - це найбільш широко розповсюджений 2D та 3D графічний API в індустрії, що використовується тисячами додатків для найрізноманітніших комп'ютерних платформ.

OpenGL дозволяє розробникам програмного забезпечення для ПК, робочих станцій та суперкомп'ютерних апаратних засобів створювати високопродуктивні, візуально переконливі графічні програми на таких ринках, як CAD, створення контенту, енергія, розваги, розробка ігор, виробництво, медична та віртуальна реальність. OpenGL розкриває всі функції найновішого графічного обладнання [7].

3.2 Початок виконання програми

На першому етапі необхідно зчитати вхідний файл. В даному випадку цей файл має розширення *.off. Формат, заснований на ASCII, використовується для опису 3D-об'єктів. Визначає у плоскості набори полігонів і вершин, утворюючих поверхню об'єкта. Кожен файл *.off починається з ключового слова "OFF". На наступній строчці міститься кількість вершин, граней та кутів моделі. Вершини і грані відображаються з вимірюваннями x, y, z, по кожній на строку. Далі наведений приклад запису куба у такому файлі.

```
OFF
8 12 12
-0.6 0.6 0.6
-0.6 -0.6 0.6
```

0.6 -0.6 0.6

0.6 0.6 0.6

-0.6 0.6 -0.6

-0.6 -0.6 -0.6

0.6 -0.6 -0.6

0.6 0.6 -0.6

3 0 1 2

3 0 2 3

3 1 5 2

3 5 6 2

3 2 6 7

3 2 7 3

3 0 4 5

3 0 5 1

3 5 4 7

3 5 7 6

3 0 3 7

3 0 7 4

3.3 Опис класу Mesh

Отже, для реалізації збереження координат кожної 3D моделі використовується клас Mesh. Також цей клас використовується для зміни параметрів моделі, а також вигляду моделі при редагуванні.

Задля зручності обчислень та візуалізації був створений клас Vec3, в якому описуються всі потрібні операції з вершинами, такі як перетворення з полярної системи координат в декартову та навпаки, проекція на необхідну вісь, отримання ортогональних проекцій і подібні.

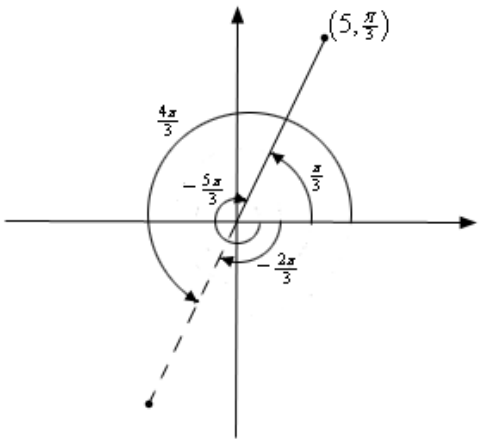


Рис. 3.1 – Полярна система координат

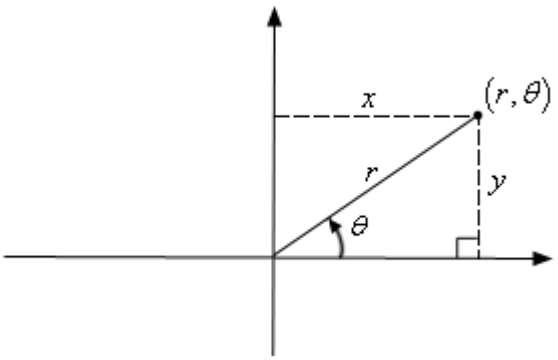


Рис. 3.2 – Перехід з полярних до декартових

Як показано на рис. 3.2 задля переходу з полярної системи у декартову за допомогою прямокутного трикутника необхідно обчислити координати за такими формулами:

$$x = r * cos\theta$$
$$y = r * sin\theta \quad (3.1)$$

Перехід до полярної системи можна зробити за наступними формулами.

$$r = \sqrt{(x^2 + y^2)}$$
$$\theta = tan^{-1}(y/x) \quad (3.2)$$

Клас Mesh використовує всі ці методи для потрібних цьому операцій. Наприклад, для створення трикутників між вершинами цей клас виконує пошук усіх комбінацій за допомогою Depth-first search (пошука у глибину). Це рекурсивний алгоритм, який використовує ідею зворотного відстеження. Він почитає вичерпний пошук усіх вузлів ідучи вперед, якщо це можливо, якщо ні, то шляхом зворотного відстеження. Усі вузли будуть відвідуватися на поточному шляху до тих пір, поки не пройдуть усі невідомі вузли, після чого буде обраний наступний шлях. Основна ідея реалізації така:

1. Обрати початковий вузол і додати всі його суміжні вузли в стек.

2. Обрати вузол зі стека, щоб вибрати наступний вузол, який потрібно відвідати, і додати всі його суміжні вузли в стек.

3. Повторити цей процес, поки стек не порожній. Однак треба переконатися, що вузли, які відвідуються, позначені. Це дозволить не відвідувати той же вузол. Якщо не позначити відвідані вузли і відвідати той самий вузол не один раз, можна опинитися в нескінченному циклі.

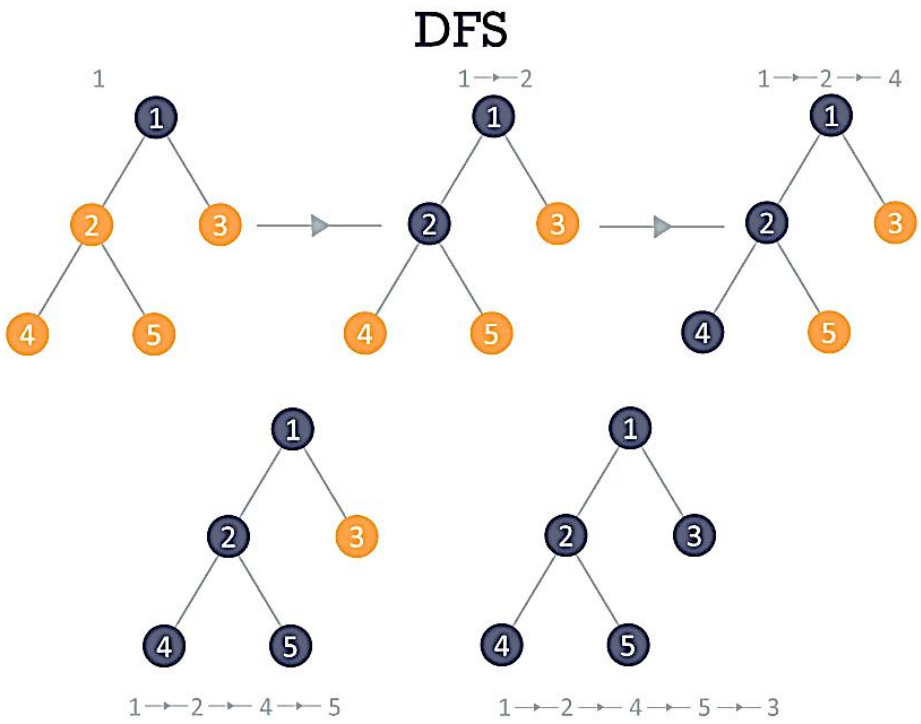


Рис. 3.3 – Алгоритм пошуку у глибину

Також клас Mesh містить в собі метод subdivision(), який в свою чергу реалізує операцію розбиття на підрозділи.

Розбиття на підрозділи - це особливий випадок уточнення, який є ключовим для успіху деяких найбільш широко використовуваних типів параметричних патчів та їх сукупних поверхонь. Поверхня може бути «доопрацьована», коли існує такий алгоритм, що можна ввести більше контрольних точок, зберігаючи форму поверхні точно такою ж. Для інтерактивних та дизайнерських цілей це дозволяє дизайнеру ввести більшу роздільну здатність для більш тонкого контролю, не вводячи небажаних побічних ефектів у форму. Для більш аналітичних цілей він дозволяє

розбивати поверхню на частини, часто адаптивно, при цьому вірний оригінальній формі.

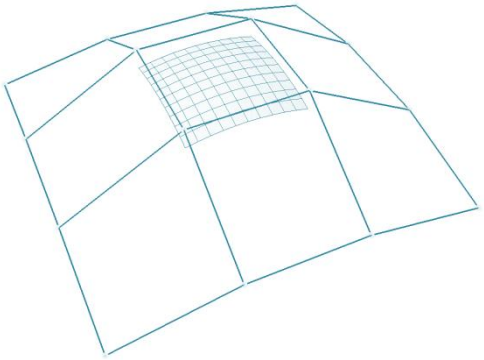


Рис. 3.4 – Поверхня

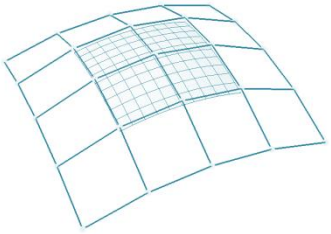


Рис. 3.5 – Поділена поверхня

У випадках, проілюстрованих вище, рівномірно поділені клітинки дають таку ж граничну поверхню, що і оригінал.

Гранична поверхня залишається однаковою для більшої кількості контрольних точок (приблизно 4 рази з кожною ітерацією підрозділу), і ці точки знаходяться ближче до (але не на) поверхні. Може здаватися, що ці нові контрольні точки можуть вигідно використовуватися для зображення поверхні, але використовувати однакову кількість точок, оцінених у відповідних рівномірно розташованих параметричних місцях на поверхні, як правило, простіше і ефективніше.

Треба зауважити також, що точки клітки зазвичай не мають жодних нормальних векторів, пов'язаних з ними, хоча можна чітко прораховувати нормалі для довільних розташувань на поверхні так само, як це робиться для їх положення. Отже, якщо відображати клітку як затінену поверхню, нормальні вектори в кожній з контрольних точок повинні бути налаштовані. Отже, і положення, і нормалі точок на тоншій клітці є обома наближеннями.

Можна зробити висновок, що розбиття на підрозділи дозволить уточнити сітку довільної топології, але одержані точки не будуть лежати на

граничній поверхні, і будь-які нормальні вектори, що витікають з цих точок і пов'язані з цими точками, будуть лише наближеннями до граничної поверхні.

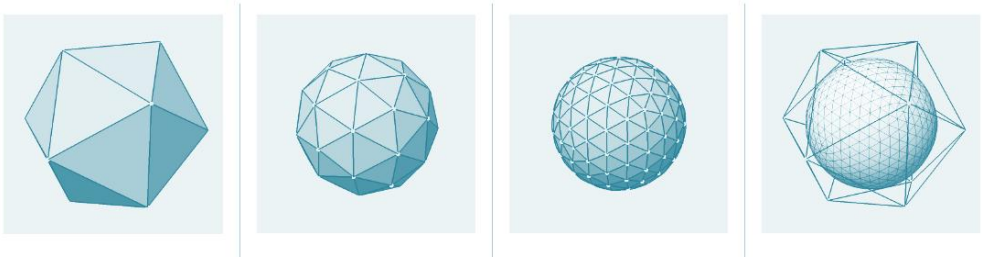


Рис. 3.6 – Операція розбиття на підрозділи

Клас Mesh містить в собі метод torrance(), що реалізує операцію освітлення Кука-Торренса. Модель освітлення Кука-Торренса використовується для розрахунку відбитого світла, тож розсіяне світло треба обчислювати за класичною формулою Ламберта, в якій освітленість точки залежить тільки від кута між нормаллю до поверхні в даній точки, і положенням джерела світла [8]. Обчислюється як скалярний добуток нормалі і нормалізованого положення джерела світла:

$$K_d = N \bullet L \qquad (3.3)$$

Кількість відбитого світла залежить від трьох чинників:

1. Коефіцієнт Френеля (F).
2. Геометрична складова, що враховує самозатінення (G).
3. Компонент, що враховує шорсткість поверхні (D).

Загальна формула для обчислення відбитого світла така:

$$K = \frac{F \cdot G \cdot D}{(\vec{V} \bullet \vec{N}) \cdot (\vec{L} \bullet \vec{N})} \qquad (3.4)$$

Обчислення геометричної складової:

$$G = \min\left(1, \frac{2(\vec{H} \bullet \vec{N})(\vec{V} \bullet \vec{N})}{(\vec{V} \bullet \vec{H})}, \frac{2(\vec{H} \bullet \vec{N})(\vec{L} \bullet \vec{N})}{(\vec{V} \bullet \vec{H})}\right) \quad (3.5)$$

де N - нормаль в точці, V - вектор погляду, L - положення джерела світла, H - нормалізована сума векторів L і V. Всі вектори повинні бути нормалізовані.

Компонент, що враховує шорсткість поверхні - це розподіл мікрограней поверхні, для більш точного обліку відбитого від них світла. Зазвичай, для обчислення цього компонента використовують розподіл Бекмана:

$$D = \frac{1}{4m^2(\vec{H} \bullet \vec{N})^4} \cdot e^{\left(\frac{(\vec{H} \bullet \vec{N})^2 - 1}{m^2(\vec{H} \bullet \vec{N})^2}\right)} \quad (3.6)$$

де параметр m (від 0 до 1) визначає шорсткість поверхні. Чим він більший, тим поверхня шорстка, отже, відображає світло навіть під широкими кутами.

Коефіцієнт Френеля.

Для обчислення коефіцієнта Френеля існує багато формул, але в даному випадку доцільніше застосовувати апроксимацію Шліка:

$$F = F_0 + \left(1 - (\vec{V} \bullet \vec{N})\right)^5 \cdot (1 - F_0) \quad (3.7)$$

де F0 - кількість відбитого світла при нормальному падінні (перпендикулярно поверхні).

Для того чтобы не вводить дополнительный параметр, и не усложнять процесс вычислением степени, можно использовать другую формулу:

$$F = \frac{1}{1 + (\vec{V} \bullet \vec{N})} \quad (3.8)$$

З поправкою на те, що метали добре відображають - вона дає так само непоганий результат.

Таким чином, можна підставити її в вихідну формулу і отримати очікуваний результат.

Для зручності роботи з фреймворком OpenGL було створено класи GLShader, GLProgram, GLError.

3.4 Опис класу GLShader

Щоб краще розуміти для чого взагалі потрібен цей клас, треба розглянути в цілому що таке шейдер та їх види.

Шейдер (англ. Shader «затіняючий») - комп'ютерна програма, передбачена для виконання сценаріїв процесора відеокарти (GPU).

В залежності від стадії графічного конвеєра, шейдери діляться на кілька типів: вершинний, фрагментний (піксельний) та геометричний.

Вершинними шейдерами роблять анімації персонажів, трави, дерев, створюють хвилі на воді та інше. У вершинних шейдерах програмісту доступні дані, пов'язані з вершинами, наприклад: координати вершини у просторі, текстурні координати, кольори та вектор нормалі.

Геометричні шейдери здатні створити нову геометричну форму та можуть використовуватися для створення частин, зміни деталей моделей, створених силуетами тощо. На відміну від вершинного шейдера, він здатен обробляти не тільки одну вершину, а й цілий примітив. Примітивом може бути відрізок (дві вершини) і трикутник (три вершини), за наявності інформації про суміжні вершини для трикутників примітива може бути оброблено до шести вершин [9].

Піксельні шейдери виконують текст текстури, освітлення та різноманітні текстурні ефекти, такі як оновлення, переклад, туман, Bump Mapping та пр. Піксельні шейдери також використовуються для пост-ефектів.

Піксельний шейдер працює з фрагментами растрового зображення та з текстурами - обробляє дані, пов'язані з пікселями (наприклад, колір, глибина,

текстурні координати). Піксельний шейдер використовується на останній стадії графічного конвеєра для формування фрагментів зображення.

Клас GLShader описує саме векторний шейдер, він встановлює вихідний код у шейдері. Будь-який вихідний код, який раніше зберігався в об'єкті шейдера, повністю замінюється. Далі клас компілює рядки вихідного коду, визначеному шейдером за допомогою OpenGL. Статус компіляції буде зберігатися як частина стану об'єкта шейдера. Якщо шейдер був скомпільований без помилок то він вважається готовим до використання. Компіляція шейдера може вийти з ладу з кількох причин. Незалежно від того, вдала компіляція чи ні, інформацію про компіляцію можна отримати зі стану об'єкта шейдера.

Для взаємодії з параметрами класу можна використовувати наступні методи:

id() – отримати id моделі;

name() – отримати ім’я моделі;

type() – отримати тип моделі;

source() – отримати змінну _source, котра зберігає шлях до джерела шейдера бібліотеки OpenGL;

setSource(const std::string & source) - задати змінну _source, щлях до джерела шейдера бібліотеки OpenGL;

3.5 Опис класу GLProgram

Спочатку клас створює полотно, в якому буде відображатися модель. Далі після ініціалізації моделі, GLProgram малює модель за допомогою вже створених даних класом Mesh. Отже, можна вважати, що цей клас безпосередньо взаємодіє з OpenGL.

3.6 Опис класу GLError

Цей клас міститься в обох класах, і в GLProgram, і в GLShader. Він створений для того, щоб при кожній взаємодії з OpenGL можна було відстежити, що всі операції виконані без помилок, а якщо все ж помилка сталася, то GLError виводить у лог повідомлення де саме це відбулося.

3.7 Опис класу Camera

Клас Camera реалізує методи для коректного відображення моделі, її повертання, переміщення у вікні та віддалення від неї. Основні принципи роботи цього класу пов’язані з кватерніоном.

