ГЛАВА 17

Запросы о преградах: зачем делать больше работы, чем требуется?

Бенджамин Липчак

ИЗ ЭТОЙ ГЛАВЫ ВЫ УЗНАЕТЕ . . .

Действие	Функция
Создание и удаление объектов запроса	`glGenQueries/glDeleteQueries`
Определение запросов о перекрывании ограничивающего многоугольника	`glBeginQuery/glEndQuery`
Извлечение результатов запроса о преградах	`glGetQueryObjectiv`

Сложные сцены содержат сотни объектов и многие тысячи многоугольников. Осмотрите комнату, в которой сейчас находитесь. Посмотрите на мебель, объекты, других людей или животных и подумайте, какие мощности нужно задействовать, чтобы точно визуализировать их сложный внешний вид. Возможно, кто-то сидит за компьютером на ящике в пустой студии, но большинство все же находится в среде, над точной визуализацией которой пришлось бы попотеть.

Подумайте теперь о вещах, которые вы не можете видеть: объекты, скрытые другими объектами, лежащие в ящиках или даже в соседней комнате. С большинства точек наблюдения эти объекты невидимы. Если вы будете визуализировать сцену, эти объекты будут нарисованы, но со временем что-то будет нарисовано поверх них. Так зачем тогда вообще делать работу впустую?

Так знайте же, существуют запросы о преградах (occlusion queries)! В этой главе мы опишем новую мощную функцию, включенную в OpenGL 1.5, с помощью которой можно существенно сэкономить на обработке вершин и наложении текстур за счет рисования небольших нетекстурных фрагментов. Часто подобный компромисс весьма выгоден. Ниже мы рассмотрим использование детектирования преград и докажем, что оно позволяет существенно увеличить частоту смены кадров.

Мир до запросов о преградах

Чтобы продемонстрировать увеличение производительности при использовании запросов о преградах, нужна экспериментальная контрольная группа. Для этого мы

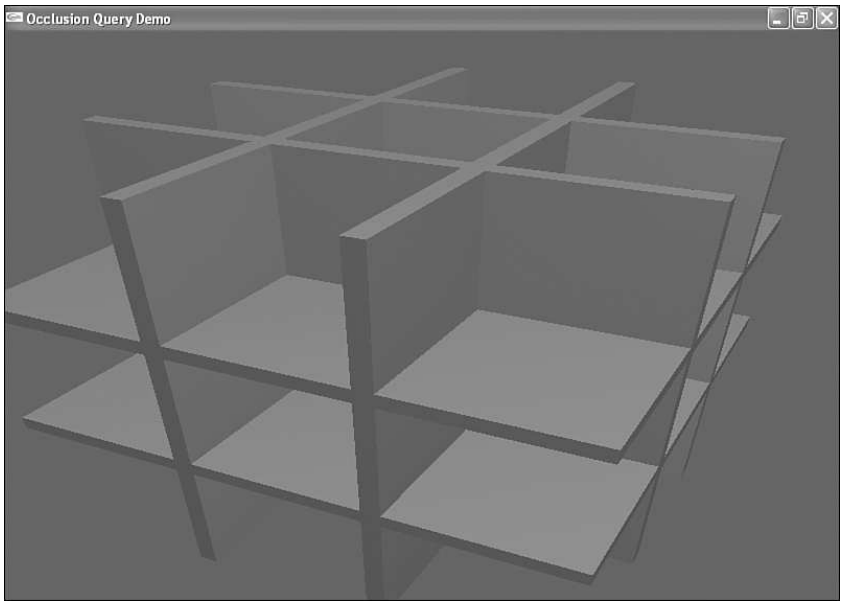

Рис. 17.1. Основной преградой является сетка, образованная шестью стенами

нарисуем сцену без детектирования преград. Выбранная сцена довольно сложна, в ней присутствует множество объектов, видимых и невидимых в различные моменты времени.

