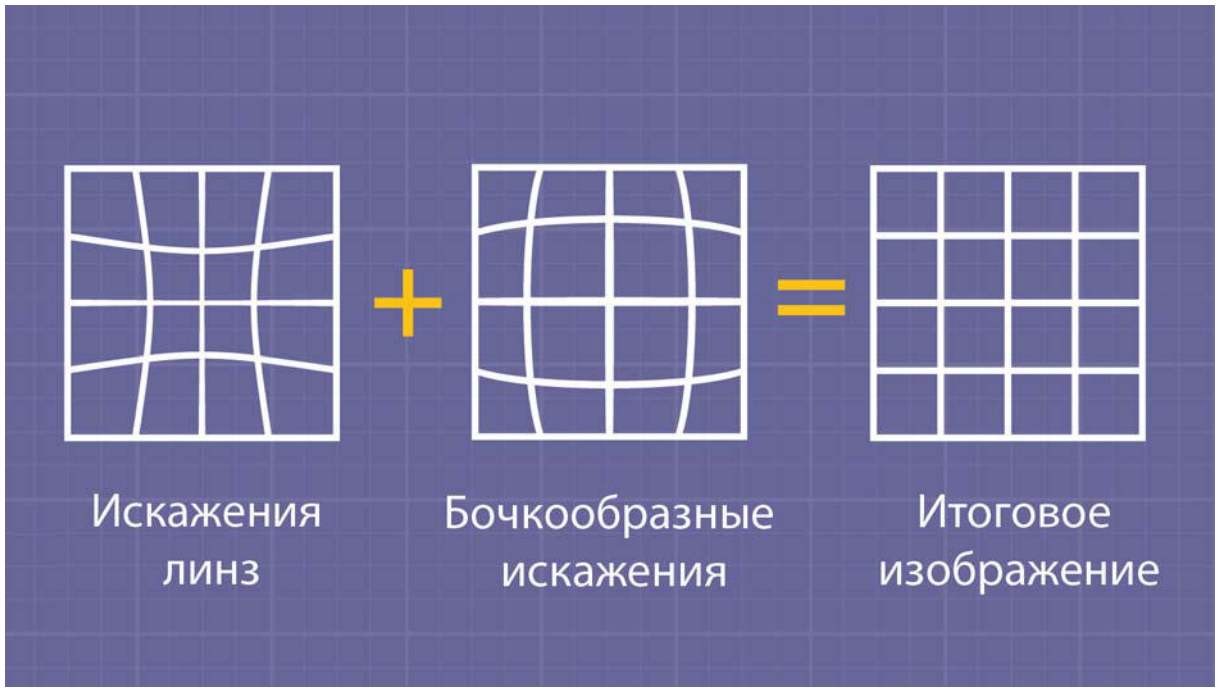

Рис. 10.1. Коррекция искажения *VR* линз

При том что коррекция искажения позволяет решить проблему фокусировки, искажение «бочки» приводит к уменьшению воспринимаемого разрешения дисплея. Чтобы избежать потери качества, многие *VR*-игры рендерят картинку с разрешением в 1,3–1,5 раза больше, чем «родное» разрешение экрана. В *UE*4 введено понятие относительного разрешения рендеринга [*screen percentage*] в процентах, и это очень важный параметр при оптимизации качества и производительности. Относительное разрешение может быть как меньше 100% (для повышения частоты кадров), так и больше 100% (для повышения качества за счет избыточной выборки, или суперсэмплинга [*supersampling*]).

Поэтому первоначальные технические требования для рендеринга в *VR* нужно умножить в полтора раза. Таким образом, для *Rift CV*1 и *Vive* (у обоих шлемов два экрана с разрешением 1800×1200 каждый), общее разрешение рендеринга получается 2160×1200×1,5, а именно 3200×1800 при 90 *fps*, или, другими словами, 350 млн пикселей в секунду! В противовес этому, в 2*D*-игра с разрешением 1080*p* при 60 *fps* требует обработки всего лишь 124 млн пикселей в секунду.

