
CGDC
中国游戏开发者大会
CHINA GAME DEVELOPERS CONFERENCE
2014

使用OpenGL 4.x为移动游戏开发图形特效

曹家音
内容技术开发工程师
英伟达(NVIDIA)

摘要

- OpenGL 和 Tegra K1 简介
- OpenGL4 的初始化
- OpenGL4 带来的新特性
- OpenGL4 高级优化技巧
- Tegra K1 Demo
- 总结

OpenGL的现状

- OpenGL最新版本为4.4
- 过去几年中，OpenGL产生了很大的变化
  - 发布了5个以上的版本
  - 发布了70个以上的ARB extension
- OpenGL新版本带来了很多新特性
  - 增加了开发者的开发效率
  - 提高了程序运行的效率
  - 提供了一些新的功能

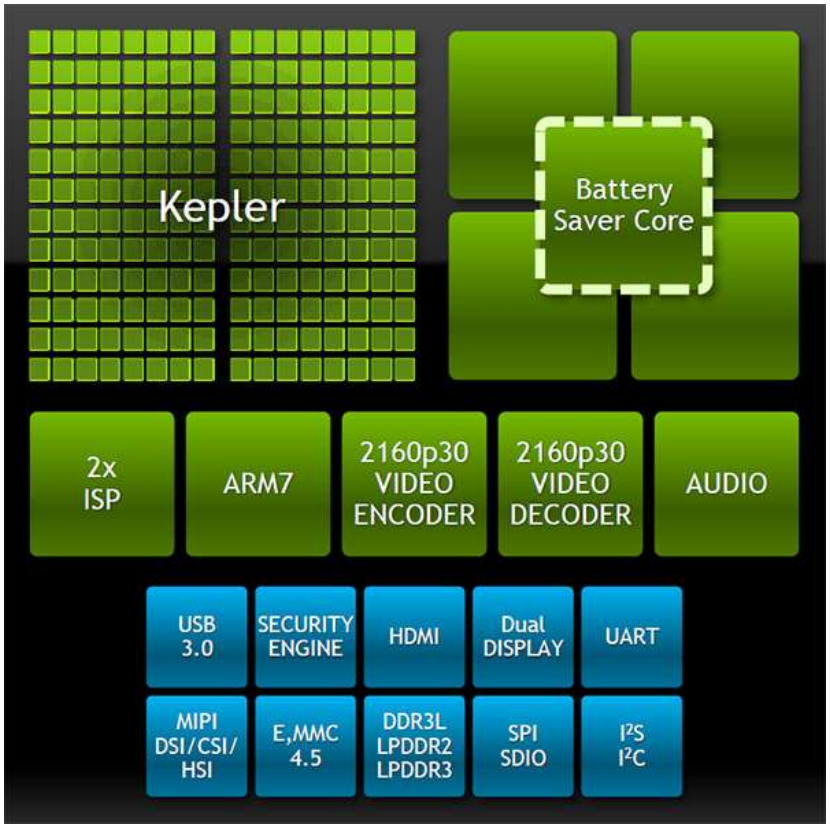
OpenGL与Direct 3D

- OpenGL 2 ≈ DX9
  - Shader
- OpenGL 3 ≈ DX10
  - Geometry Shader
- OpenGL 4 ≈ DX11
  - Tessellation & Compupte Shader

- 图形特性与OS相对独立，可以在Android上运行DX11的新特性！

Tegra K1

- GPU: Kepler架构
  - 192个CUDA core
  - 统一的Shader架构
  - 与Geforece平台一致
- CPU:
  - 4+1核
  - ARM Cortex-A15
  - 2.3GHz



完全支持OpenGL 4.4

- 与GeForce显卡共用一套驱动程序，可以渲染PC平台的游戏画面
- 相对于现在常用的OpenGL ES 2.0有着非常大的优势
  - PC平台的高级图形特效
  - 更好的API性能
  - 移植PC平台更加容易

Tegra K1 Demo

OpenGL 4.x初始化

- 把OpenGL API与EGL进行绑定
  - eglBindAPI(EGL_OPENGL_API)
- 检查是否有OpenGL支持：
  - eglGetConfigAttrib(display, configs[i], EGL_RENDERABLE_TYPE, &renderableFlags);
  - if ((renderableFlags & EGL_OPENGL_BIT) == 0)
    continue; // skip this config; no GL support

OpenGL 4.x初始化

- 在创建EGL context的时候，加入如下属性：
  - EGLint contextAttrs[] = { EGL_CONTEXT_MAJOR_VERSION_KHR,4, EGL_CONTEXT_OPENGL_PROFILE_MASK_KHR,EGL_CONTEXT_OPENGL_COMPATIBILITY_PROFILE_BIT_KHR, EGL_NONE };
  - eglCreateContext(display, config, NULL, contextAttrs);

调用OpenGL4.x的函数

- 应用程序必须首先得到函数指针：
  - 函数指针定义：typedef void (EGLAPIENTRYP PFNGLPATCHPARAMETERFVPROC)(GLenum pname, const GLfloat *values);
  - 得到函数指针：glPatchParameterfv = (PFNGLPATCHPARAMETERFVPROC)eglGetProcAddress("glPatchParameterfv");
- 然后通过函数指针进行调用