Вначале нарисуем "основную преграду". *Преградой* (occluder) будем называть большой объект на сцене, имеющий тенденцию закрывать (или скрывать) другие объекты сцены. Как правило преграда прорисована не очень детально, тогда как объекты, которые она закрывает, имеют существенно бо́льшую степень детализации. Хорошими примерами подобных преград являются стены, потолки и полы. В нашей сцене (рис. 17.1) основной преградой является решетка, образованная шестью стенками. Из листинга 17.1 видно, что эти сцены в действительности являются масштабированными кубами.

Листинг 17.1. Основная преграда и шесть масштабированных и транслированных сплошных кубов

```
// Вызывается для рисования закрывающей сетки
void DrawOccluder(void)
{
    glColor3f(0.5f, 0.25f, 0.0f);
    glPushMatrix();
    glScalef(30.0f, 30.0f, 1.0f);
    glTranslatef(0.0f, 0.0f, 50.0f);
    glutSolidCube(10.0f);
    glTranslatef(0.0f, 0.0f, -100.0f);
    glutSolidCube(10.0f);
    glPopMatrix();
```

```
    glPushMatrix();
    glScalef(1.0f, 30.0f, 30.0f);
    glTranslatef(50.0f, 0.0f, 0.0f);
    glutSolidCube(10.0f);
    glTranslatef(-100.0f, 0.0f, 0.0f);
    glutSolidCube(10.0f);
    glPopMatrix();
    glPushMatrix();
    glScalef(30.0f, 1.0f, 30.0f);
    glTranslatef(0.0f, 50.0f, 0.0f);
    glutSolidCube(10.0f);
    glTranslatef(0.0f, -100.0f, 0.0f);
    glutSolidCube(10.0f);
    glPopMatrix();
}
```

В каждой ячейке сетки мы собираемся поместить сферу со сложной мозаичной текстурой. Эти сферы являются "закрываемыми объектами" (occludees) — объектами, возможно закрываемыми преградой. Чтобы усилить нагрузку, требуется очень много вершин и текстуры, и на этом фоне выигрыш от использования запросов о преградах будет нагляднее. Поступим точно так же, как в предыдущей главе, где мы демонстрировали буферные объекты, — сделаем ставку на вершины!

На рис. 17.2 показано изображение, полученное при выполнении кода из листинга 17.2. Если объем работы показался вам чересчур большим, можете уменьшить степень мозаичности в функции `glutSolidSphere` с сотен до меньших величин. Если же, наоборот, ваша реализация OpenGL даже не почувствовала нагрузки, — сделайте мозаичное представление более подробным.

Листинг 17.2. Рисование в цветном кубе 27 сфер с детальным мозаичным представлением

```
// Вызывается для рисования сферы
void DrawSphere(GLint sphereNum)
{
    ...
    glutSolidSphere(50.0f, 100, 100);
    ...

}
void DrawModels(void)
{
    ...
    // Включаем текстурирование только для сфер
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    // Рисуем 27 сфер в цветном кубе
    for (r = 0; r < 3; r++)
    {
        for (g = 0; g < 3; g++)
        {
```

