Ork 3.1 文档

1. 简介

Ork 是一个小型 OpenGL 渲染内核。不能与 Ogre 库混淆：尽管 Orks 和 Ogres 有一些共同的特征，但 Orks 更小 :-) 。Ork 库提供了与 3D 渲染相关的核心功能：向量和矩阵的数学计算类，OpenGL API 的面向对象视野，从硬盘加载渲染资源如格网，纹理或贴图的框架。它还提供了一个基于计划任务框架的超小的通用"玩具"场景图来描述 3D 场景。

2. Core 类

Ork 的 Core 类在 core 和 math 模块中都有提供。第一个模块提供了内存管理机制（称为智能指针框架），以及日志消息，测量时间或 STL 集合迭代次数的通用类。第二个模块提供了 2D 和 3D 中线性代数相关的类（向量和矩阵）。

2.1 智能指针

core 中主要的类是 orkObject 类，以及相关的 orkptr 和 orkstactic_ptr 模板。这些类提供了一个智能指针框架，如，确保对象不再被引用时的自动注销。

注意：

- 通过 USE_SHARED_PTR 预处理器标识符， ork::ptr 扩展了 std::tr1::shared_ptr；不使用此标识符时，ork::ptr 则完全在 Ork 中执行（使用 ork::Object 内建计数器）。注意，使用 USE_SHARED_PTR 标识符时，在应用中既可以使用 ork::ptr 也可以使用 std::tr1::shared_ptr: 因为 ork::ptr 是 的子类，一个 Ork 函数返回一个 ork::ptr 的结果可以显式的存储在 std::tr1::shared_ptr 中。反过来，显示地传递 std::tr1::shared_ptr 到 Ork 函数中来请求一个 ork::ptr ，因为有一个隐式的 ork::ptr 构造函数使用 std::tr1::shared_ptr 作为参数。

所有继承于 ork::Object 的类都有这种自动销毁机制。这些类的所有实例都有一个引用计数器，在 ork::Object 中声明，每次一个引用对象的新的 ork::ptr 或 ork::static_ptr 被创建时增加（删除时减少）。例如，如下代码：

```
void f() {
    ptr<Object> o = new Object(); // shortcut for ptr<Object>(new Object());
// ...
}
```

一个对象连同新的操作符在大块内存上被创建时，它的引用计数器随着 ptr 引用的增加而增加。在函数末尾引用的析构函数自动被调用（如同 C++ 中的定义），对象的引用计数器也减少。如果计数为空，那么对象的析构函数被调用。

注意：

- 感谢 C++ 的隐式转换规则，你可以像使用普通指针那样使用智能指针。尤其是你可以像上文那样写 `ptr<Object> o = new Object();` 来替代使用显示构造函数调用 `ptr<Object> o = ptr<Object>(new Object());` . ork:ptr 类也定义了 `=, ==, !=, ->` 操作符，以至于你可以分配，比较和解引用智能指针。最后你可以使用 `o.cast<T>()` 抛一个智能指针到另一个智能指针类型 `ptr<T>` (注意如果 D 是 C 的子类， `ptr<D>` 可以在任何 `ptr<C>` 期望的地方使用，而不用抛给它)
- ork::static_ptr 模板和 ork::ptr 类似，但是是静态变量专用。在程序结尾调用 ork::Object::exit 可以设置所有此类静态引用为 NULL ，这将销毁所有被静态引用的对象。注意，在调试编译模式中， exit 方法检查所有被销毁的 ork::Object 实例，这对检测内存溢出很有用。一种确保这个方法被调用的方式是在程序结尾使用 `atexit (Object::exit);` 。
- 你可以通过重写 ork::Object::doRelease 方法来改变当一个对象的引用计数器变为 0 时，对象销毁的默认行为。

限制条件：

- 在同一个对象上不要混用智能指针和普通指针（如果用了请非常小心）。否则引用计数器会不准确，当仍有普通指针引用时对象可能会被销毁。例如：

```
void f(ptr<Object> o);
void g() {
    Object *o = new Object();
    f(o); // shortcut for f(ptr<Object>(o));
delete o; // Error: double delete!
}
```

当 o 传递给 f 时，一个智能指针被隐式创建，因此函数将智能指针作为参数。 o 的计数器加 1 （不是 2 ，因为第一个引用是普通指针）。但是在调用之后，这个智能指针也被隐式地销毁了，计数器变为 0 ，因此自动删除 o 。显示地删除导致了一个错误，因为对象已经

被删除掉了。
- 不要创建智能指针的循环。实际上，如果对象之间的互相引用循环已经不在引用，循环中的每个对象仍被其他对象引用着。因此整个循环不能被删除，导致内存溢出。避免这种循环的一个办法是在循环的某处使用普通指针（这样做要非常小心 - 见上文）。如下：

```
class C
{
public:
    ptr<D> d;
    ~C() {
        d->c = NULL; // deletes normal pointers to this
    }
};
class D
{
public:
    C* c; // using ptr<C> would create smart pointer cycles!
};
```

因为在 c 在 D 中是一个普通指针，而且由于类 C 的对象通过智能指针在别的所有地方都被管理， D 中被 c 引用的对象能随时被删除。为了避免通过这种引用访问一个被删除的对象， c 在自己的析构函数中擦出了对自己的引用。我们称 c 为一个 弱指针 ，因为引用的对象可以随时被删除（相比之下，智能指针被称为 强指针 ，因为只有当引用被设置为 NULL 时，对象才能被删除）。

2.2 消息日志

Ork 日志框架可以用于记录消息。该消息可以是错误消息，警告消息，提醒消息或调试消息。它有标题和内容。主类是 ork::Logger 。该类在静态变量中为错误，警告，提醒和调试存储多个日志，每个目录一个。它也定义了一个 ork::Logger::log 方法，在给定标题下记录消息。用法如下：

```
if (Logger::INFO_LOGGER != NULL) {
    Logger::INFO_LOGGER->log("my topic", "my message");
}
```

ork::Logger 类输出消息为标准输出。 ork::FileLogger 子类将消息写入 HTML 文件。ork::FileLogger 能链接到其他记录者。那么将可以同时在标准输出和 HTML 文件中记录消息。也可以自定义日志子类。使用 ork::Logger::addTopic 方法，所有记录者可以配置为仅将消息输出到相关的一个或多个标题下（默认它们都输出所有消息，无论标题是什么）。标题在 Ork 库中被定义如下：

- CORE ：智能指针消息
- OPENGL ：OpenGL消息
- LINKER ：OpenGL 贴图链接消息

- COMPILER ：OpenGL 贴图编译器消息
- RESOURCE ：资源消息
- SCENEGRAPH ：场景图消息
- SCHEDULER ：计划任务消息

2.3 数学

math 模块提供了 2D 和 3D 中线性代数有关的类（向量和矩阵）。

- ork::math::vec2, ork::math::vec3, ork::math::vec4, ork::math::mat2, ork::math::mat3 和 ork::math::mat4 模板描述了 2D, 3D 和 4D 向量和矩阵。它们提供了所有的通用操作，如向量和矩阵的加法与减法，矩阵乘向量和矩阵乘矩阵，矩阵除法，矩阵转置，向量归一化，向量点积和向量叉积等。它们可以被实例化为 `float`，`double`，甚至 `int` 类型。一些实体如 ork::math::vec3f, ork::math::vec3d, ork::math::mat3f, ork::math::mat3d 等已经被预定义。
- ork::math::box2 和 ork::math::box3 模板描述了 2D 和 3D 边界盒。它们提供了函数来扩展边界盒，并且测试盒子是否包含一个点或其他盒子，或与其他盒子交叉。

3 渲染框架

render 模块提供了 Ork 渲染框架，一个 OpenGL API 的面向对象视野。 对 C++ 程序员来说，使用 OpenGL API 非常困难，不自然。第一个原因是 OpenGL “对象”简单的用标识符描述，而不是类的实例。第二个也许是最重要的原因是这些“对象”在使用之前必须被放入一个“范围”。实际上，这些特性与面向对象语言的基本原则是不兼容的。 Ork 通过在 OpenGL 顶层提供一个小型的面向对象 API 来解决此问题。指如下三条：

- 其一，Ork 基于 OpenGL 3.3 核心配置文件（部分支持 4.0 和 4.1）。为了保持小型化，兼容的配置被设计时排除在外。
- 其二，OpenGL 的帧缓冲，程序，着色器，一致变量，纹理，取样器，缓冲区，查询指令等都用萃取 C++ 类重写。
- 其三，最重要的，这些类封装了 OpenGL 不堪的细节。特别是隐藏了对象标识符，以及在能够使用它之前需要绑定到对象的做法。

这位程序员带来了许多好处，如下面例子所示。假设你用一个纹理程序，想在屏外帧缓冲中画一个格网。假定对象已经创建好，使用 OpenGL API ，你需要：

- 设置程序为当前程序： `glUseProgram(myProgram);`
- 选择纹理单元并设置为当前纹理： `glActiveTexture(GL_TEXTURE0 + myUnit);`
- 绑定纹理到该单元： `glBindTexture(GL_TEXTURE_2D, myTexture);`
- 绑定纹理单元到程序： `glUniform1i(glGetUniformLocation(myProgram, "mySampler"), myUnit);`
- 设置格网 VBO 为当前 VBO： `glBindBuffer(GL_ARRAY_BUFFER, myVBO);`
- 指定 VBO 中顶点的格式： `glVertexAttribPointer(0, 4, GL_FLOAT, false, 16, 0);`
- 在 VBO 中启用顶点属性： `glEnableVertexAttribArray(0);`

- 设置帧缓冲为当前帧缓冲： `glBindFramebuffer(GL_FRAMEBUFFER, myFramebuffer);`
- 最后！画这个 VBO： `glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);`

使用 Ork API 你只需简单的两步（第一步在每次画之前不需要被重复，除非你想画一个不同的纹理）：

- 设置程序一致变量的值： `myProgram->getUniformSampler("mySampler")->set(myTexture);`
- 在帧缓冲中用程序画格网： `myFramebuffer->draw(myProgram, *myMesh);`

