○一 lua基本

2019年8月10日 9:35

〇二 lua输出

2019年8月10日 9:36

```
--[[
t={a=10,sds="20s0"};
print(t["sds"]);
print(t.a); -- 这两种模式都可以输出
--]]

t1={a=10,sds="20s0"};
t2={"1a","2b","33c"};

for key,val in pairs(t1) do
	print(key..":"..val);
end;
```

〇三 函数的声明和使用

2019年8月10日 9:37

```
--[[ 16 函数的声明和使用 函数就是一种类型
-- 函数不用指定参数类型和返回类型

function fact(n)
	if n ==1 then
		return n;
	else
		return n * fact(n-1);
	end
end;

print(fact(1)); --1
print(fact(4)); -- 24
print(fact);  -- function: 00B6B4C8
--]]
```

〇四 关于function函数的作为参数

2019年8月10日 9:38

17-关于function函数的作为参数传递和匿名函数的用法

```
-- 将函数作为参数 类似C#中的委托与事件

function fun_t17 (tab , fun)
	for k,v in pairs(tab) do
		fun(k,v);
	end
end;

tab = {key1 ="val1" , key2 = "val2" };

function f1 (k,v)
	print(k..":"..v);
end;

fun_t17(tab,f1); -- 直接将函数名作为类型
fun_t17(tab,	-- 使用匿名函数
	function(k,v)
		print(k.."-"..v);
	end);

--将函数与函数内部所有作为参数使用

--]]
```

〇五 关于thread和userdata类型

2019年8月10日 9:39

18-关于thread和userdata类型

thread 线程 lua中没有线程 使用的是协程
协程在同一时刻只能运行一个 而线程可以并行多个

userdata 自定义类型 可以根据需要进行定义
当然甚至可以对现有类型(包括现有函数)进行重写

--]]

〇六 全局变量和局部变量的声明和使用

2019年8月10日 9:39

19-全局变量和局部变量的声明和使用

```
a = 5; --全局变量
print(a);
a = "aa";
print(a);
print(type(a));
print("------------------------");

local b =50;    -- 局部变量
function fun_t19()
    c = 10;     -- 全局变量
    local d = 20; -- 局部变量
end;
print(c); --没有调用函数的时候 无法访问c 和d
fun_t19();
print(c,d); -- c 可以访问 但是局部变量
print("------------------------");

do
    local a = 10; --local 在语句块结束的时候销毁
    b = 11; --全局
    print(a,b);
end;
print(a,b);

--]]
```

〇七 Lua中的多变量同时赋值

2019年8月10日 9:39

```
--[[ 20-Lua中的多变量同时赋值
a,b = 10;
a,b = 11,10.22;
a,b,c,d = 12,10,"sdad";
print(a,b,c);
a,b = b,a;
print(a,b);
b,a,c = a,c,b;
print(a,b,c,d);
print("------------------");
-- 可以返回多个值的函数
function fun_t20()
	return 10,20;
end;
print(fun_t20());
x = fun_t20();
print(x);
x1,x2 = fun_t20();
print(x1,x2);
--]]

--[[ 21-while循环
-- 1.while循环 2.for循环 3.repeat unitl --(do while)
while(循环条件) do
	if()  then
		statement1;
	end
	statement2;
end;

1.数值for循环
```

```
for var = start,end,step do 变量从start 腑tep操作 到stop
        statement;
end;
for i = 1,10,2(每次增加2 默任为1)
        sum = sum + i;
end;
tab1 = {"app1","app2"};
tab2 = {aa="a" ,bb = "b"}
for k,v in pairs(tab1);
        print(k..":"..v); -- 1:app1 2:app2
for k,v in pairs(tab2);
        print(k..":"..v); -- aa:a 2:bb:b
repeat
        循环体
unitl(continue);

i = 1;
repeat
        print(i);
        i = i + 1;
until( i == 10); -- 满足条件不再执行

--]]
```

〇八 Lua流程控制

2019年8月10日 9:40

```
--[[ 24-Lua流程控制
if (条件真) then   --只有 false / uni 为假?
        do somethings;
end;

i= 1;
if (i==1) then
        print("yes");
else
        print("no");
end;

if() then dosth
elseif () then dosth
elseif () then dosth
else dosth
end;

--]]
```

〇九 Lua中的function之 可变参数函数

2019年8月10日 9:40

```
--[[ 25-Lua中的function之 可变参数函数
-- 分为全局函数 和局部函数
function (return v1(, v2))/ local funcition
26-Lua函数中的可变参数
function priStr(s1,s2);
pristr("aaa"); / pristr("aaa","bbb");

自定义的可变参数函数 类似重载
function fun_t25(...)  --用三个点表示参数内容
	-- print(arg[1]); arg 里面保存传过来的参数表
	-- 如果传过来 10 "a" 则
	-- arg = {10,"a",2} 最后的2是第三自带数据 是参数长度
	-- 也就是 arg = {"1" = 10,"2" = "a" ,"3" = 2}
	local args = {...}; -- 直接使用局部变量获取参数表 是不存在最后的长度
	local arg_len = #args; -- #得到  arg的长度
	sum = 0;
	for k,v in pairs(args) do
		sum = sum + v;
	end;
	print(sum);
end;

fun_t25(10);
fun_t25(14,16);
fun_t25(18,20,11);
fun_t25(20,30,40,50);


--]]
```

一十 Lua中的数学运算符

2019年8月10日 9:41

```
--[[ 27-Lua中的数学运算符
数学运算符
+ - * / %(取余) ^(幂函数)

关系运算符
== ~=(不等) > < >= <=

逻辑运算符
and or not
--]]
--[[ 29-Lua字符串定义和转义字符
\n \r 等等 " '' ";
--]]
```

十一 Lua字符串常见操作

2019年8月10日 9:41

```
--[[ 30-Lua字符串常见操作
-- lua实现了string 类似面对对象的
str1 = "My name is Seacition";
--全大写
str2 = string.upper(str1);
--全小写
str3 = string.lower(str1);
-- 替换
str4 = string.gsub(str1,"s","1",2);
-- 返回 = (操作str, 替换str,换成str , 换多少 不写默认全部 )

-- 字符串查找 返回位置索引
index = string.find(str1,"is");

--字符串反转
str5 = string.reverse(str1);

-- 字符串格式化 格式化输出
number1 = 5;
number2 = 10;
print("加法运算: "..number1.." + "..number2.." = "..(number1 + number2));
str6 = string.format("加法运算: %d + %d = %d",number1,number2,number1
+number2)
print(str6);

--char 将整型数字转成字符并连接， byte 转换字符为整数值(可以指定某个字符，默认第一个字符)。
string.char(arg) 和 string.byte(arg[,int])

--计算字符串长度。
string.len(arg); = #arg
```

```
--返回字符串string的n个拷贝
string.rep(string, n)
--]]
```

十二 Lua中的数组的基本特性和定义

2019年8月10日 9:42

```
--[[ 32-Lua中的数组的基本特性和定义

--声明和使用数组
arg = {"arg","lua"};
for i=1,3 do
	print(arg[i]);
end;

-- 表的索引可以为负数 因为数组是通过表实现的{value=key}
arg2 ={};
for i=-3,3 do
	arg2[i] = i;
	print(arg2[i]);
end;


-- 多维数组
arg3s = {{"小名","软件","小","软件"},{"小莉"}}
print(arg3s[2][1]);
--]]
```

十三 Lua中的迭代器函数-pairs ipairs

2019年8月10日 9:42

```
--[[ 34-Lua中的迭代器函数-pairs ipairs
-- 表 迭代器 Paris() fparis() -遇到空值结束
arg6 = {"a","21s","sd"};
for k,v in paris(arg6) --迭代索引和内容
        print(k..":",,v);
end;

-- 自定义迭代器
for 迭代变量 in 迭代函数(对应返回变量)
        dosth;
end;


--]]
```

十四 LUa中的表以及表操作

2019年8月10日 9:43

```
--[[ 35-LUa中的表以及表操作
table1 ={"aa","bb","c","dddd"};
--浅拷贝
table2 = table1;
-- 回收
table1[1] =nil;
table2 = nil; --此时 table1 也是空
table1 ={"aa","bb","c","dddd"};
--函数操作
-- 拼接
print(table.concat(table1));
print(table.concat(table1,"-"));
print(table.concat(table1,"-",2));
print(table.concat(table1,"-",2,4));

--添加
table1[#table1+1] = "ee";
table.insert(table1,"fff");
table.insert(table1,2,"ww");


-- 移除
table1[2] = nil; -- 2号 为空但存在
table.remove(table1,2); --后面的数据往前移动
print(table1[2]);



--]]
```

十五 table的排序和取得最大值

2019年8月10日 9:43

```
--[[ 38-table的排序和取得最大值
-- 表的排序
table38 = {"a3","b","2","a","B","A3"};
for k,v in pairs(table38) do
	print(v);
end;
print("----------------");
table.sort(table38);
for k,v in pairs(table38) do
	print(v);
end;


-- 5.2之后的版本取消了 table.max
-- 取最大值的方法 可以自己写一个:
table38_2 = {10,50,322,1,5444,123212};
max38 = table38_2[1];
for i = 2, #table38_2 do
	if (table38_2[i] > max38) then
		max38 = table38_2[i];
	end
end;
print(max38);
--]]
```

十六 Lua中的模块(module)

2019年8月10日 9:43

```
--[[ 39-Lua中的模块(module)
--将模块放在一个单独的lua文件中
module39 = {  };
module39.var = "mk";
module39.fun = function ()
        print("that is module funcition");
end;

-- 在另一个文件调用之前
-- 先声明
require "模块名字";
-- 后调用
print(require.var);

--Lua中的C包

--]]
```

十七 Lua中的元表(Metatable)

2019年8月10日 9:44

```
--[[ 41-Lua中的元表(Metatable)
-- 普通表
table41 = {1,2,2,3};
-- 元表 对普通表的一些行为 索引操作 表计算的扩展
metatable41 = {};
table41_2 = setmetatable(table41,metatable41);
print(table41,table41_2);
table41_3 = { table41, metatable41_2 };

--_index 元方法
-- 当你通过键来访问 table 的时候，如果这个键没有值?

-- 那么Lua就会寻找 该table的metatable（假定有metatable）
-- 中的__index 键。如果__index包含一个表格，
-- Lua会在表格中查找相应的键。
-- 我们可以在使用 lua 命令进入交互模式查看：
mytable = setmetatable({key1 = "value1"}, {
  __index = function(mytable, key)
    if key == "key2" then
      return "metatablevalue"
    else
      return nil
    end
  end
})
-- 其他操作 https://www.runoob.com/lua/lua-metatables.html
-- __newindex 元方法用来对表更新，
-- 为表添加操作符
-- __call 元方法在 Lua 调用一个值时调用
-- __tostring 元方法用于修改表的输出行为。
--]]
```

十八 Lua中的协同程序(coroutine)

2019年8月10日 9:44

--[[8-什么是Lua中的协同程序(coroutine)

基本语法方法描述 https://www.runoob.com/lua/lua-coroutine.html
coroutine.create()
创建 coroutine，返回 coroutine， 参数是一个函数，
当和 resume 配合使用的时候就唤醒函数调用

coroutine.resume()
重启 coroutine，和 create 配合使用

coroutine.yield()
挂起 coroutine，将 coroutine 设置为挂起状态?
这个和 resume 配合使用能有很多有用的效果

coroutine.status()
查看 coroutine 的状态
注：coroutine 的状态有三种：dead，suspended，running?
具体什么时候有这样的状态请参考下面的程序

coroutine.wrap（）
创建 coroutine，返回一个函数，一旦你调用这个函数，
就进入 coroutine，和 create 功能重复

coroutine.running()
返回正在跑的 coroutine，一个 coroutine 就是一个线程?
当使用running的时候，就是返回一个 corouting 的线程号

```
co = coroutine.create(
    function(i)
        print(i);
    end
```

```
)

coroutine.resume(co, 1)   -- 1
print(coroutine.status(co))  -- dead

print("----------")

co = coroutine.wrap(
   function(i)
      print(i);
   end
)

co(1)

print("----------")

co2 = coroutine.create(
   function()
      for i=1,10 do
         print(i)
         if i == 3 then
            print(coroutine.status(co2))  --running
            print(coroutine.running()) --thread:XXXXXX
         end
         coroutine.yield()
      end
   end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2))   -- suspended
print(coroutine.running())
```

```
print("----------")
--]]
```

十九 Lua中简单模式下文件的读取

2019年8月10日 9:45

```
--[[ 53-Lua中简单模式下文件的读取


-- 打开文件操作语句如下：
-- io.open (filename [, mode]) 目录 , 模式
-- r+ 以可读写方式打开文件，该文件必须存在。
-- w+ 打开可读写文件，若文件存在则文件长度清为零 若文件不存在则建立该文件。

-- 简单模式
-- 简单模式使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。
-- 以只读方式打开文件
file = io.open("d:\\1.txt", "r")

-- 设置默认输入文件为 test.lua
io.input(file)

-- 输出文件12行
print(io.read())
print(io.read())
print(io.read()) -- 没有第三行 输出为nil
-- 关闭打开的文件
io.close(file)

-- 以附加的方式打开只写文件
file = io.open("d:\\1.txt", "a")

-- 设置默认输出文件为 test.lua
io.output(file)

-- 在文件最后一行添加 Lua 注释
io.write("\n--  test.lua 文件末尾注释")
```

-- 关闭打开的文件
io.close(file)

-- 完全模式
-- 通常我们需要在同一时间处理多个文件。我们需要使用 file:function_name ?
-- 创　?io.function_name 方法。以下实例演示了如何同时处理同一个文件:
--]]
--[[57-Lua中的垃圾回收机制
-- lua 提供了自动垃圾回收的工?
-- 鳜ttps://www.runoob.com/lua/lua-garbage-collection.html

-- Lua 运行了一个垃圾收集器来收集所有死对象
-- 垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。

--]]

二十 Lua中的面向对象怎么实现

2019年8月10日 9:45

```
--[[ 58-Lua中的面向对象怎么实现

--我们知道，对象由属性和方法组成。LUA中最基本的结
--构是table，所以需要用table来描述对象的属性。
--lua 中的 function 可以用来表示方法。那么LUA中的
--类可以通过 table + function 模拟出来

-- 元类
Rectangle = {area = 0, length = 0, breadth = 0}

-- 派生类的方法 new
function Rectangle:new (o,length,breadth)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  self.length = length or 0
  self.breadth = breadth or 0
  self.area = length*breadth;
  return o
end

-- 派生类的方法 printArea
function Rectangle:printArea ()
  print("矩形面积为 ",self.area)
end;




-- 创建对象
-- 创建对象是为类的实例分配内存的过程。
-- 每个类都有属于自己的内存并共享公共数据。
r = Rectangle:new(nil,10,20)
```

```
-- 访问属性
-- 我们可以使用点号(.)来访问类的属性：
print(r.length)

-- 访问成员函数
-- 我们可以使用冒号：来访问类的成员函数：
r:printArea()


-- 完整实例

-- Meta class
Shape = {area = 0}
-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- 基础类方法 printArea
function Shape:printArea ()
  print("面积为 ",self.area)
end

-- 创建对象
myshape = Shape:new(nil,10)
myshape:printArea()

Square = Shape:new()
-- 派生类方法 new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
```

```
  self.__index = self
  return o
end

-- 派生类方法 printArea
function Square:printArea ()
  print("正方形面积为 ",self.area)
end

-- 创建对象
mysquare = Square:new(nil,10)
mysquare:printArea()

Rectangle = Shape:new()
-- 派生类方法 new
function Rectangle:new (o,length,breadth)
  o = o or Shape:new(o)
  setmetatable(o, self)
  self.__index = self
  self.area = length * breadth
  return o
end

-- 派生类方法 printArea
function Rectangle:printArea ()
  print("矩形面积为 ",self.area)
end

-- 创建对象
myrectangle = Rectangle:new(nil,10,20)
myrectangle:printArea()

-- 函数重写
-- Lua 继承  https://www.runoob.com/lua/lua-object-oriented.html


--]]
```

二一 Lua 数据库访问

2019年8月10日 9:46

```
---[[ 三 Lua 数据库访问

-- Lua 连接MySql 数据库:
require "luasql.mysql"

--创建环境对象
env = luasql.mysql()

--连接数据库
conn = env:connect("organization","root","123456","127.0.0.1",3306)

--设置数据库的编码格式
conn:execute"SET NAMES UTF8"

--执行数据库操作
cur = conn:execute("select * from tbl_user")

row = cur:fetch({},"a")

--文件对象的创建
file = io.open("role.txt","w+");
--
while row do
    var = string.format("%d %s\n", row.id, row.name)

    print(var)

    file:write(var)

    row = cur:fetch(row,"a")
end
```

file:close() --关闭文件对象
conn:close() --关闭数据库连接
env:close() --关闭数据库环境

--]]

○一 Unity初识

2019年8月10日 9:28

入门学习知识点

2019.2.24 - 初开始接触完整的unity的使用和制作
在此之前玩了一个2D大项目对其进行汉化，简单的汉化，但是text组件的编码导致中文乱码
所以无奈之下只能进行资源的修改
最后还是实现了，其中对一组件 资源 等等有体验

1.材质、代码、各种资源可以直接加载到对象上，可以被称之为组件
2.带有各种组件的游戏对象可以被作为预设，这个预设包含游戏内容所需要的数据和效果
3.这种预设既可以直接使用，也可以在动态建立的时候进行修改
4.碰撞的时候 可以让一个被碰走的物体的质量远小于要碰上去的物体 其中
    rigidbody 中的一个 mass数值1->0.01 越小该物体的质量越小 就容易被其他物体弹开
5.物理材质。可以决定摩擦弹性等等
    具体的操作是建立一个物理材质 physic Material
    然后将bounciness的值设为0->1 值越大碰撞的物体越容易被弹开
6.设置里的 physicManager 可以改变重力gravity 的值 是负数 去掉符号值越大越重
7.tag 标签 给组件或者对象进行种类或者属性标记，这样在代码中就可以利用这些标签来修改
8.看了一些其他的项目

2019.2.25 - 系统的学习unity界面和功能使用基础
1.放大、移动快捷键 qwert
2.edit--snap setting 这个设置可以让物体按固定的长度/大小/角度 进行变化 按住ctrl进行操作

3.关于预设 prefab 如果预设生成的游戏对象 的某个属性发生了变化，那么预设对此生成的游戏对象

相应的属性就无法被影响，但没有被修改的属性 任然会收到预设属性的修改而改变

2019.2.26 - 系统学习地形制作、碰撞 刚体

1.刚体让物体带有物质结构和性质，包括摩擦，重力，质量，弹性

2.碰撞器 collider 在一般的地面 地形会有自带的，是可以检测物体与物体直接的接触函数

一般碰撞函数有三个 在不同的时期由系统自动调用触发

OnCollisionEntet 刚刚进入

OnCollisionStay 保持接触

OnCollisionExit 刚刚离开接触分离

如果物体之间需要发生碰撞，至少有一者存在刚体 ，需要检测的一方需要有碰撞器

3.触发器，碰撞器勾选 is trigger 的时候会变成触发器 成为 Trigger

其中不会发生碰撞，而是会穿过去，但是仍然会触发碰撞的函数

此时的函数为触发函数 OnTriggerEnter 。。

〇二 Unity之API笔记

2019年8月10日 9:31

2019.2.27 API学习

1.学会看官网的document 文档手册

https://docs.unity3d.com/Manual/index.html

2.游戏基本设置中的数据

//一些游戏中比较重要的数据

```
//deltatime 每一帧之间的时间间隔
Debug.Log("time.deltaTIme = " + Time.deltaTime);
//游戏时间方面的
Debug.Log("time.fixedTime = " + Time.fixedTime);
Debug.Log("time.time = " + Time.time);

//timeSinceLevelLoad 游戏从开始加载所需要用的时间
Debug.Log("time.timeSinceLevelLoad = " + Time.timeSinceLevelLoad);

//framecount 游戏当前帧数
Debug.Log("time.frameCount = " + Time.frameCount);
//realTimeSinceStartUp 游戏从开始到暂停获得这个数据的时间
Debug.Log("time.realtimeSinceStartup = " + Time.realtimeSinceStartup);
Debug.Log("time.smoothDeltaTime = " + Time.smoothDeltaTime);

// timescale 游戏间歇设置数据
Debug.Log("time.timeScale = " + Time.timeScale);
Debug.Log("time.unscaledDeltaTime = " + Time.unscaledDeltaTime);
```

3.让物体向前走 物体.Translate（vector3.forward * TIme.deltaTime ）

因为这个句子是在update中写的 所以会运行很多次 物体就会运行的很快

解决就是 乘以帧数之间的时间间隔 这样时间间隔* 当前执行次数就是一秒

也就是一秒走了forward的一米

4.////创建 对象的方法

```
//GameObject cubeBox = new GameObject("Cube");

////第二种方法 克隆任意Object
//GameObject.Instantiate(Prefabs);

////第三种 利用创建原始的形状、对象
//go = GameObject.CreatePrimitive(PrimitiveType.Sphere);
```

////给游戏对象添加组件的方式

//go.AddComponent<Rigidbody>(); //添加刚体

//go.AddComponent<APIEvent>(); //添加自己写的脚本代码都可以

5.////设置对象的的激活状态

```
//Debug.Log(go.activeInHierarchy);
//go.SetActive(false);
//Debug.Log(go.activeInHierarchy);
//Debug.Log(go.tag);

////对象和对象的组件都从 gameObject继承而来， 组件本身是没有名字的
////所以 下面两个输出的结果是一样的 都是之前生成的Sphere
//Debug.Log(go.name);
//Debug.Log(go.GetComponent<Transform>().name);

////可以通过基类gameObject的静态方法 findObjectOfType 来找到泛型（你需要的）对象
//Light light = GameObject.FindObjectOfType<Light>();
////enabled 和 active类似 enabled对组件的使用可否进行操作
////active是对gameObject游戏对象
//light.enabled =  false ;

////对于每一个gameObject游戏对象 都存在他在场景中的坐标
////因此，每一个游戏对象都有不可取消的组件 transform
////以下是查找游戏对象的所有 transform 并且不会查找未激活的对象
//Transform[] ts = GameObject.FindObjectsOfType<Transform>();
//foreach(Transform t in ts)
//{
//    Debug.Log(t);
//   Debug.Log(t.name); //输出获取到（激活的）每一个对象的名字 和对象本身名字
//
```

6.协程的概念，和使用方法coroutine

对于普通的方法，就类似编程中的函数调用，会等带调用的函数执行完毕后

再执行调用函数之后的内容

协程方法：类似线程一样，调用的协程方法，不影响源调用代码的执行 两者是同步的

协程可以看作是C#与unity在线程之间的操作

其二，协程方法可以中间暂停，任意代码出暂停多少时间

遇到了 yield 和return yield 也是返回 但是他用在函数里 可以在某一时刻退出并

返回当前计算保存的值， return 只会返回最终值或者中间计算结果

启动： StartCoroutine("Fade"); //这里写方法的名字即可 在没有参数的情况下

关闭： StopCoroutine("Fade");

==========

2019.2.28

1.//给游戏对象gameObject发送消息 包括以下的所有组件

//target.BroadcastMessage("hellow"); //带返回的 如果没有接受者 会报错

//包含三个参数的发送消息，没有接受者不会报错 发送的attack 是一个函数

//target.BroadcastMessage("Attack",null,SendMessageOptions.DontRequireReceiver);

2.插值和平均

//向前平移 速度恒定 和插值不一样 插值是按比例进行的所以会逐渐变慢

//cube.position = new Vector3(-1, 1, Mathf.MoveTowards(0, 10, 1f));

//乒乓运动 来回的运动，

//cube.position = new Vector3(-1,1, Mathf.PingPong(Time.time * speed, 10));

3.//GetAxis 用来对轴移动，可以双方向移动， 而且具有加速度和 惯性作用

//cube.Translate(Vector3.right * Time.deltaTime * Input.GetAxis("Horizontal"));

//GetAxisRaw 对轴移动，双方向移动 但是返回值为 1或者 -1 效果就是立马进行运动和停止

cube.Translate(Vector3.right * Time.deltaTime * Input.GetAxisRaw("Horizontal"));

cube.Translate(Vector3.forward * Time.deltaTime * Input.GetAxis("Vertical"));

==========

2019.3.2

1.专有向量对于2D平面的坐标组合 vector2 向量

当需要修改一个物体的transform组件的坐标的时候 不能使用以下方式

//transform.position.x = 1;//错误的

正确的操作是 先建立一个可以操作的vector2 然后把位置赋值给这个向量

然后修改向量的具体坐标 如 vector2.x = 2;

最后把这个向量代表的坐标在赋值回去给transform组件的position

Vector2 nowPosition = transform.position;

nowPosition.x = 11;

transform.position = nowPosition;

2.向量长度限定，在角度不变的情况下，让长度边短

print(Vector2.ClampMagnitude(vb, 5));

3.向量运动的一些运动方式

//Movetowards匀速运动 lerp是 先快后慢的运动

va = Vector2.MoveTowards(va, vb, Time.deltaTime);

==========

2019.3.3

1.vector2的一些常用方法

//print(va.magnitude); //平方和

//print(vb.sqrMagnitude); //平方和开根号

//下面这个函数的意思是 将vb向量的长度固定到10 但是方向不变 也就是

//从 6 8 10勾股 到 3 4 5 勾股 所以最后的结果是 (3.0,4.0) 向量

//print(Vector2.ClampMagnitude(vb, 5));

向量的加减乘除

Vector2 res1 = vb - va;

```
Vector2 res2 = va + vb;
Vector2 res3 = va * vb;
Vector2 res4 = vc / 3;
Vector2 res5 = vc / vb;
```

==========

2019.3.4

1.随机数的一些方法

```
//Random.InitState(0);//随机数参数种子 固定的 所以每次随机数分布都是一样的
Random.InitState((int)System.DateTime.Now.Ticks);//利用时间计时周期做种
//print(Random.Range(4, 10));//生成随机数 整数 从4-9 不包含最大的
//print(Random.Range(4,5f));//生成随机小数
```

2.生成随机位置

```
//让cube的位置在一个一米的圆*3范围内随机生成
//cube.position = Random.insideUnitCircle * 3;
cube.position = Random.insideUnitSphere * 3;//球体
```

3.四元数quaternion

```
//以下都是输出对象的旋转角 第一个是vector3 第二个是四元数模式
//cube.eulerAngles = new Vector3(1, 1, 1);//正确给物体的旋转赋值

print(cube.eulerAngles);//欧拉角
//四元数是计算中的使用 一般在代码中不直接使用
print(cube.rotation);//四元数 世界坐标
print(cube.localRotation);//局部坐标的 旋转角四元数
```

4.三元和四元的转换

```
//Quaternion.LookRotation 把三元方向转换成四元数角度方向
（实用）如何让主角的面对角度指向敌人
//获取敌人减去主角的 向量方向
Vector3 dir = enemy.position - player.position;
dir.y = 0;//忽略海拔地形导致的人物 “弯腰”
//设置主角的方向 = 把向量转换成四元数角度后的方向 一下就转过去的方法
player.rotation =  Quaternion.LookRotation(dir);
//主角方向利用插值 一点一点的转向最终方向
player.rotation = Quaternion.Slerp(player.rotation, target, Time.deltaTime);
```

5.刚体组件设置物体的position 并利用改变position来移动物体 比直接用transform更快

```
//print(playerRgd.position == playerRgd.transform.position); //两者相等
移动物体的一个改良，利用带插值运算的 MovePosition方法可以更加平滑的移动物体
//刚体也可以设置物体的移动和坐标等等，并且由于跳过计算 比使用 transform组件更快
//playerRgd.position = playerRgd.transform.position + Vector3.forward * Time.deltaTime * 10;
```

```
//这里利用MovePosition 因为利用了插值运算 所以移动比直接设置位置移动 要更平滑
playerRgd.MovePosition(playerRgd.transform.position + Vector3.back * Time.deltaTime * 20);
```

6.给物体施加一个力

```
playerRgd.AddForce( int );//单位是牛顿
```

7.摄像机的相关数据和方法

```
//获取主摄像机上的camera组件
//Camera mainCamera = GameObject.Find("MainCamera").GetComponent<Camera>();
//或者利用camera的标签tag = MainCamera 这个条件也可以获得
mainCamera = Camera.main;
//获取鼠标与屏幕的交点射线
Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
RaycastHit hit; //这个是碰撞的物体
bool isCollider = Physics.Raycast(ray, out hit); //检测是否有物体与射线碰撞
if (isCollider)
{
   print(hit.collider); //是就输出与射线碰撞到的物体
}
```

8.一些基本的工程项目文件路径

```
print(Application.dataPath);                //工程数据路径
print(Application.streamingAssetsPath); //可以通过文件流读取的路径
print(Application.persistentDataPath);   //可以进行实体化的数据路径
print(Application.temporaryCachePath);        //临时的缓冲数据路径
```

9.其他的一些功能

```
加载网页 Application.OpenURL("http://www.sizhisheng.icoc.bz");
界面截屏 ScreenCapture.CaptureScreenshot("ScreenShot.jpg");
加载场景 需要导入命名空间 Using UnityEngine.sceneManager
   //打开场景的不同方法 和 打开模式 默认模式是销毁当前 加载新的
   SceneManager.LoadScene(1);
   //SceneManager.LoadScene("Menu");
```

〇三 C#中的 is 和 as

2019年8月10日 9:33

一、C#类型的转换

在c#中类型的转换分两种：显式和隐式，基本的规则如下：

- 1、基类对象转化为子类对象，必须显式转换，规则：(类型名) 对象。
- 2、值类型和引用类型的转换采用装箱(boxing)或拆箱(unboxing).
- 3、子类转化为基类对象。
- 4、基本类型互相之间转化可以用Covent类来实现。
- 5、字符串类型转换为对应的基本类型用Parse方法，除了String类型外其他的类型都可以用Parse方法。
- 6、用GetType可以取得对象的精确类型。
- 7、子类转化为基类，采用隐式转换。

二、C#中的is

检查一个对象是否兼容于其他指定的类型,并返回一个Bool值,如果一个对象是某个类型或是其父类型的话就返回为true，否则的话就会返回为false。永远不会抛出异常
如果对象引用为null，那么is操作符总是返回为false，因为没有对象可以检查其类型。

例如

代码如下:

```
object o = new object();
if (o is Label)
{
   Label lb = (Label)o;
   Response.Write("类型转换成功");
}
else
{
   Response.Write("类型转换失败");
}
```

三、C#中as的转换规则

- 1、检查对象类型的兼容性，并返回转换结果，如果不兼容则返回null；
- 2、不会抛出异常；
- 3、如果结果判断为空，则强制执行类型转换将抛出NullReferenceException异常；
- 4、用as来进行类型转换的时候，所要转换的对象类型必须是目标类型或者转换目标类型的派生类型

例如

代码如下:

```
object o = new object();
Label lb = o as Label;
if (lb == null)
{
   Response.Write("类型转换失败");
}
else
{
   Response.Write("类型转换成功");
```

}

使用as操作符有如下几点限制

第一个就是，不用在类型之间进行类型转化，即如下编写就会出现编译错误。

代码如下:

```
NewType newValue = new NewType();
NewType1 newValue = newValue as NewType1;
```

第二个就是，不能应用在值类型数据，即不能如下写（也会出现编译错误）。

代码如下:

```
object objTest = 11;
int nValue = objTest as int;
```

四、as与is的区别

1、AS在转换的同事兼判断兼容性，如果无法进行转换，则 as 返回 null（没有产生新的对象）而不是引发异常。有了AS我想以后就不要再用try-catch来做类型转换的判断了。因此as转换成功要判断是否为null。

2、AS是引用类型类型的转换或者装箱转换，不能用与值类型的转换。如果是值类型只能结合is来强制转换
3、IS只是做类型兼容判断，并不执行真正的类型转换。返回true或false，不会返回null，对象为null也会返回false。

4、AS模式的效率要比IS模式的高，因为借助IS进行类型转换的化，需要执行两次类型兼容检查。而AS只需要做一次类型兼容，一次null检查，null检查要比类型兼容检查快。

五、在进行类型转换的时候，可以按照如下的方式进行选择

1、Object => 已知引用类型
使用as操作符来完成

2、Object => 已知值类型
先使用is操作符来进行判断，再用类型强转方式进行转换

3、已知引用类型之间转换
首先需要相应类型提供转换函数，再用类型强转方式进行转换

4、已知值类型之间转换
最好使用系统提供的Convert类所涉及的静态方法

六、(int)和Int32.Parse(),Convert.ToInt32()三者的区别

1、(int)转换：用在数值范围大的类型转换成数值范围小的类型时使用，但是如果被转换的数值大于或者小于数值范围，则得到一个错误的结果，利用这种转换方式不能将string转换成int，会报错。

2、Int32.Parse()转换：在符合数字格式的string到int类型转换过程中使用，并可以对错误的string数字格式的抛出相应的异常。

3、Convert.ToInt32()转换：使用这种转换，所提供的字符串必须是数值的有效表达方式，该数还必须不是溢出的数。否则抛出异常。

〇四 Time类中的方法属性

2019年8月12日 17:27

```
//一些游戏中比较重要的数据

//deltatime 每一帧之间的时间间隔
Debug.Log("time.deltaTIme = " + Time.deltaTime);
//游戏时间方面的
Debug.Log("time.fixedTime = " + Time.fixedTime);
Debug.Log("time.time = " + Time.time);

//timeSinceLevelLoad 游戏从开始加载所需要用的时间
Debug.Log("time.timeSinceLevelLoad = " + Time.timeSinceLevelLoad);

//framecount 游戏当前帧数
Debug.Log("time.frameCount = " + Time.frameCount);
//realTimeSinceStartUp 游戏从开始到暂停获得这个数据的时间
Debug.Log("time.realtimeSinceStartup = " + Time.realtimeSinceStartup);
Debug.Log("time.smoothDeltaTime = " + Time.smoothDeltaTime);

// timescale 游戏间歇设置数据
Debug.Log("time.timeScale = " + Time.timeScale);
Debug.Log("time.unscaledDeltaTime = " + Time.unscaledDeltaTime);
```

