第3章 Chapter 3

基本图元

绘制三维图形离不开三维坐标系，在三维图形程序中，任何复杂物体的模型都是由基本图元组成的。因此三维图形开发接口中的坐标系和基本图元是开发三维图形程序的基础，也是理解三维图形开发接口高级功能的基础。本章介绍 OpenGL ES 中使用的三维坐标系和基本图元类型。

3.1 OpenGL ES 坐标系

在几何空间中，绝大多数情况下使用笛卡儿坐标系为参照系来表示图形，表示二维图形时使用二维笛卡儿坐标系，表示三维图形时使用三维笛卡儿坐标系。

3.1.1 左手和右手坐标系

三维笛卡儿坐标系根据 z 坐标轴相对于 x、y 坐标轴方向的不同，可分为左手坐标系和右手坐标系。在左手体系中，坐标轴的定义符合法则：左手四个手指的旋转方向从 x 轴到 y 轴，大拇指的指向就是 z 轴。右手体系依次类推，如图 3.1 所示。OpenGL ES 使用右手坐标系。

3.1.2 OpenGL ES 默认坐标系

默认状态下，OpenGL ES 中的坐标原点(0,0,0)位于 iPhone 屏幕中央。

OpenGL ES 使用自己的单位，具体含义由程序员自己定义，比如一个单位代表一个像素、一米、一英里等。

默认状态下，OpenGL ES 的单位是 iPhone 屏幕宽和高的一半，如图 3.2 所示。

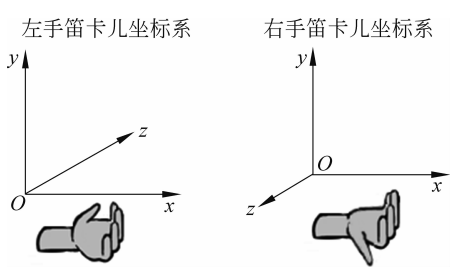

图 3.1 左手和右手笛卡儿坐标系

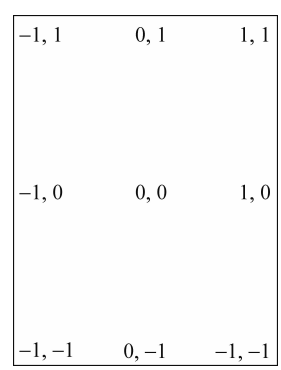

图 3.2 iPhone 默认的 Open GL ES 坐标系

3.2 图元

对象的3D外观通常被称为几何形状。在OpenGL ES中,对象的几何形状一般由一系列的图元构成。

在OpenGL ES中,一共有三种基本图元,分别是点、线和三角形。

通常三维图元是多边形。一个多边形是由至少三个顶点描绘的三维形体。最简单的多边形是三角形。OpenGL ES使用三角形组成大多数多边形,因为三角形的三个顶点一定是共面的。应用程序可以用三角形组合成大而复杂的多边形。

图元往往采用顶点数组加以定义,并根据顶点的拓扑关系加以连接。OpenGL ES支持7种拓扑关系,分别是点列表(GL_POINTS)、线列表(GL_LINES)、线环(GL_LINE_LOOP)、线条带(GL_LINE_STRIP)、三角形列表(GL_TRIANGLES)、三角形条带(GL_TRIANGLE_STRIP)和三角形扇(GL_TRIANGLE_FAN),如图3.3所示。

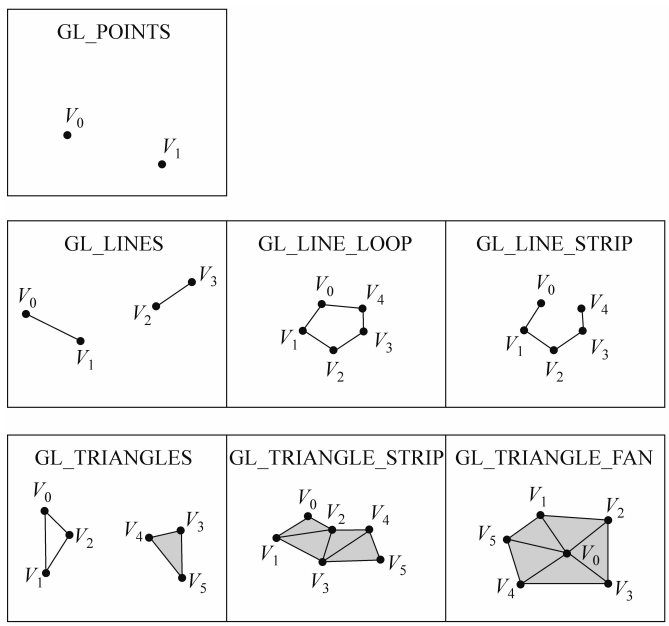

图3.3 图元的拓扑结构

3.2.1 点图元

没有再比点更有用、更简单的图元了,因此我们最先介绍它。在3D空间中绘制点虽然简单,却十分有用。

点的列表简称点表,是顶点的集合,被渲染为孤立的点,如图3.4所示。应用程序可以将它们用于渲染星空,或多边形表面的虚线。

3.2.2 渲染点图元

渲染点图元的技术是渲染所有图元的基础,下面详细介绍如何渲染点图元。

1. 准备项目

将第 2 章所创建的 OpenGLESFromScratch 项目复制一份，并将目录名字改为 POINTS，如图 3.5 所示。

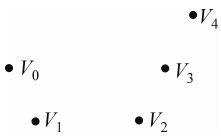

图 3.4 点表

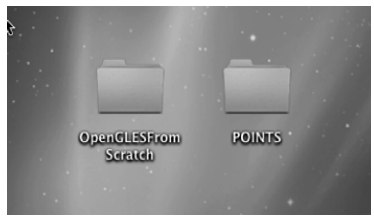

图 3.5 复制初始项目为 POINTS

进入 POINTS 目录，打开项目 OpenGLESFromScratch，并将项目改名为 POINTS。修改项目名的方法是，单击 Project 菜单，在弹出菜单中单击 Rename 命令，如图 3.6 所示。