在封装之下， Ork 自动选择一个纹理单元，绑定纹理，再绑定单元到程序，设置当前程序，设置当前格网 VBO ，设置当前帧缓冲等等。而且， Ork 自动最小化了执行此 API 时 OpenGL 的状态变化（没有重排序的 API 调用：比如， Ork 将不会调整以下顺序 `myFBO->draw(myProgram1, *myVBO); myFBO->draw(myProgram2, *myVBO); myFBO->draw(myProgram1, *myVBO); myFBO->draw(myProgram2, *myVBO);` 来最小化 `glUseProgram` 的调用；然而，对序列 `myFBO->draw(myProgram1, *myVBO); myFBO->draw(myProgram1, myVBO); myFBO->draw(myProgram2, *myVBO);myFBO->draw(myProgram2, *myVBO);` 来说， Ork 仅调用 2 次 `glUseProgram` ，而不是 4 次）

Ork 渲染框架被组织为一些概念：通过参数（少数仍为传递中的固定部分）配置的 *帧缓冲* 被用于将 *格网* 画到一个或多个用一个 *模块* 集链接到 *程序* 上的 *渲染目标*（可以是纹理）上。格网定义了一个由用户自定义的，通过 一致变量 配置，程序处理的顶点 *属性* 集（如法线和颜色没有预定义的属性），每个顶点一个。

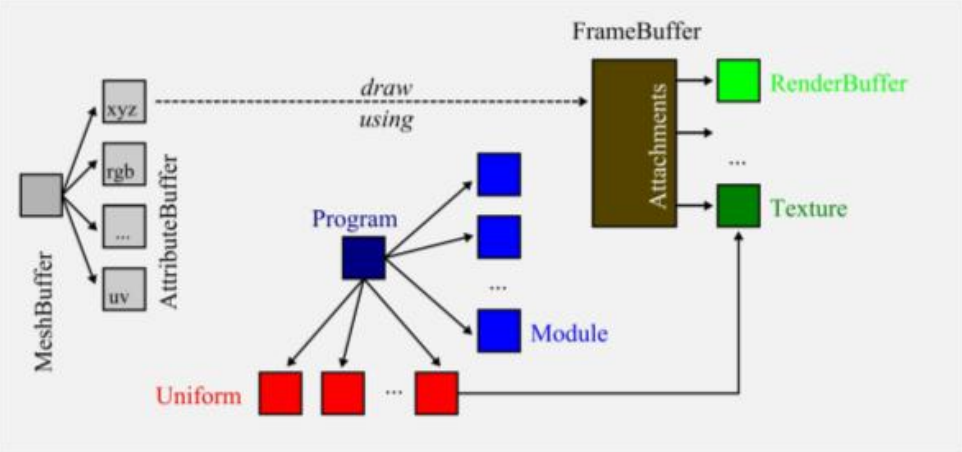

图1：Ork 渲染框架

渲染框架为所有的元素都定义了类。这些类扩展了 ork::Object 类，以至于当它们不再使用时被 GPU 资源自动删除（例如当 ork::Texture 对象不再被引用而删除时，同时删除 GPU 显存中相应的 OpenGL 纹理）

渲染 API 完全涵盖了 OpenGL 3.3 的核心配置，包括部分的 OpenGL 4.0 和 4.4 的核心配置 APIs （支持镶嵌着色器，但一致变量子程序，二进制着色器和程序，管道对象，分离着色器，和多视场目前不支持）。

3.1 格网

格网是拓扑图形（点，线，线段，三角形，三角条带等）中有组织的顶点列表。每个顶点有一些属性。属性由用户定义并可以是任何值（位置，法线，颜色，纹理坐标等）。属性在 ork::AttributeBuffer 中被集合到 ork::MeshBuffer 。一个属性缓冲描述了顶点属性的存储格式。该格网所有顶点的该属性的值没有存储在它自己的属性缓冲中，而是存储在 ork::Buffer 中。如果使用 ork::CPUBuffer ，该缓冲就存储在 CPU 寄存器，如果使用 ork::GPUBuffer ，就存储在 GPU 显存。

感谢 ork::AttributeBuffer 中的 stride 和 offset 参数，一个格网顶点的属性有多种组织方式。以下是一些可能性：

- 每个属性类型使用一个缓冲。例如使用一个缓冲存储所有顶点的位置，另一个存储颜色，另一个存储纹理坐标等。这时一些缓冲可以在 CPU 寄存器，而另一些在 GPU 显存。
- 使用一个缓冲存储所有属性。这时属性可以在缓冲中交叉或不交叉：
  - 非交叉模式时，缓冲由第一个属性的所有值开始，接着第二个属性的所有制，以此类推。这样的缓冲包含所有位置，然后所有颜色，然后所有纹理坐标等等。
  - 交叉模式时，缓冲先存储第一个顶点的所有属性，然后第二个顶点，如此直到最后一个。这样的缓冲包含第一个顶点的位置，颜色，纹理坐标，第二个顶点的位置，颜色，纹理坐标，等，以此类推。

此灵活性的优点使得你可以按你所愿组织你的格网数据，但是它增加了格网生成的复杂度。为了简化在单一缓冲中使用交叉存储模式生成格网，可以使用 ork::Mesh 模板。例如：

```
ptr< Mesh<vec4f, unsigned byte> > m;
m = new Mesh<vec4f, unsigned byte>(TRIANGLE_STRIP, GPU_STATIC);
m->addAttributeType(0, 2, A32F, false);
m->addAttributeType(1, 2, A32F, false);
m->addVertex(vec4f(-1, -1, 0, 0));
m->addVertex(vec4f(1, -1, 1, 0));
m->addVertex(vec4f(-1, 1, 0, 1));
m->addVertex(vec4f(1, 1, 1, 1));
```

用拓扑三角带生成格网，每个顶点表现为 `vec4f` 结构（可以使用任何自定义结构描述顶点）。每个顶点两个属性，由 ork::Mesh::addAttributeType 描述。第一个由两个浮点型组成，与第二个一样（可以解释为一个 2D 位置 和 一个 2D 纹理坐标）。顶点由 ork::Mesh::addVertex 定义，使用类型作为模板参数传递。 ork::MeshBuffer 结果可以从 ork::Mesh::getBuffer 提取。

3.1.1 属性标识符

每个顶点属性都有一个表现为整型的标识符（由 ork::AttributeBuffer 构造函数指定，或 addAttributeType 方法）。必须在着色器中使用 location 布局限定符来指定这些标识符。比如，使用 `layout(location=0) in vec3 position;` 。

3.1.2 格网索引

一个由三角拓扑定义的格网必须被定义为，由给定的第一个三角形的 3 个顶点，接着第二个三角形的 3 个顶点，如此直到最后一个三角形。这样导致空间浪费，因为一个顶点被相邻的多个三角形共享而重复多次（其他拓扑图形也有同样的问题）。限制这种重复的一个方法是使用三角带，但并不总是方便。另一个方法是使用格网索引。这时顶点在一个数组中被描述一次，然后组成三角形的顶点被数组中的索引描述。

这种格网在 ork 中使用 ork::MeshBuffer::setIndicesBuffer 定义。这个方法用 ork::AttributeBuffer 作为参数，描述了其他属性缓冲中的格网顶点的索引，在顶点数组查看。这也可以在 ork::Mesh 中使用 ork::Mesh::addIndice 方法完成（顶点索引的格式由 ork::Mesh 模板的第二个参数指定）。

3.1.3 修改格网

通过修改包含实际顶点数据的 ork::Buffer 可以修改格网。在 ork::CPUBuffer 中你可以直接修改 CPU 寄存器中缓冲的内容。在 ork::GPUBuffer 中，既可以使用 ork::GPUBuffer::setData 也可以使用 ork::GPUBuffer::setSubData 方法，或者使用 ork::GPUBuffer::map 映射缓冲内容到 CPU 寄存器，然后在反映射前直接修改此映射的缓冲。

3.1.4 绘制格网

一个格网必须在 ork::FrameBuffer::draw 方法中，要么使用 ork::MeshBuffers 参数，要么使用 ork::Mesh 参数来绘制。第一种情况下，你可以指定一个和 ork::MeshBuffers 本身不同的格网拓扑和顶点数（这可以用来绘制格网的一部分）。两种情况都可以指定格网必须被绘制的次数（这称为几何副本）。

3.2 模块

一个格网必须由链接到程序的着色器集合绘制。这些着色器使用顶点属性计算顶点在渲染目标中的投影坐标，使用颜色或其他数据在目标中写入。他们也可以使用一致变量，这些变量在格网渲染中属于常量。

Ork 中，ork::Program 是由一个或多个链接在一起的 ork::Module 组成。每个 ork::Module 对象集合了一些对象，如一个（或部分）顶点着色器，一个（或部分）镶嵌控制着色器，一个（或部分）镶嵌高程着色器，一个（或部分）几何着色器和一个（或部分）片段着色器（所有部分都是可选项）。这些部分必须要么单独定义在自己的文件中：

```
... vertex shader GLSL code ...
```

myModuleVS.glsl

```
... tessellation control shader GLSL code ...
```

myModuleTCS.glsl

```
... tessellation evaluation shader GLSL code ...
```

myModuleTES.glsl

```
... geometry shader GLSL code ...
```

myModuleGS.glsl

```
... fragment shader GLSL code ...
```

myModuleFS.glsl

要么所有都在一个文件中，但是被其后的预处理器指令（该选项可以在着色器之间减少冗余，常用在使用同一个一致变量和用户定义数据类型和函数时）分隔开：

```
 ... common GLSL code ...
#ifdef _VERTEX_
 ... vertex shader GLSL code ...
#endif
#ifdef _TESS_CONTROL_
 ... tessellation control shader code ...
#endif
#ifdef _TESS_EVAL_
 ... tessellation evaluation shader code ...
#endif
#ifdef _GEOMETRY_
 ... geometry shader GLSL code ...
#endif
#ifdef _FRAGMENT_
 ... fragment shader GLSL code ...
#endif
```