10.2. Уменьшение задержки

Как мы уже убедились, увеличение задержки обновления изображения в *VR*-шлеме имеет исключительно негативные последствия для пользователя, ставя под сомнение убедительность взаимодействия с виртуальной реальностью. Требование минимизации задержки вывода намного важнее для *VR*, чем для других способов визуализации. К счастью, существуют различные методы, уменьшающие или даже устраняющие основные причины задержек в конвейере рендеринга. В этой главе рассматриваются два основных способа борьбы с задержками: трансформация шкалы времени [*timewarping*] и рендеринг в передний (кадровый) буфер [*front buffer rendering*].

Прежде чем мы перейдем к тому, что такое трансформация шкалы времени (или перепроецирование) и рендеринг в передний (кадровый) буфер, давайте быстро пройдемся по конвейеру рендеринга, чтобы лучше понять, как эти возможности сочетаются с предыдущим опытом. (Хотя прикладные разработчики не сосредоточены на особенностях рендеринга и им редко приходится иметь дело с этими подробностями, поскольку *UE*4 скрывает большинство деталей реализации, по-прежнему важно знать эти нюансы, как минимум для того, чтобы правильно диагностировать возникающие проблемы.)

Для построения одного кадра игры несколько компонентов работают сообща, чтобы создать окончательное изображение. Центральный процессор (ЦП или *CPU*) опрашивает пользовательский ввод, совершает игровые действия (обсчитывает физическую модель мира, шаги искусственного интеллекта и т. д.) и в итоге отправляет вызовы отрисовки, которые будут выполняться на видеочипе (*GPU*). Традиционно после того, как *GPU* закончил рендеринг этих вызовов отрисовки, он отправляет их на дисплей для сканирования, который затем (построчно слева направо, сверху вниз) переводит изображение на экран для пользователя. рис. 10.2 иллюстрирует это.

Заметка

Большинство современных *VR*-шлемов реализуют технологию *low-persistence*, которая позволяет значительно сократить размытость изображения при движении головой. Смысл заключается в том, чтоб задержать вывод изображения до момента за доли секунды до следующего обновления дисплея, чтобы предотвратить показ пользователю «плохой» (рваной/размытой) картинки. Тем не менее эта книга для упрощения предполагает работу с традиционным дисплеем.

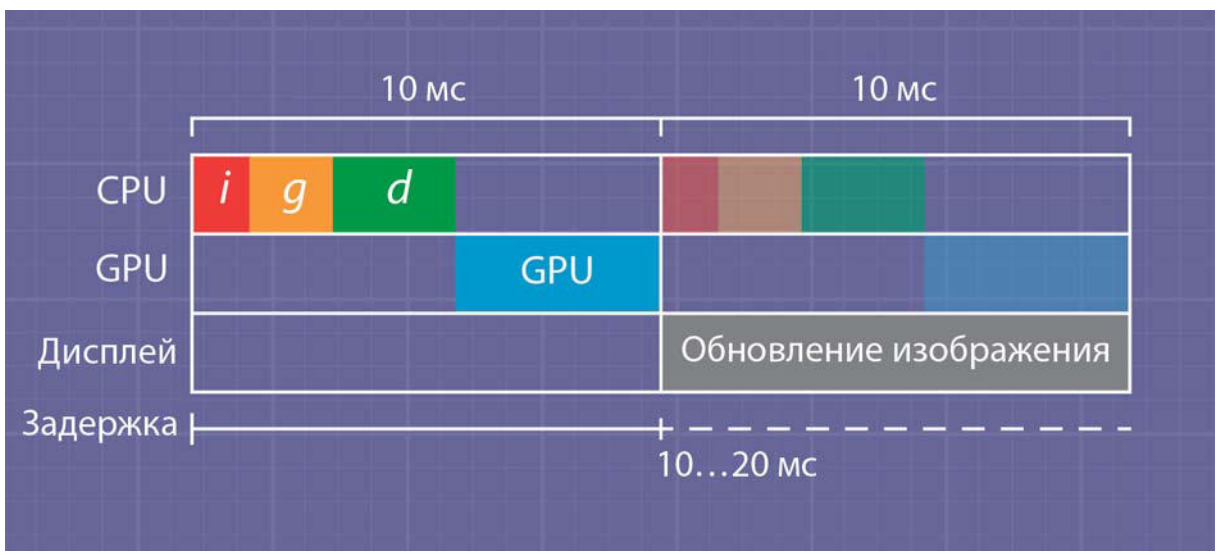

Рис. 10.2. Упрощенное представление конвейера рендеринга одного кадра; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Обратите внимание, что рис. 10.2 является не самым оптимальным вариантом настройки с точки зрения производительности. Отметим длительные периоды, когда *GPU* или *CPU* ничего не делают, ожидая окончания работы друг друга. Также отметим целый кадр задержки от ввода (*i*) до обновления изображения.