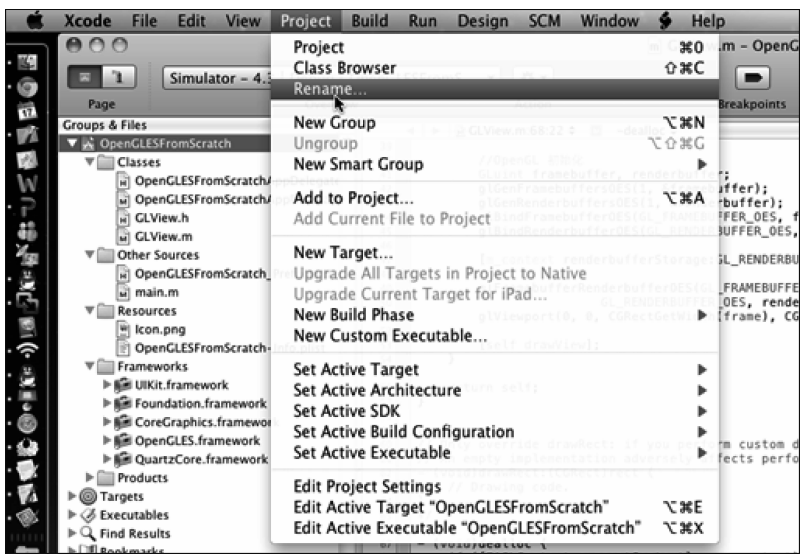

图 3.6 项目改名菜单

在弹出对话框中输入新项目名 POINTS，单击 Rename 按钮即可改名，如图 3.7 所示。

2. 定义顶点数组

在 GLView.m 中，定义顶点属性数组。我们定义 3 个顶点，每个顶点包含 X 坐标和 Y 坐标，存在一个 float 类型的数组中，见以下粗体代码：

```
#import "GLView.h"

//定义顶点坐标
const float pointVertices[]={
    -0.5,0.5,
    0.5,0.5,
    0.0,0.0,
};
```

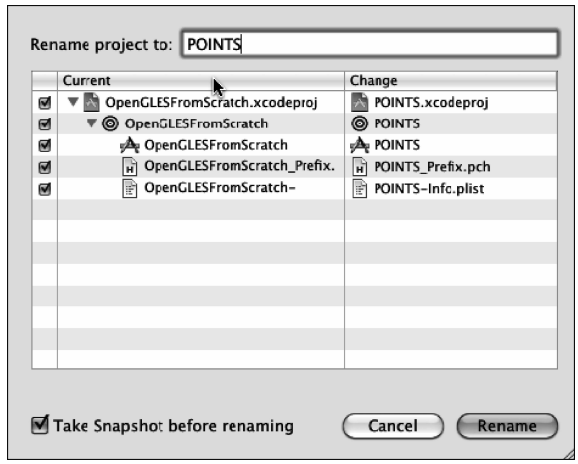

图 3.7 项目改名对话框

```
@implementation GLView
```

以上代码中，pointVectices 数组包含 6 个元素，每两个元素表示一个顶点的坐标，前一个表示 X 坐标，后一个表示 Y 坐标。

根据前面介绍的 iPhone 默认的 OpenGL ES 坐标系，第一个点(－0.5，0.5)应出现在屏幕左上部分，第二个点(0.5，0.5)应出现在屏幕右上部分，而第三个点(0，0)应出现在屏幕中央。

3. 渲染点表

修改 drawView 方法，添加渲染点表相关的代码，见以下粗体代码：

```
-(void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &pointVertices[0]);
    GLsizei vertexCount=sizeof(pointVertices)/(sizeof(float)*2);
    glDrawArrays(GL_POINTS, 0, vertexCount);
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

OpenGL ES 是一个客户端－服务器结构，客户端为 CPU 端，服务器为 GPU 端。

OpenGL ES 又是一个状态机，由许多设置项组成，影响着渲染的方方面面。因为此状态机在要做的每一件与 OpenGL ES 有关的事中都会发挥作用，所以了解 OpenGL ES 状态机的默认设置，掌握获取其当前设置的信息以及对其进行设置的方法尤为重要。

以上代码中，首先调用 glEnableClientState 函数以启用客户端顶点数组状态。与之对应的 glDisableClientState 用于禁用客户端状态。这两个函数的原型如下：

```
void glEnableClientState(GLenum array);
void glDisableClientState(GLenum array);
```

参数 array 指定要启用或禁止的某个功能，可以是以下常量。

GL_COLOR_ARRAY：颜色数组。

GL_NORMAL_ARRAY：法线数组。

GL_POINT_SIZE_ARRAY_OES：点尺寸数组。

GL_TEXTURE_COORD_ARRAY：纹理坐标数组。

GL_VERTEX_ARRAY：顶点坐标数组。

接下来调用 glVertexPointer 函数以设置用于渲染的顶点坐标数据。其函数原型如下：