Існує декілька шляхів, щоб реалізувати обертання об'єктів. Багато програмістів використовують для цього матриці обертання або кути Ейлера. Кожне з цих рішень добре працює до тих пір, поки не потрібно здійснити гладку інтерполяцію між двома різними положеннями об'єктів. Наприклад, об'єкт, який просто вільно обертається у просторі. Якщо зберігати обертання, як матрицю, чи у виді кутів Ейлера, то гладка інтерполяція виявиться досить об’ємною по обчисленням та буде не такою гладкою, як при інтерполяції кватерніонами.

Кватерніони були введені Гамільтоном у XVIII столітті. Кватерніони використовують 4-х вимірне розширення множини комплексних чисел, іншими словами - це гіперкомплексні числа [10]. Отже, кватерніон q задається четвіркою чисел *(x, y, z, w):*

$$w + xi + yj + zk, \text{ де } i^2 = j^2 = k^2 = -1 \qquad (3.9)$$

Його також можна записати у вигляді [w, v], де w називають скаляр, а $v = (x, y, z)$ — вектор.

У кватерніоні параметри одиничного вектора множаться на синус половини кута оберту. Четвертий компонент - косинус половини кута оберту. Далі представлений псевдокод запису кватерніона:

```
// Кут оберту RotationAngle задан у радіанах
x = RotationAxis.x * sin(RotationAngle / 2)
y = RotationAxis.y * sin(RotationAngle / 2)
z = RotationAxis.z * sin(RotationAngle / 2)
w = cos(RotationAngle / 2)
```

Таблиця 3.1

Таблиця з прикладами значень кватерніонів

0	X	Y	Z	Обертання
1	0	0	0	-
0	1	0	0	180˚ навколо осі x
√0.5	√0.5	0	0	90˚ навколо осі x
√0.5	- √0.5	0	0	- 90˚ навколо осі x

Для взаємодії з параметрами класу можна використовувати наступні методи:

getFovAngle() - отримати кут огляду камери;

setFovAngle(float a) - встановити кут огляду камери;

getAspectRatio() – встановити співвідношення сторін;

getScreenWidth() – отримати ширину екрана;

getScreenHeight() - отримати довжину екрана;

3.8 Опис класу Vec4

Кватерніон зручно розглядати як 4d вектор, і деякі операції з ним виконуються як над векторами. Задля цього, окрім класу Vec3, був створений клас Vec4. Нижче наведені приклади операцій додавання, віднімання та множення таких об’єктів.

$$q_1 + q_2 = [x_1 + x_2, y_1 + y_2, z_1 + z_2, w_1 + w_2]$$
$$q_1 - q_2 = [x_1 - x_2, y_1 - y_2, z_1 - z_2, w_1 - w_2]$$
$$q_1 s = [x_1 s, y_1 s, z_1 s, w_1 s] \quad (3.10)$$

У роботі з кватерніонами частіше за все доводиться їх множити один на одного. Це дуже корисна операція, оскільки при множенні одного кватерніона на інший, отримується перше обертання, обернене на друге. Важно пам’ятати, що множення кватерніонів некомутативне, це означає, що порядок операндів важливий. Тобто q * q' це не те ж саме, що і q' * q. Сенс операції додавання можна описати як «суміш» обертань, тобто виходить обернення, що знаходиться між q і q'.

Множення на скаляр на обертання не впливає. Кватерніон, помножений на скаляр, представляє те ж саме обертання, крім випадку множення на 0. При множенні на 0 можна отримати «невизначене» обертання.

Псевдокод додавання 2-х кватерніонів:

```
// представлення через кут осі
quaternion1 = [0, 0, 1, a];
quaternion2 = [1, 0, 0, b];
// Кватерніони
quaternion1 = [0, 0, sin(a/2), cos(a/2)];
quaternion2 = [sin(b/2), 0, 0, cos(b/2)];
// Сума кватерніонів
quaternion1 + quaternion2 = [sin(b/2), 0, sin(a/2), cos(a/2) + cos(b/2)]
```

Також слід розрізняти операції норма та модуль:

Норма(norm):

$$N(q) = x^2 + y^2 + z^2 + w^2 \quad (3.11)$$

Модуль(magnitude) або довжина кватерніона:

$$|q| = \sqrt{N(q)} = \sqrt{(x^2 + y^2 + z^2 + w^2)} \quad (3.12)$$

Через модуль кватерніона можна нормалізувати. Нормалізація кватерніона - це приведення до довжини 1 (так як і у векторах):

```
length = pow( pow(v.x, 2)+ pow(v.y, 2) + pow(v.z, 2), 0.5 )
normal.x = v.x / length
normal.y = v.y / length
normal.z = v.z / length
```

Для взаємодії з параметрами класу можна використовувати наступні методи:

init(T x, T y, T z, T w) – ініціалізація параметрів x, y, z, w вектора(квартеніона);

squaredLength() – отримати норму квартеніона;

length () – отримати довжину квартеніона;

projectOn (const Vec4 & N, const Vec4 & P) – отримати проекцію квартеніона;

3.9 Опис класу Ray

Цей клас реалізує алгоритм кидання променів і відповідає за підсвітку моделі під час редагування.

Кидання променів(Ray Tracing) - це технологія рендеринга (тобто відтворення, створення) тривимірної графіки, де використовується цей принцип. Спеціальний алгоритм відстежує шлях променя від об'єкта освітлення.

Ray Tracing дозволяє створювати неймовірно реалістичне освітлення, котре практично не відрізняється від реального [11].

Наприклад, є камера (для спрощення камера це матеріальна точка). Також у є площина проектування, яка є набором пікселів. Від камери, до кожного пікселя площині проектування проводяться вектори (первинні промені) і для кожного променя знаходиться найближчий об'єкт сцени, який він перетинає. Кольором, що відповідає точці перетину, можна зафарбувати відповідний піксель на площині проектування. Повторивши цю процедуру

для всіх точок площини проектування - отримується зображення тривимірної сцени.

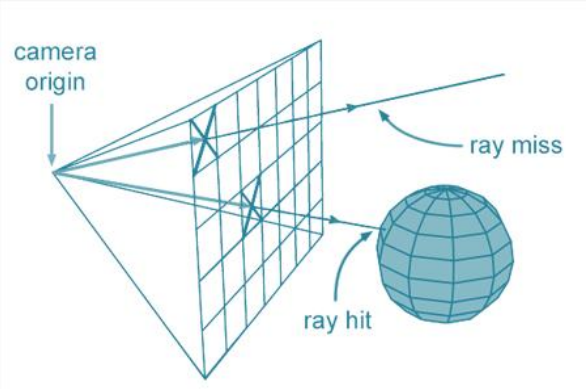


Рис. 3.7 – Операція кидання променів

Колір кожного пікселя можна розрахувати незалежно від інших (тому що промені, що випускаються з камери ніяк не впливають один на одного - їх можна обробляти паралельно). Алгоритм не містить обмежень на геометрію об'єктів (можна використовувати багатий набір примітивів: площини, сфери, циліндри тощо).

3.10 Опис класу BVH

Клас BVH розроблений для роботи з класом Ray, оскільки необхідно створити об’єкт, з яким будуть взаємодіяти промені світла. Bounding Volume Hierarchy (BVH) - Ієрархія обмежуючих об’ємів. Історично BVH дерева використовуються для розрахунку зіткнень. Але останнім часом BVH активно намагаються задіяти у рейтрейсингу в зв'язку з тим, що в анімованих сценах BVH можна швидше перебудовувати, і як правило можна перебудовувати не все дерево.

Виділяють 5 типів BVH дерев:

1. Sphere tree
2. AABB tree (Axis Aligned Bounding Box)

3. OBB tree (Oriented Bounding Box)
4. k-DOP (Discrete Oriented Polytope)
5. SSV (Swept Sphere Volume)

В класі BVH використовується саме AABB дерево. AABB дерево - бінарне дерево, утворене ієрархією описуючих "боксів", що лежать уздовж осі координат. Кожен AABB охоплює всіх своїх дітей. Тобто кожен вузол дерева являє собою AABB і два посилання на дітей: "позитивного" і "негативного". Тут терміни "позитивний" і "негативний" не означають, що один кращий за інший, просто так склалося в результаті дотримання правил побудови дерева. По кожній координаті, "дитина" має менший або рівний своєму батькові розмір і може перетинатися зі своїм "братом". Листям дерева є трикутники.

Для взаємодії з параметрами класу можна використовувати наступні методи:

getLeftChild() – отримати піддерево, що знаходиться ліворуч;

getRightChild() – отримати піддерево, що знаходиться праворуч;

isALeaf() - отримати змінну isLeaf, що визначає чи є це дерево листом;

getTriangle() – отримати трикутник;

Розглянемо блок-схему алгоритму представлену у Додатку В. AABBs дерево будується з полігональної моделі, представленої непустим списком трикутників, в такий спосіб:

1. Якщо множина трикутників T складається тільки з одного трикутника, переходимо до пункту 7.
1. Створюємо новий вузол U.
2. Для множини трикутників T обчислюємо AABB, який запам'ятовуємо в вузлі U дерева.
3. Вибираємо межу розділення - площину, що проходить через середину найбільшої сторони AABB паралельно двом найменшим граням.

4. Множину T розбиваємо на дві непусті підмножини Tpositive і Tnegative по положенню центрів проекцій трикутників на нормаль площини, що розділяє.
5. Рекурсивно повторюємо послідовність дій для підмножин Tpositive і Tnegative, починаючи з пункту 1.
6. Трикутник призначаємо листом дерева, припиняємо рекурсію.

Паралельність AABB вісям координат є як основною перевагою цього методу, так і його основним недоліком. Можна дуже добре заощадити на витраті пам'яті, спростити і прискорити методи перевірки перетину вузла дерева з деякими геометричними примітивами. З іншого боку AABB програє Oriented bounding boxes (OBBs) tree по точності охоплення набору трикутників. Як наслідок, зростає кількість ітерацій необхідних для перевірки перетину з деревом. AABB займає мало пам'яті за рахунок того, що не треба зберігати орієнтацію "боксу". Більш того, нормовані всі величини за розміром кореневого "боксу", можна перейти від змінних з плаваючою точкою до змінних з фіксованою точкою. Швидкість роботи підвищується за рахунок того, що методи перевірки перетину геометричних примітивів з AABB простіше методів для OBB. Також дерева можна "злегка підрізати", припиняючи рекурсію побудови дерева при досягненні певного (досить малого) розміру AABB. Для підвищення швидкості можна також кешувати вузли, з якими в минулий раз відбувся перетин, що дозволяє в деяких ситуаціях отримати помітний виграш. Ще одним незаперечним плюсом AABBs дерева є можливість його застосування для деформованих об'єктів.

Висновки до розділу 3

Створена архітектура дозволяє з легкістю додавати новий функціонал, змінювати вже створений або ж видаляти непотрібні функції.

Дослідження різних алгоритмів дозволило реалізувати достатньо зрозумілу систему з точки зору проектування. Програма написана на мові C ++, з використанням бібліотеки OpenGL для візуалізації та бібліотеки Qt для взаємодії з користувачем. Для реалізованої програми використовується система автоматизації збірки make. Вхідними даними для програми є файл * .off, результатом роботи програми є модифікована * .off. Вихідні дані є коректними. Програма була протестована на тестових завданнях, які показали її коректну роботу.

РОЗДІЛ 4

ДЕМОНСТРАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Початковий екран програми

Інтерфейс користувача повинен бути інтуїтивно зрозумілим, тож це стало однією з найголовніших задач при розробці дизайну програми. Великий простір форми займає Label — пояснювальна напис з назвою програми(«3D Modeling System»). Трохи нижче одразу після назви розміщений компонент Button — кнопка, з підписом «Choose model», що дозволяє користувачу одразу зрозуміти, як розпочати роботу з програмою.

Рис. 4.1 – Початковий екран програми

Отже, завантажимо модель для початку редагування. Для демонстрації роботи була обрана модель голови людини, оскільки вона є досить складною і гарно домонструє зміни при редагуванні.

4.2 Головний екран програми

У формі головного екрану(Рис. 4.2) реалізовані кнопки з основними операціями. Вони умовно поділені на три групи.

Група, що розташовується праворуч, являє собою групу операцій, що змінюють модель за принципом роботи окремого алгоритму, вони можуть буду ввімкнені, чи вимкнуті, в залежності від необхідного результату.

Група, що розташовується ліворуч, має схожий принцип роботи, але до того ж можна задати необхідні параметри. Всі вони зберігаються у змінних типу float, окрім перших двох. Більше того, операцію BVH було вирішено додати для того, щоб продемонструвати роботу побудови дерева. Ця група працює за таким принципом: якщо необхідно внести зміни для заданого коефіцієнта, потрібно просто натиснути на поле з вводом(за замовчуванням там вже введені параметри).

Остання(третя) група кнопок розташована в нижній частині екрана і являє собою так звану панель керування швидкого доступу. Як можна побачити на рис. 4.2, в ній знаходяться кнопки, що дозволяють зберігати відредаговану модель, завантажити нову модель, перезавантажити поточну модель або взагалі закрити поточну модель та повернутися на початковий екран програми.

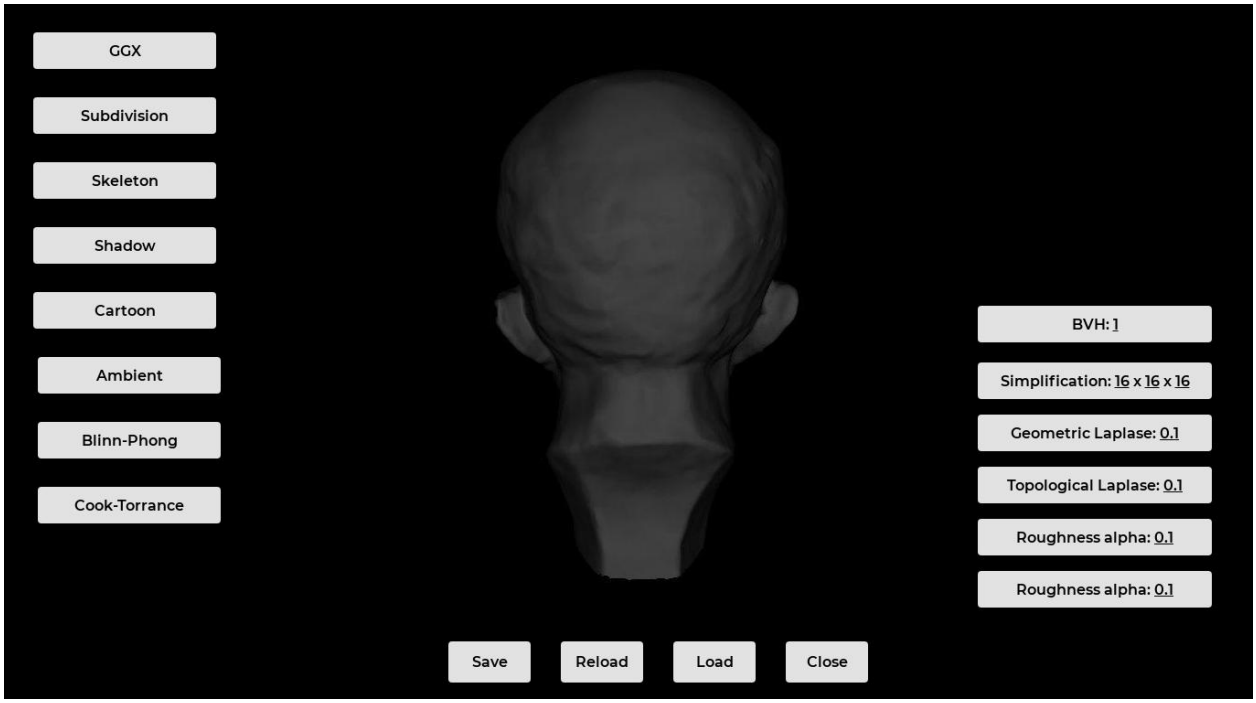


Рис. 4.2 – Головний екран програми

На цьому етапі можна розпочинати редагування. Обрана модель завантажилася у систему оберненою, отже для зручності необхідно її повернути. Це можна зробити двома способами, оскільки за переміщення та обернення моделі відповідає миша, а також hot keys(гарячі клавіши).

Для того, щоб повернути модель за допомогою миші потрібно утримувати нажатою ліву кнопку миші та вести курсор у необхідну сторону, для переміщення — утримувати праву кнопку миші та вести курсор у сторону, в яку потрібно перемістити модель. Щоб збільшити чи зменшити розмір моделі потрібно прокрутити колесо миші.

Для повертання та переміщення моделі за допомогою гарячих клавіш необхідно натиснути клавіши зі стрілками для переміщення та клавіши w, a, s, d для обертання(w та s по осі Y, a та d по осі X).