一个模块的着色器都在同一个单文件 myModule.glsl 中

其中每个 ifdef 部分都是一个选项（该代码包含在所有的顶点，镶嵌，几何和片段 OpenGL 着色器中，除非它内置在 indef 段落中。这时它仅被包含在相关的 OpenGL 着色器中）。共有代码可以包含一致变量，和顶点，镶嵌，几何以及片段部分共有的数据类型。

ork::Program 是由一个或多个链接在一起的 ork::Module 组成。使用多个 ork::Module 创建一个程序可以定义模块化代码，一个模块使用在另一个模块中实施的函数。例如，可以在 “wood” ork::Module 中这样使用：

```
... common GLSL code ...
#ifdef _VERTEX_
 ... vertex shader GLSL code ...
#endif
#ifdef _FRAGMENT_
layout(location=0) out vec4 color;
// function prototype, no implementation
vec3 illuminate(vec3 p);
void main() {
   vec3 p = ... // position
   vec3 light = illuminate(p);
   color = ... // compute reflected light
}
#endif
```

使用 `illuminate` 抽象函数计算 p 点的入射光，并且一个 “point light” 模块定义了该入射光如何被计算：

```
uniform vec3 lightPos;
uniform vec3 lightColor;
vec3 illuminate(vec3 p) {
    vec3 v = p - lightPos;
    return lightColor / dot(v, v);
}
#ifdef _VERTEX_
#endif
#ifdef _FRAGMENT_
#endif
```

一个优点是你可以定义其他版本的 `illuminate` 函数，例如对一个斑点，你可以结合 “wood” 模块而不需要重写它。注意我们是在共用段落中定义的 `illuminate` 函数，所以我们可以在每个顶点的着色器中加光，或每个像素的片段着色器中加光。另一个优点是可以结合 “point light” 和 “spot light” 模块以及其他 “材料”模块，例如“大理石”模块（这意味着一个 ork::Module 可以被多个 ork::Program 使用）。因此在所有材料着色器之间可以避免代码冗余。

3.2.1 一致变量

一个程序的一致变量由 ork:Uniform 类和它的子类表示。可以以 ork::getUniform 和 getUniformXxx 方法提取。一致变量区块内部的一致变量可以用同一种方法提取，或通过 ork::Program::getUniformBlock 和 ork::UniformBlock::getUniform 两步提取。

和在 OpenGL 中一样， ork::Uniform 的值是“稳定的”，意味着该值永远保持不变除非你用 ork::Uniform::setValue （或 ork::Uniform 子类中指定 set 方法）修改它。

一致变量区块中定义的一致变量设置的值和默认区块中的一致变量一样。在封装下，Ork 自动为一致变量区块创建 ork::GPUBuffer 对象，在需要的是向主寄存器映射和反映射。

重要：为了最好的表现，不应在渲染循环中使用 ork::Program::getUniform ，而应照如下所

做：

```
... initialization
for (each frame) {
    ...
    myProgram->getUniform1f("x")->set(...);
    myFrameBuffer->draw(myProgram, *myMesh);
    ...
}
```

使用：

```
... initialization
ptr<Uniform1f> x = myProgram->getUniform1f("x");
for (each frame) {
    ...
    x->set(...);
    myFrameBuffer->draw(myProgram, *myMesh);
    ...
}
```

3.3 纹理

OpenGL 纹理在 Ork 中用 ork::Texture 类和其子类表现。每个纹理都有一个*内部格式*来描述其每个像元组份（或颜色通道）的数量，以及每个组份的字节数。每个纹理都有一个参数集，用 ork::Texture::Parameters 表示。在 Ork 中，纹理参数是不变的，如只有纹理的内容能在运行时改变。

纹理参数在纹理构造函数中指定。一个初始化纹理内容也可以在构造函数中使用 ork::Buffer 对象指定。 CPUBuffe(NULL) 可以撤销内容初始化。 ork::GPUBuffer 可以对已经存在于 GPU 上的缓冲做纹理内容初始化（缓冲的内容复制到纹理，而不是引用）。纹理内容可以用 ork::Texture::getImage 方法读回 CPU 。

纹理内容有两种方法可以在运行时改变。第一种是复制从缓冲对象复制新内容，第二种是从帧缓冲附件中：

- setSubImage 方法指定了纹理的哪一个子部分必须被改变，新值由 ork::CPUBuffe 或 ork::GPUBuffer 提供。
- copyPixels 方法复制帧缓冲附件（由 ork::FrameBuffer::setReadBuffer 指定）的一部分到纹理的一个子部分。

3.3.1 压缩纹理

Ork 支持压缩纹理，如，纹理在 GPU 中的内容被压缩。有一些专门的方法来读写这些纹理内容：

- 在纹理构造函数中初始化内容，压缩纹理也一样，但需要提供在缓冲参数中的压缩纹理的大小，由 ork::Buffer::Parameters 类提供
- 纹理内容必须用 ork::Texture::getCompressedImage 方法读回 CPU 。压缩纹理大小可以用 ork::Texture::getCompressedSize 方法获取
- 纹理内容必须用 setCompressedSubImage 方法在运行时改变

3.3.2 一致变量取样器

一个纹理可以使用 ork::UniformSampler 对象非常简单的在一个程序中限定到一个一致变量取样器。例如，如果 p 是如下模块组成的程序：

```
uniform samplerCube envMap;
...
void main() {
   ...
   vec4 c = texture(envMap, d);
   ...
}
```

那么你可以像如下这样绑定纹理到它的 envMap 一致变量取样器：

```
ptr<TextureCube> t = ...
p->getUniformSampler("envMap")->set(t);
```

3.4 帧缓冲

ork::FrameBuffer 类用于表现默认帧缓冲和 OpenGL 帧缓冲对象。和 OpenGL 中一样，帧缓冲有一些附件。这些附件与他们的帧缓冲相关联。如，当一个帧缓冲用来绘制一个格网时，附属于该帧缓冲的纹理和渲染缓冲将自动成为新的当前渲染目标。 Ork 中这个思路被扩展到流水线状态（视场，模板和深度测试，剔除，融合和写状态等）。这意味着每个帧缓冲都有自己的流水线状态。当绘制一个格网时，该帧缓冲的流水线状态自动设置替换前一个帧缓冲的流水线状态。

帧缓冲附件由 ork::FrameBuffer::setRenderBuffer 和 ork::FrameBuffer::setTextureBuffer 定义。

帧缓冲流水线状态由 ork::FrameBuffer::Parameters 类表现并且用 ork::FrameBuffer 类的 getter 和 setter 方法修改。

默认帧缓冲是通过 ork::FrameBuffer::getDefault 静态方法接入的（帧缓冲附件不能被改变，但流水线状态可以被改变）。用户自定义帧缓冲由 ork::FrameBuffer 构造函数创建。

ork::FrameBuffer 的主要方法是绘制格网的 draw 方法（也有一个方便的绘制四边形的 drawQuad 方法）。也有一些方法用来清除渲染目标，从附件中读取，复制像素等：

- ork::FrameBuffer::readPixels 方法用于从帧缓冲附件（由 ork::FrameBuffer::setReadBuffer 选择）中复制像素到 ork::Buffer （ CPU 的或 GPU 的）中。
- ork::FrameBuffer::copyPixels 方法用于从帧缓冲附件（由 ork::FrameBuffer::setReadBuffer 选择）中复制像素到一个纹理。

3.4.1 多渲染目标

帧着色器可以同时写入多个帧缓冲附件。在这之前你必须为 setDrawBuffers 绘制方法选择附件。例如，选择 COLOR0 和 COLOR2 附件：

```
ptr<FrameBufer> fb = ...
fb->setDrawBuffers(FrameBuffer::COLOR0 | FrameBuffer::COLOR2);
```

3.4.2 与 OpenGL 交互

可以一起使用 Ork 和原生 OpenGL API。例如使用 Ork 中尚未支持的 OpenGL 特性。然而，由于 Ork 保留了很多关于当前 OpenGL 状态的内部静态变量供自己使用，在 Ork 外部直接调用 OpenGL 修改这些 OpenGL 状态会导致内部 Ork 状态和真实 OpenGL 状态的不一致。为避免此问题，你必须遵守如下所以的 OpenGL 直接调用的混合方法：

```
... // Ork code
FrameBuffer::resetAllStates();
... // direct OpenGL calls
FrameBuffer::resetAllStates();
... // Ork code
```

3.5 一个例子

本节给出了一个完整实例的代码，演示了如何使用纹理，模块，程序，格网和帧缓冲：

```
#include "ork/render/FrameBuffer.h"
#include "ork/ui/GlutWindow.h"
using namespace ork;
class SimpleExample : public GlutWindow
{
public:
    ptr< Mesh<vec2f, unsigned int> > m;
    ptr<Program> p;
    SimpleExample() : GlutWindow(Window::Parameters())
```

ork::GlutWindow 类提供了基本 GLUT 功能性的面向对象视野，称为窗口和用户接口事件回调函数。为了定义用户自己的应用的用户接口，这个类的方法可以被重写。这些方法是 redisplay, reshape, mouseClick, mouseMotion, keyTyped, specialKey, 和 idle 。你可以实例化这个子类来创建一个窗口。注意，如果初始化窗口的大小被设置为 (0,0) ，那么将会创建一个全屏窗口：

```
{
        m = new Mesh<vec2f, unsigned int>(TRIANGLE_STRIP, GPU_STATIC);
        m->addAttributeType(0, 2, A32F, false);
        m->addVertex(vec2f(-1, -1));
        m->addVertex(vec2f(+1, -1));
        m->addVertex(vec2f(-1, +1));
        m->addVertex(vec2f(+1, +1));
```

以上代码创建了一个四边形构成的格网，一个顶点仅一个 2 个浮点型组成的位置属性（属性标识符为 0 ）。然后给格网添加了四个顶点，如一个四边形：

```
unsigned char data[16] = {
            0, 255, 0, 255,
            255, 0, 255, 0,
            0, 255, 0, 255,
            255, 0, 255, 0
        };
        ptr<Texture2D> tex = new Texture2D(4, 4, R8, RED, UNSIGNED_BYTE,
            Texture::Parameters().mag(NEAREST), Buffer::Parameters(), CPUBuffer
(data))
```

