1. Вывод видео

Самый портабельный способ YUV видеорендеринга - отрисовка кадров с помощью OpenGL. Цветовая конвертация YUV->RGB производится на графическом ускорителе. Суть очень проста - нужно нарисовать прямоугольник с текстурой. Текстура обновляется с каждым новым кадром. В классическом OpenGL видеорендеринге необходимо 3 текстуры - по одной на каждую цветовую плоскость: Y, U и V. Во время рендеринга фрагментный шейдер пересчитывает тексели из этих трех текстур в выводимый на экран пиксель RGB согласно стандарту BT.601, BT.709 и др. Главная задача - быстро обновлять текстуры. Некоторые платформы предоставляют специализированные типы текстур, позволяющие записывать в них данные в цветовом пространстве YUV, либо отдельные его компоненты (перемежающиеся U->V). Классические текстуры OpenGL не поддерживают формат YUV. В остальном рендеринг имеет схожие шаги:

- если доступен очередной кадр для рендеринга, обновить текстуры
- нарисовать прямоугольник

Нарисовать прямоугольник - что может быть проще? Опытные OpenGL разработчики, могут пропустить оставшуюся часть данного раздела.

Технически OpenGL представляет собой конечный автомат. Сначала, с помощью последовательности gl вызовов, задается последовательность команд для отрисовки сцены (переходов между состояниями), затем отрисовывается сцена (запускается автомат).

Немного о конвейерах. В OpenGL есть понятие конвейер. Конвейер может быть фиксированным или программным. Фиксированный конвейер располагает только фиксированным набором gl функций для отрисовки сцен и создания эффектов. Фиксированный конвейер используется в устаревших версиях стандартов:

- в мобильных системах до OpenGL ES 2.0
- в настольных системах до OpenGL 3.0(?)

Программный конвейер предоставляет возможность выполнить произвольный код на стороне GPU, на специализированном вычислительном устройстве - шейдере. Это позволяет реализовывать самые разнообразные графические алгоритмы.

Немного о шейдерах. Шейдер - это вычислительное микроядро. GPU содержит большое количество шейдеров (десятки-сотни), работающих параллельно. Шейдер исполняет микропрограмму, написанную на языке GLSL. GLSL имеет C/C++ подобный синтаксис. Программа шейдера компилируется во время исполнения графическим драйвером конкретного устройства. Шейдеры бывают: вершинными и фрагментными (в

настольных версиях OpenGL могут быть и другие). Вершинный шейдер отвечает за геометрические трансформации, он работает с координатами вершин. Фрагментный шейдер отвечает за растеризацию (раскраску) моделей, он оперирует текселями (пикслями тектур) и пикселями итогового изображения. GLSL не стоит на месте, а постоянно развивается привнося с собой несовместимые изменения. Для детектирования версии во время компиляции шейдера можно использовать встроенные в препроцессор макросы _VERSION_ и GL_ES. Первый содержит целочисленную константу версии GLSL, второй информирует о том, что используется встраиваемая система.

В задаче вывода YUV кадров, вершинный шейдер можно использовать для геометрических трансформаций: соотношение сторон, поворот, параллельный перенос, масштабирование и т.д., фрагментный шейдер - для цветовой конвертации, и простого постпроцессинга: регулировка яркости, насыщенности, цветности и т.п.

Для поддержки наиболее широкого круга устройств, имеет смысл ориентироваться на спецификацию OpenGL ES 2.0. Несмотря на то, что она разработана для мобильных систем, в ней собрана выжимка функционала настольной версии. OpenGL ES 2.0 поддерживается такими библиотеками как Qt 5, Google Native Client.

Немного о рисовании. OpenGL предоставляет возможность рисования различными примитивами: линиями, треугольниками, четырехугольниками и др. В OpenGL ES рисовать можно максимум треугольниками, поэтому мы будем рисовать наш прямоугольник с помощью двух треугольников с общей стороной. В 3D графике рисование треугольниками с общей стороной позволяет сократить количество вершин, описывающих модель, в результате сокращается объем данных, передаваемых с CPU на GPU и объем вычислений вершинного шейдера.

