OpenGL

OpenGL (**Open** **G**raphics **L**ibrary) - спецификация, определяющая платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику.

Приводимое далее описание ориентировано на C#, где доступ к библиотечным функциям OpenGL осуществляется через соответствующий объект класса OpenGL. При этом названия методов идентичны оригинальным функциям, за небольшими исключениями:

1) Префиксы библиотек gl, glu и тд пропущены (например LoadIdentity() вместо glLoadIdentity())
2) Окончание многих функций конкретизирующих тип опущены, а их реализация осуществлена перегрузкой (например в именовании функций glColor3f, glColor3d, glColor3i окончания 3f, 3d, 3i говорит о том что в качестве параметров передаются 3 числа типа float, double и integer соответственно. В данном случае все они доступны через различные перегрузки метода Color())
3) Так как C# является управляемым кодом работа с неуправляемой памятью и указателями хоть и возможна, но несколько затруднена. По этой причине в некоторых случаях могут быть небольшие изменения в списках параметров или добавление перегрузок с измененными параметрами. Так, в следующей функции glVertexPointer(INT size, DWORD type, INT stride, PVOID pointer) параметр pointer является указателем на массив данных, а параметр type задает их тип - GL_FLOAT, GL_DOUBLE, GL_INT и тд. В данном случае, что бы избавиться от возни с маршалингом и выделением \ освобождением неуправляемой памяти добавлены несколько перегрузок вида VertexPointer(int size, int stride, float[] pointer) .

В классе OpenGL так же объявлены константы из библиотеки OpenGL. Их имена идентичны оригинальным, но в силу специфики языка C# для обращения к ним необходимо указывать имя типа. Например, следующий вызов функции glEnable(GL_BLEND) в нашем случае сведется к вызову метода Enable(OpenGL.GL_BLEND). Само создание объекта типа OpenGL осуществляется при создании устройства вывода (класс OGLDevice) и доступ к нему можно получить при помощи свойства gl данного объекта (RenderDevice) или объекта типа OGLDeviceUpdateArgs передаваемого в качестве параметра методу OnDeviceUpdate(). Данный метод, как и сама работа с устройством OpenGL реализуются в параллельном потоке. Обращение к устройству OpenGL из другого потока не допускается (создание многопоточного рендера возможно, но это достаточно специфическая архитектура, например рендинг частей экрана в текустуры а потом их объединение).

Для большинства функций библиотеки OpenGL при отладке DEBUG конфигурации осуществляется проверка ошибок выполнения и их вывод в окно вывода Microsoft Visual Studio. Поэтому при отладке и написании кода связанного с OpenGL необходимо также контролировать ошибки библиотеки OpenGL в окне вывода.

Параметры рендера

Используемое правило объявления вершин:

void FrontFace(uint mode)

- mode - константа задающая порядок перечисления вершин:
  - OpenGL.GL_CW - по часовой стрелке
  - OpenGL.GL_CCW - против часовой стрелки

Исключение полигонов из процесса растеризации и фрагментного шейдера:

void Enable(OpenGL.GL_CULL_FACE)

void CullFace (uint mode)

- mode - константа определяющая, какие полигоны отбрасывать
  - OpenGL. GL_FRONT - повернутые лицевой стороной
  - OpenGL. GL_BACK - повернутые изнанкой

Режим отрисовки, позволяет определить как именно будут рисовать указанные полигоны:

void PolygonMode(uint face, uint mode)

- face - константа определяющая, к каким полигонам применяется правило
  - OpenGL. GL_FRONT - повернутые лицевой стороной
  - OpenGL. GL_BACK - повернутые изнанкой
  - OpenGL. GL_FRONT_AND_BACK - догадайтесь...
- mode - режим отображения
  - GL_FILL - полигоны рисуются целиком, как закрашенная плоскость
  - GL_LINE - рисуются контуры полигонов, так называемая каркасная визуализация
  - GL_POINT - рисуются вершины полигонов

Порядок наложения (смешивания) цветов:

void Enable(OpenGL.GL_BLEND)

void BlendFunc(uint sfactor, uint dfactor)

- sfactor, dfactor - константы определяющие значение в формуле наложения цветов: result_color = src * sfactor + dst * dfactor, могут принимать следующие значения: GL_ZERO, GL_ONE, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, GL_SRC_ALPHA_SATURATE. Наиболее привычное (нормальное) наложение цветов задается правилом: src * SRC_ALPHA + dst * ONE_MINUS_SRC_ALPHA

