U3D 笔记

郑华

2018 年 8 月 28 日

目录

第一章　基础

入门参考：https://unity3d.com/cn/learn/tutorials

1.1　如何将脚本与具体对象绑定

1. 右键asset 文件夹，创建 C# 脚本
2. 编写脚本
3. 将asset 中的脚本拖拽到 hiearch 视图中的MainCamera 中
4. 如果脚本是作用于场景中的某个物体，**则将该脚本拖拽到该物体上**

1.2　序列化- [SerializedField]

通常情况下，GameObject 上挂的 MonoBehaviour 脚本中的**私有变量**不会显示在 *Inspector* 面板上，即不会被序列化。

但如果指定了 SerializedFiled 特性，就可以被序列化了。

```
public class Test : MonoBehaviour
{
    public string Name;
    [SerializeField]
    private int Hp;
}
```

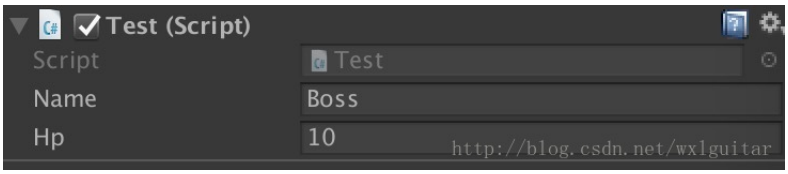


图 1.1: 序列化操作 -在 Inspector 上显示

1.3 常用技巧

- ctrl + d 复制
- shift + 鼠标 等比例缩放
- shift + alt + 鼠标 原地等比例缩放
- 在Unity 编辑器中输入汉字 需要借助其他文本拷贝粘贴
- q、w、e、r、t 在操作 UI 时尽量使用 T，以避免 z 轴发生的变化

