УДК 004.925

Методы оптимизации под OPENGL

© И.В. Дикан, Ю.С. Белов

КФ МГТУ им. Н.Э. Баумана, Калуга, 248000, Россия

Рассмотрены основные методы оптимизации графических приложений, написанных с использованием OpenGL. Описаны методы низкоуровневой оптимизации, в том числе использующие различные расширения OpenGL. Приведены улучшенные методы отрисовки графических объектов. Изложены методы изменения формата графических объектов для повышения производительности конечных приложений.

Ключевые слова: *VBO, VAO, OpenGL, 3D-графика, DXT, оптимизация, шейдеры, MipMapping, Occlusion Query, Frustum Culling.*

Оптимизация рендеринга является важным этапом разработки конечного приложения, поскольку позволяет увеличить число кадров в секунду, снизить требования к аппаратному обеспечению в приложениях реального времени, а для рендеров — уменьшить время, затрачиваемое на отрисовку конечной сцены. При нарушении принципов оптимизации мы можем получить низкую производительность даже на самых простых сценах, поэтому не стоит пренебрегать ими.

Большинство способов оптимизации основано на использовании специализированных расширений драйвера, о которых многие разработчики забывают или вовсе не хотят знать. Однако одними расширениями не обойтись — спектр проблем, которые необходимо решить, слишком велик и затрагивает множество подсистем рендеринга. При этом стоит учитывать, что увеличение производительности от использования одного или нескольких методов может быть минимально, а может достигать 100 % и более. Все зависит от особенностей приложения.

Рассмотрим основные методы оптимизации приложений: низкоуровневые, рисования моделей и оптимизации моделей.

Низкоуровневые оптимизации. Использование старых функций OpenGL, например Begin-End, серьезно уменьшает производительность, поскольку они имеют большие накладные расходы. К тому же в новых версиях OpenGL эти функции объявлены как deprecated. От них следует отказаться в пользу новых расширений типа ARB_vertex_buffer_object (VBO), позволяющих хранить вертексы в одном буфере в памяти GPU. Следующий пример демонстрирует разницу старого и нового подходов:

```
struct Vertex {
   float x, y, z;
}
```

```
void drawOldWay(ref const Vertex[] vertices) { // ста-
рый стиль отрисовки
   Begin(GL_TRIANGLES);
      foreach(v; vertices)
// все вертексы каждый раз передаются
// из оперативной памяти в память GPU
         Vertex3f(v.x, v.y, v.z);
   End();
}
uint createVBO(ref const Vertex[] vertices) {// функ-
ция создания VBO
   uint id;
   // создаётся буфер
   GenBuffers(1, &id);
// привязывается к текущему контексту
BindBuffer(GL_ARRAY_BUFFER, id);
   // данные загружаются в быструю память GPU
BufferData(GL_ARRAY_BUFFER,
Vertex.sizeof   *   vertices.length,   vertices.ptr,
GL_STATIC_DRAW);
   return id;
}
void bindVBO(uint id) {
   // включается нулевой атрибут,
// по которому обычно располагаются координаты
EnableVertexAttribArray(0);
BindBuffer(GL_ARRAY_BUFFER, id);
// OpenGL уведомляется о содержимом буфера
VertexAttribPointer(
      0,       // номер атрибута (обычно это 0)
      3,       // количество элементов
      GL_FLOAT,// тип элементов
      GL_FALSE,// элементы не нуждаются в нормализации
      0,    // все элементы идут друг за другом
      null     // базовое смещение отсутствует
);
}
void unbindVBO() { DisableVertexAttribArray(0); }
void drawNewWay(uint id, uint count) { // отрисовка с
использованием VBO
   bindVBO(); // прикрепление VBO
DrawArrays(GL_TRIANGLES, 0, count); // отрисовка тре-
угольников
// отвязка VBO, так как оно более не используется
unbindVBO();
}
```

Если набор буферов при рисовании с использованием VBO постоянен, то лучше применять ARB_vertex_array_object (VAO). По аналогии с display lists VAO позволяет «записать» последовательность команд для рендеринга и затем «воспроизвести» их, т. е.