以上代码创建了一个 4*4 像素的检查框纹理，每个像素有一个 8 位通道，从 CPU 寄存器缓冲初始化：

```
p = new Program(new Module(330, NULL, "\
            #version 330\n\
            uniform sampler2D sampler;\n\
            uniform vec2 scale;\n\
            layout(location = 0) out vec4 data;\n\
            void main() {\n\
                data = texture(sampler, gl_FragCoord.xy * scale).rrrr;\n\
            }\n"));
        p->getUniformSampler("sampler")->set(tex);
    }
```

以上代码创建了一个单个模块组成的程序，该模块是由单片段着色器组成。它设置自己的“采样器”一致变量的值到前一个纹理：

```
virtual void redisplay(double t, double dt)
    {
        ptr<FrameBuffer> fb = FrameBuffer::getDefault();
        fb->clear(true, false, false);
        fb->draw(p, *m);
        GlutWindow::redisplay(t, dt);
    }
```

以上代码我们重写了 ork::GlutWindow::redisplay 方法，在每一个帧中调用，来重绘窗口内容。我们清除了默认帧缓冲，使用了前一个程序来绘制上文的格网。为了更新窗口内容，重写的方法必须被调用（用 glSwapBuffer）：

```
virtual void reshape(int x, int y)
    {
        FrameBuffer::getDefault()->setViewport(vec4<GLint>(0, 0, x, y));
        p->getUniform2f("scale")->set(vec2f(1.0f / x, 1.0f / y));
        GlutWindow::reshape(x, y);
        idle(false);
    }
};
```

以上代码重写了 ork::GlutWindow::reshape 方法，当窗口被创建或改变大小时被调用，此举是为了使帧缓冲的视场适应窗口大小。

```
int main(int argc, char** argv)
{
    atexit(Object::exit);
    ptr<SimpleExample> app = new SimpleExample();
    app->start();
return 0;
}
```

最后我们创建了一个 ork::Window 子类的实例，并且调用了它的 ork::Window::start 方法来启动用户界面事件处理循环（这个方法没有返回值）。在应用停止之前，我们注册了 ork::Object::exit 方法来适时删除未占用资源。

4 资源框架

渲染类不提供任何载入 3D 格网，从 PNG 文件（或其他任何格式的图像文件）载入纹理内容，或从文本文件载入着色器源代码的工具。这也是为什么上文例子中，格网，纹理和着色器源代码被手动包含在 C++ 代码中。当然这种方式不是很方便。资源模块提供了一个框架来从硬盘载入内容和其他数据。此外该框架还提供运行时更新资源的能力。这对着色器特别有用，因为你可以在硬盘上修改着色器并看到修改的效果，而不需要重启应用。

Ork 资源由 ork::ResourceManager 管理，使用 ork::ResourceLoader 载入实际资源内容。ork::ResourceLoader 类是抽象类，但是提供了 ork::XMLResourceLoader 实体子类。顾名思义，它使用 XML 文件载入资源。一个 XML 资源文件描述了资源的"元数据"（如纹理的缩小率和扩大率筛选器）并指定资源数据（如纹理图像本身）的位置。这些类的典型用法如下：

```
ptr<XMLResourceLoader> resLoader = new XMLResourceLoader();
resLoader->addPath("my-resources/textures");
resLoader->addPath("my-resources/shaders");
resLoader->addPath("my-resources/meshes");
```

我们首先通过创建 ork::XMLResourceLoader 开始，然后通过增加资源文件的位置来配置。在该例子中 my-resources 路径中的子路径通过类型排序，所以我们添加了 3 个路径（添加文件夹并不遍历其子文件夹）。

```
ptr<ResourceManager> resManager = new ResourceManager(resLoader, 8);
```

然后我们使用前一个资源载入器创建 ork::ResourceManager 。第二个构造函数参数 8 是未占用资源的缓存大小。实际上，资源管理器可以临时缓存以代替立即删除未占用资源，以至于如果短期内再次需要它时，不需要从硬盘再次载入。默认缓存大小为 0 ，意味着资源未使用时会立即删除。

一旦资源载入器和管理器被创建和配置，就可以像下面所示轻易的载入资源：

```
ptr<Texture> t = resManager->loadResource("envMap").cast<Texture>();
```

注意：

- 一个资源仅被载入一次。如果试图载入已经存在的资源，将直接返回这个资源实例。因此你不能同时拥有一个资源的多个副本。

这段代码是在一个叫做 envMap.xml 的文件中查找资源载入器路径，这个文件如下：

```
<?xml version="1.0" ?>
<textureCube name="envMap" source="myEnvMap.png"
    internalformat="RGB8" min="LINEAR_MIPMAP_LINEAR" mag="LINEAR"/>
```

注意该文件仅包含纹理元数据。这个纹理图像包含在资源载入器配置路径查找的 myEnvMap.png 文件中。

注意：

- 在开发阶段 ork::XMLResourceLoader 非常方便，因为可以很容易的用一个文本编辑器改变单个资源，甚至在运行时。但是不利方面是，当你分发应用时：你可能不想让用户在"明文"（特别是着色器）中看到你的资源，尤其不想让他们能够在运行时修改这些资

源。 ork::CompiledResourceLoader 就解决这个问题：它禁止了运行时资源更新，从 ork::ResourceCompiler 生成的单个数据文件（需要时你可以加密）载入所有资源。为了使用这些类，你需要首先用 ork::ResourceCompiler （ ork::XMLResourceLoader 的子类）运行你的应用。这会聚集所有的资源到两个文件中，然后在 ork::CompilerResourceLoader 中使用。

4.0.1 档案资源文件

此外，也可以使用包含多个资源的 *档案文件* 来代替一个资源一个文件的许多小 XML 文件。实际上二者能够同时使用，如，一些资源可以由独立文件载入，而另一些从一个或多个档案文件一次载入。为了从档案文件 myArchive.xml 载入资源，必须如下配置：

```
resLoader->addArchive("my-resources/myArchive.xml");
```

在档案文件中，独立资源在档案元素中一个接一个存放（使用 name 属性寻找）：

```
<?xml version="1.0" ?>
<archive>
    ...
    <textureCube name="envMap" source="myEnvMap.png"
        internalformat="RGB8" min="LINEAR_MIPMAP_LINEAR" mag="LINEAR"/>
    ...
</archive>
```

4.0.2 更新资源

如上所述， Ork 资源管理器可以在运行时更新已经载入的资源。例如你修改硬盘上的纹理图片，纹理缩小筛选器，着色器源代码，格网等你会在运行的应用中看到这些变化： 3D 模型上新的纹理图片，新纹理滤波的效果，新着色器的效果，给 3D 模型使用的新的格网的效果，等。这些都在 ork::ResourceManager::updateResource 方法中实现。

这个方法检查硬盘上资源文件的最新修改时间来检测资源载入后是否有变化。如果有则更新资源，更新成功则返回 true 。实际上，更新也会失败，例如 GLSL 着色器中有语法错误。这时着色器不更新，如，旧版本的着色器仍然使用。 updateResource 方法可以用多种方法调用：当用户按下某个键时，固定的时间间隔后，或窗口得到焦点时，等等。

在开发期间，通常会使用预处理器指令选择一个选项来检测模块中的一些选项：

```
#define OPTION2
void main() {
#ifdef OPTION1
    ...
#endif
#ifdef OPTION2
    ...
#endif
...
#ifdef OPTION3
    ...
#endif
}
```

用 Ork 你可以在运行时改变第一行来选择另一个选项。这个变化可以同时在文本编辑器和应用中手动编辑。

4.1 格网资源

一个格网资源以如下方式载入：

```
ptr<MeshBuffers> m = resManager->loadResource("cube.mesh").cast<MeshBuffers>();
```

cube.mesh （ .mesh 非常重要）必须有如下格式（注释不属于文件的一部分）：

```
-1 1 -1 1 -1 1          // bounding box xmin xmax ymin ymax zmin zmax
triangles               // mesh topology (points, lines, linestrip, etc)
4                       // number of attributes per vertex
0 3 float false // attribute 1: identifier, components, format, auto normalize
1 3 float false // attribute 2: identifier, components, format, auto normalize
2 2 float false // attribute 3: identifier, components, format, auto normalize
3 4 ubyte true // attribute 4: identifier, components, format, auto normalize
36                      // number of vertices
-1 -1 +1 0 0 +1 0 0 255 0 0 0  // vertex 0: position normal uv color
+1 -1 +1 0 0 +1 1 0 255 0 0 0  // vertex 1: position normal uv color
+1 +1 +1 0 0 +1 1 1 255 0 0 0  // ...
+1 +1 +1 0 0 +1 1 1 255 0 0 0
// ...
-1 -1 -1 0 -1 0 0 0 255 255 0 0 // vertex 35: position normal uv
color
0                       // number of indices (for indexed meshes)
// indices (empty this case)
```

4.2 模块资源

一个模块资源以如下方式载入：

```
ptr<Module> m = resManager->loadResource("myModule").cast<Module>();
```

如果顶点，镶嵌，几何和片段着色器（所有着色器都是可选的）都在不同的文件中，myModule.xml 文件必须有如下格式：

```
<?xml version="1.0" ?>
<module name="myModule" version="330"
        vertex="myModuleVS.glsl"
        tessControl="myModuleTCS.glsl"
        tessEvaluation="myModuleTES.glsl"
        geometry="myModuleGS.glsl"
        fragment="myModuleFS.glsl"
        options="OPTION1,DEBUG">
    <uniformSampler name="envMapSampler" texture="envMap"/>
    <uniform1f name="..." x="..."/>
    <uniform2f name="..." x="..." y="..."/>
    <uniform3f name="..." x="..." y="..." z="..."/>
    <uniform4f name="..." x="..." y="..." z="..." w="..."/>
    ...
</module>
```

或者如果都在一个单一文件中，被预处理器指令分隔，则为如下格式：

```
<?xml version="1.0" ?>
<module name="myModule" version="330" source="myModule.glsl"
options="OPTION1,DEBUG">
    <uniformSampler name="envMapSampler" texture="envMap"/>
    <uniform1f name="..." x="..."/>
    <uniform2f name="..." x="..." y="..."/>
    <uniform3f name="..." x="..." y="..." z="..."/>
    <uniform4f name="..." x="..." y="..." z="..." w="..."/>
    ...
</module>
```

uniformXxx 是可选的：如果一个一致变量被声明，它将为该模块的程序的一致变量设置初始化值。 options 属性也是可选的。它包含了一个逗号来分隔将注入 source 着色器的预处理器指令。这在从单一 GLSL 文件的大量共用代码中创建多个着色器时非常有用。