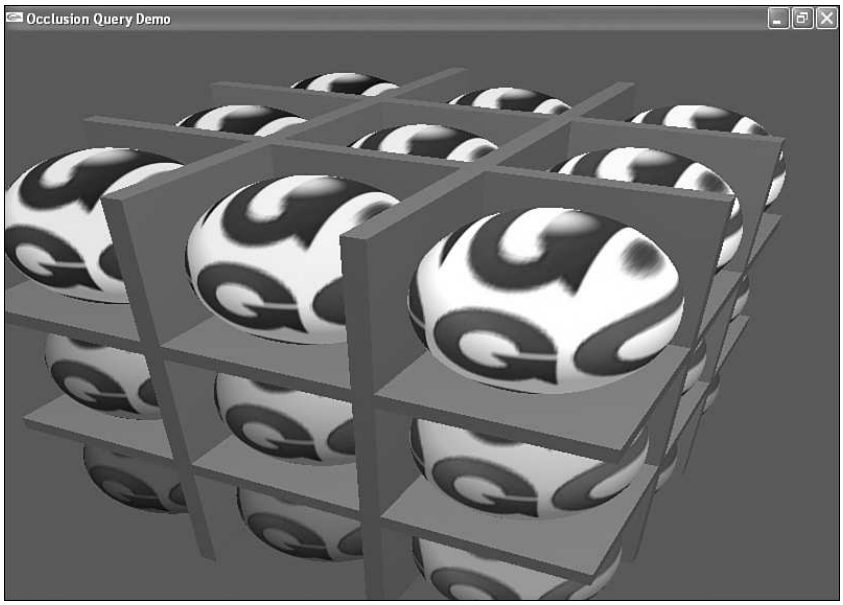

Рис. 17.2. В качестве закрываемых объектов выступают 27 детально прорисованных сфер

```
            for (b = 0; b < 3; b++)
            {
                glColor3f(r * 0.5f, g * 0.5f, b * 0.5f);
                glPushMatrix();
                glTranslatef(100.0f * r - 100.0f,
                100.0f * g - 100.0f,
                100.0f * b - 100.0f);
                DrawSphere((r*9)+(g*3)+b);
                glPopMatrix();
            }
        }
    }
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
}
```

Выполнение кода листинга 17.2 создает требуемое изображение. Если бы нас устраивала производительность визуализации, главу можно было бы завершить прямо сейчас. Однако при достаточно замысловатом мозаичном представлении сфер, при наложении сложной множественной текстуры или использовании для визуализации сферы затенения фрагментов частота смены кадров будет просто неприемлемой. Поэтому читайте дальше!

Ограничивающий объем					
Name	Тетраэдр	Куб	Октаэдр	Додекаэдр	Икосаэдр
Число граней	4	6	8	12	20
Число вершин	4	8	6	20	12
% избыточных пикселей	136%	120%	81%	58%	34%

Рис. 17.3. Различные ограничивающие объемы: достоинства и недостатки

Рамки

Теория метода детектирования затенения состоит в том, что, если ограничивающий объем (рамка) объекта не виден, сам объект также не виден. *Ограничивающим* (bounding) называется объем, полностью содержащий объект. Весь смысл детектирования закрываемых объектов заключается в рисовании просто ограничивающего объема, позволяющего определить, можно ли избежать рисования сложного объекта. Таким образом, чем сложнее ограничивающий объем, тем больше он не подходит для задачи оптимизации, которой мы пытаемся добиться.

Простейшим ограничивающим объемом является куб. Восемь вершин, шесть граней. Подобный ограничивающий объем легко создать для любого объекта, просто найдя его минимальную и максимальную координаты по осям x, y и z. Для наших сфер с радиусом 50 единиц идеально подойдет ограничивающий объем в виде куба со стороной 100 единиц.

Используя подобные простые и произвольные ограничивающие объемы, помните о компромиссе. Ограничивающий объем может иметь очень малое число вершин, но он будет захватывать гораздо большее число пикселей, чем исходный объект. Разместив в стратегических местах несколько дополнительных вершин, вы можете превратить ограничивающий объем в более полезную форму и существенно снизить служебные издержки, связанные с дополнительным рисованием. К счастью, ограничивающий объем рисуется без всякой текстуры или затенения, поэтому затраты на его рисование часто ниже, чем для исходного объекта. На рис. 17.3 приведен пример того, как различные формы ограничивающих объемов влияют на число пикселей и вершин.