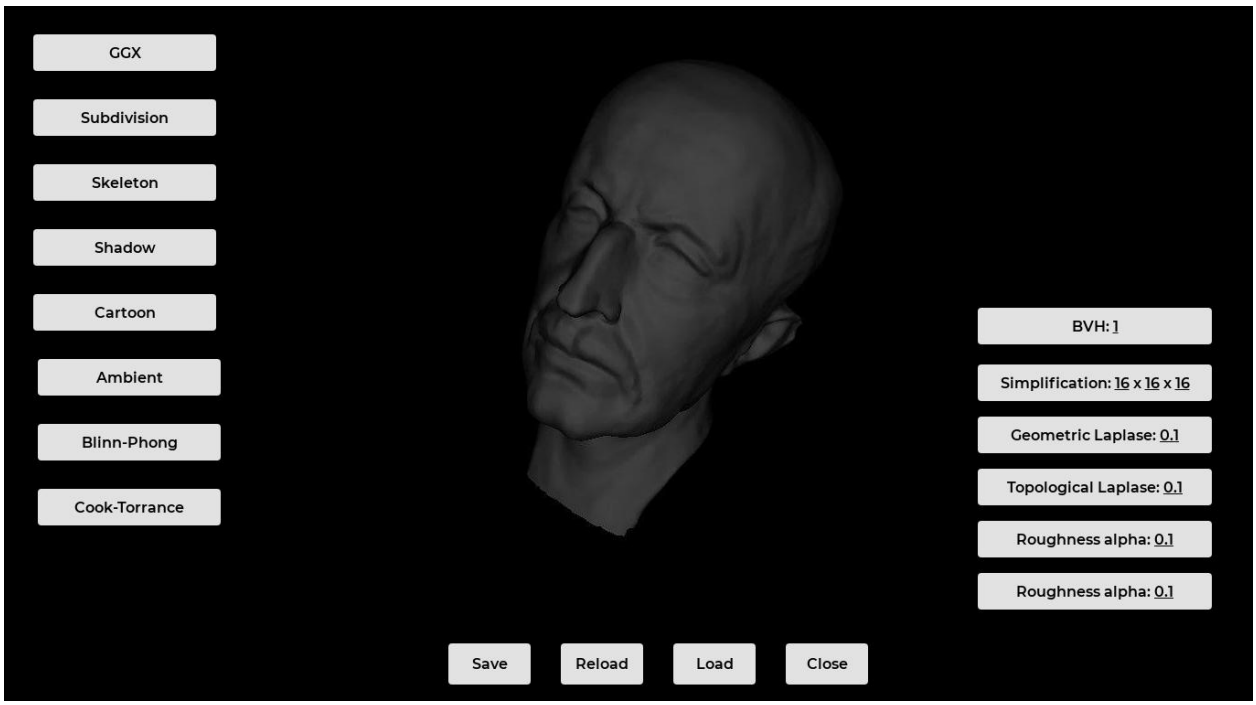


Рис. 4.3 – Обертання

Тепер, коли модель розташована та повернута зручно, можна спробувати застосувати деякі алгоритми.

На рис. 4.4 зображена та ж сама модель, для якої застосований вигляд скелету. Такий вигляд зазвичай використовується для того, щоб краще переглянути побудову моделі. Для цього необхідно натиснути на кнопку з назвою «Skeleton».

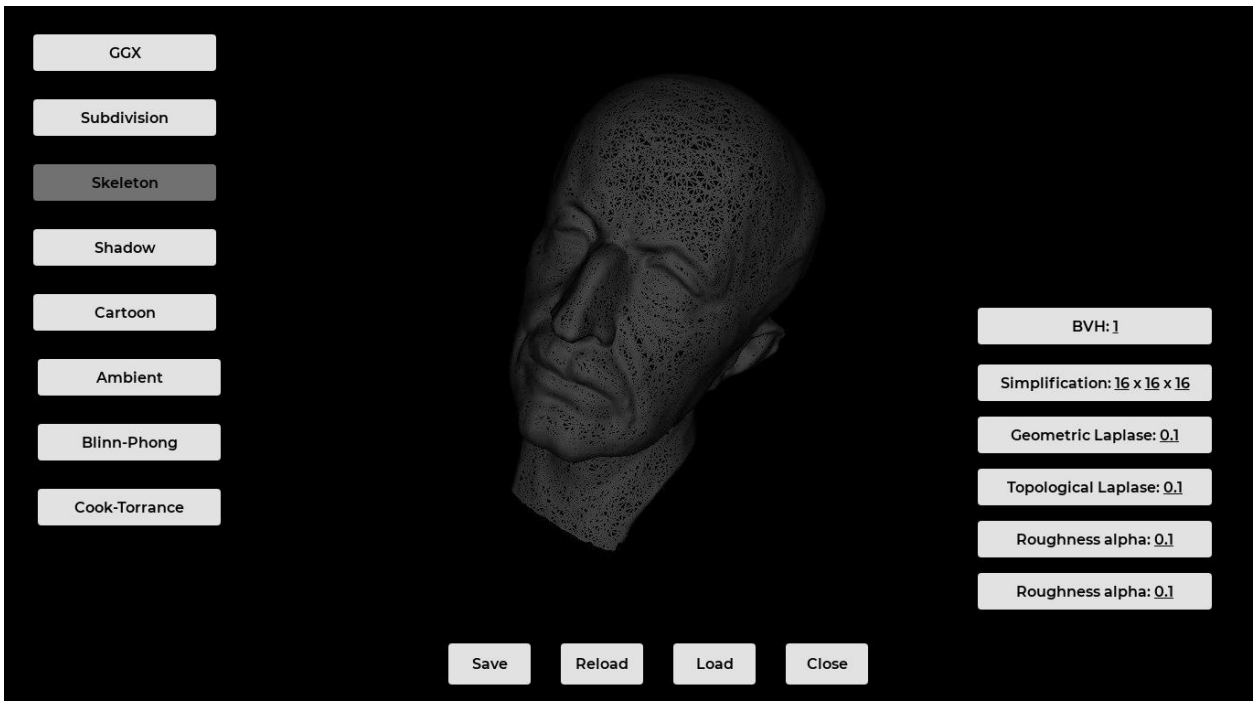


Рис. 4.4 – Скелет моделі

На рис. 4.4 зображено збільшений вигляд скелету, на якому дуже чітко видно, що 3D модель складається з безлічі точок, які з’єднані між собою трикутниками.

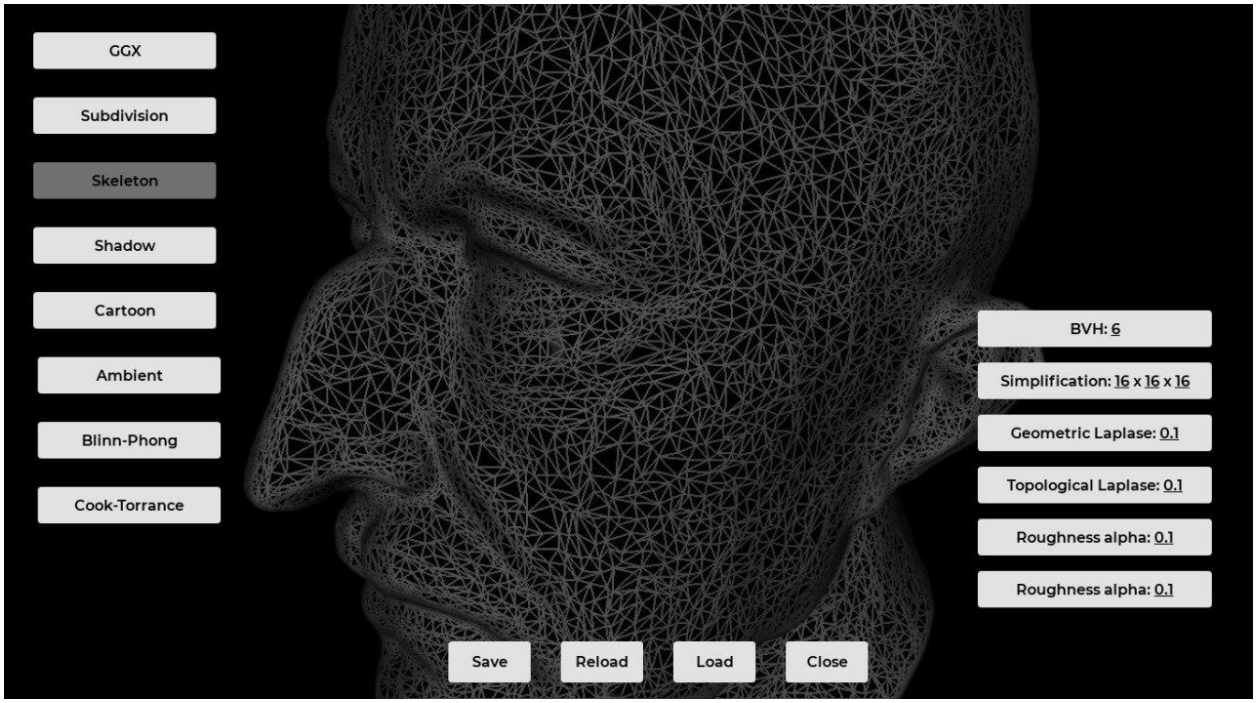


Рис. 4.5 – Збільшений скелет

Також, треба зауважити, що при ввімкненому режимі скелета кнопка “Skeleton” змінила колір, для зручності, щоб користувач одразу бачив , які режими ввімкнені.

Для того, щоб повернутися до звичайного представлення моделі необхідно ще раз натиснути кнопку “Skeleton” і режим скелету вимкнеться.

Рис. 4.6 – Модель освітлення Кука-Торренса

Після вимкнення відображення скелету моделі, відповідна кнопка перестала підсвічуватися, а після натиснення на кнопку «Cook-Torrance» ввімкнулася Модель освітлення Кука-Торренса, результат якої зображено на рис. 4.6.

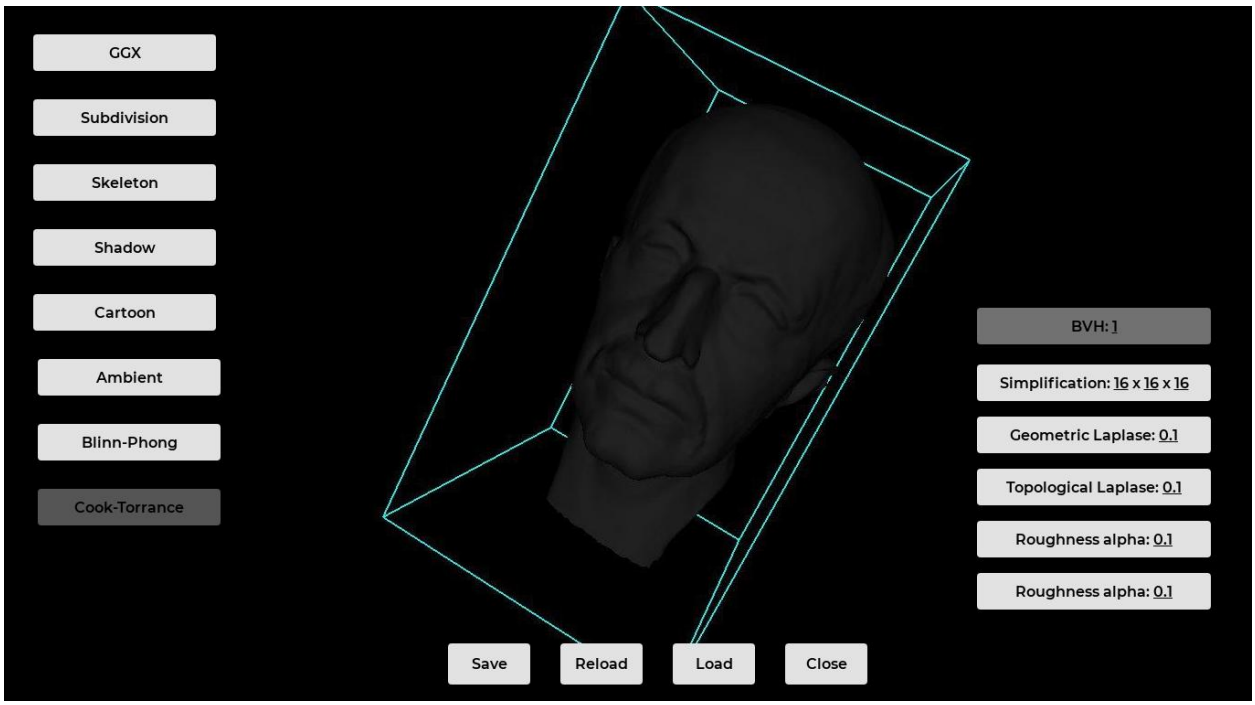


Рис. 4.7 – Побудова BVH з кількістю гілок 1

Щоб ввімкнути функцію відображення дерева BVH, необхідно натиснути на відповідну кнопку справа. На рис. 4.7 зображено отриманий результат, а саме побудова дерева з 1 гілкою(за замовчуванням) та одночасне виконання функцій з правої та лівої груп.

Для зміни кількості гілок у дереві необхідно натиснути на поле для вводу даних кнопки «BVH» та ввести потрібне число, наприклад 6. Далі для відображення результату потрібно підтвердити операцію, натиснувши на довільній зоні цієї кнопки. Отриманий результат зображений на рис. 4.8.

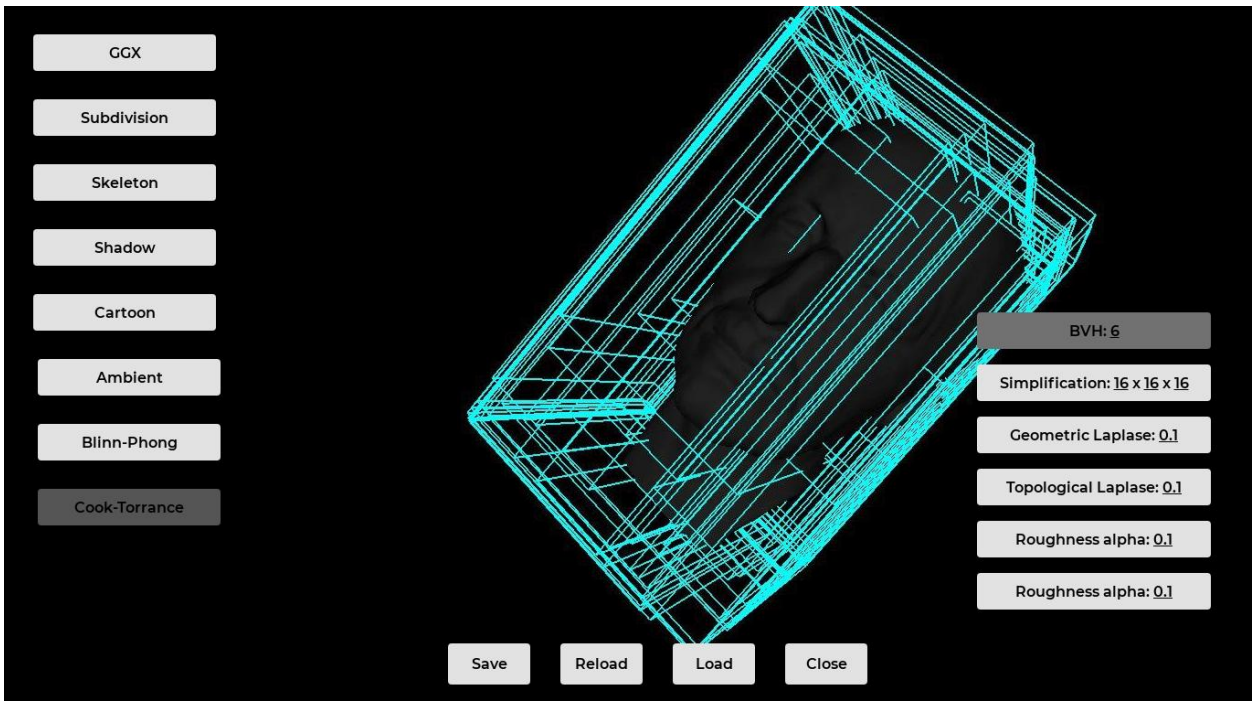


Рис. 4.8 – Побудова BVH з кількістю гілок 6

Тепер вимкнемо відображення BVH, натиснувши на довільній зоні кнопки «BVH», та ввімкнемо кнопку «Topological Laplase» з вже введеним параметром 0.1, що виконує зглажування моделі. Результат показаний на рис. 4.9. Потім треба натиснути кнопку «Save», щоб зберегти модель.

