第 1 篇 准备

（

在引擎的消费上下文（“消费”一词不好？需要修改）中（如init shader context, render context）：

1.把外部数据转为内部数据

如：init shader data, init shader state

2.使用Result.tryCatch

为了提高性能；

为了简化代码；

3.把内部数据转为外部数据

如：init shader state –》 main state，通过Result.tryCatch把异常转为Result

）

# 第1章 了解Web 3D和搭建开发环境

## 1.1 了解3D编程

XXXXXX

#### 1.一级小标题

#### 2.二级小标题

## 1.2 了解Web 3D的历史

## 1.3 了解WebGL的基础知识

### 1.3.1 渲染管线

### 1.3.2 坐标系变换

## 1.4 准备预备知识

## 1.5 搭建开发环境

### 1.5.1 了解Reason版本和开发环境

### 1.5.2 下载及安装VSCode

### 1.5.3 在VSCode中搭建Reason的开发环境

## 1.6 编写第一个Reason程序

# 第2章 搭建测试环境

## 2.1 了解自动化测试

### 2.1.1 单元测试

### 2.1.2 集成测试

### 2.1.3 端对端测试

## 2.2 了解运行测试

### 2.2.1 通过log测试

### 2.2.2 通过WebGL-Inspector测试WebGL

### 2.2.3 通过Spector测试WebGL

## 2.3 了解性能测试

### 2.3.1 使用Profile测试

### 2.3.2 使用Timeline测试

### 2.3.3 使用Console.profile和Performance测试

### 2.3.4 使用Memory测试

第 2 篇 基础

本篇主要介绍了基础知识；

# 第3章 函数式编程与函数反应式编程

（介绍本章内容）

## 3.1 函数式编程

（介绍函数式编程是什么、函数式编程与面向对象编程的区别）

### 3.1.1 函数式编程思想

（介绍思想）

下面给出一个简单的例子，用函数式编程和面向对象编程分别实现（使用Reason语言），从而让我们能清楚地看到两者的区别：

（给出例子）

使用函数式编程有以下优点：

（1）粒度小

相比面向对象编程以类为单位，函数式编程以函数为单位，粒度更小。

正所谓：

我只想要一个香蕉，而面向对象却给了我整个森林

（2）性能好

大部分人认为函数式编程差，主要基于下面的理由：

* 柯西化、函数组合等操作增加时间开销
* map、reduce等操作，会进行多次遍历，增加时间开销
* Immutable数据每次操作都会被复制为新的数据，增加时间和内存开销

但是通过“Reason的编译优化+Immutable/Mutable结合使用+递归/迭代结合使用”，可以解决这些问题：

* 由于Bucklescript编译器在编译时的优化，柯西化等操作和Immutable数据被编译成了优化过的js代码，大幅减小了时间开销
* 由于Reason支持Mutable和for,while迭代操作，所以可以在性能热点使用它们，提高性能。

（3）擅长处理数据，适合3D领域编程

通过高阶函数、柯西化、组合等工具，函数式编程可以像流水线一样对数据进行管道操作，非常方便。

3D程序有大量的数据要操作，从函数式编程的角度来看：

3D程序=数据+逻辑

因此，我们可以：

* 使用Immutable/Mutable、Data Oriented等思想和数据结构表达数据；
* 使用函数表达逻辑；
* 使用组合、柯西化等工具，把数据和逻辑关联起来。

### 3.1.2 本书相关的函数式编程知识点

本书使用Reason语言来实现函数式编程。它属于“非纯函数式编程语言”，而我们熟知的Haskell，属于“纯函数式编程语言”。所以，本书的函数式编程属于“非纯函数式”的。

（介绍两者的区别）

使用非纯函数式编程语言主要是因为以下几点：

（1）更高的性能

Reason支持Mutable、迭代操作，提高了性能。

（2）更简单易用

* Reason允许非纯操作，所以不需要使用Haskell中的各种Monad来保持“纯”，减少了用户负担。
* Reason使用严格求值，而它相对于惰性求值更简单。