注意：

- 如果通过 Ork 资源框架载入， GLSL 源文件可以使用 `#include "....glsl"` 包含在其他资源文件中： ork::XMLResourceLoader 将自动检测这些指令并用引用文件的内容替换它们。

由一个或多个模块组成的程序可以不声明任何资源文件载入：

```
ptr<Program> p1 = resManager->loadResource("myModule;").cast<Program>();
ptr<Program> p2 = resManager->loadResource("module1;module2;").cast<Program>();
```

第一行载入由单个 myModule 模块文件组成的程序（；分号很重要）。第二行载入由两个模块 module1 和 module2 组成的程序（同上）。

4.3 纹理资源

一个纹理资源以如下方式载入：

```
ptr<Texture> t = resManager->loadResource("myTexture").cast<Texture>();
```

myTexture.xml 文件必须有如下格式：

```
<?xml version="1.0" ?>
<texture3D name="myTexture" source="image.png" internalformat="..."
    min="..." mag="..." wraps="..." wrapt="..." wrapr="..." .../>
```

只有 name, source 和 internalformat 属性是强制属性，texture3D 可以被其他纹理类型属性替代，如， texture1D, texture2D, texture2DArray, textureCube 等。这些纹理图片可以是 JPEG, PNG, BMP, TGA, PSD 或 HDR (= 辐射 RGBE) 格式（每个格式都有一些限制条件）。

注意：

- 一个像素一个浮点型的原始格式也受支持。比如一个原始文件必须以五个 32 位整型结尾，第一个是 `0xCAFEBABE`，其他是宽度，高度，位深和像素组份数。

2D 纹理图片直接对应于 2D 纹理的内容。对 1D 纹理来说，纹理图片是一条线。对 3D 纹理来说，2D 纹理图片表现的是 3D 纹理上 z 向切片，是垂直分布的（同样的处理也应用在 2D 数组纹理中的 2D 层）。最后，对一个纹理立方体， 2D 纹理图片表现的是立方体的 6 个面，如下图右图所示分布（ PX 为 x 轴正方向， NX 为 x 轴负方向，以此类推 - 对立方体纹理数组，布局类似，所有的立方体纹理层都被垂直入栈）：

图2：3D 或 2D 纹理数组（左）以及纹理立方体（右）的纹理图片布局

4.3.1 渲染目标

有时你需要使用一些纹理来作为帧缓冲中的渲染目标。这时纹理内容就无关紧要。这些纹理可以用 ork::Texture 构造函数创建，但是它也可以由资源框架载入。好处是可以更容易的共享渲染目标（以为资源仅载入一次，如果代码中同一个渲染目标纹理在多个地方被载入，使用的仍是同一个纹理）。如下：

```
resManager->loadResource("renderbuffer-64-I32F").cast<Texture>();
```

第一部分给出了纹理大小（假设是 2D 四方纹理）。最后一部分是纹理内部格式。如果需要相同格式的多个分离纹理，可以命名为 `renderbuffer-64-I32F-1`, `renderbuffer-64-I32F-2` 等。

4.4 用户自定义资源

通过扩展 ork::Resource 类可以自定义资源。 Ork 中的共用模式为资源定义了一个基本类（所以可以不需要资源框架便实例化和使用），而且这个基本类的子类也可以扩展 ork::Resource，能够通过 Ork 资源框架实例化该资源。这个基本类有如下格式：

```
class MyClass : public Object
{
public:
    MyClass(int p1, int p2)
    {
        init(p1, p2);
    }
    ...
protected:
    MyClass();
void init(int p1, int p2)
    {
        this->p1 = p1;
        this->p2 = p2;
    }
virtual void swap(ptr<MyClass> c)
    {
        std::swap(p1, c->p1);
        std::swap(p2, c->p2);
    }
private:
int p1, p2;
};
```

当类直接实例化时，使用的是公共构造函数。当类通过 Ork 资源框架实例化时，使用的是受保护的构造函数（该构造函数没有参数）。 该构造函数代码位于分离的 init 方法中以至于在基本类和资源子类之间不被重复（见下文）。 swap 方法用于用另一个实例替代当前实例。这是当内容在硬盘中发生变化时， Ork 如何动态更新资源的方法（没有魔法，你必须自己执行 swap 方法）。也有一些方法用来清除渲染目标，从附件中读取，复制像素等：

资源子类使用 ork::ResourceTemplate 类定义：

```
class MyClassResource : public ResourceTemplate<0, MyClass>
{
public:
    MyClassResource(ptr<ResourceManager> manager, const string &name,
            ptr<ResourceDescriptor> desc, const TiXmlElement *e = NULL) :
        ResourceTemplate<0, MyClass>(manager, name, desc)
    {
        e = e == NULL ? desc->descriptor : e;
int p1;
int p2;
        checkParameters(desc, e, "name,p1,p2,");
        getIntParameter(desc, e, "p1", &p1);
        getIntParameter(desc, e, "p2", &p2);
        init(p1, p2);
    }
};
extern const char myClass[] = "myClass";
static ResourceFactory::Type<myClass, MyClassResource> MyClassType;
```

这使 myClassResource 扩展了 MyCass 和 ork::Resource （见 ork::ResourceTemplate 定义）。构造函数必须有预定义签名，它的作用是解码 XML 描述器来提取传递到超类的 init 方法的参数。最后两行在 myClass 名字下注册了资源。意思是当 Ork 资源载入器遇到 `<myClass name="..." p1="..." p2="...">` 元素便创建一个 MyClassResource 对象，通过解码 XML 属性来初始化自己。

4.5 一个例子

我们可以使用 Ork 资源框架重写上文的例子来载入资源：

```
#include "ork/resource/XMLResourceLoader.h"
#include "ork/resource/ResourceManager.h"
#include "ork/render/FrameBuffer.h"
#include "ork/ui/GlutWindow.h"
using namespace ork;
class SimpleExample : public GlutWindow
{
public:
ptr<ResourceManager> resManager;
    ptr<MeshBuffers> m;
    ptr<Program> p;
    SimpleExample() : GlutWindow(Window::Parameters())
    {
        ptr<XMLResourceLoader> resLoader = new XMLResourceLoader();
        resLoader->addPath("resources/textures");
        resLoader->addPath("resources/shaders");
        resLoader->addPath("resources/meshes");
        resManager = new ResourceManager(resLoader);
        m = resManager->loadResource("quad.mesh").cast<MeshBuffers>();
        p = resManager->loadResource("basic;").cast<Program>();
    }
// rest of the code unchanged
};
```

该格网，纹理和模块资源现在可以定义在分离的文件中：

```
-1 1 -1 1 0 0
trianglestrip
1
0 3 float false
4
-1 -1 0
+1 -1 0
-1 +1 0
+1 +1 0
0
```

图3： quad.mesh 文件

```
<?xml version="1.0" ?>
<texture2D name="checkerboard" source="checkerboard4x4.png" internalformat="R8"
    min="NEAREST" mag="NEAREST" wraps="CLAMP" wrapt="CLAMP"/>
```

图4： checkerboard.xml 文件

```
<?xml version="1.0" ?>
<module name="basic" version="330" source="basicModule.glsl">
    <uniformSampler name="sampler" texture="checkerboard"/>
</module>
```

图5： basic.xml 文件

```
#ifdef _VERTEX_
layout(location=0) in vec4 vertex;
out vec2 uv;
void main() {
    gl_Position = vertex;
    uv = vertex.xy * 0.5 + vec2(0.5);
}
#endif
#ifdef _FRAGMENT_
uniform sampler2D sampler;
in vec2 uv;
layout(location=0) out vec4 color;
void main() {
    color = texture(sampler, uv);
}
#endif
```

图6： basicModule.glsl 文件

5 场景图

大多数有趣的 3D 场景都包含了不止一个四边形。这时你必须管理每个对象的一个或多个格网（例如你想根据不同细节水平使用不同的格网），纹理，模块，等等。也不得不管理参考系将这些对象放入全局场景中去，并忽略没有出现在视野中的对象，等。

那么你有一些选项来绘制场景：

- 使用一个在一次循环中计算出每个光源的贡献的模块，一次传递渲染每个对象
- 多次传递渲染每个对象，例如每次只传递一个光源的贡献
- 可以在一个包含位置，法线和材料标识符的 G-buffer 中渲染所有对象，通过渲染全场景四边形来表现所有对象的明暗（称为延时着色）
- 所有的情况下，你可能都需要一个第一次传递（或多个）来绘制场景到阴影贴图中。这里你可以选择每个光源使用一个阴影贴图（如果使用上文第一种方法，这样做是很有必要的），或者为所有光源重复使用同一个阴影贴图（这样你必须使用第二种方法：为第一个光源绘制阴影贴图，为该光源绘制对象，然后用第二个光源的阴影贴图替换该阴影贴图内容，在为第二个光源绘制对象，以此类推）。
- 所有情况下，你也需要后处理传递来表现色调映射或原野景深效果。或许也想在 3D 场景的顶端叠加一些图片或文字。

为了保持所有这些选项都是开放的， Ork 提供了一个小型但是完全可扩展的"玩具"场景图（比较低效，例如该场景图不能用于成百上千个节点组成的场景）。实际上一个 Ork 场景图是一个通用场景节点 ork::SceneNode *树*，每个节点可以看作包含一个状态（字段）和一个行为（方法）的对象。这些状态由一个参考系（与父节点有关），一些格网，模块和一致变量（可以引

用纹理），以及任何用户定义的值所组成。这些行为由完全由用户定义的结合基本任务（绘制格网，设置投影矩阵等）和控制结构（顺序，循环等）的方法组成。场景节点，方法和任务在下文介绍。