Использование буфера глубины (ака z-буфер - требуется для проверки порядка положения отображаемых полигонов). При этом задается функция теста глубины (GL_LEQUAL - фрагмент проходит тест если его значение глубины меньше или равно хранимому в буфере, об остальных типах легко догадаться по их названию - GL_NEVER, GL_LESS, GL_EQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, GL_ALWAYS). Перед началом работы с буфером его необходимо очистить вместе с буфером шаблона (Stencil Buffer - дополнительный буфер, соответствующий размеру выводимого кадра. Каждый раз когда точка рисуется на экран, то кроме сравнения с глубиной в Z-буфере, она проходит еще и Stencil тест).

```
gl.Enable(OpenGL.GL_DEPTH_TEST);
gl.DepthFunc(OpenGL.GL_LEQUAL);
gl.ClearDepth(1.0f);    // 0 - ближе, 1 - далеко
gl.ClearStencil(0);
```

Параметры сглаживания (сглаживание для полигонов на современых видео картах зачастую работает не очень коректно, так как для этого уже давно используются более прогрессивные технологии вроде MSAA)

```
gl.Enable(OpenGL.GL_LINE_SMOOTH);
gl.Hint(OpenGL.GL_LINE_SMOOTH_HINT, OpenGL.GL_NICEST);
//gl.Enable(OpenGL.GL_POLYGON_SMOOTH);
//gl.Hint(OpenGL.GL_POLYGON_SMOOTH_HINT, OpenGL.GL_NICEST);
```

Очистка экрана (закраска в указанный цвет)

```
gl.ClearColor(0, 0, 0, 0); // цвет фона - r, g, b, a
```

Проецирование

Для загрузки матриц преобразования используется метод void LoadMatrixf([] m), где m - указатель на массив из 16 элементов содержащий значения матрицы. Элементы матрицы в массиве перечисляются по столбцам, а не по строкам. Для преобразования матрица DMatrix3 и DMatrix4 предусмотрен метод double[] ToArray(bool glorder = false) и float[] ToFloatArray(bool glorder = false), параметр glorder как раз и отвечает за порядок элементов.

Прежде чем загружать значения матрицы, нужно задать матричный режим, иными словами определить матрицу, над которой в будут производиться дальнейшие операции. Делается это при помощи метода void MatrixMode (uint mode), где параметр mode принимает одно из следующих значений:

- GL_PROJECTION - матрица проекций (переход из пространства камеры в однородное пространство), обычно задается при инициализации приложения и изменениях размеров области отображения.
- GL_MODELVIEW - объекто-видовая матрица,

В результате задания матриц OpenGL будет осуществлять преобразования в виде: P_transformed = M_projection * M_modelview * P

Зададим матрицу проекций:

```
gl.MatrixMode(OpenGL.GL_PROJECTION);
var pMatrix = Perspective(60, (double)e.Width / e.Height, 0.1, 100);
gl.LoadMatrix(pMatrix.ToArray(true));
```