Немного о системах координат. Система вершинных координат X;Y;Z в OpenGL нормирована в диапазоне значений [-1;+1]. Система тектсурных координат U;V (не путать с компонентами цветового пространства YUV) нормирована в диапазоне [0;1]. Внезапно! Сделано это для того, чтобы смысловой диапазон текстурных координат был таким же, как и размер текстуры. Для отрисовки плоского двумерного изображения мы будем игнорировать координату Z. На рисунке 1 изображены системы вершинных и текстурных координат, с которыми будем работать дальше. Следует заметить, что начало координат текстур расположено в левом нижнем углу, в то время как начало координат цифровых изображений обычно располагается в левом верхнем углу. Это нужно учитывать при сопоставлении координат вершин координатам текстур (выворачивать их наизнанку).

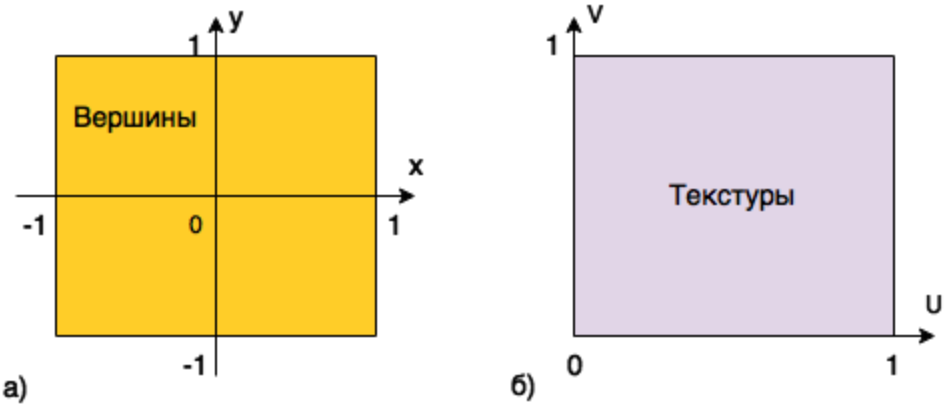


Рисунок 1 Системы координат OpenGL: (а) вершинные, (б) текстурные.

Немного о проекциях. Для отрисовки трехмерной сцены на плоском экране, необходимо сделать проекцию трехмерного мира на двумерную плоскость. Проекции бывают разными, в том числе перспективные и ортогональные. В перспективной проекции параллельные прямые на бесконечном удалении сходятся в одной точке. В ортогональной проекции параллельные прямые остаются параллельными на любом удалении. Применение проекции и геометрические трансформации производятся умножением квадратной матрицы model-view-projection на матрицу-столбец координат каждой вершины.

Немного о матрицах. Матрицы в OpenGL ориентированы по столбцам (column-major), в отличие от классических матриц линейной алгебры, которые ориентированы по строкам (row-major):

$$RowMajor = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}, ColumnMajor = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

Рассмотрим матрицы, которые могут быть полезны для реализации отрисовки YUV кадров.

$$Ortho = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}, (1)$$

Матрица ортогональной проекции Ortho (1) предназначена для того, чтобы на экране с произвольным соотношением сторон сохранялись пропорции трехмерных

объектов. Чтобы квадрат не превращался в прямоугольник/параллелограмм, круг в эллипс и т.п.

$$Translate = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{pmatrix}, (2)$$

$$Scale = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, (3)$$

Матрица параллельного переноса Translate (2) и матрица масштабирования Scale (3) могут быть использованы для масштабирования и смещения YUV кадра (resize/crop), а также для задания произвольного соотношения сторон (AspectRatio).

$$Rotate_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, (4)$$

$$Rotate_y = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, (5)$$

$$Rotate_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, (6)$$

Матрица вращения Rotate вокруг оси Z (6) может быть использована для поворота изображения на произвольный угол.

Матрица “модель” рассчитывается следующим образом:

$$Model = Rotate_z \cdot Translate \cdot Scale, (7)$$

В видеорендерах OSAL используется именно такая матрица Model. Она с успехом использовалась в деме MovieFlex с вращающимся элементом видеостены.

Порядок умножений имеет значение. В (7) сначала производится поворот вокруг начала координат, затем параллельный перенос, затем масштабирование. Каждая

операция работает с результатом предыдущей. Например, если в (7) поменять местами Translate и Scale, то параллельный перенос будет произведен с масштабным коэффициентом Scale. И, вероятно, мы получим не совсем то, что ожидали. По аналогии, если поменять местами Rotate и Translate, то вращение будет вокруг точки, смещенной относительно начала координат.

