Стружанський Б.Р.
студент,
Національний університет «Львівська політехніка»

ОПТИМІЗАЦІЯ МОБІЛЬНИХ ДОДАТКІВ, ЯКІ ВИКОРИСТОВУЮТЬ OPENGL ES НА ПЛАТФОРМІ ANDROID

Великий асортимент мобільних пристроїв різних цінових категорії, які працюють під керуванням мобільної операційної системи Android, а відтак різної потужності породжує проблему некоректної роботи складних графічних мобільних додатків. Для того, щоб графічно-складний додаток чи гра працювали коректно здійснюють оптимізацію продуктивності.

Коли ми запускаємо додаток на потужному пристрої, все працює швидко і правильно. Якщо ми запустимо його на слабкому пристрої, все почне гальмувати, що дуже неприємно. OpenGL ES – це відкрита графічна бібліотека, яка здійснює швидку візуалізацію графіки. Візуалізація буде швидкою, якщо наш додаток буде побудований за принципами побудови високо продуктивних додатків [3].

Перед тим, як почати оптимізацію, потрібно спочатку оцінити роботу. Огляд вручну (тобто проста оцінка швидкості роботи на око) є недостатньо точним. Найкращий спосіб виміряти те, як швидко працює програма, – підрахувати кількість кадрів, які ми візуалізуємо в секунду. Функція vsync або функція вертикальної синхронізації, яку підтримують всі пристрої на Android, які зараз присутні на ринку, скорочує максимальну кількість кадрів в секунду, які ми можемо отримати, до 60. Загально відомо, що подібна частота кадрів цілком підходить для будь-якого додатку. Хоча непогано було б, якщо частота досягала б 60 кадрів в секунду, насправді цього не так просто досягти. Задовільним показником є більше 30 кадрів в секунду, якого достатньо для більшості додатків.

Некоректне відображення часто зумовлене не різною потужністю графічних процесорів, а тим, що викликається безліч методів OpenGL ES в кожному кадрі і ці методи займають дуже багато пам’яті. Це означає, що вони фактично викликають С-код, який потребує більше пам’яті, ніж виклик методу Java на Dalvik.

Виділимо основні причини низької продуктивності в OpenGL ES:

- безліч змін станів у кожному кадрі (наприклад, змішування, включення/ вимикання нанесення текстури і т. д.);
- велика кількість змін матриць в кожному кадрі;
- безліч операцій прив’язування текстур в кожному кадрі;
- багаторазові зміни вершин, кольору та координат текстури в кожному кадрі [2].

Всі ці проблеми фактично пов’язані зі змінами станів. Графічний процесор працює як складальна лінія на фабриці. Поки перша лінія обробляє вхідні товари, кінцева лінія закінчує вже оброблені на всіх попередніх стадіях

продукти. Спробуємо провести паралель з автомобільним конвеєром. Продукція на виробничій лінії проходить через кілька станів. Спочатку застосовуються інструменти, з якими працюють фахівці заводу, потім кріплення, що використовуються для з'єднання частин машини, потім в хід йде фарба, якою покривають машини, і т. д. Звичайно, справжні автомобільні заводи мають у своєму розпорядженні безліч ліній, але припустимо, що лінія всього одна. У такому разі кожна стадія лінії буде зайнята до тих пір, поки ми не змінимо стан. Коли ми змінюємо один етап, лінія буде продовжувати працювати, поки всі збираються машини не будуть закінчені. Тільки після цього ми можемо перейти до іншого стану і збирати машини нового кольору або з новими кріпленнями.

Ключовий момент полягає в тому, що виклик функцій glDrawElements () або glDrawArrays () виконується не відразу. Замість цього команда поміщається в буфер, який асинхронно обробляється графічним процесором. Це означає, що виклик методів малювання не блокується. Тому навряд чи варто вимірювати, скільки часу займе виклик glDrawElements (), оскільки сама робота може бути виконана пізніше. Саме тому ми вимірюємо продуктивність в кадрах за секунду. Коли фреймбуфери міняються місцями (в OpenGL ES застосовується подвійна буферизація), OpenGL ES виконує всі відстрочені операції. Знову порівняємо OpenGL ES з автомобілебудівним заводом. У той час як нові елементи надходять в командний буфер за допомогою виклику функцій glDrawElements () або glDrawArrays (), конвеєр графічного процесора повинен закінчити візуалізацію поточних елементів з попередніх викликів [2].