уменьшить накладные расходы. Приведенный ниже пример демонстрирует это:

```
uint createVAO(uint vboId) {
uint id;
   GenVertexArrays(1, &id);
// с этого момента OpenGL будет «записывать» всю
// последовательность изменений конвейера
   BindVertexArray(id);
   bindVBO(vboId); // привязка VBO
   BindVertexArray(0); // окончание «записи»
   unbindVBO();
   return id;
}
void drawVAO(uint id, uint count) { // VAO дает воз-
можность использовать
// всего три команды
   BindVertexArray(id);
   DrawArrays(GL_TRIANGLES, 0, count);
   BindVertexArray(0); // отвязка VAO
}
```

Текстуры в OpenGL состоят из изображения, мип-уровней (mip-level) и параметров обработки самой текстуры (режимы фильтрации, отсечения текстурных координат и др.). Изменение параметров обработки может быть затратным. Возникают ситуации, когда набор таких параметров одинаков для множества текстур и задание их для каждой текстуры может быть неэффективным. В этом случае необходимо использовать расширение ARB_sampler_objects, поддерживающее сэмплеры (sampler object), которые позволяют хранить параметры отдельно от текстуры, т. е. при создании текстуры ей не требуется вновь указывать различные параметры. Сэмплеры привязаны к текстурным блокам (texture unit), поэтому один сэмплер можно использовать с несколькими текстурами.

Приведем пример. Вариант выполнения с каждой текстурой:

```
BindTexture(TEXTURE_2D, tid);
TexParameteri(TEXTURE_2D, TEXTURE_MAG_FILTER, LINEAR);
```

Способ, когда достаточно установить значение сэмплера один раз:

```
SamplerParameteri(sid, TEXTURE_MAG_FILTER, LINEAR);
```

Теперь можно легко привязать сэмплер к текстурному блоку, и он будет использоваться для каждой текстуры:

```
BindSampler(texUnitNum, sid);
```

Часто бывает нужно изменить какой-либо параметр у шейдера, текстуры или другого объекта OpenGL (иногда без изменения внутреннего состояния конвейера). Для этого придется получать текущий идентификатор, привязывать новый, выполнять какие-либо действия, потом привязывать старый. Все это увеличивает накладные расходы, что отрицательно сказывается на производительности. Поэтому необходимо использовать расширение EXT_direct_state_access, которое предоставляет bindless интерфейс для изменения объектов. Это расширение уменьшает накладные расходы и упрощает код, предоставляет функции для работы со всеми объектами OpenGL, его использование всегда желательно. Например, чтобы изменить значение uniform-переменной потребуется сделать следующее:

```
uint oldSid;
GetIntegerv(CURRENT_PROGRAM, &oldSid);
UseProgram(sid);
Uniform1f(loc, value);
UseProgram(oldSid);
```

а благодаря DSA:

```
ProgramUniform1f(sid, loc, value);
```

Желательно избегать использования функций многих аргументов, так как это может накладывать дополнительные накладные расходы. Например, для передачи vec4 в шейдер лучше применять Uniform4fv вместо Uniform4f, если, конечно, координаты вектора хранятся в пользовательских типах последовательно, что позволяет просто получить указатель на них.

Изменение состояния конвейера — операция не из «дешевых». Например, если необходимо применить одну текстуру к двум разным моделям, то лучше эти модели рисовать последовательно, чтобы было как можно меньше команд привязки (BindTexture). Это же относится и к остальным объектам OpenGL: шейдерам, сэмплерам, VBO, VAO и др.

Неправильно:

```
text1.bind();
mesh1.draw();
text2.bind();
mesh2.draw();
text1.bind();
mesh3.draw();
```

Правильно:

```
text1.bind();
```

```
mesh1.draw();
mesh3.draw();
text2.bind();
mesh2.draw();
```