〇五 游戏对象

2019年8月12日 17:29

```
//创建 对象的方法
GameObject cubeBox = new GameObject("Cube");

//第二种方法 克隆任意Object
GameObject.Instantiate(Prefabs);

//第三种 利用创建原始的形状、对象
go = GameObject.CreatePrimitive(PrimitiveType.Sphere);
//给游戏对象添加组件的方式
go.AddComponent<Rigidbody>(); //添加刚体
go.AddComponent<APIEvent>(); //添加自己写的脚本代码都可以

//设置对象的的激活状态
Debug.Log(go.activeInHierarchy);
go.SetActive(false);
Debug.Log(go.activeInHierarchy);
Debug.Log(go.tag);

//对象和对象的组件都从 gameObject继承而来， 组件本身是没有名字的
//所以 下面两个输出的结果是一样的 都是之前生成的Sphere
Debug.Log(go.name);
Debug.Log(go.GetComponent<Transform>().name);

//可以通过基类gameObject的静态方法 findObjectOfType 来找到泛型（你需要的）对象
Light light = GameObject.FindObjectOfType<Light>();
//enabled 和 active类似 enabled对组件的使用可否进行操作  active是对gameObject游戏对象
light.enabled = false;

对于每一个gameObject游戏对象 都存在他在场景中的坐标
因此，每一个游戏对象都有不可取消的组件 transform
以下是查找游戏对象的所有 transform 并且不会查找未激活的对象
Transform[] ts = GameObject.FindObjectsOfType<Transform>();
foreach (Transform t in ts)
{
    Debug.Log(t);
```

```
    Debug.Log(t.name); //输出
    获取到（激活的）每一个对象的名字 和对象本身名字

}

go = GameObject.Find("Main Camera");//通过find函数按对象名字进行查找
go.SetActive(false);

通过对象的标签获得
GameObject gameObject = GameObject.FindGameObjectWithTag("MainCamera");

if (gameObject != null)
{
    gameObject.SetActive(false);
}
```

〇六 Unity生命周期函数

2019年8月12日 17:30

```
// Start is called before the first frame update
void Start()
{
    Debug.Log("start");
}

// Update is called once per frame
void Update()
{
    Debug.Log("update");
}

private void Reset()
{
    print("reset");
}

private void OnEnable()
{

}
private void Awake()
{

}

private void OnRenderImage(RenderTexture source, RenderTexture destination)
{

}
```

```
private void FixedUpdate()
{

}

private void OnApplicationPause(bool pause)
{
  // Debug.Log("pause");
}

private void OnMouseDown()
{
   print("Down");
}

private void OnMouseUp()
{
   print("Up");
}

private void OnMouseDrag()
{
   print("Drag");
}

private void OnMouseEnter()
{
   print("Enter");
}

private void OnMouseExit()
{
   print("Exit");
```

```
}


private void OnMouseOver()
{
    print("over");
}

private void OnMouseUpAsButton()
{
    //当鼠标按下和松开都在同同一个游戏对象上才会触发的方法
    print("UpAsButton"+gameObject);
}
```

〇七 消息传递

2019年8月12日 17:31

```
//给游戏对象gameObject发送消息 包括以下的所有组件
target.BroadcastMessage("hellow"); //带返回的 如果没有接受者 会报错

//包含三个参数的发送消息，没有接受者不会报错 发送的attack 是一个函数
target.BroadcastMessage("Attack",null,SendMessageOptions.DontRequireReceiver);
```

〇八 欧拉角与四元数

2019年8月12日 17:33

```
//GetAxis 用来对轴移动，可以双方向移动， 而且具有加速度和 惯性作用
//cube.Translate(Vector3.right * Time.deltaTime * Input.GetAxis("Horizontal"));
//GetAxisRaw 用来对轴移动，可以双方向移动 但是返回值为 1或者 -1 效果就是立马进行运动和停止
//cube.Translate(Vector3.right * Time.deltaTime * Input.GetAxisRaw("Horizontal") *5);
//cube.Translate(Vector3.forward * Time.deltaTime * Input.GetAxis("Vertical")     *5);


//获取敌人减去主角的 向量方向
Vector3 dir = enemy.position - player.position;
dir.y = 0;//忽略海拔地形导致的人物 “弯腰”


//设置主角的方向 = 把向量转换成四元数角度后的方向 一下就转过去的方法
//player.rotation = Quaternion.LookRotation(dir);

Quaternion target = Quaternion.LookRotation(dir);//最终方向
//主角方向利用插值 一点一点的转向最终方向
player.rotation = Quaternion.Slerp(player.rotation, target, Time.deltaTime);
```

〇九 通过刚体控制物体移动旋转

2019年8月12日 17:34

```
//刚体也可以设置物体的移动和坐标等等，并且由于跳过计算 比使用 transform组件更快
//playerRgd.position = playerRgd.transform.position + Vector3.forward * Time.deltaTime * 10;
//这里利用MovePosition 因为利用了插值运算 所以移动比直接设置位置移动 要更平滑
//playerRgd.MovePosition(playerRgd.transform.position + Vector3.back * Time.deltaTime * 20);

//获取敌人减去主角的 向量方向
//Vector3 dir = enemy.position - playerRgd.transform.position;
//dir.y = 0;//忽略海拔地形导致的人物 “弯腰”
//Quaternion target = Quaternion.LookRotation(dir);//最终方向
////主角方向利用插值 一点一点的转向最终方向
//playerRgd.MoveRotation(Quaternion.Slerp(playerRgd.rotation, target, Time.deltaTime));

//if (playerRgd.rotation == target)//当两者角度一样的时候 开始移动 但是失败了
//    playerRgd.MovePosition(playerRgd.transform.position + Vector3.forward * Time.deltaTime);
playerRgd.AddForce(Vector3.forward * force);//给物体施加一个力;
```

一十 摄像机

2019年8月12日 17:34

```
private Camera mainCamera;
    // Start is called before the first frame update
    void Start()
    {
        //获取主摄像机上的camera组件
        //mainCamera = GameObject.Find("MainCamera").GetComponent<Camera>();
        //或者利用camera的标签tag = MainCamera 这个条件也可以获得
        mainCamera = Camera.main;
    }

    // Update is called once per frame
    void Update()
    {
        //获取鼠标与屏幕的交点射线
        Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit; //这个是碰撞的物体
        bool isCollider = Physics.Raycast(ray, out hit); //检测是否有物体与射线碰撞
        if (isCollider)
        {
            print(hit.collider); //是就输出与射线碰撞到的物体
        }
    }
```

十一 工程项目文件路径

2019年8月12日 17:35

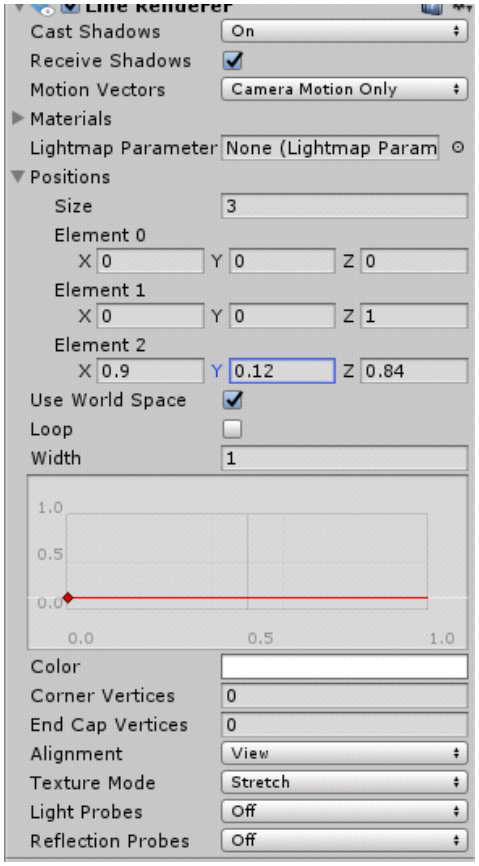
```
//print(Application.dataPath); //工程数据路径
// print(Application.streamingAssetsPath); //可以通过文件流读取的路径
// print(Application.persistentDataPath); // 可以进行实体化的数据路径
// print(Application.temporaryCachePath); // 临时的缓冲数据路径


if (Input.GetMouseButtonDown(0))
{
    Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit; //这个是碰撞的物体
    bool isCollider = Physics.Raycast(ray, out hit); //检测是否有物体与射线碰撞

    if (text.GetComponent<Collider>() == hit.collider)
    {
        //如果text的碰撞器和点击到的物体的碰撞器相等 说明点击到的就是text
        Application.OpenURL("http://www.sizhisheng.icoc.bz");
        print(hit.collider);
        print(text.GetComponent<Collider>());
    }

}
```

十二 LineRenderer 画线组件

2019年8月14日 8:25

在Unity 5.5中对 LIneRenderer 这个组件进行了优化

具体使用LineRenderer 来画线 在任意物体组件可以添加 LineRenderer 组件



组件参数详解

是否启动阴影 一般是2D的此项只对边缘起作用

允许自定义材质给线本身

线有几个点 每个点由一个三维向量构成空间点坐标

并以此按这些点进行连接 绘制成线

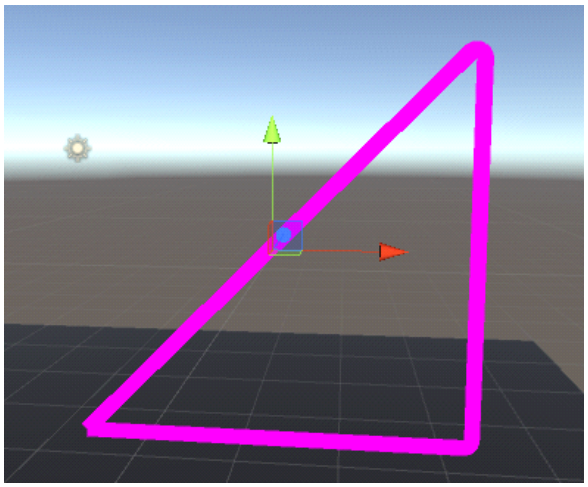
线本身的坐标有两种情况 世界坐标 或者游戏物体坐标

先的宽度 可以调节百分比(基于最大宽度进行百分比调节)

线的颜色 (在有材质的情况下才会生效)

折线拐点的圆滑度(数值越高越平滑)

贴图模式 根据贴图 还是摄像机



实现鼠标按住画任意线段

在摄像机前面放一个黑色的Plane

在移动的时候添加点的坐标到positions里面

在摄像机前面放一个黑色的Plane
在移动的时候添加点的坐标到positions里面
优化的方法 就是 在鼠标不动的时候不要进行添加点
只有鼠标移动的时候才会进行添加


```
private void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            pointList.Clear();
            pointList.Add(GetPoint());
            lineRenderer.positionCount = pointList.Count;
            lineRenderer.SetPositions(pointList.ToArray());
        }
        if (Input.GetMouseButton(0))
        {
            if (Vector3.Distance(
                lineRenderer.GetPosition(lineRenderer.positionCount - 1),
                GetPoint()) > 0.1f)
            {
                pointList.Add(GetPoint());
                lineRenderer.positionCount = pointList.Count;
                lineRenderer.SetPositions(pointList.ToArray());
            }

        }
        if (Input.GetMouseButtonUp(0))
        {

        }
    }

    public Vector3 GetPoint()
    {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        bool isHit = Physics.Raycast(ray, out hit);
        if(isHit)
        {
            return hit.point + new Vector3(0,-0.8f,-0.1F);
        }
        else
        {
```

```
        return Vector3.one;
    }
}
```

十三 Toggle组合选项

2019年8月14日 10:01

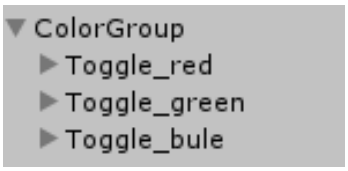
实现用UI 做到多选项单选的功能

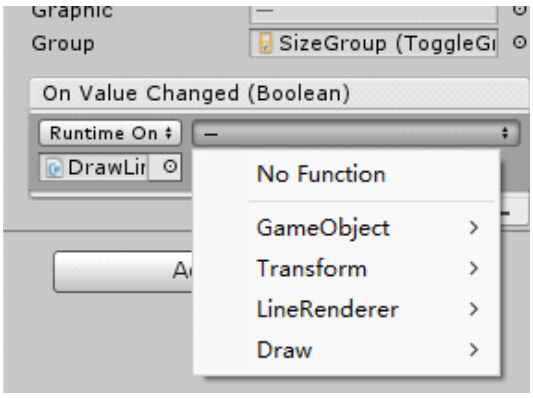
制作流程

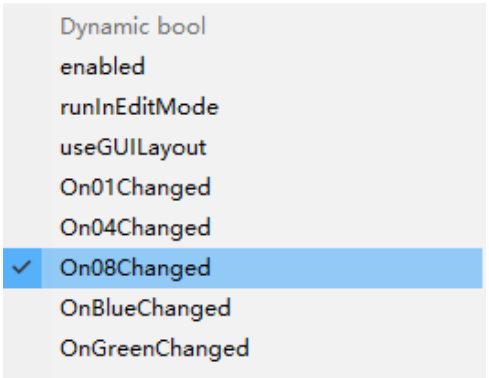
- 创建UI
- 创建一个toggleGroup游戏空物体 并添加组件 ToggleGroup
- 在ToggleGroup下创建几个Toggle
- 设置 isOn 属性去掉勾(选中默认选项为true)
- 在ToggleGroup中添加代码

```
#region OnValueChanged method
 public void OnRedChanged(bool isOn)
 {
    if (isOn)
    {
        paintColor = Color.red;
        SetLineColor(paintColor, paintColor);
    }
 }
 public void OnGreenChanged(bool isOn)
 {
    if (isOn)
    {
        paintColor = Color.green;
        SetLineColor(paintColor, paintColor);
    }
 }
 #endregion
```

- 并将这个含有代码组件的物体给Toggle的响应
- 选择动态触发函数即可





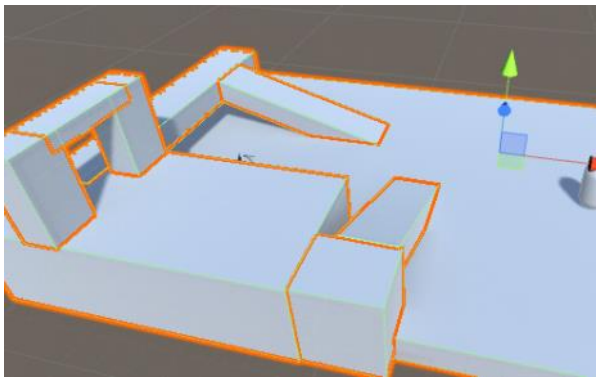


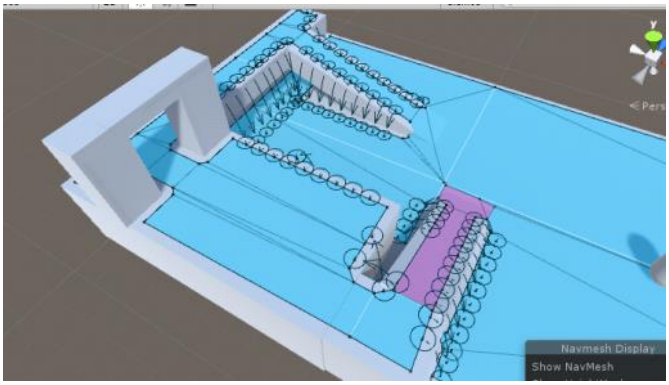
十四 Unity导航系统

2019年8月15日 15:30

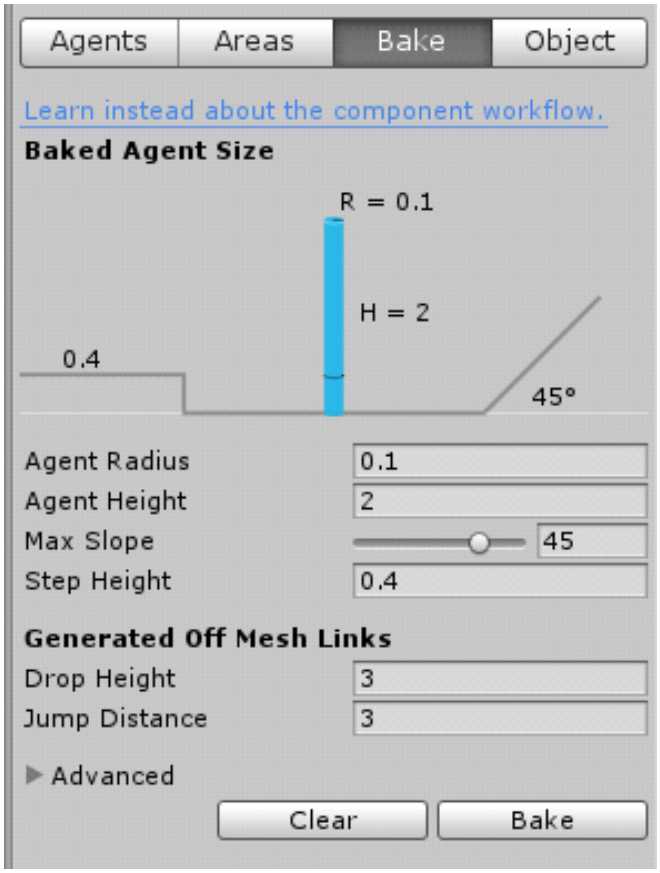
使用 Navigation组件进行实现自动导航

建立地形环境 并设置所有需要烘焙的地形为静态的





从 Window - Navigation 打开导航面板 点击烘焙 得到右图 蓝色部分是可以自动寻路走到的地方



烘焙参数详解

Baked Agent 导航目标

R = radius -- Agent Radius

导航边距 可以走的部分周围有一部分不能走

防止人物掉下去 添加一个边距可以保护人物

H = height -- Agent Height

导航高度 一般由人物高度和建筑物空隙高度

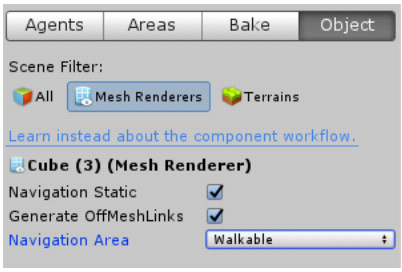
比如大门有2米高 人物有3 米 这样就过不去

最大坡度 最大可以烘焙到的坡度 大于这个度数的坡度上不去

走路的每一步 走多高 如果一个台阶高度大于它 就上不去

跳跃距离和高度设置 如果有两层 最多能从高处跳下去的

最高高度和距离 超过了这个数据 不会从这里跳下去



选择需要可以跳跃的地图区域 勾选 Generate OffMeshLinks 即可跳跃

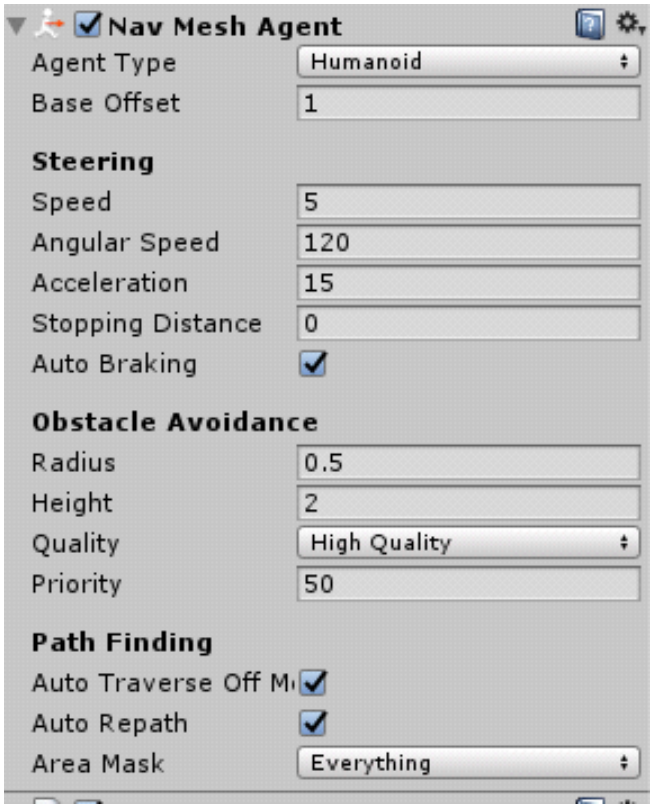
人物设置自动寻路

在人物添加一个 NavMeshAgent组件

组件详解

操作对象的类型 一般是人物

在人物添加一个 NavMeshAgent组件



组件详解

操作对象的类型 一般是人物

操作对象的移动数据

Auto Braking
如果到达目的地 就立马停止 否则有减速

操作对象的 碰撞盒 自身的大小和高度

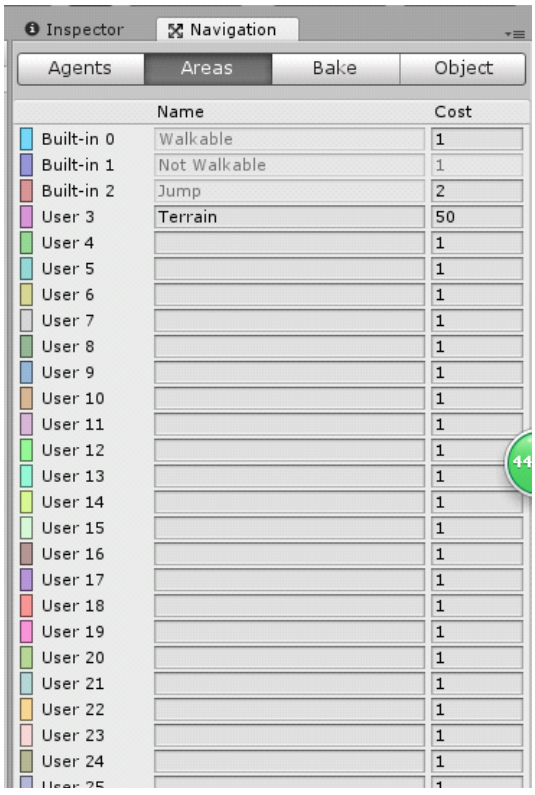
路径自适应 自动获取

自动转弯

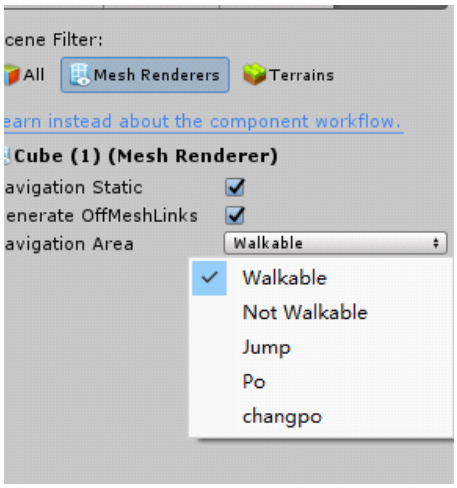
并利用 NavMeshAgent 中的agnet 设置 位置即可自动寻路 其中包含一些 移动加速度 碰撞等

自动寻路设置权重

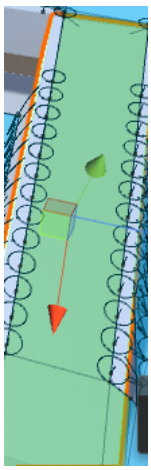
Navigation寻路导航界面下，Areas分页下是在给导航区域分类（相当于分层），以及为每个分类设置不同的消费Cost，意义在于，导航算法中会计算出的是累加起来消耗最低的路径（不一定是视觉上最短可行路径）。例如，设置地面上有一滩沼泽，把该地形新建一个分类，并设置一个很高的消费，那么在正常情况下，寻路将会绕过该区域，走其他消费更低的路径。但若此时游戏中动态生成的物体阻挡了其他路径，只有该路径可走，那么角色将会穿过该沼泽地形进行导航。



解释 对于一些长坡 可以设置他的 权重高 在计算的时候后考虑



设置了代价的路



在计算的时候 优先考虑代价和权重最低的寻路方式
但是如果出现了必须从代价高的地方走
比如最短路被堵死了

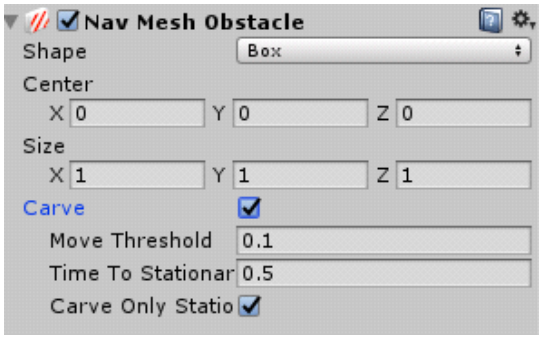
但是如果出现了必须从代价高的地方走
比如最短路被堵死了
只能选择代价和权重高一点的路走

所以该界面的作用在于，可为每种地形自定义分类，并可自定义其可行走的难易程度，来影响导航路径的选择。

利用 NavMeshObstacle 动态规避障碍

普通的烘焙地图是静态的 当地图发生变化的时候 需要重新计算
利用 NavMeshObstacle 可以方便的解决

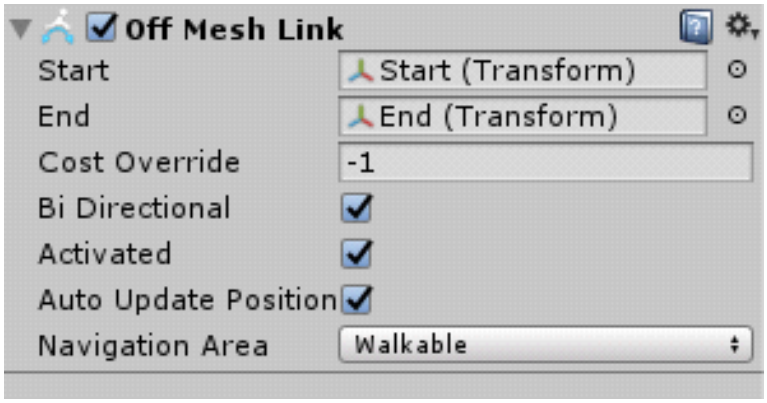
基本的规避障碍物的解决方案
在 障碍物 身上添加 NavMeshObstacle 组件即可 一般针对不经常运动的物体



动态解决方案
让 障碍物 动态影响 导航网格 即可
勾选 Carve 选项即可 如果当前 障碍物 是运动的 就不建议勾选修改导航网格 否则会消耗极大性能

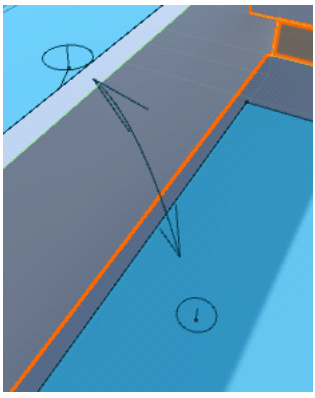
使用 OffMeshLink 组件设置跳跃点

OffMeshLink组件可以在任意游戏物体上添加 并根据需要添加跳跃点 以及跳跃方位和距离



起始位置
中点位置
寻路计算花费
寻路是否是双向的
是否启用当前连接点
游戏模式 自动更新坐标
可以达到的(可以自定义当前地点的寻路花费 和权重)

效果图



设计点击地图自动导航实例 - 自定义运动模式

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class Player : MonoBehaviour
{

    private NavMeshAgent agent;
    //public Transform target;

    public float rotateSmoothing = 7;
    public float speed = 4;

    // Use this for initialization
    void Start()
    {
        agent = GetComponent<NavMeshAgent>();
        //agent.destination = target.position;
        agent.updatePosition = false; // 设置 运动导航非自动
        agent.updateRotation = false; //设置 旋转导航非自动  只是解除了移动组件与物体的绑定 组件还是会移动
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit))
            {
                agent.nextPosition = transform.position;
                agent.SetDestination(hit.point);
            }
        }

        Move();
    }
    private void Move()
    {
```

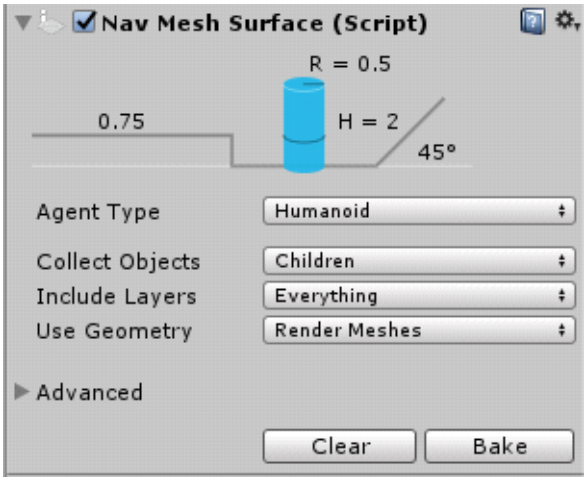
```
        if (agent.remainingDistance < 0.5f)
            return;
        agent.nextPosition = transform.position;
        transform.rotation = Quaternion.Lerp(transform.rotation,
            Quaternion.LookRotation(agent.desiredVelocity), rotateSmoothing * Time.deltaTime);
        transform.Translate(Vector3.forward * speed * Time.deltaTime);
    }
}
```

十五 导航网格动态生成

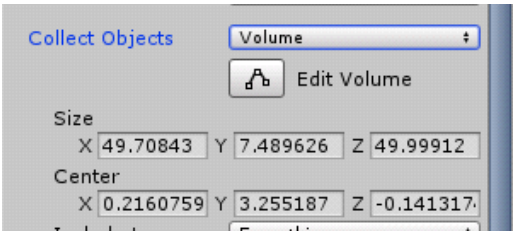
2019年8月15日 16:28

NavMashSurface生成导航网格

利用 代码 在有需要的时候 动态生成导航网格
使用 NavMashSurface (插件组件) 来实现
功能和 自带的 Navigation 相同 都是用来设置 导航网格的烘焙



对于普通的Navigation 会读取长宽高(也可以自定义)
选择 Navigation 中自定义的 参数 但是要与寻路人是同样的类型
烘焙区域选择 可以自定义区域 / 全部区域 / 当前组件的子物体
选择烘焙区域的层级
是否使用几何学



使用代码在游戏进行中动态生成导航网格

```
public GameObject builderPrefab;
private NavMeshSurface surface;
void Start()
{
    surface = GetComponent<NavMeshSurface>();
}

void Update()
{

    if (Input.GetMouseButtonDown(1))
    {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit))
        {
            GameObject go = GameObject.Instantiate(builderPrefab, hit.point, Quaternion.identity);
            go.transform.parent = this.transform;
            surface.BuildNavMesh();
        }
```

```
    }
}
```

○一 初识AssesBoundle

2019年8月12日 17:40

Assets 资源 Bundle 打包 AssetBundle资源打包 为压缩包

1.在游戏运行的时候加载，减少游戏运行硬盘读取开销

2.包依赖，模型依赖于贴图，则模型包会依赖于贴图包 并存在于硬盘

3.lzmc lzm4压缩算法 减少包的大小 下载apk的时候可以不包含AB包资源 等需要的时候在服务器下载

4.根据文件类型会被打包 分为：serialized file（序列化文件） 和 resound files （源文件）

5.在原始资源的下面有打包地方,后缀是随意的,只要开发者知道,其次不分大小写

打包的指定可以带/ xxx/xxx 可以创建结构目录

6.简单的打包代码

```
[MenuItem("Seacition/Build AssetBundles")] //在编译器中添加菜单 并且添加菜单选项
static void BuildAllAssestBundles()
{
    string ABdirectory = "AssetBundlesDirectory";
    if (Directory.Exists(ABdirectory) == false) // 如果目录不存在就创建
    {
        Directory.CreateDirectory(ABdirectory);
    }

    //按指定路径 方式 平台 打包到 dir路径中 dir无法自动创建需要提前创建或者利用代码创建
    BuildPipeline.BuildAssetBundles(ABdirectory,
BuildAssetBundleOptions.None,BuildTarget.StandaloneWindows64);
}
```

〇二 具体工程

2019年8月12日 17:42

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;
using UnityEngine.Networking; //unityWebRequest


public class LoadFiles : MonoBehaviour
{
    private string cubePath = "AssetBundlesDirectory/scenes/cube.u3d";
    private string share = "AssetBundlesDirectory/share.u3d";

    // Start is called before the first frame update
    IEnumerator Start()
    {
        ////从目录读取包
        //AssetBundle.LoadFromFile(cunePath); //先加载共享资源 贴图材质等
        //AssetBundle ab = AssetBundle.LoadFromFile(share);
        ////实例化包里面的预设对象
        //GameObject cubeBlackPrefab = ab.LoadAsset<GameObject>("CubeBlack");
        ////在游戏场景中实例化
        //Instantiate(cubeBlackPrefab);

        //也可以利用全部读取然后逐个实例化
        //Object[] obs = ab.LoadAllAssets();
        //foreach (Object o in obs)
        //{
        //    Instantiate(o);
        //}

        ////////////////第一种加载AB LoadFromMemory( 同步和异步 )
        //异步加载AB的方式  异步创建的是AB包请求 异步调用 所以没有执行完毕就会继续 所以需要返回一个请求
        //因此 当前函数体需要作为协程 并返回等待 等待完成后继续
        //AssetBundleCreateRequest ABrequest =
AssetBundle.LoadFromMemoryAsync(File.ReadAllBytes(cubePath));
        //yield return ABrequest;
        //AssetBundle ab = ABrequest.assetBundle;
```

```
        ////同步加载AB的方式 下面这句话执行完毕后才继续后面的代码
        //AssetBundle ab =  AssetBundle.LoadFromMemory(File.ReadAllBytes(cubePath));
        ////使用包内的资源
        ////实例化包里面的预设对象
        //GameObject cubeBlackPrefab = ab.LoadAsset<GameObject>("CubeBlack");
        ////在游戏场景中实例化
        //Instantiate(cubeBlackPrefab);


        //////////////////第二种加载AB LoadFromFile
        //AssetBundle ab = AssetBundle.LoadFromFile(cubePath);
        //GameObject cubeBlackPrefab = ab.LoadAsset<GameObject>("CubeBlack");
        //Instantiate(cubeBlackPrefab);


        //////////////////第三种加载AB 网络请求 异步
        //需要利用到缓存 第一次从网络下载存到本地 所以需要判cashe
        while (Caching.ready == false)
        {
            yield return null;
        }
        //www 有异常不会报错 会继续执行
        //WWW www = WWW.LoadFromCacheOrDownload(
        //    @"file://M:\UNITYGAME\studyProject\class10_AssetBundle\AssetBundlesDirectory\scenes
\cube.u3d", 1);

        //网页中的斜杠是 ////
        //WWW www = WWW.LoadFromCacheOrDownload(
        //      @"http://localhost/AssetBundlesDirectory/scenes/cube.u3d", 1);
        //yield return www;
        //if (string.IsNullOrEmpty(www.error) == false)
        //{
        //    Debug.Log(www.error);
        //    yield break; // 不执行下面的协程代码 返回协程
        //}
        //AssetBundle ab = www.assetBundle;


        //////////////////使用UnityWebRequest
        //string url = @"http://localhost/AssetBundlesDirectory/scenes/cube.u3d";
        string url = @"file://M:\UNITYGAME\studyProject\class10_AssetBundle\AssetBundlesDirectory\scenes
\cube.u3d";
```

```
        UnityWebRequest request = UnityWebRequestAssetBundle.GetAssetBundle(url);
        yield return request.Send();
        //AssetBundle ab = DownloadHandlerAssetBundle.GetContent(request);
        AssetBundle ab = (request.downloadHandler as DownloadHandlerAssetBundle).assetBundle;


        GameObject cubeBlackPrefab =     ab.LoadAsset<GameObject>("CubeBlack");
        Instantiate(cubeBlackPrefab);

    }




    // Update is called once per frame
    void Update()
    {

    }
}
```