При рисовании ограничивающих объемов мы собираемся активизировать запросы о преградах, которые подсчитают число фрагментов, прошедших проверку по глубине (а также сличение с трафаретом, если оно активизировано). Следовательно, для нас не важно, как выглядят ограничивающие объемы. Фактически мы даже можем не рисовать их на экране. Именно поэтому перед визуализацией ограничивающего объема мы избавляемся от всего ненужного, включая запись в буфере цвета.

```
glShadeModel(GL_FLAT);
glDisable(GL_LIGHTING);
```

```
glDisable(GL_COLOR_MATERIAL);
glDisable(GL_NORMALIZE);
// Текстурирование уже деактивизировано
...
glColorMask(0, 0, 0, 0);
```

Ну что ж, хватит разговоров, займемся собственно запросами. Вначале нам нужно определиться с их именами. В рассматриваемом примере мы затребовали 27 имен (по одному на каждый запрос о сфере) и предоставили указатель на массив, в котором должны храниться новые имена:

```
// Генерируем имена запросов о препятствии
glGenQueries(27, queryIDs);
```

Завершив работу, удаляем объекты запросов, указывая массив, содержащий эти 27 имен:

```
glDeleteQueries(27, queryIDs);
```

Объекты запросов о препятствии не связываются подобно другим объектам OpenGL (например, текстурным или буферным). Они создаются путем вызова функции `glBeginQuery`. Это действие отмечает начало запроса. Объект запроса имеет внутренний счетчик, отслеживающий число фрагментов, которые имеют шансы дойти до буфера кадров, — если мы не отключили маску записи буфера цвета. В начале запроса счетчик обнуляется.

Далее мы рисуем ограничивающий объем. Внутренний счетчик объекта запроса увеличивается на единицу всякий раз, когда фрагмент проходит проверку по глубине (следовательно, он не закрывается основной преградой — нарисованной ранее сеткой). Для некоторых алгоритмов надо точно знать число нарисованных фрагментов, но в нашем случае важно только знать, равен счетчик нулю (все фрагменты отброшены при проверке по глубине) или нет (какая-то часть ограничивающего объема видна).

Завершив рисование ограничивающего объема, отмечаем окончание запроса, вызывая функцию `glEndQuery`. Это сообщает OpenGL, что мы закончили запрос, и разрешает подавать следующий запрос или затребовать результаты этого. Поскольку мы рисуем 27 сфер, то производительность можно повысить, используя 27 различных объектов запроса. Таким образом, путем промежуточного считывания результатов запросов мы можем, не нарушая конвейер, выстроить в очередь рисование всех 27 ограничивающих объемов.

В листинге 17.3 иллюстрируется визуализация ограничивающих объемов, обособленных началом и концом запроса. Затем мы перерисовываем основную преграду и рисуем нужные сферы. Обратите внимание на код визуализации ограничивающего объема там, где мы оставляем активизированной маску записи буфера цвета. Таким образом можно увидеть и сравнить различные формы ограничивающих объемов.