Чтобы устранить эти недостатки, мы можем для начала попробовать убрать паузы в работе процессоров путем введения параллельной конвейерной архитектуры рендеринга (рис. 10.3).

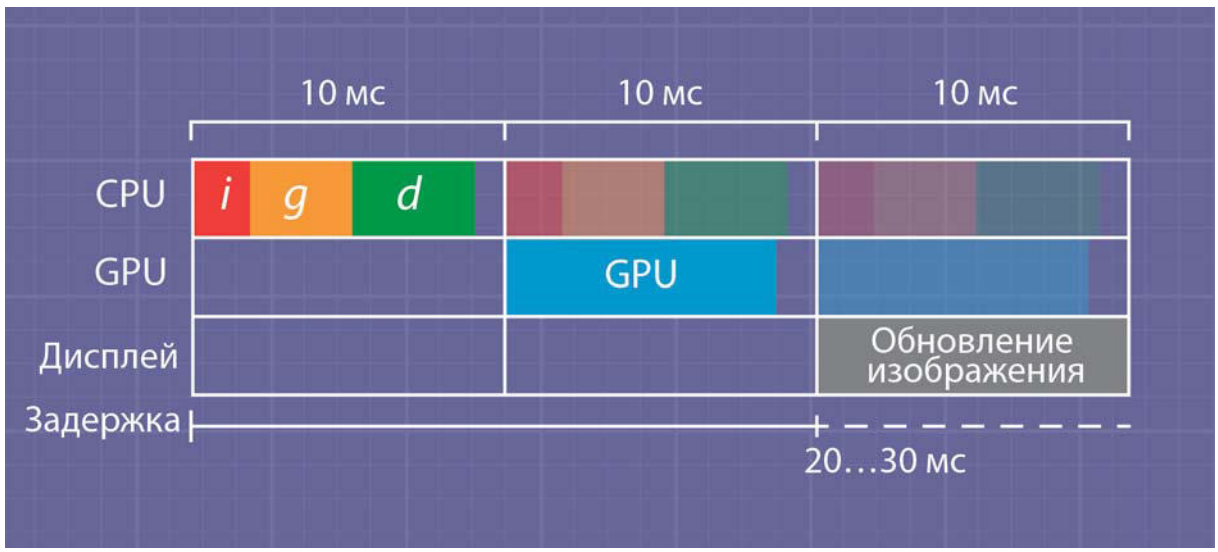

Рис. 10.3. Упрощенное представление конвейера рендеринга с двумя кадрами задержки и новой параллельной архитектурой; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

По сравнению с предыдущим рисунком, рис. 10.3 демонстрирует лучшую загрузку *CPU* и *GPU*; у них нет паузы в 10 мс, потому что графический процессор теперь отображает предыдущий кадр, в то время как процессор начинает работу над следующим кадром. Однако легко заметить увеличение задержки еще на целый кадр.

Другая проблема возникает, когда надоедливая задача вызова команд отрисовки начинает занимать слишком большую часть времени работы CPU и не помещается в 10 мс, отведённые на рендеринг сцены. К счастью, теперь у нас есть многоядерные процессоры, и если вы готовы смириться еще с одним кадром задержки (и на самом деле многие игровые движки идут на такой компромисс), вы можете запустить задачу на отдельном ядре, в то время как обработка пользовательского ввода и игровые вычисления выполняются на другом (рис. 10.4).