Зададим объектно-видовую матрицу:

```
gl.MatrixMode(OpenGL.GL_MODELVIEW);
var deg2rad = Math.PI / 180;    // Вращается камера, а не сам объект
var cameraTransform = (DMatrix3)Rotation(deg2rad * cameraAngle.X, deg2rad *
cameraAngle.Y, deg2rad * cameraAngle.Z);
var cameraPosition = cameraTransform * new DVector3(0, 0, cameraDistance);
var cameraUpDirection = cameraTransform * new DVector3(0, 1, 0);

// Мировая матрица (преобразование локальной системы координат в мировую)
var mMatrix = DMatrix4.Identity;    // нет никаких преобразований над объекта
// Видовая матрица (переход из мировой системы координат к системе координат
камеры)
var vMatrix = LookAt(DMatrix4.Identity, cameraPosition, DVector3.Zero,
cameraUpDirection);

// матрица ModelView
var mvMatrix = vMatrix * mMatrix;
gl.LoadMatrix(mvMatrix.ToArray(true));
```

```
/// <summary>
/// Матрица перспективной проекции
/// </summary>
/// <param name="verticalAngle">Вертикальное поле зрения в градусах. Обычно между 90
(очень широкое) и 30 (узкое)</param>
/// <param name="aspectRatio">Отношение сторон. Зависит от размеров устройства вывода
(окна)</param>
/// <param name="nearPlane">Ближняя плоскость отсечения. Должна быть больше 0</param>
/// <param name="farPlane">Дальняя плоскость отсечения</param>
private static DMatrix4 Perspective(double verticalAngle, double aspectRatio,
double nearPlane, double farPlane) {
    var radians = (verticalAngle / 2) * Math.PI / 180;
    var sine = Math.Sin(radians);
    if (nearPlane == farPlane || aspectRatio == 0 || sine == 0)
        return DMatrix4.Zero;
    var cotan = Math.Cos(radians) / sine;
    var clip  = farPlane - nearPlane;
    return new DMatrix4(
        cotan/aspectRatio,  0,  0,  0,
        0,  cotan,  0,  0,
        0,  0,  -(nearPlane+farPlane)/clip,  -(2.0*nearPlane*farPlane)/clip,
            0,  0,  -1.0,  1.0
        );
    }


/// <summary>
/// Умножение матрицы на видовую матрицу, полученную из точки наблюдения.<para/>
/// Вектор up не должен быть параллелен линии зрения от глаза к центру.
/// </summary>
/// <param name="matrix">Проекционная матрица</param>
/// <param name="eye">Положение камеры в мировых координатах</param>
/// <param name="center">Направление взгляда в мировом пространстве</param>
/// <param name="up">Направление вверх, которое следует рассматривать по отношению к
глазу.</param>
/// <returns>Произведение матрицы и видовой матрицы</returns>
private static DMatrix4 LookAt(DMatrix4 matrix, DVector3 eye, DVector3 center,
DVector3 up) {
   var forward = (center - eye).Normalized();
   if (forward.ApproxEqual(DVector3.Zero, 0.00001))
        return matrix;
   var side = (forward * up).Normalized();
   var upVector = side * forward;
   var result = matrix * new DMatrix4(
        +side.X,      +side.Y,      +side.Z,      0,
        +upVector.X,  +upVector.Y,  +upVector.Z,  0,
        -forward.X,   -forward.Y,   -forward.Z,   0,
         0,            0,            0,           1
   );
   result.M14 -= result.M11 * eye.X + result.M12 * eye.Y + result.M13 * eye.Z;
   result.M24 -= result.M21 * eye.X + result.M22 * eye.Y + result.M23 * eye.Z;
   result.M34 -= result.M31 * eye.X + result.M32 * eye.Y + result.M33 * eye.Z;
   result.M44 -= result.M41 * eye.X + result.M42 * eye.Y + result.M43 * eye.Z;
   return result;
}
```