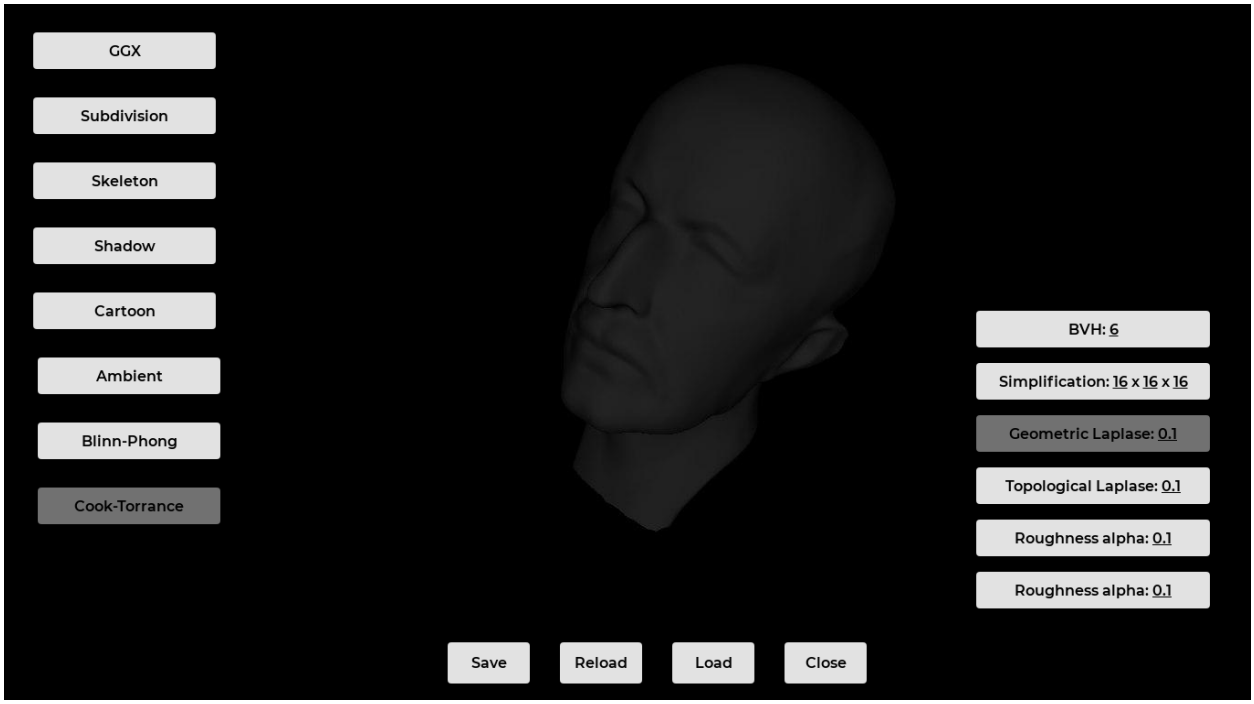


Рис. 4.9 – Topological Laplase

Висновки до розділу 4

У даному розділі представлена робота програми на прикладі редагування конкретної моделі. Був детально описаний алгоритм роботи програми та наведені відповідні знімки екрана.

Створена система моделювання дозволяє без зайвих зусиль редагувати моделі. Користувачеві не знадобиться ніякого досвіду у графіці або досвіду роботи с системами редагування, адже створений графічний інтерфейс є досить зручним та зрозумілим.

ВИСНОВКИ

У даному дипломному проекті було поставлено завдання створити систему 3D моделювання розроблену для операційної системи на ядрі Linux з зрозумілим та легким у використанні графічним інтерфейсом. Для отримання кращого результату були проаналізовані найпопулярніші рішення, а також описані їх полюси та мінуси. Також був проведений аналіз різних алгоритмів моделювання який показав, що найкращим рішенням для поставленої задачі буде використовувати полігональне моделювання.

В ході виконання дипломного проекту було розроблене та реалізоване програмне забезпечення для системи 3D моделювання. В ході розробки використовувались алгоритми, що дозволяють виконувати різні операції редагування з будь-якими 3D моделями. Також були реалізовані алгоритми для відображення та підсвіти моделі під час використання програмного забезпечення. Дане програмне забезпечення написано на мові C++ з використанням OpenGL для візуалізації графіки та Qt для створення графічного інтерфейсу.

Також у даному дипломному проекті представлена робота програми на конкретній моделі. Під час демонстрації програми були виконані певні операції із моделлю, а також представлені відповідні знімки екрана.

ЛІТЕРАТУРА

1) Сфери реалізації 3D-моделювання [Електронний ресурс] – режим доступу: https://sites.google.com/site/3dmodeluvana/realizacia-3d-modeluvanna-sferi-ta
2) Найпопулярніші програми для 3D моделювання [Електронний ресурс] – режим доступу: https://cspsid-gelios.ru/uk/20-programm-dlya-3d-modelirovaniya-mgnovennoe-sozdanie-press-form.html
3) Полігональна сітка [Електронний ресурс] – режим доступу: https://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%96%D0%B3%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0_%D1%81%D1%96%D1%82%D0%BA%D0%B0
4) Неоднорідний раціональний B-сплайн [Електронний ресурс] – режим доступу: https://uk.wikipedia.org/wiki/NURBS
5) Сплайн [Електронний ресурс] – режим доступу: https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%BB%D0%B0%D0%B9%D0%BD
6) PyQt Wikipedia [Електронний ресурс] – режим доступу: https://en.wikipedia.org/wiki/PyQt
7) OpenGL Wiki [Електронний ресурс] – режим доступу: https://www.khronos.org/opengl/
8) Модель Кука-Торранса [Електронний ресурс] – режим доступу: https://sites.google.com/site/sxralanwebsite/theme/2-3-task-2-cook-shading-model
9) Шейдери(Вершинні, Геометричні) [Електронний ресурс] – режим доступу: https://uk.wikipedia.org/wiki/%D0%A8%D0%B5%D0%B9%D0%B4%D0%B5%D1%80

10) Кватерніони [Електронний ресурс] – режим доступу: https://ru.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D1%82%D0%B5%D1%80%D0%BD%D0%B8%D0%BE%D0%BD
11) OpenGL ES 3.0. Посібник розробника – Ден Гінсбург
12) Qt 5.10. Профійне програмування на C++ – Макс Шлеє
13) Програмування комп'ютерної графіки – Боресков А.В.
14) MatLAB Programming: 1st (First) Edition – David Kuncicky.
15) Open GL 4. Мова шейдеров. Книга рецептів – Д. Вольф
16) Основи комп'ютерної графіки – П. Ширлі

Додатки

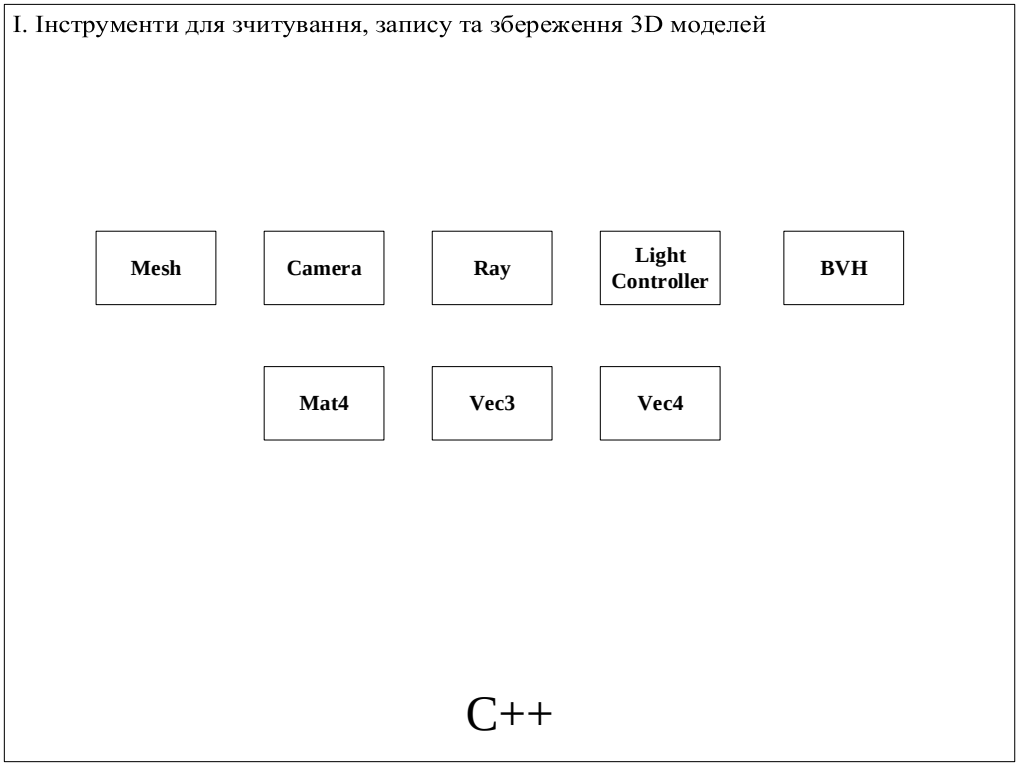
ДОДАТОК А

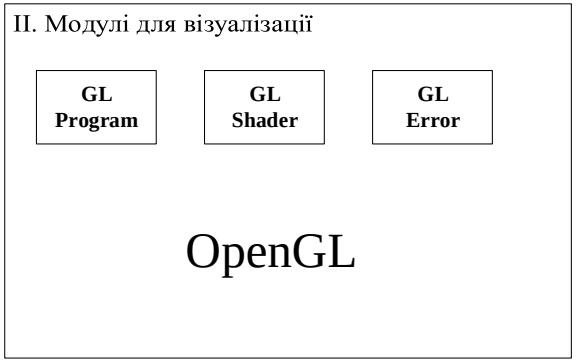
Система 3D моделювання

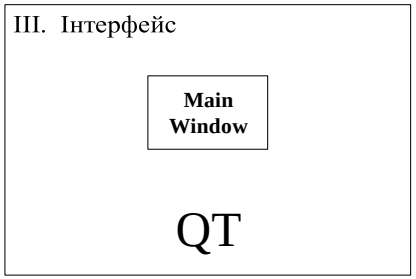
Структура проекту

Аркушів 1

Київ – 2020 року







					ДП 467200.04 Д1		
						Літера	Маса	Масштаб
Зм.	Арк.	№ докум	Підпис	Дата	Система 3D моделювання			
Розробила		Сакович М.В.						
Перевірив		Регіда П.Г.			Структура проекту			
Т. контр.						Аркуш 1	Аркушів 1	
Н. контр.		Сімоненко В.П.			Дипломний проект	НТУУ "КПІ", ФІОТ, ІО-63		
Затверд.		Регіда П.Г.						

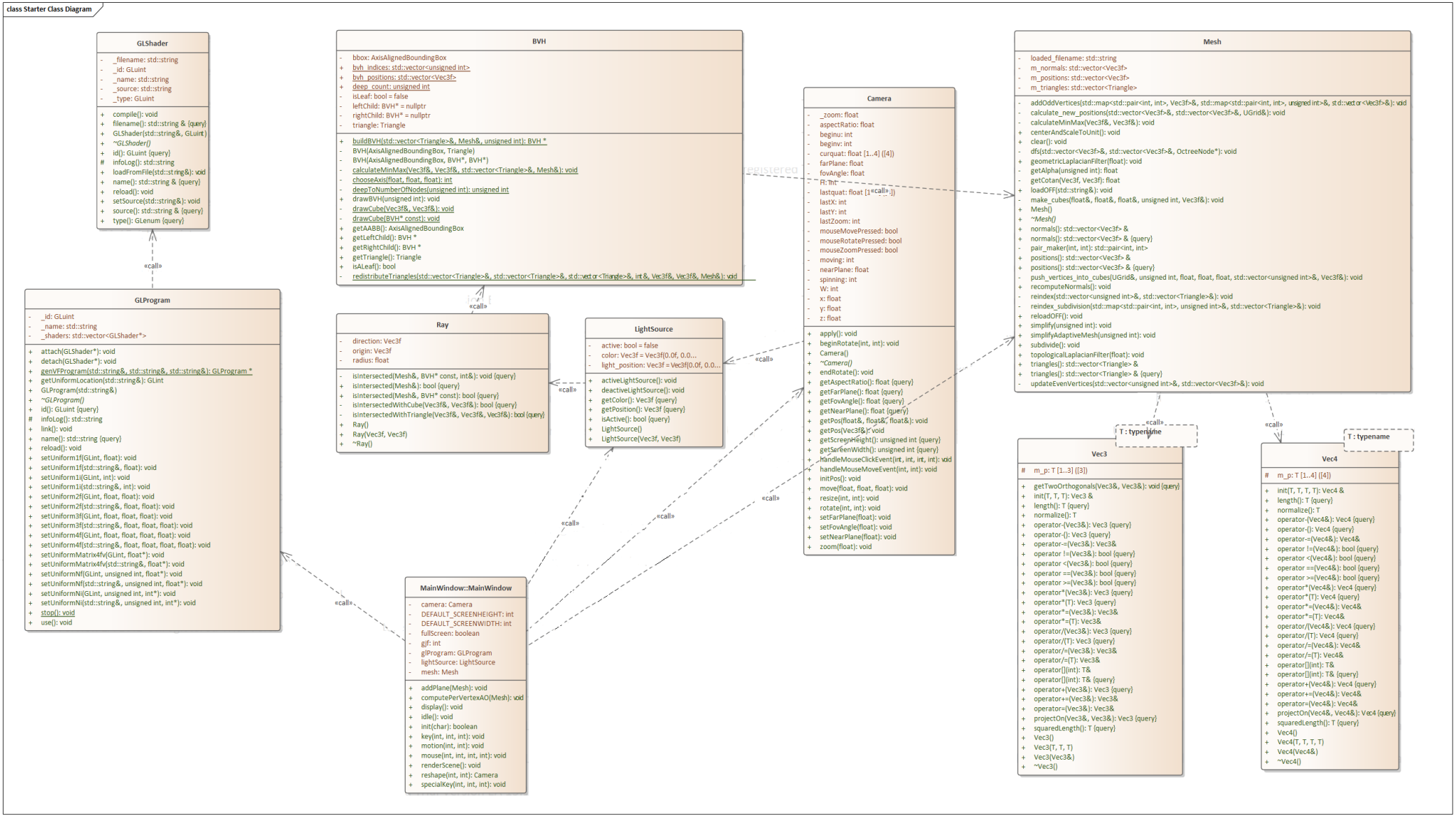
ДОДАТОК Б

Система 3D моделювання

Діаграма класів

Аркушів 1

Київ – 2020 року



						ДП 4682.05.000 Д2	
Зм.	Арк.	№ докум.	Підпис	Дата		Система 3D моделювання	Літера / Маса / Масштаб
Розробила		Сакович М.В.				Діаграма класів.	Аркуш 1 / Аркушів 1
Перевірив		Регіда П.Г.					
Т. контр.						Дипломний проект	НТУУ "КПІ", ФІОТ, ІО-63
Н. контр.		Сіменченко В.П.					
Затверд.		Регіда П.Г.					

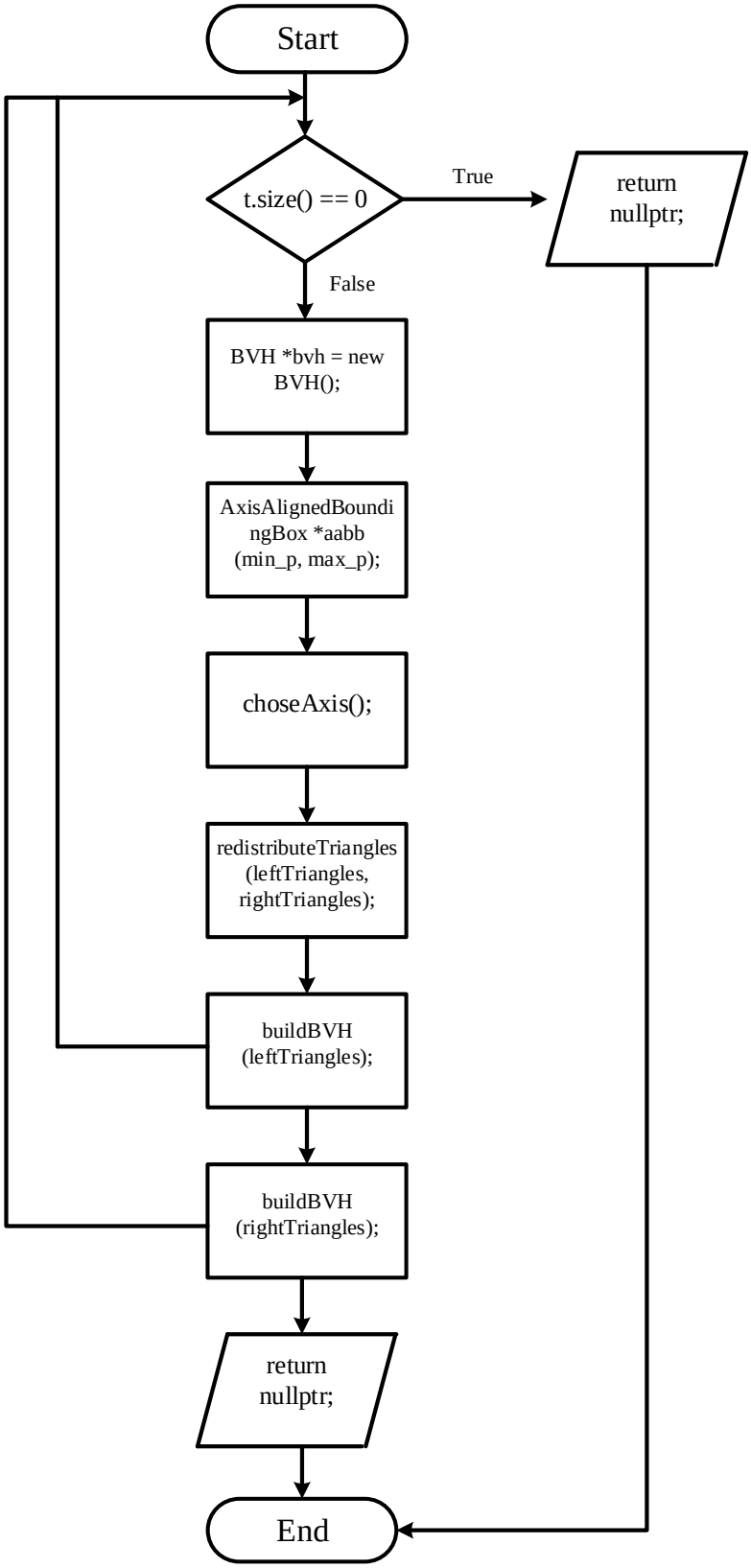
ДОДАТОК В

Система 3D моделювання

Алгоритм побудови BVH

Аркушів 1

Київ – 2020 року



					ДП 467200.04 ДЗ			
						Літера	Маса	Масштаб
Зм.	Арк.	№ докум	Підпис	Дата	Система 3D моделювання			
Розробила		Сакович М.В.						
Перевірив		Регіда П.Г.			Алгоритм побудови BVH			
Т. контр.						Аркуш 1	Аркушів 1	
Н. контр.		Сімоненко В.П.			Дипломний проект	НТУУ "КПІ", ФІОТ, ІО-63		
Затверд.		Регіда П.Г.						