1.4 MonoBehaviour 生命周期、渲染管线

1.4.1 脚本渲染流程

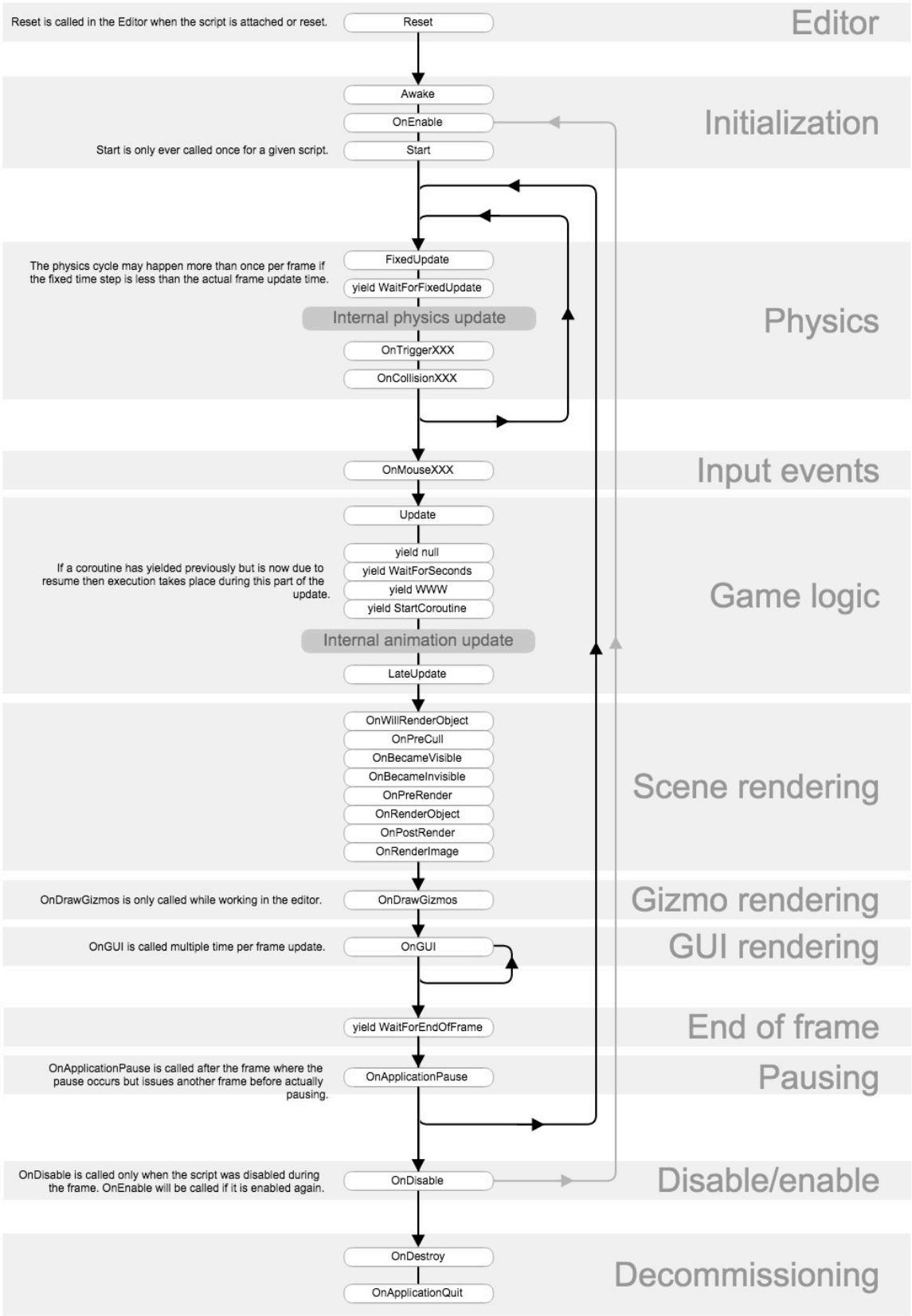


图 1.2: 脚本生命周期核心方法

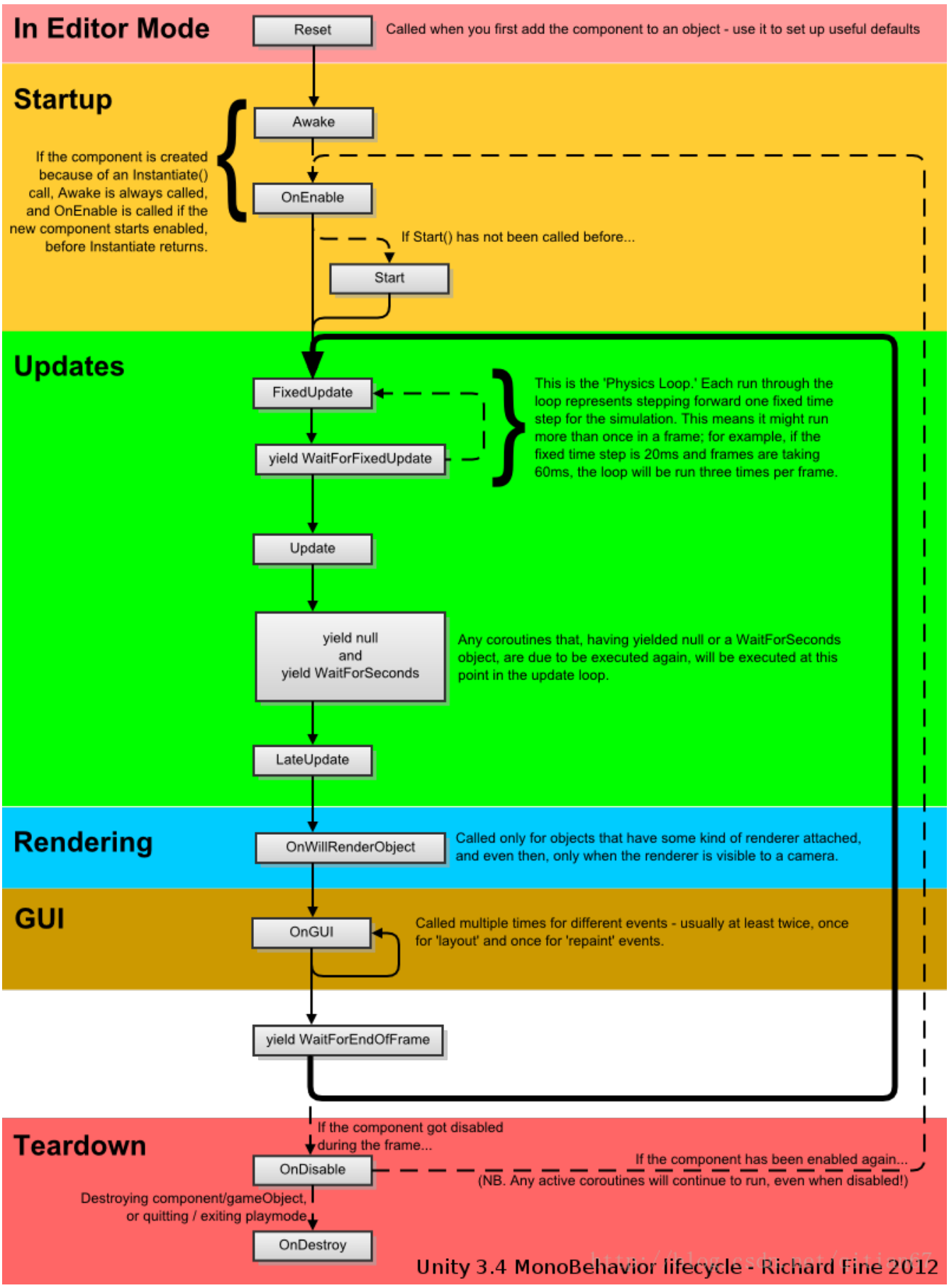

In Editor Mode
Reset
Called when you first add the component to an object - use it to set up useful defaults
Startup
Awake
If the component is created because of an Instantiate() call, Awake is always called, and OnEnable is called if the new component starts enabled, before Instantiate returns.
OnEnable
If Start() has not been called before...
Start
Updates
FixedUpdate
This is the 'Physics Loop.' Each run through the loop represents stepping forward one fixed time step for the simulation. This means it might run more than once in a frame; for example, if the fixed time step is 20ms and frames are taking 60ms, the loop will be run three times per frame.
yield WaitForFixedUpdate
Update
yield null
and
yield WaitForSeconds
Any coroutines that, having yielded null or a WaitForSeconds object, are due to be executed again, will be executed at this point in the update loop.
LateUpdate
Rendering
OnWillRenderObject
Called only for objects that have some kind of renderer attached, and even then, only when the renderer is visible to a camera.
GUI
OnGUI
Called multiple times for different events - usually at least twice, once for 'layout' and once for 'repaint' events.
yield WaitForEndOfFrame
Teardown
If the component got disabled during the frame...
OnDisable
If the component has been enabled again...