Цим зумовлені такі особливості:

- На заміну актуальної прив'язаної текстури йде багато пам'яті. Будь-які ще не оброблені, але використовують текстуру елементи в командному буфері повинні спочатку бути візуалізовані. [1]
- Зміна вершин, кольору або текстурних координат також вимагає багато пам'яті. Будь-які ще не візуалізовані елементи, розташовані в командному буфері, які застосовують старі вказівники, повинні спочатку бути візуалізовані [1].
- Зміна стану змішування вимагає багато пам'яті. Будь-які ще не візуалізовані, але потребуючі чи не потребуючі змішування елементи у командному буфері, до яких повинні бути застосовані старі матриці, повинні спочатку бути візуалізовані.
- Багато пам'яті потрібно на зміну моделі-виду або проекційної матриці. Будь-які елементи, що перебувають у командному буфері, які поки що не були оброблені і до яких повинні застосовуватися старі матриці, доведеться відображати в першу чергу. Конвеєр зупиниться [1].

Виділимо основні способи оптимізації:

- **Зменшення розміру текстури**

Використовуючи цю меншу за розміром текстуру, отримаємо вище швидкість заповнення. Швидкість, на якій графічні процесори можуть завантажувати тексели і візуалізувати пікселі у фреймбуфері, називається

швидкістю заповнення. Необхідно стежити за накладенням: чим менше елементи перетинаються, тим краще.

- **Зменшуємо кількість викликів методів OpenGL ES/JNI**

Необхідно зменшити кількість викликів до мінімуму, наприклад для побудови двохвимірної графіки. Часто можна виключати методи, що обертають елементи, чий його збільшують.

- **Концепція зв'язування вершин**

Доволі часто для елементів використовують одні й ті ж властивості вершин знову і знову, але їх достатньо встановити лише один раз, оскільки кожен елемент застосовує одну і ту ж модель, яка завжди використовує одні і ті ж властивості вершин. Доцільно не вносити зайвої інформації в стан OpenGL ES [1].

- **Об'єднання в групи (batching)**

Елементи об'єднуються в групи, а отже скорочується кількість викликів glDrawElements ()/glDrawArrays (). Аналогічний процес застосовується і в 3D-графіці, він називається клонуванням (instancing).

Оцінювання швидкості роботи програми здійснюється шляхом підрахунку кількісті кадрів, які ми візуалізуємо в секунду. [4]

Завжди, кількість кадрів в секунду трохи коливається. Це можна пояснити фоновими процесами, які виконуються паралельно з додатком. Ми ніколи не зможемо витратити на додаток всі системні ресурси. При оптимізації програми, не можна використовувати емулятор, необхідно запустити програму на реальному пристрої в нормальному стані, в якому він використовується протягом дня. Це відобразить ситуацію, з якою зіткнеться користувач.

Оптимізація коду візуалізації здійснюється тільки після того, як завершиться робота над ним і в тому випадку, якщо виникнуть проблеми з роботою пристрою. Передчасна оптимізація часто призводить до необхідності переписати весь код візуалізації, оскільки в деяких випадках код після оптимізації стає неможливо підтримувати. [2]

Список використаних джерел:

1. Цехнем М. Программированиє игр под Android / М. Цехнем – Питер: СПб, 2013. – 688 с.

2. Rogers R. Learning Android Game Programming / R. Rogers – Addison Wasley, 2012. – 476 с.

3. Вікіпедія «OpenGL» [електронне джерело]. – Режим доступу: http://uk.wikipedia.org/wiki/OpenGL#.D0.A1.D0.BF.D0.B5.D1.86.D0.B8.D1.84.D1.96.D0.BA.D0.B0.D1.86.D1.96.D1.8F

4. Коматинени С. Android 4 для профессионалов / Коматинени С., Маклин Д. – Нью Йорк: Вільямс, 2012. – 880 с.