调用OpenGL 4.x的函数（续）

- 通过上述方式调用OpenGL 4.x函数则需要大量的函数指针，会增加开发成本
- 解决方案：REGAL
  - 跨平台的开源代码库（https://github.com/p3/regal）
  - 包含了OpenGL的所有函数（不需要额外获取指针）
  - 增加了一些高级的OpenGL特性
  - 保留了一些旧的特性（例如，固定管线）
  - 增加了很多方便Debug的特性，从而加速开发

如何使用REGAL

- 设置一些基本的参数
  - LOCAL_STATIC_LIBRARIES :=regal_static
  - Include $(BUILD_SHARED_LIBRARY)
  - $(call import-module, regal_static)
- 在工程里面引用REGAL提供的头文件
  - #include <GL/Regal.h>

如何使用REGAL（续）

- REGAL初始化
  - RegalMakeCurrent(eglGetCurrentContext())
- 直接调用OpenGL 4.x的函数
  - glPatchParameteri(…)

Direct State Access(DSA)

- OpenGL是一个很大的状态机，有很多函数用来切换OpenGL的状态
  - glActiveTexture, glBindTexture等
- 频繁的状态切换和管理使得程序代码显得冗余，随着应用程序变得越来越复杂，状态的维护就相对比较困难：
  - 当程序员想恢复某一个状态的时候，需要大量的状态切换函数

Direct State Access(DSA)

- EXT_direct_state_access
- 该扩展为OpenGL增加了很多新的函数，可以用来直接更改某些对象的状态
- 很多OpenGL的新的特性也会有DSA版本的函数调用

DSA应用实例

- 例如，更改一个Texture的采样过滤：
- 没有DSA
  - glActiveTexture(GL_TEXTURE0)
  - glBindTexture(GL_TEXTURE_2D, id)
  - glTexParameteri(GL_TEXTURE_2D, GL_TEX_MIN_FILTER, GL_LINEAR)
- 使用DSA
  - glTextureParameteriEXT(id , GL_TEXTURE_2D , GL_TEX_MIN_FILTER , GL_LINEAR)

DSA支持的对象

- DSA支持很多OpenGL的对象
  - Texture Object
  - Vertex Array Object
  - Frame Buffer Object
  - Program Object
  - Buffer Object
  - 等等

OpenGL中的Debug

- OpenGL的错误信息一般是通过glGetError来返回的
  - 程序员往往不能直接知道问题的所在，所以需要在很多地方调用glGetError，增加很多开销
  - 很多错误描述并不清晰，错误不分级别
  - 需要用宏或者if语句把glGetError包起来，从而可以在Release版本把这些错误检查去掉

新的Debug方式

- ARB_debug_output
- 注册一个回调函数，当OpenGL的调用出错误的时候，会自动进入该函数
  - 由驱动程序调用该回调函数，而不是开发者
  - 没有必要用宏或者if语句
  - 可以随时动态开关Debug功能
  - 错误信息分一定的级别
  - 错误信息不是简单的枚举，而是更具有描述性的String

使用新的Debug功能

- // Callback defination
- void APIENTRY DebugFunc(GLenum source, GLenum type, GLuint id, GLenum severity, GLsizei length, const GLchar* message, GLvoid* userParam){ … }

- // Register the callback
- glDebugMessageCallback(DebugFunc, NULL);

- // Enable debug messages and ensure they are not async
- glEnable(GL_DEBUG_OUTPUT);
- glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);

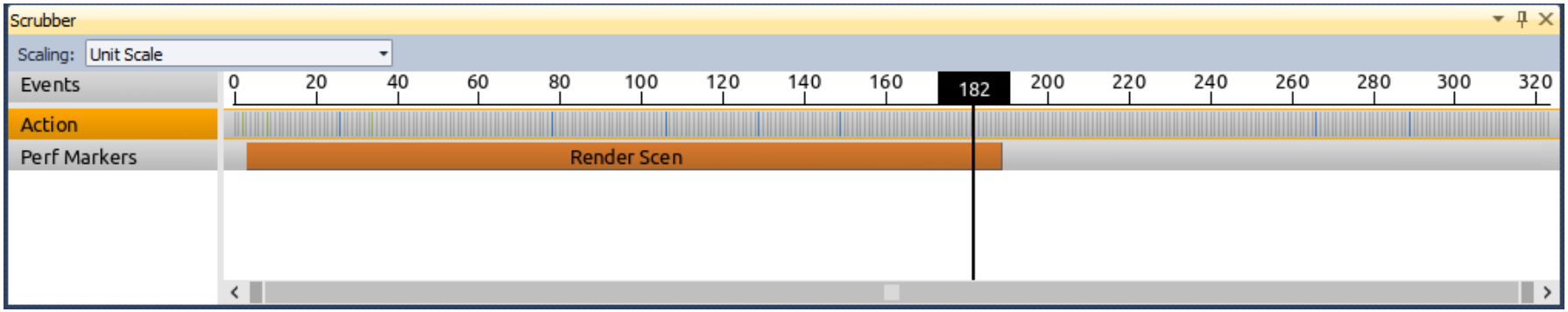
- 在强制线程同步后，可以通过Call Stack直接看到OpenGL出错的地方

插入自己的Debug错误信息

- 程序员可以在代码中加入自己的错误信息，甚至可以加入PerfMarker
  - **// Add a marker to the debug notations**
  - glPushDebugGroup(GL_DEBUG_SOURCE_APPLICATION, SCENE_RENDER_ID, 11, “Render Scene”);
  - **// Perform application rendering**
  - Render_Scene();
  - **// Closes the marker**
  - glPopDebugGroup();