(NB. Any active coroutines will continue to run, even when disabled!)
Destroying component/gameObject, or quitting / exiting playmode
OnDestroy
Unity 3.4 MonoBehavior lifecycle - Richard Fine 2012

图 1.3: 简要核心方法

update：当其**所在的物体**属于未激活的话（active为false），该物体上所有脚本中包含的协程代码都是不会被执行的。

1.4.2 核心方法

1. Reset ：
2. Awake ：脚本唤醒函数，**当游戏对象被创建的时候**，游戏对象**绑定的脚本会在该帧 (Frame) 内**执行 *Awake() 函数*，**无论**脚本是否处于激活 (**enable**) 状态。
3. OnEnable ：激活函数，当脚本被激活时调用。
4. Start ：该函数在脚本被激活的时候执行，位于 Awake 之后，该函数同样也是在游戏对象被创建的帧里，**不同的是**，如果脚本处于不激活状态 (MonoBehaviour.enable = false), start 函数是不会执行的。
5. FixedUpdate ：
6. yield WaitForFixedUpdate ：
7. OnTriggerXXX ：
8. Update ：**只要处于激活状态的脚本**，*都会在每一帧里调用 Update() 函数*，该函数也是最为常用的一个函数，用来更新逻辑。
9. LateUpdate ：延迟更新函数
10. OnWillRenderObject ：
11. OnGUI ：绘制界面函数。
12. yield WaitForEndOfFrame ：
13. OnDisable ：
14. OnDestroy ：在当前脚本销毁时调用该函数。

1.5 Unity 委托

定义 `public delegate void MyDelegate(int num);`

委托就是C# 封装的 **C++ 的函数指针**。

定义一个委托 MyDelegate，如同定义一个类一样，此时的委托没有经过实例化是无法使用

的，而他的实例化必须接收一个返回值和参数都与他等同的函数，此处的委托 MyDelegate 只能接收返回值为 void，参数为一个 int 的函数

实例化委托 ：`MyDelegate _MyDelegate=new MyDelegate(TestMod);`

以`TestMod` 函数实例化一个`MyDelegate` 类型的委托`_MyDelegate`，此处`TestMod` 函数的定义就应如下：

```
public void TestMod(int _num);
```

之后调用`_MyDelegate(100)` 时就完全等同于调用`TestMod(100)`

1.6 Unity 协程

1.6.1 开启方式

协程：协同程序，在主程序运行的同时，开启另外一段逻辑处理，来协同当前程序的执行。

StartCoroutine(string MethodName)

- 参数是方法名
- 形参方法可以有返回值

StartCoroutine(IEnumerator method)