下面介绍相关知识点：

1.不可变数据

创建不可变数据之后，对其任何的操作，都会返回一个复制后的新数据。

示例如下：

Reason的变量默认为immutable：

let a = 1;

/\* a为immutable \*/

Reason也有专门的不可变数据结构，如Tuple、List、Record。

这里以Record为例，它类似于Javascript中的Object。

（1）定义Record的类型：

type person = {

age: int,

name: string

};

（2）定义Record的值：

let me = {

age: 5,

name: "Big Reason"

};

（3）使用这个Record，如修改“age”的值：

let newMe = {

...me,

age: 10

};

Js.log(newMe === me); /\* false \*/

newMe是从me复制而来的。任何对newMe的修改，都不会影响me。

在Wonder中的应用：

（1）在编辑器中的应用

编辑器的所有数据都是Immutable的，这样的好处是：

* 不用关心数据之间的关联关系，因为每个数据都是独立的
* 不用担心状态被修改，减少了很多bug
* 实现Redo/Undo功能时非常简单，直接把Immutable的数据压入History的栈里即可，不用深复制/恢复数据。

（2）在引擎中的应用

大部分函数的局部变量都是Immutable的（如使用tuple,record结构）。

（call Discriminated Union Type to Sum Type? ? ?）

### 3.1.3 函数式编程示例

（给出一个综合的示例，只用函数式编程实现）

## 3.2 函数反应式编程

### 3.2.1 函数反应式编程思想

### 3.2.2 了解Most库

### 3.2.3 函数反应式编程示例

# 第4章 Reason基础

(介绍Reason特色，以及为什么要用Reason)