Листинг 17.3. Начало запроса, рисование ограничивающего объема, завершение запроса, перерисовывание на реальной сцене

```
// Вызывается для рисования объектов сцены
void DrawModels(void)
{
    GLint r, g, b;
    if (occlusionDetection || showBoundingVolume)
```

```
{
    // Рисуем ограничивающие объемы после рисования основной
    // преграды
    DrawOccluder();
    // Все, что нам нужно знать об ограничивающем объеме, -
    // получающиеся коды глубины
    glShadeModel(GL_FLAT);
    glDisable(GL_LIGHTING);
    glDisable(GL_COLOR_MATERIAL);
    glDisable(GL_NORMALIZE);
    // Текстурирование уже деактивизировано
    if (!showBoundingVolume)
        glColorMask(0, 0, 0, 0);
    // Рисуем 27 сфер в цветном кубе
    for (r = 0; r < 3; r++)
    {
        for (g = 0; g < 3; g++)
        {
            for (b = 0; b < 3; b++)
            {
            if (showBoundingVolume)
            glColor3f(r * 0.5f, g * 0.5f, b * 0.5f);
            glPushMatrix();
            glTranslatef(100.0f * r - 100.0f,
                         100.0f * g - 100.0f,
                         100.0f * b - 100.0f);
            glBeginQuery(GL_SAMPLES_PASSED,
                         queryIDs[(r*9)+(g*3)+b]);
            switch (boundingVolume)
            {
                case 0:
                glutSolidCube(100.0f);
                break;
                case 1:
                glScalef(150.0f, 150.0f, 150.0f);
                glutSolidTetrahedron();
                break;
                case 2:
                glScalef(90.0f, 90.0f, 90.0f);
                glutSolidOctahedron();
                break;
                case 3:
                glScalef(40.0f, 40.0f, 40.0f);
                glutSolidDodecahedron();
                break;
                case 4:
                glScalef(65.0f, 65.0f, 65.0f);
                glutSolidIcosahedron();
                break;
            }
            glEndQuery(GL_SAMPLES_PASSED);
            glPopMatrix();
            }
```

```
            }
        }
        if (!showBoundingVolume)
        glClear(GL_DEPTH_BUFFER_BIT);

        // Восстанавливаем нормальное состояние рисования
        glShadeModel(GL_SMOOTH);
        glEnable(GL_LIGHTING);
        glEnable(GL_COLOR_MATERIAL);
        glEnable(GL_NORMALIZE);
        glColorMask(1, 1, 1, 1);
    }
    DrawOccluder();
    // Включаем текстурирование только для сфер
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    // Рисуем 27 сфер в цветном кубе
    for (r = 0; r < 3; r++)
        {
        for (g = 0; g < 3; g++)
            {
            for (b = 0; b < 3; b++)
                {
                glColor3f(r * 0.5f, g * 0.5f, b * 0.5f);
                glPushMatrix();
                glTranslatef(100.0f * r - 100.0f,
                             100.0f * g - 100.0f,
                             100.0f * b - 100.0f);
                DrawSphere((r*9)+(g*3)+b);
                glPopMatrix();
                }
            }
        }
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
}
```

Функция `DrawSphere` содержит логику, согласно которой мы решаем, рисовать ли сферу. Результаты запросов содержатся в 27 объектах запросов. Ну что ж, давайте найдем, какие объекты скрыты, а какие придется рисовать.

Запрос объекта запроса

Момент истины близок. Присяжные вернулись, чтобы огласить вердикт. Мы хотим нарисовать только необходимое, поэтому надеемся, что все запросы дадут ответ “ни один из фрагментов не виден”. Разумеется, если говорить о поставленной задаче, такой ответ невозможен.

Вне зависимости от того, под каким углом мы смотрим на сетку, если не увеличивать масштаб, в поле зрения попадет минимум 9 сфер. В худшем случае вы увидите все сферы на трех гранях сетки: 19 штук. Тем не менее даже в худшем случае не требуется рисовать 8 сфер. Т.е. в пересчете на вершины получаем почти 30% экономии. В лучшем же случае экономия составляет 66%, так как 18 сфер отбрасываются. Если увеличить масштаб, сфокусировав взгляд на одной сфере, мы можем добиться невероятной экономии, не рисуя 26 сфер!

Итак, как определить свою удачу? Нужно просто запросить объект запроса. Это звучит немного путано, но речь идет об обычном и понятном запросе состояния OpenGL. Просто так получилось, что в нем фигурирует нечто, названное объектом запроса. В функции `glGetQueryObjectiv` (листинг 17.4) мы смотрим, равен ли нулю счетчик фрагментов, прошедших проверку по глубине, и если да — сферы не рисуем.