```
void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);
```

第一个参数 size：指定每个顶点坐标的维数。必须是 2、3、4。默认值是 4。

第二个参数 type：指定数组中每个数组元素的数据类型，也就是每个坐标值采用何种数据类型，可以是 GL_FLOAT、GL_BYTE、GL_SHORT 或 GL_FIXED，初始值是 GL_FLOAT。

第三个参数 stride：指定两个连续顶点坐标在顶点坐标数组中间隔多少字节。如果设置为 0，表示顶点坐标是紧凑排列的。初始值为 0。关于顶点间隔，后面还会详细阐述。

第四个参数 pointer：指定顶一个顶点的第一个坐标的指针。初始值为 0。

我们的实例中，渲染的是二维点，所以第一个参数设置为 2；顶点数组坐标数组定义为浮点型，所以第二个参数设置为 GL_FLOAT；顶点与顶点之间没有间隔，所以第三个参数设置为 0；由于从第 0 个点开始渲染（有时并不从第 0 个点开始渲染），所以第四个参数为该数组第 0 个元素的地址。

接下来这行代码用于计算顶点的个数：

```
GLsizei vertexCount=sizeof(pointVertices)/(sizeof(float) * 2);
```

这里用整个顶点坐标数组的字节数除以每个顶点坐标所占用的字节数求得。

接下来采用 glDrawArrays 函数正式渲染图元。该函数的原型如下：

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

第一个参数 mode：指定渲染哪种类型的图元，可以是前面介绍的 7 种不同的拓扑结构，分别用这些常量作为参数：GL_POINTS、GL_LINES、GL_LINE_STRIP、GL_LINE_LOOP、GL_TRIANGLES、GL_TRIANGLE_STRIP、GL_TRIANGLE_FAN。

第二个参数 first：表示从顶点数组中的哪个位置开始渲染。

第三个参数 count：表示多少个顶点构成这些图元。

最后采用 glDisableClientState 用于禁用客户端顶点数组状态。前面提到，OpenGL ES 是一个状态机，状态的改变会一直持续到下一次改变为止。为避免状态的改变对下一次渲染造成的副作用，在渲染完毕之后，将相关状态恢复到默认值。

4. 运行

运行后,屏幕上出现了 3 个小白点,如图 3.8 所示。

5. 设置点大小

你可能已经想到了,刚才渲染的 3 个点非常小(就 1 个像素),以至于很难在屏幕上看到。这是因为 OpenGL ES 默认点大小为 1.0。要改变点的大小,可以使用 glPointSize 函数。该函数的原型如下:

```
void glPointSize(GLfloat size);
void glPointSizex(GLfixed size);
```

这两个函数函数名只差一个字母,第二个函数以 x 结尾。功能基本一样的,这是因为 OpenGL ES API 是基于 C,而不是 C++,C 是不支持函数重载的。为了区分不同参数的相同功能函数,OpenGL ES 采用不同后缀的函数名加以定义。比如这两个函数,第一个函数 glPointSize 是接收浮点型参数的,而 glPointSizex 是接收整数为参数的。其他很多 OpenGL ES 的函数都是这么处理的。

glPointSize 接收一个参数 size,表示渲染点的直径的大小。以像素为单位。

以下代码采用 30 个像素渲染点图元,渲染后的结果如图 3.9 所示。

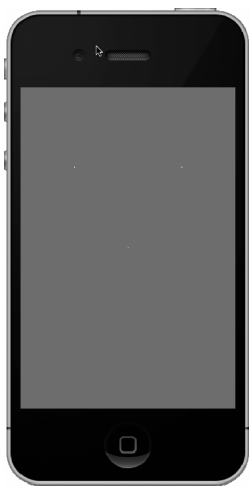

图 3.8 渲染点图元结果

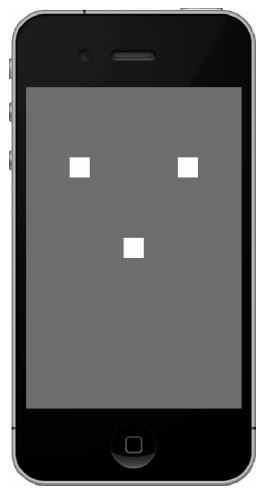

图 3.9 使用 30 个像素渲染点图元

```
- (void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &pointVertices[0]);
    GLsizei vertexCount=sizeof(pointVertices)/(sizeof(float) * 2);
    glPointSize(30.0f);                              //设置点大小
    glDrawArrays(GL_POINTS, 0, vertexCount);
    glPointSize(1.0f);                               //恢复点大小默认值(1 个像素)
```

```
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

6. 点的反走样

尽管可以用无限的精度来指定图元的大小，但是屏幕上的像素数是有限的，这就会使得图元的边缘看起来像锯齿。反走样技术可以光滑这些边缘，使其更具有真实感。

可以通过将 GL_POINT_SMOOTH 参数传递给 glEnable 函数来启用点的反走样功能。

前面提到，OpenGL ES 是客户端－服务端结构，又是一个状态机。使用 glEnableClientState、glDisableClientState 函数可以启用或禁用客户端状态。服务端状态的启用或禁用应使用 glEnable、glDisable 函数。这两个函数的原型如下：

```
void glEnable(GLenum cap);
void glDisable(GLenum cap);
```