## 4.1 Reason语法

## 4.2 Reason示例

# 第5章 Reason-React和Redux基础

## 5.1 Reason-React基础知识

## 5.2 Redux基础知识

## 5.3 Reason-React 和 Redux综合应用示例

第 3 篇 搭建雏形

这一篇主要搭建引擎和编辑器的雏形。

# 第6章 编写一个最小的3D程序“绘制三角形”

（介绍本章内容）

## 6.1 需求分析

（

修改？：把“引擎和编辑器的需求分析”（

////给出本书涉及的所有功能模块？

根据目前的知识，识别出核心的功能模块（

编辑器核心功能需求：编辑静态场景，导入/导出，发布本地包

根据编辑器的核心功能需求，分析出引擎的核心功能需求！

（如引擎：光照和非光照。（不是识别出所有的功能））

）

）提前；然后此处结合他们的分析，来给出Demo程序的需求分析？

////或许不应该这样做！因为本书的开发是迭代开发，一开始并不知道引擎和编辑器的全貌，也就是并不知道所有或大部分的功能模块

）

首先，我们分析程序的目标和特性；

接着，根据特性，我们进行头脑风暴，识别出功能关键点和功能扩展点；

最后，根据功能关键点和功能扩展点，我们给出具体的需求。

### 6.1.1目标和特性

目标

从程序中提炼出最小的引擎和编辑器。

特性

为了这个目标，程序应该具备以下的特性：

* 简单

程序应该很简单，便于我们分析和提炼。

* 可扩展

为了使从程序中提炼的引擎真正可使用，3D程序需要搭建好框架，在以后填充更多的内容。

### 6.1.2头脑风暴

（提出非功能扩展点（比如优化性能），从而引出“三个三角形”而不是“两个三角形”的功能需求！）

让我们来进行头脑风暴。

1.目标具体化

为了使程序尽量简单，只需要最少的功能；

为了能够扩展，结构需要完整；

2.从功能上分析

最简单的功能就是没有任何交互，只能绘制模型；

最简单的模型就是三角形；

识别功能关键点：

* 只绘制三角形

3.从结构上分析

程序应该有以下两步：

初始化

主循环

进一步分解，识别出最明显的子步骤：

（1）初始化 ＝ 初始化Shader

识别功能扩展点：

①多种GLSL

因为在真正的3D程序中，Shader通常有多个，而每个Shader的一个顶点GLSL和片段GLSL都不一样，所以会有多种GLSL。

（2）主循环的一帧 ＝ 清空画布 ＋ 渲染； 渲染 ＝ 设置WebGL的状态 ＋使用Shader ＋ 设置相机 ＋绘制模型

识别功能扩展点：

②多个渲染模式

比如渲染所有透明模型和渲染所有非透明模型。

③多个WebGL状态

（略）

④多个相机

用户可以切换相机。

⑤多个模型

（略）

⑥每个模型有不同的Transform，即不同的位置、旋转和缩放。

（略）

### 6.1.3结论

需求包含非功能性需求和功能性需求，下面分别分析：

非功能性需求

目前暂时不考虑。

功能性需求

首先分析下功能扩展点：

对于扩展点一，已经是功能性需求了；

对于扩展点二，为了简单，只渲染所有非透明模型；

对于扩展点三，因为只有一个渲染模式，所以所有模型对应的WebGL状态都是一样的（进一步解释清楚？）；

对于扩展点四，为了简单，只有一个相机；

对于扩展点五，意味着有多个三角形；

对于扩展点六，为了简单，每个三角形有不同的位置，不考虑旋转和缩放。

根据功能关键点和上面的分析，我们给出功能性需求：

* 只渲染，没有交互
* 场景有三个三角形
* 所有三角形都是非透明的
* 只有一个固定的相机
* 有两类GLSL

## 6.2 总体设计

（给出main函数和index.html的伪代码（只包含顶层代码））

（分析伪代码）

## 6.3 具体实现

（

1.实现顶层代码

2.重构

3.当实现到geometry数据时，使用vbo实现（需要增加“初始化 vbo”逻辑）

4.实现剩余代码

5.分析代码的不足之处（在“提炼引擎”时，要解决这些问题！）

）

/\*

### 6.3.1 给出模块代码

### 6.3.2 给出每个模块的具体代码，具体分析

\*/

## 6.4 总结

（本章结论）

／／／／（给出本章完整程序所在的Github地址）

//程序在个人主页，而个人主页在本书开头（序言？）中给出

# 第7章 从程序中提炼引擎和编辑器

（增加总体分析：

1.分析引擎、编辑器的关系、职责等

如 从数据角度来看： 编辑器＝用户交互数据？ 。。。。。。

）

## 7.1 引擎需求分析

（性能：高）

（可扩展性：高）

（可维护性：高）

## 7.2 编辑器需求分析

（性能：低）

（可扩展性：中）

（可维护性：高）

## 7.3 提炼引擎

（

wrap type:

note: named wrap type file: XxxWT instead of Xxx is to avoid to conflict with model files

js api不应该接受wrap type参数（用户不需要知道！）

）

（

explain why use Result against "throw exception":

the advantage of use result:

1.explictly show "it may has error" in type

e.g.

for setTransformTranslation api, if throw exception, the editor not know it throw from the type and compile and the func name! so editor not handle it(how should editor handle the exception: catch it and log error message)!

）

（

Result:

For Js API:

Should handle Result inner，the input and output shouldn’t has Result type！

对于性能重要的地方（如Engine－init shader data iterate, render iterate），使用“throw error”，然后在外层Result.tryCatch，转换为Result

）

（尽量多使用List

https://2ality.com/2018/01/lists-arrays-reasonml.html

对于以下情况，使用List：

元素少（《1000》

不需要在任意位置读／写元素

如果是以上情况的反面，先考虑用type array（data oriented），然后再考虑使用array（如shaderCacheMap-.shaderCache）

）

### 7.3.1 设计边界

（

engine:

1.init

2.exec user logic before loop

3.loop

）

### 7.3.2 提炼数据

（

use state to store data

provide api for user to operate it

）

### 7.3.3 分析用户逻辑

（

1.device logic

2.shader logic

3.add gameObject logic

4.set camera data

）

### 7.3.4 识别领域模型

（

User

Data:

state

View:

canvas

Director:

init

(分析：为什么传入canvas id, context param 参数，而不是设置到state上!)

loop:

extract Render:

opt: create vbo firstly when render instead of when Director->init

GameObject:

Transform

Geometry

Material:

one material->one shader

Shader:

GLSL

Program

Camera:

vMatrix

pMatrix

DeviceManager

Utils:

Vector

Matrix

Error

Immutable/MutableHashMap

...

