Технічні науки

УДК 004.946

Марков Дмитро Костянтинович

студент

Інституту прикладного системного аналізу

Національного технічного університету України

«Київський політехнічний інститут імені Ігоря Сікорського»

Марков Дмитрий Константинович

студент

Института прикладного системного анализа

Национального технического университета Украины

«Киевский политехнический институт имени Игоря Сикорского»

Markov Dmitriy

Student of the

Institute for Applied System Analysis of the

National Technical University of Ukraine

«Igor Sikorsky Kyiv Polytechnic Institute»

ПОРІВНЯЛЬНИЙ АНАЛІЗ АЛГОРИТМІВ ОПТИМІЗАЦІЇ РЕНДЕРИНГУ В СУЧАСНИХ КОМП'ЮТЕРНИХ ІГРАХ
СРАВНИТЕЛЬНЫЙ АНАЛИЗ АЛГОРИТМОВ ОПТИМИЗАЦИИ РЕНДЕРИНГА В СОВРЕМЕННЫХ КОМПЬЮТЕРНЫХ ИГРАХ
COMPARISON OF RENDER OPTIMIZATION ALGORITHMS IN MODERN COMPUTER GAMES

Анотація. *В даній роботі було розглянуто різні алгоритми оклюзивного виключення, буде проведено розбір кожного з них, його історію, необхідність у ньому, математичну і логічні основи алгоритму.*

Також було розроблено власний рендер двигун, в якому дані алгоритми були реалізовані та протестовані на декількох тестових сценах.

Результатом роботи буде порівняльна характеристика розглянутих алгоритмів, визначення рівня ефективності та кількості ресурсів, що потребує кожен алгоритм. Буде визначено сильні та слабкі сторони кожного с алгоритмів і зроблено висновки на основі отриманих даних. Ця характеристика дасть можливість використовувати сильні сторони алгоритмів та захищати іншими алгоритмами їх слабкі місця.

Ключові слова: *оклюзивне виключення, рендер, порівняння алгоритмів, реалізація алгоритмів, оптимізації рендеру.*

Аннотация. *В данной работе будут рассмотрены разные алгоритмы оклюзивного отсечения, будет проведено разбор каждого из них, их историю, необходимость возникновения, математическую и логическую основу алгоритмов. Также был разработан собственный рендер движок, в котором данные алгоритмы были реализованы и протестированы на нескольких тестовых сценах.*

Результатом работы будет сравнительная характеристика рассмотренных алгоритмов, определение уровня эффективности и количества ресурсов, которые необходимы каждому алгоритму. Будет определено сильные и слабые стороны каждого из алгоритмов и сделано выводы на основе полученных данных. Эта характеристика даст возможность использовать сильные стороны алгоритмов и защищать другими алгоритмами их слабые места.

Ключевые слова: *оклюзивное отсечение, рендер, сравнение алгоритмов, реализация алгоритмов, оптимизации рендера.*

Summary. *In this paper different occlusion culling algorithms will be considered. Also will be performed analysis of each of them, their history, the need arising, mathematical and logical basis for algorithms. There was also*

developed rendering engine in which these algorithms have been implemented and tested on several test scenes.

The result of the paper is comparative characteristic of algorithms to determine the level of efficiency and the number of resources required for each algorithm. Strengths and weaknesses of each of the algorithms will be determined and will be made conclusions based on the data obtained. This feature will make it possible to use the strengths of the algorithms and protect their weaknesses by other algorithms.

Key words: *occlusion culling, render, algorithms comparison, algorithms implementation, render optimization.*

Постановка проблеми. Рівень якості графічної складової в сучасних комп'ютерних іграх постійно зростає і для того, щоб було можливо відповідати цим стандартам потрібно знаходити можливості для оптимізації.

Аналіз останніх досліджень і публікацій. Дослідження складають праці таких компаній, як Umbra Software [1; 2] та Computer Graphics Laboratory (Zurich)[4].

Формулювання цілей статті (постановка завдання). Аналіз існуючих алгоритмів оклюзивного відсічення та їх порівняльний аналіз використовую різні тестові сцени та набори об'єктів.

Виклад основного матеріалу. Типова сцена в комп'ютерній грі або в анімаційному фільмі складається с сотень об'єктів. Кожен з цих об'єктів має свій меш, тобто набір полігонів, які формують модель об'єкту. Не рідко виникають випадки, коли меш складається з десятків, якщо не сотень, тисяч конвексів. Тому, якщо неправильно управляти рендером цих об'єктів, то це може дуже сильно вплинути на продуктивність та фрейм-рейт (FPS). Особливу увагу технологіям оптимізації рендерингу приділяють при створенні анімаційних фільмів та в комп'ютерних іграх.