Листинг 17.4. Проверка результатов запроса и рисование сферы только тогда, когда это необходимо

```
// Вызывается для рисования сферы
void DrawSphere(GLint sphereNum)
    {
    GLboolean occluded = GL_FALSE;
    if (occlusionDetection)
        {
        GLint passingSamples;
        // Проверяем, скроется ли сфера за преградой
        glGetQueryObjectiv(queryIDs[sphereNum], GL_QUERY_RESULT,
                           &passingSamples);
        if (passingSamples == 0)
        occluded = GL_TRUE;
        }
    if (!occluded)
        {
        glutSolidSphere(50.0f, 100, 100);
        }
    }
```

Вот и все. Последовательно проверяются запросы по всем сферам и определяется, требуется ли рисовать рассматриваемую сферу. В программу включен режим деактивизации детектирования преград, чтобы можно было посмотреть, насколько при этом падает производительность. В зависимости от того, сколько сфер видно, детектирование преград повышает производительность в два или более раз.

Можно также проверить, доступен ли результат немедленно. Если мы не будем визуализировать 27 ограничивающих объемов, а затребуем каждый результат немедленно, операция визуализации ограничивающего объема может еще находиться в конвейере и результат может быть не готов. Чтобы проверить, готов ли результат, можно запросить функцию `GL_QUERY_RESULT_AVAILABLE`. Если он не готов, запрос `GL_QUERY_RESULT` будет остановлен до получения результата. Поэтому в ожидании результата можно указать приложению сделать что-то полезное. Мы запланировали на промежуточные моменты массу работы, направленной на то, чтобы к моменту окончания 27-го запроса был готов результат первого запроса.

Другие запросы состояния включают запрос о текущем активном имени запроса (какой запрос активен между парой `glBeginQuery`/`glEndQuery`, если такой существует) и числе битов в счетчике прошедших элементов (зависит от реализации). В принципе реализация может иметь счетчик с нулевым числом битов. В этом случае запросы о преградах бесполезны, и их не следует использовать. В листинге 17.5 показано, как данная возможность проверяется при инициализации приложения (непосредственно после проверки расширения и точек входа).

Листинг 17.5. Проверка доступности запросов о преградах

```
// Убеждаемся, что доступны требуемые функциональные возможности!
version = glGetString(GL_VERSION);
if ((version[0] == '1') && (version[1] == '.') &&
    (version[2] >= '5') && (version[2] <= '9'))
{
    glVersion15 = GL_TRUE;
}
if (!glVersion15 &&  !gltIsExtSupported("GL_ARB_occlusion_query"))
{
fprintf(stderr, "Neither OpenGL 1.5 nor GL_ARB_occlusion_query"
                " extension is available!\n");
Sleep(2000);
exit(0);
}
    // Загружаем указатели на функции
    if (glVersion15)
{
    glBeginQuery = gltGetExtensionPointer("glBeginQuery");
    glDeleteQueries = gltGetExtensionPointer("glDeleteQueries");
    glEndQuery = gltGetExtensionPointer("glEndQuery");
    glGenQueries = gltGetExtensionPointer("glGenQueries");
    glGetQueryiv = gltGetExtensionPointer("glGetQueryiv");
    glGetQueryObjectiv =
        gltGetExtensionPointer("glGetQueryObjectiv");
    glGetQueryObjectuiv =
        gltGetExtensionPointer("glGetQueryObjectuiv");
    glIsQuery = gltGetExtensionPointer("glIsQuery");
}
else
{
    glBeginQuery = gltGetExtensionPointer("glBeginQueryARB");
    glDeleteQueries = gltGetExtensionPointer("glDeleteQueriesARB");
    glEndQuery = gltGetExtensionPointer("glEndQueryARB");
    glGenQueries = gltGetExtensionPointer("glGenQueriesARB");
    glGetQueryiv = gltGetExtensionPointer("glGetQueryivARB");
    glGetQueryObjectiv =
        gltGetExtensionPointer("glGetQueryObjectivARB");
    glGetQueryObjectuiv =
        gltGetExtensionPointer("glGetQueryObjectuivARB");
    glIsQuery = gltGetExtensionPointer("glIsQueryARB");
}
if (!glBeginQuery || !glDeleteQueries || !glEndQuery ||
    !glGenQueries || !glGetQueryiv || !glGetQueryObjectiv ||
```

```
    !glGetQueryObjectuiv || !glIsQuery)
{
    fprintf(stderr, "Not all entrypoints were available!\n");
    Sleep(2000);
    exit(0);
}
// Убеждаемся, что число битов счетчика запросов не равно нулю
glGetQueryiv(GL_SAMPLES_PASSED, GL_QUERY_COUNTER_BITS,
             &queryCounterBits);
if (queryCounterBits == 0)
{
    fprintf(stderr, "Occlusion queries not really supported!\n");
    fprintf(stderr, "Available query counter bits: 0\n");
    Sleep(2000);
    exit(0);
}
```