- 参数是方法名 (`TestMethod()`), 方法中**可以包含多个参数**
- `IEnumrator` 类型的方法不能含有`ref`或者`out` 类型的参数，**但可以含有被传递的引用**
- **必须有有返回值**，且返回值类型为`IEnumrator`, 返回值使用（*yield retuen* + 表达式或者值，或者 *yield break*）语句

1.6.2 终止方式

StopCoroutine(string MethodName) 只能终止指定的协程

StopAllCoroutine() 终止所有协程

1.6.3 yield 方式

yield return 挂起，**程序遇到**`yield` **关键字时会被挂起**，暂停执行，**等待条件满足时从当前位置继续执行**

- `yield return 0` or `yield return null`: 程序在下一帧中从当前位置继续执行
- `yield return 1,2,3,.....`: 程序等待 1，2，3... 帧之后从当前位置继续执行
- `yield return new WaitForSeconds(n)`: 程序等待 n 秒后从当前位置继续执行
- `yield new WaitForEndOfFrame()`: 在所有的渲染以及 GUI 程序执行完成后从当前位置继续执行
- `yield new WaitForFixedUpdate()`: 所有脚本中的 FixedUpdate() 函数都被执行后从当前位置继续执行
- `yield return WWW()`: 等待一个网络请求完成后从当前位置继续执行
- `yield return StartCoroutine()`: 等待一个协程执行完成后从当前位置继续执行

yield break 如果使用`yield break` 语句，将会导致**如果协程的执行条件不被满足**，*不会从当前的位置继续执行程序，而是直接从当前位置跳出函数体，回到函数的根部*

相当于：**return; + 暂停**

1.6.4 执行原理

协程函数的返回值是`IEnumerator`, 它是一个迭代器，**可以把它当成执行一个序列的某个节点的指针**，它提供了两个重要的接口，分别是`Current`(返回当前指向的元素) 和`MoveNext()`(将指针向后移动一个单位，*如果移动成功，则返回 true*)

`yield` 关键词**用来声明序列中的下一个值或者是一个无意义的值**,如果使用`yield return x`(x 是指一个具体的对象或者数值) 的话，那么`MoveNext` 返回为`true` 并且`Current` 被赋值为`x`, 如果使用`yield break` 使得`MoveNext()` 返回为`false`

如果`MoveNext` 函数返回为`true` **意味着协程的执行条件被满足，则能够从当前的位置继续往下执行**。否则不能从当前位置继续往下执行。

委托 + 协程 https://blog.csdn.net/qq992817263/article/details/51514449

- 实现延时
- 实现给定函数传参
- 实现特定功能

```
// 延时执行

// <param name="action">执行的委托</param>
// <param name="obj">委托的参数</param>
// <param name="delaySeconds">延时等待的秒数</param>
public IEnumerator DelayToInvokeDo(Action<GameObject> action, GameObject obj,float
    delaySeconds)
{
    yield return new WaitForSeconds(delaySeconds); // delaySeconds 后执行
    action(obj); // 特定功能
}
// 使用例子
StartCoroutine(
    DelayToInvokeDo(
        delegate(GameObject task) {
            task.SetActive(true);
            task.transform.position = Vector3.zero;
            task.transform.rotation = Quaternion.Euler(Vector3.zero);
            task.doSomethings();
        },
        /*传参*/GameObject.Find("task1"),
        1.5f)/*End 匿名委托*/
    );/*End 协程初始*/
```

第二章　事件

2.1　必然事件

继承自MonoBehaviour 类后，自动会**按序**提供以下方法:

- Awake(): 在加载场景时运行，用于在游戏开始前完成变量初始化、以及游戏状态之类的变量。
- Start(): 在第一次启动游戏时执行，用于游戏对象的初始化，在Awake() 函数之后。
- Update(): 是在每一帧运行时必须执行的函数，用于更新场景和状态。
- FixedUpdate(): 与Update() 函数相似，但是在固定的物理时间后间隔调用，用于物理状态的更新。
- LateUpdate(): 是在Update() 函数执行完成后再次被执行的，有点类似收尾的东西。

2.2　碰撞事件

U3D 的碰撞检测。具体分为三个部分进行实现，碰撞发生进入时、碰撞发生时和碰撞结束，理论上不能穿透

- OnCollisionEnter(Collision collision) 当碰撞物体间刚接触时调用此方法
- OnCollisionStay(Collision collision) 当发生碰撞并保持接触时调用此方法
- OnCollisionExit(Collision collision) 当不再有碰撞时，既从有到无时调用此函数