Nsight中的PerfMarker

- 如上代码中，可以在Nsight中看到其相应的PerfMarker



Debug功能的限制

- 回调函数中是有一些限制的：
  - 不能够调用其他OpenGL和Windows相关的函数
  - 可能是在其他线程中异步调用的
- 回调函数同样是有一些开销的
  - 在Release版本中，去掉这些错误检查
- 回调函数返回的信息
  - 不同的供应商提供的错误信息描述是不一致的
  - 不要在应用程序中去解析这些字符串

传统的Shader工作方式

- 传统的OpenGL的Shader工作方式如下：
  - 创建Shader对象
  - 编译Shader对象
  - 创建Program对象
  - 绑定相应阶段的Shader对象
  - 链接所绑定的Shader对象
  - 使用前调用glUseProgram

该方式存在的一些问题

- Vertex Shader和Fragment Shader经常不是一一对应的：
  - Pre-Z中，多个Vertex Shader会对应一个简单的Fragment shader
  - 后处理算法中，Vertex Shader基本是一样的，不过Fragment shader是不一致的

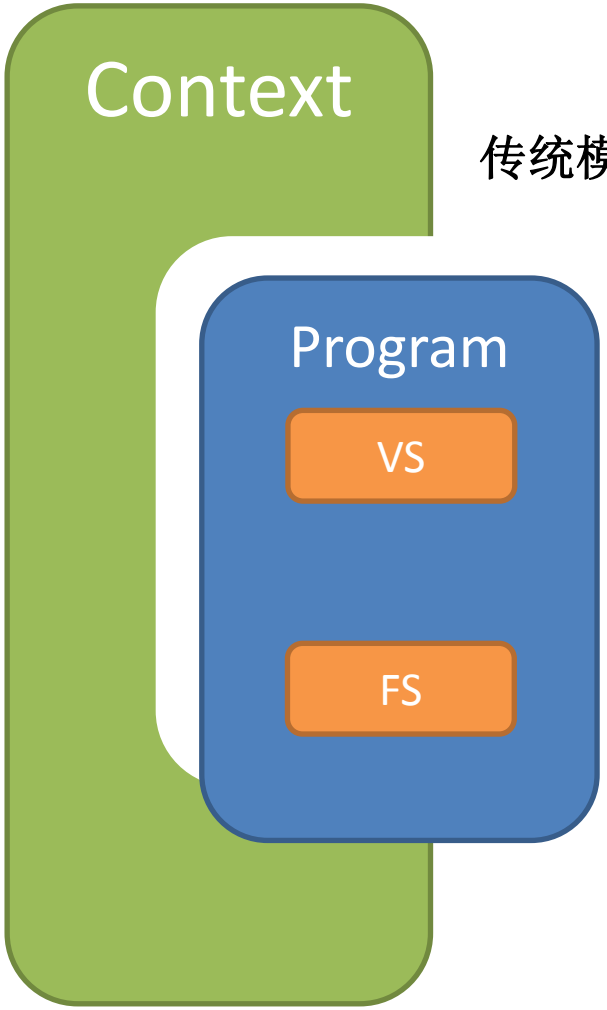
- 最糟糕的情况：
  - N个vertex shader+M个fragment shader = MxN个Program

- 当Stage数量更多的时候（例如Tessellation，Geometry Shader），情况会变得更加糟糕

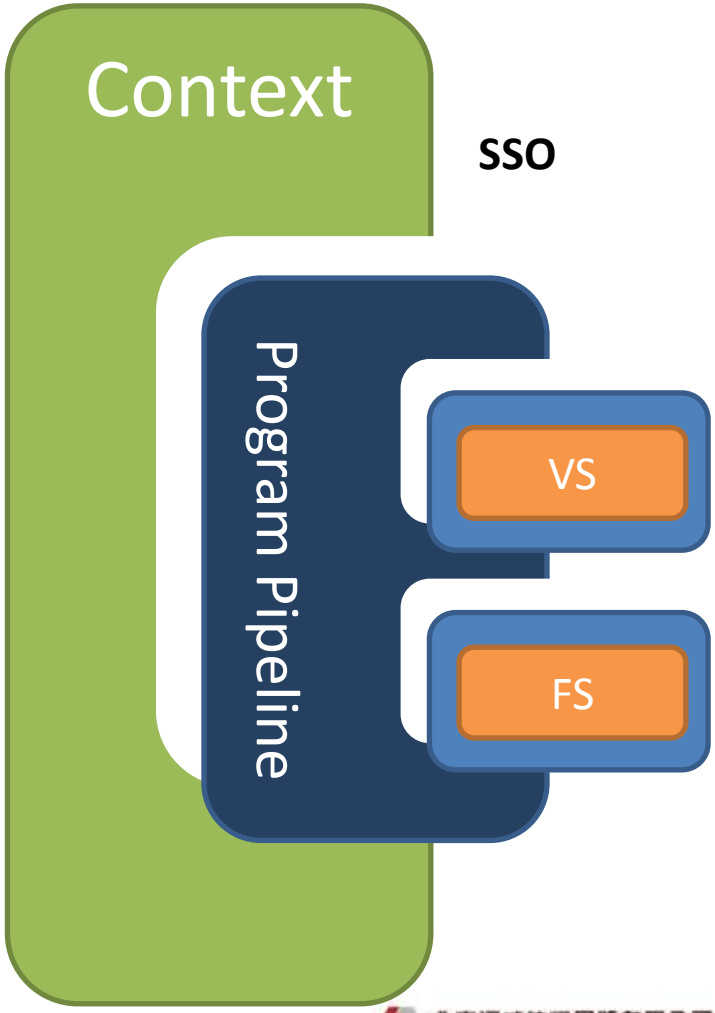
Separate Shader Objects

- ARB_separate_shader_objects
- 一个Program可以只绑定一个类型的Shader
- 增加了一个新的类型：Program Pipeline
- 允许不同类型的Shader进行动态的链接
- 可以动态绑定不同阶段的Shader，而不需要在初始化的时候设置好