Матрица Model-View-Projection получается следующим образом:

$$MVP = Ortho \cdot Model, (8)$$

Матрицу (8) достаточно пересчитывать только в следующих случаях:

- при изменении размера окна вывода
- при смене ориентации мобильного устройства: портретная <-> ландшафтная (частный случай предыдущего)
- при изменении значения соотношения сторон
- при изменении параметров геометрических трансформаций (поворот, масштаб, перенос и т.п.)

Матрицу (8) будет использовать вершинный шейдер (GLSL):

$$gl\_Position = MVP \cdot position, (9)$$

Немного о рисовании треугольниками. Как было сказано ранее, рисовать мы будем треугольниками с общей стороной (Triangle strip). На рисунке 2 а) изображен пример рисования YUV кадра.

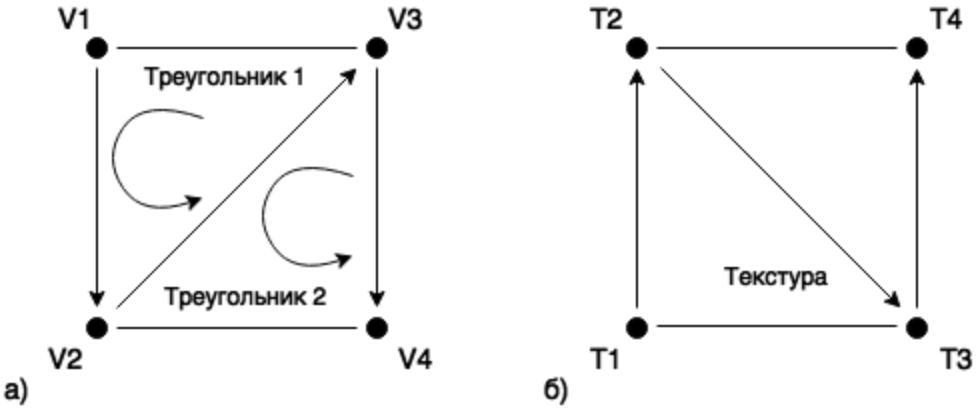


Рисунок 2 Отрисовка YUV кадра: а) порядок следования координат вершин, б) сопоставление координат текстуры координатам вершин.

Рисование происходит против часовой стрелки. Сначала рисуется треугольник 1: V1 -> V2 -> V3. Затем, начиная с последней вершины треугольника 1 V3, вдоль общей стороны, рисуется треугольник 2: V3 -> V2 -> V4. На рисунке А.2 б) приведен порядок следования координат текстуры T1…T4 для вершин V1…V4. Вершине V1 соответствует координата текстуры T1, вершине V2 текстура T2 и т.д. Начало координат текстур OpenGL находится в левом нижнем углу, а начало координат YUV кадра - в левом верхнем углу. Поэтому, в приведенном примере, координаты текстуры заданы в таком порядке, чтобы ее содержимое перевернулось в вертикальном направлении. Значения координат вершин и текстур задаются равными граничным значениям, например: V1 (-1;0), V2(-1;1), и т.д., T1 (0;0), T2(0;1) и т.п. Соотношение сторон реального изображения задается в матрице Model наряду с прочими геометрическими трансформациями.

Немного о деинтерлейсе. На первый взгляд может показаться, что для реализации деинтерлейса подойдет фрагментный шейдер: извлекать тексели только из четных или нечетных строк текстуры. Однако, это будет работать только тогда, когда размер текстуры будет точно соответствовать размеру буфера экрана. В противном случае текстурный сэмплер будет делать интерполяцию текселей из соседних строк и столбцов, перемешивая тем самым верхнее и нижнее поле. В этом случае можно попробовать задать тип интерполяции текстур “ближайший сосед”, ухудшив тем самым качество выводимого изображения. Другой вариант - делать сначала рендеринг в текстуру, а затем выводить ее на экран, но этот способ требует достаточно больших изменений и потенциально хуже с точки зрения производительности. Возможно, есть эффективный способ реализации деинтерлейса средствами шейдера, но мы рассмотрим более простой способ, требующий минимальных изменений “классической” схемы.