Там це більш важливо з різних причин. Процес створення анімаційних фільмів виглядає так: спочатку створюється надзвичайно складні за своєю структурою моделі об'єктів, котрі складаються с мільйонів полігонів та сотень костей, потім с цих моделей складається сцена, потім вони анімуються, потім рендерять частину фільму. Процес рендерингу маленької частинки фільму може займати дні або навіть неділі. Саме тому в таких складних сценах потрібно відсікати невидимі частини. В грі є інша проблема, якщо в фільмах показують попередньо відрендерену картинку, то тут потрібно підмальовувати сцену в режимі реального часу, а часу дуже мало, зазвичай нормальною планкою швидкості ставлять 60 кадрів за секунду, отже часу на один кадр 1/60, що дуже мало [1].

У комп'ютерній 3D графіці, визначення прихованої поверхні (також відоме як видалення прихованих поверхонь (HSR), оклюзивне виключення (OC) або видимого визначення поверхні (VSD)) це процес, який використовується для визначення того, які поверхні і частини поверхні не видно з певної точки зору. Алгоритм визначення прихованих поверхонь є вирішення проблеми видимості, яка була однією з перших серйозних проблем в області комп'ютерної 3D графіки. Процес визначення прихованої поверхні іноді називають приховуванням, і такий алгоритм іноді називають приховувач (англ. hider). Аналог для візуалізації лінії приховане видалення ліній (англ. hidden line removal, HLR).

Приховане визначення поверхні являє собою процес, при якому поверхні, які не повинні бути видні користувачеві (наприклад, тому що вони знаходяться за непрозорими об'єктами, такими як стіни) не рендеряться. Незважаючи на прогрес в області апаратного забезпечення все ще існує потреба в поліпшених алгоритмах візуалізації. Відповідальність рендер двигуну в тому, щоб дозволити створювати великі віртуальні простори і, оскільки розмір віртуального світу наближається до нескінченності двигун не повинен сповільнитися та має працювати на

постійній швидкості. Оптимізація цього процесу полягає в тому, щоб змінити віртуальну сцену, об'єкти на ній та їх складність так, щоб це не було помітно користувачеві.

Є багато методів для визначення прихованої поверхні [2]. Вони майже всі базуються на сортуванні, і зазвичай змінюються в порядку, в якому виконується сортування і як поділяється проблема. Сортування великої кількості графічних примітивів зазвичай робиться за допомогою парадигми «розділяй і володарюй».

В даній роботі будуть розглянуть різні алгоритми оклюзивного виключення, буде проведено розбір кожного з них, його історію, необхідність у ньому, математичну і логічні основи алгоритму [3]. Також було розроблено власний рендер двигун, в якому дані алгоритми були реалізовані та протестовані на декількох тестових сценах [4].

Результатом роботи буде порівняльна характеристика розглянутих алгоритмів, визначення рівня ефективності та кількості ресурсів, що потребує кожен алгоритм. Буде визначено сильні та слабкі сторони кожного с алгоритмів і зроблено висновки на основі отриманих даних. Ця характеристика дасть можливість використовувати сильні сторони алгоритмів та захищати іншими алгоритмами їх слабкі місця [5].

Огляд досліджуваних алгоритмів

Виключення за областю зору (Frustum Culling)

Опис алгоритму

Найпростіший й найбільш ефективний алгоритм виключення - View Frustum Culling (VFC) або просто Frustum Culling - досить універсальний і швидкий. В черзі рендеру його необхідно застосовувати якомога раніше, тому що при незначних витратах процесорного часу може вдасться відкинути значну частину невидимої геометрії і не виконувати її подальшу обробку [7].