Separate Shader Objects



传统模型



SSO

Separate Shader Objects代码

- **// Create shaders**
- GLuint fprog = glCreateShaderProgramv(GL_FRAGMENT_SHADER, 1, &fstext);
- GLuint vprog = glCreateShaderProgramv(GL_VERTEX_SHADER, 1, &vstext);

- **// Bind pipeline**
- glGenProgramPipelines(1, &pipe);
- glBindProgramPipelines(pipe);

- **// Bind shaders**
- glUseProgramStages(pipe, GL_FRAGMENT_SHADER_BIT, fprog);
- glUseProgramStages(pipe, GL_VERTEX_SHADER_BIT, vprog);

一些必要的更改

- 由于Shader的链接是运行时动态进行的，所以需要Shader代码中显式的声明相关的输入输出，即使是GLSL内置的参数

- **// Rdeclare gl_Position**
- out gl_PerVertex { vec4 gl_Position; };

Shader 中进行文件引用

- 在Shader中进行文件的引用，可以在不同的Shader脚本中共享公共的内容
- 在OpenGL中，没有文件系统的概念
- 在Shader引用前，必须把相应的需要引用的内容进行注册：
  - glNamedStringARB(GL_SHADER_INCLUDE_ARB, strlen(filename), filename, strlen(shader_content), shader_content);