参数 cap：指定一个表示某种 GL 能力的常量。

以下启用点的反走样，渲染点图元，渲染后的结果如图 3.10 所示。

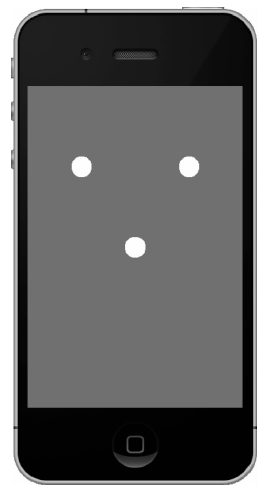

图 3.10 启用反走样渲染点图元

```
- (void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &pointVertices[0]);
    GLsizei vertexCount=sizeof(pointVertices)/(sizeof(float) * 2);
    glPointSize(30.0f);
    glEnable(GL_POINT_SMOOTH);                    //启用点反走样
    glDrawArrays(GL_POINTS, 0, vertexCount);
    glDisable(GL_POINT_SMOOTH);                   //恢复到默认值(禁止点反走样)
    glPointSize(1.0f);
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

3.2.3 线图元

线图元包括三种不同的拓扑结构，分别是线表(GL_LINES)、线条(GL_LINE_STRIP)和线环(GL_LINE_LOOP)。

线段列表简称线表，是一组孤立的直线段，如图 3.11 所示。线表对诸如给三维场景加

入雨雪或大雨这样的任务很有用。应用程序通过填写一个顶点数组来创建线表。一个线表的顶点数必须大于或等于2,并且必须是偶数,渲染 n 条线段,需要 $2n$ 个顶点。

线段条带简称线条,是一种由连接的线段所组成的图元,如图3.12所示。应用程序可用线段来创建一个不闭合的多边形。线段条带中的线段依次公用一个顶点。因此,如果渲染 n 条线段,则需要 $n+1$ 个顶点。

线段绕环简称线环,是一种由连接的线段所组成的图元,如图3.13所示。应用程序可用线段来创建一个闭合的多边形。线段绕环中的线段依次公用一个顶点,并且最后一个顶点与第一个顶点也组成一条线段。因此,如果渲染 n 条线段,则需要 n 个顶点。

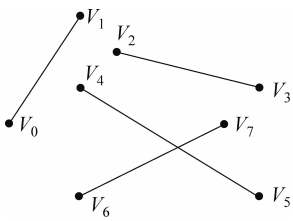

图3.11 线表

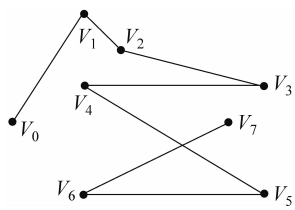

图3.12 线条

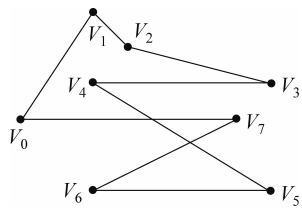

图3.13 线环

如果应用程序使用基于线环的多边形,那么不能保证顶点是共面的。

3.2.4 渲染线图元

在3D空间中渲染线图元与渲染点图元,在技术上没有什么本质区别,在调用glDrawArrays时,使用GL_LINES、GL_LINE_STRIP或GL_LINE_LOOP指定渲染线图元即可。

1. 准备项目

如同渲染点图元实验,从OpenGLESFromScratch项目复制一份,将目录名称改为Line,如图3.14所示。

进入Line目录,打开项目OpenGLESFromScratch,并将项目改名为Line。具体操作方法请参考渲染点图元实验。

图3.14 复制初始项目为Line

2. 定义顶点数据

首先渲染线表,定义线表顶点数据,8个顶点,表示4条线段:

```
//定义线表顶点坐标
const float linesVertices[]={
    -0.8,0.6,
    -0.4,-0.8,
    -0.5,0.6,
    -0.1,-0.8,
    0.5,0.6,
    0.1,-0.8,
    0.8,0.6,
    0.4,-0.8,
};
```

3. 渲染线表

```
- (void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &linesVertices[0]);
    GLsizei vertexCount=sizeof(linesVertices)/(sizeof(float) * 2);
    glDrawArrays(GL_LINES, 0, vertexCount);
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

由以上代码可见，渲染线表与渲染点表并无太大区别，仅仅在于调用 glDrawArrays 函数时，第一个参数不同，渲染点表使用 GL_POINTS，而渲染线表则使用 GL_LINES。为了代码的可读性，顶点数组的变量名也有所修改，但它并不影响代码的运行。

4. 运行

以上代码的运行结果是渲染了 4 条线段，如图 3.15 所示。

5. 设置线宽

默认的线宽值是 1.0，可以使用以下函数改变线宽：

```
void glLineWidth(GLfloat width);
void glLineWidthx(GLfixed width);
```