Еще одним запросом, о котором вам следует знать, является `glIsQuery`. Эта команда просто проверяет, принадлежит ли заданное имя существующему объекту запроса (в таком случае возвращается `GL_TRUE`, в противном — `GL_FALSE`).

Резюме

Иногда при визуализации сложных сцен мы растрачиваем аппаратные ресурсы на визуализацию объектов, которые никогда не будут видны. Дополнительной работы можно избежать, проверяя, какие объекты проявятся на окончательном изображении. Рисуя вокруг объекта ограничивающий объем, мы быстро аппроксимируем объект на сцене. Если преграды на сцене закроют ограничивающий объем, они спрячут и реальный объект. Облекая визуализацию ограничивающего объема в запрос, мы можем подсчитать число пикселей, которые он занимал бы. Если число пикселей равно нулю, мы можем гарантировать, что исходный объект также не будет нарисован, поэтому его визуализацию можно пропустить. В зависимости от сложности объектов на сцене и того, насколько часто они закрываются, повышение производительности может быть невероятным.

Справочная информация

glBeginQuery

Цель:	Отметить начало запроса о преградах
Включаемый файл:	`<glext.h>`
Синтаксис:	`void glBeginQuery(GLenum target, GLuint id);`
Описание:	Начинает запрос о преградах, имеющий указанное имя, обнуляя счетчик прошедших выборок. Если объект запроса не существует, он создается. Если запрос с указанным именем уже обрабатывается или задано нулевое имя объекта запроса, генерируется ошибка

Параметры:	
target (тип GLenum)	Тип объекта запроса. Значение должно быть равно GL_SAMPLES_PASSED
id (тип GLuint)	Имя объекта запроса
Что возвращает:	Ничего
См. также:	glEndQuery, glGenQueries, glDeleteQueries, glIsQuery

glDeleteQueries

Цель:	Удалить один или несколько объектов запроса
Включаемый файл:	<glext.h>
Синтаксис:	void glDeleteQueries(GLsizei *n*, const GLuint **ids*);
Описание:	Удаляет объекты запроса. При этом содержимое удаляется, а имена помечаются как неиспользуемые. Если на удаление заявлено имя неиспользуемого объекта запроса или нулевое, оно просто игнорируется
Параметры:	
n (тип GLsizei)	Число удаляемых объектов запроса
ids (тип const GLuint*)	Указатель на массив, содержащий имена удаляемых объектов запроса
Что возвращает:	Ничего
См. также:	glBeginQuery, glEndQuery, glGenQueries, glIsQuery

glEndQuery

Цель:	Отметить конец запроса о преградах
Включаемый файл:	<glext.h>
Синтаксис:	void glEndQuery(GLenum *target*);
Описание:	Помечает конец текущего запроса о преградах. Если никакой запрос о преградах в настоящее время не обрабатывается, будет сгенерирована ошибка
Параметры:	
target (тип GLenum)	Тип завершаемого объекта запроса. Значение должно быть равно GL_SAMPLES_PASSED
Что возвращает:	Ничего
См. также:	glBeginQuery, glDeleteQueries, glQueryObjectiv, glGetQueryObjectuiv