2.3　触发器事件

类似于红外线开关门，有个具体的范围，然后进入该范围时，执行某种动作，离开该范围时执行某种动作。类似于物体于一个透明的物体进行碰撞检测，理论上需要穿透，在 U3D 中通过

勾选 Is Trigger 来确定该物体是可以穿透的。

- OnTriggerEnter() 当其他碰撞体进入触发器时，执行该方法
- OnTriggerStay() 当其他碰撞体停留在该触发器中，执行该方法
- OnTriggerExit() 当碰撞体离开该触发器时，调用该方法

第三章　实体-人物、物体、组件

3.1　实体类

GameObject 类, 游戏基础对象，用于填充世界。

复制　Instantiate(GameObject) 或 Instantiate(GameObject, position, rotation)

- GameObject 指生成克隆的**游戏对象**，也可以是 **Prefab 的预制品**
- position 克隆对象的初始位置，类型为Vector3
- rotation 克隆对象的初始角度，类型为Quaternion

销毁　Destroy(GameObject xx)- 立即销毁 或Destroy(GameObject xx, Time time)- 几秒后销毁

可见否　通过设置该参数调整该实体是否可以在游戏中显示,具体设置方法为gameObject.SetActive(true

游戏中获取

1. 在整个场景中寻找名为 xx 的游戏对象，并赋予 obj 变量

   obj = GameObject.Find("xx");

2. 当需要获取某个gameObject 下的组件时, 使用Transform.Find.GetComponent

   gameObjVar1.transform.Find("ImageItemIcon/TextMonthCardLeftDays").\label{GetComponen\

3. 但是设置为激活状态则需要如下通过gameObject 进行设置。

   obj = gameObjVar1.transform.Find("xx/xx").gameObject.SetActive(true);

3.2 Prefabs -预设体

prefabs 基础：https://www.cnblogs.com/yuyaonorthroad/p/6107320.html

动态加载 Prefabs：https://blog.csdn.net/linshuhe1/article/details/51355198

在进行一些功能开发的时候，我们常常将一些**能够复用的对象**制作成.**prefab 的预设物体**，然后将预设体存放到 Resources 目录之下，使用时再动态加载到场景中并进行实例化。例如：子弹、特效甚至音频等，都能制作成预设体。

概念 组件的集合体, 预制物体可以实例化成游戏对象.

作用 可以重复的创建具有相同结构的游戏对象。

3.2.1 预设动态加载到场景

预设体资源加载 ->

假设预设体的位置为下图所示

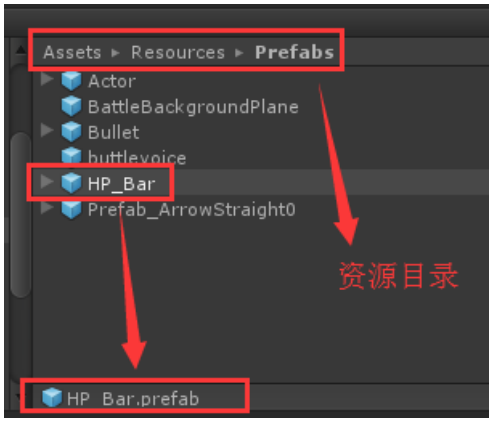


图 3.1: Prefab 资源位置

```
//加载预设体资源
GameObject hp_bar = (GameObject)Resources.Load("Prefabs/HP_Bar");
```

通过上述操作，实现从资源目录下**载入**HP_Bar.prefab **预设体，用**一个GameObject **对象来存放**，此时该预设物体并未真正载入到场景中，因为还未进行实例化操作。

预设体实例化 ->

实例化使用的是MonoBehaviour.Instantiate 函数来完成的，**其实质就是从预设体资源中克隆出一个对象**，它具有与预设体完全相同的属性，并且被加载到当前场景中

完成以上代码之后，在当前场景中会出现一个实例化之后的对象，并且其父节点默认为当场的场景最外层，如下图所示。

图 3.2: Prefab 实例后位置

实例化对象属性设置 ->

完成上述步骤之后，我们已经可以在场景中看到实例化之后的对象，但是通常情况下我们**希望**我们的**对象之间层次感分明**，而且这样也方便我们进行对象统一管理，而不是在 Hierarchy 中看到一大堆并排散乱对象，所以我们**需要为对象设置名称以及父节点等属性**。