Представление 3х мерного объекта

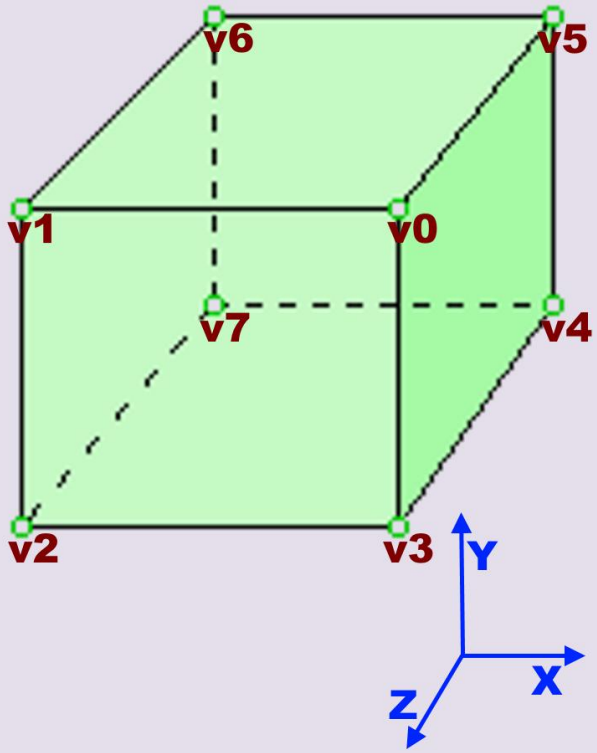
Для рендинга методами VA или VB, необходимо представить описание объекта в виде массива структур содержащих свойства вершин. В качестве примера, ниже приводится описание вершин кубика, перечисленных против часовой стрелки, в массиве vertices состоящим из структур типа Vertex. А сами полигоны формируются через массив индексов indices, где значения являются индексами соответствующих вершин в массиве vertices, а каждый полигон является гранью, т.е. задается 4мя вершинами.

```
[StructLayout(LayoutKind.Sequential, Pack =1)]
private struct Vertex
{
    // Координата
    public readonly float vx, vy, vz;

    // Нормаль
    public readonly float nx, ny, nz;

    // Цвет
    public readonly float  r,  g,  b;

    public Vertex(
               float vx, float vy, float vz,
               float nx, float ny, float nz,
               float r,  float g,  float b)
    {
        this.vx =vx; this.vy =vy; this.vz =vz;
        this.nx =nx; this.ny =ny; this.nz =nz;
        this.r  =r;  this.g  =g;  this.b  =b;
    }
}
```




```
private static readonly Vertex[] vertices = {
    //           vx    vy    vz     nx    ny    nz     r    g    b
    new Vertex( .5f,  .5f,  .5f,    nf,   nf,   nf,    1f,  1f,  1f),
    new Vertex(-.5f,  .5f,  .5f,   -nf,   nf,   nf,    1f,  1f,  0f),
    new Vertex(-.5f, -.5f,  .5f,   -nf,  -nf,   nf,    1f,  0f,  0f),
    new Vertex( .5f, -.5f,  .5f,    nf,  -nf,   nf,    1f,  0f,  1f),
    new Vertex( .5f, -.5f, -.5f,    nf,  -nf,  -nf,    0f,  0f,  1f),
    new Vertex( .5f,  .5f, -.5f,    nf,   nf,  -nf,    0f,  1f,  1f),
    new Vertex(-.5f,  .5f, -.5f,   -nf,   nf,  -nf,    0f,  1f,  0f),
    new Vertex(-.5f, -.5f, -.5f,   -nf,  -nf,  -nf,    0f,  0f,  0f),
};
private static readonly float nf = (float)(1 / Math.Sqrt(3));

private readonly static uint[] indices = {
    // Первая последовательность
       5, 6,    0,  1,  // {v0,v5,v6,v1} - верхня грань
    /* 0, 1 */  3,  2,  // {v0,v1,v2,v3} - передняя грань
    /* 3, 2 */  4,  7,  // {v7,v4,v3,v2} - нижняя грань
    // Вторая последовательность
       2, 1,    7, 6,   // {v1,v6,v7,v2} - левая грань
    /* 7, 6 */  4, 5,   // {v4,v7,v6,v5} - задняя грань
    /* 4, 5 */  3, 0    // {v0,v3,v4,v5} - правая грань
};
```

Рендинг методом VA (Vertex Array)

Метод Vertex Array позволяет отображать объект, информация о котором хранится в оперативной памяти. Для этого в коде рендинга объекта необходимо указать какие именно данные будут браться из памяти при помощи метода: void EnableClientState(uint array), где параметр array задаёт тип используемых данных:

- OpenGL.GL_VERTEX_ARRAY - массив вершин
- OpenGL.GL_COLOR_ARRAY - массив цветов вершин
- OpenGL.GL_NORMAL_ARRAY - массив нормалей вершин
- OpenGL.GL_INDEX_ARRAY - массив индексов
- OpenGL.GL_TEXTURE_COORD_ARRAY - массив текстурных координат

Далее для каждого используемого типа данных нужно задать их расположение в памяти (передать указатель на массив соответствующих элементов) для чего служит группа методов:

- void NormalPointer(int stride, [] pointer)
- void ColorPointer(int size, int stride, [] pointer)
- void VertexPointer(int size, int stride, [] pointer)
- void IndexPointer (uint type, int stride, [] pointer)
- void TexCoordPointer(int size, uint type, int stride, [] pointer)

Их параметры:

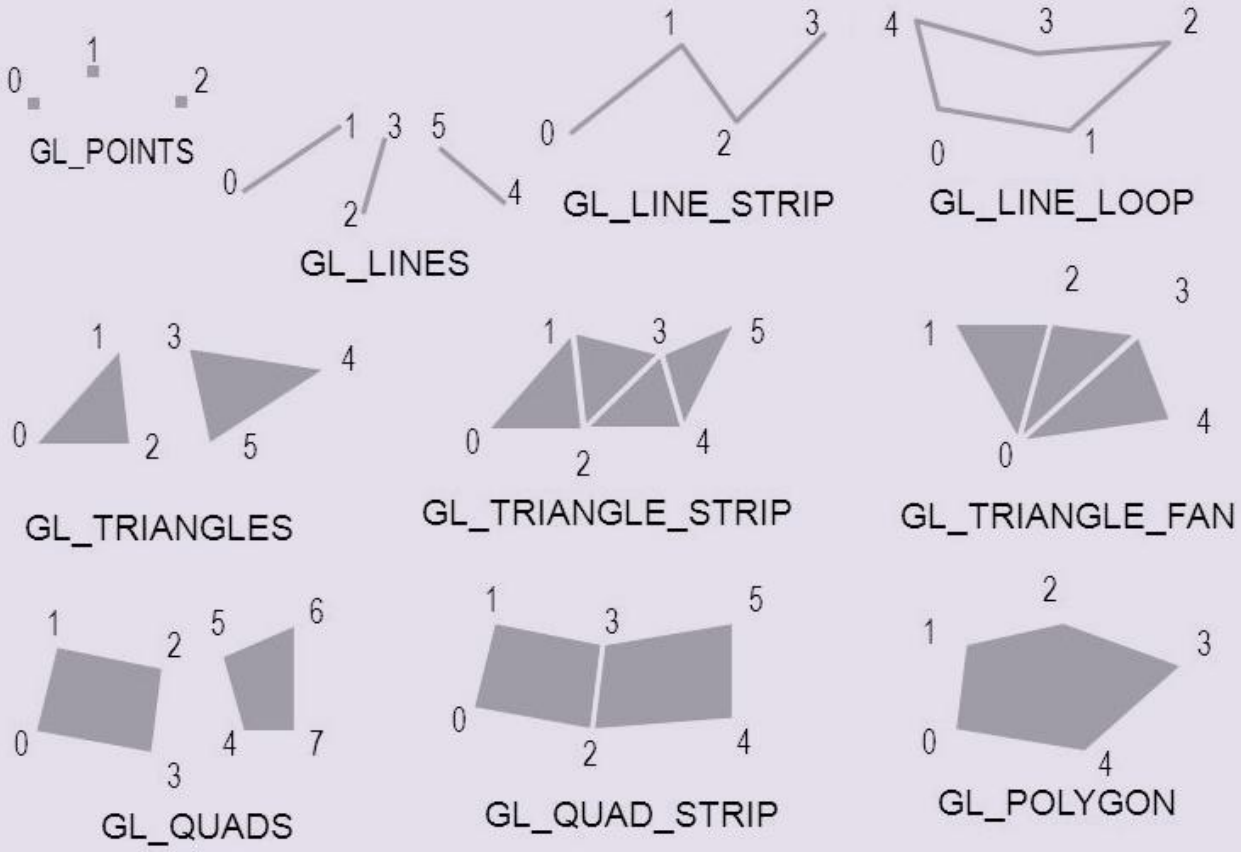
- pointer - указатель на первый элемент соответствующих элементов в массиве данных (например, &vertices[0].vx, &vertices[0].nx, &vertices[0].r)
- size - размер векторов содержащихся в массиве. Иными словами количество элементов описывающих величину (например 3 если вершины задаются как {x,y,z} или 4 если задаются как {x,y,z,w}. Аналогично с цветом в зависимости от того как он задается {r,g,b или {a,r,g,b})
- stride - шаг, то есть размер между элементами данных. Обычно равен размеру всей структуры - sizeof(Vertex) или 0, если элементы в памяти расположены последовательно.

После задания расположения всех данных и при необходимости изменения параметров отображения, можно отрендерить объект, вызвав метод:

void DrawElements(uint mode, int count, [] indices)

Параметры:

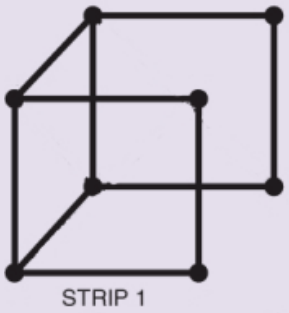
- indices - указатель на массив индексов вершин
- count - количество обрабатываемых вершин в массиве indices
- mode - тип используемых примитивов, в свою очередь также задаёт правило перечисления вершин:



В приведенном примере отображение предполагается осуществлять за два прохода:

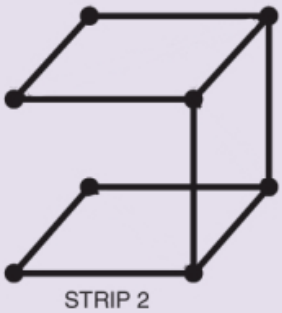
gl.DrawElements(OpenGL.GL_QUAD_STRIP, 8, &indices[0]);

gl.DrawElements(OpenGL.GL_QUAD_STRIP, 8, &indices[8]);



По окончанию отображения данных информируем о прекращении использования данных размещенных в оперативной памяти при помощи метода:

void DisableClientState (uint array), параметр array аналогичен одноименному параметру в методе EnableClientState() описанном выше.



Рендинг методом VB (Vertex Buffer)

Метод Vertex Buffer позволяет отображать объект, информация о котором хранится непосредственно в видео памяти графического адаптера. Так как обычными средствами невозможно напрямую адресовать область видео памяти, работа с ним осуществляется целиком через OpenGL и начинается с создания объектов буферов вершин и индексов:

void GenBuffers(int n, uint[] buffers)

- n - количество создаваемых объектов буфера
- buffers - массив, в который будут помещены идентификаторами созданных объектов.

Для нашей задачи потребуется создать два объекта для вершин и индексов. После создания объекты буфера имеют нулевую длину и не содержат никаких данных. Но прежде чем приступить к заполнению или любой другой работе с объектом буфера необходимо его связать с соответствующим индентификатором:

void BindBuffer(uint target, uint buffer)

- target - целевой объект определяет, будет ли этот буферный объект хранить данные массива вершин или данные массива индексов. Любые атрибуты вершин, такие как вершинные и текстурные координаты, нормали и цветовые составляющие, должны использовать значения OpenGL.GL_ARRAY_BUFFER. Индексный массив, должен быть привязан к OpenGL.GL_ELEMENT_ARRAY_BUFFER. Обратите внимание, что этот целевой флаг помогает VBO выбирать наиболее эффективное расположения буферных объектов, например, некоторые системы могут предпочесть размещать индексы или вершины не в оперативной, а в видеопамяти.
- buffer - идентификатор объекта буфера вершин (возвращаемый GenBuffers()) или 0 - недействительный индентификатор. Привязка к 0 "развяжет" объект буфера, т.е. переключит обратно к нормальному режиму без использования данного буфера.

Теперь, когда буфер был инициализирован, можно скопировать в него данные:

void BufferData(uint target, int size, IntPtr data, uint usage

- target - целевой объект (см описание BindBuffer())
- size - размер передаваемых данных в байтах
- data - указатель на массив исходных данных. Если вместо в качестве указателя идет NULL (IntPtr.Zero), то резервируется только пространство памяти с заданным размером данных.
- usage - флаг указывающий как именно будет использоваться буферный объект. Возможные значение: GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, GL_DYNAMIC_COPY, GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY

- «STATIC» означает, что данные не будут изменены (заданы один раз и используются много раз),
- «DYNAMIC» означает, что данные будут часто меняться (заданы и используются повторно),
- «STREAM» означает, что данные будут изменены каждый кадр (указано один раз и используется один раз).
- «DRAW» означает, что данные будут отправляться на GPU для рисования (приложение в GL),
- «READ» означает, что данные будут считаться приложением клиента (GL для приложения),
- «COPY» означает, что данные будут использоваться как рисование, так и чтение (GL to GL).

Как правило для VBO полезно использовать флаг DRAW, а флаги COPY и READ могут оказаться полезными для PBO или FBO (пиксельного / фреймового буфера).
Диспетчер памяти выберет наилучшее место для размещения объекта буфера в памяти на основе перечисленных флагов, например GL_STATIC_DRAW и GL_STREAM_DRAW могут использовать видеопамять, а GL_DYNAMIC_DRAW может использовать AGP память. Любые связанные с _READ_ буферы будут хороши в системной или AGP-памяти, поскольку данные должны быть легко доступны.