Для початку - невеликі технічні подробиці. Для коректного проектування полігонів на екран до всіх вершин застосовується проекційна

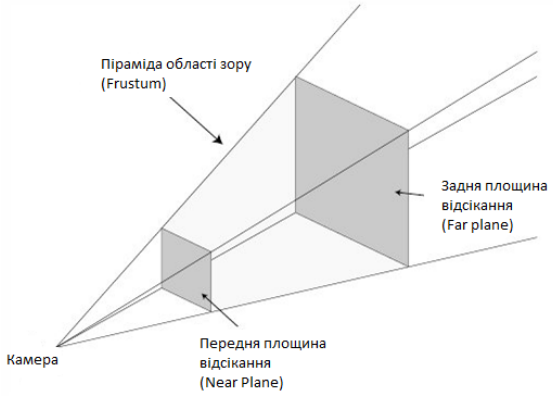

Рис. 1. Піраміда зору (View Frustum) [17]

матриця (в разі перспективної проекції вона будується виходячи з кута огляду камери (FOV), дальність видимості і ще деяких параметрів і задає перехід від тривимірного простору сцени в простір камери) і виконується перспективний розподіл, в результаті чого координати вершин видимої частини сцени лежать всередині одиничного куба (фактично, це екранні координати вершини і глибини). Якщо після множення на матрицю вершина виявилася поза екраном (тобто поза одиничного куба), її можна не малювати. Таким чином, для всіх вершин в просторі сцени обмежує обсягом була зрізана піраміда видимості (View Frustum, Frustum), а в просторі камери frustum перетворюється в одиничний куб [1].

Це можна пояснити трохи простіше. Немає необхідності малювати об'єкти, що знаходяться позаду або збоку від камери і не потрапляють в поле зору. Рендерити потрібно тільки ті об'єкти, які знаходяться в полі видимості. Цей обсяг і є усіченої пірамідою, все що знаходиться поза пірамідою знаходиться і поза екраном.

Суть алгоритму в наступному. Виходячи зі сказаного вище, для визначення видимості вершини необхідно перевірити, чи знаходиться обробляється вершина всередині усіченої піраміди, або помножити вершини на матрицю проектування і перевірити, чи знаходиться вершина

всередині одиничного куба. На практиці зручно використовувати перший варіант: знайти рівняння всіх площин піраміди (це можна зробити з матриці) і перевірити, чи знаходиться точка перед цими площинами [8].

Незважаючи на те, що повертексна перевірка і відсікання здійснюється на рівні графічного API (як правило, апаратно), на більш високому рівні програміст може використовувати знання про структуру сцени для перевірок і відсікання великих груп вершин, розташованих локально. Для цього, як правило, на видимість перевіряються тільки нескладні геометричні фігури, що складаються з декількох вершин, і, якщо вся фігура виявляється поза видимості, то частина сцени, укладена в неї, може бути відкинута. У ролі таких фігур (обмежують обсягів) можуть виступати сфера, паралелепіпеди (AABB) і т.д. Таким чином, нескладно організувати відсічення компактних моделей (в тому числі динамічних). При роботі з великими приміщеннями або ландшафтами сцену необхідно організувати ієрархічно, тому як перевірка кожної вершини або кожного трикутника неефективна. У разі великих рівнів, розбитих ієрархічно на вузли, для кожного вузла будується обмежуюча фігура (зазвичай або куля або паралелепіпед - AABB), який і перевіряється на перетин з Frustum. Якщо вузол знаходиться цілком у Frustum, то і все його нащадки теж, що скорочує кількість перевірок (іноді для нащадків досить перевіряти не всі площини відсікання).

Реалізація алгоритму

Пряма реалізація виглядає так, шість сторін піраміди видимого простору називаються площинами відсікання (clipping planes). Для простоти думайте про площині як про нескінченно великі аркуші паперу (з лицьової та зворотної сторони). Площина визначається чотирма числами, які зазвичай називають A, B, C і D. Ці чотири числа задають орієнтацію площини в просторі.

Замість того, щоб ставити значення нормалі площини як X, Y, Z, використовуються змінні A, B і C. Потрібно ще одне додаткове значення для завдання відстані від площини до початку координат. Це відстань представляється як D. Описуючи площину встановлюються змінні A, B і C

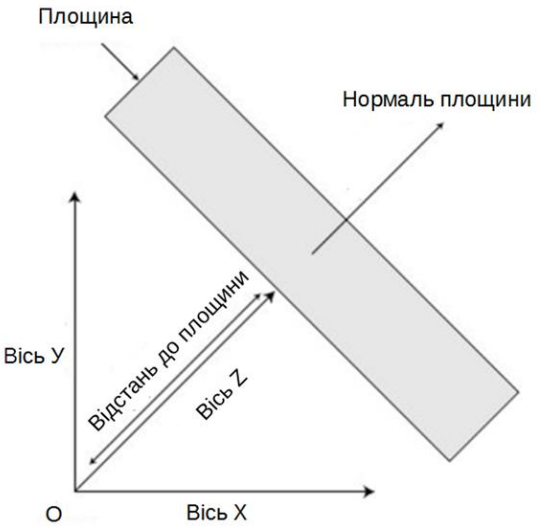

Рис. 2. Перевірка площини

як значення нормалі площини, а в D - відстань від площини до початку координат. Як тільки нормаль і відстань задані, можна використовувати площину для того, щоб перевірити, чи знаходиться зазначена точка перед площиною або за нею.