-->Notice：常见错误：对**未初始化**的hp_bar 进行属性设置，**设置之后的属性在实例化之后无法生效**。这是因为我们最后在场景中**显示的其实并非实例化前的资源对象，而是一个克隆对象**，所以假如希望设置的属性在最后显示出来的对象中生效，我们需要对实例化之后的对象进行设置。

正确的设置代码如下，可以看到**实例化对象**已成功挂在到**父节点 Canvas** 上，在层次视图效果如下图所示：

```
GameObject hp_bar = (GameObject)Resources.Load("Prefabs/HP_Bar");

//搜索画布的方法!
GameObject mUICanvas = GameObject.Find("Canvas");
hp_bar = Instantiate(hp_bar);
hp_bar.transform.parent = mUICanvas.transform;
```

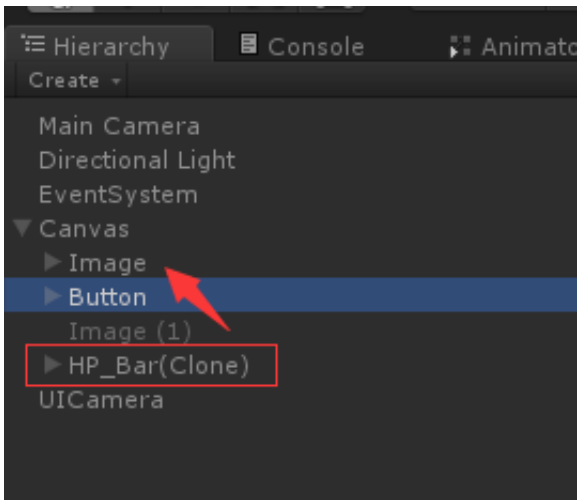


图 3.3: Prefab 对象设置父子关系

简化写法 上述实例步骤与属性设置代码可以简化为

```
GameObject hp_bar = (GameObject)Instantiate(Resources.Load("Prefabs/HP_Bar"));
GameObject mUICanvas = GameObject.Find("Canvas");
hp_bar.transform.parent = mUICanvas.transform;
```

预制体添加脚本 在预制体上不能直接添加脚本，首先需要将其拖入场景，然后再对其操作，这个时候可以添加脚本，添加组件等，在完成这些操作后，在 Inspector 选项中选中 Apply, 然后删除其在场景中的刚才拖过来的，即可。

3.3 获取实体上的组件

调用方式 `GameObject.GetComponent<Type>().xx = xx;`

- `cube1.GetComponent<RigidBody>().mass = 20;` //设置重量
- `cube1.GetComponent<BoxCollider>().isTrigger = true;` //**开启 Trigger 穿透方式**检测
- `cube2.GetComponent<Test>().enable = false;` //**禁用 Test 脚本**

3.4 物理作用实体类

`Rigidbody` 类，一种特殊的游戏对象，该类对象可以在物理系统的控制下来运动。

AddForce() 此方法调用时`rigidBody.AddForce(1, 0, 0);`，会施加给刚体一个瞬时力，在力的作用下，会产生一个加速度进行运动。

AddTorque() 给刚体添加一个扭矩。

Sleep() 使得刚体进入休眠状态，且至少休眠一帧。类似于暂停几帧的意思，这几帧不进行更新、理论位置也不进行更新。

WakeUp() 使得刚体从休眠状态唤醒。

第四章　世界空间相关 3D 基础

4.1　Transform 类

https://blog.csdn.net/yangmeng13930719363/article/details/51460841

4.1.1　位置

transform.position = new Vector3(1, 0, 0);

4.1.2　旋转

transform.Rotate(x, y, z);

transform.eulerAngles = new Vector3(x, y, z);

4.1.3　缩放

transform.localScale(x, y, z); // 基准为 1、1、1，数为缩放因子。

4.1.4　平移

transform.Translate(x, y, z);

4.1.5　Transform.localPosition

`position` 是世界坐标中的位置，可以理解为绝对坐标

`localPosition` 是**相对于父对象的位置**，是相对坐标，既父级窗体为原点坐标

4.1.6 注意

在变化的过程中需要乘以 Time.deltaTime , 否则会出现大幅不连贯的画面。

4.2 摄像机 -Camera

4.2.1 Clear Flags

清除标记。决定屏幕的哪部分将被清除。一般用户使用对台摄像机来描绘不同游戏对象的情况，有 3 中模式选择：

- `Skybox`：天空盒。默认模式。在屏幕中的空白部分将显示当前摄像机的天空盒。如果当前摄像机没有设置天空盒，会默认用 Background 色。
- `Solid Color`：纯色。选择该模式屏幕上的空白部分将显示当前摄像机的 background 色。
- `Depth only`：仅深度。该模式用于游戏对象不希望被裁剪的情况。
- `Dont Clear`：不清除。该模式不清除任何颜色或深度缓存。其结果是，每一帧渲染的结果叠加在下一帧之上。一般与自定义的 shader 配合使用。