5.1 节点

一个场景节点是一个 ork::SceneNode 类的实例。与场景节点相联系的**状态**由以下组成：

- 一个参考系，是相对于其父节点而言的。该参考系可以从 ork::SceneNode::getLocalToParent 获取：这是一个转换矩阵来转换局部参考系的局部坐标到其父参考系的坐标。你也可以使用 ork::SceneNode::getLocalToWorld （或相反则用 ork::SceneNode::getWorldToLocal ）转换局部坐标到世界坐标（如，场景图的根节点参考系的坐标）。最终可以用 ork::SceneNode::getLocalToCamera 和 ork::SceneNode::getLocalToScreen 转换局部参考系的点到相机系或屏幕系（相机系是在场景管理器中定义为“相机”的节点的参考系 - 见下文）
- 一个边界盒，定义于节点的局部参考系中。该边界盒可以用于表现视锥剪裁（见 **方法**）
- 一个标志位集合，可以随意使用（预定义语义没有预定义标识符）。例如可以将一些节点标识为“对象”，一些为“光源”，一些为“透明”，一些为“覆盖”等。标识符可以在循环中引用，以至于你可以迭代将对象标识为“光源”（见**方法**中示例）
- 一个值集合。这些值可以存储节点指定值，如指定的纹理，指定的环境或漫射颜色等
- 一个模块集合。例如你可以使用一个模块在最终的帧缓冲中绘制对象，另一个在阴影贴图中绘制对象等
- 一个格网集合。例如可以在每个细节水平上拥有一个格网
- 一个任意 ork::Object 类集合。这里你可以存储任何你在场景节点（的方法）中需要的值

一个节点的**行为**定义为 ork::Method 类的集合。下文中（见**方法**）做出了解释。最终每个场景节点有一个子节点列表，你可以使用 ork::SceneNode::getChild, ork::SceneNode::addChind 和 ork::SceneNode::removeChild 获取和修改。

因为一个 Ork 场景节点是全通用的，并没有为光源，对象，相机等特别设置场景节点类。实际上场景图中的所有节点都是 ork::SceneNode 类的实例，但是由于它们完全由用户指定，因此状态和行为可以完全不同。

5.1.1 场景节点资源

场景节点可以使用 Ork 资源框架载入。一个场景节点必须有如下格式：

```
<node name="cubeNode" flags="object,castshadow">
    <translate x="0" y="0" z="10"/>
    <rotatex angle="5"/>
    <rotatey angle="10"/>
    <rotatez angle="15"/>
    <bounds xmin="-1" xmax="1" ymin="-1" ymax="1" zmin="-1" zmax="1"/>
    <uniform3f id="ambient" name="ambientColor" x="0" y="0.1" z="0.2"/>
    <module id="material" value="plastic"/>
    <mesh id="geometry" value="cube.mesh"/>
    <field id="..." value="..."/>
    <method id="draw" value="objectMethod"/>
    <method id="shadow" value="shadowMethod"/>
</node>
```

node 元素内部的元素都是可选的并且顺序可调整。定义局部节点到父节点的转换（第一个转换最后被执行）可以有任何数量的 translate, rotatex, ratatey, 和 ratatez 元素。 bounds 元素在局部坐标中定义了边界盒。如果有一个格网（如本例中），该元素不是必要的：边界盒会自动设置为格网的边界盒。你也可以用嵌套的 module 和 mesh 元素声明一些一致变量（uniform1f,
uniform2f, uniform3f, uniform4f, uniformMatrix3f, uniformMatrix4f 或 uniformSampler ），模块和格网。可以使用 field 和 method 元素声明任意的字段和方法（ id 是字段或方法的名字，value 是它的值，该值必须是一个资源的名字）。最后，可以在 flags 属性中关联一些标识符到场景节点中，它们是一个逗号分割的标识符列表。

也可以使用 Ork 资源框架载入全部的场景图。实际上你可以在一个场景节点资源中指定该节点的子节点。可以通过引用或通过值来指定：

```
<node name="myScene">
    <node flags="camera">
        <module id="material" value="cameraModule"/>
        <method id="draw" value="cameraMethod"/>
    </node>
    <node name="myCube" value="cubeNode"/>
    <node flags="overlay">
        <method id="draw" value="logMethod"/>
    </node>
</node>
```

这里， camera 和 overlay 节点是通过值引用，直接在 myScene 节点内部（节点可以在节点内部嵌套而无限制），但是 myCube 节点是通过引用上面的 cubeNode 节点指定的。通过引用指定节点的好处是你可以更容易的重复使用它们。嵌套节点不能被分别载入的时候（不得不载入整个根节点而载入它们），而作为独立资源引用它们。

5.1.2 场景管理器

ork::SceneManeger 管理一个场景图：

- 它用 ork::SceneManager::getRoot 给出场景图的根节点。你可以用 ork::SceneManager::setRoot 改变此节点。
- 它用 ork::SceneManager::getCameraNode 定义那些场景图的节点是“相机”。此节点必须有 “draw” 方法，名字由 ork::SceneManager::setCameraMethod 指定。此方法定义了全场景必须如何被渲染（特别是它必须定义所有的阴影，光线，渲染和后处理操作）。
- 它有一个辅助的 ork::ResourceManager 类，可以在运行时加载和更新场景图资源。
- 最后场景管理器提供了 update 和 draw 方法。 update 方法在你使用 ork::SceneNode::setLocalToParent 改变节点的转换矩阵之后，更新由该节点到世界，相机和屏幕画面的转换矩阵。 draw 方法则简单的执行相机节点的 “draw” 方法。因此它的应为完全由用户定义，可以用于向前渲染策略，延时渲染方法，或任何其他策略。

5.2 方法

ork::Method 类定义一个场景节点的行为。它可以是一个基本任务或一个使用顺序，循环或方法调用的联合任务。基本任务在下节阐述。这里我们介绍联合它们的控制结构，使用例子来演示方法是如何在一个场景图中组织的（像场景节点一样，方法可以在 Ork 资源框架下由 XML 文件载入。下面的例子会说明这些 XML 文件必须如何定义）。

如上所述，一个场景是通过在被定义为“相机”节点的场景节点中执行一系列方法所绘制，该节点必须表现所有需要用来渲染场景的操作。该方法一般是场景中最复杂的方法。我们在这里介绍一个有四个操作的例子：一个操作用来更新生动的对象，一个用来绘制阴影贴图，一个用来绘制对象，一个用来绘制覆盖层（例如一个帧率指示器）。该方法被组织为一个 4 个循环的序列：一个操作一个循环，序列保证操作按顺序执行：

```
<?xml version="1.0" ?>
<sequence>
    <foreach var="o" flag="dynamic" parallel="true">
        <callMethod name="$o.update"/>
    </foreach>
    <foreach var="l" flag="light">
        <callMethod name="$l.draw"/>
    </foreach>
    <foreach var="o" flag="object" culling="true">
        <callMethod name="$o.draw"/>
    </foreach>
    <foreach var="o" flag="overlay">
        <callMethod name="$o.draw"/>
    </foreach>
</sequence>
```

sequence 元素包含任意多个任务：这些任务可以是基本任务，顺序任务，循环任务等。它们一个接一个执行。

loop 任务执行指定为一个*无序*的场景节点集合中嵌套元素的任务（这些嵌套任务可以是任意的：基本任务，顺序任务，循环任务等）。这些嵌套任务在每个场景节点中顺序执行。场景节点通过 flag 属性指定：实际上默认情况下，循环任务在有特别标识符的场景图的所有节点上都执行。此默认行为可以在 culling 和 parallel 选项中改变：

- 如果设置了 culling 选项（如上文第三个循环），该循环被应用于具有特别标识符*以及*在视锥（准确说是视锥与边界盒相交部分）中的场景节点。
- 如果设置了 parallel 选项（如上文第一个循环），该循环被应用于与该循环平行的（默认循环按顺序应用，一个接一个）循环指定的场景节点。这仅对 CPU 任务有效，如果 OpenGL 语境不支持多线程。

var 属性是循环变量。用于指代循环当前被应用于哪个场景节点。例如如果循环变量 `var="1"` 那么在嵌套任务 $1 指代循环当期应用的场景节点（嵌套循环必须使用不同的循环变量名字）。

callMethod 任务在其他场景节点中执行另一些方法。它只有一个指定目标场景节点和目标方法的单个 name 参数，以点分隔。目标节点可以是以下的其中一种：

- this: 该例中目标方法在当前方法被执行的场景节点中调用
- $var: 该例中目标节点是循环当前应用的场景节点。当然这样的目标仅能在循环内部使用。
- flag: 在其他例子中目标名字被看作一个标识符名字，目标场景节点被定义为具有该标识符的节点（必须只有一个这样的节点）。

我们现在可以这样理解上文的例子：在所有具有 dynamic 标识符的节点中并行调用 update 方法，其次在具有 light 标识符的节点中调用 draw 方法，然后在所有具有 object 标识符并且其可视的节点中调用 draw 方法，最后在所有具有 overlay 标识符的节点中调用 draw 方法。

注意该例使用方法调用来表现绘制阴影贴图，绘制对象或绘制覆盖层的实际工作。它可能已经包括在循环中直接执行的实际任务，但这并不是那么通用。实际上，通过上文的组织，你可以使用如面向对象语言中的多态性，在不同场景节点中以不同方式来实施同样的方法。例如一个光源的 draw 方法既可以绘制阴影贴图也可以什么都不做，这取决于是否该光源会投下阴影。

5.3 任务

本节展示了用于实施场景节点方法的基本任务。它们都通过 XML 表现，但你也可以编程实现。有选择帧缓冲及其附件的任务，设置其相关流水线状态的任务，选择一个程序的任务，设置从局部到世界，相机或屏幕空间的转换矩阵的任务，绘制格网的任务，以及显示信息的任务。本节也会演示如何在需要的时候定义自己的任务。

5.3.1 设置目标任务

ork::SetTargetTask 任务选择一个帧缓冲和它的附件。它可以用来选择默认帧缓冲，或者一个有一些特定附件的屏外帧缓冲。选择一个帧缓冲，使用：

```
<setTarget/>
```