）

### 7.3.5 使用引擎

（

- use by import es6 module

- use reason

- use js

- use single engine js

1.extract api

2.generate index

move to npm(e.g. wonder-generate-index)

3.package engine to one dist file;

4.import it in html page;

5.invoke api to run

）

## 7.4 提炼编辑器

（

／＊

在编辑器的角度，对引擎的wrap type分类：

1.公共的

e.g. Result, ////HashMap

2.

＊／

editor dependent on engine-. Ed api

and wrap type（loose coupling？ 依赖集中在engine adapter-.wrap type；其它地方只依赖wrap type的类型，不依赖它相关的functions（如ShaderName-.create函数））

）

（

refactor:change loopId from int to abstract type

）

（

should add discuss to text.md:

editor data can separate to:

1.show data

2.persistent data

we can have only one state(named show state instead of ui state???):

can move editorState->loopId to uiState;

mark loopId to mutable;

(should verify: change loopId shouldn't cause re-render!!!)

(disadvantage:

1.not all immutable

2.mix the two logic!

)

or we can have two states!

we choose "two states" design!!!

）

（

# explain why need editor

explain why

explain the relation between editor and engine

）

### 7.4.1 设计边界

（

analyze border:

how can user use it?

give image (draft image with only one "start" button)

start/stop button + canvas

package editor to one dist file;

import it in html page;

）

### 7.4.2 提炼数据

（

## whether to use redux

discuss: global data(store) vs local data(state)

only use global data, no local data?

## design data

state:

ui state(redux store)

engine state

////note: no editor state!!!!

still need editor state for not show data(which not affect ui)!!!

(

by dicuss "where to store loopId" and avoid re-render!

)

why?

no need to update All, Inspector, ...

just update ui state->all, inspector state, ...

need extract ui data from engine state to ui state!

should update engine state and ui state in store->reducer?

）

### 7.4.3 引入引擎

（

use adapter

invoke engine file according to namespace

(work by es6 module)

）

### 7.4.4 使用编辑器

## 7.5 优化引擎性能

（

wrap type:

“vao” has no wrap type, because Gl.vao is unique;

“shaderCacheMap”-.cache data has no wrap type（use array（float） instead）, because it is used when render for optimize;

add “FieldName” wrap type;

）

### 7.5.1 测试性能

（

# benchmark test: show 7000 triangles

# find problem:

profile

timeline?

memory

）

（

# optimize

(optimize each one and compare benchmark)

）

### 7.5.2 优化“初始化”逻辑

（

optimize:

get shader info only when debug

why set "isDebug" to state data instead of state?

(and why mutable?)

）

### 7.5.3 优化“主循环”逻辑

（

- cache location

////utils: add sparse map

- set camera data only once

////(split uniform data to camera data, model data)

- cache uniform data

not all uniform data need cache

(e.g. matrix data not cache; array number(int/float) data can cache)

/\*

need test difference perf!!!

add MutableHashMap for performance:

uniformCacheMap, shaderCacheMap is MutableHashMap

\*/

discuss whether to reduce state in Render->render iterate?

now reduce it instead of:

1.reduce small data(e.g. uniformCacheMap)

2.use mutable(e.g. change shaderCacheMap to mutable hashMap)

reduce state has these advantage:

1.improve code maintainablity

unify to reduce one state, instead of many small data;

immutable data has little bug than mutable data;

reduce state has these disadvantage:

1.bad performance

spend more time for gc

- cache use program

- use vao(extension)

get vao extension:

directly unsafeGet, because most devices support it!

judge last vao:

not improve perf in benchmark!