ДОДАТОК Г

Система 3D моделювання

Лістинг реалізації

Аркушів 33

Київ – 2020 року

```
#include <algorithm>

#include "Mesh.h"
#include "AxisAlignedBoundingBox.h"

class BVH{
private:
  AxisAlignedBoundingBox bbox;
  BVH * leftChild = nullptr;
  BVH * rightChild = nullptr;
  bool isLeaf = false;
  Triangle triangle;

  BVH(AxisAlignedBoundingBox bbox, Triangle triangle): bbox(bbox), isLeaf(true), triangle(triangle) {}
  BVH(AxisAlignedBoundingBox bbox, BVH * l, BVH * r): bbox(bbox), leftChild(l), rightChild(r) {}

  static int chooseAxis(float x_distance, float y_distance, float z_distance);
  static void calculateMinMax(Vec3f & min_p, Vec3f & max_p, const std::vector<Triangle> & t, const
Mesh & mesh);
  static void redistributeTriangles(std::vector<Triangle> & triangles_left, std::vector<Triangle> &
triangles_right,
                        const std::vector<Triangle> & t, const int & axis, const Vec3f & max_p, const Vec3f &
min_p, const Mesh & mesh);
  static void drawCube(const Vec3f & min_p, const Vec3f & max_p);
  static void drawCube(BVH * const bvh);
  static unsigned int deepToNumberOfNodes(unsigned int currentDeep);

public:
  static std::vector<Vec3f> bvh_positions;
  static std::vector<unsigned int> bvh_indices;
  static unsigned int deep_count;

  static BVH * buildBVH(const std::vector<Triangle> & t, const Mesh & mesh, unsigned int deep_count1);

  BVH * getLeftChild() {return leftChild;}
  BVH * getRightChild() {return rightChild;}
  AxisAlignedBoundingBox getAABB() {return bbox;}
  bool isALeaf() {return isLeaf;}
  Triangle getTriangle() {return triangle;}
  void drawBVH(unsigned int currentDeep);
};

#include <queue>
#include <cmath>
#include "BVH.h"

using namespace std;

int BVH::chooseAxis(float x_distance, float y_distance, float z_distance){
        // 0 x, 1 y, 2 z
        if( x_distance >= y_distance && x_distance >= z_distance ) return 0;
        else if ( y_distance >= x_distance && y_distance >= z_distance ) return 1;
        else return 2;
}
```

```
void BVH::calculateMinMax(Vec3f & min_p, Vec3f & max_p, const vector<Triangle> & t, const Mesh &
mesh){
	for(unsigned int i = 0; i < t.size(); i++){
		if( max_p[0] < mesh.positions()[ t[i][0] ][0] ) max_p[0] = mesh.positions()[ t[i][0] ][0];
		if( max_p[1] < mesh.positions()[ t[i][0] ][1] ) max_p[1] = mesh.positions()[ t[i][0] ][1];
		if( max_p[2] < mesh.positions()[ t[i][0] ][2] ) max_p[2] = mesh.positions()[ t[i][0] ][2];
		if( min_p[0] > mesh.positions()[ t[i][0] ][0] ) min_p[0] = mesh.positions()[ t[i][0] ][0];
		if( min_p[1] > mesh.positions()[ t[i][0] ][1] ) min_p[1] = mesh.positions()[ t[i][0] ][1];
		if( min_p[2] > mesh.positions()[ t[i][0] ][2] ) min_p[2] = mesh.positions()[ t[i][0] ][2];

		if( max_p[0] < mesh.positions()[ t[i][1] ][0] ) max_p[0] = mesh.positions()[ t[i][1] ][0];
		if( max_p[1] < mesh.positions()[ t[i][1] ][1] ) max_p[1] = mesh.positions()[ t[i][1] ][1];
		if( max_p[2] < mesh.positions()[ t[i][1] ][2] ) max_p[2] = mesh.positions()[ t[i][1] ][2];
		if( min_p[0] > mesh.positions()[ t[i][1] ][0] ) min_p[0] = mesh.positions()[ t[i][1] ][0];
		if( min_p[1] > mesh.positions()[ t[i][1] ][1] ) min_p[1] = mesh.positions()[ t[i][1] ][1];
		if( min_p[2] > mesh.positions()[ t[i][1] ][2] ) min_p[2] = mesh.positions()[ t[i][1] ][2];

		if( max_p[0] < mesh.positions()[ t[i][2] ][0] ) max_p[0] = mesh.positions()[ t[i][2] ][0];
		if( max_p[1] < mesh.positions()[ t[i][2] ][1] ) max_p[1] = mesh.positions()[ t[i][2] ][1];
		if( max_p[2] < mesh.positions()[ t[i][2] ][2] ) max_p[2] = mesh.positions()[ t[i][2] ][2];
		if( min_p[0] > mesh.positions()[ t[i][2] ][0] ) min_p[0] = mesh.positions()[ t[i][2] ][0];
		if( min_p[1] > mesh.positions()[ t[i][2] ][1] ) min_p[1] = mesh.positions()[ t[i][2] ][1];
		if( min_p[2] > mesh.positions()[ t[i][2] ][2] ) min_p[2] = mesh.positions()[ t[i][2] ][2];
	}
}

void BVH::redistributeTriangles(vector<Triangle> & triangles_left, vector<Triangle> & triangles_right,
const vector<Triangle> & t,
    const int & axis, const Vec3f & max_p, const Vec3f & min_p, const Mesh & mesh){
	vector<float> centroids;
	for(unsigned int i = 0; i < t.size(); i++){
		float p0 = mesh.positions()[ t[i][0] ][axis];
		float p1 = mesh.positions()[ t[i][1] ][axis];
		float p2 = mesh.positions()[ t[i][2] ][axis];
		float centroid = ( p0 + p1 + p2 ) / 3.0f;
		centroids.push_back(centroid);
	}
	sort( centroids.begin(), centroids.end() );
	unsigned int median_index = centroids.size() / 2;
	for(unsigned int i = 0; i < t.size(); i++){
		if( i < median_index ) triangles_left.push_back( t[i] );
		else triangles_right.push_back( t[i] );
	}
}

BVH * BVH::buildBVH(const vector<Triangle> & t, const Mesh & mesh, unsigned int deep_count1){
	if( deep_count < deep_count1 ) deep_count = deep_count1;
	if( t.size() == 0 ) return nullptr;
	float max_float = numeric_limits<float>::max();
	float min_float =  - ( numeric_limits<float>::max() - 1);
	Vec3f max_p = Vec3f(min_float, min_float, min_float);
	Vec3f min_p = Vec3f(max_float, max_float, max_float);
	calculateMinMax(min_p, max_p, t, mesh);
```

```
        AxisAlignedBoundingBox aabb(min_p, max_p);
        if( t.size() == 1 ){
                BVH * bvh = new BVH(aabb, t[0]);
                return bvh;
        }else{
                int axis = chooseAxis(max_p[0] - min_p[0], max_p[1] - min_p[1], max_p[2] - min_p[2]);
                vector<Triangle> triangles_left, triangles_right;
                redistributeTriangles(triangles_left, triangles_right, t, axis, max_p, min_p, mesh);
                BVH * left_c = buildBVH(triangles_left, mesh, deep_count1 + 1);
                BVH * right_c = buildBVH(triangles_right, mesh, deep_count1 + 1);
                BVH * bvh = new BVH(aabb, left_c, right_c );
                return bvh;
        }
}

void BVH::drawBVH(unsigned int currentDeep){
 unsigned int k = 0;
 bvh_positions.clear();
 bvh_indices.clear();

        unsigned int numberOfNodes = deepToNumberOfNodes(currentDeep);
        if( numberOfNodes == 0 ) return;
        unsigned int numberOfNodesOfPreviousLayer = deepToNumberOfNodes(currentDeep - 1);

 //BFS
 std::queue<BVH *> visited, unvisited;
 BVH * current;
 unvisited.push(this);
 while ( ! unvisited.empty() ){
  current = unvisited.front();
  if (current->getLeftChild()  != nullptr) unvisited.push( current->getLeftChild()  );
  if (current->getRightChild() != nullptr) unvisited.push( current->getRightChild() );
  visited.push(current);
                if( k >= numberOfNodesOfPreviousLayer ) drawCube(current);
  unvisited.pop();
  k++;
  if( k >= numberOfNodes ) break;
 }
}

void BVH::drawCube(BVH * const bvh){
 Vec3f min_p = ( bvh->getAABB() ).getMinPoint();
        Vec3f max_p = ( bvh->getAABB() ).getMaxPoint();
        drawCube(min_p, max_p);
}

unsigned int BVH::deepToNumberOfNodes(unsigned int currentDeep){
 unsigned int sum = 0;
 for(unsigned int i = 1; i <= currentDeep; i++){
  sum += pow(2, i - 1);
 }
 return sum;
}
```

```
void BVH::drawCube(const Vec3f & min_p, const Vec3f & max_p){
        int first_index = bvh_positions.size() - 1;
        Vec3f p1 = Vec3f(min_p[0], max_p[1], max_p[2]);
        Vec3f p2 = Vec3f(min_p[0], max_p[1], min_p[2]);
        Vec3f p3 = Vec3f(max_p[0], max_p[1], min_p[2]);
        Vec3f p4 = Vec3f(max_p[0], max_p[1], max_p[2]);
  Vec3f p5 = Vec3f(min_p[0], min_p[1], max_p[2]);
  Vec3f p6 = Vec3f(min_p[0], min_p[1], min_p[2]);
  Vec3f p7 = Vec3f(max_p[0], min_p[1], min_p[2]);
  Vec3f p8 = Vec3f(max_p[0], min_p[1], max_p[2]);

        bvh_positions.push_back(p1);
        bvh_positions.push_back(p2);
        bvh_positions.push_back(p3);
        bvh_positions.push_back(p4);
  bvh_positions.push_back(p5);
  bvh_positions.push_back(p6);
  bvh_positions.push_back(p7);
  bvh_positions.push_back(p8);

        bvh_indices.push_back(first_index + 1);
        bvh_indices.push_back(first_index + 2);
        bvh_indices.push_back(first_index + 2);
        bvh_indices.push_back(first_index + 3);
        bvh_indices.push_back(first_index + 3);
        bvh_indices.push_back(first_index + 4);
        bvh_indices.push_back(first_index + 4);
        bvh_indices.push_back(first_index + 1);

  bvh_indices.push_back(first_index + 5);
        bvh_indices.push_back(first_index + 6);
        bvh_indices.push_back(first_index + 6);
        bvh_indices.push_back(first_index + 7);
        bvh_indices.push_back(first_index + 7);
        bvh_indices.push_back(first_index + 8);
        bvh_indices.push_back(first_index + 8);
        bvh_indices.push_back(first_index + 5);

  bvh_indices.push_back(first_index + 1);
        bvh_indices.push_back(first_index + 5);
        bvh_indices.push_back(first_index + 2);
        bvh_indices.push_back(first_index + 6);
        bvh_indices.push_back(first_index + 3);
        bvh_indices.push_back(first_index + 7);
        bvh_indices.push_back(first_index + 4);
        bvh_indices.push_back(first_index + 8);
}


#pragma once

#include "Vec3.h"

class Camera {
```

```
public:
  Camera ();
  virtual ~Camera () {}

  inline float getFovAngle () const { return fovAngle; }
  inline void setFovAngle (float newFovAngle) { fovAngle = newFovAngle; }
  inline float getAspectRatio () const { return aspectRatio; }
  inline float getNearPlane () const { return nearPlane; }
  inline void setNearPlane (float newNearPlane) { nearPlane = newNearPlane; }
  inline float getFarPlane () const { return farPlane; }
  inline void setFarPlane (float newFarPlane) { farPlane = newFarPlane; }
  inline unsigned int getScreenWidth () const { return W; }
  inline unsigned int getScreenHeight () const { return H; }

  void resize (int W, int H);

  void initPos ();

  void move (float dx, float dy, float dz);
  void beginRotate (int u, int v);
  void rotate (int u, int v);
  void endRotate ();
  void zoom (float z);
  void apply ();

  void getPos (float & x, float & y, float & z);
  inline void getPos (Vec3f & p) { getPos (p[0], p[1], p[2]); }

  // Connecting typical GLUT events
  void handleMouseClickEvent (int button, int state, int x, int y);
  void handleMouseMoveEvent (int x, int y);

private:
  float fovAngle;
  float aspectRatio;
  float nearPlane;
  float farPlane;

  int spinning, moving;
  int beginu, beginv;
  int H, W;
  float curquat[4];
  float lastquat[4];
  float x, y, z;
  float _zoom;

  bool mouseRotatePressed;
  bool mouseMovePressed;
  bool mouseZoomPressed;
  int lastX, lastY, lastZoom;
};


#include "Camera.h"
```

```
#include <GL/glew.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <iostream>
#include <string>

void
trackball(float q[4], float p1x, float p1y, float p2x, float p2y);

void
negate_quat(float *q, float *qn);

void
add_quats(float *q1, float *q2, float *dest);

void
build_rotmatrix(float m[4][4], float q[4]);

void
axis_to_quat(float a[3], float phi, float q[4]);

using namespace std;

static int _spinning, _moving;
static int _beginu, _beginv;
static float _curquat[4];
static float _x, _y, _z;
static float __zoom;
static bool ini = false;

Camera::Camera () {
  fovAngle = 45.0;
  aspectRatio = 1.0;
  nearPlane = 0.1;
  farPlane = 10000.0;

  spinning = 0;
  moving = 0;
  beginu = 0;
  beginv = 0;

  trackball (curquat, 0.0, 0.0, 0.0, 0.0);
  x = y = z = 0.0;
  _zoom = 3.0;

  mouseRotatePressed = false;
  mouseMovePressed = false;
  mouseZoomPressed = false;
  lastX = 0;
  lastY = 0;
  lastZoom = 0;
}
```

```
void Camera::resize (int _W, int _H) {
  H = _H;
  W = _W;
  glViewport (0, 0, (GLint)W, (GLint)H);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  aspectRatio = static_cast<float>(W)/static_cast<float>(H);
  gluPerspective (fovAngle, aspectRatio, nearPlane, farPlane);
  glMatrixMode (GL_MODELVIEW);
}


void Camera::initPos () {
  if (!ini) {
  _spinning = spinning;
  _moving = moving;;
  _beginu = beginu;
  _beginv = beginv;
  _curquat[0] = curquat[0];
  _curquat[1] = curquat[1];
  _curquat[2] = curquat[2];
  _curquat[3] = curquat[3];
  _x = x;
  _y = y;
  _z = z;;
  __zoom = _zoom;
  ini = true;
  } else {
    spinning = _spinning;
    moving = _moving;;
    beginu = _beginu;
    beginv = _beginv;
    curquat[0] = _curquat[0];
    curquat[1] = _curquat[1];
    curquat[2] = _curquat[2];
    curquat[3] = _curquat[3];
    x = _x;
    y = _y;
    z = _z;;
    _zoom = __zoom;
  }
}


void Camera::move (float dx, float dy, float dz) {
  x += dx;
  y += dy;
  z += dz;
}


void Camera::beginRotate (int u, int v) {
  beginu = u;
  beginv = v;
```

```
  moving = 1;
  spinning = 0;
}


void Camera::rotate (int u, int v) {
  if (moving) {
    trackball(lastquat,
                (2.0 * beginu - W) / W,
                (H - 2.0 * beginv) / H,
                (2.0 * u - W) / W,
                (H - 2.0 * v) / H);
    beginu = u;
    beginv = v;
    spinning = 1;
    add_quats (lastquat, curquat, curquat);
  }
}


void Camera::endRotate () {
  moving = false;
}


void Camera::zoom (float z) {
  _zoom += z;
}


void Camera::apply () {
  glLoadIdentity();
  glTranslatef (x, y, z);
  GLfloat m[4][4];
  build_rotmatrix(m, curquat);
  glTranslatef (0.0, 0.0, -_zoom);
  glMultMatrixf(&m[0][0]);
}


void Camera::getPos (float & X, float & Y, float & Z) {
  GLfloat m[4][4];
  build_rotmatrix(m, curquat);
  float _x = -x;
  float _y = -y;
  float _z = -z + _zoom;
  X = m[0][0] * _x +  m[0][1] * _y +  m[0][2] * _z;
  Y = m[1][0] * _x +  m[1][1] * _y +  m[1][2] * _z;
  Z = m[2][0] * _x +  m[2][1] * _y +  m[2][2] * _z;
}

void Camera::handleMouseClickEvent (int button, int state, int x, int y) {
        if (state == GLUT_UP) {
      mouseMovePressed = false;
```

```
      mouseRotatePressed = false;
      mouseZoomPressed = false;
   } else {
      if (button == GLUT_LEFT_BUTTON) {
         beginRotate (x, y);
         mouseMovePressed = false;
         mouseRotatePressed = true;
         mouseZoomPressed = false;
      } else if (button == GLUT_RIGHT_BUTTON) {
         lastX = x;
         lastY = y;