В случае чересстрочной развертки, YUV кадры содержат два перемежающихся поля: верхнее и нижнее. Можно заметить, что задав текстуре удвоенную ширину и половинную высоту, мы получим изображение, равное по площади чересстрочному, но фактически это будет изображение двух полей, расположенных рядом. Пример такого “преобразования” показан на рисунке 3.

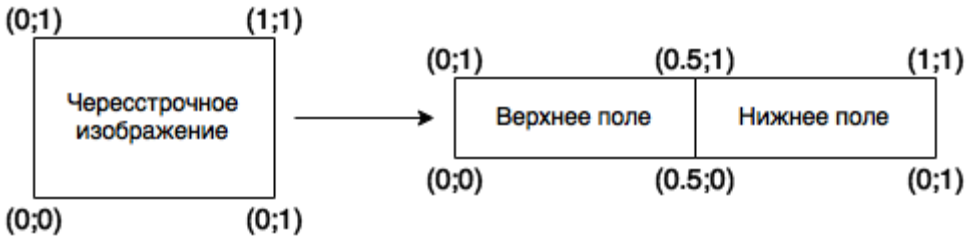


Рисунок 3 Текстурное представление чересстрочного изображения отдельными полями.

Далее вместо отрисовки всей текстуры, отрисовываем одну из половин, текстуры, соответствующую верхнему или нижнему полю. В самом простом случае достаточно отрисовывать только верхнее или только нижнее поле. Это требует минимальных изменений в коде видеорендера. В более продвинутой реализации можно производить поочередный вывод верхних и нижних полей с удвоенной частотой кадров в заданном порядке.

Плюс такого деинтерлейса заключается в том, что текстурный семплер, растягивая изображение по вертикали, произведет “бесплатную” линейную интерполяцию. Это будет визуально лучше, чем изображение с дублированными полями, эквивалентное грубой интерполяции “ближайший сосед”.

EGL, EAGL, GLUT,... WTF???!!!

Сам по себе OpenGL выводить графику на экран не умеет. Для этого существуют специальные интерфейсы, предназначение которых - связать контекст OpenGL с контекстом графического окна операционной системы. На настольных системах могут быть доступны сразу несколько таких интерфейсов, а в мобильных как правило ограничиваются чем-то одним. В Android используется стандарт EGL, разработанный консорциумом Khronos. В iOS и OSX используется EAGL - эпловский вариант EGL, написанный на Obj-C. В Qt этот момент скрыт от разработчика. В Windows/Linux используется GLUT и др.: https://www.opengl.org/wiki/Related_toolkits_and_APIs В связи с этим, способы инициализации графической подсистемы на разных платформах могут существенно различаться. При этом код рендеринга будет практически идентичен.

1.1 Вывод видео Android

Вывод видео в ОС андроид - одна из самых изученных и в то же время самых загадочных областей.

Главным фактором, осложняющим разработку под Android, является гетерогенность конечного приложения. Графический интерфейс для Android по факту можно разрабатывать только на Java. В то время, как весь код Native Codec SDK расположен на нативной стороне. Возникает вопрос - как максимально эффективно связать видеовывод из нативного кода с Java UI?

1.1.1 Вывод видео Android ANativeWindow

В стандартном мультимедиа фреймворке андроида Stagefright для вывода видео используется интерфейс `ANativeWindow`. Он представляет собой абстракцию “очередь графических буферов”. Для таких элементов управления как `SurfaceView` можно

получить указать на `Surface`, передать его в нативную часть и получить из него указатель на `ANativeWindow`. Далее работа сводится к простым шагам:

- конфигурирование буферов (цветовой формат, размеры, число буферов и др.)
- получение пустых буферов из очереди
- наполнение буферов данными
- постановка наполненных буферов в очередь

Все просто и логично. Но разработчики Android посчитали излишним предоставлять пользователю NDK весь функционал `ANativeWindow`, зарезав цветовой формат YUV, большинство конфигурационных настроек и возможность снять с очереди более одного буфера. Но мы-то знаем, что все это `ANativeWindow` делать умеет. Достаем из репозитория AOSP необходимые заголовочные файлы и начинаем использовать `ANativeWindow` по полной программе. И это даже работает, почти на всех устройствах. Ключевое слово - почти, которое изрядно подпортит нервы, когда на чипе nVidia будет зеленый экран, на exynos4 - кривой страйд хромы, а на amlogic приложение вообще упадет. Здесь речь идет не о всех процессорах, а только о некоторых моделях.