(because each gameObject->geometry->vao is different!!! should improve perf when there are share geometrys!)

）

### 7.5.4 分析最终优化结果

# 第8章 改进引擎和编辑器

## 8.1 加入契约检查

（说明为什么需要契约检查）

### 8.1.1 引擎加入契约检查

（

////在运行时－》函数执行时检查isDebug，而不是在运行时－》module（文件）执行时，

requireCheck,

ensureCheck,

requireAndEnsureCheck.

requireCheckBythrow,

ensureCheckBythrow.

Shader-.compileShader: not use test, because need log two infos

Shader-.linkProgram: not use test for perf

Init shader iterate and render iterate use xxxBythrow！

）

（

xxxCheck的bodyFunc中也可能返回Result：

解决方案：

1.bodyFunc不再返回Result

如Matrix-. buildPerspective

2.增加xxxCheckBodyResult函数，处理该情况

）

（

解决循环依赖：

为了解决Matrix-. buildPerspective-.contract check 依赖 StateData-.Datatype-. CoordinateTransformationMatrix-.Matrix的循环问题：

extract Data to StateData, DebugData

）

（

重构：使用sequence代替bind, map

Render-. \_sendUniformShaderData

）

### 8.1.2 编辑器加入契约检查

（

RenderStore-.stop:

let stop = state => {

RenderPst.unsafeGetLoopId(PersistentData.getState())

|> Result.tap(DomExtend.cancelAnimationFrame)

|> Result.handleError(Error.throwError);

DataAPIEngine.unsafeGetState()

|> Result.bind(state =>

state

|> DeviceManagerAPIEngine.unsafeGetGl

|> Result.tap(DeviceManagerAPIEngine.clear)

)

|> Result.handleError(Error.throwError);

{...state, isStart: false};

};

分开处理还是只处理一次error？

应该分开处理，因为：

这是分别处理persistent state和engine state，两者都终止了（最终都setState或者副作用调用（此处为副作用调用，意思是返回unit））。