Оптимизации рисования моделей. Использование фиксированного конвейера (fixed pipeline) может серьезно сказаться на производительности, поскольку фиксированный конвейер содержит в себе реализацию возможностей, которые часто бывают просто не нужны (например, свет, туман и др.). В этом случае лучше подходит программируемый конвейер (programmable pipeline) с использованием шейдеров, которые обеспечивают гибкую реализацию всего необходимого без ущерба для производительности [1]. Также фиксированный конвейер в новых версиях OpenGL объявлен как deprecated. В общем случае при инициализации приложения достаточно загрузить и привязать шейдер. Все последующие команды рисования будут передавать данные именно ему. Приведем пример, демонстрирующий загрузку и использование шейдеров:

```
// функция проверки состояния шейдера или линковки
void checkStatus(bool isLinking = false)(uint id,
string name) {
int status;
// сообщение компиляции
char str[4096];
// получение статуса шейдера
static if(isLinking)
      GetProgramiv(id, LINK_STATUS, &status);
else
      GetShaderiv(id, COMPILE_STATUS, &status);
   // получение информационного сообщения
static if(isLinking)
    GetProgramInfoLog(id, str.sizeof, null, str.ptr);
else
      GetShaderInfoLog(id, str.sizeof, null, str.ptr);
auto s = str.ptr.to!string;
if(status) { // в случае успеха статус != 0
      if(s.length)
         writeLog(`%s: %s`, name, s);
      return;
}
enum act = isLinking ? `linking` : `compiling`;
// в случае неуспеха — генерация исключения
throwError("%s %s failed: %s\n", act, name, s);
}
…
```

```
// объявление вертексного и фрагментного шейдера
string vs = `#version 330
// переменная для передачи текстурных координат в
фрагментный шейдер
out vec2 fs_TexCoord;
layout(location = 0) in vec3 vs_Position; // координа-
ты
layout(location = 1) in vec2 vs_TexCoord; // текстур-
ные координаты
uniform mat4 vs_Model; // матрица модели
uniform mat4 vs_ViewProj; // матрица камеры вместе с
проекцией
void main() {
// записываем текстурные координаты для фрагментного
шейдера
fs_TexCoord = vs_TexCoord;
     vec4 v = vs_Model * vec4(vs_Position, 1.0); //
     трансформируем вершину
gl_Position = vs_ViewProj * v; // записываем результат
}
`, fs = `#version 330
in vec2 fs_TexCoord;
uniform sampler2D fs_Texture;
void main() {
gl_FragColor = texture(fs_Texture, fs_TexCoord); //
выборка из текстуры
}
`;
// создание шейдерной программы
auto prog = CreateProgram();
// проверка на ошибки
prog || throwError(`can't create shader program`);
// очистка ресурсов в случае неудачи
scope(failure)
DeleteProgram(prog)
// создание вертексного шейдера
auto vs = CreateShader(VERTEX_SHADER);
vs || throwError(`can't create vertex shader`);
// удаление шейдера после успешного выхода из функции
scope(exit)
   DeleteShader(vs);
// создание фрагментного шейдера
auto fs = CreateShader(FRAGMENT_SHADER);
fs || throwError(`can't create fragment shader`);
scope(exit)
   DeleteShader(fs);
// загрузка исходного кода
ShaderSource(vs, 1, vs.ptr, null);
// компиляция
```

```
CompileShader(vs);
// проверка статуса вертексного шейдера
checkStatus(vs, `vertex shader`);
// теперь то же самое, но с фрагментным шейдером
ShaderSource(fs, 1, fs.ptr, null);
CompileShader(fs);
checkStatus(fs, `fragment shader`);
// привязываем шейдеры к шейдерной программе
gl AttachShader(prog, vs);
gl AttachShader(prog, fs);
// линковка программы
gl LinkProgram(prog);
// проверка линковки
checkStatus!true(prog, `shader program`);
// привязка шейдера к контексту
UseProgram(prog);
```

Теперь при рисовании достаточно установить матрицы объектов:

```
auto loc = GetUniformLocation("vs_Model");
UniformMatrix4fv(loc, 1, FALSE, matrix.ptr);
```

Многие объекты на сцене достаточно удалены от камеры, поэтому имеют малые размеры. Использовать высокодетализированные текстуры вместе с ними не имеет смысла, так как может быть видно всего лишь несколько пикселов из всей текстуры, а это ведет к увеличению кэш-промахов и падению производительности. В OpenGL присутствует техника под названием мипмаппинг (mipmapping), суть которой состоит в том, что для исходной текстуры создаются уменьшенные копии, каждая из которой в 4 раза меньше предыдущей. Копии создаются до тех пор, пока не получится текстура размером в 1×1 пиксел. Во время рендеринга определяется расстояние до объекта, в зависимости от которого выбирается подходящая текстура (мип-уровень). Это позволяет не использовать высокую детализацию там, где в этом нет смысла, а значит, повысить производительность. Недостатком подхода является увеличение расходов памяти на 1/3. Но для современных видеокарт с их объемами памяти это не должно быть проблемой. Сгенерировать мип-уровни можно следующим способом:

```
BindTexture(TEXTURE_2D, tid); // привязываем текстуру
// заставляем OpenGL сгенерировать мип-уровни
GenerateMipmap(TEXTURE_2D);
```

Стоит учесть, что качество мип-уровней, сгенерированных драйвером OpenGL, не всегда может быть лучше. В этом случае можно самостоятельно загружать собственные изображения по мип-уровням.