На практике часто возникает необходимость доступа к данным объекта буфера или их части в процессе работы приложения. Например для их изменения. Это реализуется при помощи отображения содержимого или его части в память:

IntPtr MapBuffer(uint target, uint access)

- target - целевой объект (см описание BindBuffer())
- access - режим доступа (GL_READ_ONLY, GL_WRITE_ONLY, GL_READ_WRITE) Режим доступа зависит от того что мы хотим делать с данными - считывать, записывать или все вместе.

В качестве результата метод возвращает указатель на область памяти с отображением содержимого буфера или NULL (IntPtr.Zero) в случае если операцию не удалось выполнить. Обратите внимание, если GPU все еще работает с объектом-буфером, вызов MapBuffer() приведет к ожиданию (бездействию) завершения выполнения GPU своего задания с соответствующим объектом буфера. Это может стать причиной проблем синхронизации, для ее решения можно предварительно вызвать BufferData() с NULL в качестве указателя. В этом случае предыдущие будут отброшены, и MapBuffer() немедленно вернет новый выделенный указатель, даже если графический процессор все еще работает с предыдущими данными. Однако данный метод применителен лишь в том случае, когда надо обновить весь набор данных.

По окончанию работы с отображенной памятью буфера-объекта ее необходимо освободить при помощи метода bool UnmapBuffer(uint target). В случае успеха функция возвращает true. В случае неудачи возвращает false, это как правило так же свидетельствует о том, что содержимое буфера-объекта было повреждено (например в следствии изменения разрешения экрана или каких-то других системных событий). В этом случае необходимо повторно передать данные.

Для приведенного примера кубика, создадим два объекта буфера и поместим в них содержимое массивов vertices и indices. Код рендера по большому счету отличается от метода VA в способе привязки буферов. Если в VA для функций gl*Pointer() передавались указатели на массивы данных, то в VB вместо указателей передаются смещения в байтах внутри буфера на соответствующие данные. Аналогичная ситуация и с методом DrawElements(). Поведение данных методов зависит от того связан ли соответствующий тип буфера с чем-то в данный момент времени или нет (BindBuffer()).

Согласно правилам хорошего тона - мусор за собой следует убирать, поэтому не нужные объекты буферов нужно удалять. В том числе и при завершении работы приложения. Делается это при помощи метода:

void DeleteBuffers(int n, uint[] buffers)

- n - количество удаляемых объектов буфера
- buffers - массив, в котором содержатся идентификаторы удаляемых объектов.

Приложение: Работа с указателями в C#

Для любой работы с неуправляемой памятью напрямую и использования указателей необходимо в настройках проекта на вкладке построение поставить галочку напротив "Разрешить небезопасный код". Любой небезопасный код должен быть размещен в блоке unsafe. Тело метода или даже содержимого всего класса может также выступать в качестве такого блока если добавить данное ключевое слово в его определение:

- unsafe { int* ptr = ...; }
- public unsafe void Foo() { int* ptr = ...; }
- public unsafe class { void Foo() { int* ptr = ...; } }

Так как платформа .Net является управляемым кодом (managed), что в частности означает, что реальное положение в памяти данных может быть изменено средой CLR в любой момент прямо в процессе работы приложения. То получение

указателя любого управляемого объекта осуществляется при помощи специального оператора fixed, что в частности запрещает среде CLR в течении работы с блоком данных изменять расположение данных в памяти.

```
public float[] array = new float[16];
fixed (float* ptr = array) {
    for (int i = 0; i < 16; ++i)
        *(ptr + i) = (float)i;
}
```

Синтаксис и возможности работы с указателями идентичны C\C++. В качестве аналога void* часто используется специальный тип IntPtr, поддерживающий возможности приведения типа:

```
float* ptr_1;
IntPtr ptr_2 = (IntPtr) ptr_1; // или new IntPtr(ptr_1);
void*  ptr_3 = (void *) ptr_2; // или ptr_2.ToPointer();
```

При объявлении структуры Vertex желательно воспользоваться атрибутом:

```
[StructLayout(LayoutKind.Sequential, Pack =1)]
```

В этом случае явно задается последовательное расположение данных в памяти и их выравнивание в памяти (аналог #pragma pack(1) в C\C++).

Получить смещение в байтах внутри структуры например до свойства nx, можно при помощи маршалинга:

```
int offset_nx = (int)Marshal.OffsetOf(typeof(Vertex), "nx"));
```

Получить указатель на первый элемент nx в массиве vertices можно следующим образом:

```
IntPtr.Add(&vertices,  offset_nx);
```