〇三 在编译器环境使用AB包

2019年8月12日 17:42

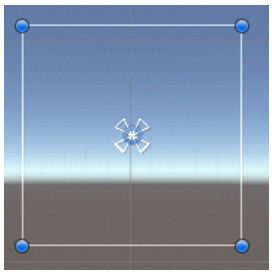
```
using System.IO;
using UnityEditor; //使用打包空间
using UnityEngine;

public class CreateAssetsBundle
{
    [MenuItem("Seacition/Build AssetBundles")] //在编译器中添加菜单 并且添加菜单选项

    static void BuildAllAssestBundles()
    {
        string ABdirectory = "AssetBundlesDirectory";
        if (Directory.Exists(ABdirectory) == false) // 如果目录不存在就创建
        {
            Directory.CreateDirectory(ABdirectory);
        }

        //按指定路径 方式 平台 打包到 dir路径中 dir无法自动创建需要提前创建或者利用代码创建
        BuildPipeline.BuildAssetBundles(ABdirectory, BuildAssetBundleOptions.None,
BuildTarget.StandaloneWindows64);

    }
}
```

○— Rect Transform 解析

2019年8月16日 9:25

什么是RectTransform

RectTransform组件 是普通不可以移除的Transform组件的子类
作为UI的transform 不与屏幕像素进行关联使用
可以认为是 承载 UI 组件的 panel 代表了UI组件的区域



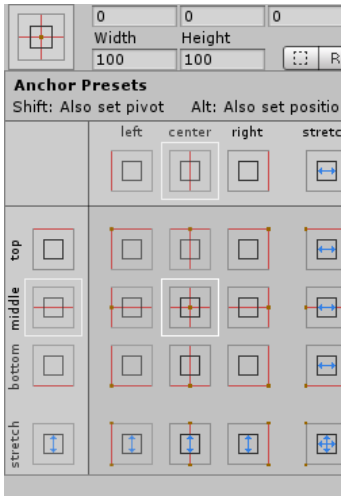
边缘的白框 就可以认为是RectTransform

中间的画板 是 锚点

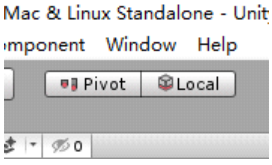
中心的蓝色点 是位置计算点

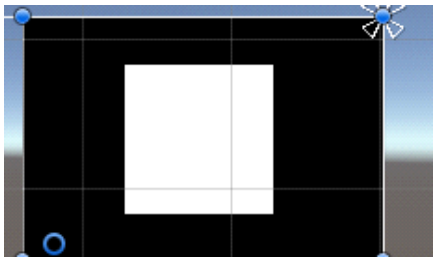
锚点 是针对于 父物体的 对齐方式与 填充UI模式

锚点的各种对其方式 按住ALT 可以快速填充UI

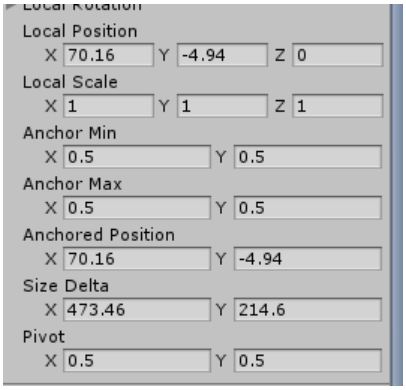


图片的中心点(轴心点) 用来进行UI 自适应 和位置相对固定
图中 蓝色的圈 就是 中心点 通过 下面的 Pivot 按钮启动





中心点/ 锚点 和父物体/子物体的相对位置 有图关系



○一 初识FairyGUI

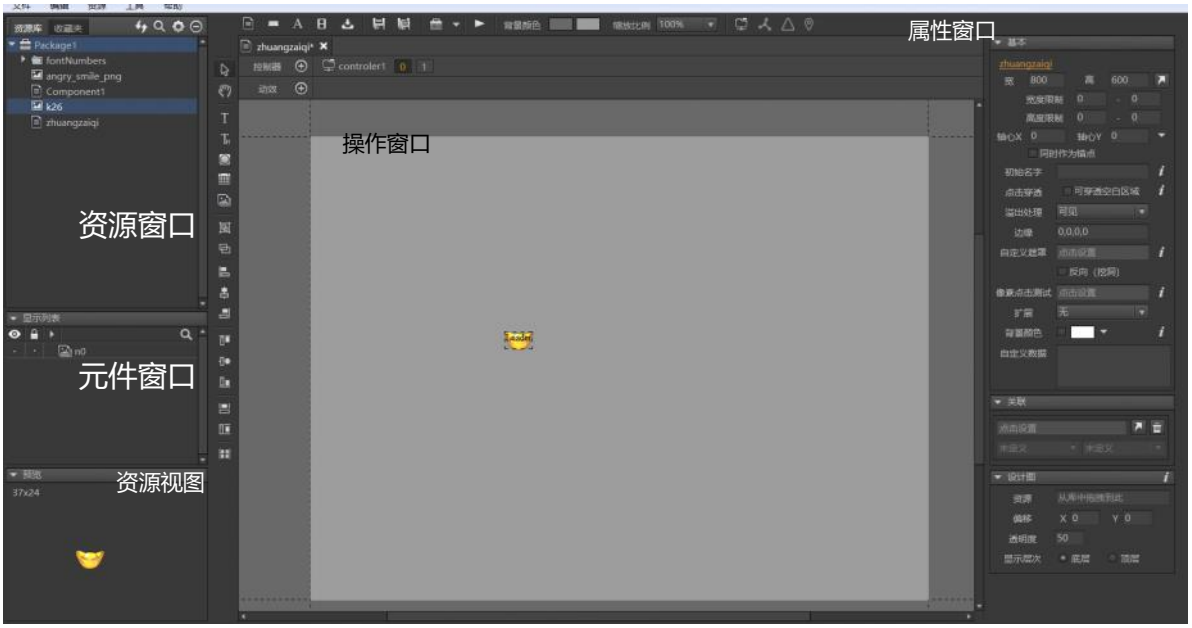
2019年7月30日　　21:40

初识UI编辑工具 FairyGUI

FairyGUI是一款开源免费的针对程序猿, 美工和策划使用的UI编辑工具
并且可以和普通的游戏引擎进行联动和导入导出资源
支持的有 UE4 unity cocos cryEngine ……

二 FairyGUI 使用入门

软件界面与使用介绍



注意 定义项目的时候 需要单独定义当前项目的文件夹 不然会散落在目录里面

三 名词解释与注意事项

- 组件: 类似一个画布, 一个项目可以用很多画布, 每个画布上可以有很多东西
- 元件: 上面的在组件里面的东西, 就是元件, 元件由功能模块,图片,动图,动画等资源构成
- 自定义素材: 可以用序列化图片制作动图 可以用位图进行制作字体等
- 资源: 每一个项目有一个资源包资源包里可以新建文件夹进行管理

四 元件的使用

- 文本: 用来编辑文字 可以自定义字体, 可以用来制作输入框 包括密码 等限制性输入
- 富文本: 可以用来交互的文本
- 动画: 可以是gif 也可以是连续图片制作而成的
- 关联: 将一个元件作为父亲 另一个为关联, 父亲的操作会使关联者动作跟随和位置相对

- 控制器: 设置控制器后, 可以在同一个组件中设置不同的画布 控制器控制同一个元件在不同的配布中有不同的操作和不同的属性 以达到 动态帧和 切换动画的效果
- 动效: 类似关键帧动画 可以对 位置 颜色 大小等进行帧更改 然后动起来就是动态效果
- 按钮: 可以设置图片四种按钮模式的按钮 带文字
- 列表: 带滚动的 可以添加资源(既可以是文字 也可以是设置好的按钮)
- 动态条: 滑动条 进度条 滚动条
  滑动条: 可以手动滑动的条 用户可以操作
  进度条: 通过绑定事件或者参数 可以自动进行进度加载
  滚动条: 用户不可操作,与其他元件关联,比如面板 面板改变 滚动条自动变化
  具体设置 在列表中配合使用, 在列表中的溢出, 将滚动条赋值 就得到
  此外由于列表的大小不一定,所以最好固定滑块的相对大小

〇二 FairyGUI与Unity的联动

2019年8月3日 10:30

五 FairyGUI 打包导出到unity

打包之前需要设置打包资源 右键元件或者组件 作为导出
设置打包路径,设置打包选项 之后就可以打包了
打包的路径建议放在unity的Resources/FairyGUI路径下

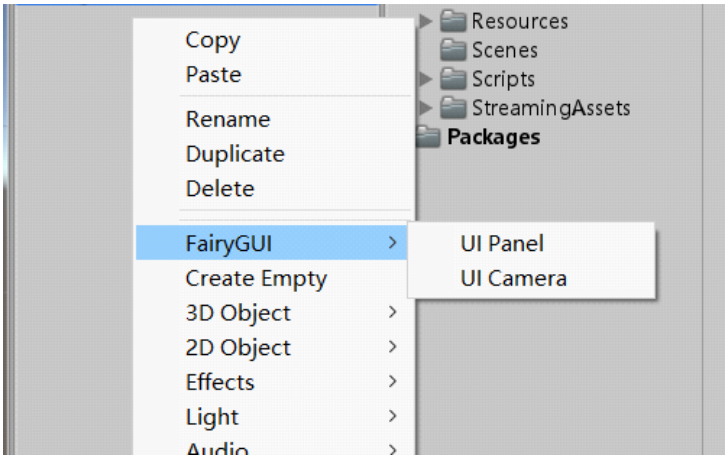
在unity使用UI资源 两种方法

- 第一种: 新建一个FairyGUI的panel 同时自动生成一个只渲染此组件的Camera
  在panel中选择包 并选择包中的组件 取消主摄像机对UI的渲染 避免UI双重渲染
- 第二种: 新建一个空物体,挂载以下代码即可.

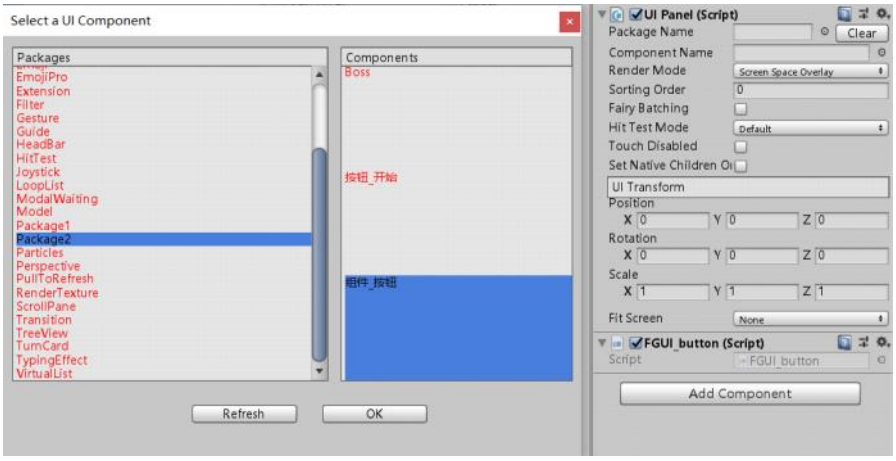
```
//设置渲染面板大小
GRoot.inst.SetContentScaleFactor(800, 600);
//获取GUI打包资源
UIPackage.AddPackage("FGUI/Package1");
//从打包资源获取组件 需要从物体转换成 GUI的组件 用AS 或者自带的 ascom方法即可
//GComponent component = UIPackage.CreateObject("Package1", "列表") as GComponent;
GComponent component ..= UIPackage.CreateObject("Package1", "列表") .asCom;
//instance是一个UI实例 我们将组件给当前实例UI
GRoot.inst.AddChild(component);
```

六 FGUI资源在unity中的使用 以及函数响应

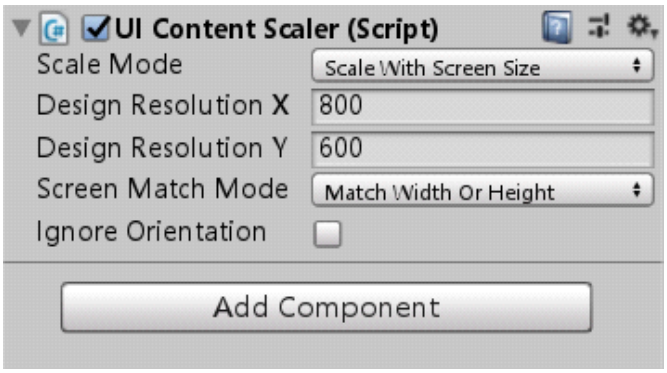
将发布好的资源, 在unity导入的FGUIpackage包中, 新建一个Uipanel



在panel中添加资源, 在资源中找到组件 作为 Groot的子对象



设置渲染的画布大小 需要添加一个脚本 任意地方



GRoot是已经实例化的FGUI在unit与的SDK支持 所有的资源都需要在这个类中进行操作

用代码实例进行一个简单的 点击按钮播放动效的实例

```
using UnityEngine;
using FairyGUI;

public class FGUI_button : MonoBehaviour
{
    private GComponent mainGUI; //主 组件显示panel
    private GComponent bossGIU; //BOSS出现的 panel

    void Start()
    {
        //当前对象挂载了UI 直接获取 BOSS的ui没有 所以进行构建并转换
        mainGUI = GetComponent<UIPanel>().ui;
        bossGIU = UIPackage.CreateObject("Package2", "Boss").asCom;
        // 利用lambda表达式 进行添加事件响应 n0是按钮资源的名字
        mainGUI.GetChild("n0").onClick.Add(  ()=> { PlayUI(bossGIU);}  );
    }

    private void PlayUI(GComponent targetComponent)
    {
        //隐藏当前 按钮
        mainGUI.GetChild("n0").visible = false;
        //在FGUI中添加一个BOSS的新的panel
        GRoot.inst.AddChild(targetComponent);
        //获取FGUI中的 动效 并得到具有动态效果的那一副 t1
        Transition t = targetComponent.GetTransition("t1");
        //播放这个动效 同时利用lambda表达式 写一个回调函数
        t.Play(() =>
        {
```

```
            mainGUI.GetChild("n0").visible = true;
            GRoot.inst.RemoveChild(targetComponent);
        });
    }
}
```

〇三 控制器

2019年8月4日　15:00

关于控制器的使用和代码范例

控制器对于一个组件里面一些元件不用的状态进行控制 在不同的配布下可以拥有不同的状态
这样比较好实现类似timeline的效果 我这里自己根据代码补全写出来的简单控制
主要实现了点击按钮后游戏实现新手引导 让你去点击包裹 点击包裹后可以重复运行


```
private GComponent mainUI;
   private GComponent buttonStart;
   private GComponent buttonPackage;
   private GGroup yindao;
   private Controller controller;

   void Start()
```

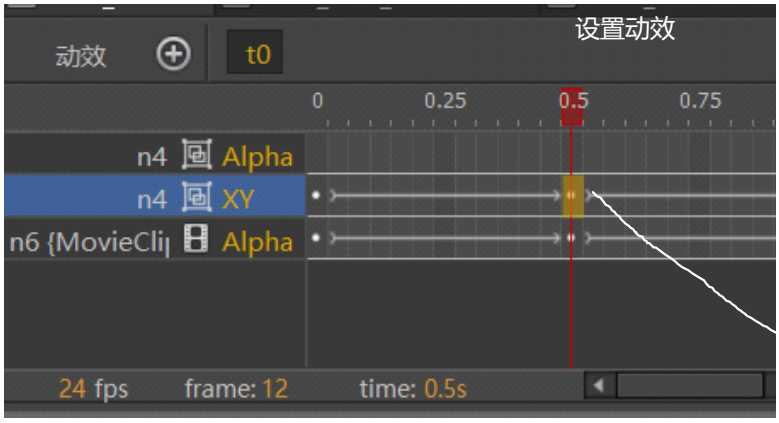
```
{
    mainUI = GetComponent<UIPanel>().ui;
    buttonStart = mainUI.GetChild("n8").asCom;
    buttonPackage = mainUI.GetChild("n0").asCom;
    yindao = mainUI.GetChild("n7").asGroup;

    controller = mainUI.GetController("c1"); //获取控制器

    buttonStart.onClick.Add(() => { Play(); });
    buttonPackage.onClick.Add(() => { Back(); });

}

private void Play()
{
    controller.SetSelectedIndex(1); //设置到执行第二个配布
}

private void Back()
{
    controller.SetSelectedIndex(0);
}
```

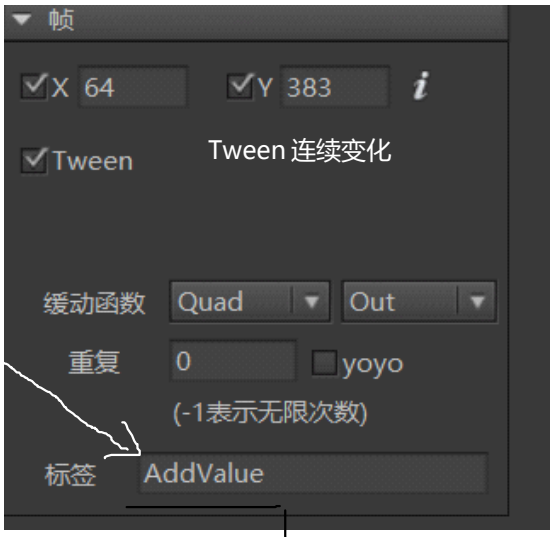
〇四 动效的使用

2019年8月6日 17:00

关于动效的实际使用以及代码实现

用动效制作的动画, 如果需要用代码进行控制, 那么需要在一些动画关键点 或者在某些关键帧设置 标签 这些标签之后可以在代码里面设置 并启用 比如在某个点让战斗力持续变化增加






```
// doween 要引用库 using DG.Tweening;
// FairyGUI导入到Unity已经导入了库

private GComponent mainUI;
private GComponent buffUI;

private float startValue;
private float endValue;

void Start()
{
    mainUI = GetComponent<UIPanel>().ui;
    buffUI = UIPackage.CreateObject("Package3", "组件_增益").asCom;
    //在动效设置的 标签点 进行

    buffUI.GetTransition("t0").SetHook("AddValue", AddAttackValue);
    mainUI.GetChild("n0").onClick.Add(() => { PlayUI(buffUI); });
}

private void PlayUI(GComponent targetCom)
{
    mainUI.GetChild("n0").visible = false;
    GRoot.inst.AddChild(targetCom);
    Transition t = targetCom.GetTransition("t0");
    startValue = 222222;
    int add = Random.Range(8000, 15000);
    endValue = startValue + add;
    targetCom.GetChild("n2").text = startValue.ToString();
    targetCom.GetChild("n5").text = add.ToString();
    t.Play(() =>
    {
        mainUI.GetChild("n0").visible = true;
        GRoot.inst.RemoveChild(targetCom);
    });
}


private void AddAttackValue()
```

```
{
    //参数：初始值，根据时间和帧数确定每一帧需要改变的数据，最终数据，运行时间
    DOTween.To(
        () => startValue,
        x => { buffUI.GetChild("n2").text = Mathf.Floor(x).ToString(); },
        endValue,
        1.7f
   ).SetEase(Ease.Linear).SetUpdate(true); //设置计算不受帧数影响
}
```

〇五 列表的使用

2019年8月8日 9:00

列表的创建与动态装载器 代码及其实现

列表中可以手动的添加按钮 并且可以设置滑动方式 让列表在任何平台使用操作
列表还支持装载器 只需要提供一个按钮模板 和按钮图片
利用代码可以做到动态生成想要的列表按钮组合

操作步骤

- 生成一个组件 内包含一个较大内容的列表 设置列表的滚动方式和溢出方式
- 创建一个按钮元件 并在按钮元件中再添加一个装载器(将装载器的名字命名为 icon)
- 给按钮设置控制器 名字为 button (默认设置按下效果会自动生成并配置控制器)
- 记住要设置默认按钮模板的显示图片为不可见 这样在动态生成按钮列表的时候 不会多次渲染
- 在代码中设置按钮的个数 渲染函数 设置为虚拟列表 这样才能进行滚动和动态装载

代码

```
private GComponent mainUI;
private GList list;

void Start()
{
    mainUI = GetComponent<UIPanel>().ui;
    list = mainUI.GetChild("n0").asList;
    //设置虚拟列表 循环列表一定是虚拟列表 虚拟列表可以通过装载器动态生成按钮
    list.SetVirtualAndLoop();
    //设置列表的渲染函数
    list.itemRenderer = RenderList;
    //设置列表长度
    list.numItems = 5;
    //设置列表的监听事件 每次滑动的时候就会调用
    list.scrollPane.onScroll.Add(DoEffect);
    DoEffect(); //不滑动的时候 最开始也会变大
}

//特效 让列表中间那个按钮变大
private void DoEffect()
{
    //定义中间的那个选哟有特效按钮长度
    float listCenter = list.scrollPane.posX + list.viewWidth / 2;
    //numChildren 是列表中当前在渲染的个数 不是总个数
    for (int i = 0; i < list.numChildren; i++)
    {
        GObject item = list.GetChildAt(i);
        float itemCenter = item.x + item.width / 2;
        float itemWidth = item.width;
        float distance = Mathf.Abs(itemCenter - itemWidth);
```

```
            if(distance < itemWidth)
            {
                float distanceRange = 1 + (1 - distance / itemWidth) * 0.2f;
                item.SetScale(distanceRange, distanceRange);
            }
            else
            {
                item.SetScale(1, 1);
            }
        }
    }
    //自定义的渲染函数
    private void RenderList(int index ,GObject obj)
    {
        GButton button = obj.asButton;
        //设置icon的中心锚点
        button.SetPivot(0.5f, 0.5f);
        button.icon = UIPackage.GetItemURL("Package_List", "n" + (index + 1));
    }
```

〇六 进度条和下拉选项

2019年8月8日 14:54

进度条和下拉选项的实现

进度条和下了菜单制作按流程即可, 主要是通过代码进行控制, 从而避免了UI的复杂创作

FairyGUI内置了一个计时器 可以用一条语句实现时间动态 不需要额外设置

所以可以利用这个来制作技能CD

下拉菜单中的选项有自己的索引值 和对应的内容 内容是string 需要利用 Convert类(在system库)进行 数据类型转换

详细代码如下

```
        private GComponent mainUI;
        private GProgressBar progressBar;
        private GComboBox comboBox;

        private void Start()
        {
            mainUI = GetComponent<UIPanel>().ui;
            progressBar = mainUI.GetChild("n0").asProgress;
            comboBox = mainUI.GetChild("n1").asComboBox;

            progressBar.value = 0;
            comboBox.onChanged.Add(SetCompleteTime);

        }

        //选择下拉菜单后 改变进度条加载时间
        private void SetCompleteTime()
        {
            //设置进度值
            progressBar.value = 0;
            //设置加载到多少进度 , 设置加载时间
            progressBar.TweenValue(100,Convert.ToInt32(comboBox.value));
        }
```

〇七 技能释放与冷却

2019年8月8日 14:59

利用进度条简单独立制作了技能释放与冷却的效果

制作技能式的进度条 将进度条的进度和背景设置为同一个技能图片
背景图片的透明度降低为一半 数值显示为当前数值
进度图片使其填充为360度填充 最大数值 改为技能冷却CD
并且改变进度数值为反向 从15-0 这样的变化
这里为了方便 我获取了按键输入 并提取按键的数字内容 并根据数字对应响应某个技能

```
private GComponent mainUI;
private List<Skill> skills;
private List<GProgressBar> skillBar;
private string[] skillNames = { "蛟龙镇海", "灰烬冰河", "碧落凝珠", "焚天力绝" };
KeyCode currentKey;

//内部的技能类
private class Skill
{
    public GProgressBar skill;
    public string skillName;
    public float skillTime;
    public bool isInCD;

    public Skill(GProgressBar skill, string skillName, float skillTime, bool isInCD)
    {
        this.skill = skill;
        this.skillName = skillName;
        this.skillTime = skillTime;
        this.isInCD = isInCD;
    }

    public void ShowSkill()
    {
        Debug.Log("你使出了技能 " +
            "[" + this.skillName +
            "] 对敌人造成了"
            + UnityEngine.Random.Range(100, 999) +
            "伤害");
    }
}

private void Start()
{
    currentKey = KeyCode.Alpha1;
    skills = new List<Skill>();
    skillBar = new List<GProgressBar>();
    mainUI = GetComponent<UIPanel>().ui;

    skillBar.Add(mainUI.GetChild("n10").asProgress);
    skillBar.Add(mainUI.GetChild("n11").asProgress);
    skillBar.Add(mainUI.GetChild("n12").asProgress);
    skillBar.Add(mainUI.GetChild("n13").asProgress);

    for (int i = 0; i < skillBar.Count; i++)
    {
        skills.Add(new Skill(skillBar[i], skillNames[i], 0, false));
    }
}


private void Update()
{
    SetCD();
}
```

```
/// <summary>
/// 设置 在cd内 无法进行再次技能释放
/// </summary>
private void SetCD()
{
    foreach (var item in skills)
    {
        if (item.skillTime >= item.skill.max)
        {
            item.skillTime = 0;
            item.isInCD = false;
        }
        if (!item.isInCD) //如果没有CD 不显示CD数字
        {
            item.skill.GetChild("title").visible = false;
        }
        else
        {
            item.skillTime += Time.deltaTime;
            item.skill.GetChild("title").visible = true;
        }
    }
}

void OnGUI()
{
    // 获取键盘 输入信息
    Event e = Event.current;
    if (e.isKey)
    {
        currentKey = e.keyCode; //获取按键信息
        string key = System.Text.RegularExpressions.Regex.Replace(currentKey.ToString(), @"[^0-9]+", "");
        int keyNumber = Convert.ToInt32(key) - 1; //将按键转换成数字
        if (keyNumber < 4)
        {
            if (!skills[keyNumber].isInCD)
            {
                skills[keyNumber].isInCD = true;
                skills[keyNumber].skill.value = skills[keyNumber].skill.max;
                skills[keyNumber].skill.TweenValue(0, (float)skills[keyNumber].skill.max);
                skills[keyNumber].ShowSkill();
            }
        }
    }

}
```

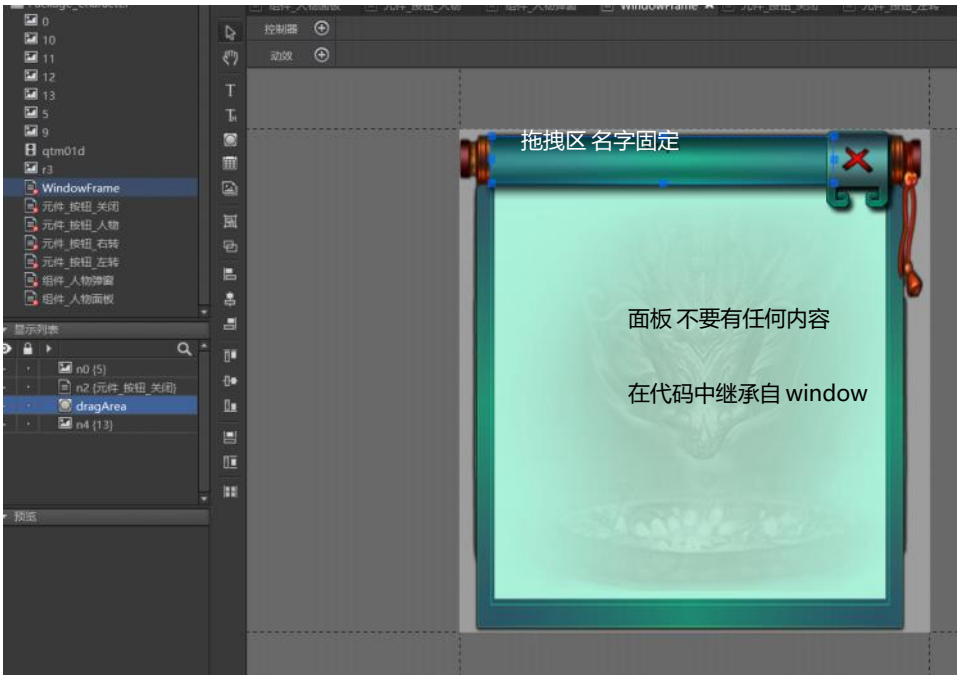
〇八 弹出式菜单

2019年8月8日　17:09

弹出式菜单制作一个人物属性面板

需要两个内容

- 第一 是 frame 并且这个组件的名字必须是frame 他继承自FGUI的windo



第二是需要另一个组件来承载这个面板组件 所以一共要两个组件面板

实现注意事项

属性面板继承自FGUI中的window 面板 并由show()函数启动

启动之前需要重写inint()父类方法 对面板内容进行渲染

渲染函数可以不必关闭 设置visible属性可以对其进行隐藏和显示 性能要比再次打开要好

若要实现面板的拖拽, 不必写任何代码 只需要设置一个空的图像 将其名字设为dragArea即可

面板与面板内容分离 面板单独写 在加载面板里面加载 在加载后的面板里面写 内容(图片/文字/按钮)

```
//生成并显示一个图2组件
      playerButton.onClick.Add(() => { playerWindow.Show(); playerWindow.SetVisible(true); });
//显示之前和进行重写渲染函数
      protected override void OnInit()
      {
            //给面板进行赋值
            this.contentPane = UIPackage.CreateObject("Package_Character", "组件_人物弹窗").asCom;

            this.contentPane.GetChild("n3").onClick.Add(() => { RotateLeft(); });
            this.contentPane.GetChild("n4").onClick.Add(() => { RotateRight(); });
            this.contentPane.GetChild("frame").asCom.GetChild("n2").onClick.Add(() => { ClosePanel(); });
      }
```

〇九 3D渲染2D

2019年8月9日 10:46

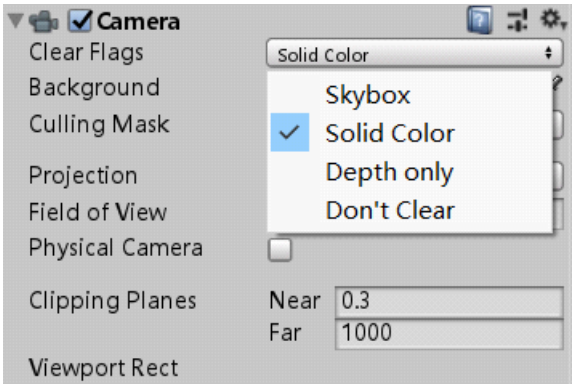
将3D模型利用摄像机单独渲染并转成材质贴图为2D图像

这里和三渲二有一点区别就是 模式不同
一个利用shader制作 一个利用Material制作

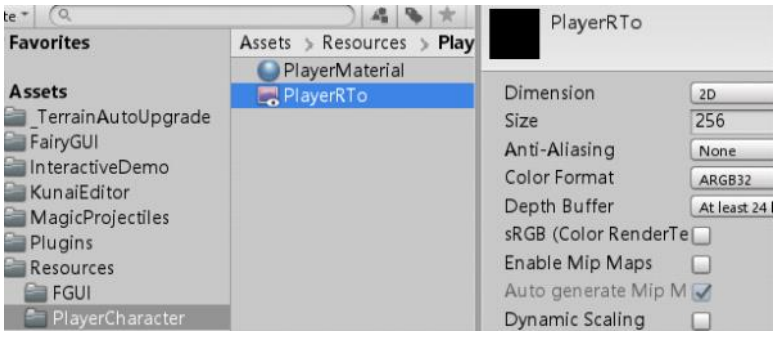
制作方法

设置一个摄像机对准 3D模型并设置模型和摄像机对应渲染

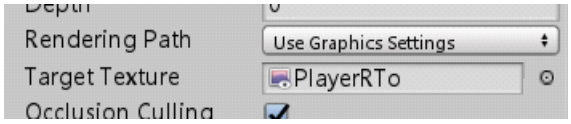
设置摄像机的clear flags 为单色(以方便不渲染后面的内容)



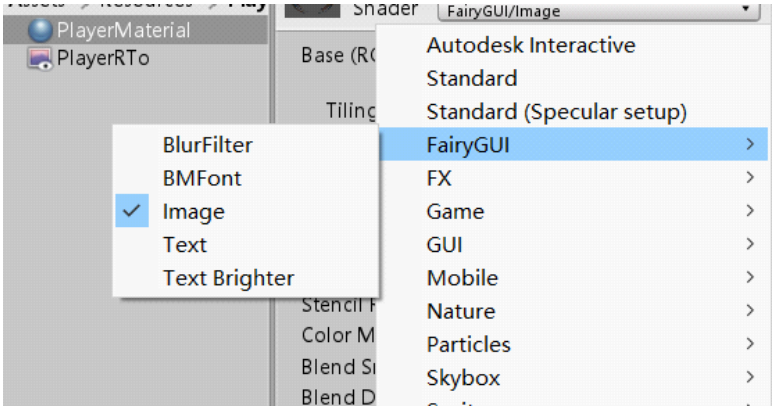
在Resources目录下新建一个可以加载的RenderTexture



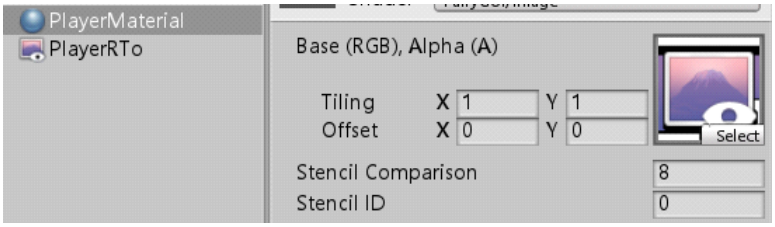
将RenderTexture 赋值给 摄像机进行渲染



同样在Resources目录下新建一个material 并设置材质模式 为 FGUI的 image



将摄像机渲染模式下的 RenderTexture 赋值给 image 模式的Material



这样就得到了一个渲染着模型的材质了

接下来利用代码渲染在面板上

```
this.contentPane = UIPackage.CreateObject("Package_Character", "组件_人物弹窗").asCom;
//获取组件面板中的图片元件
GGraph holder = contentPane.GetChild("n2").asGraph;
//从建立好的资源中加载 模型贴图
RenderTexture renderTexture = Resources.Load<RenderTexture>("PlayerCharacter/PlayerRTo");
//加载材质
Material material = Resources.Load<Material>("PlayerCharacter/PlayerMaterial");
//创建图像 并设置材质和贴图
Image img = new Image();
img.texture = new NTexture(renderTexture);
img.material = material;
//给组件面板渲染图像进行赋值
holder.SetNativeObject(img);
```