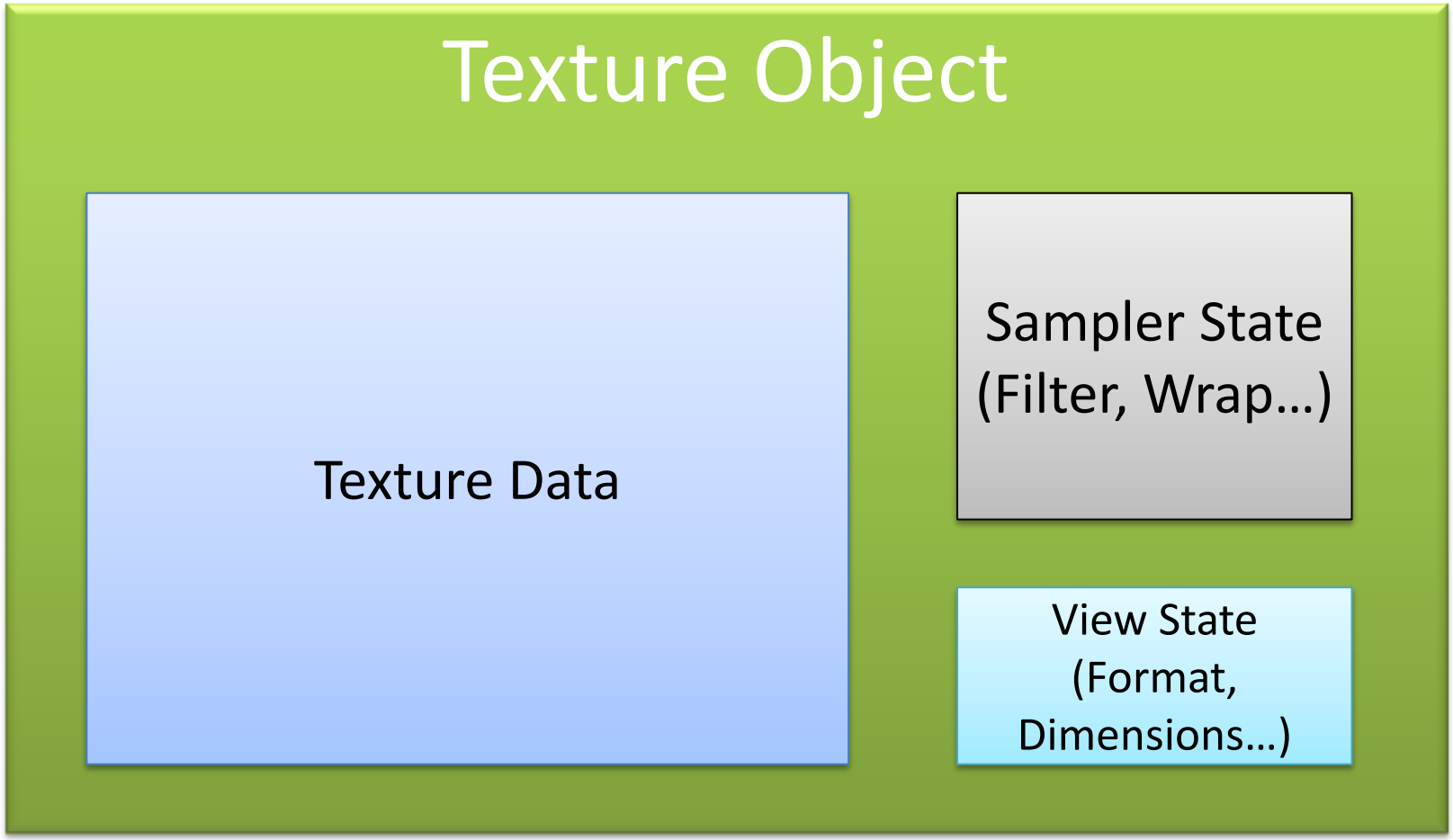
显式的位置绑定

- Shader中的一些内容可以通过特殊的关键字显示的绑定到指定的位置
  - **// specify the bind point for a buffer of uniform data**
  - layout(binding=1) uniform ConstBuffer { … };
  - **//specify the bind point for a Sampler**
  - layout(binding=2) uniform sampler2D texture;
  - **// specify the buffer used to store normals for deferred shading**
  - layout(location=3) out vec4 normalData;

显式位置绑定的优势与限制

- 优势
  - 节省一定的代码工作
  - 固定某一个常用资源的绑定位置，例如ViewMatrix等，从而每帧只需要更新一次，不需要每个Draw call都更新
- 限制
  - 程序员需要保证Shader脚本与应用程序中的位置定义是一致的

Texture的改进



Texture Storage

- 传统的Texture创建的一些问题：
  - 每次只能创建一个mipmap级别
  - 可能会导致一些错误，比如读取没有创建的mipmap级别
  - Draw Call前会有安全性检查
- Texture Storage带来的一些优势：
  - 简化Texture的创建方式
  - 由于创建的Texture是不可写的，所以可以节省一些Draw call的开销
  - （注意，这里不可写的内容仅仅是Texture的一些参数，例如尺寸，格式等，数据仍然是可以更改的）

Texture Storage的用法

- // Classic OpenGL texture creation
- glBindTexture(GL_TEXTURE_2D, id);
- for (i = 0; i<9, i++)

  glTexImage2D(GL_TEXTURE_2D, i,GL_RGBA8, 256>>i,256>>i, 0, GL_RGBA, GL_FLOAT, NULL);

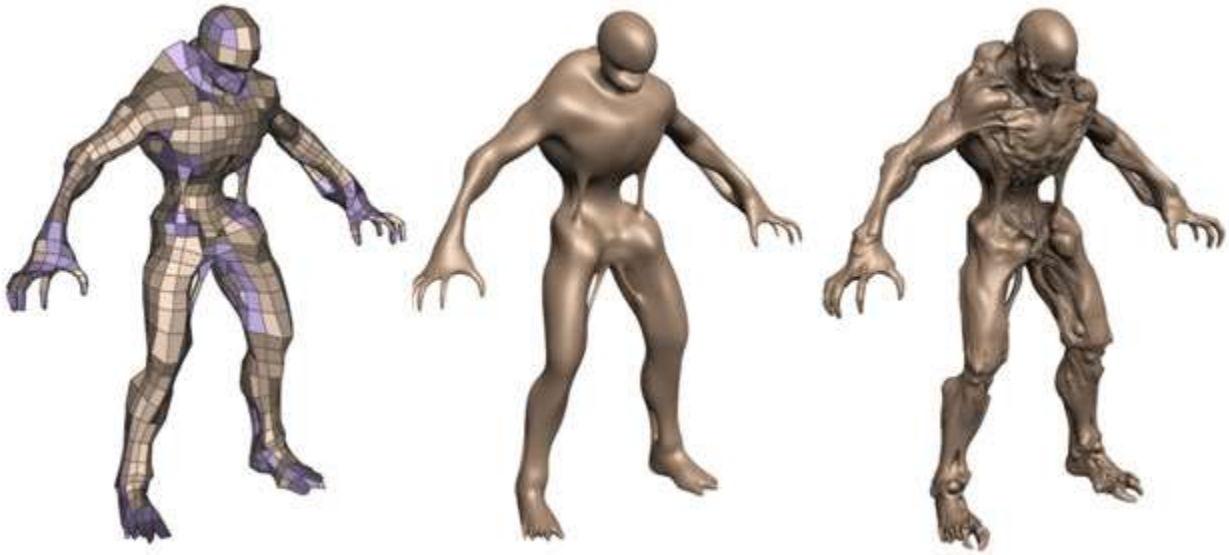
- // DSA-style version with Texture Storage
- glTextureStorage2DEXT(id, GL_TEXTURE2D, 9, GL_RGBA8,256, 256);

Texture中的Sampler Object

- Texture会默认使用内置的Sampler进行采样
- 当Texture绑定的通道中有其他Sampler绑定时，Texture将采用其绑定的Sampler，而不是内置的。
- 其他的API的做法不同