4.2.2 Culling Mask -剔除遮罩

剔除遮罩，选择所要显示的`layer`, **摄像机将看到勾选的层，忽略未被勾选的层。**

4.2.3 Projection -透视模式

透视 摄像机模式，**截锥体**

正交 前后显示一样，不存在远小近大的样子。**长方体**

4.2.4 Clipping Planes -裁剪模式

剪裁平面。摄像机开始渲染与停止渲染之间的距离。

4.2.5 Viewport Rect

标准视图矩形。用四个数值来控制摄像机的视图将绘制在屏幕的位置和大小，使用的是屏幕坐标系，数值在 0 1 之间。坐标系原点在左下角。

4.2.6 Depth -控制渲染顺序

深度。用于控制摄像机的渲染顺序，较大值的摄像机将被渲染在较小值的摄像机之上。

4.2.7 Rendering Path -渲染路径

渲染路径。用于指定摄像机的渲染方法。

Use Player Settings：使用Project Settings-->Player 中的设置。

Forward：快速渲染。摄像机将所有游戏对象将按每种材质一个通道的方式来渲染。

Defferred: 延迟光照 Legacy Vertex Lit：顶点光照。摄像机将对所有的游戏对象座位顶点光照对象来渲染。

Legacy Deferred Lighting：延迟光照。摄像机先对所有游戏对象进行一次无光照渲染，用屏幕空间大小的 Buffer 保存几何体的深度、法线已经高光强度，生成的 Buffer 将用于计算光照，同时生成一张新的光照信息 Buffer。最后所有的游戏对象会被再次渲染，渲染时叠加光照信息 Buffer 的内容。

4.2.8 Target Texture -目标纹理

用于将摄像机视图输出并渲染到一张贴图sss。一般用于制作导航图或者画中画等效果。

4.2.9 HDR -高动态光照渲染

高动态光照渲染。用于启用摄像机的高动态范围渲染功能。

4.3 3D 模型

4.3.1 Mesh

4.3.2 Texture

4.3.3 Material

尽管是近似的灰色，也同样会因为材质的不同显示出不同的效果，入灰色 T 恤衫和灰色不锈钢，首先是他们对光照的反应不同（漫反射、平面），其次是表面的各种属性。

4.3.4 骨骼动画

第五章　键盘鼠标控制

5.1　普通按键 -keyDown(KeyCode xx)

方式一

- 定义按键码：KeyCode keycode;
- 判断键是否被按下：if(Input.GetKeyDown(keycode)){}
- 在Inspirit -> Keycode 指定关联按键

方式二

- 在Update 中更新添加如下代码
- if(Input.GetKeyDown(KeyCode.UpArrow))
- KeyCode.xx 包括了键盘所有的按键,常用的 AWSD 如下,鼠标同 (Input.GetMouseButtonDown(0) 0 左键，1 右键)
  - if (Input.GetKeyDown(KeyCode.S)) 按下 S 键
  - if (Input.GetKey(KeyCode.S)) 按住 S 键
  - if (Input.GetKeyUp(KeyCode.W)) 抬起 S 键

5.2　根据输入设备 -getAxis()

参数分为两类：

一、触屏类

1. Mouse X 鼠标沿屏幕 X 移动时触发 Mouse Y 鼠标沿屏幕 Y 移动时触发 Mouse ScrollWheel 鼠标滚轮滚动是触发

```
float mouseX = Input.GetAxis("Mouse X");
float mouseY = Input.GetAxis("Mouse Y");

transform.Rotate(Vector3.Up * mouseX * rotateSpeed); // 根据具体需求进行操作
```

二、键盘类

1. Vertical 键盘按上或下键时触发
2. Horizontal 键盘按左或右键时触发

```
float horizontal = Input.GetAxis("Horizontal");
float vertical = Input.GetAxis("Vertical");

Vector3 desPos = (transform.forward * vertical + transform.right * horizontal) *
    Time.deltaTime * moveSpeed;

_rigidBody.position += desPos;
```