         mouseMovePressed = true;
         mouseRotatePressed = false;
         mouseZoomPressed = false;
      } else if (button == GLUT_MIDDLE_BUTTON) {
         if (mouseZoomPressed == false) {
            lastZoom = y;
            mouseMovePressed = false;
            mouseRotatePressed = false;
            mouseZoomPressed = true;
         }
      }
   }
}

void Camera::handleMouseMoveEvent (int x, int y) {
           if (mouseRotatePressed == true)
      rotate (x, y);
   else if (mouseMovePressed == true) {
      move ((x-lastX)/static_cast<float>(getScreenWidth ()), (lastY-y)/static_cast<float>(getScreenHeight
()), 0.0);
      lastX = x;
      lastY = y;
   }
   else if (mouseZoomPressed == true) {
      zoom (float (y-lastZoom)/getScreenHeight ());
      lastZoom = y;
   }
}

#include <cstdio>

#if defined(_WIN32)
#pragma warning (disable:4244)
#endif
#include <cmath>

#define TRACKBALLSIZE  (0.8f)

static float tb_project_to_sphere(float, float, float);
static void normalize_quat(float [4]);

void
vzero(float *v)
```

```
{
   v[0] = 0.0;
   v[1] = 0.0;
   v[2] = 0.0;
}

void
vset(float *v, float x, float y, float z)
{
   v[0] = x;
   v[1] = y;
   v[2] = z;
}

void
vsub(const float *src1, const float *src2, float *dst)
{
   dst[0] = src1[0] - src2[0];
   dst[1] = src1[1] - src2[1];
   dst[2] = src1[2] - src2[2];
}

void
vcopy(const float *v1, float *v2)
{
   register int i;
   for (i = 0 ; i < 3 ; i++)
      v2[i] = v1[i];
}

void
vcross(const float *v1, const float *v2, float *cross)
{
   float temp[3];

   temp[0] = (v1[1] * v2[2]) - (v1[2] * v2[1]);
   temp[1] = (v1[2] * v2[0]) - (v1[0] * v2[2]);
   temp[2] = (v1[0] * v2[1]) - (v1[1] * v2[0]);
   vcopy(temp, cross);
}

float
vlength(const float *v)
{
   return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
}

void
vscale(float *v, float div)
{
   v[0] *= div;
   v[1] *= div;
   v[2] *= div;
}
```

```
void
vnormal(float *v)
{
   vscale(v,1.0/vlength(v));
}

float
vdot(const float *v1, const float *v2)
{
   return v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2];
}

void
vadd(const float *src1, const float *src2, float *dst)
{
   dst[0] = src1[0] + src2[0];
   dst[1] = src1[1] + src2[1];
   dst[2] = src1[2] + src2[2];
}

void
trackball(float q[4], float p1x, float p1y, float p2x, float p2y)
{
   float a[3]; /* Axis of rotation */
   float phi;  /* how much to rotate about axis */
   float p1[3], p2[3], d[3];
   float t;

   if (p1x == p2x && p1y == p2y) {
      /* Zero rotation */
      vzero(q);
      q[3] = 1.0;
      return;
   }

   vset(p1,p1x,p1y,tb_project_to_sphere(TRACKBALLSIZE,p1x,p1y));
   vset(p2,p2x,p2y,tb_project_to_sphere(TRACKBALLSIZE,p2x,p2y));

   vcross(p2,p1,a);

   vsub(p1,p2,d);
   t = vlength(d) / (2.0*TRACKBALLSIZE);

   if (t > 1.0) t = 1.0;
   if (t < -1.0) t = -1.0;
   phi = 2.0 * asin(t);

   axis_to_quat(a,phi,q);
}

void
axis_to_quat(float a[3], float phi, float q[4])
{
```

```
   vnormal(a);
   vcopy(a,q);
   vscale(q,sin(phi/2.0));
   q[3] = cos(phi/2.0);
}

static float
tb_project_to_sphere(float r, float x, float y)
{
   float d, t, z;

   d = sqrt(x*x + y*y);
   if (d < r * 0.70710678118654752440) {    /* Inside sphere */
      z = sqrt(r*r - d*d);
   } else {          /* On hyperbola */
      t = r / 1.41421356237309504880;
      z = t*t / d;
   }
   return z;
}

#define RENORMCOUNT 97

void
negate_quat(float q[4], float nq[4])
{
   nq[0] = -q[0];
   nq[1] = -q[1];
   nq[2] = -q[2];
   nq[3] = q[3];
}

void
add_quats(float q1[4], float q2[4], float dest[4])
{
   static int count=0;
   float t1[4], t2[4], t3[4];
   float tf[4];

#if 0
printf("q1 = %f %f %f %f\n", q1[0], q1[1], q1[2], q1[3]);
printf("q2 = %f %f %f %f\n", q2[0], q2[1], q2[2], q2[3]);
#endif

   vcopy(q1,t1);
   vscale(t1,q2[3]);

   vcopy(q2,t2);
   vscale(t2,q1[3]);

   vcross(q2,q1,t3);
   vadd(t1,t2,tf);
   vadd(t3,tf,tf);
   tf[3] = q1[3] * q2[3] - vdot(q1,q2);
```

```
#if 0
printf("tf = %f %f %f %f\n", tf[0], tf[1], tf[2], tf[3]);
#endif

   dest[0] = tf[0];
   dest[1] = tf[1];
   dest[2] = tf[2];
   dest[3] = tf[3];

   if (++count > RENORMCOUNT) {
      count = 0;
      normalize_quat(dest);
   }
}

static void
normalize_quat(float q[4])
{
   int i;
   float mag;

   mag = sqrt(q[0]*q[0] + q[1]*q[1] + q[2]*q[2] + q[3]*q[3]);
   for (i = 0; i < 4; i++) q[i] /= mag;
}

void
build_rotmatrix(float m[4][4], float q[4])
{
   m[0][0] = 1.0 - 2.0 * (q[1] * q[1] + q[2] * q[2]);
   m[0][1] = 2.0 * (q[0] * q[1] - q[2] * q[3]);
   m[0][2] = 2.0 * (q[2] * q[0] + q[1] * q[3]);
   m[0][3] = 0.0;

   m[1][0] = 2.0 * (q[0] * q[1] + q[2] * q[3]);
   m[1][1]= 1.0 - 2.0 * (q[2] * q[2] + q[0] * q[0]);
   m[1][2] = 2.0 * (q[1] * q[2] - q[0] * q[3]);
   m[1][3] = 0.0;

   m[2][0] = 2.0 * (q[2] * q[0] - q[1] * q[3]);
   m[2][1] = 2.0 * (q[1] * q[2] + q[0] * q[3]);
   m[2][2] = 1.0 - 2.0 * (q[1] * q[1] + q[0] * q[0]);
   m[2][3] = 0.0;

   m[3][0] = 0.0;
   m[3][1] = 0.0;
   m[3][2] = 0.0;
   m[3][3] = 1.0;
}

#include "Vec3.h"

class LightSource{
  private:
```

```
  Vec3f light_position = Vec3f(0.0f, 0.0f, 0.0f);
  Vec3f color = Vec3f(0.0f, 0.0f, 0.0f);
  bool active = false;
 public:
  LightSource();
  LightSource(Vec3f light_position, Vec3f color);
  void activeLightSource() {active = true;}
  void deactiveLightSource() {active = false;}
  Vec3f getColor() const {return color;}
  Vec3f getPosition() const {return light_position;}
  bool isActive() const {return active;}
  //void moveXBy(float delta) {light_position += Vec3f(delta, 0.0f, 0.0f);}
};

#include "LightSource.h"

LightSource::LightSource(): active(false) {}

LightSource::LightSource(Vec3f light_position, Vec3f color): light_position(light_position), color(color),
active(false) {}

#pragma once

#include <cmath>
#include <iostream>
#include <algorithm>

#include "Vec3.h"

template <class T>
class Mat4 {
public:
  class Exception {
  public:
    inline Exception (const std::string & msg) : m_msg ("Blade Mat4 Exception: " + msg) {}
    inline const std::string & msg () const { return m_msg; }
  protected:
    std::string m_msg;
  };

  inline Mat4 (void)      { loadIdentity (); }
  inline Mat4 (const Mat4 & mat) {
    for (unsigned int i = 0; i < 16; i++)
      m_m[i] = mat[i];
  }
  ~Mat4() {}
  inline Mat4 (T a00, T a01, T a02, T a03,
    T a10, T a11, T a12, T a13,
    T a20, T a21, T a22, T a23,
    T a30, T a31, T a32, T a33) {
    set (a00, a01, a02, a03,
      a10, a11, a12, a13,
      a20, a21, a22, a23,
      a30, a31, a32, a33);
```

```
}
inline void set (T a00, T a01, T a02, T a03,
  T a10, T a11, T a12, T a13,
  T a20, T a21, T a22, T a23,
  T a30, T a31, T a32, T a33) {
   m_m[0] = a00; m_m[1] = a01; m_m[2] = a02; m_m[3] = a03;
   m_m[4] = a10; m_m[5] = a11; m_m[6] = a12; m_m[7] = a13;
   m_m[8] = a20; m_m[9] = a21; m_m[10] = a22; m_m[11] = a23;
   m_m[12] = a30; m_m[13] = a31; m_m[14] = a32; m_m[15] = a33;
}
inline void set (const T * a) { for (unsigned int i = 0; i < 16; i++) m_m[i] = a[i]; }
inline void setNull () {  set (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0); }
inline void loadIdentity () { setNull (); m_m[0] = m_m[5] = m_m[10] = m_m[15] = 1.0; }
inline T& operator[] (int index) { return (m_m[index]); }
inline const T& operator[] (int index) const { return (m_m[index]); }
inline T& operator() (int i, int j) { return (m_m[4*i+j]); }
inline const T& operator() (int i, int j) const { return (m_m[4*i+j]); }
inline Mat4& operator= (const Mat4 & mat) {
   for (unsigned int i = 0; i < 16; i++)
      m_m[i] = mat[i];
   return (*this);
}
inline Mat4 operator+ (const Mat4 & mat) const {
   Mat4 res;
   for (unsigned int i = 0; i < 16; i++)
      res[i] = m_m[i] + mat[i];
   return res;
}
inline Mat4 operator- (const Mat4 & mat) const {
   Mat4 res;
   for (unsigned int i = 0; i < 16; i++)
      res[i] = m_m[i] - mat[i];
   return res;
}
inline Mat4 operator- () const {
   Mat4 res;
   for (unsigned int i = 0; i < 16; i++)
      res[i] = -m_m[i];
   return res;
}
inline Mat4 operator* (const Mat4 & mat) const {
   Mat4 tmp;
   tmp.setNull ();
   for (unsigned int i = 0; i < 4; i++)
      for (unsigned int j = 0; j < 4; j++)
         for (unsigned int k = 0; k < 4; k++)
            tmp(i, j) += (*this)(k, j)*mat(i,k);
         return tmp;
      }
      inline Mat4 operator* (T s) const {
         Mat4 res;
         for (unsigned int i = 0; i < 16; i++)
            res[i] = s*m_m[i];
         return res;
```

```
}
inline Mat4 operator/ (T s) const {
   Mat4 res;
   for (unsigned int i = 0; i < 16; i++)
      res[i] = m_m[i]/s;
   return res;
}
inline Vec3<T> operator* (const Vec3<T> & v) const {
   T tmp[4];
   tmp[0] = v[0];
   tmp[1] = v[1];
   tmp[2] = v[2];
   tmp[3] = 1.0;
   T res[4];
   res[0] = 0;
   res[1] = 0;
   res[2] = 0;
   res[3] = 0;
   for (unsigned int i = 0; i < 4; i++)
      for (unsigned int j = 0; j < 4; j++)
         res[i] += tmp[j]*(*this)(j, i);
      return Vec3<T> (res[0], res[1], res[2])/res[3];
   }
   inline Mat4& operator+= (const Mat4 & mat) {
      Mat4 tmp = (*this) + mat;
      (*this) = tmp;
      return (*this);
   }
   inline Mat4& operator-= (const Mat4 & mat) {
      Mat4 tmp = (*this) - mat;
      (*this) = tmp;
      return (*this);
   }
   inline Mat4& operator*= (T & s) {
      Mat4 tmp = (*this) * s;
      (*this) = tmp;
      return (*this);
   }
   inline Mat4& operator*= (const Mat4 & mat) {
      Mat4 tmp = (*this) * mat;
      (*this) = tmp;
      return (*this);
   }
   inline bool operator == (const Mat4 & mat) const {
      for (unsigned int i = 0; i < 16; i++)
         if (m_m[i] != mat.m_m[i])
            return false;
         return true;
      }
      inline bool operator != (const Mat4 & mat) const {
         return !((*this)==mat);
      }
      inline T * data () { return m_m; }
      inline const T * data () const { return m_m; }
```

```
inline Mat4 & transpose () {
   for (unsigned int i = 0; i < 4; i++)
      for (unsigned int j = i+1; j < 4; j++)
         std::swap ((*this) (i, j), (*this) (j, i));
      return (*this);
   }
   inline Mat4 & invert () {
      double inv[16], det;
      int i;
      inv[0] = m_m[5]  * m_m[10] * m_m[15] -
      m_m[5]  * m_m[11] * m_m[14] -
      m_m[9]  * m_m[6]  * m_m[15] +
      m_m[9]  * m_m[7]  * m_m[14] +
      m_m[13] * m_m[6]  * m_m[11] -
      m_m[13] * m_m[7]  * m_m[10];

      inv[4] = -m_m[4]  * m_m[10] * m_m[15] +
      m_m[4]  * m_m[11] * m_m[14] +
      m_m[8]  * m_m[6]  * m_m[15] -
      m_m[8]  * m_m[7]  * m_m[14] -
      m_m[12] * m_m[6]  * m_m[11] +
      m_m[12] * m_m[7]  * m_m[10];

      inv[8] = m_m[4]  * m_m[9] * m_m[15] -
      m_m[4]  * m_m[11] * m_m[13] -
      m_m[8]  * m_m[5] * m_m[15] +
      m_m[8]  * m_m[7] * m_m[13] +
      m_m[12] * m_m[5] * m_m[11] -
      m_m[12] * m_m[7] * m_m[9];

      inv[12] = -m_m[4]  * m_m[9] * m_m[14] +
      m_m[4]  * m_m[10] * m_m[13] +
      m_m[8]  * m_m[5] * m_m[14] -
      m_m[8]  * m_m[6] * m_m[13] -
      m_m[12] * m_m[5] * m_m[10] +
      m_m[12] * m_m[6] * m_m[9];

      inv[1] = -m_m[1]  * m_m[10] * m_m[15] +
      m_m[1]  * m_m[11] * m_m[14] +
      m_m[9]  * m_m[2] * m_m[15] -
      m_m[9]  * m_m[3] * m_m[14] -
      m_m[13] * m_m[2] * m_m[11] +
      m_m[13] * m_m[3] * m_m[10];

      inv[5] = m_m[0]  * m_m[10] * m_m[15] -
      m_m[0]  * m_m[11] * m_m[14] -
      m_m[8]  * m_m[2] * m_m[15] +
      m_m[8]  * m_m[3] * m_m[14] +
      m_m[12] * m_m[2] * m_m[11] -
      m_m[12] * m_m[3] * m_m[10];

      inv[9] = -m_m[0]  * m_m[9] * m_m[15] +
      m_m[0]  * m_m[11] * m_m[13] +
      m_m[8]  * m_m[1] * m_m[15] -
```

```
m_m[8]  * m_m[3] * m_m[13] -
m_m[12] * m_m[1] * m_m[11] +
m_m[12] * m_m[3] * m_m[9];

inv[13] = m_m[0]  * m_m[9] * m_m[14] -
m_m[0]  * m_m[10] * m_m[13] -
m_m[8]  * m_m[1] * m_m[14] +
m_m[8]  * m_m[2] * m_m[13] +
m_m[12] * m_m[1] * m_m[10] -
m_m[12] * m_m[2] * m_m[9];

inv[2] = m_m[1]  * m_m[6] * m_m[15] -
m_m[1]  * m_m[7] * m_m[14] -
m_m[5]  * m_m[2] * m_m[15] +
m_m[5]  * m_m[3] * m_m[14] +
m_m[13] * m_m[2] * m_m[7] -
m_m[13] * m_m[3] * m_m[6];

inv[6] = -m_m[0]  * m_m[6] * m_m[15] +
m_m[0]  * m_m[7] * m_m[14] +
m_m[4]  * m_m[2] * m_m[15] -
m_m[4]  * m_m[3] * m_m[14] -
m_m[12] * m_m[2] * m_m[7] +
m_m[12] * m_m[3] * m_m[6];

inv[10] = m_m[0]  * m_m[5] * m_m[15] -
m_m[0]  * m_m[7] * m_m[13] -
m_m[4]  * m_m[1] * m_m[15] +
m_m[4]  * m_m[3] * m_m[13] +
m_m[12] * m_m[1] * m_m[7] -
m_m[12] * m_m[3] * m_m[5];

inv[14] = -m_m[0]  * m_m[5] * m_m[14] +
m_m[0]  * m_m[6] * m_m[13] +
m_m[4]  * m_m[1] * m_m[14] -
m_m[4]  * m_m[2] * m_m[13] -
m_m[12] * m_m[1] * m_m[6] +
m_m[12] * m_m[2] * m_m[5];

inv[3] = -m_m[1] * m_m[6] * m_m[11] +
m_m[1] * m_m[7] * m_m[10] +
m_m[5] * m_m[2] * m_m[11] -
m_m[5] * m_m[3] * m_m[10] -
m_m[9] * m_m[2] * m_m[7] +
m_m[9] * m_m[3] * m_m[6];

inv[7] = m_m[0] * m_m[6] * m_m[11] -
m_m[0] * m_m[7] * m_m[10] -
m_m[4] * m_m[2] * m_m[11] +
m_m[4] * m_m[3] * m_m[10] +
m_m[8] * m_m[2] * m_m[7] -
m_m[8] * m_m[3] * m_m[6];

inv[11] = -m_m[0] * m_m[5] * m_m[11] +
```

```
                m_m[0] * m_m[7] * m_m[9] +
                m_m[4] * m_m[1] * m_m[11] -
                m_m[4] * m_m[3] * m_m[9] -
                m_m[8] * m_m[1] * m_m[7] +
                m_m[8] * m_m[3] * m_m[5];

                inv[15] = m_m[0] * m_m[5] * m_m[10] -
                m_m[0] * m_m[6] * m_m[9] -
                m_m[4] * m_m[1] * m_m[10] +
                m_m[4] * m_m[2] * m_m[9] +
                m_m[8] * m_m[1] * m_m[6] -
                m_m[8] * m_m[2] * m_m[5];

                det = m_m[0] * inv[0] + m_m[1] * inv[4] + m_m[2] * inv[8] + m_m[3] * inv[12];
                if (det == 0)
                  throw Exception ("Matrix non-invertible (null determinant).");
                det = 1.0 / det;
                for (i = 0; i < 16; i++)
                  m_m[i] = inv[i] * det;
                return (*this);
              }

    /// Left axis
              inline void setAxis (unsigned int axis, const Vec3<T> & x) { for (unsigned int i = 0; i < 3; i++)