一十 弹窗响应

2019年8月10日 8:46

利用弹窗响应, 结合列表制作简单的背包系统

背包系统需要

- 列表以及与列表绑定的按钮 按钮
- 按钮需要严格遵从命名规则 图标为 icon 文字为title 事件响应为button

按钮所需内容	默认按钮边框	选中按钮边框	背景图片	显示图片	文字
命名规则	Button	任意	任意	Icon	Title
备注	按钮本体	默认控制器	显示在最底层的格子背景	最后覆盖最上层的格子显示	格子文字一般不显示

- 按钮需要负载一个相同规则的装载器 实现以上命名规则之后 即可快速实现
- 完整的弹窗和嵌入的列表列表与带装载器的按钮进行绑定

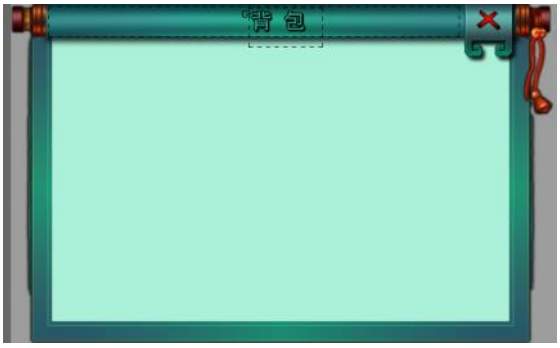
响应弹窗规则

- 需要一个写好的默认窗口框架 并且框架满足以下命名规则

使用内容	背景图片	关闭按钮	拖拽图形区域	文字
命名规则	background	closeButton	dragArea	Title
备注	弹窗的背景	显示或者隐藏弹窗的按钮	可以拖拽弹窗的拖拽区域	文字显示

- 新建一个弹窗组件 拖入上面的框架后 将框架命名为 frame 这样才能实现以上表格功能

新建一个新的组件----->
将框架拖入 并设置列表
设置列表溢出滚动条



滚动条与列表的绑定

需要设置滚动条 包含

- 滚动背景
- 滚动把手
- 上下箭头
- 一般设置把手大小固定

- 上下箭头
- 一般设置把手大小固定

将做好的滚动条与列表绑定

- 点击列表的溢出选项
- 拖入滚动条到滚动条组件
- 默认是紧贴 可以设置边距

列表的设置可以参照右图

程序代码部分

```
protected override void OnInit()
  {
    this.contentPane = UIPackage.CreateObject("Package_Backpack", "组件_背包弹窗").asCom;
    list = this.contentPane.GetChild("itemList").asList;
    list.itemRenderer = ListRender;//设置自定义的渲染函数
    list.numItems = 30;//设置列表数量
    for (int i = 0; i < list.numItems; i++)
    {
      if (list.GetChildAt(i).asButton.icon != null)
      {
        GButton button = list.GetChildAt(i).asButton;
        button.onClick.Add(() =>
        {
          ClickButtonItem(button);
        });
      }

    }

  }

  //自定义的渲染函数 渲染内容索引 渲染内容
  private void ListRender(int index, GObject obj)
  {
    //我们列表里面的都是button 所以将obj转换
    GButton button = obj.asButton;
    //设置按钮的图标和标题文字
    button.icon = UIPackage.GetItemURL("Package_Backpack", "i" + index);
    button.title = index.ToString();

  }

  private void ClickButtonItem(GButton button)
  {
    useItemButton.icon = button.icon;
    useItemButton.title = "物品" + button.title;
  }
```

十一 遥感设计与实现

2019年8月10日 15:29

〇一 粒子系统初见基本模块

2019年8月11日 15:26

Particle System 粒子系统的简介

可以有三种启动方式

- 在Assets下创建
- 在场景资源的effect加载创建
- 在物体添加粒子组件(但是会缺少材质)

粒子系统主要模块参数

Particle system 基本模块属性

属性	含义	数值	备注
Playback Speed	粒子发散时间	>0	
Playback Time	粒子发射总时间		
Speed Range	粒子移动速度范围	{x y}	
Particles	一次循环发射的粒子数		
Duration	发射时间	>0	
Loop	粒子是否循环发射		
Prewarn	粒子预热		让粒子生成和轨迹预先计算
Start Delay	延迟启动时间		
Start Life Time	单个粒子的生存周期		
Start Speed	起始发射速度		初速度 受形状 碰撞 重力影响
3D start size	勾选后 修改3D视角大小		与自身的scale无关
Start size	发射初始大小		与scale和3Dsize 关联
3D start rotation	起始发射 3D旋转角		与3D 视角有关的旋转
Start Rotation	起始发射旋转	范围量	粒子发射的时候随机旋转量
Flip Rotation	运动过程中旋转偏移	范围量	粒子在运动的时候旋转量
Start color	起始发射颜色		
Gravity Modifier	重力修饰量		粒子受到重力影响量
Simulation space	模拟空间场		Local 粒子位置与物体一致 world 粒子位置与世界一致

			custom粒子位置与运动一致
Delta time	变化时间		运动周期的时间变幻
Scaling Model	大小模型		Local 大小只与自身scale一致 Hierarchy 与父物体一致 Shape 与形状一致
Play on awake	是否启动运行		是 则游戏运行就会播发 否 需要调用组件方法才播放
Emitter velocity	发射器操作		粒子发射器操作对象 受 Transform 或 刚体rigidBody
Max Particles	最大粒子数目		单次循环能发射的最大粒子数
Auto random seed	运动随机		旋转 速度的变化是随机的
Stop action	粒子生命周期结束动作		可以被 隐藏 销毁 继续运动等
Culling Mode	补充模式		
Ring buffer mode	循环缓冲模式		

〇二 Emission/Shape

2019年8月11日 16:27

emission 发射模块 是粒子系统的基本模块

Rate over time 发射个数

Rate over distance 发射距离

Burst 发射爆发 实现某个时间点爆炸发射

Time	Count	Cycles	Interval	Probability
爆发时间	爆发粒子数目	循环次数	时间间隔	

Shape 形状模块 设置发射器的发射形状

可以对发射器改变发射形状和范围

可以根据需求发射点线面等具体

也可以设置为点/球体发射器/圆锥发射器等发射模型

Velocity over lifetime 模块 控制旋转

可以在local/world 模式下 让粒子围绕轴进行旋转

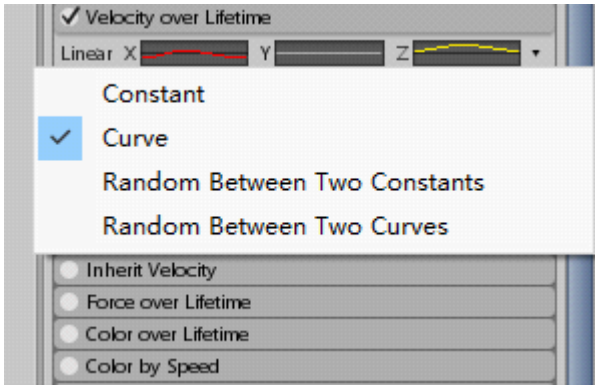
〇三 曲线参数设置

2019年8月11日　　17:11

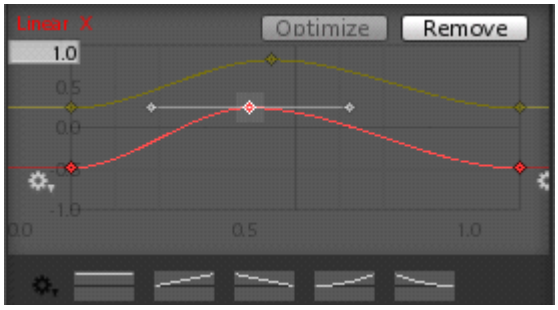
参数的四种设置模式以及特点

- Constant 恒定模式
- Curve 曲线模式
- Random Between two constant 在两个恒定值之间随机
- Random between two curves 在两个曲线之间岁间

在轴操作设置为curve模式 可以得到曲线操作



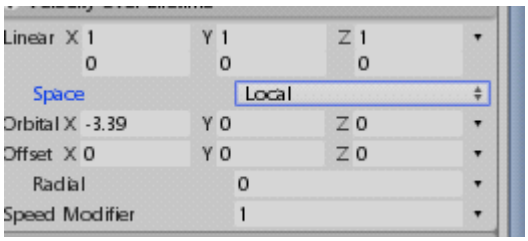
设置曲线 一般有折线 和 曲线 曲线开始为切线



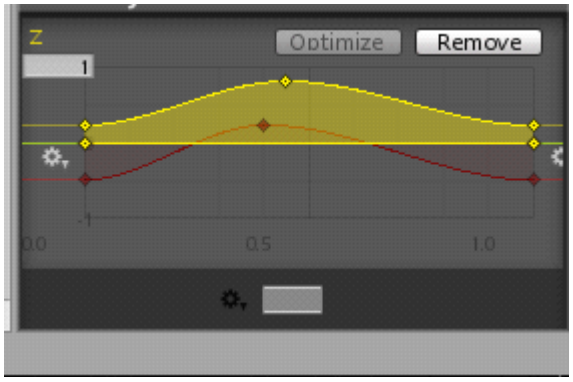
如果需要单独设计某一条曲线 可以点击其他曲线会变为灰色不可设置

进而只对高亮参数进行曲线设置

在两条直线 也就是固定值之间设置参数



在两个曲线之间设置 也就是坐标区域之间



其他类型数据 生命周期 移动旋转和速度 数目
都可以通过曲线进行随机动态设置

〇四 粒子动态颜色设置

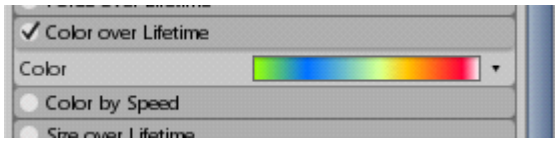
2019年8月11日　　17:25

在粒子移动时改变粒子颜色

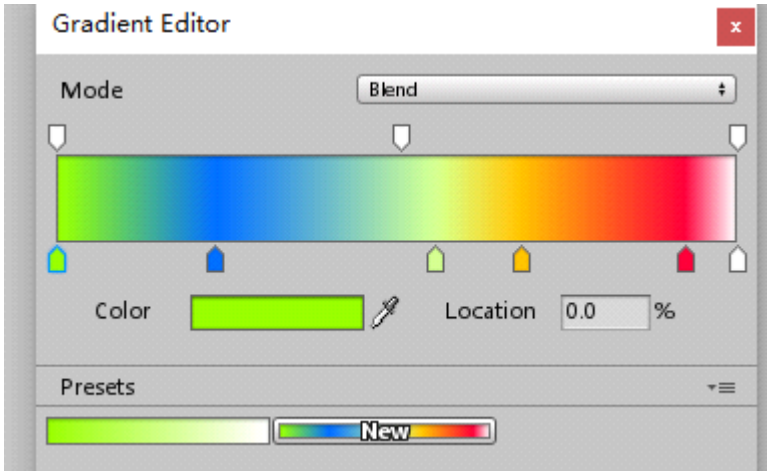
通过粒子的生命周期 利用曲线修改粒子颜色

也可以通过粒子的移动速度周期

选中 color over lifetime 模块勾选使用



点击颜色区域弹出颜色修改对话框



其中上方三角标号用来设置当前颜色的变化 如阿尔法值 深度

下方的三角标号用来设置新的颜色 自带渐变

这样的设置 让粒子在生命周期按以下颜色进行变化

绿 - 蓝 - 绿 - 黄 - 红 - 白)

〇五 实例粒子特效开发

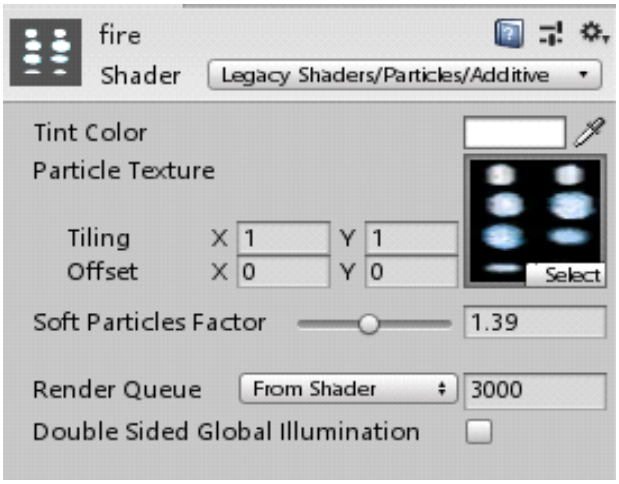
2019年8月12日 9:25

利用飞机喷火 冒烟 开火等特效制作综合案例

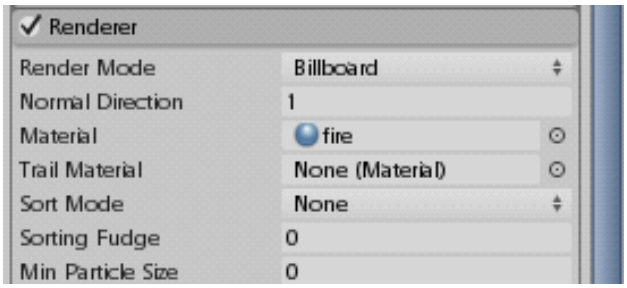
在基本的粒子系统中替换粒子材质

这里替换成动画帧 操作步骤如下

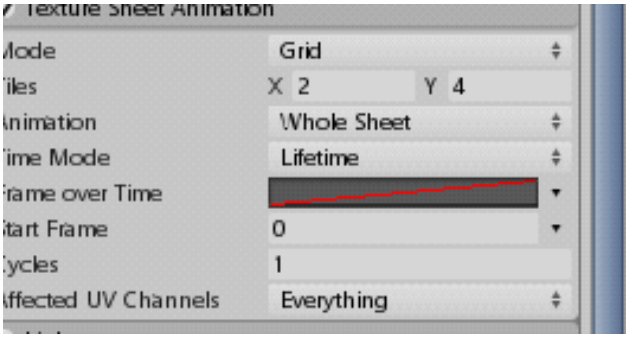
- 设置一个空材质 并将模式设置为 Particle - Activity
- 设置材质贴图为指定序列帧(单张图片)
- 设置背景颜色为白色 (消除原贴图背景) 若需要调成黑色 则需要设置材质为 alpha blended 模式



- 设置粒子的render 的材质为 上面设置好的材质



- 启动Texture Shader Animation 模块
- 设置材质动画帧的 ties XY 由贴图序列帧排列决定
- 同时可以设置在序列帧中随机帧开始播放 start frame 设置为 between即可



如何解决多粒子特效之间互相遮挡

- 如果需要层级 则修改 render模块中的 sorting fudge 值越大越先渲染(就在越底层)

○一 着色器概念 简介 种类

2019年8月11日 21:06

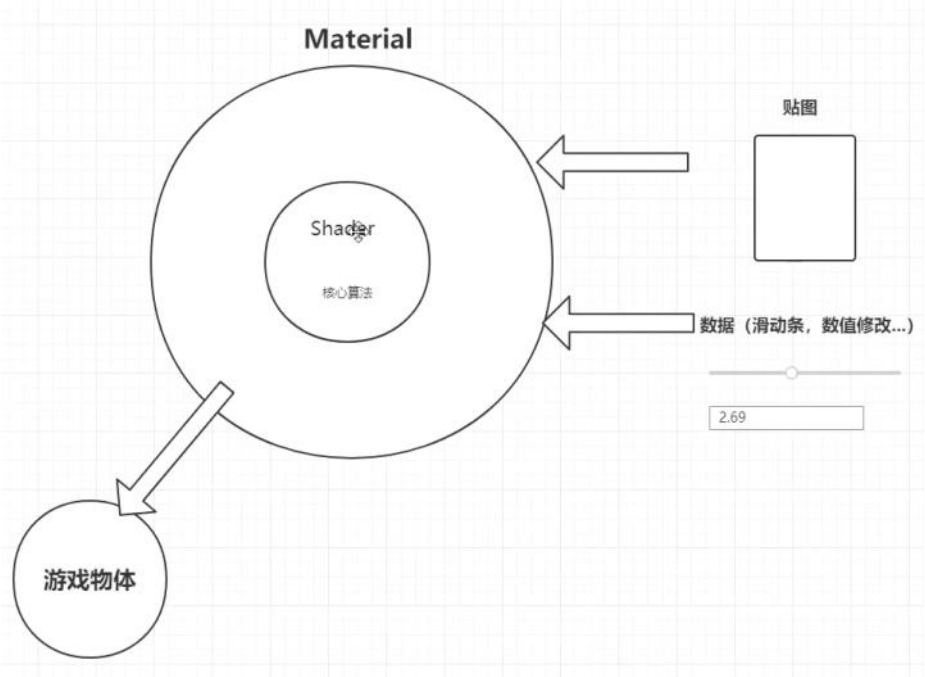
Shader 着色器的概念

计算机图形学领域中，着色器（英语：shader）是一种计算机程序，原本用于进行图像的浓淡处理（计算图像中的光照、亮度、颜色等），但近来，它也被用于完成很多不同领域的工作，比如处理CG特效、进行与浓淡处理无关的影片后期处理、甚至用于一些与计算机图形学无关的其它领域。

使用着色器在图形硬件上计算渲染效果有很高的自由度。尽管不是硬性要求，但目前大多数着色器是针对GPU开发的。GPU的可编程绘图管线已经全面取代传统的固定管线，可以使用着色器语言对其编程。构成最终图像的像素、顶点、纹理，它们的位置、色相、饱和度、亮度、对比度也都可以利用着色器中定义的算法进行动态调整。调用着色器的外部程序，也可以利用它向着色器提供的外部变量、纹理来修改这些着色器中的参数。除了普通的光照模型，着色器还可以调整图像的色相、饱和度、亮度、对比度，生成模糊、高光、有体积光源、失焦、卡通渲染、色调分离、畸变、凹凸贴图、色键（即所谓的蓝幕、绿幕抠像效果）、边缘检测等效果。

着色器的工作内容和方式

着色器是一种算法综合 是计算和像素输出的核心
它由贴图 数据 法线 坐标 等信息 利用算法和函数 输出具体像素点
并赋值给材质 材质决定游戏物体的外观 (形状有网格确定)



着色器的种类

二维着色器

二维着色器处理的是数字图像，也叫纹理，着色器可以修改它们的像素。二维着色器也可以参与三维图形的

渲染。目前只有“像素着色器”一种二维着色器。

像素着色器（英语：pixel shader）

也叫片段着色器（英语：fragment shader），用于计算“片段”的颜色和其它属性，此处的“片段”通常是指单独的像素。最简单的像素着色器只有输出颜色值；复杂的像素着色器可以有多个输入输出[2]。像素着色器既可以永远输出同一个颜色，也可以考虑光照、做凹凸贴图、生成阴影和高光，还可以实现半透明等效果。像素着色器还可以修改片段的深度，也可以为多个渲染目标输出多个颜色。

三维图形学中，单独一个像素着色器并不能实现非常复杂的效果，因为它只能处理单独的像素，没有场景中其它几何体的信息。不过，像素着色器有屏幕坐标信息，如果将屏幕上的内容作为纹理传入，它就可以对当前像素附近的像素进行采样。利用这种方法，可以实现大量二维后期特效，例如模糊和边缘检测。

像素着色器还可以处理管线中间过程中的任何二维图像，包括精灵和纹理。因此，如果需要在栅格化后进行后期处理，像素着色器是唯一选择。

三维着色器

三维着色器处理的是三维模型或者其它几何体，可以访问用来绘制模型的颜色和纹理。顶点着色器是最早的三维着色器；几何着色器可以在着色器中生成新的顶点；细分曲面着色器（英语：tessellation shader）则可以向一组顶点中添加细节。

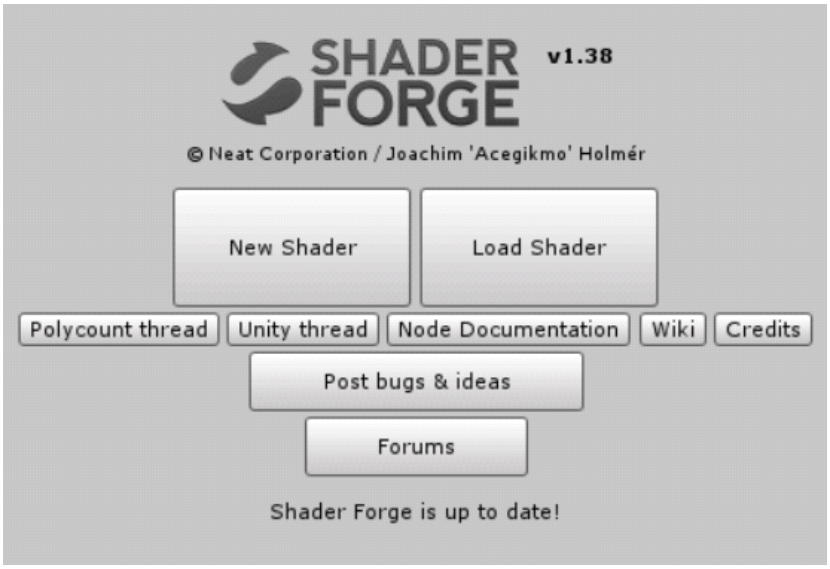
顶点着色器处理每个顶点，将顶点的空间位置投影在屏幕上，即计算顶点的二维坐标。同时，它也负责顶点的深度缓冲（Z-Buffer）的计算。顶点着色器可以掌控顶点的位置、颜色和纹理坐标等属性，但无法生成新的顶点。顶点着色器的输出传递到流水线的下一步。如果有之后定义了几何着色器，则几何着色器会处理顶点着色器的输出数据，否则，光栅化器继续流水线任务。
几何着色器可以从多边形网格中增删顶点。它能够执行对CPU来说过于繁重的生成几何结构和增加模型细节的工作。Direct3D版本10增加了支持几何着色器的API, 成为Shader Model 4.0的组成部分。OpenGL只可通过它的一个插件来使用几何着色器，但极有可能在3.1版本中该功能将会归并。几何着色器的输出连接光栅化器的输入。

○二 Shader Forge 可视化着色器插件

2019年8月11日 22:22

强大的可视化着色器编写插件 Shader Forge

在Window/Shader Forge 打开编辑界面



Shader Forge Editor 使用面板

<---- 操作 shader

<---- 说明文档

<---- 反馈

几种常用shader创建解释

无光照反射 有光照反射 基本型 自定义型 2D类型 粒子特效(发光 沙尘 烟雾) 天空 屏幕效果

编辑面板基本使用介绍

面板使用快捷键与详情

如果 勾选了 auto update 预览材质球会在每次修改后进行更新
但是这样更容易因为自己的错误操作导致不可回复的后果 一般关闭 等确认无误之后进行更新

打开节点面板之后 会在右侧显示所有节点 或者按住键盘的某个键 可以打开当前键开头的节点 和鼠标右键一样(带分类)
绿色的节点是属性节点 可以显示在shader上的 同时可以进行修改

其他快捷键与使用说明

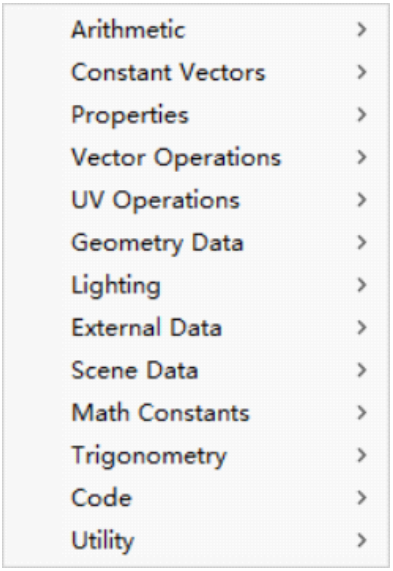
- ALT + 右键 --- 滑动可以切除节点与面板之间的关联
- ALT + 左键 --- 选中 或 框选部分节点 同时进行操作
- Ctrl +左键 --- 选中当前节点作为父亲节点 其余有关联节点跟随移动

其他设置在基础的部分时候保持默认即可

〇三 shader节点分类与说明

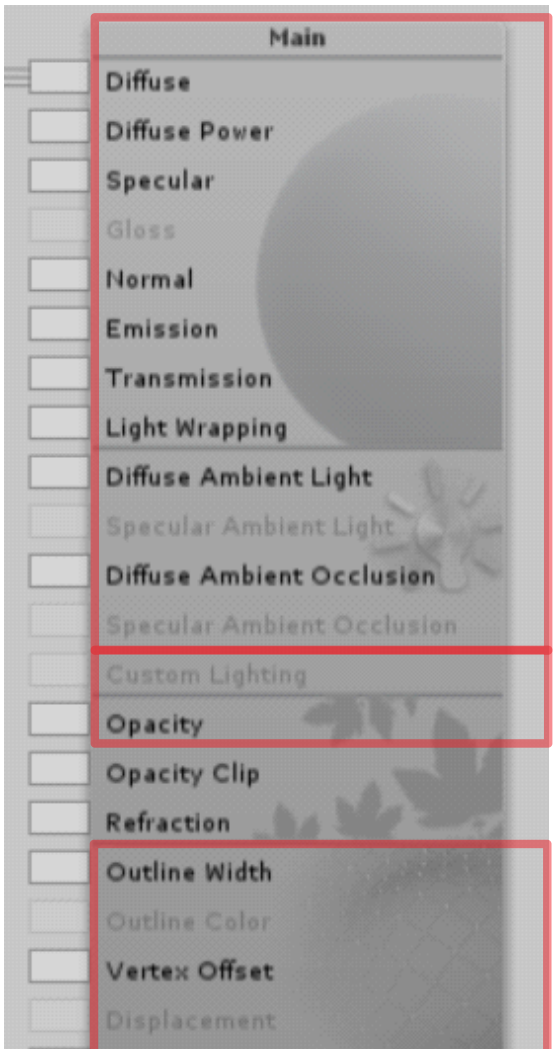
2019年8月12日 12:54

基本的属性与分类说明



1. 基本的数学公式
2. 向量与常量 vector(v2 v4 矩阵)
3. 基本shader属性常量(贴图 颜色)
4. 向量操作(计算向量长度 距离 投影 单位化)
5. UV坐标 操作
6. 模型信息(UV 法线 网格 顶点)
7. 灯光信息(颜色 方向 位置)
8. 外部数据节点(屏幕 时间 世界坐标)
9. 场景信息(景深 颜色 阴影)
10. 数学常量(e π)
11. 三角函数(con sin)
12. 代码
13. 工具(整理 排序)

通道面板与通道介绍



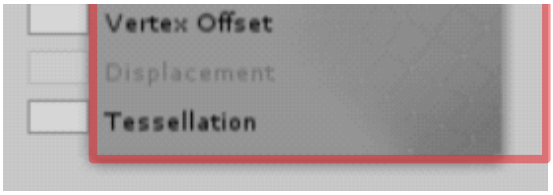
通道面板

颜色通道

透明度

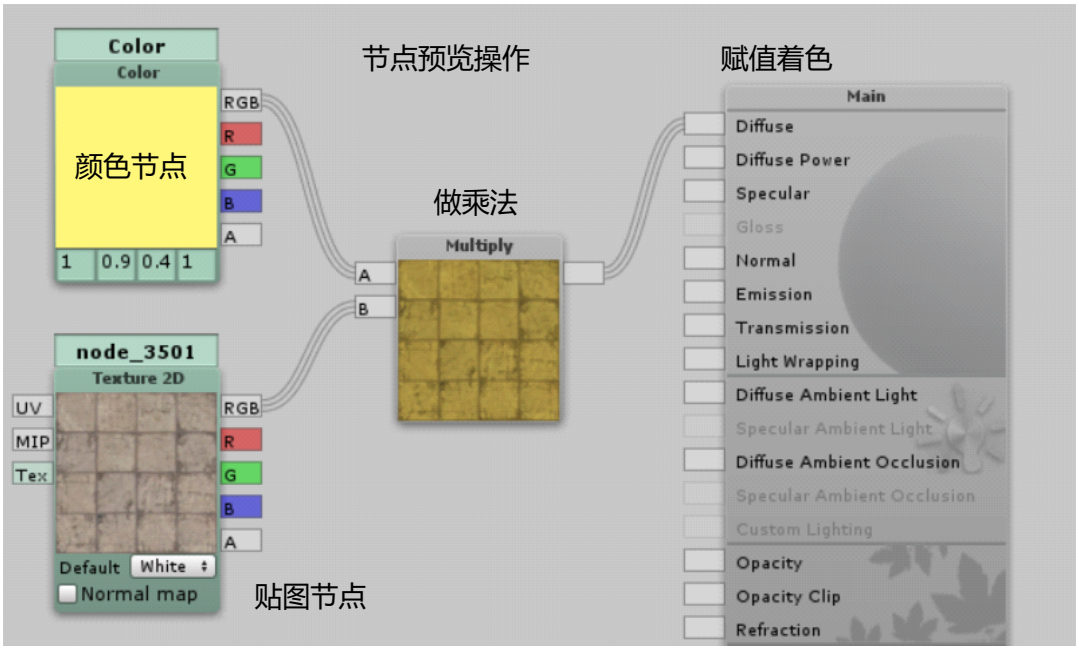
描边

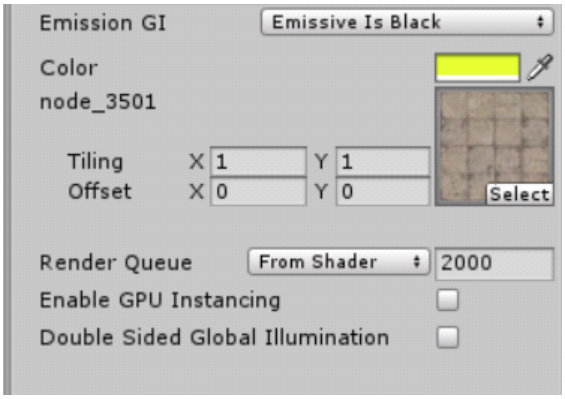
位置坐标



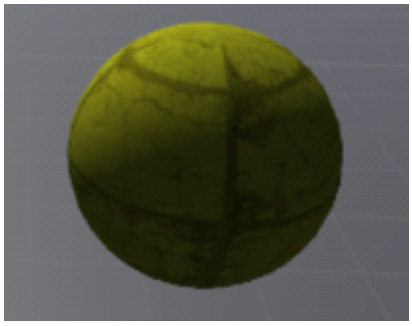
节点的融合说明

节点融合只是预览结果 最终的效果需要重新在shader属性进行设置



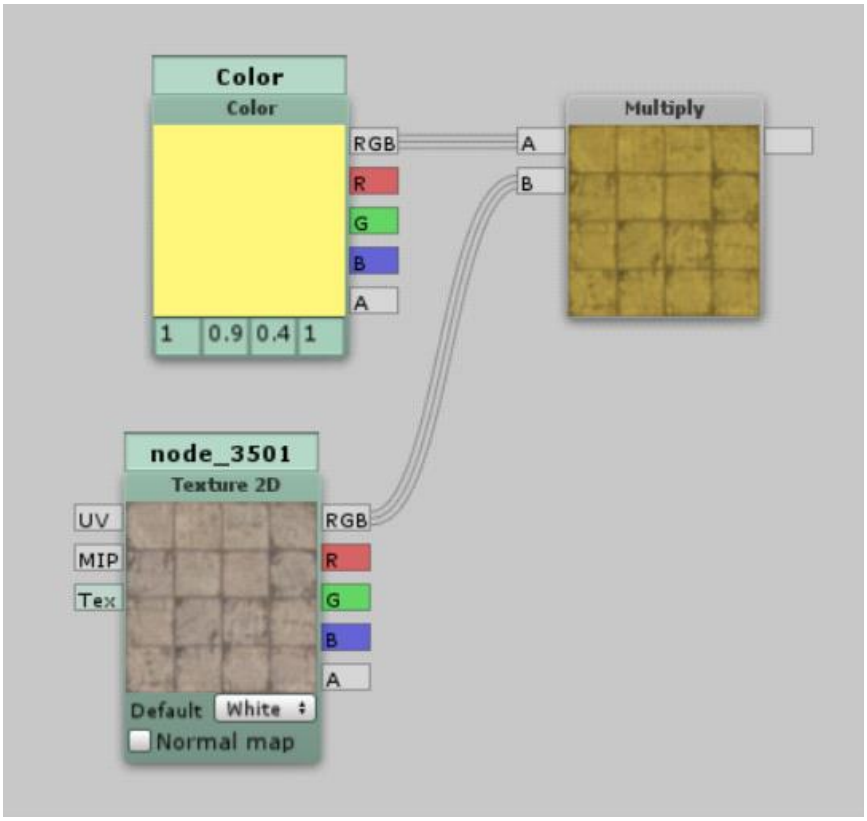


最终附在 Material 的 shader 在物体的表现效果



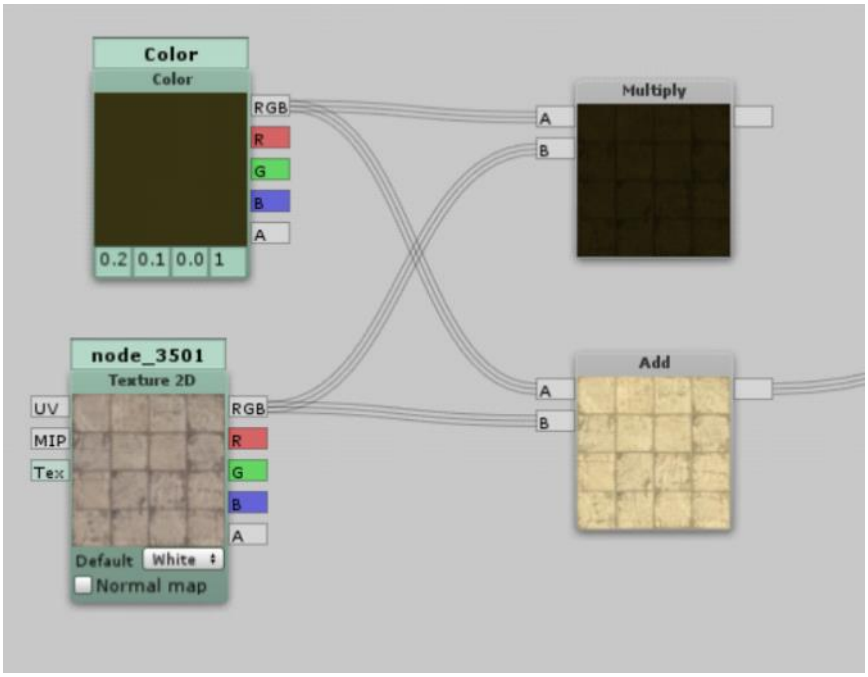
乘法节点说明

乘法multiply 对颜色(RGBA) 范围在0-1 的乘法 (白色= 111; 黑色= 000)
白 * 黑 = 黑 (0*1 =0) 白 * 绿 = 绿 (1* x = x) 黑 * 红 = 黑 (0 * y = 0)
multiply 最多乘五个 多个可以乘了之后乘积再乘 并且颜色越来越深