Если требуется вывести множество одинаковых объектов, но с разными координатами, поворотом, размером и т. д., необходимо использовать расширение EXT_draw_instanced, позволяющее за один вызов нарисовать сразу множество объектов [2]. Каждый объект будет рисоваться в отдельной инстанции, при этом к нему можно прицепить различные атрибуты (например, матрицу трансформации). Для этого достаточно упаковать атрибуты в обычный VBO, а при его привязке использовать функцию VertexAttribDivisor, которая укажет OpenGL, что этот буфер необходимо использовать поинстанционно, а не повершинно. Для рисования применяются функции с приставкой Instanced, например DrawElementsInstanced.

Сортировка объектов по дальности удаления от камеры положительно влияет на производительность, так как GPU сможет отбрасывать полигоны, которые лежат позади уже нарисованных.

Отрисовка объектов, лежащих за пирамидой видимости, негативно влияет на производительность, поскольку заставляет GPU совершать операции над объектами, которые пользователь никогда не увидит. Использование frustum culling позволяет легко исключить их из отрисовки. Реализация frustum culling сводится к построению ограничивающего параллелепипеда (bounding box) для модели. Также строятся уравнения плоскости усеченной пирамиды камеры. При рисовании проверяется каждая из восьми точек bounding box на принадлежность этой пирамиде. Если все точки лежат вне пирамиды, то объект можно не рисовать.

В паре с frustum culling может также использоваться расширение ARB_occlusion_query, позволяющее находить объекты, которые не будут видны вообще (например, мелкие объекты позади крупных).

Оптимизации моделей. Использование GL_QUADS и других типов примитивов, отличных от GL_TRIANGLE, обычно плохо сказывается на производительности. GPU умеет рисовать только треугольники, поэтому остальные типы примитивов преобразуются драйвером именно в них, что не всегда происходит оптимально и может иметь дополнительные накладные расходы.

Применение индексированной геометрии позволит уменьшить количество вершин и объем занимаемой ими памяти, что также ускорит вывод. Использование 8-битных индексов для маленьких моделей может негативно сказаться на производительности. Быстрее всего GPU работают с 16-битными индексами. Также стоит избегать 32-битных индексов, поскольку они занимают больше памяти. В случае больших моделей лучше использовать несколько буферов.

Применение GL_TRIANGLE_STRIP вместо GL_TRIANGLES полезно, так как количество индексов в лентах треугольников обычно меньше, что положительно влияет на использование памяти и скорость рисования. Расширение NV_primitive_restart (также доступно

на видеокартах ATI) позволяет избавиться от дегенератов (degenerate triangles).

Вертексы и индексы должны быть отсортированы так, чтобы максимально задействовать pre-vertex cache и post-vertex cache. GPU всегда использует pre-vertex cache и читает вертексы «наперед», поэтому полезно расположить вертексы, используемые через небольшие промежутки времени, близко один к другому [3]. В post-vertex cache GPU хранит несколько последних трансформированных вертексов, что позволяет не выполнять для них повторные трансформации. Использование лент треугольников дает возможность в максимальной степени задействовать post-vertex cache.

Сжатие текстур увеличивает производительность благодаря уменьшению количества пересылаемых данных по шине. Основных способов сжатия текстур три: DXT1, DXT3 и DXT5. Первый позволяет сжимать текстуры в 8 раз, но имеет альфа-канал всего в один бит, т. е. полупрозрачные текстуры не поддерживаются. Второй и третий имеют поддержку альфа-канала, но сжимают текстуры в 4 раза хуже. Отметим, что DXT3 может давать лучшее качество, если исходное изображение содержит резкие переходы альфа-канала. Стоит учесть, что степень сжатия всегда гарантирована. DXT — это алгоритм сжатия с потерями. Качество исходного изображения может незначительно измениться, но в общем случае разница между сжатой и оригинальной текстурами не заметна. Использование сжатия позволяет также увеличить скорость загрузки текстур и уменьшить размер установочного пакета приложения.