glGenQueries

Цель:	Вернуть неиспользуемые имена объектов запроса
Включаемый файл:	`<glext.h>`
Синтаксис:	`void glGenQueries(GLsizei` *`n`*`, const GLuint *`*`ids`*`);`
Описание:	Возвращает неиспользуемые имена объектов запроса. Впоследствии с помощью функции `glBeginQuery` можно создавать и запускать объекты запроса
Параметры:	
`n` (тип `GLsizei`)	Число возвращаемых имен объектов запроса
`ids` (тип `GLuint*`)	Указатель на массив, заполняемый неиспользуемыми именами объектов запроса
Что возвращает:	Ничего
См. также:	`glDeleteQueries`, `glBeginQuery`, `gllsQuery`

glGetQueryiv

Цель:	Запросить состояние цели — объекта запроса
Включаемый файл:	`<glext.h>`
Синтаксис:	`void glGetQueryiv(GLenum` *`target`*`, GLenum` *`pname`*`, GLint *`*`params`*`);`
Описание:	Запрашивает состояние заданного объекта запроса. Данное состояние характеризует не конкретный объект запроса, а все объекты запроса
Параметры:	
`target` (тип `GLenum`)	Тип запрашиваемого объекта запроса. Значение должно быть равно `GL_SAMPLES_PASSED`
`pname` (тип `GLenum`)	Запрашиваемое состояние объекта запроса. Значением может быть одна из следующих констант: `GL_CURRENT_QUERY`: имя активного в настоящее время объекта запроса о преградах. Если активных объектов запроса о преградах нет, возвращается нуль `GL_QUERY_COUNTER_BITS`: зависящее от реализации число битов в счетчике, используемом для накопления прошедших выборок
`params` (тип `GLint*`)	Указатель на положение в памяти, где будут записаны результаты запроса
Что возвращает:	Ничего
См. также:	`glGetQueryObjectiv`, `glGetQueryObjectuiv`, `glIsQuery`

glGetQueryObject

Цель:	Запросить состояние объекта запроса
Включаемый файл:	`<glext.h>`
Синтаксис:	`void glGetQueryObjectiv(GLuint id, GLenum pname, GLint *params);` `void glGetQueryObjectuiv(GLuint id, GLenum pname, GLuint *params);`
Описание:	Запрашивает состояние объекта запроса с заданным именем. Если имя не соответствует существующему объекту запроса, или объект запроса в настоящее время активен (находится внутри `glBeginQuery`/`glEndQuery`), выдается ошибка
Параметры:	
id (тип `GLuint`)	Имя объекта запроса, состояние которого запрашивается
pname (тип `GLenum`)	Запрашиваемое состояние объекта запроса. Значением может быть одна из следующих констант: `GL_QUERY_RESULT`: указывает значение счетчика прошедших выборок объекта запроса `GL_QUERY_RESULT_AVAILABLE`: указывает, доступен ли результат сразу. Если результат запроса будет доступен через некоторое время, возвращается значение `GL_FALSE`. В противном случае возвращается значение `GL_TRUE`, которое также указывает, что доступны и результаты всех предыдущих запросов
params (тип `GLint*/GLuint*`)	Указатель на ячейку памяти, куда записываются результаты запроса
Что возвращает:	Ничего
См. также:	`glBeginQuery`, `glEndQuery`, `glGetQueryiv`, `glIsQuery`

glIsQuery

Цель:	Запросить, соответствует ли данное имя объекту запроса
Включаемый файл:	`<glext.h>`
Синтаксис:	`GLboolean glIsQuery(GLuint id);`
Описание:	Запрашивает, является ли указанное имя именем объекта запроса
Параметры:	
id (тип `GLuint`)	Искомое имя объекта запроса
Что возвращает:	Если объект запроса с искомым именем был создан (с помощью `glBeginQuery`) и не был удален, возвращается значение `GL_TRUE` (тип `GLboolean`). В противном случае возвращается значение `GL_FALSE`
См. также:	`glBeginQuery`, `glEndQuery`, `glGenQueries`, `glDeleteQueries`