(*this)(axis, i) = x[i]; }
              inline Vec3<T> getAxis (unsigned int axis) const { return Vec3<T> (m_m[4*axis],
m_m[4*axis+1], m_m[4*axis+2]); }
              inline Vec3<T> getTranslation () const { return Vec3<T> (m_m[12], m_m[13], m_m[14]); }
              inline static Mat4 translation (const Vec3<T> & t) {
                Mat4 m;
                m.loadIdentity ();
                m[12] = t[0];
                m[13] = t[1];
                m[14] = t[2];
                return m;
              }
              inline static Mat4 rotate (const Vec3<T> & axis, T angle) {
                Vec3<T> a = normalize (axis);
                T s = sin (angle);
                T c = cos (angle);
                T oc = 1.0 - c;
                return Mat4 (oc * a[0] * a[0] + c,         oc * a[0] * a[1] - a[2] * s,  oc * a[2] * a[0] + a[1] *
s,  0.0,
                  oc * a[0] * a[1] + a[2] * s,  oc * a[1] * a[1] + c,         oc * a[1] * a[2] - a[0] * s,  0.0,
                  oc * a[2] * a[0] - a[1] * s,  oc * a[1] * a[2] + a[0] * s,  oc * a[2] * a[2] + c,         0.0,
                  0.0,                          0.0,                          0.0,                          1.0);
              }
              inline static Mat4 scale (const Vec3<T> & s) {
                Mat4 m;
                m.loadIdentity ();
                m[0] = s[0];
                m[5] = s[1];
                m[10] = s[2];
                return m;
```

```
            }
            inline static Mat4 lookAt(const Vec3<T> & eye, const Vec3<T> & center, const Vec3<T> &
up) {
               Mat4 m;
               m.loadIdentity();
               Vec3<T> f (normalize (center - eye));
               Vec3<T> s = normalize (cross (f, up));
               Vec3<T> u = normalize (cross (s, f));
               m[0] = s[0];
               m[4] = s[1];
               m[8] = s[2];
               m[1] = u[0];
               m[5] = u[1];
               m[9] = u[2];
               m[2] = -f[0];
               m[6] = -f[1];
               m[10] = -f[2];
               m[12] = -dot(s, eye);
               m[13] = -dot(u, eye);
               m[14] = dot(f, eye);
               return m;
            }
            inline static Mat4 perspective (float fovy, float aspectRatio, float n, float f) {
               Mat4 m;
               const float deg2rad = M_PI/180.0;
               float fovyRad = deg2rad*fovy;
               float tanHalfFovy = tan (fovyRad/2.0);
               for (unsigned int i = 0; i < 16; i++)
                  m[i] = 0.0;
               m[0] = 1.0 / (aspectRatio * tanHalfFovy);
               m[5] = 1.0 / tanHalfFovy;
               m[10] = -(f + n) / (f - n);
               m[11] = -1.0;
               m[14] = -2.0*(f * n) / (f - n);
               return m;
            }
         protected:
            T m_m[16];
         };

template <class T> Mat4<T> operator* (const T &s, const Mat4<T> &M) {
         return (M * s);
      }

template <class T> Mat4<T> transpose (Mat4<T> & M) {
      return M.transpose ();
   }

template <class T> Mat4<T> inverse (Mat4<T> & M) {
   return M.invert ();
}

template <class T> std::ostream & operator<< (std::ostream & output, const Mat4<T> & m) {
   for (unsigned int i = 0; i < 4; i++) {
```

```
        for (unsigned int j = 0; j < 4; j++)
            output << m(j,i) << (j < 3 ? " " : "");
        output << std::endl;
      }
      return output;
    }

    template <class T> std::istream & operator>> (std::istream & input, Mat4<T> & m) {
    for (unsigned int i = 0; i < 4; i++)
      for (unsigned int j = 0; j < 4; j++)
        input >> m (j, i);
      return input;
    }

    typedef Mat4<float> Mat4f;
    typedef Mat4<double> Mat4d;


#pragma once
#include <cmath>
#include <vector>
#include <map>
#include <set>
#include "Vec3.h"
#include "Triangle.h"
#include "OctreeNode.h"

struct UGridCell{
  Vec3f meanPosition;
  Vec3f meanNormal;
  unsigned int count;
  std::set<unsigned int> indices;

  UGridCell(): meanPosition(0.0f, 0.0f, 0.0f), meanNormal(0.0f, 0.0f, 0.0f), count(0) {}

  void addNewVertex(unsigned int index, Vec3f new_vertex, Vec3f new_normal){
    meanPosition = meanPosition * count + new_vertex;
    meanNormal = meanNormal * count + new_normal;
    count++;
    meanPosition /= count;
    meanNormal /= count;
    indices.insert(index);
  }
};

struct UGrid{
  unsigned int nx, ny, nz;
  std::vector<UGridCell> cells;

  UGrid(unsigned int x_, unsigned int y_, unsigned int z_): nx(x_), ny(y_), nz(z_) {
    cells.clear();
    cells.resize(nx * ny * nz);
  }
```

```
  UGridCell & getCell(unsigned int x, unsigned int y, unsigned int z){
    return cells[z + nz * y + nz * ny * x];
  }
};

/// A Mesh class, storing a list of vertices and a list of triangles indexed over it.
class Mesh {
public:
   inline Mesh () {}
   inline virtual ~Mesh () {}

   inline std::vector<Vec3f> & positions () { return m_positions; }
   inline const std::vector<Vec3f> & positions () const { return m_positions; }
   inline  std::vector<Vec3f> & normals () { return m_normals; }
   inline const std::vector<Vec3f> & normals () const { return m_normals; }
   inline std::vector<Triangle> & triangles () { return m_triangles; }
   inline const std::vector<Triangle> & triangles () const { return m_triangles; }

   /// Empty the positions, normals and triangles arrays.
   void clear ();

         /// Loads the mesh from a <file>.off
         void loadOFF (const std::string & filename);

   /// Compute smooth per-vertex normals
   void recomputeNormals ();

   /// scale to the unit cube and center at original
   void centerAndScaleToUnit ();

   // Topological laplace operator
   void topologicalLaplacianFilter (float laplace_alpha);

   void reloadOFF();

   // Geometric laplace operator
   void geometricLaplacianFilter (float laplace_alpha);

   // OCS simplification
   void simplify(unsigned int resolution);
   void simplifyAdaptiveMesh (unsigned int n);

   //Loop subdivision
   void subdivide();


private:
   std::vector<Vec3f> m_positions;
   std::vector<Vec3f> m_normals;
   std::vector<Triangle> m_triangles;
   std::string loaded_filename;

   std::pair<int, int> pair_maker(int a, int b);
   float getCotan(Vec3f v1, Vec3f v2);
```

```
   void calculateMinMax(Vec3f & min_p, Vec3f & max_p);
   void make_cubes(float & x_scale, float & y_scale, float & z_scale, unsigned int resolution, Vec3f &
min_p);
   void push_vertices_into_cubes(UGrid & grid, unsigned int resolution,
     float x_scale, float y_scale, float z_scale, std::vector<unsigned int> & index_map, Vec3f & min_p);
   void calculate_new_positions(std::vector<Vec3f> & new_positions, std::vector<Vec3f> &
new_normals, UGrid & grid);
   void reindex(const std::vector<unsigned int> & index_map, std::vector<Triangle> & new_triangles);
   float getAlpha(unsigned int n){ return (1.0f / 64.0f) * (40.0f - pow( (3.0f + 2.0f * cos(2 * M_PI / n) ), 2 )
); }
   void addOddVertices(std::map<std::pair<int, int>, Vec3f> & oddVertices, std::map<std::pair<int, int>,
unsigned int> & oddVertexIndices, std::vector<Vec3f> & new_positions);
   void updateEvenVertices(std::vector<unsigned int> & valence, std::vector<Vec3f> & new_positions);
   void reindex_subdivision(std::map<std::pair<int, int>, unsigned int> & oddVertexIndices,
std::vector<Triangle> & new_triangles);
   void dfs(std::vector<Vec3f> & new_positions, std::vector<Vec3f> & new_normals, OctreeNode *
octreeNode);
};


#include "Mesh.h"

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <limits>

using namespace std;

void Mesh::clear () {
   m_positions.clear ();
   m_normals.clear ();
   m_triangles.clear ();
}

void Mesh::loadOFF (const std::string & filename) {
   clear ();
         ifstream in (filename.c_str ());
   if (!in)
      exit (1);
         string offString;
   unsigned int sizeV, sizeT, tmp;
   in >> offString >> sizeV >> sizeT >> tmp;
   m_positions.resize (sizeV);
   m_triangles.resize (sizeT);
   for (unsigned int i = 0; i < sizeV; i++)
      in >> m_positions[i];
   int s;
   for (unsigned int i = 0; i < sizeT; i++) {
      in >> s;
      for (unsigned int j = 0; j < 3; j++)
         in >> m_triangles[i][j];
   }
```

```
    in.close ();
    centerAndScaleToUnit ();
    recomputeNormals ();
    loaded_filename = filename;
}

void Mesh::recomputeNormals () {
    m_normals.clear ();
    m_normals.resize (m_positions.size (), Vec3f (0.f, 0.f, 0.f));
    for (unsigned int i = 0; i < m_triangles.size (); i++) {
        Vec3f e01 = m_positions[m_triangles[i][1]] -  m_positions[m_triangles[i][0]];
        Vec3f e02 = m_positions[m_triangles[i][2]] -  m_positions[m_triangles[i][0]];
        Vec3f n = cross (e01, e02);
        n.normalize ();
        for (unsigned int j = 0; j < 3; j++)
            m_normals[m_triangles[i][j]] += n;
    }
    for (unsigned int i = 0; i < m_normals.size (); i++)
        m_normals[i].normalize ();
}

void Mesh::centerAndScaleToUnit () {
    Vec3f c;
    for  (unsigned int i = 0; i < m_positions.size (); i++)
        c += m_positions[i];
    c /= m_positions.size ();
    float maxD = dist (m_positions[0], c);
    for (unsigned int i = 0; i < m_positions.size (); i++){
        float m = dist (m_positions[i], c);
        if (m > maxD)
            maxD = m;
    }
    for  (unsigned int i = 0; i < m_positions.size (); i++)
        m_positions[i] = (m_positions[i] - c) / maxD;
}

void Mesh::topologicalLaplacianFilter (float laplace_alpha){
  vector<Vec3f> new_positions(m_positions.size(), Vec3f(0.0f, 0.0f, 0.0f) );
  vector<Vec3f> delta(m_positions.size(), Vec3f(0.0f, 0.0f, 0.0f) );
  vector<unsigned int> counts_neighbers(m_positions.size(), 0);
  for(unsigned int index = 0; index < m_triangles.size(); index++){
    for(unsigned int i = 0; i < 3; i++){
      unsigned int edge_index_i = m_triangles[index][i];
      unsigned int edge_index_j = (i == 2) ? m_triangles[index][0] : m_triangles[index][i + 1];
      new_positions[edge_index_j] += m_positions[edge_index_i];
      counts_neighbers[edge_index_j] += 1;
    }
  }
  for(unsigned int i = 0; i < m_positions.size(); i++) new_positions[i] /= counts_neighbers[i];
  for(unsigned int i = 0; i < m_positions.size(); i++) delta[i] = m_positions[i] - new_positions[i];
  for(unsigned int i = 0; i < m_positions.size(); i++) m_positions[i] -= laplace_alpha * delta[i];
  recomputeNormals();
}
```

```
void Mesh::reloadOFF(){
  loadOFF(loaded_filename);
}

void Mesh::geometricLaplacianFilter (float laplace_alpha){
  vector<Vec3f> new_positions(m_positions.size(), Vec3f(0.0f, 0.0f, 0.0f) );
  vector<Vec3f> delta(m_positions.size(), Vec3f(0.0f, 0.0f, 0.0f) );
  vector<float> counts_neighbers(m_positions.size(), 0.0f);
  map<pair<int, int>, float> weights;

  //To build a map for weights
  for(unsigned int index = 0; index < m_triangles.size(); index++){
    for(unsigned int i = 0; i < 3; i++){
      unsigned int edge_index_i = m_triangles[index][i];
      unsigned int edge_index_j = (i == 2) ? m_triangles[index][0] : m_triangles[index][i + 1];
      unsigned int third_vertex_index = (i == 0) ? m_triangles[index][2] : m_triangles[index][i - 1];
      Vec3f edge1 = m_positions[edge_index_i] - m_positions[third_vertex_index];
      Vec3f edge2 = m_positions[edge_index_j] - m_positions[third_vertex_index];
      weights[pair_maker(edge_index_i, edge_index_j)] += 0.5f * getCotan(edge1, edge2);
    }
  }
  //To calculate new_positions
  for(unsigned int index = 0; index < m_triangles.size(); index++){
    for(unsigned int i = 0; i < 3; i++){
      unsigned int edge_index_i = m_triangles[index][i];
      unsigned int edge_index_j = (i == 2) ? m_triangles[index][0] : m_triangles[index][i + 1];
      new_positions[edge_index_j] += weights[pair_maker(edge_index_i, edge_index_j)] *
m_positions[edge_index_i];
      counts_neighbers[edge_index_j] += weights[pair_maker(edge_index_i, edge_index_j)];
    }
  }
  for(unsigned int i = 0; i < m_positions.size(); i++) new_positions[i] /= counts_neighbers[i];
  for(unsigned int i = 0; i < m_positions.size(); i++) delta[i] = m_positions[i] - new_positions[i];
  for(unsigned int i = 0; i < m_positions.size(); i++) m_positions[i] -= laplace_alpha * delta[i];
  recomputeNormals();
}

pair<int, int> Mesh::pair_maker(int a, int b){
   if ( a < b ) return pair<int, int>(a,b);
   else return pair<int, int>(b,a);
}

float Mesh::getCotan(Vec3f v1, Vec3f v2){
  float angle = acosf(dot(v1, v2) / sqrt(length(v1) * length(v2)));
  return cos(angle) / sin(angle);
}

void Mesh::calculateMinMax(Vec3f & min_p, Vec3f & max_p){
          for(unsigned int i = 0; i < m_triangles.size(); i++){
                    if( max_p[0] < m_positions[ m_triangles[i][0] ][0] ) max_p[0] = m_positions[
m_triangles[i][0] ][0];
                    if( max_p[1] < m_positions[ m_triangles[i][0] ][1] ) max_p[1] = m_positions[
m_triangles[i][0] ][1];
```

```
                if( max_p[2] < m_positions[ m_triangles[i][0] ][2] ) max_p[2] = m_positions[
m_triangles[i][0] ][2];
                if( min_p[0] > m_positions[ m_triangles[i][0] ][0] ) min_p[0] = m_positions[
m_triangles[i][0] ][0];
                if( min_p[1] > m_positions[ m_triangles[i][0] ][1] ) min_p[1] = m_positions[
m_triangles[i][0] ][1];
                if( min_p[2] > m_positions[ m_triangles[i][0] ][2] ) min_p[2] = m_positions[
m_triangles[i][0] ][2];

                if( max_p[0] < m_positions[ m_triangles[i][1] ][0] ) max_p[0] = m_positions[
m_triangles[i][1] ][0];
                if( max_p[1] < m_positions[ m_triangles[i][1] ][1] ) max_p[1] = m_positions[
m_triangles[i][1] ][1];
                if( max_p[2] < m_positions[ m_triangles[i][1] ][2] ) max_p[2] = m_positions[
m_triangles[i][1] ][2];
                if( min_p[0] > m_positions[ m_triangles[i][1] ][0] ) min_p[0] = m_positions[
m_triangles[i][1] ][0];