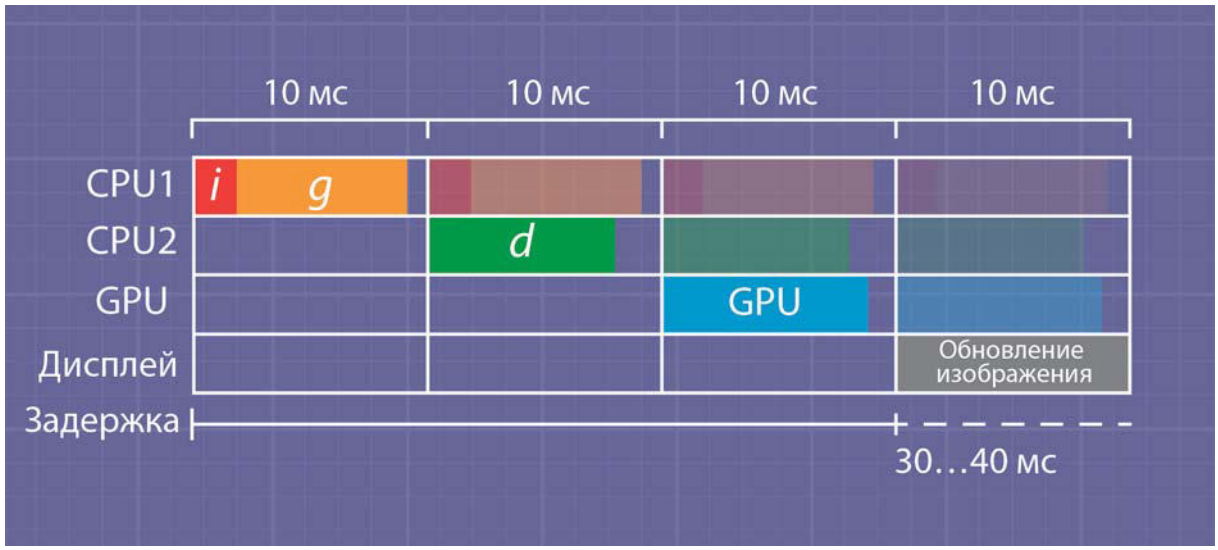

Рис. 10.4. Упрощенное представление конвейера рендеринга с тремя кадрами задержки и параллельной архитектурой с выделением кадра на задачи визуализации; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Однако оказывается, что можно пойти дальше. Вы можете уменьшить три кадра задержки (рис. 10.4) до двух, если *GPU* не будет дожидаться окончания вызова *GPU* всех команд отрисовки, а лишь предоставит центральному процессору достаточную фору, чтобы *CPU* мог начать подачу данных *GPU* с небольшой задержкой (хороший компромисс, чтобы отыграть обратно один кадр задержки; рис. 10.5).

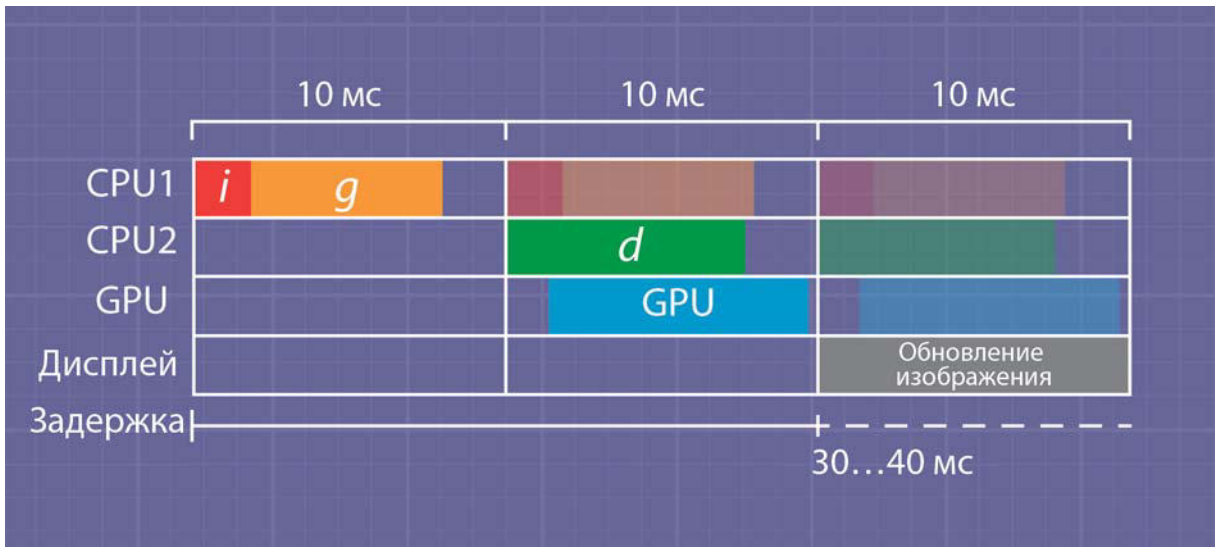

Рис. 10.5. Упрощенное представление конвейера рендеринга с тремя кадрами задержки и параллельной архитектурой с запуском отрисовки и *GPU* параллельно; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд

Рис. 10.5 демонстрирует удачный баланс между производительностью и задержкой. Можно ли сделать лучше? Конечно! Самые внимательные заметят пузырь бездействия, когда *GPU* ждет подачи вызовов отрисовки, и, возможно, задавались вопросом, как его уменьшить. Да — как *Oculus*, так и *Valve* справляются с тем, что они называют адаптивной очередью с опережающим запуском (*Adaptive Queue Ahead and Running Start*). В принципе с точки зрения рис. 10.5 они просто перемещают задачу рисования зеленого цвета до второго кадра, чтобы дать *GPU* наибольшее время обработки всех вызовов отрисовки.

Эти принципы позволяют значительно оптимизировать использование *CPU* и *GPU*. Можем ли мы уменьшить задержку ввода? Во-первых, обратите внимание, что входные данные опрашиваются в начале каждого кадра, поэтому даже в лучшем случае, без конвейерной обработки, все еще существует целый кадр задержки. Но почему вы не можете просто переместить опрос пользовательского ввода в конец кадра? Вы можете! Фактически это именно то, что *UE*4 делает с *Camera Component* и флагом *Lock to HMD* и *Motion Controller Component* с флагом *Low Latency Update*. Этот вид позднего обновления входных данных показан на рис. 10.6.

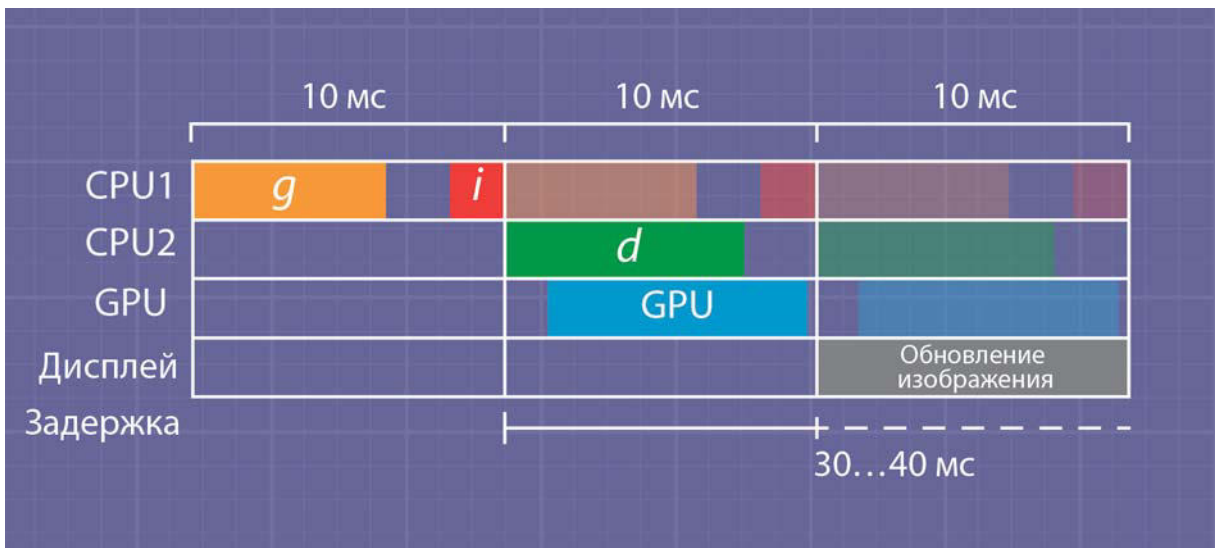

Рис. 10.6. Простая визуализация конвейера рендеринга с одним фреймом задержки и новой архитектурой с запуском отрисовки и *GPU* параллельно; *i* = ввод, *g* = игра, *d* = отрисовка. Каждая колонка — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Это выглядит лучше, но есть еще одна вещь, которую вы можете сделать: временной скачок [*timewarping*]. Основная идея временного скачка состоит в том, чтобы взять данные рендеринга из *GPU*, а затем преобразовать это изображение, чтобы имитировать поворот камеры на основе нового положения шлема. Как показано на рис. 10.7, теперь можно уменьшить задержку вращения шлема всего до 2 мс для первой части рендеринга. (Обратите внимание, что не все пакеты *SDK*/среды выполнения реализуют временной скачок, а те, которые реализуют, реализуют его по-разному. Помните об этих различиях. Также обратите внимание, что эта функция включается через различные среды выполнения, а не через движок, так что разработчику не нужно ничего делать, чтобы включить его.)