参数 width：指定渲染时的线条宽度，默认值为 1。

以下代码采用 30 个像素渲染线条，效果如图 3.16 所示。

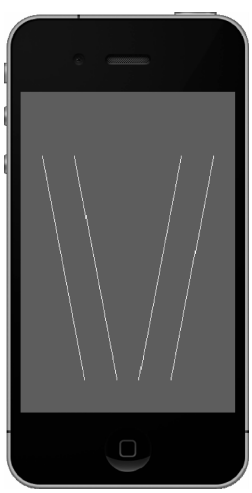

图 3.15　渲染线表

图 3.16　使用 30 像素线宽渲染线段

```
- (void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &linesVertices[0]);
    GLsizei vertexCount=sizeof(linesVertices)/(sizeof(float) * 2);
    glLineWidth(30.0f);                                    //设置线宽为 30 像素
    glDrawArrays(GL_LINES, 0, vertexCount);
    glLineWidth(1.0f);                                     //设置线宽为 1 像素(默认值)
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

6. 线的反走样

与点的反走样类似,通过调用带有参数 GL_LINE_SMOOTH 参数的 glEnable 或 glDisable 函数来启用或禁用它。

以下代码启用线的反走样,效果如图 3.17 所示。

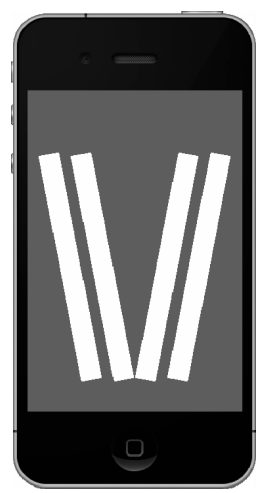

图 3.17 线的反走样

```
- (void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &linesVertices[0]);
    GLsizei vertexCount=sizeof(linesVertices)/(sizeof
    (float) * 2);
    glLineWidth(30.0f);                    //设置线宽为 30 像素
    glEnable(GL_LINE_SMOOTH);
    glDrawArrays(GL_LINES, 0, vertexCount);
    glDisable(GL_LINE_SMOOTH);
    glLineWidth(1.0f);                                     //设置线宽为 1 像素(默认值)
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

7. 渲染线条、线环

线条、线环的渲染与线表的渲染没有太大区别,要注意的是定义顶点数组时线段条数与顶点个数的关系。

以下代码使用 9 个顶点,在屏幕上方使用 GL_LINE_STRIP 方式渲染一条由 8 条线段组成的折线段;使用 4 个顶点,在屏幕中间使用 GL_LINE_LOOP 方式渲染一个由 4 条线段

组成的棱形,效果如图 3.18 所示。

图 3.18　渲染线条、线环

```
//定义线条顶点坐标
const float lineStripVertices[]={
    -0.8,0.7,
    -0.6,0.9,
    -0.4,0.7,
    -0.2,0.9,
    0.0,0.7,
    0.2,0.9,
    0.4,0.7,
    0.6,0.9,
    0.8,0.7,
};
//定义线环顶点坐标
const float lineLoopVertices[]={
    0.0, 0.4,
    0.6, 0.0,
    0.0,-0.4,
    -0.6, 0.0,
};

-(void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &linesVertices[0]);
    GLsizei vertexCount=sizeof(linesVertices)/(sizeof(float)*2);
    glLineWidth(30.0f);                                    //设置线宽为 30 像素
    glEnable(GL_LINE_SMOOTH);
    glDrawArrays(GL_LINES, 0, vertexCount);
    glDisable(GL_LINE_SMOOTH);
    glLineWidth(1.0f);                                     //设置线宽为 1 像素(默认值)

    //使用 GL_LINE_STRIP 渲染折线段
    glVertexPointer(2, GL_FLOAT, 0, &lineStripVertices[0]);
    vertexCount=sizeof(lineStripVertices)/(sizeof(float)*2);
    glDrawArrays(GL_LINE_STRIP, 0, vertexCount);
    //使用 GL_LINE_LOOP 渲染菱形
    glVertexPointer(2, GL_FLOAT, 0, &lineLoopVertices[0]);
    vertexCount=sizeof(lineLoopVertices)/(sizeof(float)*2);
    glLineWidth(20.0f);
    glEnable(GL_LINE_SMOOTH);
    glDrawArrays(GL_LINE_LOOP, 0, vertexCount);
```

```
    glDisable(GL_LINE_SMOOTH);
    glLineWidth(1.0f);
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

3.2.5 三角形图元

三角形图元是3D物体的最重要的组成部分,这是因为如下。

(1) 一个三角形的顶点总是共面的,因为三点确定一个平面。

(2) 三角形是不可能内凹的。

(3) 三角形不可能自相交。

(4) 任何多边形都可以分解为一些三角形的组合。

通过组合三角形可以形成更大、更复杂的多边形和模型。

图3.19演示了一个立方体的构成,立方体的每个面由两个三角形构成。还可以在其表面应用纹理和材质,从而使其看起来更具有立体感。有关纹理和材质的内容在后面几章讨论。

使用三角形还可以组成表面看起来是光滑曲面的三维图元,图3.20演示了如何使用三角形构成一个球体。当在球体表面应用材质和光照后,球体看起来很真实,特别是使用高洛德着色模式时。

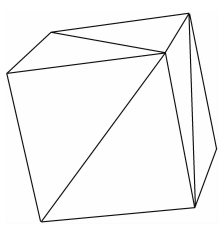

图3.19 用12个三角形组成一个立方体

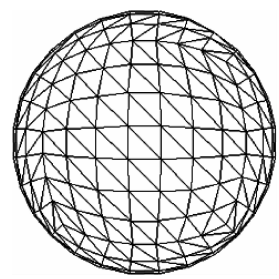

图3.20 使用三角形组成球体

三角形图元包括三种不同的拓扑结构,分别是三角形表(GL_TRIANGLES)、三角形条(GL_TRIANGLE_SRIP)和三角形扇(GL_TRIANGLE_FAN)。

三角形列表简称三角形表,表示孤立的三角形的列表至少有三个顶点,并且顶点总数能被三整除,如图3.21所示。

三角形条带简称三角形条,它是一系列相互连接的三角形。因为连在一起,不需为每个三角形重复指定所有三个顶点,最开始的三个顶点绘制成一个三角形,然后将其后所指定的每一个顶点与紧挨着它的前两个顶点绘制成新的三角形,这就意味着在第一个三角形之后,每一个点都对应这一个三角形。如图3.22所示,只需要5个顶点就能画出4个三角形。

大多数三维场景中的对象都是用三角形条带构成的。这是因为三角形条带可以高效地利用内存和运行时间渲染出复杂的对象。

三角形扇形简称三角形扇,与三角形条很相似,不同之处是,三角形扇共享一个顶点。

最先的一个顶点被认为是第一个公共中心顶点，随后的每一对顶点与此公共中心顶点形成一个新的三角形。如图 3.23 所示，使用 5 个顶点渲染出 4 个三角形。

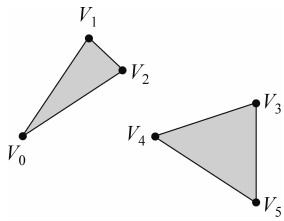

图 3.21　三角形表

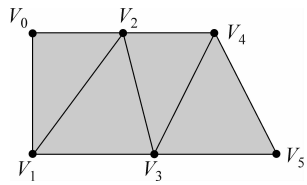

图 3.22　三角形条

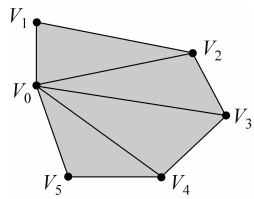

图 3.23　三角形扇

3.2.6　渲染三角形图元

在 3D 空间中渲染三角形图元与渲染点图元、线图元相比，在技术上没有什么本质区别，在调用 glDrawArrays 时，使用 GL_TRIANGLES、GL_TRIANGLE_SRIP 和 GL_TRIANGLE_FAN 指定渲染三角形图元即可。

1. 准备项目

如同渲染点图元实验，从 OpenGLESFromScratch 项目复制一份，将目录名称改为 Triangle，并将项目名称改为 Triangle，具体操作参考之前的实验。

2. 定义三角形顶点数据

首先准备渲染三角形表，定义三角形顶点数据，共 9 个顶点，表示 3 个三角形：