Для обчислення шести площин піраміди видимого простору ви комбінуєте поточну матрицю перетворення виду і матрицю проекції. Потім ви маєте справу безпосередньо з комбінованої матрицею для обчислення значень A, B, C і D кожної площині.

Нижче представлений псевдокод, в якому комбінуються дві необхідні матриці і на їх підставі обчислюються значення площин (значення площин розміщуються у відповідні об'єкти PLANE):

```
// Graphics = раніше ініціалізовний cGraphics
PLANE Planes [6];
// Робочі матриці
MATRIX Matrix, matView, matProj;

// Отримуємо матриці виду і проекції та комбінуємо їх
```

```
Graphics.GetDeviceCOM()->GetTransform(PROJECTION, &matProj);
Graphics.GetDeviceCOM()->GetTransform(VIEW, &matView);
MatrixMultiply(&Matrix, &matView, &matProj);

// Конструюємо площини
// Передня площина + аналогічно інші площини
Planes [0].a = Matrix._14 + Matrix._13;
Planes [0].b = Matrix._24 + Matrix._23;
Planes [0].c = Matrix._34 + Matrix._33;
Planes [0].d = Matrix._44 + Matrix._43;
PlaneNormalize(&Planes [0], &Planes [0]);
```

Тепер маємо площину (або набір площин), орієнтовану в заданому напрямку. Щоб перевірити, чи знаходиться точка перед площиною або за нею, ви обчислюєте скалярний добуток (dot product). Скалярний добуток - це спеціальна операція з векторами (координатами), зазвичай застосовується для обчислення кута між двома векторами.

При перевірці місця розташування точки відносно площини скалярний добуток повідомляє вам відстань від точки до площини. Якщо значення більше нуля, точка знаходиться перед площиною. Якщо значення менше нуля - точка знаходиться за площиною.

Для обчислення скалярного добутку використовується функція PlaneDotCoord:

```
FLOAT PlaneDotCoord(CONST PLANE *pP, CONST VECTOR3 *pV);
```

Функції D3DXPlaneDotCoord надається структура площині (що містить значення площині) і точка (вектор, що міститься в об'єкті VECTOR3). Перевірка визначить з якого боку площини знаходиться точка. Після повернення з функції PlaneDotCoord ви отримуєте відстань від точки до площини. Це значення може бути рівним нулю, позитивним або негативним.

Якщо повернене значення дорівнює 0, точка лежить на площині. Якщо значення менше 0, то точка знаходиться за площиною; якщо більше - перед нею. Ось приклад перевірки:

```
// Plane = раніше створений об'єкт PLANE
// XPos, YPos, ZPos = точка для перевірки
```

```
float Dist = PlaneDotCoord(&Plane,
&VECTOR3(XPos, YPos, ZPos));

// Якщо dist > 0 точка перед площиною
// Якщо dist < 0 точка за площиною
// Якщо dist == 0 точка на площині
```

Хоча перевірка окремої точки корисна, можна піти далі і перевіряти цілі об'єкти на знаходження всередині піраміди видимого простору. Об'єктами, що перевіряються можуть бути куби, паралелепіпеди і сфери.

Для кубів і паралелепіпедів ви перевіряєте кутові вершини. Якщо всі вершини знаходяться за площиною, значить куб або паралелепіпед поза пірамідою видимого простору (і поза полем зору). Якщо хоч одна вершина всередині піраміди, або перед будь-площиною (не всі вершини розташовані позаду одній площині), означає, що куб або паралелепіпед видимі. Що стосується сфери, то щоб вона була видима, відстань від кожної площині до центру сфери має дорівнювати або бути більше радіусу.