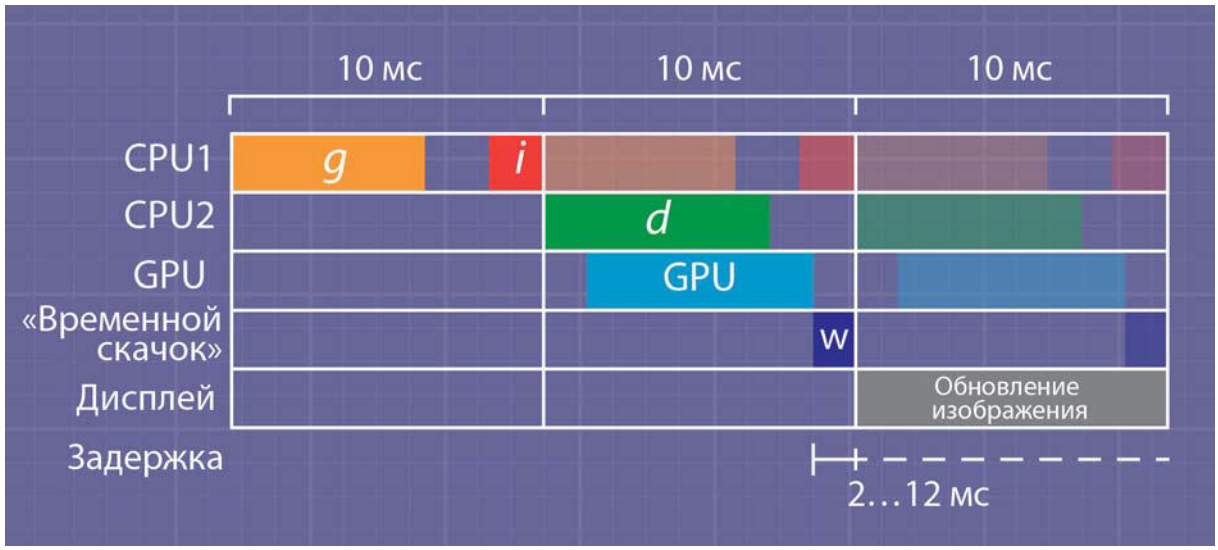

Рис. 10.7. Простая визуализация линии рендеринга с менее одного кадра ожидания из-за временного преобразования; *i* = ввод, *g* = игра, *d* = отрисовка. Каждая колонка — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Ранее я сказал, что *GPU* выводит отрендеренный кадр на дисплей, но это не совсем так. На самом деле то, что делает графический процессор, это отрисовка того, что называется задним буфером (*back buffer*), в то время как дисплей фактически читает из переднего (или кадрового) буфера (*front buffer*), непосредственно не затрагивает *GPU*. Это происходит из-за несоответствия того, как дисплей обновляется (помните, слева направо, сверху вниз) и как графический процессор отображает геометрию в сцене, а также механизма этого взаимодействия. Графический процессор не выполняет аккуратную отрисовку сверху и слева к вниз и направо, как это делает дисплей; вместо этого он отображает каждый вызов рисования, который он получает от процессора по очереди, один поверх другого, подобно художнику, накладывающему штрихи на полотно. Таким образом, вы сталкиваетесь с этой досадной проблемой при рендеринге непосредственно в буфер, из которого дисплей читает; то есть, если дисплей начинает показывать новый кадр, прежде чем вы закончите его рендеринг, он может фактически отображать на экране неполное изображение. Для решения этой проблемы существует механизм заднего буфера; графический процессор рендерит кадр в него, и когда дисплей готов к новому изображению, он меняет буфер, если кадр готов. Если кадр еще не готов, *GPU* просто отображает старое изображение снова, не показывая неполное изображение.