```
const float trianglesVertices[]={
    -0.6,-0.4,
    -0.4,-0,
    -0.2,-0.4,

    0.6,-0.4,
    0.4,-0,
    0.2,-0.4,

    -0.2,0,
    0.0,-0.4,
    0.2,0,
};
```

3. 渲染三角形列表

```
-(void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    //渲染三角形表
```

```
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &trianglesVertices[0]);
    GLsizei vertexCount=sizeof(trianglesVertices)/(sizeof(float) * 2);
    glDrawArrays(GL_TRIANGLES, 0, vertexCount);
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

4. 运行

运行以上代码，结果在屏幕上渲染了三个独立的三角形，如图 3.24 所示。

5. 三角形面的剔除

虽然三角形无限薄，但是它还是有两个面，这就表示可以从两个方向观察它。有时需要对其两面进行不同的显示绘制。当预先知道三角形只有一个面是可见的时候，就没有必要对其两面都进行绘制。例如一个球，其内部就是永不可见的。通过剔除(Culling)处理，可以通知 OpenGL ES 不去绘制某一面，这样可以节省 OpenGL ES 的渲染开支。

通过调用带有 GL_CULL_FACE 参数的 glEnable 或者 glDisable 函数以启用或禁用面的剔除，默认情况下是禁止剔除的。

通过调用以下函数来指定剔除哪个面：

```
void glCullFace(GLenum mode);
```

参数 mode：指定剔除三角形的正面，还是背面。可以为 GL_FRONT、GL_BACK、GL_FRONT_AND_BACK 三个参数。默认情况下是剔除背面(GL_BACK)。

注意：若指定为 GL_FRONT_AND_BACK，三角形将不会被渲染，但是其他图元如点、线的渲染不受影响。

如何确定三角形的正面和背面呢？依据是三角形的绕向(Winding)，也就是所指定顶点的顺序。默认情况下，OpenGL ES 将三角形逆时针绕向的面作为正面，而顺时针方向的面作为背面。

可以通过以下函数来改变这一默认设置：

```
void glFrontFace(Lenum mode);
```

参数 mode：指定三角形正面顶点绕向。可以为 GL_CW 或 GL_CCW。默认情况下，为逆时针(GL_CCW)。

以下代码启用了剔除功能，并将正面三角形剔除掉，渲染的结果如图 3.25 所示。请对照先前定义顶点数据的代码，思考为何只看到一个三角形了。

```
- (void) drawView
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
```

图 3.24　渲染三角形列表

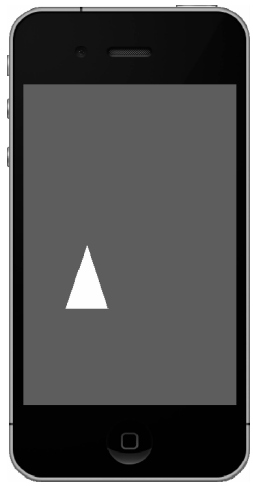

图 3.25　剔除三角形正面

```
    //渲染三角形表
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &trianglesVertices[0]);
    GLsizei  vertexCount=sizeof(trianglesVertices)/(sizeof(float) * 2);
    glEnable(GL_CULL_FACE);                    //打开面剔除功能
    glCullFace(GL_FRONT);                      //剔除正面
    glDrawArrays(GL_TRIANGLES, 0, vertexCount);
    glCullFace(GL_BACK);                       //剔除背面(恢复到默认值)
    glDisable(GL_CULL_FACE);                   //禁止面剔除功能(恢复到默认值)
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

6. 渲染三角形条

首先定义顶点数据,共 7 个顶点,表示 5 个三角形:

```
const float triangleStripVertices[]={
    -0.6,0.6,
    -0.4,0.2,
    -0.2,0.6,
    0,0.2,
    0.2,0.6,
    0.4,0.2,
    0.6,0.6,
};
```

修改 drawView 方法,以渲染三角形条:

```
-(void) drawView
```

```
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    //渲染三角形表
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, &trianglesVertices[0]);
    GLsizei  vertexCount=sizeof(trianglesVertices)/(sizeof(float)*2);
    glEnable(GL_CULL_FACE);                    //打开面剔除功能
    glCullFace(GL_FRONT);                      //剔除正面
    glDrawArrays(GL_TRIANGLES, 0, vertexCount);
    glCullFace(GL_BACK);                       //剔除背面(恢复到默认值)
    glDisable(GL_CULL_FACE);                   //禁止面剔除功能(恢复到默认值)
    //渲染三角形条
    glVertexPointer(2, GL_FLOAT, 0, &triangleStripVertices[0]);
    vertexCount=sizeof(triangleStripVertices)/(sizeof(float)*2);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, vertexCount);
    glDisableClientState(GL_VERTEX_ARRAY);

    [m_context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

运行结果如图 3.26 所示,在屏幕的上方渲染出一个梯形,该梯形由 5 个相连的三角形组成。

7. 渲染三角形扇

最后演示三角形扇的渲染,如图 3.27 所示,在屏幕中央处,渲染一个六边形。

图 3.26 使用三角形条渲染梯形

图 3.27 使用三角形渲染六边形

鉴于该几何图形相当具有规则,而用手工编码则由于各个顶点的坐标既不精确,又比较麻烦,故采用代码自动生成顶点坐标数据。

要组成一个完整的六边形，需要 6 个三角形，这 6 个三角形由 8 个顶点构成，第一个点作为共享顶点位于中心，其他 7 个顶点分布于四周。

这 8 个顶点需要占用 8×2＝16 个浮点数。因此定义一个 16 个元素的浮点型数组。

第一个顶点的坐标手工指定为(0,0)，其他顶点坐标使用循环生成，代码如下：

```
float triangleFanVertices[8*2];
triangleFanVertices[0]=0;
triangleFanVertices[1]=0;

for (int i=0; i<7; i++) {
    triangleFanVertices[(i+1)*2]=0.3*cos(M_PI/3.0*(float)i);
    triangleFanVertices[(i+1)*2+1]=0.2*sin(M_PI/3.0*(float)i);
}
```

注意：以上代码中，x 坐标为 0.3 乘以该角度的余弦值，而 y 坐标为 0.2 乘以该角度的正弦值，这不是笔误，请思考原因。

接下来渲染：

```
glVertexPointer(2, GL_FLOAT, 0, &triangleFanVertices[0]);
vertexCount=sizeof(triangleFanVertices)/(sizeof(float)*2);
glDrawArrays(GL_TRIANGLE_FAN, 0, vertexCount);
```

运行结果如图 3.27 所示。

3.3 思考题

1. OpenGL ES 支持哪几种不同拓扑结构的基本图元？
2. 试找出不同拓扑结构的图元数与顶点数的关系？
3. 如何设置渲染点的大小？
4. 如何设置线的反走样？
5. 什么是三角形的剔除？

游戏循环

任何一款游戏，其界面都是动态的，不可能静止不动。而在第3章中渲染的图形都是静态的。事实上，我们只在GLView类的initWithFrame方法中调用了drawView方法一次，这就好比我只是拿出了一张照片给人们看，这张照片上的画面是不会动起来的。要使画面动起来，就需要像放电影一样，快速地不断地向人们展示不同的照片，每张照片略有不同，比如人物的位置稍有移动。游戏中，要使画面动起来，就需要不断地渲染画面，也就是要不断地调用drawView函数，而在drawView函数中，需要根据时间的变化，使画面做相应的变化，从而产生动画效果。

在一个循环中不断地调用drawView函数，这个循环便是“游戏循环”，它是维持游戏运转的心脏。游戏循环除了驱动画面的渲染之外，还执行以下一系列有规律运行的任务。

(1) 处理用户输入(触控、加速计等)。

(2) 更新游戏状态。

(3) 更新游戏实体的AI(人工智能)。

(4) 更新游戏实体的位置。

(5) 播放背景音乐和音效。

看起来需要执行的任务不少，特别是对于一款包含很多游戏元素的游戏来说，以上每一步的处理都需要花费一定的时间。因此，游戏循环不仅需要确保这些步骤的执行，还要保证游戏运行速度的稳定。

4.1 基本的游戏循环

基本游戏循环的伪代码如下：

```
BOOL gameRunning=true;
while(gameRunning)
{
    updateGame;
    renderGame;
}
```

以上代码中，游戏循环会不停地更新游戏并将游戏画面渲染到屏幕上，直到gameRunning变量的值变为false为止。

尽管以上代码能够运行，但它有个致命的缺点：没有考虑时间控制，导致游戏运行速度与硬件速度有关。在硬件速度较慢的设备上，游戏运行会很慢，而在硬件速度很快的设备上，游戏运行又会过快。这会让游戏玩家的用户体验大打折扣。因此，既不能让游戏运行过于迟钝，也不能让游戏运行快到玩家无法操控。