Щоб перевірити будь-яку кількість точок, перевіряється кожну з них індивідуально, переконуючись, що хоча б одна перебуває перед усіма площинами. Якщо жодна із точок не заходиться всередині фрустуму, то загалом це означає, що об'єкт не видно, проте є одна ситуація коли це не так. Так може трапитись коли фрустум повністю пересікається одним із полігонів об'єкту, тобто вийде так, що жодна із точок не буде всередині фрустуму, проте, об'єкт повинен буду відрендерений. Для цього зроблена окрема перевірка.

Буфер глибини (Z-buffer)

Опис алгоритму

У комп'ютерній графіці, Z-буферизація, також відома як глибинна буферизація, є управління глибинна координат зображення в 3D-графіці, зазвичай робиться на апаратному рівні, іноді в програмному забезпеченні. Це одне з вирішень полягає видимості, що полягає в визначенні того, які

елементи сцени видно, а які приховані. Алгоритм художника є ще одним поширеним з вирішень, яке, хоча і менш ефективно, також може обробляти не-непрозорі елементи сцени.

Коли об'єкт візуалізується, глибина генерованого пікселю (Z-координата) зберігається в буфері (Z-буфер або буфер глибини). Цей буфер зазвичай виконаний у вигляді двомірного масиву (x-y) з одним елементом для кожного пікселя екрану. Якщо інший об'єкт сцени повинен бути представлений в тому ж пікселі, метод порівнює дві глибини і перекриває поточний піксель, якщо об'єкт знаходиться ближче до спостерігача. Обрана глибина потім зберігається в Z-буфері, замінюючи стару. Зрештою, Z -буфер дозволить правильно відтворити звичайне сприйняття глибини: близький об'єкт приховує той що далі. Це називається відсічення за глибиною або z-culling.

Зернистість z-буфера має великий вплив на якість сцени: 16-бітний Z-буфер може привести до артефактів (так звані «Z-бої»), коли два об'єкти перебувають дуже близько один до одного. 24-бітний або 32-бітний Z-буфер поводиться набагато краще, хоча ця проблема не може бути повністю усунена без додаткових алгоритмів. 8-бітний Z-буфер практично не використовується, так як він має занадто мало точності.

Z-буфер це технологія, яка використовується практично у всіх сучасних комп'ютерів, ноутбуків і мобільних телефонів для виконання 3D-графіки, наприклад, для комп'ютерних ігор. Z-буфер реалізований у вигляді апаратних засобів (на відеокарті, GPU) в межах цих комп'ютерів. Z-буфер також використовується (реалізований у вигляді програмного забезпечення, на відміну від апаратних засобів) для виробництва комп'ютерних спецефектів для фільмів.

Крім того, дані Z-буфер, отриманий від рендеру поверхні з використанням точкового світла дозволить використовувати техніку карт тіней (shadow mapping technique).

Навіть при досить малій зернистості, проблеми з якістю можуть виникнути, коли точність значень відстані Z-буферу не поширюється рівномірно на відстані. Ближче значення набагато точніші (і, отже, може відображати ближчі об'єкти краще), значення, які знаходяться далі. Як правило, це бажано, але іноді це прищведе до артефактів на віддалених об'єктах. Варіантом Z-буферизації, що призводить до більш рівномірно розподіленим точності називається W-буферизація [9].

На початку нової сцени, Z-буфер повинен бути очищений до певного значення, як правило, 1, так як це значення є верхньою межею (за шкалою від 0 до 1) від глибини, а це означає, що жоден об'єкт не присутній в піраміді зору [10].

Винахід концепції Z-буфера найчастіше приписується Едвіну Кетмулу (Edwin Catmull), хоча Вольфганг Штрассер (Wolfgang Straßer) також описав цю ідею у своїй дисертації в 1974.

На останніх графічних картах, управління Z-буфера використовує суттєвий шмат доступної пропускної здатності пам'яті. Різні методи були використані для зниження вартості продуктивності Z-буферизації, такі як стиснення без втрат (комп'ютерні ресурси для стиснення / розпакування дешевше, ніж смуга пропускання) і надшвидкої очистки Z-буферу, що робить застарілим "один кадр позитивний, один кадр негативний" трюк (пропуск між кадрової очистки, використовуючи позитивні та негативні величини для позначення сусідніх кадрів).

У рендері, Z-виключенням є раннє усунення пікселя на основі глибини, метод, який забезпечує збільшення продуктивності шляхом виключення прихованих поверхонь, рендер яких був би дорогим. Це є прямим наслідком Z-буферизації, де глибина кожного кандидата пікселя порівнюється з глибиною існуючої геометрії, за якою може бути прихована.