选择一个有一些特定附件的屏外帧缓冲，使用：

```
<setTarget>
    <buffer name="COLOR0" texture="..."/>
    <buffer name="COLOR2" texture="..." level="1" layer="3"/>
    ...
</setTarget>
```

name 属性指附件点（可以是颜色 COLOR 或位深 DEPTH ）。 texture 属性指明一个纹理，它有如下形式：

- *name*: 该例中渲染目标是名字为 *name* 的纹理资源
- `this` *.name*, `$v` *.name*, *flag.name*: 该例中渲染目标是目标场景节点 `this` ， `$v` , *flag*（见 *方法*）的形成一致变量采样器边界的纹理的*名字*。
- `this` *.module:name*, `$v` *.module:name*, *flag.module:name*: 该例中渲染目标是目标场景节点 `this` ， `$v` , *flag*（见 *方法*）的模块 *module* 的形成一致变量采样器边界的纹理的*名字*。

`level` 和 `layer` 属性是可选项。它们指你想要为一个 2D 数组纹理（ 3D 数组纹理，2D 立方体纹理）附上的纹理的明细层次（默认为 0 ）和层（z 切片，面）。

5.3.2 设置状态任务

ork::SetStateTask 任务设置当前选择的帧缓冲的流水线状态（读和画缓冲，模版，和深度测试，清除，混合，剔除和写状态，等等）。它有如下格式：

```
<setState readBuffer="COLOR0" drawBuffer="COLOR1"
        clearColor="true" clearStencil="false" clearDepth="true">
    <clear r="0" g="0" b="0" a="0" stencil="0" depth="1"/>
    <polygon front="FILL" back="CULL"/>
    ... TODO ...
</setState>
```

5.3.3 设置程序任务

ork::SetProgramTask 任务选择一个或多个模块组成的程序。有如下形式：

```
<setProgram setUniforms="true">
    <module name="atmosphereModule"/>
    <module name="camera.material"/>
    <module name="light.material"/>
    <module name="this.material"/>
    ...
</setProgram>
```

每个模块由一个 module 元素指定。该元素的名字有以下形式：

- *name*: 该例中模块是名字为 *name* 的模块资源
- `this` *.name*, `$v` *.name*, *flag.name*: 该例中模块是目标场景节点 `this`，`$v`，*flag*（见 *方法*）的形成一致变量采样器边界的模块的*名字*。

以上例子中程序由四个模块组成，为大气效果（由其资源名字设计）定义函数的模块，为投影 3D 点到屏幕空间定义函数的模块（被附在 “material” 名字下 “camera” 场景节点），为从一个光源照亮表面定义函数的模块（被附在 “material” 名字下 “light” 场景节点），和结合表面材料上所有函数的模块（被附在调用方法执行此任务的场景节点上）。

setUniform 属性是可选项。如果给出，一致变量定义在设置程序执行该任务的场景节点上。该选项可以用于在屏幕上绘制对象之前，在程序中为其设置指定值（如颜色，纹理等）。

5.3.4 设置转换任务

ork::SetTransformsTask 任务可以在程序中设置从局部到世界，相机或屏幕空间的转换矩阵。有如下形式（所有属性都是可选项）：

```
<setTransforms localToWorld="..." localToScreen="..."
    screen="..." screenToCamera="..." cameraToWorld="..."
    module="..." worldToScreen="..." worldPos="..." worldDir="..."/>
```

- localToWorld 属性是一致变量 mat4 的名字。该一致变量用于设置从局部画面到世界画面的转换矩阵。局部画面是调用方法执行该任务的场景节点的参考系。一致变量在当前选择的程序中设置。
- localToScreen 属性是一致变量 mat4 的名字。该一致变量用于设置从局部画面到屏幕画面的转换矩阵。局部画面是调用方法执行该任务的场景节点的参考系。屏幕画面是屏幕的参考系，除非 screen 属性被设置为：该例中是 screen 场景节点的参考系。一致变量在当前选择的程序中设置。
- screenToCamera 属性是一致变量 mat4 的名字。该一致变量用于设置从屏幕画面到相机画面的转换矩阵。一致变量在当前选择的程序中设置。
- cameraToWorld 属性是一致变量 mat4 的名字。该一致变量用于设置从相机画面到世界画面的转换矩阵。一致变量在当前选择的程序中设置。

- worldToScreen 属性是 module 模块（如果 worldToScreen 被设置， module 属性就必须被设置）的一致变量 mat4 的名字。该一致变量用于设置从世界画面到屏幕画面的转换矩阵。屏幕画面是屏幕的参考系，除非 screen 属性被设置为：该例中是 screen 场景节点的参考系。
- worldPos 属性是 module 模块（如果 worldToScreen 被设置， module 属性就必须被设置）的一致变量 vec3 的名字。该一致变量用于设置局部参考系的原点的世界坐标。局部画面是调用方法执行该任务的场景节点的参考系。
- worldDir 属性是 module 模块（如果 worldToScreen 被设置， module 属性就必须被设置）的一致变量 vec3 的名字。该一致变量用于设置局部参考系的 z 轴向量单位的世界坐标。局部画面是调用方法执行该任务的场景节点的参考系。

screen 和 module 都具有 *name*，`this` *.name*，`$v` *.name*, 或*flag.name* 的形式（见方法）。

5.3.5 绘制格网任务

ork::DrawMeshTask 任务使用当前选择的帧缓冲和程序绘制格网。它有如下格式：

```
<drawMesh name="..." count="..."/>
```

格网名字有以下格式：

- *name*: 该例中格网是名字为 *name* `.mesh` 的格网资源
- `this` *.name*，`$v` *.name*, *flag.name*: 该例中格网是目标场景节点 `this`，`$v`，*flag*（见 *方法*）的格网*名字*。

`count` 属性是可选项（默认为 1 ）。该整型指格网必须被绘制的次数（使用几何实例）。

5.3.6 信息显示任务

ork::ShowInfoTask 任务在当前帧缓冲中显示帧率和其他文本信息。它有如下格式（所有属性都是可选项：它们的默认值是下文例子之一）：

```
<showInfo x="4" y="-4" maxLines="8" fontSize="16" fontAspect="0.59375"
fontProgram="text;" font="defaultFont"/>
```

x 和 y 属性指定了信息显示的位置，像素为单位，自屏幕左下角起（或者，如果 y 是负值，则从左上角起）。 maxLines 属性为文本显示的最大行数。 fontSize 属性指字符的垂直高度，像素为单位。 fontAspect 属性指每个字符的宽/高比（假设每个字符都有固定宽度）。最后，fontProgram 属性必须为一个程序资源的名字。该程序负责绘制字符。它必须以三角形作为输入，该三角形的顶点具有屏幕空间中的 xy 坐标，和字体纹理中的 uv 为坐标（在 zw 坐标系中）。

默认信息显示任务仅显示帧率。通过 ork::ShowInfoTask::setInfo 方法可以显示附加文本。

5.3.7 日志显示任务

ork::ShowLogTask 任务和信息显示任务相似，但在屏幕上显示的是日志消息（见 日志消息）。默认此任务是关闭的，什么都不显示，直到有错误或警告日志出现。然后保持打开状态只要用户没有设置 ork::ShowLogTask 标识符为 false 。如果一个新的错误或警告日志再次出现，任务会自动再次打开。如此以往。这对通知错误和警告信息特别有用，尤其是硬盘上资源被更新之后（见 资源更新）。日志显示任务有和信息显示任务一样的格式：

```
<showLog x="4" y="-4" maxLines="8" fontSize="16" fontAspect="0.59375"
fontProgram="text;" font="defaultFont"/>
```

5.3.8 用户自定义任务

你可以通过扩展 ork::AbstractTask 类自定义属于自己的任务和任务资源。使用基础样式定义资源（见用户自定义资源），一个新的任务可以定义如下：

```
class MyTask : public AbstractTask
{
public:
    MyTask(...) { init(...); }
virtual ptr<Task> getTask(ptr<Object> context)
    {
return new Impl(...);
    }
protected:
    MyTask() { }
void init(...) { ... }
void swap(ptr<MyTask> t) { ... }
private:
    ...
class Impl : public Task
    {
public:
        ...
        Impl(...) { ... }
virtual bool run()
        {
// put your task implementation here
        }
    };
};
```

相应的资源类：

```
class MyTaskResource : public ResourceTemplate<0, MyTask>
{
public:
    MyTaskResource(ptr<ResourceManager> manager, const string &name,
            ptr<ResourceDescriptor> desc, const TiXmlElement *e = NULL) :
        ResourceTemplate<0, MyClass>(manager, name, desc)
    {
        ...
        init(...);
    }
};
extern const char myTask[] = "myTask";
static ResourceFactory::Type<myTask, MyTaskResource> MyTaskType;
```

5.4 一个例子

我们现在可以使用 Ork 场景图与 Ork 资源框架一起重写上文中的例子（也见一个例子）：

```
#include "ork/resource/XMLResourceLoader.h"
#include "ork/render/FrameBuffer.h"
#include "ork/ui/GlutWindow.h"
#include "ork/taskgraph/MultithreadScheduler.h"
#include "ork/scenegraph/SceneManager.h"
using namespace ork;
class SimpleExample : public GlutWindow
{
public:
    ptr<SceneManager> manager;
    SimpleExample() : GlutWindow(Window::Parameters())
    {
        ptr<XMLResourceLoader> l = new XMLResourceLoader();
        l->addPath("resources/textures");
        l->addPath("resources/shaders");
        l->addPath("resources/meshes");
        l->addPath("resources/methods");
        l->addPath("resources/scenes");
        ptr<ResourceManager> r = new ResourceManager(l, 8);
        manager = new SceneManager();
        manager->setResourceManager(r);
        manager->setScheduler(new MultithreadScheduler());
        manager->setRoot(r->loadResource("scene").cast<SceneNode>());
        manager->setCameraNode("camera");
        manager->setCameraMethod("draw");
    }
virtual void redisplay(double t, double dt)
    {
        ptr<FrameBuffer> fb = FrameBuffer::getDefault();
        fb->clear(true, false, true);
        manager->update(t, dt);
        manager->draw();
        Window::redisplay(t, dt);
    }
// rest of the code unchanged
};
```

使用以下新资源文件：

```
<?xml version="1.0" ?>
<node name="scene">
    <node flag="camera">
        <method id="draw" value="basicCamera"/>
    </node>

    <node flag="object">
        <mesh id="geometry" value="quad.mesh"/>
        <module id="material" value="basic"/>
    </node>
</node>
```