因为都终止了，所以应该分别处理！？

）

## 引擎

（不改动编辑器，等这几个引擎改进都完成后，再一起修改编辑器！）

（

当前问题：

0、用户定义了GLSL

用户不应该知道GLSL，只需要知道使用哪个shader

1、对GLSL进行了硬编码

如：

Render.re硬编码了Shader的Field（e.g. a\_position,u\_mMatrix, 。。。。。。）

（解释什么是Field？

顶点Field，uniform Field？

）

2、不够灵活

如：

a.Render.re-.\_sendModelUniformData 限定了GLSL必须有u\_mMatrix字段，否则在获得它的Location会报错。

b.当前限定了GLSL的字段：

顶点字段为a\_position；

uniform字段为u\_mMatrix, u\_vMatrix, u\_pMatrix, u\_color$index

（解释$index的意思！！！）

但是真正的3D程序的GLSL很复杂：

1）有可能有更多的字段

如a\_normal，u\_opacity等

2）在不同的条件下，GLSL不一样

如：

一个GameObject有/无纹理，它的GLSL是不一样的

初步设计：

1解决问题0：把GLSL放到引擎中，用户只需要根据shader name，知道使用哪个shader

2解决问题1和问题2.a：

Render应该与具体的Shader数据隔离

3解决问题2.b：

有两种思路可以解决：

1）GLSL预编译

2）组合GLSL片段

（具体说明下这两种思路的实现方案）

我们使用第二个思路，因为：

1）调试更方便

合成后的GLSL更简洁

2）更易于维护

只关心相关的GLSL片段，不受其它片段影响

进一步设计：

对于设计1: GLSL放到引擎中

有两种方案：

1）GLSL作为引擎文件（module）

2）GLSL作为外部文件（json文件）

考虑到引擎的非功能需求－》扩展性（高），选择第二个方案。

那么需要在初始化引擎前，加载GLSL文件。加载方案有：

1）直接用Promise

2）FRP

3）Async

我们在引擎和编辑器的异步操作中，都采用FRP，原因在于它有以下优势：

（论述FRP优势）

（使用流程图来说明设计方案）

对于设计3: 组合GLSL片段

我们可以把Shader数据分为三个部分：

1）shaders.json

2）shader\_libs.json

3）glsl

init shader时，根据shader name，组合相关的shaders.json-》shader\_libs.json－》glsl片段为vs、fs。

（使用流程图来说明设计方案）

对于设计2: Render应该与具体的Shader数据隔离

应该：

在init shader时，根据shader name，获得对应Shader的顶点和uniform数据，以及send的方法（如uniform3f），将其保存到state上；

然后render每个GameObject时，根据shader name，遍历它们，调用send方法发送对应的Shader数据。

（使用流程图来说明设计方案）

（给出整体的流程图来说明设计方案）

实现方案:

（注意：

shader.json 只有shaders,groups,

glsl：

分成6段（top、。。。。。。）

增加createShaderChunkSystemFile gulp task，预处理glsl

）

add glsl and createShaderChunkSystemFile gulp task:

提出tiny-wonder-glsl-compier repo

（

不能将其写到src/文件中！因为它需要编译为commonjs，而引擎是编译为es6\_global！

）

提出tiny-wonder-commonlib

（因为tiny-wonder-glsl-compier也需要用到immutableHashMap！）

add shaders.json, shader\_libs.json and load shader json：

state-.renderConfigData-.shaders, shaderLibs不使用Wrap type，因为它们只在build glsl source时使用，使用得很少。

）

## 第16章 支持光照

（

修改shader json：

shader\_libs.json:

add “from”

add normal buffer

shaders.json:

add light material shader

）

### 16.1 需求分析

### 16.2 更新引擎

### 16.2.1 增加DirectionLight组件

### 16.2.2 增加LightMaterial组件

### 16.3 更新编辑器

### 16.3.1 更新Inspector

### 16.3.2 更新Asset

### 16.3.3 更新Icon Gizmo

## 8.1 引擎提炼“GameObject+组件”架构

（

提出BasicMaterial组件：

对应修改shader

）

### 8.1.1 分析现有架构

### 8.1.2 设计

### 8.1.3 实现组件

### 8.1.4 修改GameObject

### 8.1.5 修改Render

### 8.1.6 实例：扩展组件

### 8.1.7 实现新需求

## 8.2 引擎提炼Data Oriented架构

（不改动编辑器，等这几个引擎改进都完成后，再一起修改编辑器！）

### 8.2.1 分析现有架构

### 8.2.2 设计

### 8.2.3 实现

（

8.2.4 优化

性能重要处：

初始化

渲染

创建/删除 gameObject, component

transform:

更新mmatrix

////设置父子关系

内存重要处-检查 内存泄漏和回收：

初始化

渲染

创建/删除 gameObject, component

transform:

更新mmatrix

）

### 8.2.4 实例：扩展组件

### 8.2.5 实现新需求

## 8.4 引擎提炼渲染队列

（不改动编辑器，等这几个引擎改进都完成后，再一起修改编辑器！）

### 8.4.1 分析现有架构

### 8.4.2 设计

### 8.4.3 实现

### 8.4.4 实现新需求

## 8.5 引擎支持各种应用场景

（不改动编辑器，等这几个引擎改进都完成后，再一起修改编辑器！）

### 8.5.1 分析现有架构

### 8.5.2 设计

### 8.5.3 实现

### 8.5.4 实例：支持移动端

### 8.5.5 实现新需求

## 8.6 引擎支持多线程

（不改动编辑器，等这几个引擎改进都完成后，再一起修改编辑器！）

（

### 引擎提炼子State：

（

区分：

Main State

Render State

提出allXXXData，放置不同state的子Data的公共类型

/＊

子State的上下文为性能重要的地方：

如Render-.iterate, init shader-.iterate

* /

）

）

### 8.6.1 分析现有架构

### 8.6.2 设计

### 8.6.3 实现

### 8.6.4 实例：增加线程

## 8.8 引擎使用函数式反应式编程实现主循环

（不改动编辑器，等这几个引擎改进都完成后，再一起修改编辑器！）

## ////8.10 错误处理

## 8.9 改进编辑器

9.1 更新引擎

分析现有架构

重构文件夹

错误处理

第 4 篇 增加核心功能

# 第10章 增加Redo/Undo功能

# 第11章 增加“场景管理”功能

## 11.1 需求分析

## 11.2 查看场景中的所有GameObject

## 11.3 操作GameObject

# 第12章 增加Inspector功能

## 12.1 需求分析

## 12.2 显示选中的GameObject的所有组件

## 12.3 操作组件

# 第13章 增加相机功能

## 13.1 需求分析

## 13.2 给引擎增加相机组件

## 13.3 更新编辑器

# 第14章 给编辑器增加“运行”/“停止”功能

## 14.1 需求分析

## 14.2 初步设计

## 14.3 重构：提炼新设计

## 14.4 具体实现

# 第15章 给编辑器增加“资产管理”功能

## 15.1 需求分析

## 15.2 增加Material资产

## 15.3 重构：使用Recursive Type

## 15.4 具体实现

### 15.4.1 实现Asset Tree

### 15.4.2 实现Asset Children

### 15.4.3 实现Asset Inspector

# 第17章 增加导入/导出WDB

# 第17章 支持纹理

（

修改shader json

）

## 17.1 需求分析

## 17.2 更新引擎

### 17.2.1 增加BasicSourceTexture组件

### 17.2.2 给LightMaterial组件增加Diffuse Map

## 17.3 更新编辑器

### 17.3.1 更新Inspector

### 17.3.2 更新Asset

# 第18章 增加“导入”／“导出”功能

（note: Dto type, domain type, serialization, deserialization

refer to 《Domain Modeling》－》 serialization

）

## 18.1 需求分析

## 18.2 更新引擎

### 18.2.1 增加Scene Graph文件（.wdb）

（加更多标题，不能只有一个标题）

（支持indices type -. Short and Int）

## 18.3 给编辑器增加导出功能

### 18.3.1 增加“导出场景”功能

### 18.3.1 增加“导出包”功能

## 18.4 给编辑器增加导入功能

### 18.4.1 增加“导入模型资产”功能

### 18.4.2 增加“导入包”功能

# 第19章 给编辑器增加“发布本地包”功能

第 5 篇 增加更多功能

# ////第20章 给编辑器增加“控制台”功能

# ////第21章 增强编辑器的Scene View的渲染

## 21.1 总体需求分析

## 21.2 给编辑器增加“网格平面”功能

## 21.3 给编辑器增加“Icon Gizmo”功能

### 21.3.1 需求分析

### 21.3.2 实现IMGUI

### 21.3.3 实现相机Gizmo

# 第22章 增强相机功能

## 22.1 总体需求分析

## 22.2 支持事件

### 22.2.1 需求分析

### 22.2.2 让引擎支持事件

### 22.2.3 让编辑器支持事件

## 22.3 增加FlyCameraController组件

### 22.3.1 需求分析

### 22.3.2 给引擎增加FlyCameraController组件

### 22.3.3 让编辑器支持FlyCameraController组件

## 22.4 让编辑器的Scene View的相机使用FlyCameraController组件

# 第23章 让编辑器支持PWA

第 6 篇 应用

这一篇主要讲，引擎和编辑器的应用实例。

# 第24章 使用编辑器制作3D静态场景

## 24.1 需求分析

## 24.2 分析如何实现

## 24.3 准备美术素材

## 24.4 具体实现

## 24.5 总结

# 第25章 使用引擎制作3D动态场景

（通过扩展（job、shader）来实现）

（或者实现最简单的ray tracing场景（渲染球）？）

## 25.1 需求分析

## 25.2 分析如何实现

## 25.3 准备美术素材

## 25.4 具体实现

## 25.5 总结