При використанні Z-буферу, піксель може бути виключений, як тільки його глибина відома, що дозволяє пропускати весь процес освітлення і текстурування піксель, який не буде видно в будь-якому випадку. Крім того, піксельні шейдери, котрі віднімають багато часу, як правило, не будуть використані для виключених пікселів. Це робить Z-виключення хорошим кандидатом оптимізації в ситуаціях, коли рейт заповнення, освітлення, текстурування або піксельні шейдера - це основні вузькі місця [11].

У той час як Z-буферизація дозволяє геометрії бути несортованою, сортування полігонів за рахунок збільшення глибини (таким чином, використовуючи реверсивний алгоритм художника) дозволяє кожному пікселю бути відрендерено меншу кількість разів. Це може збільшити рейт заповнення сцен з великим овердрафтом, але якщо вони не поєднанні з Z-буфером сцена страждає від серйозних проблем, таких як:

- багатокутники можуть змикатися один з одним в циклі (наприклад: трикутнику оклюзована B & B оклюзована C, C оклюзована A);
- не існує ніякого канонічного "найближчої" точки на трикутнику (наприклад: незалежно від того, сортуються трикутники відносно їх центроїда або відносно найближчої точки або найбільш віддаленої точки, ніхто не може знайти два трикутника A і B такі, що A "ближче", але в дійсності B слід проводити в першу чергу).

Таким чином, реверсивний алгоритм художника не може бути використаний в якості альтернативи Z-відсікання (без серйозно реінжинірингу), за винятком того, як оптимізації до Z-відсікання. Наприклад, оптимізація може бути, щоб зберегти багатокутники, відсортовані по осях X / Y-розташування і глибини, щоб забезпечити кордони, в спробі швидко визначити, чи два багатокутника мають оклюзію [12].

Реалізація алгоритму

Оскільки маємо двовимірний екран, то повинні мати двовимірний z-буфер:

```
int *zbuffer = new int [width*height];
```

Як відомо, пам'ять не може бути двовимірною, вона завжди лінійна, тому, в цілях швидкодії, двовимірний масив зроблений на основі одновимірного. Перетворення із двох координат в індекс:

```
int idx = x + y*width;
```

Та з індексу в дві координати:

```
int x = idx % width;
int y = idx / width;
```

Після цього, алгоритм проходить по всім трикутникам і робиться виклик пастеризатора, якому передається картинка та z-буфер [12].

```
triangle(screen_coords [0], screen_coords [1], screen_coords [2],
image, TGAColor(intensity*255, intensity*255, intensity*255, 255),
zbuffer);

 [...]

void triangle(Vec3i t0, Vec3i t1, Vec3i t2, TGAImage &image,
TGAColor color, int *zbuffer) {
    if (t0.y==t1.y && t0.y==t2.y) return; // i dont care about
degenerate triangles
    if (t0.y>t1.y) std::swap(t0, t1);
    if (t0.y>t2.y) std::swap(t0, t2);
    if (t1.y>t2.y) std::swap(t1, t2);
    int total_height = t2.y-t0.y;
    for (int i=0; i<total_height; i++) {
        bool second_half = i>t1.y-t0.y || t1.y==t0.y;
        int segment_height = second_half ? t2.y-t1.y : t1.y-t0.y;
        float alpha = (float)i/total_height;
        float beta  = (float)(i-(second_half ? t1.y-t0.y :
0))/segment_height; // be careful: with above conditions no division
by zero here
        Vec3i A = t0 + Vec3f(t2-t0)*alpha;
        Vec3i B = second_half ? t1 + Vec3f(t2-t1)*beta : t0 +
Vec3f(t1-t0)*beta;
        if (A.x>B.x) std::swap(A, B);
        for (int j=A.x; j<=B.x; j++) {
            float phi = B.x==A.x ? 1. : (float)(j-A.x)/(float)(B.x-
A.x);
            Vec3i P = Vec3f(A) + Vec3f(B-A)*phi;
```

```
                int idx = P.x+P.y*width;
                if (zbuffer [idx]<P.z) {
                    zbuffer [idx] = P.z;
                    image.set(P.x, P.y, color);
                }
            }
        }
}
```

PVS та портали (потенційно видимі набори)