这就是为什么在游戏循环中需要解决时间控制问题的原因之一。

不仅是总的游戏体验决定了时间控制的重要性，更重要的可能是如“碰撞检测”等功能的性能。在游戏对象相互碰撞时能及时检测出来相当重要，这也是人们用到的一个基本游戏机制。

若游戏实体更新过程中运动过快，它就可能在下一次碰撞检测运行前穿越其他物体。在游戏更新时让游戏实体以恒定的速度运动有助于减少碰撞漏检的概率。

衡量游戏循环速度的标准通常有两个指标，具体情况如下所示。

(1) 渲染速度，单位为每秒帧数(FPS)：FPS 与游戏画面在屏幕上每秒渲染的次数有关。对于 iPhone 来说，极限速度为 60FPS，也就是屏幕刷新的最大速率。在上述伪代码中，与 renderGame 方法的调用次数有关。

(2) 更新速度：更新速度即游戏实体更新的频率。在上述伪代码中，与 updateGame 的调用次数有关。

4.2 几种常见的游戏循环体

4.2.1 基于帧的循环体

这是最简单的一种游戏循环类型，在这种循环体中，游戏在每一个游戏周期内进行一次更新和渲染。基于帧的循环体的优点是实现起来相当简单。

例如：游戏实体的移动：

```
position.x ++;
```

基于帧的循环体的缺点是硬件设备的帧速度直接影响游戏的运行速度，硬件越快，游戏运行就越快，反之，游戏运行速度就越慢。

图 4.1 展示的是游戏帧在高速硬件上和低速硬件上不同的运行速度。速度的差别会非常明显，导致玩家的游戏体验差别也非常明显。该图也展示了基于帧的循环体中，每一帧画面只执行一次更新和一次渲染。

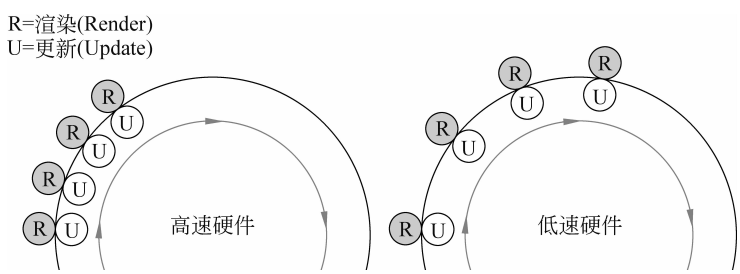

图 4.1 游戏实体在高速硬件和低速硬件上沿着圆弧路径运行

4.2.2　基于时间的不定间隔循环体

基于时间的不定间隔循环体与基于帧的循环体相似，但它会计算运行时间。对游戏实体更新时，会根据本次更新距上次更新经过了多少时间来计算其位移。从而允许实体以恒定的速度移动而不受硬件速度的影响。

例如，让实体以 1 单位/秒的速度移动：

```
position.x +=1.0f * delta;
```

虽然这个方法解决了游戏在不同硬件速度下运行速度不同的问题，但是也导致了新的问题。大多数情况下，delta 值相对较小且保持恒定，但是很容易发生某些干扰使得 delta 值显著增大从而带来如碰撞漏检等问题。比如玩家正在玩游戏时，突然打进来一个电话，游戏会被切换出去，等接完电话再切换回游戏界面时，delta 值就会变得非常大。

如图 4.2 所示，在一个游戏周期内，游戏实体在圆弧路径中可前进一步。可以看到，在该图中，如果 delta 值很小，则实体最终会撞上途中的物体，从而触发碰撞检测机制。

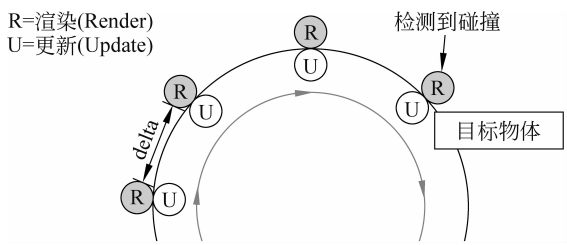

图 4.2　具有小 delta 值的帧

然而，如果 delta 值增大，则图 4.3 所示的情况就会发生。这里，游戏实体以恒定的速度移动，但帧速度降低(delta 值增大)，造成游戏实体与物体的碰撞无法检测到(物体被穿越)。

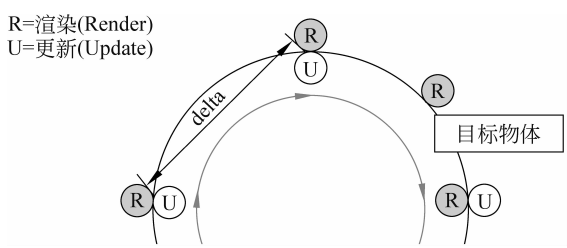

图 4.3　具有较大 delta 值的帧

4.2.3　基于时间的固定间隔循环体

此方法解决了基于时间的不定间隔循环体所带来的 delta 值突然增大造成碰撞漏检等问题。它既可以像基于时间不定间隔方法一样保证游戏稳定的运行速度，也消除了前面谈到的碰撞漏检等问题。如果已经采用基于帧的循环体，或者基于时间的不定间隔循环体，只需要稍做修改，即可切换到基于时间的固定间隔循环体。

基于时间的固定间隔循环体的核心思想是，让游戏状态在一个游戏周期内以固定时间