Что делать? Посмотрев исходники андроида от проблемных вендоров, мы видим, что логика работы с `ANativeWindow` на реальном устройстве может немного отличаться от референсного кода AOSP. Тогда мы детектируем в рантайме тип SoC и выполняем вендор-специфичные магические действия. У нас снова все работает до тех пор, пока не обнаружится новая “интересная” железка и снова нужно ломать голову, что там не так. Ситуация осложняется тем, что проблемный девайс в наличии может быть только у клиента, а также отсутствие в свободном доступе исходного кода BSP. Главный недостаток `ANativeWindow` - платформозависимость. Это наполовину приватный API.

В чем преимущества `ANativeWindow`? По сути это интерфейс доступа к общей памяти между CPU и GPU. Поэтому, с помощью `ANativeWindow` можно добиться максимальной скорости видеорендеринга без промежуточного копированиия кадров (zero-copy). Декодировать кадр можно напрямую в общую память, а вывод кадра на экран производить постановкой наполненного буфера в очередь. На этом принципе основана работа аппаратных видеокодеков в Android, у них нет понятия видеорендер/видеокапчур. Выходному порту компонента “декодер” передается экземпляр `ANativeWindow`, который по сути используется как аллокатор памяти для DPB декодера.

Более подробно о внутреннем устройстве графичесного стека Android можно почитать здесь: http://source.android.com/devices/graphics/index.html Читать следует с осторожностью, т.к. материал предназаначен для разработчиков платформ.

1.1.2 Вывод видео Android OpenGL ES 2.0

Это классический OpenGL видеорендер, описанный в A.1. Перерисовка сцены производится с максимальной скоростью редеринга (60FPS, если устройство не в режиме энергосбережения). Устроен видеорендер следующим образом:

- на стороне Java есть класс OsalVideoView, унаследованный от GLSurfaceView.
- этот класс реализует колбэки GLSurfaceView.Renderer, которые вызывают нативные методы.
- самый важный колбэк onDrawFrame, он вызывается потоком, владеющим OpenGL контекстом.
- на нативной стороне в обработчике nativeDrawFrame() делается проверка - доступен ли очередной YUV кадр для рендеринга, если доступен, то производится обновление текстуры. В любом случае (была или не была обновлена текстура) производится отрисовка сцены. Если отрисовку не производить, возникает мерцание, причина возникновения которого по видимому в том, что после вызова DrawFrame в недрах GLSurfaceView вызывается glSwapBuffers, но так как мы ничего не рисовали, то на экран вылетает мусор. По-хорошему отрисовка сцены в случае не изменившейся текстуры производиться не должна, т.к. расходует вычислительные ресурсы.

Android поддерживает альтернативный способ обновления OpenGL текстур. Так называемый `SurfaceTexture` - объект, который с одной стороны является уже знакомым нам `ANativeWindow`, а с другой стороны нативной Android-специфичной OpenGL текстурой в формате YUV. Для такого типа текстур в андроид GLSL даже есть специфичный текстурный сэмплер `samplerExternalOES`, который “на лету” преобразовывает тексели YUV -> RGB. Ну и по традиции - андроид повернут лицом только к Java программистам. Вы можете использовать объект `SurfaceTexture` совместно со стандартным Java-медиаплеером, но как только захотите использовать его самостоятельно, то столкнетесь с проблемой платформо зависимости, т.к. по сути будете работать с `ANativeWindow`. Такой вот каламбур, вроде все есть, но пользоваться этим нельзя.

1.1.3 Android offscreen rendering

Применяется для промежуточных отрисовок и/или для вытаскивания данных из GPU. Вытаскивание данных из GPU может быть применено, например, для извлечения декодированных кадров из HW декодера, но как оказалось, это можно сделать проще и эффективнее.