Опис алгоритму

Потенційно видимі набори використовуються для прискорення рендеринга 3D середовищ. Це форма оклюзивного виключення, в результаті якої набір потенційно видимих полігонів обчислюються заздалегідь, а потім індексуються під час виконання для того, щоб швидко отримати оцінку видимої геометрії. Термін ПВН іноді використовується для позначення будь-якого алгоритму оклюзивного виключення (так як по суті, це те, що майже всі алгоритми оклюзивного виключення обчислюють), хоча майже у всій літературі, вона використовується спеціально для оклюзивного виключення, що попередньо вираховують видимі набори і зв'язують ці набори з регіонами в просторі. Для того, щоб зробити цей зв'язок, камера вид-простір (безліч точок, з яких камера може зробити зображення), як правило, поділяється на багато (зазвичай опуклих) областей і ПВН обчислюється для кожного регіону [13].

Плюси алгоритму:

- Програма просто повинна обчислити набір потенційно видимих об'єктів, з огляду на те як вони виглядають с точки камери. Цей набір може бути додатково зменшений за допомогою виключення за областю зору (Frustum culling). В обчислювальному плані, це набагато дешевше, ніж обчислення на основі видимості кожного об'єкту на кожному кадрі.
- В межах кадру, час обмежений. Тільки 1/60-а секунди (передбачається, що 60 Гц для частоти кадрів є достатньою величиною) доступна для визначення видимості, але окрім видимості

ще повинні провести свої обчислення AI, фізика, або будь-які інші частини програми. На відміну від цього попередня обробка потенційно видимого набору може йти стільки часу, скільки потрібно, для того щоб точно визначити видимість.

Мінуси алгоритму:

- Існують додаткові вимоги для зберігання даних ПВН.
- Препроцесіювання може бути довгим або незручним.
- Не може бути використаний для повністю динамічних сцен.
- Видимий набір для регіону в деяких випадках може бути значно більше, ніж для точки.
- Існують різні класифікації алгоритмів ПВН щодо типу видимості набору яку вони обчислюють.

Консервативні алгоритми оцінюють видимість послідовно, таким чином, що жоден трикутник, який видно не може бути виключено. Кінцевим результатом є те, що немає помилки зображення, проте, можна значно більше витратити ресурсів на видимість, що призводить до неефективного рендеринга (в зв'язку з наданням невидимої геометрії). Акцент дослідження консервативних алгоритмів полягає в тому, щоб створити із декількох оклюдерів один [7].

Агресивні алгоритми не оцінюють видимість послідовно, таким чином, що відсутні надлишкові (невидимий) багатокутники не існують в наборі ПВН, хоча це може призвести до пропуску багатокутнику, який насправді видно, що призводе до помилок зображення. Акцент дослідження агресивних алгоритмів є зменшення потенційної помилки.

Приблизні алгоритми алгоритмі можуть призвести як до надлишковості, так і до помилок

Точні алгоритми забезпечують оптимальні набори видимості, де немає помилки зображення і надлишковості. Вони, однак, складні для реалізації і зазвичай працюють набагато повільніше, ніж інші алгоритми

видимості на основі ПВН. Видимість обчислюється точно для сцени, що поділена на клітини і портали.

Існує спосіб, щоб визначити, які частини сцени видно – це використання порталів. У рендерингу за домогою порталів, ігровий світ ділиться на багато напівзакритих областей, які з'єднані один з одним через отвори, такі як вікна і двері. Ці отвори називаються порталами. Вони зазвичай представлені полігонами, які описують їх межі.

Для візуалізації сцени з порталів, ми починаємо рендерити область, яка містить камеру. Тоді для кожного порталу в регіоні, ми розширюємо пірамідальну фігуру, що складається з площин, що проходять від фокусної точки камери. Кожен портал модифікує цю фігуру, через свій примітив (багатокутник). Об'єкти в сусідніх регіонах відсікаються так само, як і в методі виключення за обсягом зору (frustum culling) [5]. Це гарантує, що тільки видима геометрія в регіонах буде надана до рендеру.

Реалізація алгоритму

Це часто небажано або неефективно просто обчислити видимість трикутника. Для апаратної реалізації графіки краще коли об'єкти статичні і постійно знаходяться у відеопам'яті. Тому, як правило, краще, щоб обчислити видимість на основі кожного об'єкта і поділити будь-які об'єкти, які можуть бути занадто великими на декілька. Це додає консервативність, але вигодою є більш ефективне використання апаратних засобів і стиснення (оскільки видимість даних тепер за об'єктом, а не за трикутником).