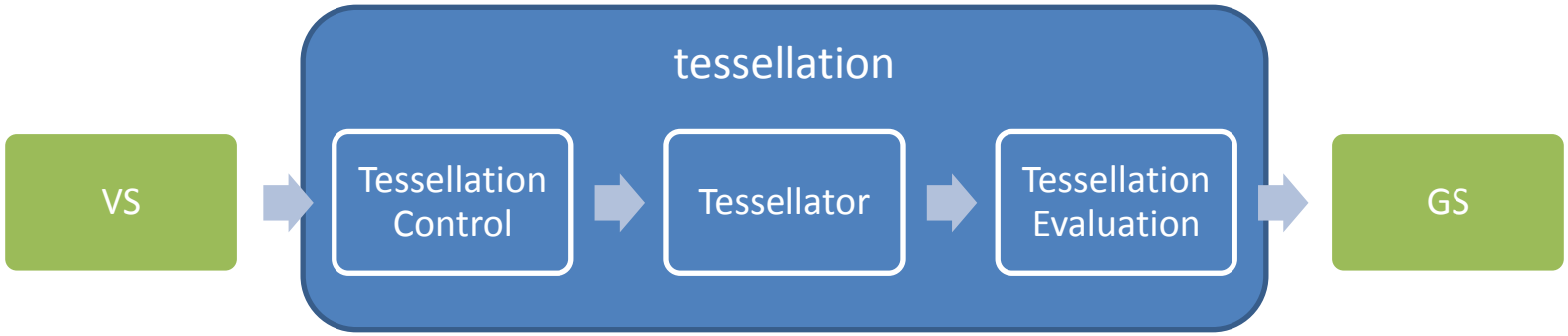
曲面细分（Tessellation）

- Tessellation用来动态的增加低模的多边形，从而丰富物体的几何信息



曲面细分（Tessellation）

- Vertex Shader与Geometry Shader中三个新的阶段
  - Tessellation Control Shader（针对每个Patch中的点）
  - Tessellator（不可编程）
  - Tessellation Evaluation Shader（针对每个输出的图元中的顶点）



与DX11 Tessellation的区别

- Tessellation Control Shader = Hull Shader
- Tessellation Evaluation Shader = Domain Shader

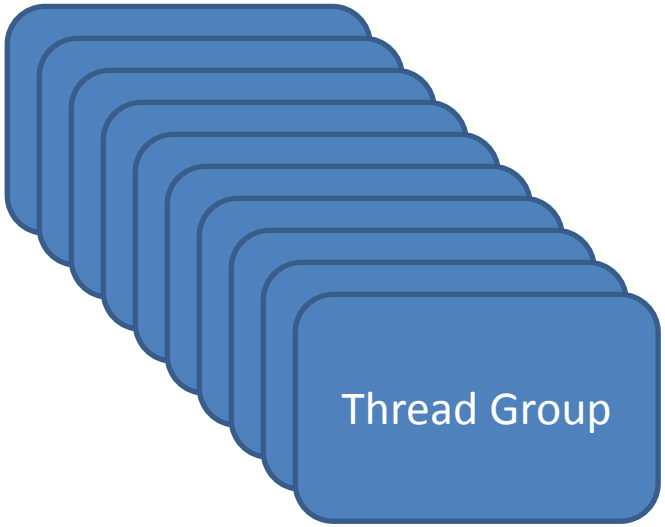
- OpenGL 的Control Shader中，需要直接写如Tess Factor，而Hull Shader需要有额外的一个Constant Function
- OpenGL可以不提供Control Shader，而直接通过应用程序指定Tess Factor
- 分割方式等内容都是在Evaluation Shader中定义的，而DX11是在Hull Shader中定义的

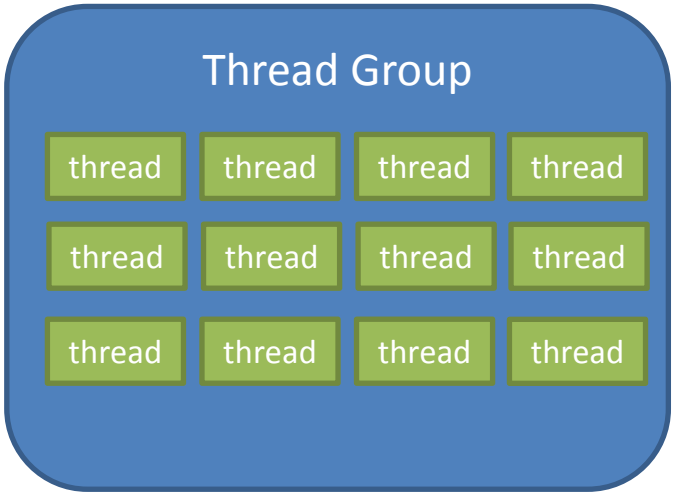
Compute Shader

- 一个完全与渲染管线独立的阶段
- 该阶段可以用来计算任何通用的数据
  - 粒子系统（SPH流体模拟）
  - 后处理（Blur）
  - 物理碰撞
  - 海水
  - 非图形相关的计算
  - 等等

Compute Shader计算模型

- 每个Dispatch call中包含了多个Thread Group，每个Thread Group又由多个Thread组成





驱动程序的开销

- 更多的Draw Call
  - 更多的图形处理工作（GPU）
  - 更多的驱动程序的开销
- 减少驱动程序的开销
  - 减少Draw Call的数量
  - 合并Draw Call的批次
    - Draw Call Breaker（Texture，Uniform）

Uniform更新

- 如果需要合并Draw Call批次的话，需要有大量的Uniform数据，可以考虑存储在如下资源中
  - Shader Storage Buffer Object (SSBO)
  - Uniform Block
  - Texture Buffer