图7： scene.xml 文件

```
<?xml version="1.0" ?>
<foreach var="o" flag="object">
    <setProgram>
        <module name="$o.material"/>
    </setProgram>
    <drawMesh name="$o.geometry"/>
</foreach>
```

图8： basicCamera.xml 文件

6 任务图

如前节所述，一个场景节点的方法由顺序和循环的基本任务组成。感谢现代 CPU 的多核心可以可以并行执行并行循环任务。这在场景数据必须串行到 GPU 或由 GPU 运行时生成（因为该数据比 GPU 显存要大的多），这样的语义下尤其有用。实际上，这种情况下，一些线程可以用来生成场景数据，主线程并行的在屏幕上显示该数据。使用预测将来画面的下一个视点，该场景数据甚至可以在这些线程中提前生成。

然而这种任务的并行执行方式不能在没有*计划任务*的时候使用：必须有一个任务的定义，以及任务之间被明确的依赖，以至于一个任务不会在生成数据所需的所有任务执行之前执行。这是为什么 Ork 在任务图模块中提供了一个框架。该框架提供如下特性：

- 基本任务可以是 CPU 或 GPU 任务。 CPU 任务可以并行执行，而 GPU 任务只能顺序执行（ OpenGL 模型仅支持顺序执行）
- GPU 任务有表达一个 OpenGL 状态的"语义"。 具有同样语义的 GPU 任务会自动被分组在一起（当它们之间的依赖允许这样时），为了最小化 OpenGL 状态变化的次数（例如当前帧缓冲，程序等的变化）
- 所有任务都有期限，一个它们必须被执行前的时间。因此可以对当前帧必须执行的任务，以及一些帧之前不需要的任务做计划。这可以用于从硬盘向 CPU ，或从 CPU 向 GPU *预抓取*一些数据，为了更好的平衡在画面之间的载入工作。
- 基于任务的算法"复杂度"（由用户提供），任务的执行时间可以被监测来预测同样类型的将来任务的执行时间。这对保证固定帧率很有用：那么在当前帧的期限到来之前，一个被预测太大而不能执行的预抓取任务则不会被执行。
- 任务间的依赖，如，一个任务必须在另一个之前执行（通常因为它需要另一个任务生成的结果），被定义于*任务图*。也可以将任务图的依赖（是任务的特殊形式）加入更大的图。注意一个任务或任务图可以同时被加入多个任务图。
- 一个任务实例仅执行一次 ：如果用户为执行计划一个任务，并且任务已经被执行过了，那它将不会再次被执行。因此如果你想在每个帧中执行这个任务，例如，你必须在每个帧中为该任务创建一个实例，然后每个实例计划一次。这条规则的唯一例外是你明确的*重调度*了这个任务。这时该任务将在任何情况下执行，即使它以及被执行过。还有，依赖于此任务的所有任务都将再次执行，以此类推。这个特性在当你改变一个任务的输入数据时很有用：通过重调度该任务，它的结果会使用新的输入重计算到汇总中。实际上所有直接或间接依赖此输入的数据，通过该任务，都会被重新计算。

6.1 基本任务

基本任务由 ork::Task 类表现。 ork::Task::isGpuTask 方法指明了该任务是否为 GPU 任务。如果是一个 GPU 任务，它的语义由 ork::Task::getContext给定。由 ork::Task::begin 和 ork::Task::end 设置和取消设置。任务本身由 ork::Task::run 方法定义。任务期限由 ork::Task::run 和 ork::Task::setDeadline 管理。

6.2 任务图

任务图由 ork::TaskGraph 类表现。为了创建任务图，你首先必须用 ork::TaskGraph::addTask 方法添加一些任务进去。你可以用 ork::TaskGraph::addDependency 定义任务之间的依赖。

ork::Scheduler 类是一个抽象类，定义了任务如何计划被执行。 ork::Scheduler::run 方法用于计划立即执行一个任务或任务图。它直到所有具有立即执行期限的任务都被执行后才返回（该任务图可以包含不立即执行的任务）。 ork::Scheduler::schedule 方法用于调度不立即执行的任务。它把这些任务放入一个待执行任务池，然后立即返回。如果该计划不支持预抓取（这个由 ork::Scheduler::supportsPrefetch 决定），这个方法则不能被调用。最后，ork::Scheduler::reschedule 方法用于再执行一些任务。它标记这些任务以及依赖这些任务的任务为“未执行”，把它们放入待执行任务池，然后立即返回。

ork::MultithreadScheduler 是一个 ork::Scheduler 的具体执行类。它的构造函数使用帧率和线程数作参数。如果帧率是 0 那么没有帧率被使用。否则计划使用该目标帧率（如果必要它在画面之间等待以避免在目标上增加帧率）。线程数指除主线程以外，调度器能使用多少线程。 0 表示所有任务都将以单线程顺序执行。这样一个调度器可以在资源框架中载入，有如下形式：

```
<?xml version="1.0" ?>
<multithreadScheduler name="myScheduler" nthreads="3" fps="0"/>
```

注意：

- ork::AbstractTask 类不是一个 ork::Task 类，而是 ork::TaskFactory 类，比如，创建任务用的。这意思是所有前节介绍的“任务”实际上都是任务工厂。这是因为，如前所述，一个任务不能在每个帧中重复执行除非每次创建一个新实例。

6.2.1 一个例子

这里有一个任务图的例子，其中有一些嵌套图。基本任务表示为正方形，任务图为矩形，箭头表示任务间依赖：

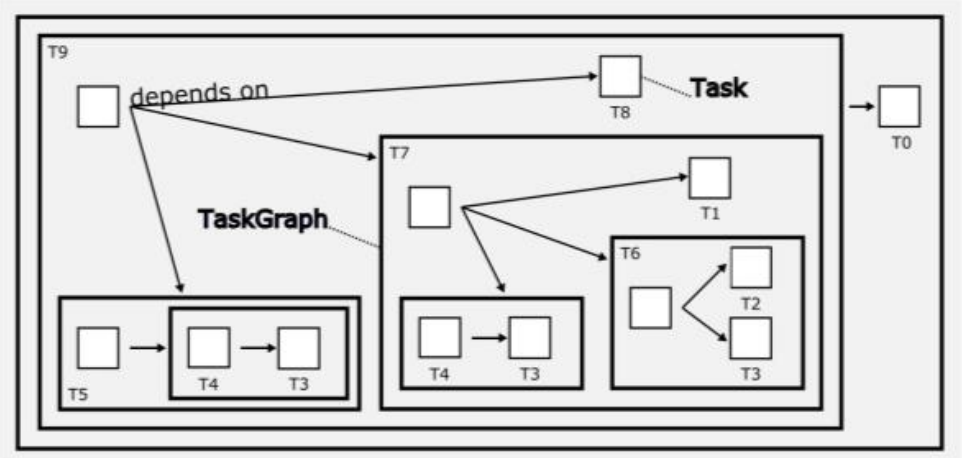

图9：一个任务图例子。注意在任务图之间任务和图都可以共享

该任务图等同于下文的"平铺"图，没有任何嵌套：

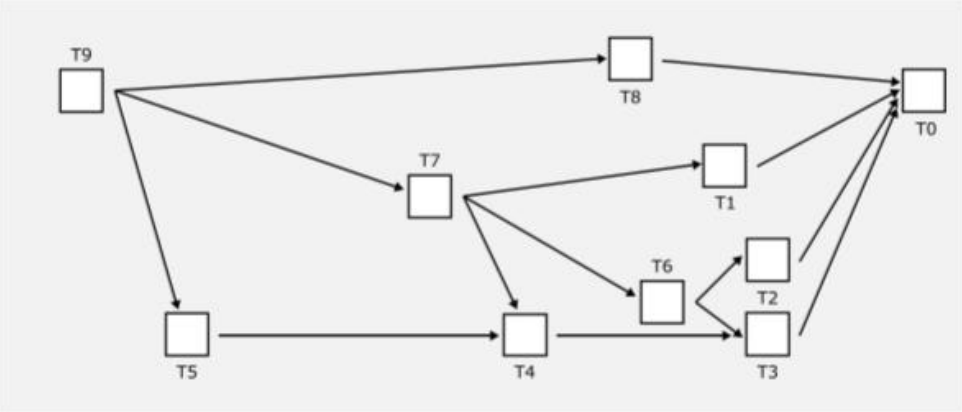

图10：相对于上文嵌套图的平铺图

注意在多个图中出现的共享任务这里只出现一次。也要注意任务图 T9 和任务 T0 之间的依赖以及被没有 T9 依赖的所有子任务和 T0 之间的依赖替代。如果 T0 任务被重调度，那么所有任务将重执行。如果 T6 任务被重调度，那么仅有 T6, T7 和 T9 任务被重执行。

7 用户接口

Ork 也提供了如 glut 提供的创建窗口和处理事件的能力。

7.1 事件句柄

ork::EventHandler 类包含当事件发生时调用的类。他们中的一些返回一个布尔值来指示是否事件被该句柄捕获。事件句柄抽象类提供了你想使用的接口系统的独立性（如，glut，微软 Windows 导航窗口，等）

事件函数有：

- redisalay: 在每个新帧上调用
- reshape: 窗口缩放时调用
- idle: 空事件时调用
- mouseClick: 当用户点击时调用；该方法使用鼠标位置，使用的键，是否被按下或松开，或是否使用的任何组合键（ ALT, CTRL, … ）
- mouseMotion: 鼠标移动时调用
- mouseWheel: 鼠标滚轮使用时调用
- mousePassiveMotion: 鼠标没有随事件移动时调用
- keyTyped: 按下一个键时调用
- keyReleased: 松开一个键时调用
- specialKey: 当特殊键按下时调用（ESC，PAGE_UP，HOME 等）
- specialKeyReleased: 特殊键松开时调用

ork::EventHandler 也提供描述事件的枚举（鼠标键的名字和状态，修改的键，特殊键名字，鼠标滚轮状态）。

7.2 窗口

窗口本身也是事件句柄，所以一旦初始化，他们的表现就像是任何其他事件句柄。他们可以通过该系统移动，缩放，导航，…

ork::Window 是窗口的抽象超类。提供了一个 ork::GlutWindow 的具体子类。基于 GLUT 实施。