Есть разные способы вытаскивания данных из GPU: использование glReadPixels и EGLImageKHR. В обоих случаях создается EGL контекст с атрибутами

```
EGL_RENDERABLE_TYPE = EGL_OPENGL_ES2_BIT
EGL_SURFACE_TYPE    = EGL_PBUFFER_BIT
```

Первый способ предполагает рендеринг в EGL Pixel Buffer Surface, второй в GL FrameBuffer, которому сопоставлен шаредный ресурс ImageKHR.

Первый способ может проигрывать второму в плане производительности (пишут, что просадка замечается, например, на Exynos 4), но при этом он максимально портабелен. Второй способ, напротив, использует платформо-зависимое приватное API для аллокации графической памяти (android::GraphicBuffer).

1.2 Вывод видео Darwin (OSX / iOS)

Вывод видео в мобильной и настольной системах Apple во многом схож. Для вывода используется OpenGL. Цикл отрисовки происходит в колбэке DisplayLink, привязанному к низкоуровневым событиям графической подсистемы. В реализации этого колбека происходит классический OpenGL видео рендеринг, описанный в A.1. Особенность iOS видеорендера. В iOS версии не используются канонічные OpenGL текстуры. Вместо них используется связка из CVPixelBuffer и CVTextureOpengles. PixelBuffer это по сути блок разделяемой памяти между cpu и gpu. На мобильном устройстве графическая и основная память - разделяемая. Поэтому копирование как таковое не требуется, но необходим способ передачи владения адресным пространством между CPU и GPU. Для этого предназначена CVTextureOpengles. В процессе отрисовки, из PixelBuffer’а создаются две текстуры (люма и перемежающаяся хрома), копирование при этом не происходит, текстуры просто “пробрасывают” указатели в gpu драйвер. Таким образом, если декодировать кадр непосредственно в PixelBuffer, а затем выводить его на экран при помощи CVTextureOpengles, можно получить максимальную производительность вывода видео (zero copy). Если проводить аналогию с Android, то массив PixelBuffer’ов будет выступать в роли ANativeWindow, а CVTextureOpengles в роли

нативной текстуры. В отличие от Android, CVTextureOpengles сэмплируется обычным сэмплером, поэтому перед выводом текселя, необходимо произвести цветовую конвертацию yuv->rgb на фрагментном шейдере.

TODO: сделать для iOS кастомный аллокатор пула PixelBuffer'ов и ложить в OMX буферы в PrivateData указатели на PixelBuffer. На выхлопе из декодера, копировать данные в PixelBuffer. Выхлоп HW декодера будет zero copy.

1.3 Вывод видео Qt

Разработка Qt подразделяется в том числе на Qt quick и на Qt widgets.

В настоящее время у нас есть опыт реализации видеорендера только для Qt widgets. Видеорендер для Qt widgets - это классический OpenGL видеорендер, описанный в A.1. Перерисовка сцены производится:

- когда обновилась текстура
- когда изменился размер окна рендеринга.

В Qt адреса всех OpenGL функций могут быть получены только во время исполнения (на самом деле это касается всех реализаций OpenGL). Поэтому в Qt для использования gl-API необходимо наследоваться от класса QOpenGLFunctions.