Uniform资源的索引

- 两种解决方案：
  - 使用内置的gl_DrawIDARB（有些硬件不支持）
  - 使用baseInstance参数进行模拟（在不使用实例化功能的前提下可以用，速度相对于前者有一定优势）

glMapBuffer优化

- 频繁的MapBuffer会产生很严重的驱动的开销
- OpenGL 4 提供了Map Persistent功能，可以在程序初始化的时候map buffer，然后在程序结束的之后Unmap
  - 程序员需要维护数据合理性，防止写入正在读取的数据

glMapBuffer优化（续）

- mapFlag = GL_MAP_WRITE_BIT|
  GL_MAP_PERSISTENT_BIT|
  GL_MAP_COHERENT_BIT;
  createFlag = mapFlag | GL_MAP_DYNAMIC_STORAGE_BIT;

- mDestHead = 0;
  mBuffSize = 3 * maxVerts * kVertexSizeBytes;

- glBindBuffer(GL_ARRAY_BUFFER, VertexBuffer);
- glBufferStorage(GL_ARRAY_BUFFER, mBuffSize, null, createFlags);

- mVertexDataPtr = glMapBufferRange(GL_ARRAY_BUFFER, 0, mBuffSize, mapFlags);

Bindless资源

- OpenGL4提供了三种新的Bindless资源
  - Bindless Vertex Data
  - Bindless Uniform
  - Bindless Texture
- Bindless资源可以直接获得其GPU地址，然后提供给Shader脚本
- 以Bindless Texture为例，其他两种资源的使用方式类似

传统的Texture工作方式

- 传统Texture的工作方式
  - 创建Texture
  - 把Texture绑定到指定的通道
  - 在Shader脚本中定义sampler
  - 调用Draw call

传统的Texture工作方式（续）

- 传统的Texture绑定方式：
  - Foreach(draw in draws) {
    foreach(texture in draw->textures)
    glBindTexture(GL_TEXTURE_2D , tex[id]);
    glDrawElements(…);
    }
- 这种绑定的一些限制
  - 同一个阶段只能够绑定有限数量的Texture
  - Draw call之间需要频繁切换绑定的贴图
  - 反复调用glBindTexture会有一些驱动开销

Bindless Texture

- 移除了Texture的绑定过程

```
// Create textures as normal, get handles from textures
GLuint64 handle = glGetTextureHandleARB(tex);

// Make resident
glMakeTextureHandleResidentARB(handle);

// Communicate 'handle' to shader... Somehow
...

// draw calls
foreach(draw) {
   glDrawElements(...);
}
```

Bindless Texture（续）

- Shader 代码
  - uniform Samplers {
    sampler2D tex[500]; // Limited only by storage
  - };
  - out vec4 oColor;
  - void main(void) {
    oColor= texture(tex[123], ...) + texture(tex[456], ...);
  - }

Bindless Texture优势与限制

- 优势
  - 可以为Shader同时提供更多数量的Texture
  - Draw Call之间不需要频繁更新Texture的绑定
  - 可以合并更多DrawCall，节省驱动程序的开销
- 限制
  - 并不是所有硬件都是支持的
  - 可以考虑使用Texture Array

利用DrawIndirect进行优化

- 常规的物体渲染循环如下：
  - foreach(object)

```
DrawElementBaseVertex(GL_TRIANGLES ,
                      object->indexNum,
                      GL_UNSIGNED_SHORT,
                      object->indexOffset,
                      object->baseVertex );
```

利用DrawIndirect进行优化（续）

- 使用DrawIndirect

```
foreach( object )
{
    updateCommand( &command );
    glDrawElementIndirect(… , &command );
}
```

利用DrawIndirect进行优化（续）

- 使用MultiDrawIndirect把Draw Call合并
- foreach(object)
  updateCommand(&commands[i++]);