Обчислювальний елемент або сектор також є перевагою, так як шляхом визначення видимої області простору, а не об'єкту, можна не тільки виключити статичні об'єкти в цих регіонах, але динамічні об'єкти.

В цілому, реалізація перевірки видимості не відрізняється від перевірки для виключення за областью зору (frustum culling) [8].

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі. Отже, як можна побачити, у кожного алгоритму є свої плюси та мінуси, проте, якщо чогось не може зробити один алгоритм – зможуть два, а те чого не зможуть зробити два алгоритми – зможуть зробити три. Саме тому, об'єднання всіх цих алгоритмів дасть максимальну ефективність. Виключення за областю зору видалить всі об'єкти що не потрапляють в піраміду зору, що в свою чергу пришвидшить алгоритми які залежать від кількості об'єктів. Після цього підключаються портали, що видаляють значну кількість об'єктів і вже потім на відеокарті Z-буфер видалить те, що залишилось.

Література:

1. Next generation occlusion culling. – Режим доступу: http://www.gamasutra.com/view/feature/164660/sponsored_feature_next_generation_.php?print=1. – Дата доступу : 04.06.16
2. GDC Vault. – Режим доступу : http://gdcvault.com/free/gdc-15. – Дата доступу: 04.06.16
3. Краткий курс компьютерной графики, аддендум: GLSL. – Режим доступу : https://habrahabr.ru/post/253791/. – Дата доступу: 04.06.16
4. GPU-Based Ray-Casting of Quadratic Surfaces. – Режим доступу: http://reality.cs.ucl.ac.uk/projects/quadrics/pbg06.pdf. – Дата доступу: 04.06.16
5. OpenGL 44 Pipeline Map. – Режим доступу: http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGL44PipelineMap.pdf – Дата доступу : 04.06.16
6. Dynamic Scene Occlusion Culling using Regular Grids. – Режим доступу: http://www.dca.fee.unicamp.br/projects/mtk/batageloM/. – Дата доступу : 04.06.16

7. Удаление невидимых поверхностей. Алгоритм, использующий Z-буфер. – Режим доступу : http://opita.net/node/58. – Дата доступу : 04.06.16
8. Удаление невидимых поверхностей: z-буфер. – Режим доступу : https://habrahabr.ru/post/248179/. – Дата доступу : 04.06.16
9. Practical, Dynamic Visibility for Games. – Режим доступу: http://blog.selfshadow.com/publications/practical-visibility/. – Дата доступу : 04.06.16
10. Hierarchical Visibility Culling with Occlusion Trees. – Режим доступу: http://dcgi.felk.cvut.cz/home/bittner/publications/cgi98-copy.pdf. – Дата доступу : 04.06.16
11. OpenGL rendering pipeline. – Режим доступу : http://www.songho.ca/opengl/files/gl_pipeline.gif. - Дата доступу : 04.06.16

References

1. Next generation occlusion culling. – Access: http://www.gamasutra.com/view/feature/164660/sponsored_feature_next_generation_.php?print=1. – Date : 04.06.16
2. GDC Vault. – Access: http://gdcvault.com/free/gdc-15. – Date : 04.06.16
3. Short course of computer graphics: GLSL. – Access: https://habrahabr.ru/post/253791/. – Date : 04.06.16
4. GPU-Based Ray-Casting of Quadratic Surfaces. – Access: http://reality.cs.ucl.ac.uk/projects/quadrics/pbg06.pdf. – Date : 04.06.16
5. OpenGL 44 Pipeline Map. – Access : http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGL44PipelineMap.pdf – Date : 04.06.16
6. Dynamic Scene Occlusion Culling using Regular Grids. – Access: http://www.dca.fee.unicamp.br/projects/mtk/batageloM/. – Date: 04.06.16

7. Invisible surface removal algorithm – Access: http://opita.net/node/58. – Date: 04.06.16
8. Invisible surface removal algorithm: z-buffer. – Access: https://habrahabr.ru/post/248179/. – Date: 04.06.16
9. Practical, Dynamic Visibility for Games. – Access: http://blog.selfshadow.com/publications/practical-visibility/. – Date: 04.06.16
10. Hierarchical Visibility Culling with Occlusion Trees. – Access: http://dcgi.felk.cvut.cz/home/bittner/publications/cgi98-copy.pdf. – Date: 04.06.16
11. OpenGL rendering pipeline. – Access: http://www.songho.ca/opengl/files/gl_pipeline.gif. - Date : 04.06.16