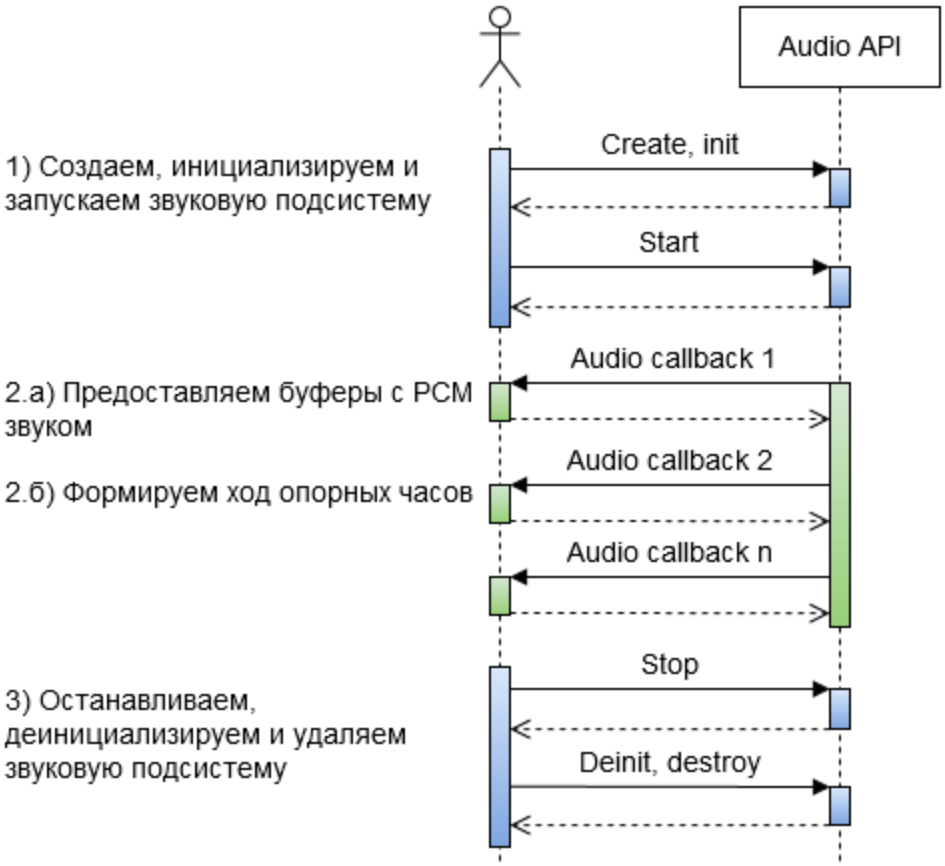
Qt видеорендер OsalVideoView - это наследник класса QOpenGLWidget. Отрисовка кадра происходит в методе paintGL. Никаких платформо-зависимых вещей не используется. Кадр на GPU передается в трех отдельных текстурах - Y, U и V.

1.3 Вывод видео Google Native Client

Видеорендер для NACL является примером классического OpenGL видеорендера. Никакие платформо-зависимые вещи не используются, хотя намек на то, что они появятся есть в примерах PPAPI. Реализован видеорендер с использованием C-only Pepper API.

2 Вывод звука

API вывода звука схож на многих платформах. Как правило это асинхронный API, в основе которого лежат колбэки для получения и/или освобождения буферов звуковых данных. В мультимедиа, вывод звука кроме своей непосредственной функции, выполняет дополнительную функцию источника опорных часов для синхронизации графа. Для эффективной реализации опорных часов, системный API вывода звука должен обладать минимальной и/или детерминированной задержкой вывода. К сожалению, этим требованиям удовлетворяют далеко не все Audio API.



2.1 Вывод звука Android

Начиная с Android 2.3+ в Android NDK появилась поддержка OpenSL ES. С помощью этого API можно относительно легко реализовать вывод потокового звука с приемлемой задержкой.

Суть сводится к созданию объекта “плеер” и регистрации колбека для предоставления аудио буферов. Затем можно запускать “плеер”, который в свою очередь начинает вызывать колбеки для получения аудио сэмплов.

Качество OpenSL ES в Android с точки зрения реализации опорных часов - хорошее.

2.2 Вывод звука Darwin

Вывод звука в мобильной и настольной системах Apple практически идентичен. Для вывода используется фреймворк AudioUnit:

- создаем граф
- добавляем в граф ноды converter и output
- конфигурируем ноду converter (задаем семплрейт, число каналов и т.п.)

- в ноду converter передаем указатель на колбэк для получения данных
- соединяем ноды и запускаем граф

После этого нам начинают прилетать колбэки для получения аудиосэмплов.

AudioUnit обладает наименьшей задержкой и всех Audio API, с которыми мне доводилось работать в контексте Native Codec SDK. Он превосходно подходит для реализации опорных часов.

2.3 Вывод звука Qt

Вывод звука в Qt не привязан ни к QtWidgets, ни к QtQuick. Аудиорендер является наследником класса QIODevice. В отличие от примеров Qt, наш аудиорендер работает не в главном потоке, а в фоновом. Оказалось, что в этой ситуации нужно организовать EventLoop и задавать высокий приоритет потоку (чтобы не возникали провалы звука). Класс Worker является реализацией EventLoop.

2.4 Вывод звука Google Native Client

Вывод звука в NACL - задача тривиальная. Для ее решения используется интерфейсы PPB_Audio и PPB_AudioConfig.

2.5 Вывод звука Win32 API

TODO