Использование объектов с различной степенью детализации положительно скажется на производительности, если во время рисования выбирать наиболее подходящую модель по расстоянию до камеры. Это позволит сократить расходы на рендеринг удаленных объектов, не нуждающихся в высокой детализации.

Таким образом, низкоуровневые оптимизации значительно уменьшают нагрузку на CPU, а оптимизации рисования и самих моделей снижают нагрузку на GPU, что помогает создавать качественное и быстрое программное обеспечение.

ЛИТЕРАТУРА

[1] Евченко А.И. *OpenGL и DirectX: программирование графики. Для профессионалов.* Санкт-Петербург, Питер, 2006, 350 с.

[2] Фрэнсис Х. *OpenGL. Программирование компьютерной графики. Для профессионалов.* Шкадов А., ред. Санкт-Петербург, Питер, 2002, 1088 с.

[3] Шикин Е.В., Боресков А.В. *Компьютерная графика. Полигональные модели.* Москва, ДИАЛОГ-МИФИ, 2001, 384 с.

Статья поступила в редакцию 05.06.2014

Ссылку на эту статью просим оформлять следующим образом:

Дикан И.В., Белов Ю.С. Методы оптимизации под OPENGL. *Инженерный журнал: наука и инновации*, 2014, вып. 11. URL: http://engjournal.ru/catalog/it/hidden/1279.html

Дикан Игорь Вадимович родился в 1994 г. Студент кафедры «Программное обеспечение ЭВМ, информационные технологии и прикладная математика» КФ МГТУ им. Н.Э. Баумана. Область научных интересов: информационные технологии, графические библиотеки. e-mail: temtaime@gmail.com

Белов Юрий Сергеевич родился в 1982 г., окончил КФ МГТУ им. Н.Э. Баумана в 2006 г. Канд. физ.-мат. наук, доцент кафедры «Программное обеспечение ЭВМ, информационные технологии, прикладная математика» КФ МГТУ им. Н.Э. Баумана. Область научных интересов: информационные технологии, компьютерное моделирование, интеллектуальный анализ данных. e-mail: ybs82@mail.ru

OpenGL performance optimization guide

© I.V. Dikan, Yu.S. Belov

Kaluga Branch of Bauman Moscow State Technical University, Kaluga, 248000, Russia

The article describes basic methods of optimizing applications written for using the graphics library OpenGL. It presents methods of low-level optimization, including those that use various extensions of OpenGL. We show improved methods of drawing graphics. Methods of changing the format of graphic objects to enhance performance of the end applications are also described.

Keywords: *VBO, VAO, OpenGL, 3D-graphics, DXT, optimization, shaders, MipMapping, Occlusion Query, Frustum Culling.*

REFERENCES

[1] Evchenko A.I. *OpenGL i DirectX: programmirovanie grafiki. Dlya professionalov* [OpenGL and DirectX: programming graphics. For professionals]. Saint-Petersburg, Piter Publ., 2006, 350 p.

[2] Frensis Kh. *OpenGL. Programmirovanie kompyuternoi grafiki. Dlya professionalov* [OpenGL. Programming computer graphics. For professionals]. [in Russian] Shkadov A., ed. Saint-Petersburg, Piter Publ., 2002, 1088 p.

[3] Shikin E.V., Boreskov A.V. *Kompyuternaya grafika. Poligonal'nye modeli* [Computer graphics. Polygonal models]. Moscow, "DIALOG-MIFI" Publ., 2001, 384 p.

Dikan I.V. (b. 1994) is a Bachelor degree student of the Department of Computer Software, IT and Applied Mathematics at Kaluga branch of Bauman Moscow State Technical University. Academic interests include information technologies, graphics libraries. e-mail: temtaime@gmail.com

Belov Yu.S. (b. 1982) graduated from Kaluga branch of Bauman Moscow State Technical University in 2006. Ph.D., Assoc. Professor of the Department of Computer Software, IT and Applied Mathematics at Kaluga branch of Bauman Moscow State Technical University. Research interests include information technologies, computer simulation, intellectual data analysis. e-mail: ybs82@mail.ru