Почему это так важно? Оказалось, операция временного преобразования действительно создает изображение слева направо, сверху вниз, и на самом деле, когда временное преобразование действует, нет никакой реальной необходимости для этого двойной буферизации, потому что вы можете гарантировать, что вы начинаете преобразование с достаточным времени для буквально гонки с обновлением дисплея, пока он не закончил. Это то, что называется рендерингом переднего (кадрового) буфера или масштабированием; в настоящее время он реализован как на *Gear VR*, так и на *Daydream VR*, открывая широкие возможности. Поскольку вы искажаете изображение за миллисекунды до его отображения на экране, на самом деле существует вероятность того, что вы можете обновить входные данные до своей деформации по мере обновления дисплея, и на почти построчной (если вы достаточно хорошо его размечаете) основе вы можете деформировать последнюю информацию на дисплее. Именно таким образом *Gear VR* оказывает и выполняет два перекоса каждое обновление (по одному для каждого глаза), что показано на рис. 10.8.

Этот рендеринг переднего (кадрового) буфера включен по умолчанию на *Gear VR*, но он должен быть активирован вручную на *Daydreeam VR* (рис. 10.9). Если вы планируете использовать позиционный трекинг, выбирайте версию *Daydream* и добавляйте поддержку *Google Cardboard* в случае использования.

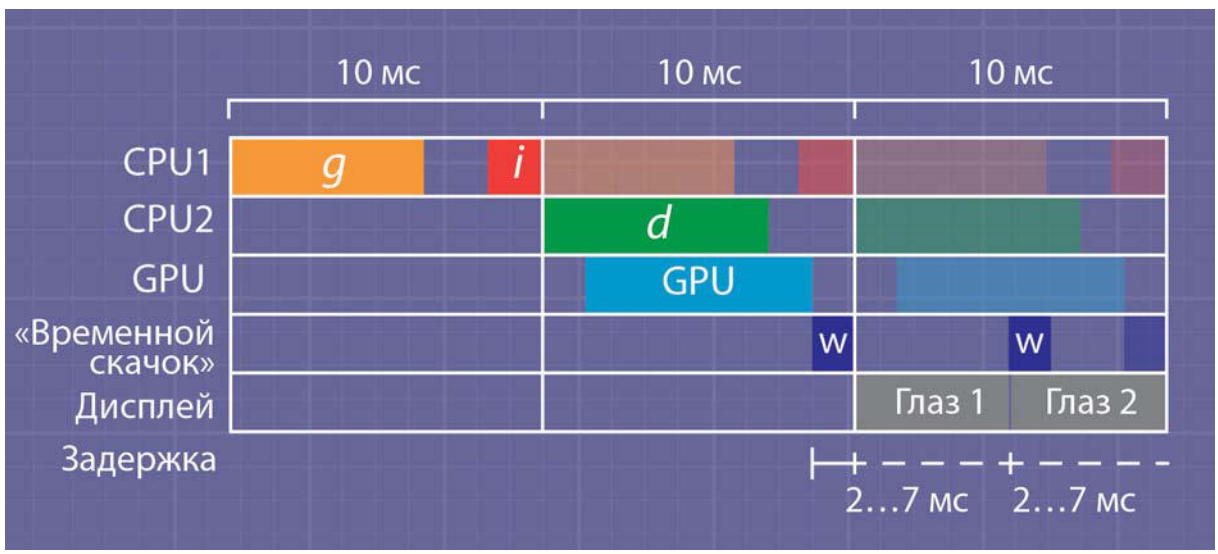

Рис. 10.8. Простая визуализация линии рендеринга с задержкой менее одного кадра из-за временного преобразования и рендеринга из переднего буфера; *i* = ввод, *g* = игра, *d* = отрисовка. Каждая колонка — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