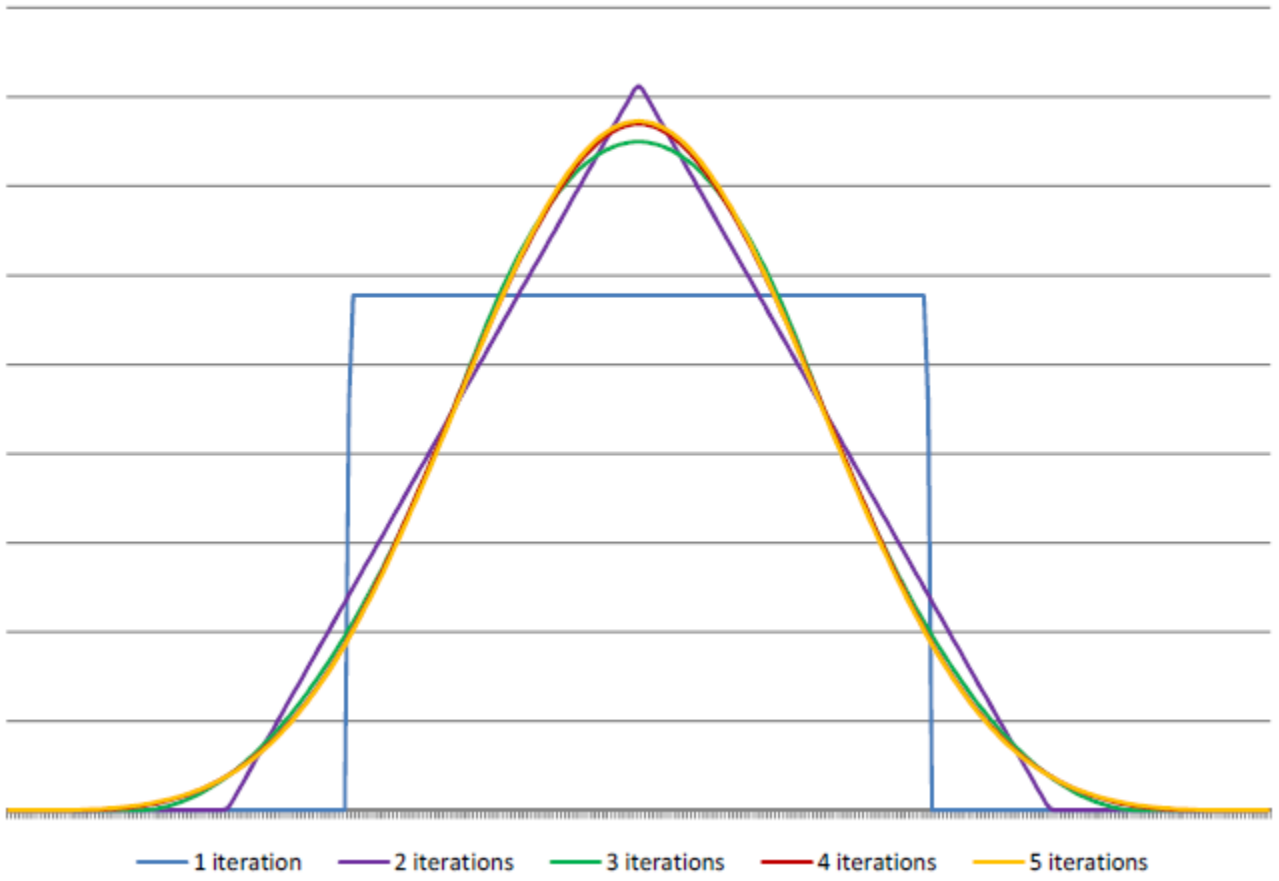
glMultiDrawElementIndirect(… , &command , commandNum);

Costant Time Gaussian Blur

- Blur的半径与开销无关（45fps，1080p）

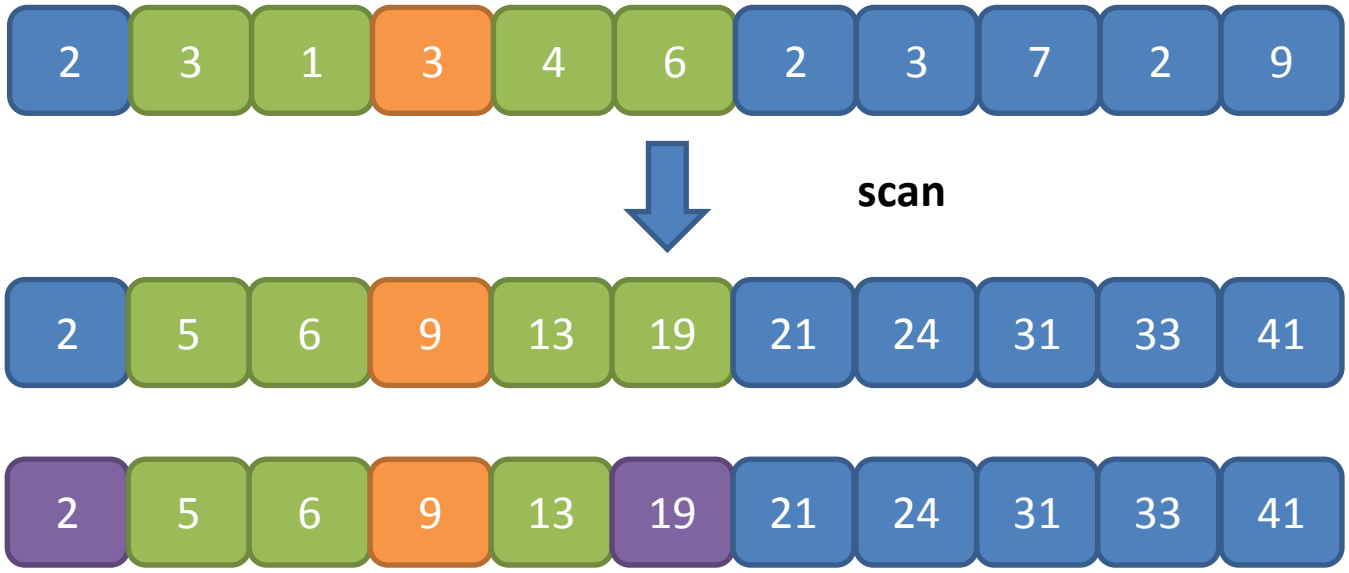
理论基础

- Guassian过滤可以通过多次Box Filter模拟



通过Scan求出Box Filter

计算第四个像素的BoxFilter，Blur半径为2



Average = (19 – 2) / (2 * 2 + 1) = 3.4

具体实现步骤

1. 应用Computer Shader计算Scan结果
   - 水平方向Scan（每一行分配一个Thread Group），计算水平方向的box filter
   - 竖直方向Scan（每一列分配一个Thread Group），计算竖直方向的box filter
2. 重复第一步，直到遍历次数足够（2-3次）

总结

- OpenGL 4的一些新的特性与功能
- OpenGL 4的优化技巧
- Tegra K1 Demo

- Q&A?

let me share

2014

www.chinagdc.com.cn