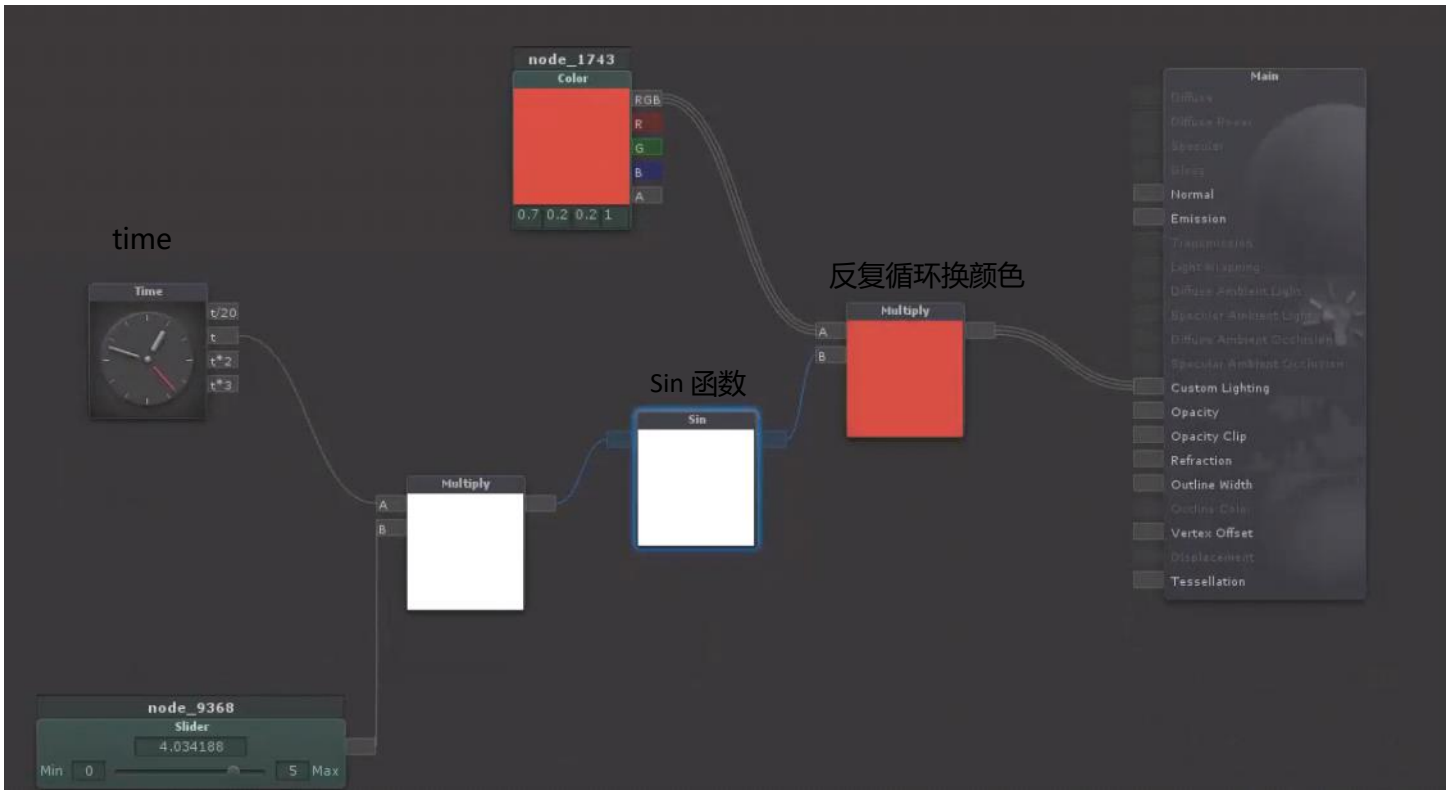
加法节点说明

加法类似 乘法 但是数值会更大 效果越深 一般作用于光亮和发光
与乘法相比较如下:



time节点

节点的输出会一直增加



〇四 数据维度转换

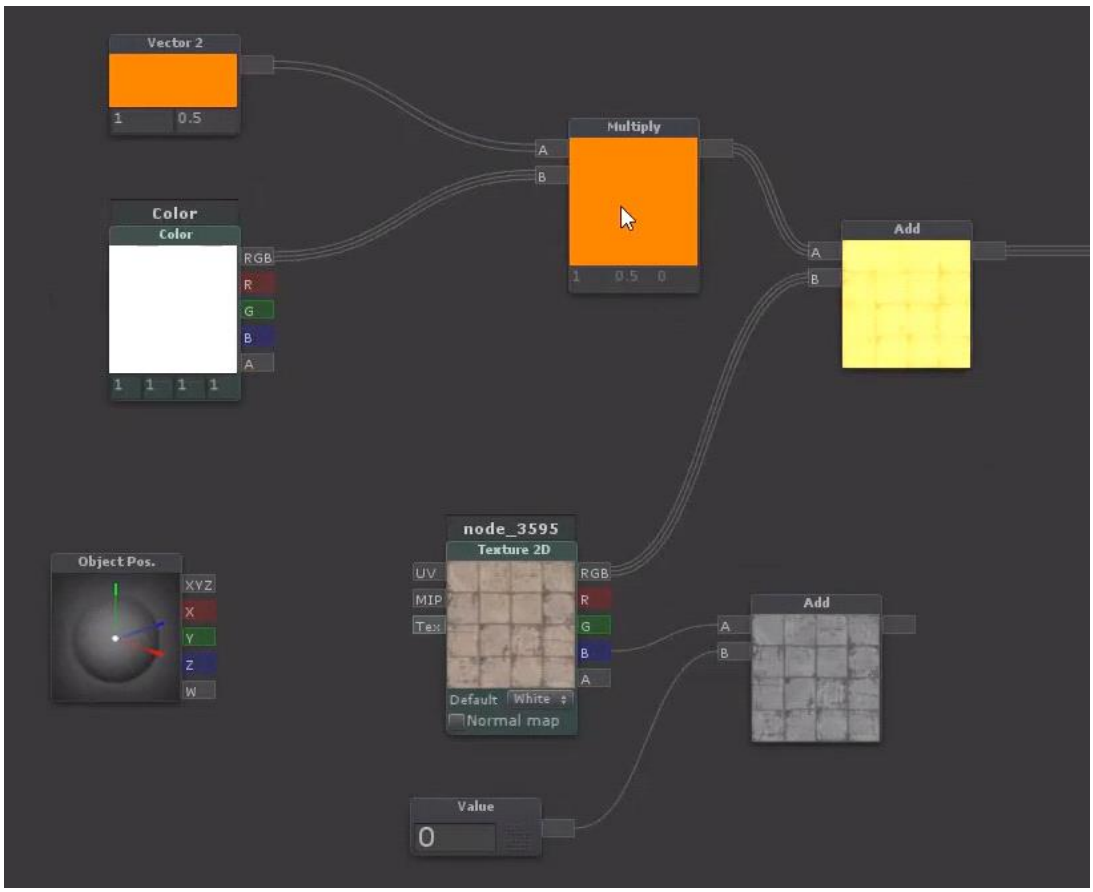
2019年8月13日 12:33

多维度不同数据的组合与转换

一维信息 : value R G B
二维信息 : 2D坐标
三维信息 : 位置 贴图信息(三维数组)
四维信息 : 颜色 旋转(四元数)

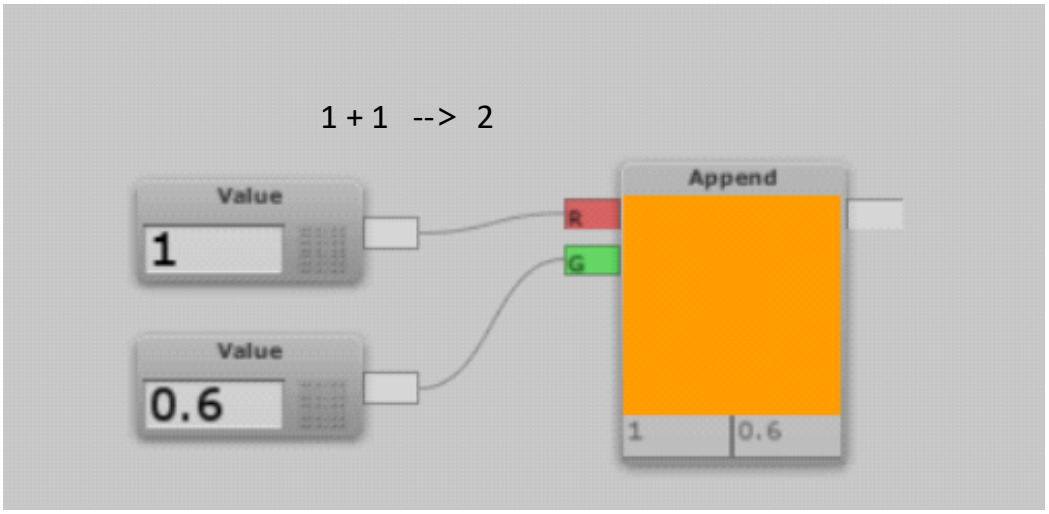
基本的节点本身不单单是某个节点属性 也会包含很多维度数据 并且可以相互进行组合操作

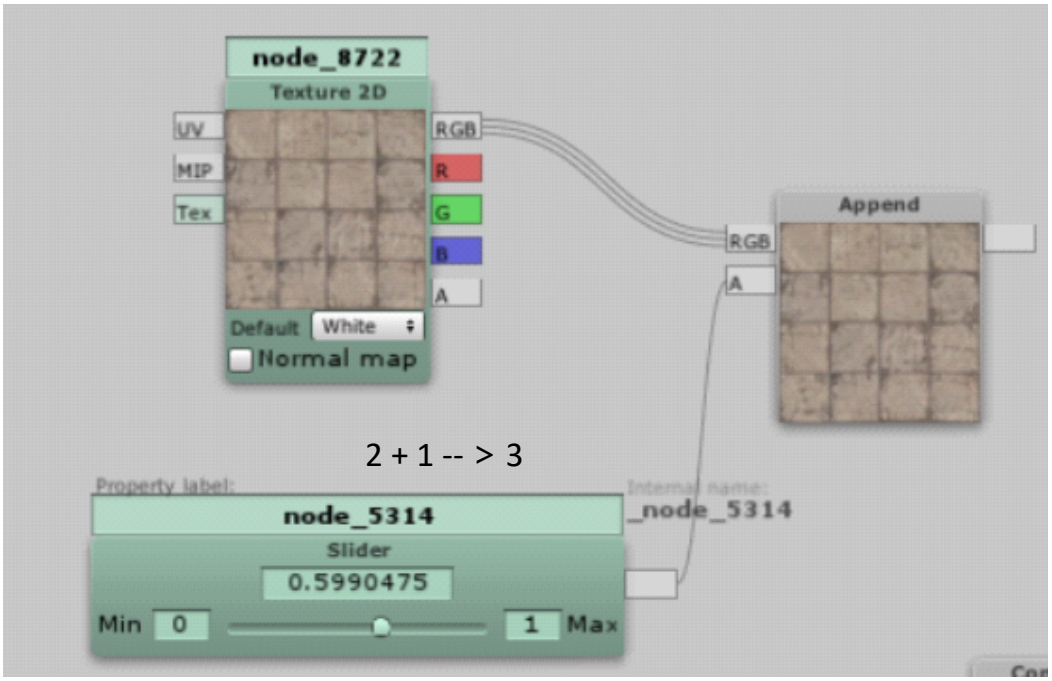
单个color 本身是一个四维信息(RGBA) 但是进行计算或者转换的时候
每一个基本单元的信息 都可以作为操作输入端并进行组合
获取定的操作维度信息 通过不同的维度信息进行组合 可以相互转换



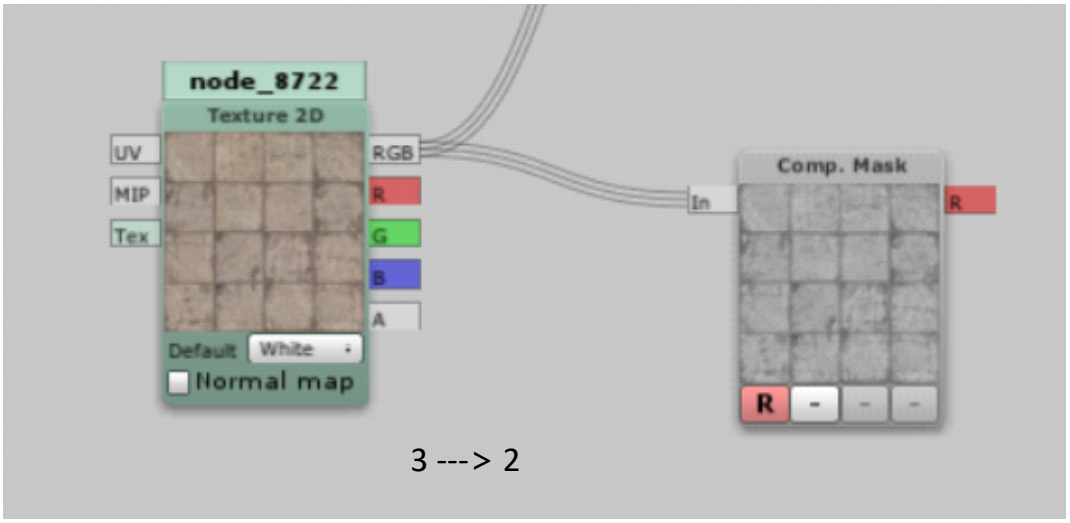
Append节点和Comp.Mask节点

Append为升维节点 可以增加通道





Comp.Mask相反 减少一个输入 通道并输出

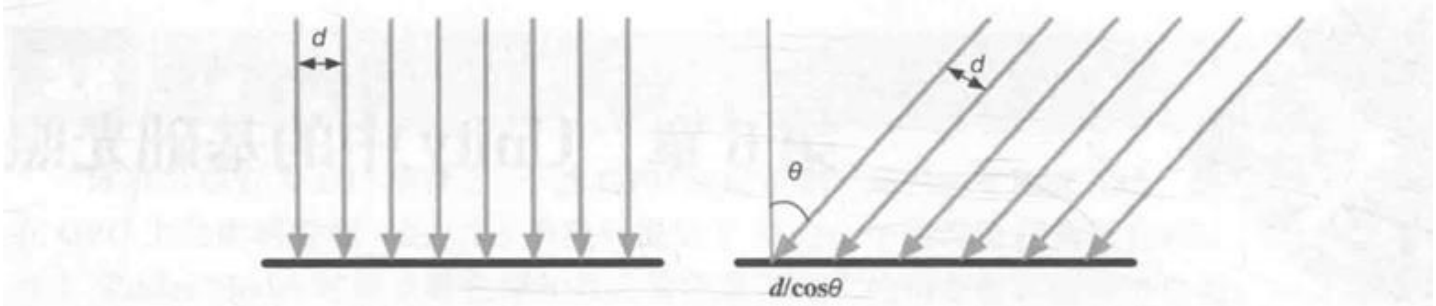


〇五 基础光照

2019年8月13日 12:43

Unity中的基础光源

光源：通常将光源当作一个没有体积的点，用 l 表示光源的方向；
在光学中，通常使用辐照度（irradiance）来量化光；
对于平行光而言，其辐照度可通过计算垂直于l 的单位面积上单位时间内穿过的能量来得到;
如果物体表面与光照方向不垂直，可以使用光源方向l和表面法线n之间夹角的余弦值得到（注意：默认方向矢量的模均为1）；



在左图中，光是垂直照射到物体表面，因此光线之间的垂直距离保持不变；而在右图中，光是斜着照射到物体表面，在物体表面光线之间的距离是 d/cosθ，因此单位面积上接收到的光线数目要少于左图

〇六 光照模型

2019年8月13日 12:51

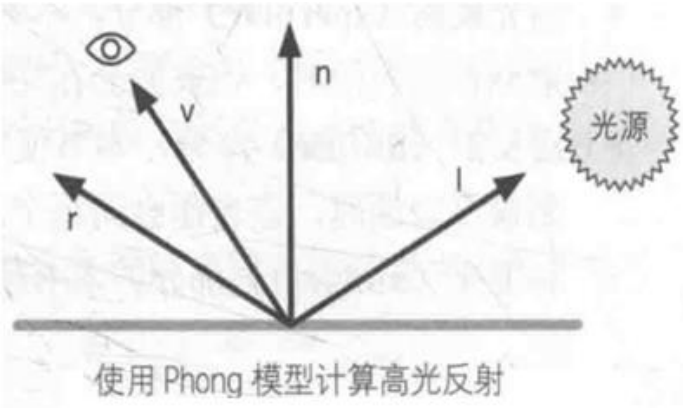
光照模型的定义

着色（shading）：是指根据材质属性(如漫反射属性等)、光源信息(如光源方向、辐照度等)
使用相应公式去计算沿着某个观察方向的出射度的过程，我们将该公式称之为“光照模型(Lighting Model)”；

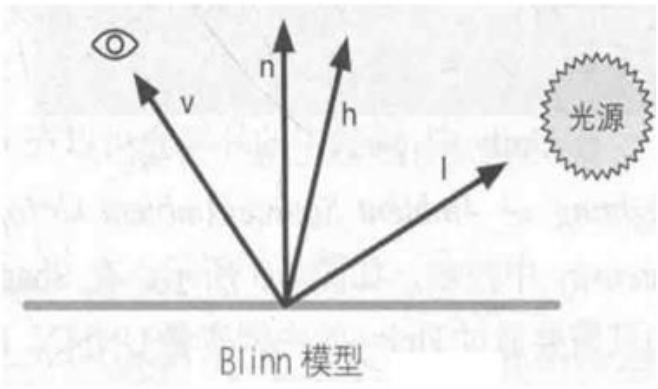
其基本方法是把进入到摄像机内的光线分为4个部分，每个部分使用一种方法来计算它的贡献度：

- 自发光部分（emissive）
- 高光反射部分（specular）
- 漫反射部分（diffuse）
- 环境光部分（ambient）

标准光照模型



使用 Phong 模型计算高光反射



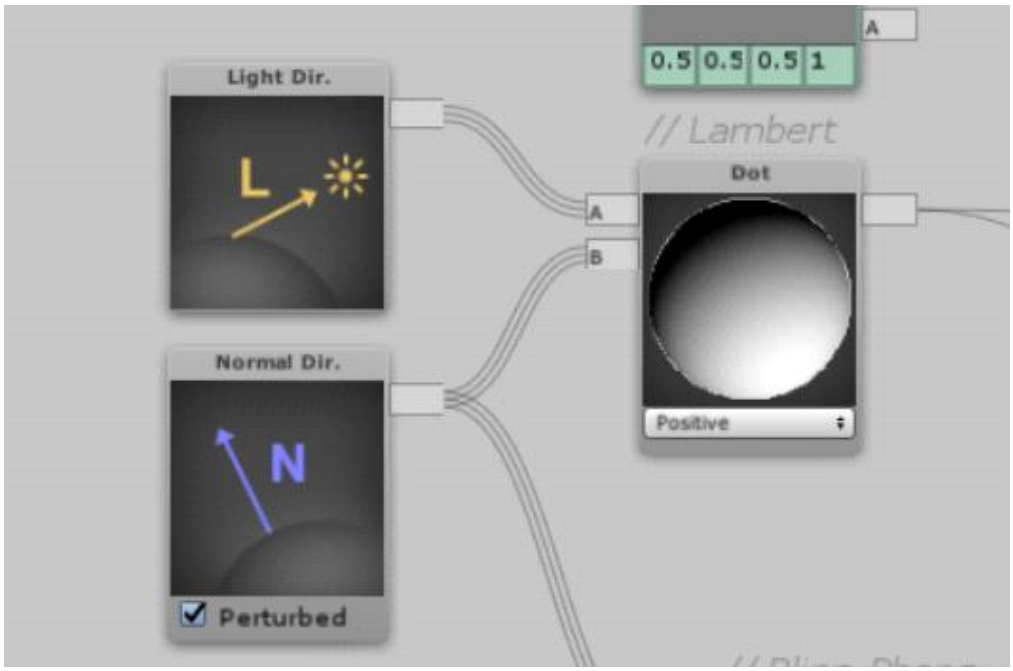
Blinn 模型

〇七 漫反射 高光 法线 贴图

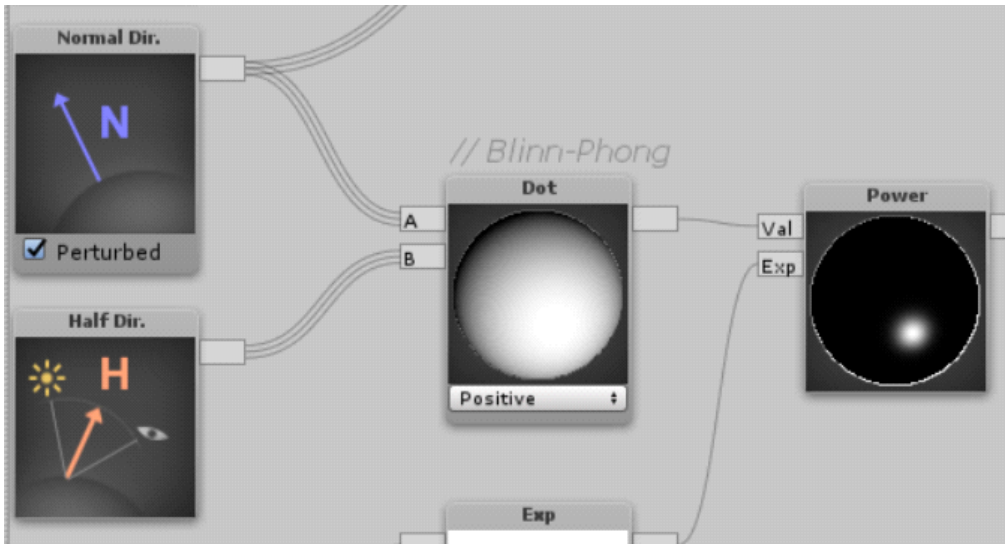
2019年8月14日 21:32

漫反射制作

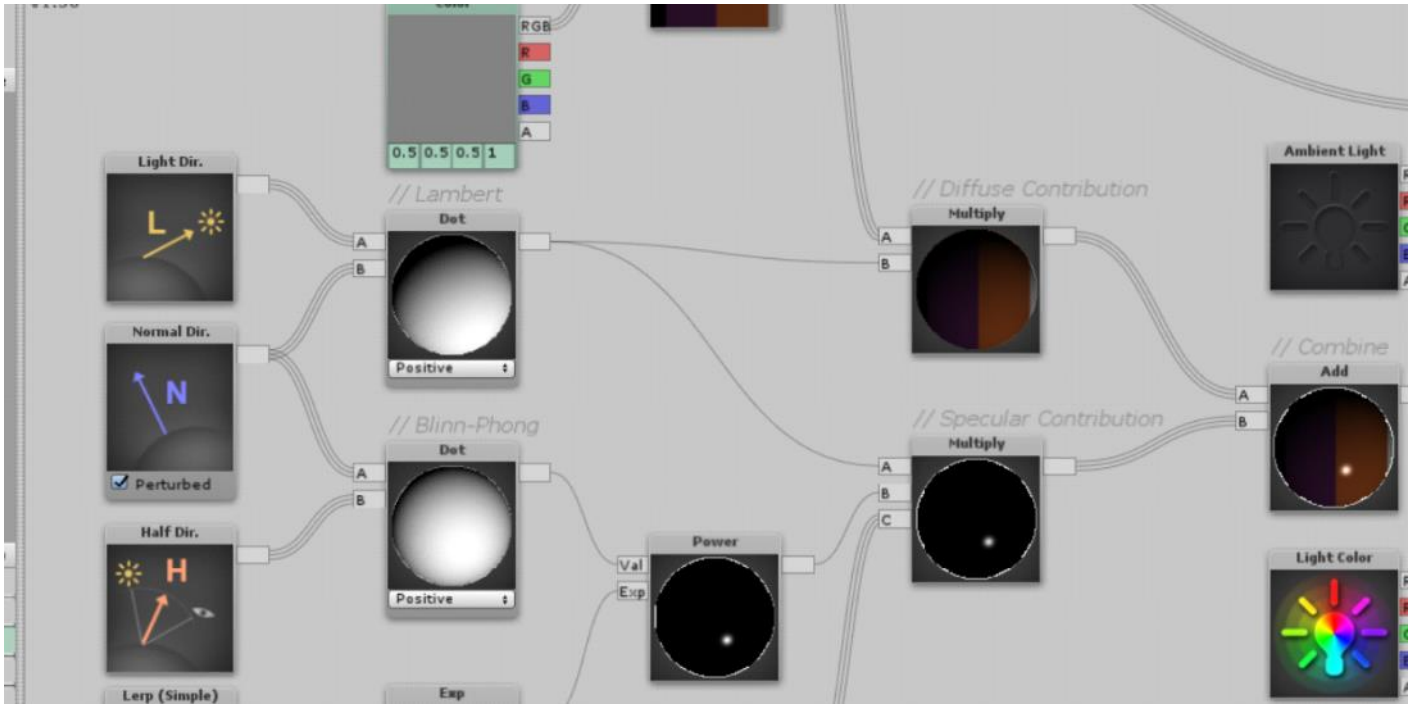
漫反射 = 反射 + 法线



高光 = 半角 + 法线 + power = 高光



两者合二为一


RGB
R
G
B
A
0.5 0.5 0.5 1
Light Dir.
L
Normal Dir.
N
Perturbed
Half Dir.
H
Lerp (Simple)
// Lambert
Dot
A
B
Positive
// Blinn-Phong
Dot
A
B
Positive
Exp
Power
Val
Exp
// Diffuse Contribution
Multiply
A
B
// Specular Contribution
Multiply
A
B
C
Ambient Light
// Combine
Add
A
B
Light Color

〇八 菲涅尔反射

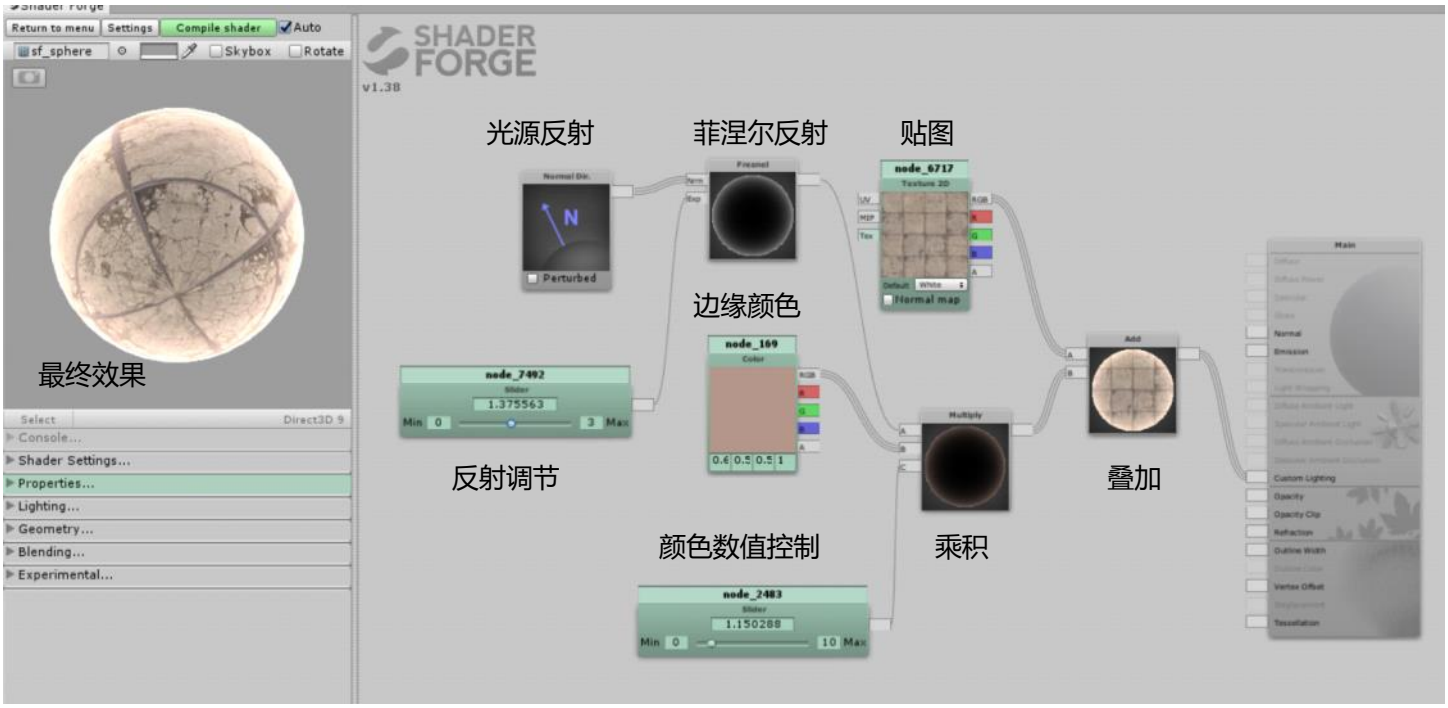
2019年8月14日 21:32

菲涅尔反射定义

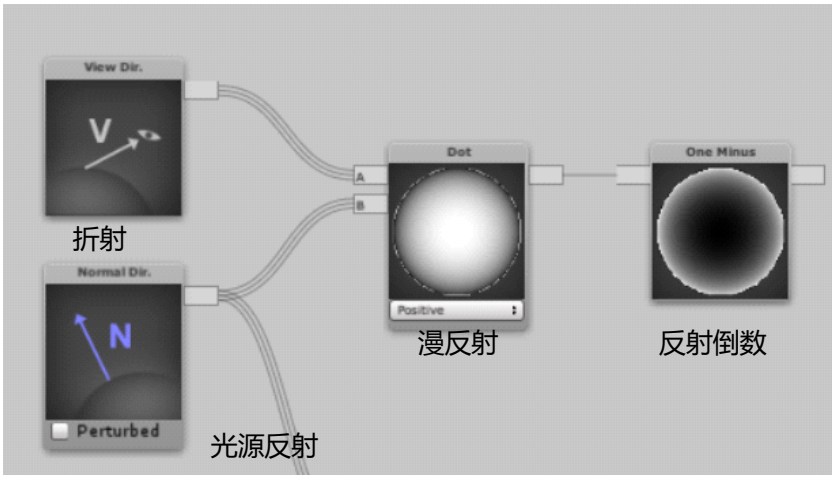
菲涅耳公式（或菲涅耳方程），由奥古斯丁·让·菲涅耳导出。用来描述光在不同折射率的介质之间的行为。由公式推导出的光的反射称之为“菲涅尔反射”。菲涅尔公式是光学中的重要公式，用它能解释反射光的强度、折射光的强度、相位与入射光的强度的关系

fresnel = fresnel基础值 + fresnel缩放量*pow(1 - dot(N, V), 5)

在Shader Forge实现



实现原理



Shader 实现

```
Shader "lijia/fresnelTest"
{
```

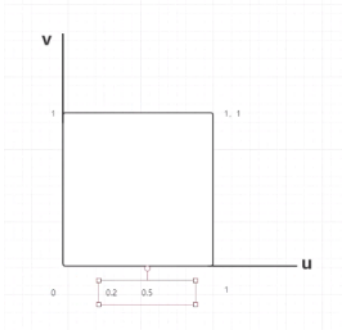
```
Properties
{
    _MainTex ("Texture", 2D) = "white" {}
    _fresnelBase("fresnelBase", Range(0, 1)) = 1
    _fresnelScale("fresnelScale", Range(0, 1)) = 1
    _fresnelIndensity("fresnelIndensity", Range(0, 5)) = 5
    _fresnelCol("_fresnelCol", Color) = (1,1,1,1)
}

SubShader
{
    Tags { "RenderType"="Opaque" }
    LOD 100

    Pass
    {
        tags{"lightmode="="forward"}

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"
        #include "Lighting.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float2 uv : TEXCOORD0;
            float3 normal : NORMAL;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
            float3 L : TEXCOORD1;
            float3 N : TEXCOORD2;
            float3 V : TEXCOORD3;
        };

        sampler2D _MainTex;
        float4 _MainTex_ST;
        float _fresnelBase;
```

```
        float _fresnelScale;
        float _fresnelIndensity;
        float4 _fresnelCol;

        v2f vert (appdata v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = TRANSFORM_TEX(v.uv, _MainTex);
            //将法线转到世界坐标
            o.N = mul(v.normal, (float3x3)unity_WorldToObject);
            //获取世界坐标的光向量
            o.L = WorldSpaceLightDir(v.vertex);
            //获取世界坐标的视角向量
            o.V = WorldSpaceViewDir(v.vertex);
            return o;
        }

        fixed4 frag (v2f i) : SV_Target
        {
            fixed4 col = tex2D(_MainTex, i.uv);

            float3 N = normalize(i.N);
            float3 L = normalize(i.L);
            float3 V = normalize(i.V);

            col.rgb *= saturate(dot(N, L)) * _LightColor0.rgb;
            //菲尼尔公式
            float fresnel = _fresnelBase + _fresnelScale*pow(1 - dot(N, V), _fresnelIndensity);
            col.rgb += lerp(col.rgb, _fresnelCol, fresnel) * _fresnelCol.a;
            return col;
        }
        ENDCG
    }
  }
}
```

〇九 UV坐标和节点

2019年8月14日 22:00

UV坐标节点

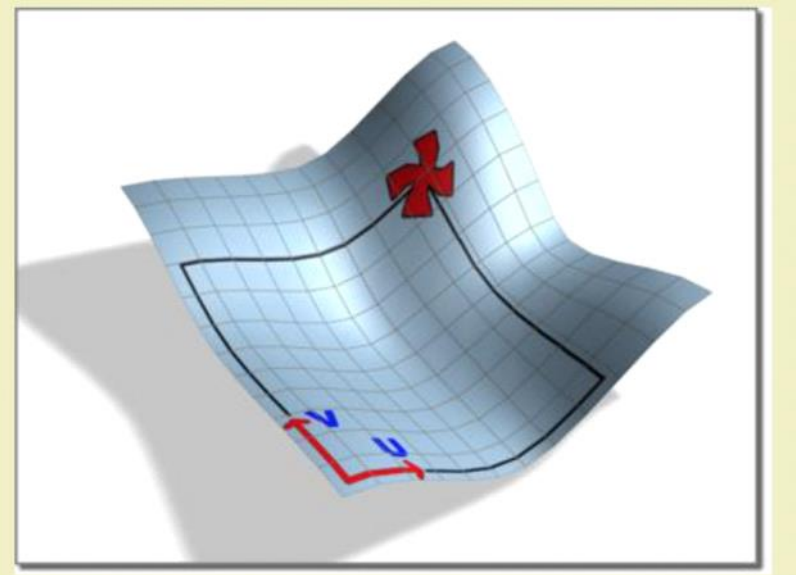


上图所示 uv坐标对应的就是坐标轴上的位置和坐标，而map指的就是我们的贴图，正常情况下map和uv是对应的1比1的关系，
但是我们希望只采样纹理图的某一部分，这时候我们就应该要修改我们的纹理坐标,做一些数学运算，来达到不同的效果！

对于三维模型，有两个最重要的坐标系统，一是顶点的位置（X，Y，Z）坐标，另一个就是UV坐标。什么是UV？简单的说，就是贴图影射到模型表面的依据。 完整的说，其实应该是UVW（因为XYZ已经用过了，所以另选三个字母表示）。U和V分别是图片在显示器水平、垂直方向上的坐标，取值一般都是0~1，也 就是（水平方向的第U个像素/图片宽度，垂直方向的第V个像素/图片高度）。那W呢？贴图是二维的，何来三个坐标？嗯嗯，W的方向垂直于显示器表面，一般 用于程序贴图或者某些3D贴图技术（记住，确实有三维贴图这种概念！），对于游戏而言不常用到，所以一般我们就简称UV了。

所有的图象文件都是二维的一个平面。水平方向是U，垂直方向是V，通过这个平面的，二维的UV坐标系。我们可以定位图象上的任意一个象素。但是一个问题是如何把这个二维的平面贴到三维的NURBS表面和多边形表面呢？ 对于NURBS表面。由于他本身具有UV参数，尽管这个UV值是用来定位表面上的点的参数，但由于它也是二维的，所以很容易通过换算把表面上的点和平面图象上的象素对应起来。所以把图象贴带NURBS是很直接的一件事。但是对于多变形模型来讲，贴图就变成一件麻烦的事了。所以多边形为了贴图就额外引进了一个UV坐标，以便把多边形的顶点和图象文件上的象素对应起来，这样才能在多边形表面上定位纹理贴图。所以说多边形的顶点除了具有三维的空间坐标外。还具有二维的UV坐标。

UV” 这里是指u,v纹理贴图坐标的简称(它和空间模型的X, Y, Z轴是类似的). 它定义了图片上每个点的位置的信息. 这些点与3D模型是相互联系的, 以决定表面纹理贴图的位置. UV就是将图像上每一个点精确对应到模型物体的表面. 在点与点之间的间隙位置由软件进行图像光滑插值处理. 这就是所谓的UV贴图.
那为什么用UV坐标而不是标准的投影坐标呢? 通常给物体纹理贴图最标准的方法就是以planar(平面),cylindrical(圆柱), spherical(球形),cubic(方盒)坐标方式投影贴图.
Planar projection(平面投影方式)是将图像沿x,y或z轴直接投影到物体. 这种方法使用于纸张, 布告, 书的封面等 - 也就是表面平整的物体.平面投影的缺点是如果表面不平整, 或者物体边缘弯曲, 就会产生如图A的不理想接缝和变形. 避免这种情况需要创建带有alpha通道的图像, 来掩盖临近的平面投影接缝, 而这会是非常烦琐的工作. 所以不要对有较大厚度的物体和不平整的表面运用平面投影方式. 对于立方体可以在x, y方向分别进行平面投影, 但是要注意边缘接缝的融合. 或者采用无缝连续的纹理, 并使用cubic投影方式. 多数软件有图片自动缩放功能, 使图像与表面吻合. 显然, 如果你的图像与表面形状不同, 自动缩放就会改变图像的比例以吻合表面. 这通常会产生不理想的效果, 所以制作贴图前先测量你的物体尺寸.