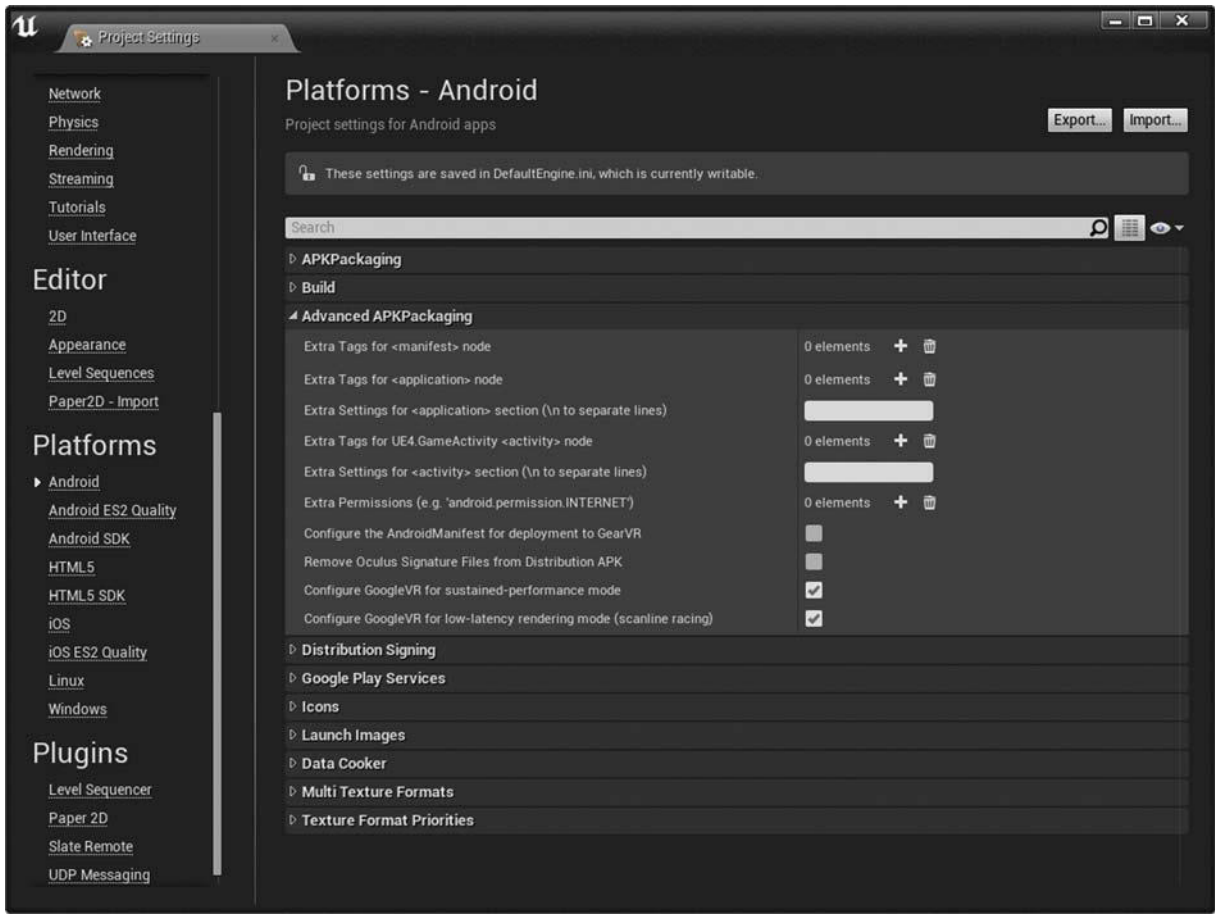

Рис. 10.9. Параметры конфигурации для включения масштабирования на *Daydream VR*

10.3. Повышение производительности

Другой столп методов рендеринга направлен на то, чтобы помочь вашему опыту *VR* визуализироваться («рендериться») за меньшее время или использовать меньше ресурсов для получения того же результата.

Эти оптимизации важны не только потому, что они позволяют вам отводить больше времени на рендеринг других аспектов вашей игры (например, теней или необычных материалов), которые могут сделать ваши миры более реалистичными, но и позволяют запускать *VR* на менее производительном оборудовании. Это может быть очень важно для мобильного *VR*, а также для снижения порога входа в *VR*.

Мы рассмотрели, сколько кадру требуется времени и как управлять механизмом отрисовки на *CPU* и *GPU*, чтобы уменьшить время ожидания. Давайте рассмотрим, как на самом деле визуализируется кадр, чтобы увидеть, какую производительность вы можете получить, просто изменив способ его рендеринга.