                if( min_p[1] > m_positions[ m_triangles[i][1] ][1] ) min_p[1] = m_positions[
m_triangles[i][1] ][1];
                if( min_p[2] > m_positions[ m_triangles[i][1] ][2] ) min_p[2] = m_positions[
m_triangles[i][1] ][2];

                if( max_p[0] < m_positions[ m_triangles[i][2] ][0] ) max_p[0] = m_positions[
m_triangles[i][2] ][0];
                if( max_p[1] < m_positions[ m_triangles[i][2] ][1] ) max_p[1] = m_positions[
m_triangles[i][2] ][1];
                if( max_p[2] < m_positions[ m_triangles[i][2] ][2] ) max_p[2] = m_positions[
m_triangles[i][2] ][2];
                if( min_p[0] > m_positions[ m_triangles[i][2] ][0] ) min_p[0] = m_positions[
m_triangles[i][2] ][0];
                if( min_p[1] > m_positions[ m_triangles[i][2] ][1] ) min_p[1] = m_positions[
m_triangles[i][2] ][1];
                if( min_p[2] > m_positions[ m_triangles[i][2] ][2] ) min_p[2] = m_positions[
m_triangles[i][2] ][2];
        }
}

void Mesh::make_cubes(float & x_scale, float & y_scale, float & z_scale, unsigned int resolution, Vec3f &
min_p){
  float max_float = numeric_limits<float>::max();
        float min_float =  - ( numeric_limits<float>::max() - 1);
        Vec3f max_p = Vec3f(min_float, min_float, min_float);
        min_p = Vec3f(max_float, max_float, max_float);
        calculateMinMax(min_p, max_p);
  min_p -= (max_p - min_p) * 0.05f;
  max_p += (max_p - min_p) * 0.05f;
  x_scale = (max_p[0] - min_p[0]) / (float) resolution;
  y_scale = (max_p[1] - min_p[1]) / (float) resolution;
  z_scale = (max_p[2] - min_p[2]) / (float) resolution;
}

void Mesh::push_vertices_into_cubes(UGrid & grid, unsigned int resolution, float x_scale, float y_scale,
float z_scale, vector<unsigned int> & index_map, Vec3f & min_p){
  for(unsigned int index = 0; index < m_triangles.size(); index++){
```

```
   for(unsigned int i = 0; i < 3; i++){
     Vec3f vertex_position = m_positions[ m_triangles[index][i] ];
     unsigned int x = (vertex_position[0] - min_p[0]) / x_scale;
     unsigned int y = (vertex_position[1] - min_p[1]) / y_scale;
     unsigned int z = (vertex_position[2] - min_p[2]) / z_scale;
     grid.getCell(x, y, z).addNewVertex(m_triangles[index][i], vertex_position, m_normals[
m_triangles[index][i] ]);
     index_map[m_triangles[index][i]] = z + y * resolution + x * resolution * resolution;
   }
  }
}

void Mesh::calculate_new_positions(vector<Vec3f> & new_positions, vector<Vec3f> & new_normals,
UGrid & grid){
  for(unsigned int x = 0; x < grid.nx; x++){
    for(unsigned int y = 0; y < grid.ny; y++){
      for(unsigned int z = 0; z < grid.nz; z++){
        new_positions.push_back(grid.getCell(x, y, z).meanPosition);
        new_normals.push_back(grid.getCell(x, y, z).meanNormal);
      }
    }
  }
}

void Mesh::reindex(const vector<unsigned int> & index_map, vector<Triangle> & new_triangles){
  for(unsigned int index = 0; index < m_triangles.size(); index++){
    if(index_map[m_triangles[index][0]] != index_map[m_triangles[index][1]] &&
      index_map[m_triangles[index][1]] != index_map[m_triangles[index][2]] &&
      index_map[m_triangles[index][0]] != index_map[m_triangles[index][2]] ){
      new_triangles.push_back(Triangle(index_map[m_triangles[index][0]],
index_map[m_triangles[index][1]], index_map[m_triangles[index][2]] ) );
    }
  }
}

void Mesh::simplify(unsigned int resolution){
  vector<Vec3f> new_positions;
  vector<Triangle> new_triangles;
  vector<Vec3f> new_normals;
  vector<unsigned int> index_map(3 * m_triangles.size());
  UGrid grid(resolution, resolution, resolution);
  Vec3f min_p;
        float x_scale, y_scale, z_scale;
  make_cubes(x_scale, y_scale, z_scale, resolution, min_p);
  push_vertices_into_cubes(grid, resolution, x_scale, y_scale, z_scale, index_map, min_p);
  calculate_new_positions(new_positions, new_normals, grid);
  for(unsigned int i = 0; i < new_normals.size(); i++) new_normals[i].normalize();
  reindex(index_map, new_triangles);

  new_positions.swap(m_positions);
  new_triangles.swap(m_triangles);
  new_normals.swap(m_normals);
}
```

```
void Mesh::simplifyAdaptiveMesh (unsigned int n){
  vector<unsigned int> ind(m_positions.size());
  for(unsigned int k = 0; k < m_positions.size(); k++) ind[k] = k;
  float max_float = numeric_limits<float>::max();
  float min_float =  - ( numeric_limits<float>::max() - 1);
  Vec3f max_point = Vec3f(min_float, min_float, min_float);
  Vec3f min_point = Vec3f(max_float, max_float, max_float);
  calculateMinMax(min_point, max_point);
  OctreeNode * octreeRoot = OctreeNode::buildOctree(min_point, max_point, ind, m_positions, n);
  std::cout << "Octree has been built successfully with " << n << " as max number of vertices per leaf." <<
'\n';

  std::cout << "But the use of octree has not yet been implemented." << '\n' << '\n';
/*
  vector<Vec3f> new_positions;
  vector<Triangle> new_triangles;
  vector<Vec3f> new_normals;
  vector<unsigned int> index_map(3 * m_triangles.size());

  dfs(new_positions, new_normals, octreeRoot);
  for(unsigned int i = 0; i < new_normals.size(); i++) new_normals[i].normalize();
  reindex(index_map, new_triangles);

  new_positions.swap(m_positions);
  new_triangles.swap(m_triangles);
  new_normals.swap(m_normals);*/
}

void Mesh::dfs(vector<Vec3f> & new_positions, vector<Vec3f> & new_normals, OctreeNode *
octreeNode){
  if(octreeNode->isLeaf()){
    vector<unsigned int> indices = octreeNode->getIndices();
    Vec3f meanPosition = Vec3f(0.0f, 0.0f, 0.0f);
    Vec3f meanNormal = Vec3f(0.0f, 0.0f, 0.0f);
    unsigned int count = 0;
    for(unsigned int i = 0; i < indices.size(); i++){
      meanPosition += m_positions[indices[i]];
      meanNormal += m_normals[indices[i]];
      count++;
    }
    meanPosition /= count;
    meanNormal /= count;
    new_positions.push_back(meanPosition);
    new_normals.push_back(meanNormal);
  }else{
    for(unsigned int i = 0; i < 8; i++){
      dfs(new_positions, new_normals, octreeNode->getChildren()[i]);
    }
  }
}

void Mesh::addOddVertices(map<pair<int, int>, Vec3f> & oddVertices, map<pair<int, int>, unsigned int>
& oddVertexIndices, vector<Vec3f> & new_positions){
  for(unsigned int index = 0; index < m_triangles.size(); index++){
```

```
   for(unsigned int i = 0; i < 3; i++){
     unsigned int edge_index_i = m_triangles[index][i];
     unsigned int edge_index_j = (i == 2) ? m_triangles[index][0] : m_triangles[index][i + 1];
     unsigned int third_vertex_index = (i == 0) ? m_triangles[index][2] : m_triangles[index][i - 1];
     oddVertices[pair_maker(edge_index_i, edge_index_j)] += (1.0f / 8.0f) *
m_positions[third_vertex_index] + (3.0f / 16.0f) * (m_positions[edge_index_i] +
m_positions[edge_index_j]);
   }
  }
  for(auto it : oddVertices){
   oddVertexIndices[it.first] = new_positions.size();
   new_positions.push_back(it.second);
  }
}

void Mesh::updateEvenVertices(vector<unsigned int> & valence, vector<Vec3f> & new_positions){
  for(unsigned int index = 0; index < m_triangles.size(); index++){
   for(unsigned int i = 0; i < 3; i++){
     unsigned int edge_index_j = (i == 2) ? m_triangles[index][0] : m_triangles[index][i + 1];
     valence[edge_index_j] += 1;
   }
  }
  for(unsigned int index = 0; index < m_triangles.size(); index++){
   for(unsigned int i = 0; i < 3; i++){
     unsigned int edge_index_j = (i == 2) ? m_triangles[index][0] : m_triangles[index][i + 1];
     unsigned int n = valence[edge_index_j];
     if( length(new_positions[edge_index_j] - m_positions[edge_index_j]) <
length(m_positions[edge_index_j]) * 0.01f ) new_positions[edge_index_j] *= (1.0f - getAlpha(n));
   }
  }
  for(unsigned int index = 0; index < m_triangles.size(); index++){
   for(unsigned int i = 0; i < 3; i++){
     unsigned int edge_index_i = m_triangles[index][i];
     unsigned int edge_index_j = (i == 2) ? m_triangles[index][0] : m_triangles[index][i + 1];
     unsigned int n = valence[edge_index_j];
     new_positions[edge_index_j] += m_positions[edge_index_i] * (getAlpha(n) / (float) n);
   }
  }
}

void Mesh::reindex_subdivision(map<pair<int, int>, unsigned int> & oddVertexIndices, vector<Triangle>
& new_triangles){
  for(unsigned int i = 0; i < m_triangles.size(); i++){
   unsigned int index0 = m_triangles[i][0];
   unsigned int index1 = m_triangles[i][1];
   unsigned int index2 = m_triangles[i][2];
   unsigned int odd0 = oddVertexIndices[pair_maker(index0, index1)];
   unsigned int odd1 = oddVertexIndices[pair_maker(index1, index2)];
   unsigned int odd2 = oddVertexIndices[pair_maker(index2, index0)];
   new_triangles.push_back(Triangle(index0, odd0, odd2));
   new_triangles.push_back(Triangle(odd0, index1, odd1));
   new_triangles.push_back(Triangle(odd2, odd1, index2));
   new_triangles.push_back(Triangle(odd0, odd1, odd2));
  }
```

```
}

void Mesh::subdivide(){
  map<pair<int, int>, Vec3f> oddVertices;
  map<pair<int, int>, unsigned int> oddVertexIndices;
  vector<Vec3f> new_positions;
  vector<Triangle> new_triangles;
  vector<unsigned int> valence(m_positions.size(), 0);

  new_positions = m_positions;

  addOddVertices(oddVertices, oddVertexIndices, new_positions);
  updateEvenVertices(valence, new_positions);
  reindex_subdivision(oddVertexIndices, new_triangles);

  new_positions.swap(m_positions);
  new_triangles.swap(m_triangles);
  recomputeNormals();
}

#include "Vec3.h"
#include "Mesh.h"
#include "BVH.h"
#include <limits>

class Ray{
  private:
    Vec3f origin;
    Vec3f direction;
    float radius;

    bool isIntersectedWithTriangle(const Vec3f & v0, const Vec3f & v1, const Vec3f & v2) const;
    bool isIntersectedWithCube(const Vec3f & min_p, const Vec3f & max_p) const;
    void isIntersected(const Mesh & mesh, BVH * const bvh, int & result) const;

  public:
    Ray(){}
    Ray(Vec3f origin, Vec3f direction_): origin(origin){
      this->radius = length(direction_);
      direction_.normalize();
      this->direction = direction_;
    }
    ~Ray(){}
    bool isIntersected(const Mesh & mesh) const;
    bool isIntersected(const Mesh & mesh, BVH * const bvh) const;
};

#include <cmath>
#include <vector>
#include "Triangle.h"
#include "Ray.h"

using namespace std;
```

```
bool Ray::isIntersected(const Mesh& mesh) const {
  float epsilon = 0.0001;
  bool intersected = false;

  vector<Triangle> triangle_indices = mesh.triangles ();
  vector<Vec3f> vertex_list = mesh.positions ();

  for(unsigned int i = 0; i < triangle_indices.size(); i++){
    Triangle threeIndices = triangle_indices[i];
    Vec3f v0 = vertex_list[threeIndices[0]];
    Vec3f v1 = vertex_list[threeIndices[1]];
    Vec3f v2 = vertex_list[threeIndices[2]];

    Vec3f v0v1 = v1 - v0;
    Vec3f v0v2 = v2 - v0;
    Vec3f pvec = cross(direction, v0v2);
    float det = dot(v0v1, pvec);

    // ray and triangle are parallel if det is close to 0
    if (fabs(det) < epsilon) continue;

    float invDet = 1.0 / det;

    Vec3f tvec = origin - v0;
    float u = dot(tvec, pvec) * invDet;
    if (u < 0.0 || u > 1.0) continue;

    Vec3f qvec = cross(tvec, v0v1);
    float v = dot(direction, qvec) * invDet;
    if (v < 0.0 || u + v > 1.0) continue;

    // positive t means intersection is in front of origin point,
    // negative t means intersection is behind the origin point.
    // The absolute value of t means the distance between intersection point and origin point.
    float t = dot(v0v2, qvec) * invDet;

    if(t > 0.0 && t <= radius){
      intersected = true;
      break;
    }
  }
  return intersected;
}

bool Ray::isIntersectedWithTriangle(const Vec3f & v0, const Vec3f & v1, const Vec3f & v2) const{
  float epsilon = 0.0001;
  Vec3f v0v1 = v1 - v0;
  Vec3f v0v2 = v2 - v0;
  Vec3f pvec = cross(direction, v0v2);
  float det = dot(v0v1, pvec);
  // ray and triangle are parallel if det is close to 0
  if (fabs(det) < epsilon) return false;
  float invDet = 1.0 / det;
  Vec3f tvec = origin - v0;
```

```
  float u = dot(tvec, pvec) * invDet;
  if (u < 0.0 || u > 1.0) return false;
  Vec3f qvec = cross(tvec, v0v1);
  float v = dot(direction, qvec) * invDet;
  if (v < 0.0 || u + v > 1.0) return false;
  float t = dot(v0v2, qvec) * invDet;
  if(t > 0.0 && t <= radius) return true;
  else return false;
}

bool Ray::isIntersectedWithCube(const Vec3f & min_p, const Vec3f & max_p) const {
  Vec3f dirInv = Vec3f(1.0f / direction[0], 1.0f / direction[1], 1.0f / direction[2]);
  float t1 = (min_p[0] - origin[0]) * dirInv[0];
  float t2 = (max_p[0] - origin[0]) * dirInv[0];
  float t3 = (min_p[1] - origin[1]) * dirInv[1];
  float t4 = (max_p[1] - origin[1]) * dirInv[1];
  float t5 = (min_p[2] - origin[2]) * dirInv[2];
  float t6 = (max_p[2] - origin[2]) * dirInv[2];
  float tmin = max( max( min(t1, t2), min(t3, t4) ), min(t5, t6) );
  float tmax = min( min( max(t1, t2), max(t3, t4) ), max(t5, t6) );

  // if tmax < 0, ray is intersecting AABB, but the whole AABB is behind the origin
  if (tmax < 0) return false;

  // if tmin > tmax, ray doesn't intersect AABB
  if (tmin > tmax) return false;
  else return true;
}

void Ray::isIntersected(const Mesh & mesh, BVH * const bvh, int & result) const {
  Vec3f min_p = ( bvh->getAABB() ).getMinPoint();
  Vec3f max_p = ( bvh->getAABB() ).getMaxPoint();
  if( this->isIntersectedWithCube(min_p, max_p) ){
    if( bvh->isALeaf() ){
      Triangle threeIndices = bvh->getTriangle();
      Vec3f v0 = mesh.positions ()[threeIndices[0]];
      Vec3f v1 = mesh.positions ()[threeIndices[1]];
      Vec3f v2 = mesh.positions ()[threeIndices[2]];
      if( this->isIntersectedWithTriangle(v0, v1, v2) ){
        result *= 0;
      }
    }else{
      if( bvh->getLeftChild() != nullptr ) isIntersected( mesh, bvh->getLeftChild(), result );
      if( bvh->getRightChild() != nullptr ) isIntersected( mesh, bvh->getRightChild(), result );
    }
  }
}

bool Ray::isIntersected(const Mesh & mesh, BVH * const bvh) const {
  int result = 1;
  this->isIntersected(mesh, bvh, result);
  if( result == 1 ) return false;
  else return true;
}
```