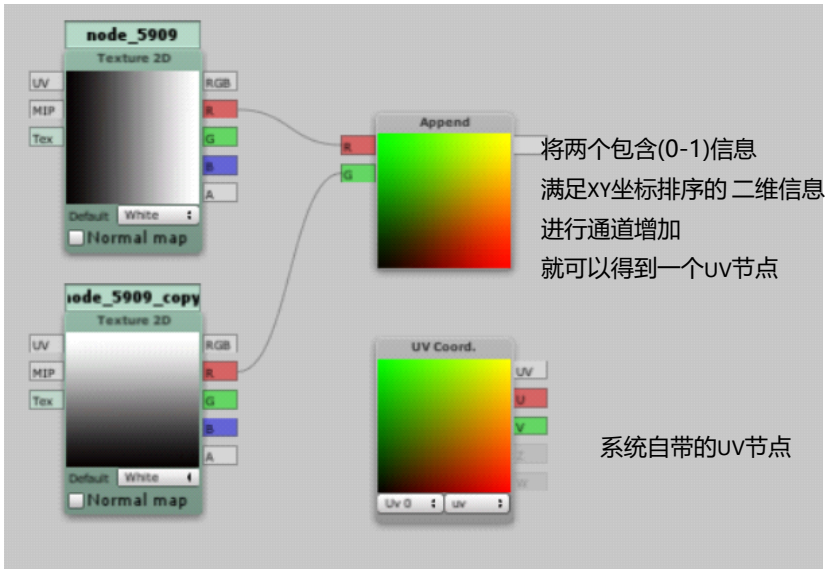
曲面上显示的局部 **UV** 坐标

大多数材质贴图都是为 3D 曲面指定的 2D 平面。因此，说明贴图位置和变形时所用的坐标系与 3D 空间中使用的 X、Y 和 Z 轴坐标不同。特别是，贴图坐标使用的是字母 U、V 和 W；在字母表中，这三个字母位于 X、Y 和 Z 之前。

U、V 和 W 坐标分别与 X、Y 和 Z 坐标的相关方向平行。如果查看 2D 贴图图像，U 相当于 X，代表着该贴图的水平方向。V 相当于 Y，代表着该贴图的竖直方向。W 相当于 Z，代表着与该贴图的 UV 平面垂直的方向。

您可能会问，为什么 2D 平面需要象 W 这样的深度坐标。一个原因是，相对于贴图的几何体对该贴图的方向进行翻转时，这个坐标是很有用的。为了实现该操作，还需要第三个坐标。另外，W 坐标对三维程序材质的作用非同小可。

在 shader Forge 实现 UV节点

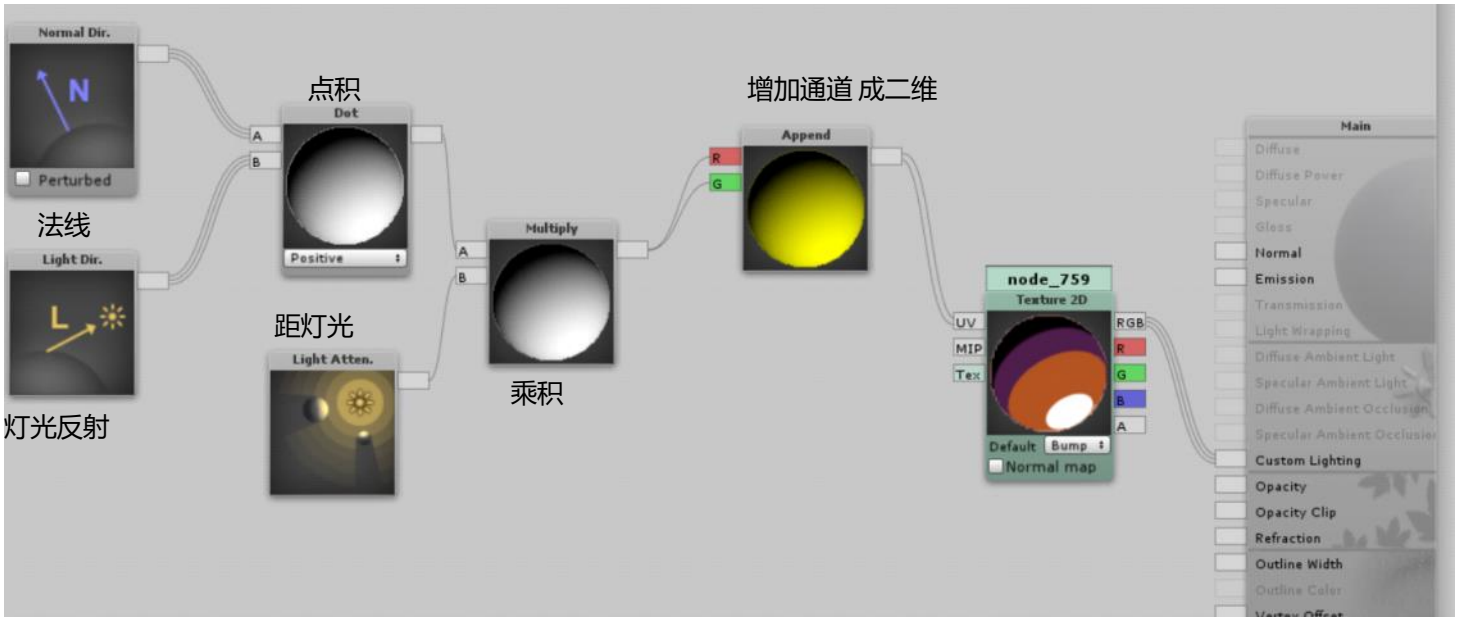


一十 卡通特效

2019年8月14日 22:21

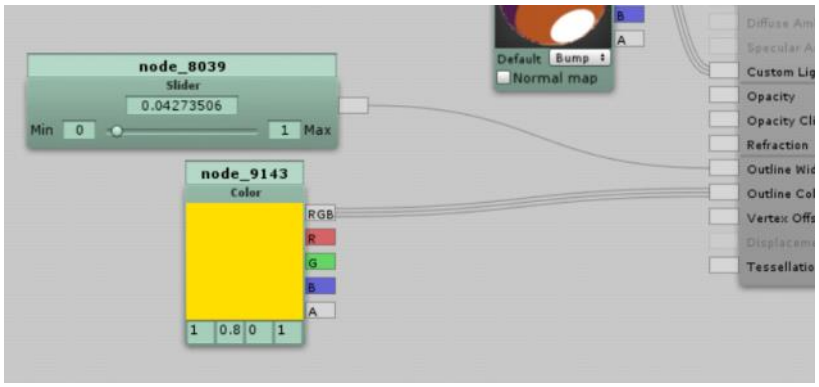
在 Shader Forge 实现卡通特效的材质

首先要实现漫反射 反射 + 法线(点积) = 漫反射



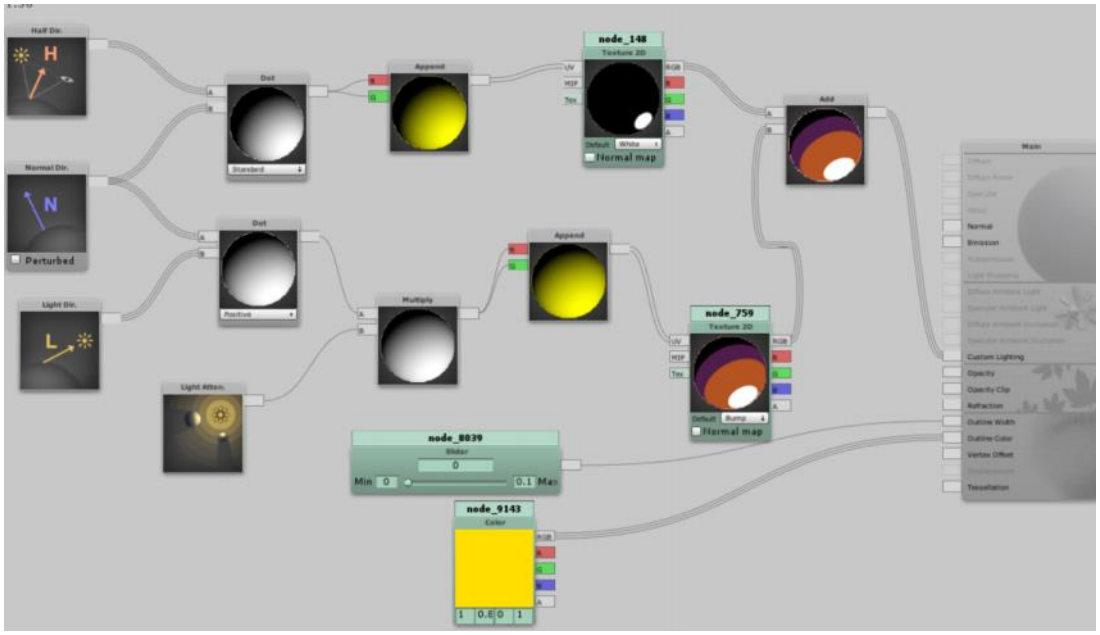
然后添加 通道 作为卡通贴图的输入 如上图

增加一个描边 一个颜色 一个颜色数值控制 连接到outline Color



再添加一个卡通高光

高光 = 半角 + 法线 (点积)



然后增加通道 并作为 高光贴图的输入

最后 卡通高光 与卡通反射进行叠加

十一 BRDF贴图

2019年8月15日 12:16

BRDF介绍

双向反射分布函数（Bidirectional Reflectance Distribution Function，BRDF）
用来定义给定入射方向上的辐射照度（irradiance）如何影响给定出射方向上的辐射率（radiance）。更笼统地说，它描述了入射光线经过某个表面反射后如何在各个出射方向上分布——这可以是从理想镜面反射到漫反射、各向同性（isotropic）或者各向异性（anisotropic）的各种反射。

BRDF（**Bidirectional Reflectance Distribution Function**，即双向反射分布函数）

$$BRDF = \frac{L_o}{E_i}$$

光线照到一个物体，首先产生了反射，吸收和透射，所以BRDF的关键因素即为多少光被反射、吸收和透射（折射）了，是怎样变化的。这时的反射多为漫反射。而要知道这些光线反射透射的变化就需要清楚三样东西，物体的表面材质、光线的波长（即它是什么样的光，是可见太阳光，节能灯光还是紫外线）和观察者与物体之间的位置关系。三维世界角度可以类似是球体的，光线角度除了纵向180°的变化，还有横向360的不同发散方向。会有相应的入射光，反射光，入射角和反射角，它们在物体表面的法平面和切平面上的关系成为了BRDF的关键参数。由于人类眼睛对光的特殊敏感性，我们之所以能看到物体都是通过光线在物体上的发射和转移实现的。而双向反射分布这样的函数表示可以更好地描述光线在物体上的变化，反射光线同时发向分布在法线两边的观察者和光源两个方向，从而使人在计算机等模拟环境下，视觉上可以看到更好的物体模拟效果，仿佛真实的物体存在。

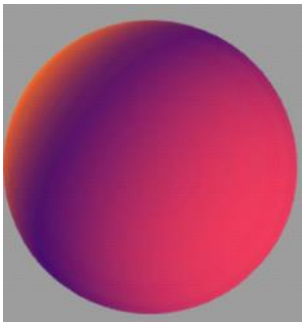
前传 直观理解推导Physically Based Rendering 的BRDF公式之微表面法线分布函数NDF

https://blog.csdn.net/xingzhe2001/article/details/83829705

BRDF里的法线分布函数D函数的各种算法于思想

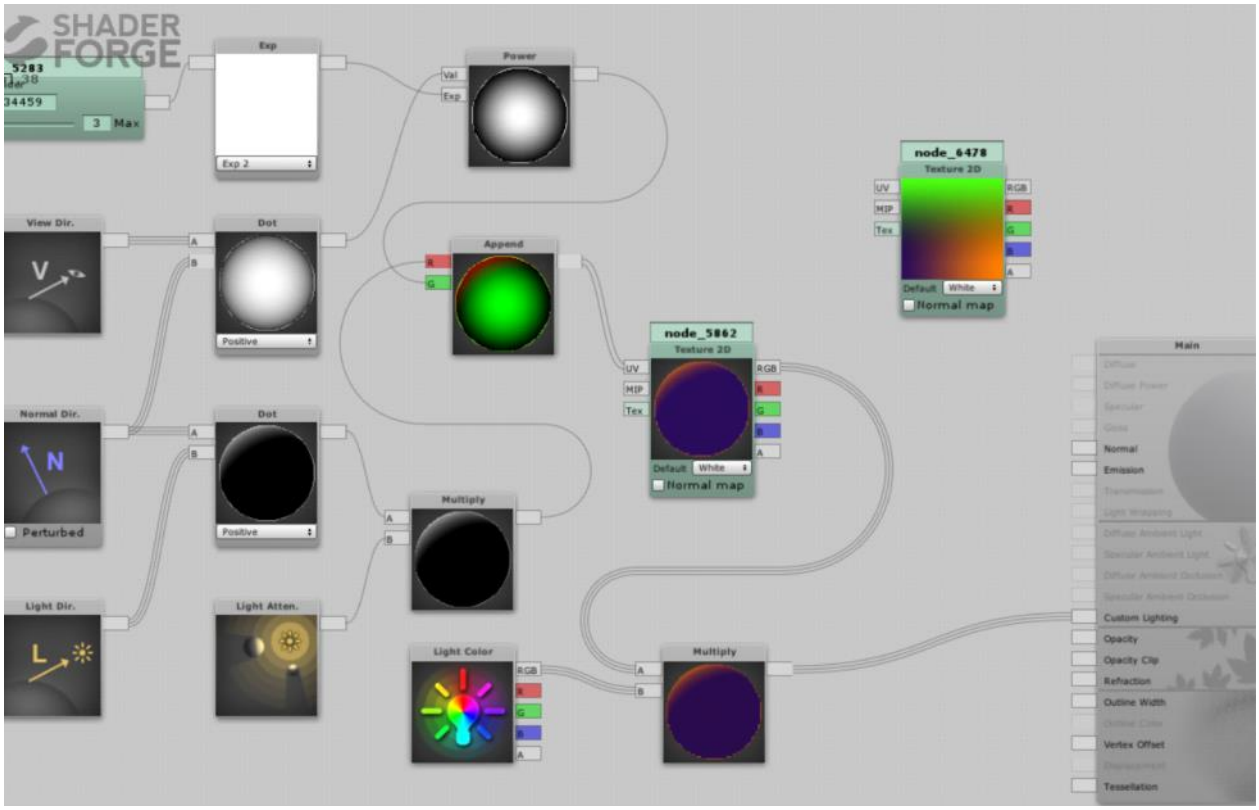
https://blog.csdn.net/xingzhe2001/article/details/83897914

在Shader Forge 实现 BRDF



制作BRDF贴图方法

- 首先利用法线与反射得到 漫反射
- 漫反射与灯光 点积
- 反射角 与法线 得到高光 并与E 进行 幂 运算
- 2 结果给 append 的 R 通道
- 3 结果给 G 通道
- append的组合通道与 带UV 的贴图进行点积 最后得到BRDF



最终效果

BRDF·基于物理的着色技术学习总结

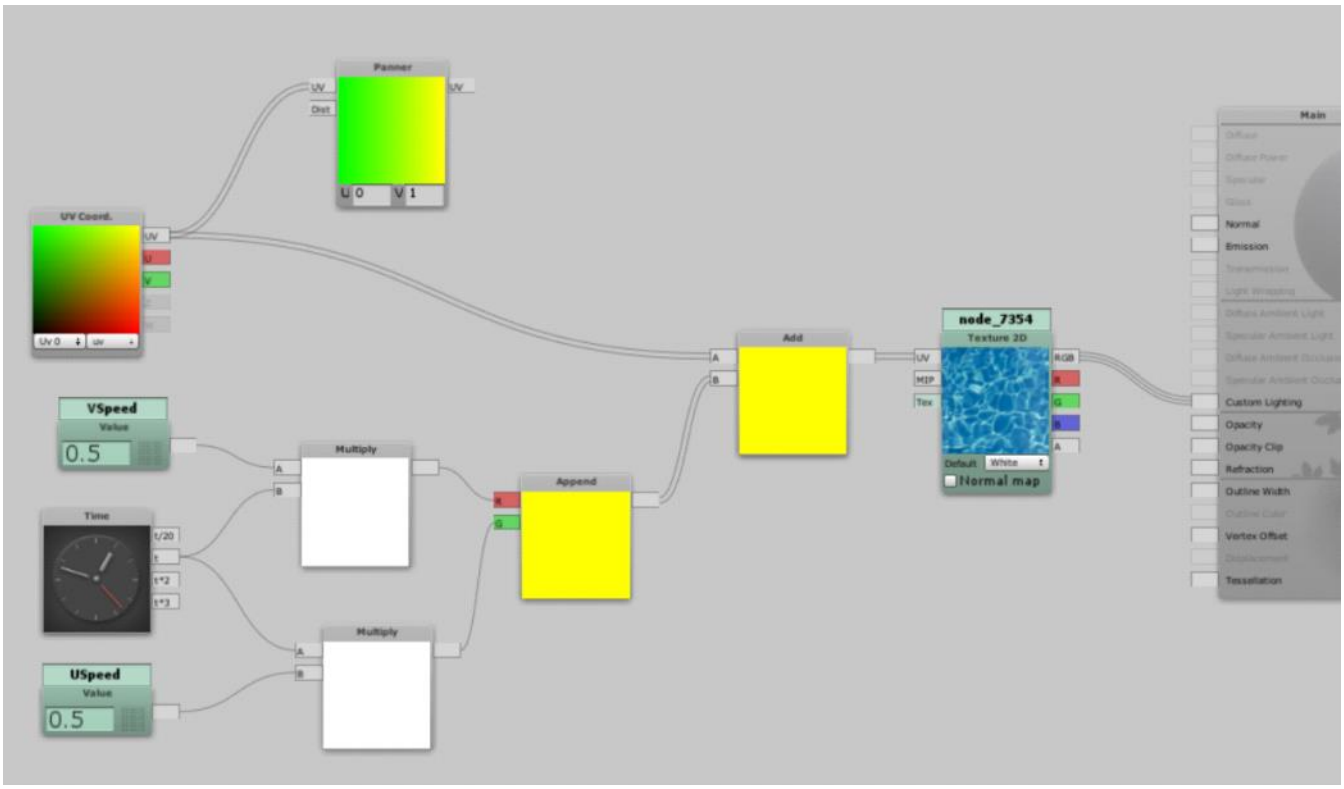
https://blog.csdn.net/yjr3426619/article/details/81098626

十二 UV效果

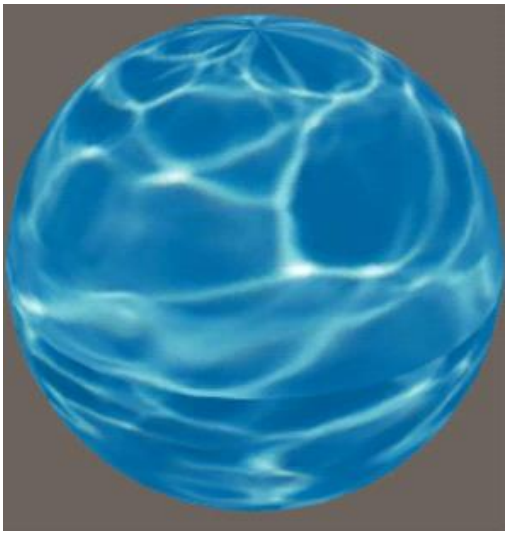
2019年8月15日 12:47

利用UV偏移 流动让图片动起来

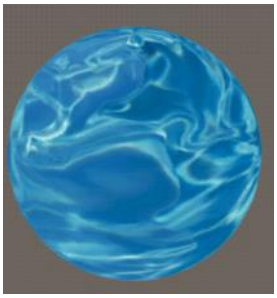
动起来的前提 是需要一个流动的数据 这里使用time
分别设置一个U和V的坐标量 与time 进行乘积
将两个数据进行升维得到 一个二维数据
将这个二维数据 与 UV 进行 叠加
新建一个 贴图 并将上面的UV信息输入
最后输出贴图信息 图片就开始根据 U 和 V 的大小进流动



最终效果

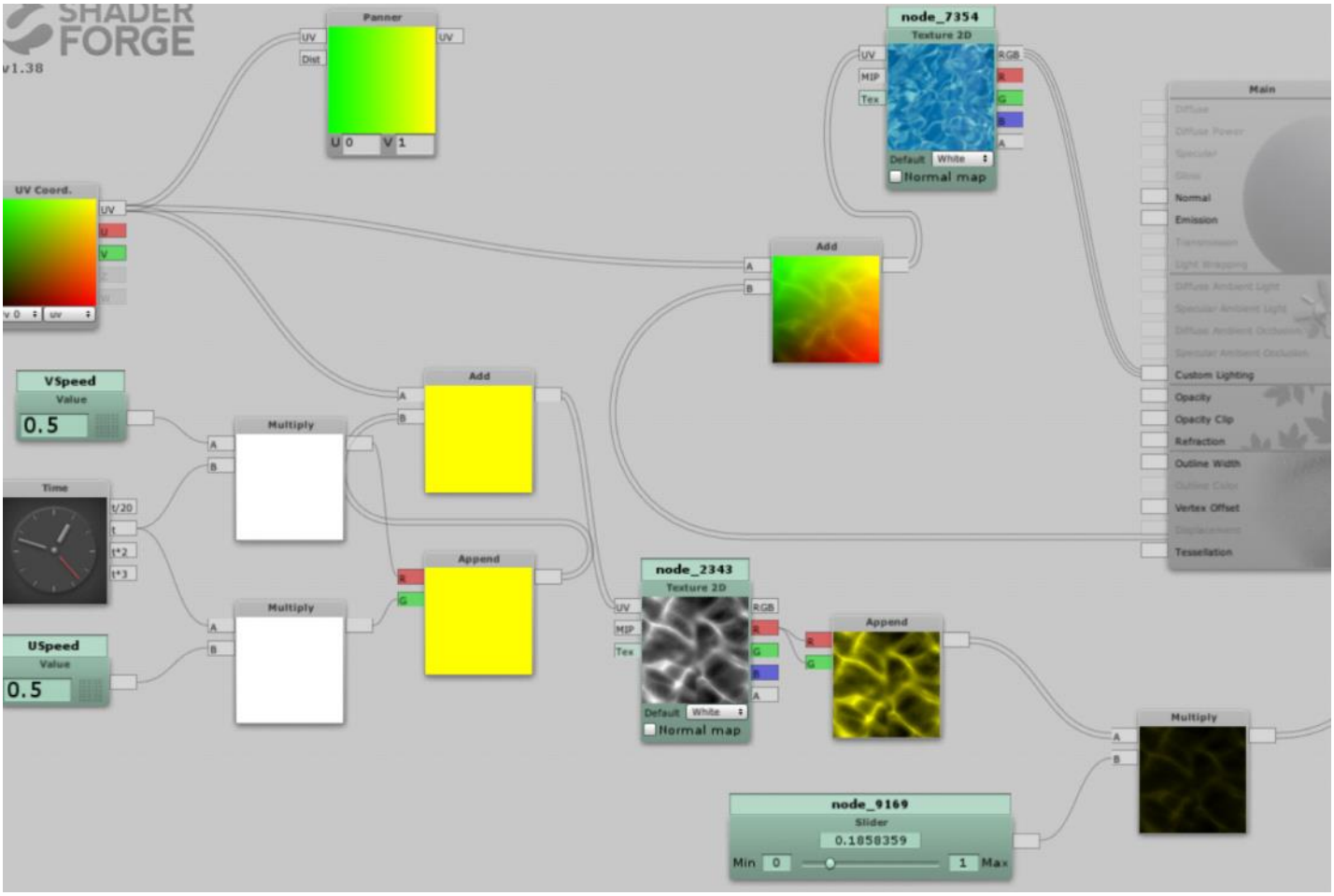


UV扭曲 添加波光粼粼的水



在UV流动的基础上
将叠加好的UV通道输出到以下内容
添加一个波纹的贴图并利用append降维到二维数据
利用一个slider 和这个数据进行乘积
将结果与 上面的 UV 坐标叠加 得到水光

最后在与之前的流动效果叠加 输出



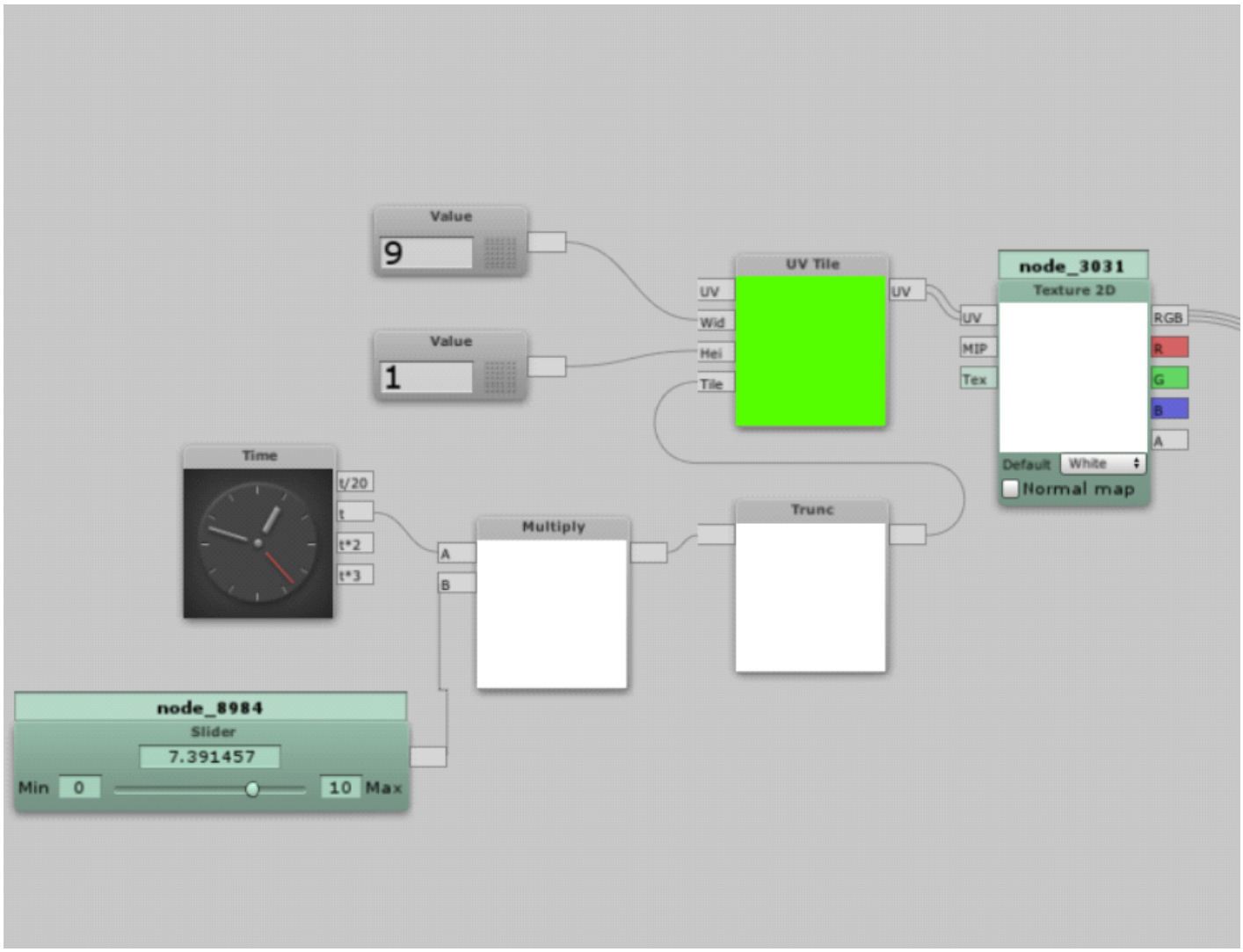

node_9169
Slider
0.1858359
Min 0
1 Max

十三 序列帧动画材质

2019年8月15日 20:40

利用UV 制作序列帧材质

主要原理是利用贴图上排列的帧图片 并利用UV 确定当前贴图所在区域
将 贴图的序列帧 长 和宽 作为UV的输入端
同时为了能动态的进行帧动画播放 需要一个自动的数据
利用时间数据 并将数据取整扩大 最后作为UV的 通道4 输出到 贴图的UV 最后给材质即可

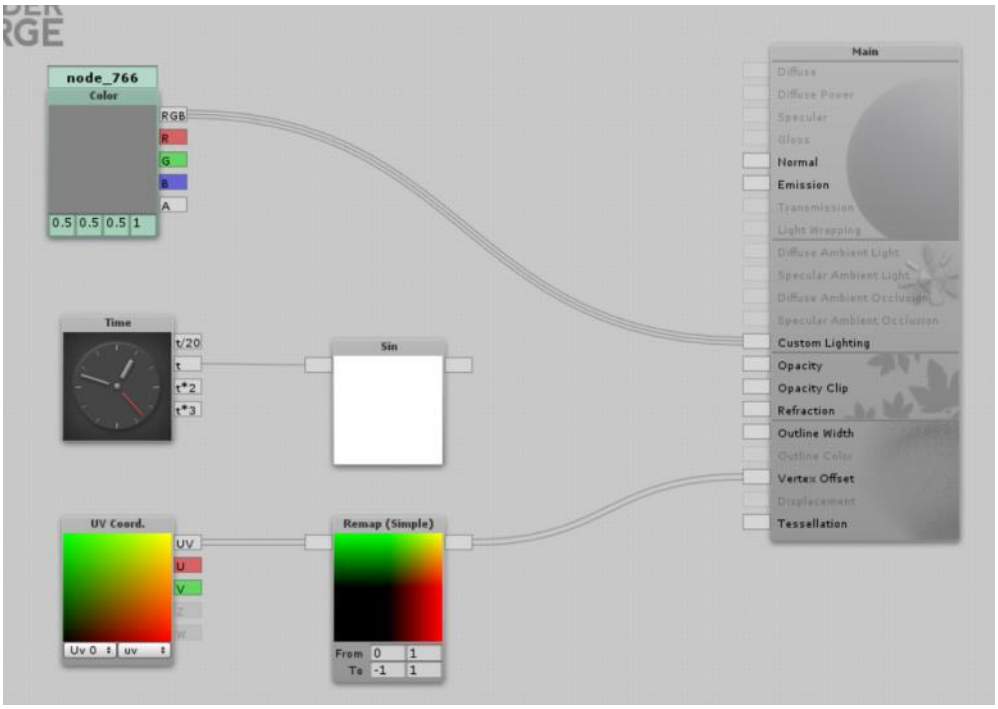


十四 屏幕特效与顶点坐标

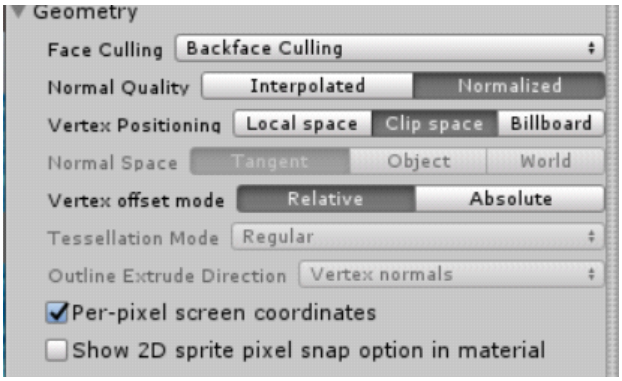
2019年8月15日 20:49

Vertex Offset 通道 -- 顶点偏移

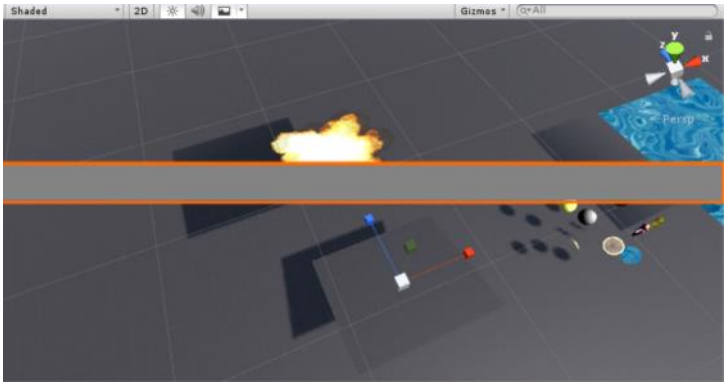
利用随机UV坐标进行顶点的移动



设置顶点参数



效果

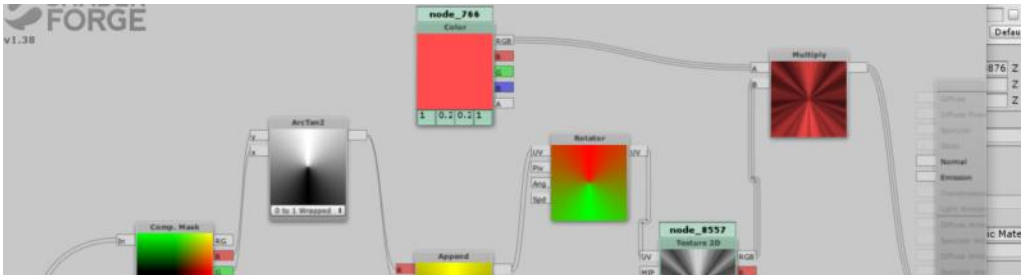


设置为顶点与世界坐标

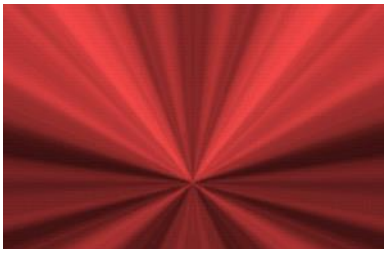
顶点由绝对 变相对

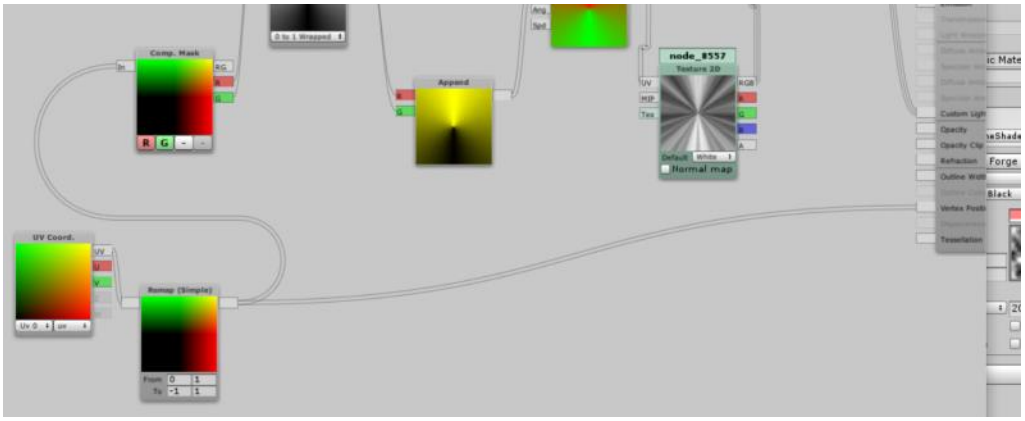
这样贴图就与摄像机视角一样了

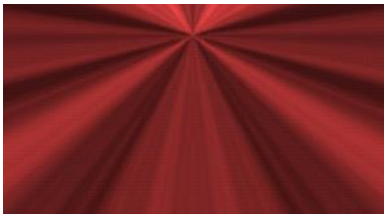
设置向内突击效果



着色结果

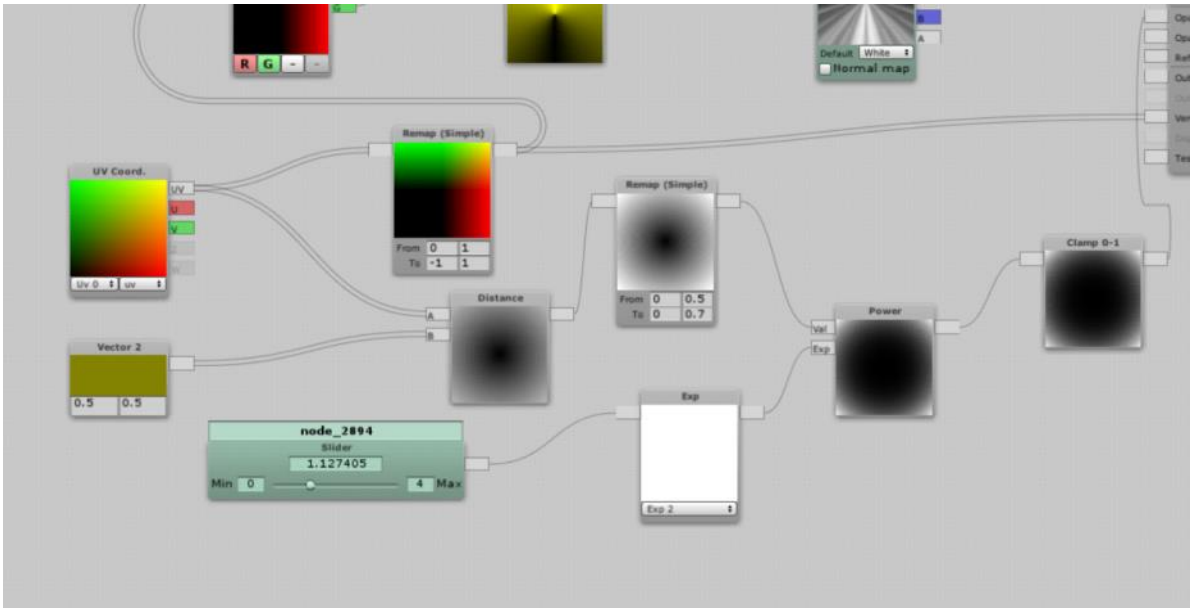




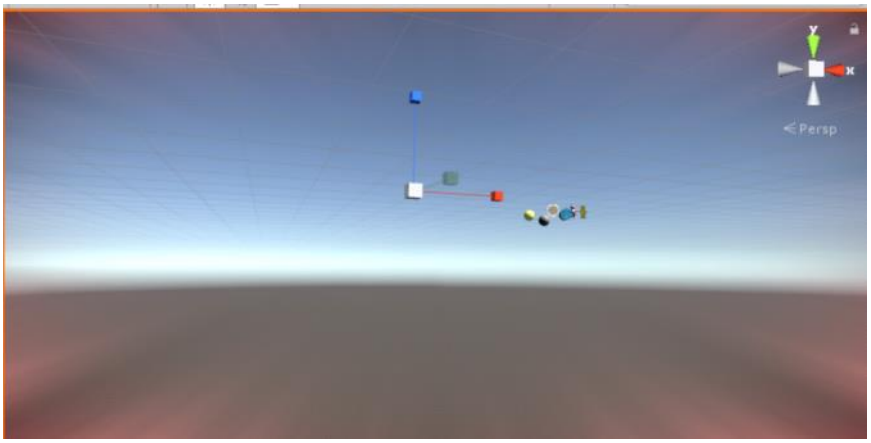


在上面的基础上 添加一个360度渐变扭曲(利用UV)的效果
与之前的效果结合

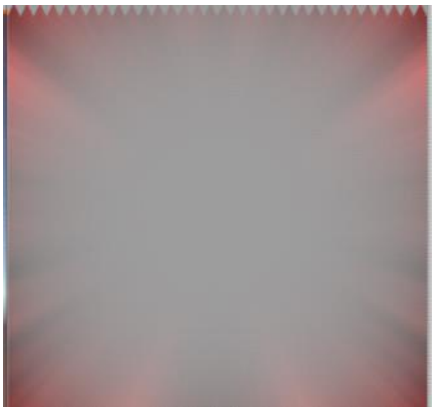
添加中心透明
添加一个UV 与 v2 信息进行融合
并利用数据进行扩大



最终效果



贴图效果

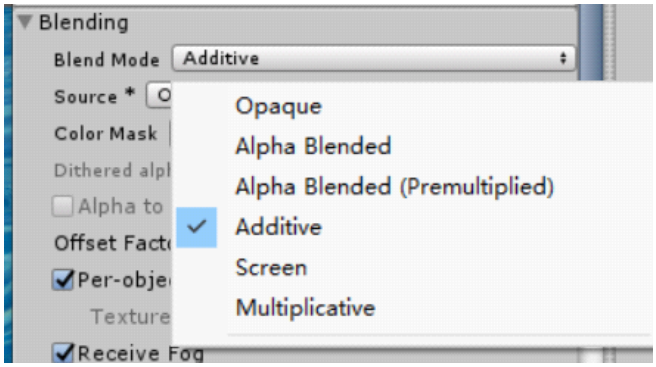


十五 自发光 扭曲 材质

2019年8月16日 12:41

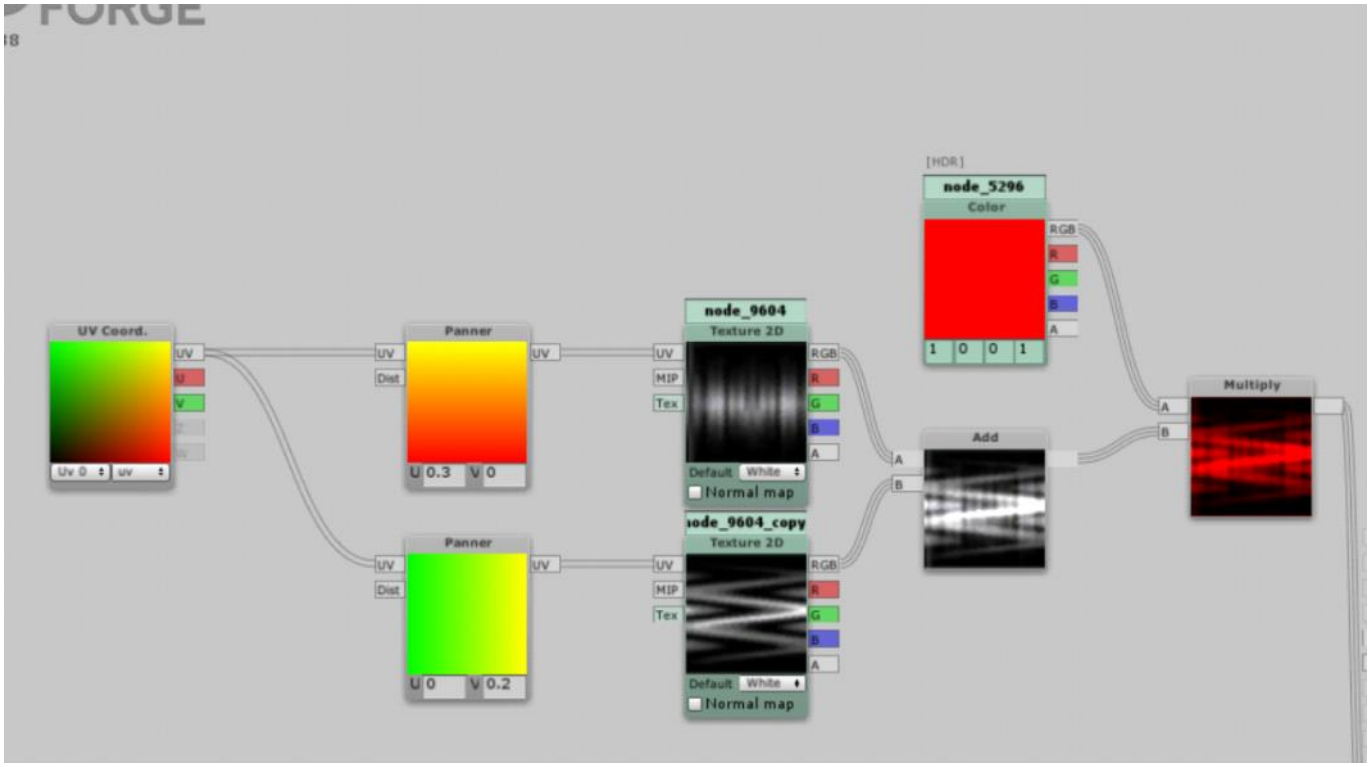
使用贴图和UV进行叠加

自发光 就是一种特效 需要修改透明通道为叠加模式

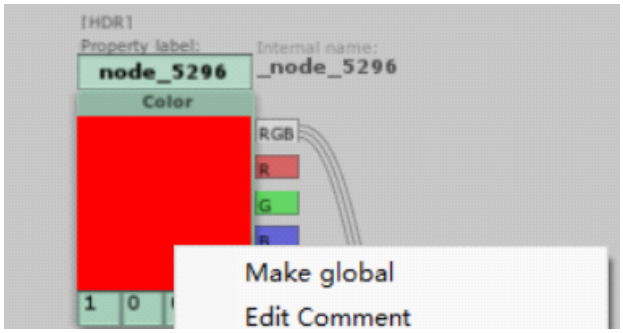


将两个UV输入贴图 使得贴图在左右和上下有流动的效果

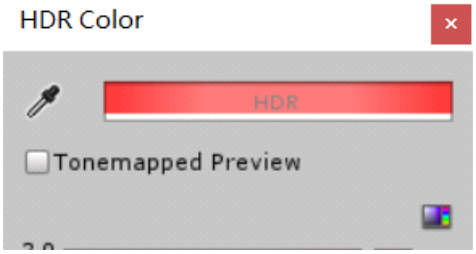
并将两个效果进行叠加 同时和一个颜色进行乘积 得到如下

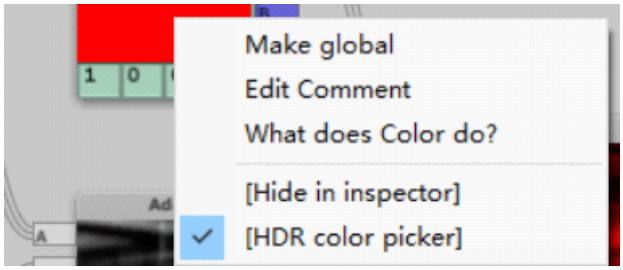


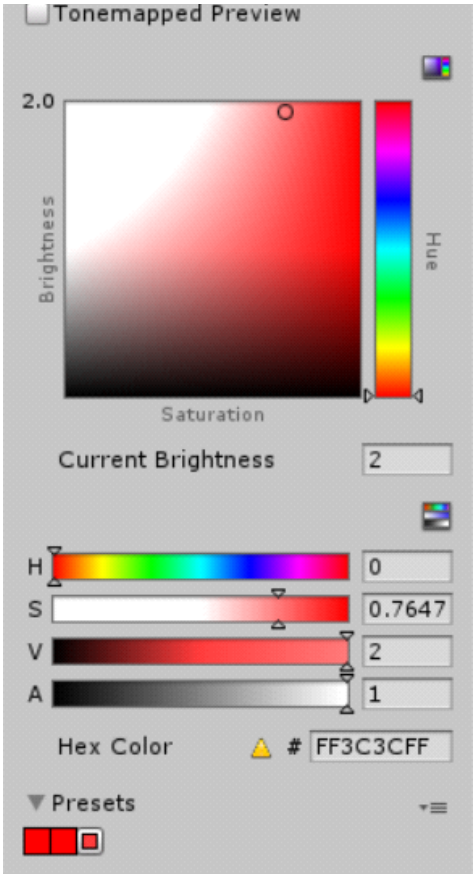
修改 color的颜色模式为 HDR



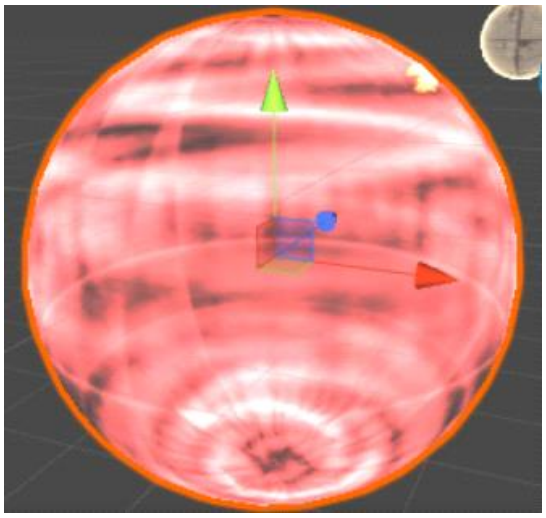
HDR颜色效果 会让边缘颜色过爆







最终贴图效果

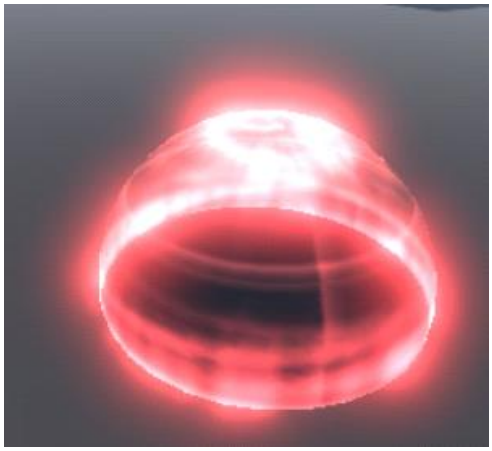


继续制作周围光溢出效果

加入 摄像机插件 官方免费 Post Processing Stack

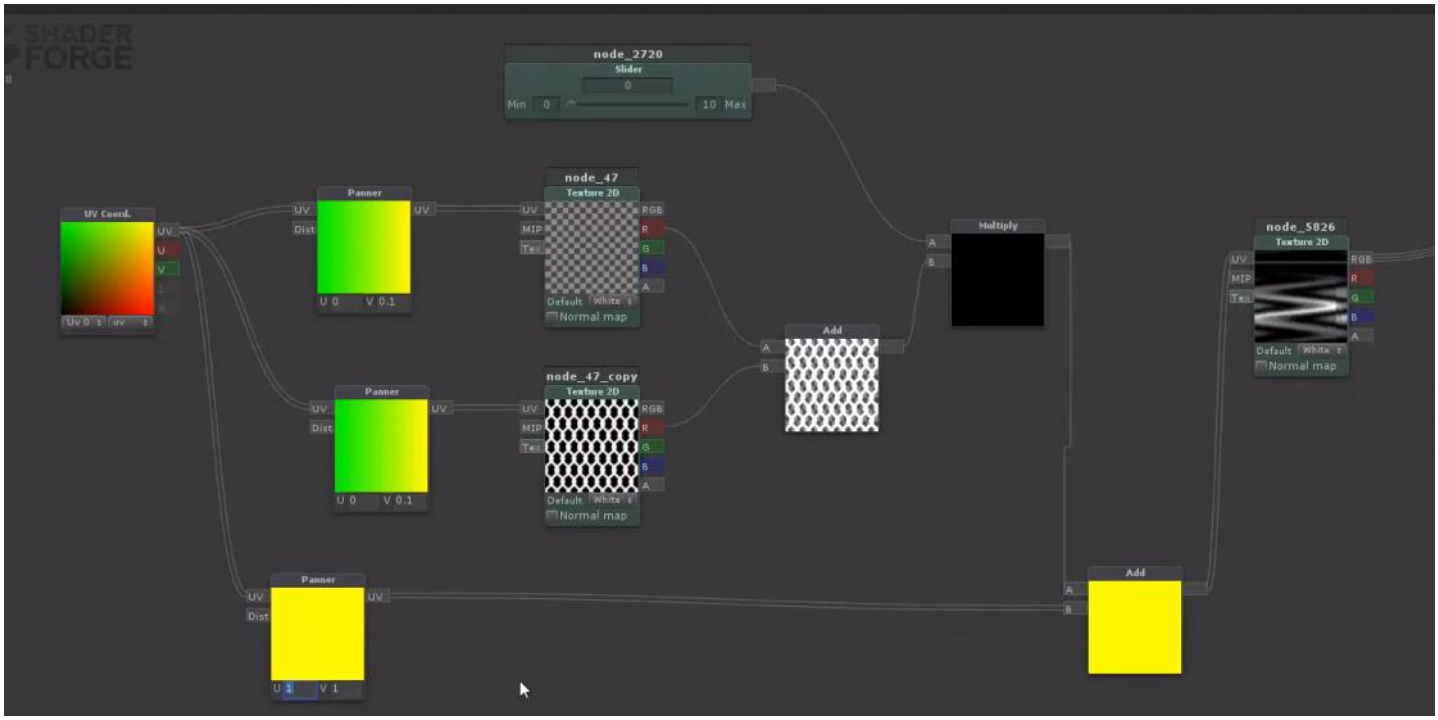
在摄像机 添加 post behavior 组件 勾选 Bloom 摄像机特效

让光流出来

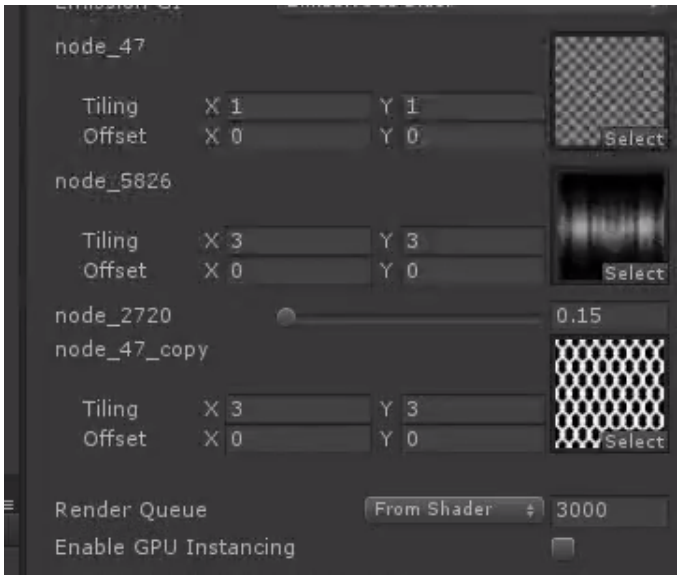


扭曲发光特效

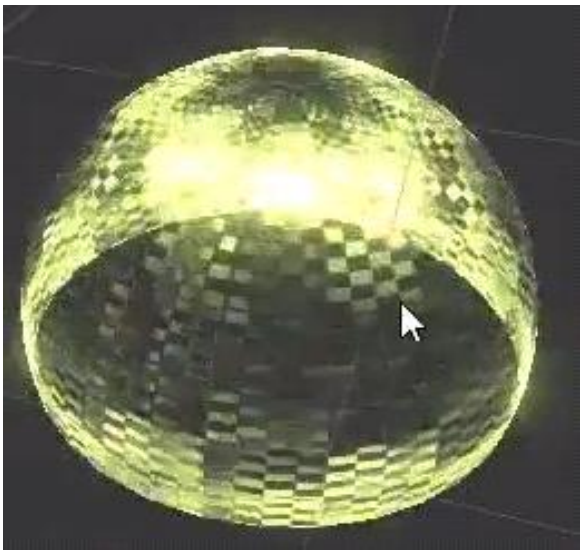
在上面的基础参数上 重新制作forge



稍微修改一下 UV坐标 并和color进行融合



最终效果



○一 UnityEditor基本介绍

2019年8月12日 14:36

关于Unity本身的说明

Unity 作为一个编辑器 其本身就是一个很丰富的程序
程序的基本要求就是可修改与良好的拓展性
所以拓展性使得Unity本身虽然不具有枪大的拓展功能
但是其可以进行Editor模式编写
这种模式下不会被编译到游戏内 也不会因为出错使得游戏无法运行
只会在使用拓展或插件的时候进行管理

此外 若需要编写编辑扩展或者插件需要指定目录
Editor/ 这个目录下的所有内容都不会被编译到游戏内

Editor模式下操作撤回

在Editor内用代码进行的操作 一般是不能进行撤回的 必须加以代码约束
比如利用窗口创建了一个空游戏物体 想撤回操作必须加以代码

```
GameObject go = new GameObject(gameObjectName);
// 参数 操作的游戏对象 当前操作的名字(最好是每一条都不重复)
Undo.RegisterCreatedObjectUndo(go, "craete a gameObject[" + go.name + "]");
```

再比如想撤回修改了某个脚本组件的一些字段的值 需要提前存入操作缓存

```
Undo.RecordObject(enemyHealthItem, item.name + "has changed health and speed successfully");
enemyHealthItem.startingHealth += changeStartHealthValue;
enemyHealthItem.sinkSpeed += changeSinkSpeedValue;
```

〇二 Unity的菜单项

2019年8月12日 14:49

如何在Unity编辑器本身添加新的菜单

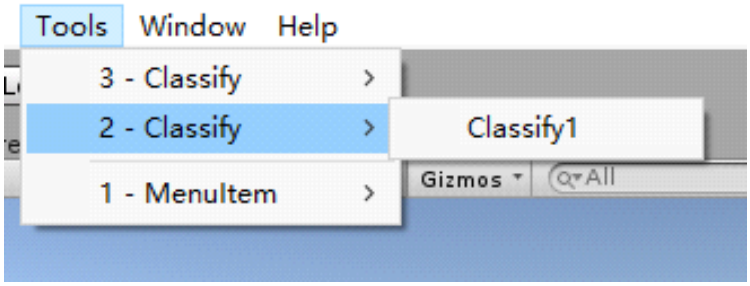
Editor/Scripts/Tools.cs
新建一个类 并引用 UnityEditor 库

```
//编辑器新建一个菜单 方法必须是静态的 路径可以放在Unity本地菜单中
[MenuItem("Tools/New A MenuItem")]
public static void NewAMenuItem()
{
    Debug.Log("点击了 新建一个菜单项");
}
```

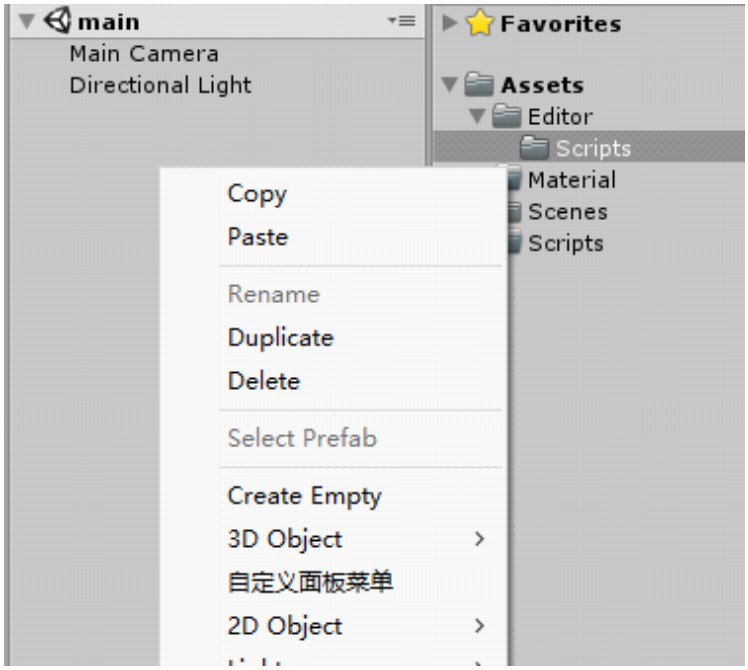
Unity菜单栏分类和优先级

[MenuItem("菜单目录",是否关闭菜单,菜单优先级)]
菜单会根据当前菜单项进行优先级分类 并根据分类结果进行排序和组合
排列组合规则

- 优先级越小 最后排在下面
- 菜单项之间的优先级大于11 会自动进行分类
- 不同类别的菜单项之间也满足优先级小在下原则
- 类别之间的菜单之间会自动生成一个分隔符



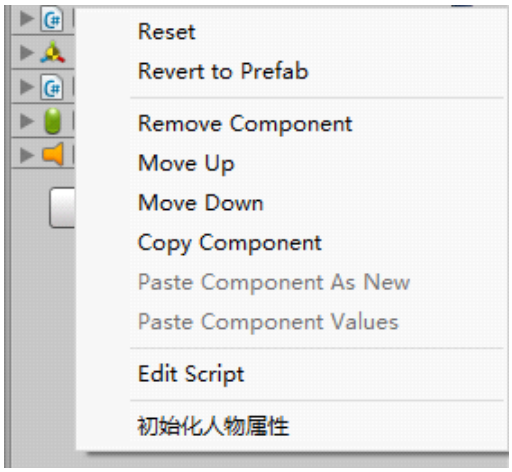
- Unity编辑器自带的菜单项同样满足这个规则
- 在面板右键得到的菜单一般依赖菜单栏中的 所以扩展的菜单也会引入到面板右键菜单



给固定组件添加菜单项

根目录是 CONTEXT / 组件名字 /菜单名字

```
//给固定的菜单添加菜单项
//这里给 PlayerHealth 组件 右键菜单添加了一项
[MenuItem("CONTEXT/PlayerHealth/初始化人物属性")]
public static void InithealthAndSpeed()
{
    Debug.Log("初始化人物属性");
}
```



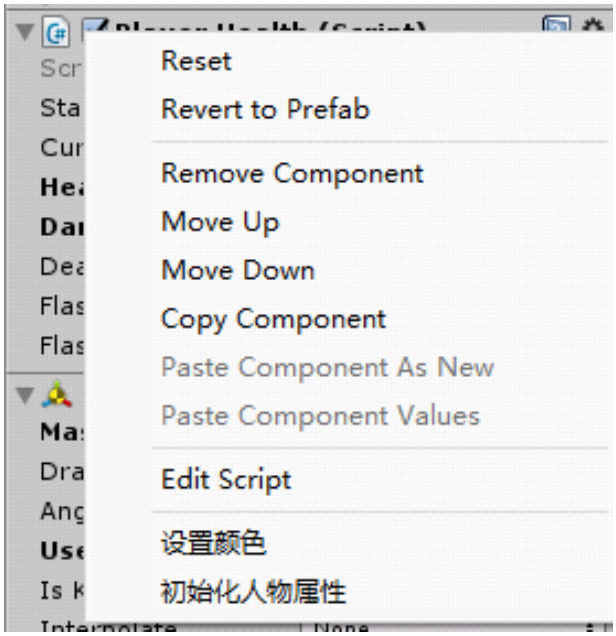
如何操作这个组件

```
[MenuItem("CONTEXT/PlayerHealth/初始化人物属性")]
```

```
public static void InithealthAndSpeed(MenuCommand mcd)
{
    //MenuCommand mcd中的context 是当前操作的组件对象 是一个Component
    CompleteProject.PlayerHealth ph = mcd.context as CompleteProject.PlayerHealth;

    if(ph != null)
    {
        ph.startingHealth = 150; //修改当前组件内的属性
        ph.flashSpeed = 20;
    }

    Debug.Log("初始化人物属性");
}
```

第二种方法 直接在可以进入代码的组件中添加右键菜单代码
利用ContextMenu标签 同时当前方法不需要为静态的 但是方法会作用到全体对象

```
    [ContextMenu("设置颜色")]
    public void SetColor()
    {
        flashColour = Color.white;
    }
```



给菜单项添加快捷键

添加快捷键在菜单名字后面加空格 加下划线然后加快捷键规则
其中组合快捷键 按%-> control #-> shift &-> alt
比如 ctrl + alt +t ==> xxx/xxx/xxx _%&T

```
//添加菜单项的快捷键
[MenuItem("Tools/5 - HotKey/HotKeyT _T", false,300)]
public static void HotKeySetT()
{
    Debug.Log("使用了快捷键 T");
}
[MenuItem("Tools/5 - HotKey/HotKeyCTRL+T _%T", false, 300)]
public static void HotKeySetCTRL_T()
{
    // %-> control #-> shift &-> alt
    Debug.Log("使用了组合快捷键 ctrl + T");
}
[MenuItem("Tools/5 - HotKey/HotKeyALT+T _&T", false, 300)]
public static void HotKeySetATL_T()
{
    // %-> control #->shift &alt
    Debug.Log("使用了组合快捷键 alt + T");
}
```

菜单操作项是否启用的验证

实现方法

- 新建一个相同的menuitem 同时设置第二个参数为treu
- 方法名字随意 并将函数返回值设置为bool
- 在方法内进行判断 启动菜单之前会先运行当前判断方法 然后才会进行菜单渲染

具体代码

```
//菜单项是否能点击 第二个参数
[MenuItem("GameObject/删除游戏物体", true, 1)]
```

```
public static bool IsDeleteGO()
{
    if (Selection.objects.Length > 0)
        return true;
    else
        return false;
}

[MenuItem("GameObject/删除游戏物体",false,1)]
public static void DeleteGO()
{
    foreach (Object item in Selection.objects)
    {
        //GameObject.Destroy(item); //不可撤销的删除
        Undo.DestroyObjectImmediate(item);//可以撤销
        //这个曹施写入到了 操作缓存中
    }
}
```

对变量添加右键菜单项

可以给我们在脚本中添加的属性/字段 添加一个右键菜单
这个属性前提是public 因为只有public 才会显示在Unity属性面板
在属性之前添加[contextMenuItem("菜单名字","方法名字")]
方法必须存在 否则编译出错

右键点击 Starting Health 属性会弹出自定义菜单选项


```
//对下面这个属性 startingHealth 添加右键菜单
//参数一 菜单项名  参数二 方法名
[ContextMenuItem("增加血量", "AddHP")]
```

```
public int startingHealth = 100;

public void AddHP()
{
    startingHealth += 10;
}
```

〇三 Selection类简介

2019年8月12日 16:24

Selection是关于选择物体和操作的类

其中包含了 访问编辑器中的选择项

静态方法

activeContext	返回当前的上下文对象，与通过 SetActiveObjectWithContext 设置时相同。
activeGameObject	返回处于活动状态的游戏对象。（显示在检视面板中的对象）。
activeInstanceID	返回实际对象选择的 instanceID。包括预制件、不可修改的对象。
activeObject	返回实际对象选择。包括预制件、不可修改的对象。
activeTransform	返回处于活动状态的变换。（显示在检视面板中的变换）。
assetGUIDs	返回所选资源的 GUID。
gameObjects	返回实际游戏对象选择。包括预制件、不可修改的对象。
instanceIDs	来自场景的实际未过滤选择，以实例 ID 形式返回，而 objects。
objects	来自场景的实际未过滤选择。
selectionChanged	当前活动/所选项发生更改时触发的委托回调。
transforms	返回顶级选择，不包括预制件。

静态变量

Static Methods

Contains	返回某个对象是否包含在当前选择中。
GetFiltered	返回按类型和模式筛选的当前选择。
GetTransforms	允许使用 SelectionMode 位掩码对选择类型进行精细控制。
SetActiveObjectWithContext	选择具有上下文的对象。

○四 Editor的窗口

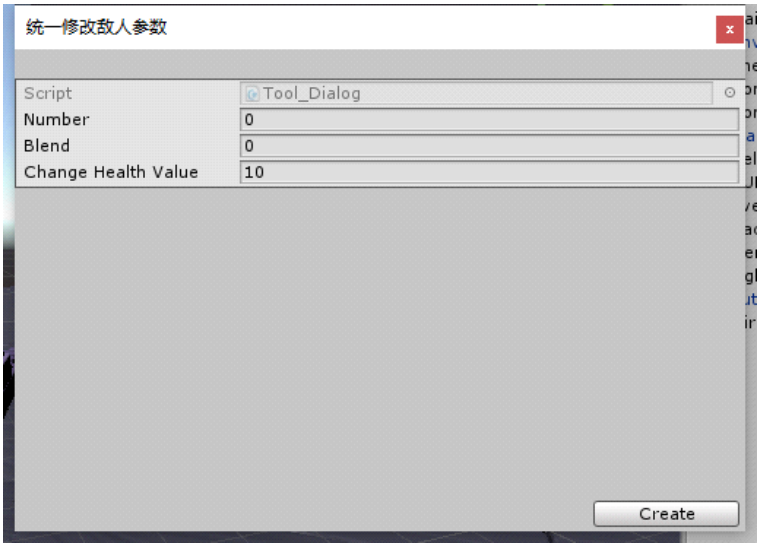
2019年8月13日 8:20

Editor 弹出式对话框的操作使用

对话框比菜单更加自由并且更容易扩展 继承自 ScriptableWizard

```
public class Tool_Dialog : ScriptableWizard
{
    //打开对话框的菜单项
    [MenuItem("Tools/6 - ChangeHealthDialog")]
    public static void ChangeHealthDialog()
    {
        ScriptableWizard.DisplayWizard<Tool_Dialog>("统一修改敌人参数");
    }
    public int number;
    public float blend;
    public int changeHealthValue = 10;
}
```

在菜单项点击设定的菜单 弹出对话框 属性显示在上面



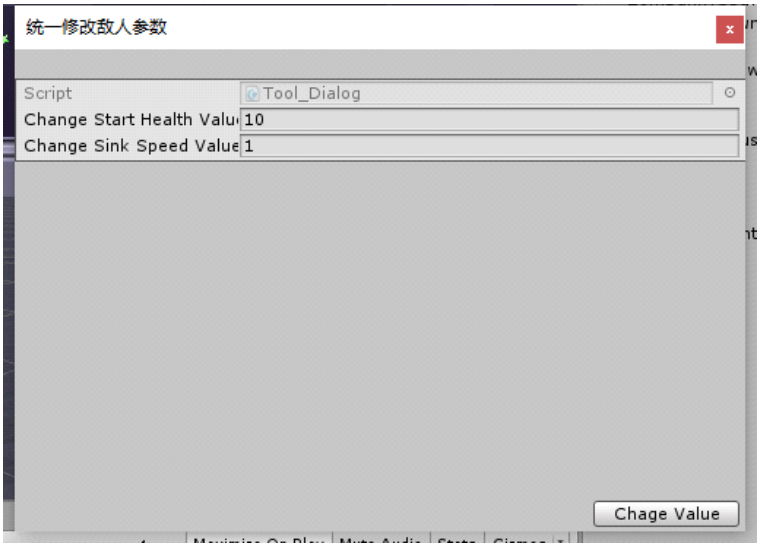
如何响应这个按钮 并对其进行修改

在创建的时候 添加第二个参数 为按钮名字

ScriptableWizard.DisplayWizard<Tool_Dialog>("统一修改敌人参数","Chage Value");

同时 这个按钮有专门的响应函数 并且这个按钮点击后会关闭弹窗

```
private void OnWizardCreate(){ }
```



其他的面板函数

```
//自带的 固有方法 每次字段更新调用
private void OnWizardUpdate()
{
    if (Selection.gameObjects.Length > 0)
    {
        //修改 帮助信息
        helpString = "您选择了" + Selection.gameObjects.Length + "个敌人";
        errorString = "";
    }
    else
    {
        //修改 错误信息
        errorString = "你未能选择任何游戏物体";
        helpString = "";
    }
}
// 选择物体发生了改变的时候调用
private void OnSelectionChange()
{
    OnWizardUpdate();
}
// 固有函数 当点击按钮进行操作
private void OnWizardCreate()
{
    Debug.Log("点击了Crete按钮");
    ShowEnemyHealthInfo();
    ChangeHealthAndSpeed();
    ShowEnemyHealthInfo();
```

```
}

//显示提示信息 提示信息需要封装到GUI中
ShowNotification(new GUIContent("修改了 " + selectedPrefabs.Length + " 个预设的参数"));
```

记录每次打开弹窗修改的值 以用于下次打开

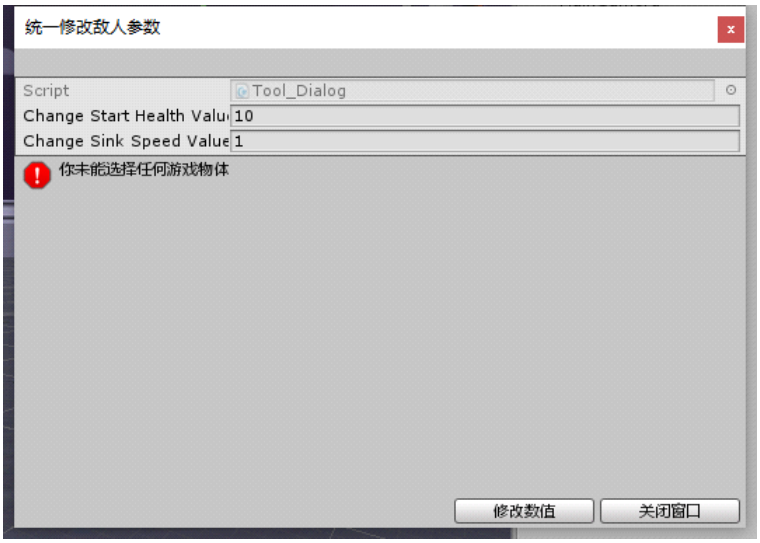
```
// 当弹窗创建的时候被调用
private void OnEnable()
{
    //从之前的值读取并 写入新的弹窗字段数据
    changeStartHealthValue = EditorPrefs.GetInt("Tool_Dialog.changeStartHealthValue");
    changeSinkSpeedValue = EditorPrefs.GetInt("Tool_Dialog.changeSinkSpeedValue");
}

//自带的 固有方法 每次字段更新调用
private void OnWizardUpdate()
{
    // 每次字段被修改后 保存在弹窗中修改的值
    EditorPrefs.SetInt("Tool_Dialog.changeStartHealthValue", changeStartHealthValue);
    EditorPrefs.SetInt("Tool_Dialog.changeSinkSpeedValue", changeSinkSpeedValue);
}
```

在面板创建其他按钮

在创建函数中添加第三个参数即可 参数是按钮的名字

Other Button 也是自带的按钮 也有自己的响应函数 但是他不会点击后关闭了弹窗

```
ScriptableWizard.DisplayWizard<Tool_Dialog>("统一修改敌人参数","Chage Value","other button");
//响应其他按钮的操作
private void OnWizardOtherButton()
{
    Debug.Log("响应其他按钮");
}
```



〇五 Editor进度条

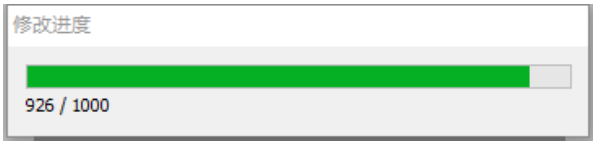
2019年8月13日 14:52

设置一个简单的Editor默认的进度条

```
//添加一个修改进度条 0-1的百分百进度
//参数 进度条名字 显示在进度条下面的信息 进度百分比(0 - 1)
EditorUtility.DisplayProgressBar("修改进度", "0 / " + selectedPrefabs.Length + "完成修改值", 0);
```

简单的在一个循环中一直设置这个进度条的信息 以实现动画效果

```
for (float i = 0; i <= 1000; i++)
{
    EditorUtility.DisplayProgressBar("修改进度", i + " / " + 1000, (float)i / 1000);
}
```



〇六 创建自定义窗口

2019年8月13日 15:09

创建丰富的满足需求的复杂窗口

Editor模式不仅提供了 ScriptableWizard 的 窗口 也可以说是对话框
也提供 EditorWindow 这个类 满足丰富窗口的创建
Editor Window 与系统内置的窗口别无二致 并且能依附和贴在系统窗口上

创建一个自定义系统窗口

```
    //打开自定义窗口的的菜单项
    [MenuItem("Tools/7 - EditorWindow")]
    public static void NewEditorWindow()
    {
        // 创建一个自定义窗口
        Tool_EditorWindow myWindow = EditorWindow.GetWindow<Tool_EditorWindow>();
        myWindow.Show();//显示窗口
    }

    //绘制窗口内容 固有方法 在这个方法内绘制窗口内的东西和响应
    private void OnGUI()
    {
        // 用GUI创建一行文字
        GUILayout.Label("当前窗口用来增加自定义游戏物体\n");
        // 设置一个可返回输入内容的文本行
        gameObjectName = GUILayout.TextField(gameObjectName);

        // 设置一个按钮 并响应
        if (GUILayout.Button("创建游戏物体"))
        {
            if (gameObjectName == "" || gameObjectName == null)
            {
                //弹窗提醒 几秒后自动消失
                ShowNotification(new GUIContent("游戏物体名字不能为空"));
            }
            else
```

```
        {
            GameObject go = new GameObject(gameObjectName);
            Undo.RegisterCreatedObjectUndo(go, "craete a gameObject[" + go.name + "]");
        }
    }
}
```

○○ DoTween介绍 及队列

2019年8月14日 17:30

对于UI进行效果实现的插件

DoTween对于UI的渐变效果 数字的变化 切换等变化进行简单的处理
DoTween本身的代码是基于UnityEngine内的框架实现的

动画效果的队列与播放

```
//设置动画队列
Sequence sequence = DOTween.Sequence();
//添加一个移动动画
sequence.Append(transform.DOMove(Vector3.up, 2));
//加入与上一个动画共同启动的动画
sequence.Join(transform.DOMove(Vector3.up, 2));
sequence.Append(transform.DOMove(Vector3.left, 2));
sequence.AppendInterval(1);//添加延迟
sequence.Append(transform.DOMove(Vector3.forward, 2));

//插入一个动画 在哪一秒插入一个动画
sequence.Insert(0, transform.DOMove(Vector3.forward, 2));
//加入回调函数 利用lambda表达式
sequence.InsertCallback(5, ()=> { CallBcak(); });
  private void CallBcak()
  {
      Debug.Log("在第五秒使用了回调函数");
  }
```

〇一 位移旋转缩放

2019年8月14日 15:18

移动旋转与缩放的变化动画

DoTween扩展了 Unity 自身的一些基本方法 包括移动 旋转等
这些移动方式都是 曲线方式的 带缓冲效果

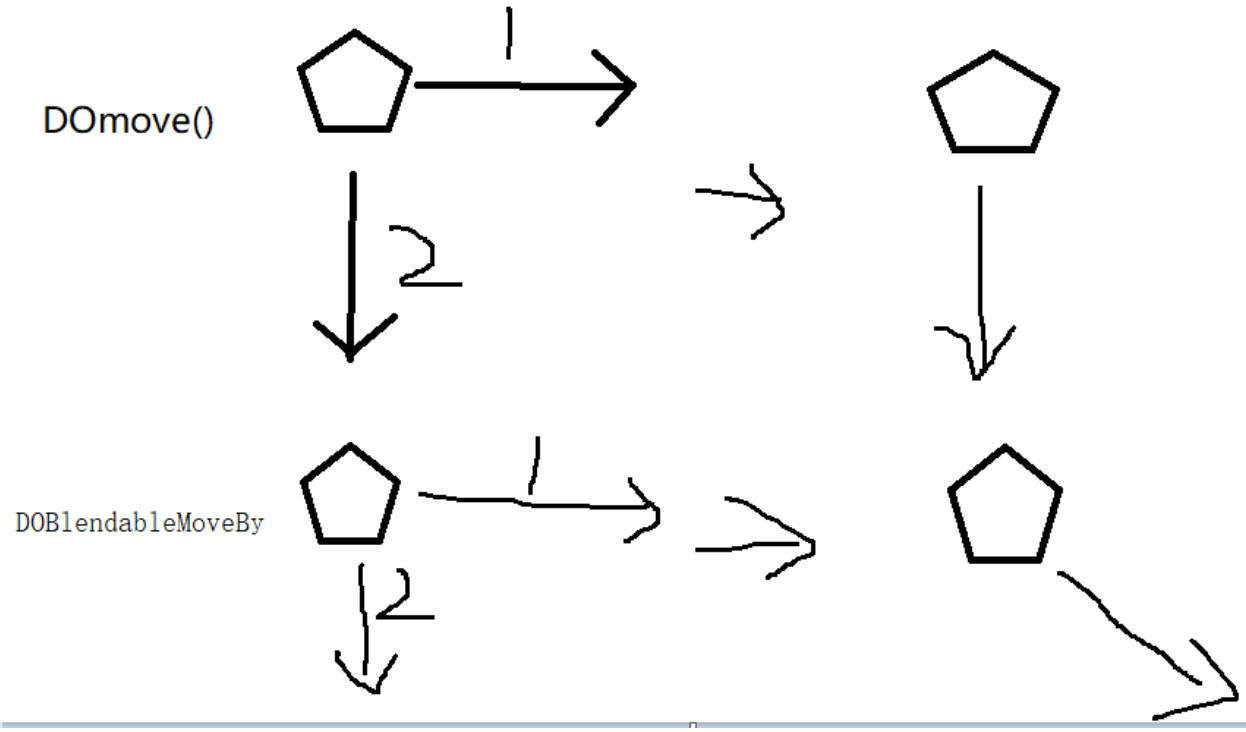
```
  private void Start()
  {
      //利用 DG 的拓展方法进行移动 世界坐标
      //transform.DOMove(Vector3.one, 3);
      //transform.DOLocalMoveX / y /z  //按XYZ 进行移动
      //transform.DOLocalMove(); 自身坐标进行移动

      //利用 DG 拓展进行旋转 欧拉角 / 四元数 /朝向旋转
      //transform.DORotate(new Vector3(100,0,0), 3);
      //transform.DORotateQuaternion(new Quaternion(0.1f, 0.1f, 0.1f, 0.1f), 2);
      //transform.DOLookAt(Vector3.one, 3);

      //利用 DG 拓展进行缩放
      //transform.DOScale(Vector3.one * 2, 2);

      //模拟实现高出物体掉落地面反弹效果 震颤 冲压效果
      //参数  punch:方向(方向力) duration:持续时间 vibrator 震动反弹频率 elastcitiy:0-规律运动 !0-根据算法运动
      //transform.DOPunchPosition(new Vector3(0, 2, 0), 5, 999, 0f);
      //关于旋转 / 缩放的时候 进行震颤 冲压回弹效果
      //transform.DOPunchRotation(new Vector3(0, 90, 0), 5, 999, 0f);
      //transform.DOPunchScale(new Vector3(3, 3, 3), 5, 999, 0f);
  }
```

○二 Blend 融合效果

2019年8月14日 16:08

对于普通的 DOMove 可能会同时出现多个移动
这种DOMove 的方法会覆盖之前的效果
可以利用 Blend 进行混合 移动 / 旋转 / 缩放 等操作

```
transform.DOMove(Vector3.one, 2);
transform.DOMove(Vector3.one * 2, 2);

//Blend 的使用 混合移动 / 旋转 /缩放 的函数
//利用 DOBlendableMoveBy 以混合不同的 移动结果和过程
transform.DOBlendableMoveBy(Vector3.one, 2);
transform.DOBlendableMoveBy( - Vector3.one * 2, 2);
```



利用Blend 进行颜色混合 红色 + 绿色 进行渐变

```
material.DOBlendableColor(Color.red, 2);
material.DOBlendableColor(Color.blue, 2);
```

〇三 材质内属性的变化

2019年8月14日 16:22

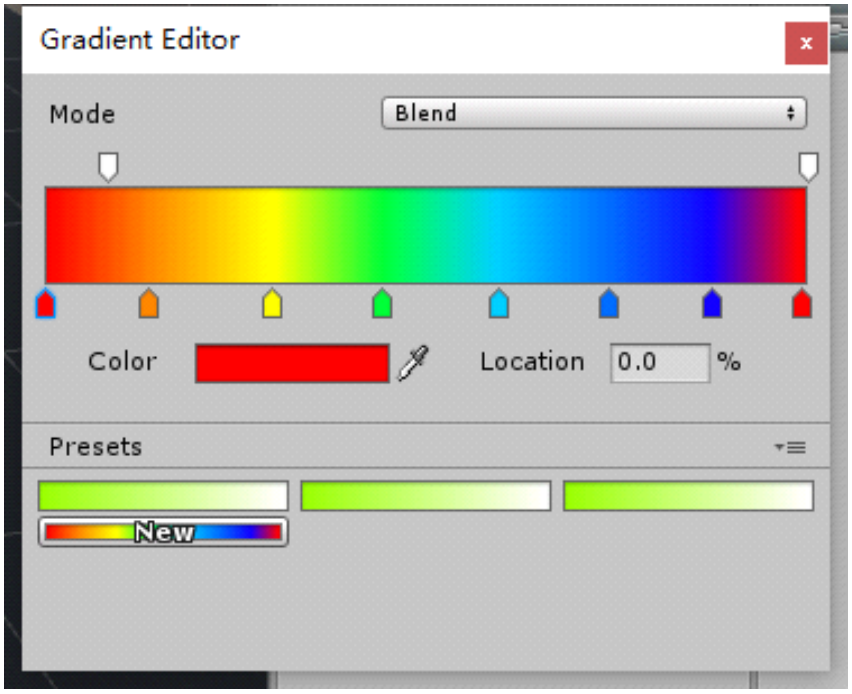
对于材质内容属性的逐渐变化

对于所有的游戏物体的组件属性 常用的基本DoTween都实现了逐渐变化效果
材质内的 颜色 渐变颜色 透明度 褪色的 变化效果动画

基本的材质 颜色 变化

```
//材质 Material 获取
Material material = GetComponent<MeshRenderer>().material;
// 获取了材质 并缓缓将暗色改变为红色
//material.DOColor(Color.red,3); //shader内必须要渐变颜色
// 如果没有正常的颜色设置 而是TintColor则
// material.DOColor(Color.red, "_TinkColor",3); //第二个参数是颜色名字
//让颜色逐渐透明
//material.DOColor(Color.clear, 2);
//利用fade 进行逐渐褪色
//material.DOFade(0f, 2);
//gradient 是 渐变颜色编辑器
material.DOGradientColor(gradient, 7);
```

其中家变颜色编辑器 public Gradient gradient;


Gradient Editor
Mode
Blend
Color
Location
0.0
%
Presets
New

〇四 摄像机效果

2019年8月14日 17:04

DoTween 作用于摄像机

```
Camera camera;

private void Start()
{
    camera = GetComponent<Camera>();
    //改变摄像机的宽高比 改变时间
    //camera.DOAspect(1, 2);
    //改变摄像机的颜色
    //camera.DOColor(Color.red, 2);
    //透视模式 逐渐改变摄像机的摄影范围 这样物体就会变大
    //camera.DOFieldOfView(1, 3);
    //正交模式 逐渐改变摄像机的摄影范围 这样物体就会变大
    //camera.DOOrthoSize(1, 3);
    //改变摄像机的像素范围 将摄像机移动并设置大小 按大小
    //camera.DOPixelRect(new Rect(360, 360, 500, 500), 3);
    //改变摄像机在game窗口的存在比例和位置 (同上) 按比例
    //camera.DORect();

    //摄像机晃动效果 晃动位置 晃动旋转
    //camera.DOShakePosition(3,1,10);
    //camera.DOShakeRotation(3,10,10);
}
```

〇五 Text效果实现

2019年8月14日 17:14

DoTween作用域text组件

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using DG.Tweening;
using UnityEngine.UI;

public class Texttest : MonoBehaviour
{
    private Text text;
    private string str = "李世东牛逼啊 李世东!!!";
    private float start = 100, end = 1000;
    private void Start()
    {
        text = GetComponent<Text>();


        //让文字一个一个的出来
        // .SetEase(Ease.Linear) 让这个动画变化速度均匀
        //text.DOText(str, str.Length/2).SetEase(Ease.Linear);

        // 让数据从开始逐渐变化到最后
        DOTween.To(
            () => start,
            x => { text.text = Mathf.Floor(x).ToString(); },
            end,
            3f
```

```
        ).SetEase(Ease.Linear).SetUpdate(true); //设置计算不受帧数影响
    }
}
```

〇六 DoTween参数设置

2019年8月15日 8:41

DoTween 的 参数设置

DG 的链式编程 函数返回值仍然是DG本身的对象
方法本身返回对象本身 可以无限的直接使用对象的方法

```
// 这里设置了这个动画的循环为无限循环 动画的运动状态为平均运动
// from 如果不带参数说明按 DOMove(Vector3.one 方向向量运行 而不是最终点
// 如果带参数 true 就是 按相反运动进行
transform.DOMove(Vector3.one, 3).SetLoops(-1, LoopType.Incremental).SetEase(Ease.Linear).From();
// 设置动画延迟
transform.DOMove(Vector3.one, 3).SetDelay(1);
// 将动画的第二个参数 从时间 改为速度大小
transform.DOMove(Vector3.one, 3).SetSpeedBased();
// 设置 动画/运动的 编号
transform.DOMove(Vector3.one, 3).SetId("MoveID_1");
// 可以通过缓存调用动画
DOTween.Play("MoveID_1");
// 设置运动相对 将动画从运动目的地 改为运动方向 (增量运动 和from 差不多)
transform.DOMove(Vector3.one, 3).SetRelative();
// 设置动画为可回收内存
transform.DOMove(Vector3.one, 3).SetRecyclable(true);
// 设置动画帧函数
// 将动画帧函数设置为 普通的 update 模式 第二个参数: 是否不依赖unity的函数
transform.DOMove(Vector3.one, 3).SetUpdate(UpdateType.Normal, true);
```

〇七 设置运动曲线

2019年8月15日　　9:10

利用DoTween和Unity设置运动曲线

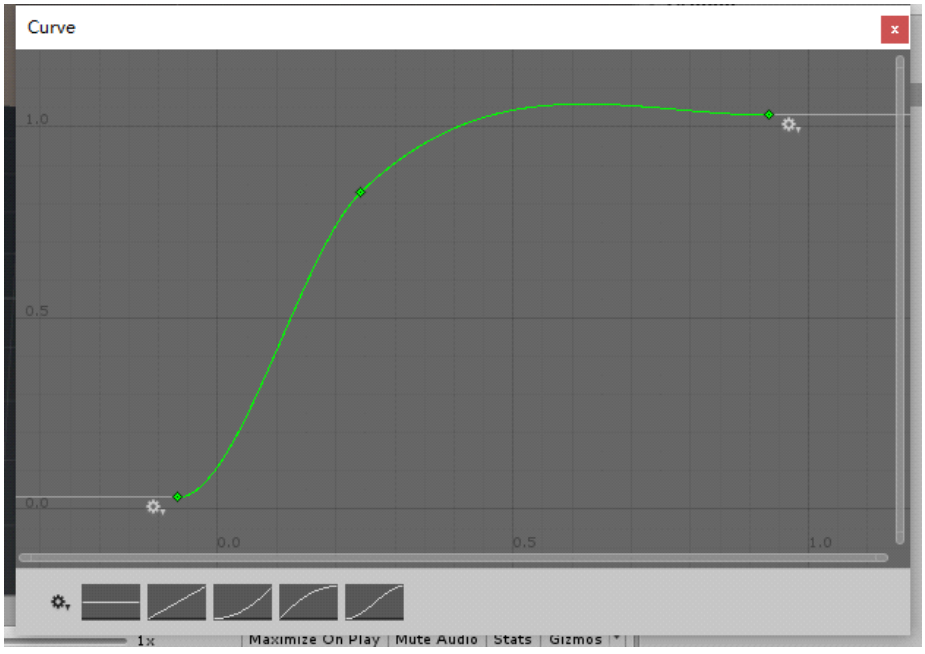
DoTween自身包含了很多关于运动曲线 速率曲线的函数
DoTween中既可以使用 setEase() 利用 EASE的运动方式来设置运动速率曲线
也可以利用Unity自带的AnimationCurve曲线编辑器来设置运动速率

DoTween设置曲线速率方法

```
// 匀速运动
//transform.DOMove(Vector3.one, 3).SetEase(Ease.Linear);
// 弹跳运动 弹跳次数 弹跳幅度
// transform.DOMove(Vector3.one * 3, 3).SetEase(Ease.Flash, 10, 3);
```

使用 Unity 的 AnimationCurve组件设置运动曲线

```
public AnimationCurve curve;
// 利用组件设置 运动曲线
transform.DOMove(new Vector3(5,0,5), 3).SetEase(curve).SetRelative();
```



利用重载 设置自定义函数返回运动曲线

```
transform.DOMove(new Vector3(5, 0, 5), 3).SetEase(EaseFun).SetRelative();
```

```
// 自定义 运动曲线 函数 return 0 - 1
private float EaseFun(float time,float duration,float overShoot,float periow)
{
    // 返回的是 运动所需时间 / 时间 = 匀速运动
    return time / duration;
}
```

〇八 回调函数

2019年8月15日 9:51

DoTween 在各种动画后的回调

```
private void CallBcak()
{
    Debug.Log("在第五秒使用了回调函数");
}
    // 回调函数
    // 运动执行完成的回调函数
    transform.DOMove(new Vector3(5, 0, 5), 3).OnComplete(() => { CallBcak(); });
    // 动画被杀死的时候调用
    transform.DOMove(new Vector3(5, 0, 5), 3).OnKill(() => { CallBcak(); });
    // 动画 第一次 刚刚开始播放的时候 调用
    transform.DOMove(new Vector3(5, 0, 5), 3).OnStart(() => { CallBcak(); });
    // 动画每次播放的时候调用
    transform.DOMove(new Vector3(5, 0, 5), 3).OnPlay(() => { CallBcak(); });
    // 动画暂定的时候调用
    transform.DOMove(new Vector3(5, 0, 5), 3).OnPause(() => { CallBcak(); });
    // 动画 在有循环的时候 每次循环完成的时候调用
    transform.DOMove(new Vector3(5, 0, 5), 3).OnStepComplete(() => { CallBcak(); });
    // 动画 在 执行帧内 调用
    transform.DOMove(new Vector3(5, 0, 5), 3).OnUpdate(() => { CallBcak(); });
```

〇九 控制函数

2019年8月15日 10:15

关于动画的控制函数

DoTween中的控制动画的函数
在 .NET 的4.6FranWork 版本中 支持异步函数和暂停

控制函数

```
//await Task.
//transform.DOPause();// 暂停
transform.DOMove(new Vector3(3, 0, 3), 3);
//await Task.Delay(TimeSpan.FromSeconds(1)); //异步调用
//transform.DOPlay(); // 播放
//transform.DORestart(); // 重启
//transform.DORewind(); // 循环播放
transform.DOSmoothRewind(); //慢慢返回之前的运动轨迹
transform.DOKill(); // 杀死当前物体的移动动画
transform.DOFlip(); //反转运动方向
// 跳转到运动时间点 是否开始播放动画
transform.DOGoto(1, true);
transform.DOPlayBackwards(); // 反向播放动画
// 如果在暂停时运行就播发 如果在播放调用就运行
transform.DOTogglePause();
```

一十 数据获取函数

2019年8月15日 10:49

DoTween 中 获取各种数据的函数

```
// 获取当前正在播放的动画
var playList = DOTween.PlayingTweens();
// 获取当前在暂停的动画
var pauseList = DOTween.PausedTweens();
// 根据动画id 得到动画 (第二个参数 是否爱播放内的动画查找)
var tweenItem = DOTween.TweensById("animtion_id",true);
// 在当前游戏对象上找 正在播放的动画
var tweenLit = DOTween.TweensByTarget(transform, true);
// 设置运动循环次数并获取当前循环次数
Tweener tweener = transform.DOMove(Vector3.one, 3).SetLoops(3);
int loopCount = tweener.CompletedLoops();
// 获取总的循环次数
int loopsCount = tweener.Loops();
// 获取当前播放的时间
float animTime = tweener.Elapsed();
// 获取动画运行进度百分比
float animDirectionalPercentage = tweener.ElapsedDirectionalPercentage();
// 获取包含循环的动画进度百分比(总历程 包含循环)
float animPercentage =  tweener.ElapsedPercentage(true);
// 获取动画是否初始化
bool isInitialized = tweener.IsInitialized();
bool isplay = tweener.IsPlaying();
```

十一 与协程函数关联

2019年8月15日 10:58

与协程进行关联返回

DoTween中包含了Unity中的协程函数 所需要的等待与返回

```
    private IEnumerator Wait()
    {
        // 按完成播放 等待返回
        yield return goTweener.WaitForCompletion();
        // 按循环播放玩 等待返回
        yield return goTweener.WaitForElapsedLoops(2);
        // 等待动画杀死后返回
        yield return goTweener.WaitForKill();
        // 等待动画播放多久后返回
        yield return goTweener.WaitForPosition(1.5f);
    }
```

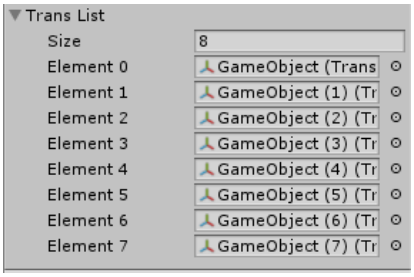
十二 路径设置

2019年8月15日 15:02

DoTween 提供设置路径

具体做法

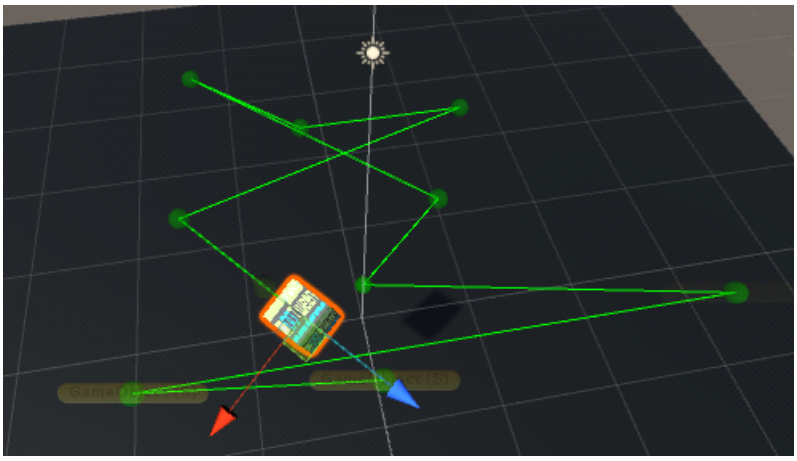
在空间内设置一定的坐标点 并在代码中获取这些点



利用 using System.Linq; 这个库 对 transform 一键转换成Position的三维向量

```
var positions = transList.Select(trans => trans.position).ToArray();

// 设置路径 点集合 运行时间 连接点的方式(曲线/直线) 运动朝向(3D/2D/不旋转) 像素点(路径之间的点个数) 路径颜色
    transform.DOPath(positions, 5, PathType.Linear, PathMode.Full3D, 50, Color.green)
        .SetOptions(true, AxisConstraint.X, AxisConstraint.X)//设置闭环路径 锁定最终方位 锁定最终旋转'
        .SetLookAt(new Vector3(0, 0, 0))//设置旋转朝向 1.lookAt Position 看向某一个位置
        .SetLookAt(transform)//2. lookAt transform 看向某一个物体 (可动可静)
        .SetLookAt(0)//3 lookAt 0 朝向路径的前方 (在没有方向和旋转的锁定的时候)
        .SetLookAt(0.3f); // 4 lookAt 0.5f 朝向路径的前方 但有偏转

    // 当 SetOptions() 为 true的时候 朝向方向 由于不是闭环 而逐渐变成 朝向路径 而不是一开始就朝向路径前
```

最后的效果 在一个连好线的轨道上连续运动



〇一 创建数据类

2019年8月16日 10:56

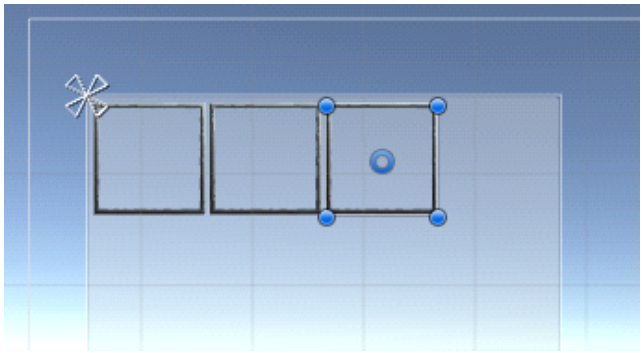
利用UGUI制作 简易的背包系统

首先建立所有的物品数据类 父类 - 所有物品的父类
包含 物品的 id name price type iconPath 等
在创建 武器类 防具类 药品类 等

〇二 创建背包Panel

2019年8月16日 16:23

创建UI 设置Canvas 的自适应 根据 屏幕(第二个) 如果是手机 则锚点在高度 端游则是在宽度
创建一个panel 给面板设置一个 Grid Layout Group 组件 用来自动排列image
设置 image 并作为一个预设 作为背包格子的背景 同时作为判断当前格子是否有物品的依据
为了方便格子计算 采用自动生成(可外部更新)



Grid Group Panel 的代码

```
public List<Transform> grids;
public int gridsCount;
public GameObject grid;

private void Awake()
{
    grids = new List<Transform>();
    for (int i = 0; i < gridsCount; i++)
    {
        GameObject go = Instantiate(grid, transform.position, Quaternion.identity);
        go.transform.parent = transform;
        grids.Add(go.transform);
    }
}
```

〇三 创建物品格子

2019年8月16日 10:56

利用Image 和text设置基本格子内容

在一个image上添加一个新的物体 以区分icon 和 内容
利用外部更新函数 来更新icon 和text