返回值是一个数，正负代表方向

第六章　时间

6.1　Time 类

该类是 U3D 在游戏中获取时间信息的接口类。常用变量如下：

表 6.1: 时间变量对照表

变量名	意义
time	单位为秒
`deltaTime`	从上一帧到当前帧消耗的时间
fixedTime	最近 FixedUpdate 的时间，从游戏开始计算
`fixedDeltaTime`	物理引擎和 FixedUpdate 的更新时间间隔
timeSceneLevelLoad	从当前 Scene 开始到目前为止的时间
realTimeSinceStartup	程序已经运行的时间
`frameCount`	已经渲染的帧的总数

第七章　数学

7.1　Random 类

随机数类

7.2　Mathf 类

数学类

7.3　坐标系

左右手坐标系：http://www.cnblogs.com/mythou/p/3327046.html

7.4　向量计算

7.5　矩阵计算

第八章　光照

8.1　光照

8.2　烘培

简介　只有静态场景才能完成烘培（Bake）操作，其目的是在游戏编译阶段完成光照和阴影计算，然后以贴图的形式保存在资源中，以这种手段避免在游戏运行中计算光照而带来的 CPU 和 GPU 损耗。

- **如果不烘培：**游戏运行时，这些阴影和反光是由 CPU 和 GPU 计算出来的。
- **如果烘焙：**游戏运行时，直接加载在编译阶段完成的光照和阴影贴图，这样就不用再进行计算，节约资源。

流程

第九章　寻路

9.1　简介

NPC 完成自动寻路的功能。

9.2　流程

- 将静态场景调至 (Navigation Static)
- 烘焙
- 添加 `Navigation Mesh Agent` 寻路组件
- 在脚本中设置组件的目标地址，添加目标

第十章　UGUI

在脚本中使用时记得加上`using UnityEngine.UI`

https://blog.csdn.net/wangmeiqiang/article/category/6364468

10.1　Spirit

在 UI 系统中，**所有的图片的显示**都必须 通过 *Spirit*。

如果建立工程时选择的是 2D 工程，那么导入的所有图片会自动设置为 *Spirit*。

如果建立工程时选择的是 3D 工程,那么导入的所有图片需要手动的设置为 *Spirit*。`Inspector -> Textu`
最后点击 Apply 保存更改。

10.2　Canvas

Canvas 画布**是承载所有 UI 元素的区域**。`Canvas` 实际上**是一个游戏对象上绑定了 Canvas 组件**。

所有的 UI 元素都**必须是 Canvas 的子对象**。如果场景中没有画布，那么我们创建任何一个 UI 元素，都会自动创建画布，并且将新元素置于其下。

在 Canvas 的`Render Mode` 中有三个选择：

1. Screen Space - Overlay 屏幕最上层，主要是 2D 效果。
2. Screen Space - Camera 绑定摄像机，可以实现 3D 效果。
3. World Space 世界空间，让 UI 变成场景中的一个物体。

10.2.1 Screen Space-Overlay -覆盖模式

Screen Space-Overlay（屏幕控件-覆盖模式）的**画布会填满整个屏幕空间**，并将画布下面的所有的 UI 元素置于屏幕的最上层，或者说**画布的画面永远“覆盖”其他普通的 3D 画面**，*如果屏幕尺寸被改变，画布将自动改变尺寸来匹配屏幕*

Screen Space-Overlay 模式的画布有 Pixel Perfect 和 Sort Layer 两个参数：

1. `Pixel Perfect`：只有`RenderMode` 为 Screen 类型时才有的选项。使 UI 元素像素对应，效果就是**边缘清晰不模糊**。

2. `Sort Layer`: `Sort Layer` 是 UGUI 专用的设置，用来指示**画布的深度**。

10.2.2 Screen Space-Camera -摄像机模式

与 Screen Space-Overlay 模式类似，画布也是**填满整个屏幕空间**，如果屏幕尺寸改变，**画布也会自动改变尺寸来匹配屏幕**。

不同的是，在该模式下，**画布会被放置到摄影机前方**。在这种渲染模式下，**画布看起来绘制在一个与摄影机固定距离的平面上**。*所有的 UI 元素都由该摄影机渲染，因此摄影机的设置会影响到 UI 画面*。在此模式下，UI 元素是由`perspective` 也就是视角设定的，视角广度由`Filed of View` 设置。

这种模式可以用来实现在 UI 上显示 3D 模型的需求，比如很多 MMO 游戏中的查看人物装备的界面，可能屏幕的左侧有一个运动的 3D 人物，左侧是一些 UI 元素。通过设置 Screen Space-Camera 模式就可以实现上述的需求，效果如下图所示：

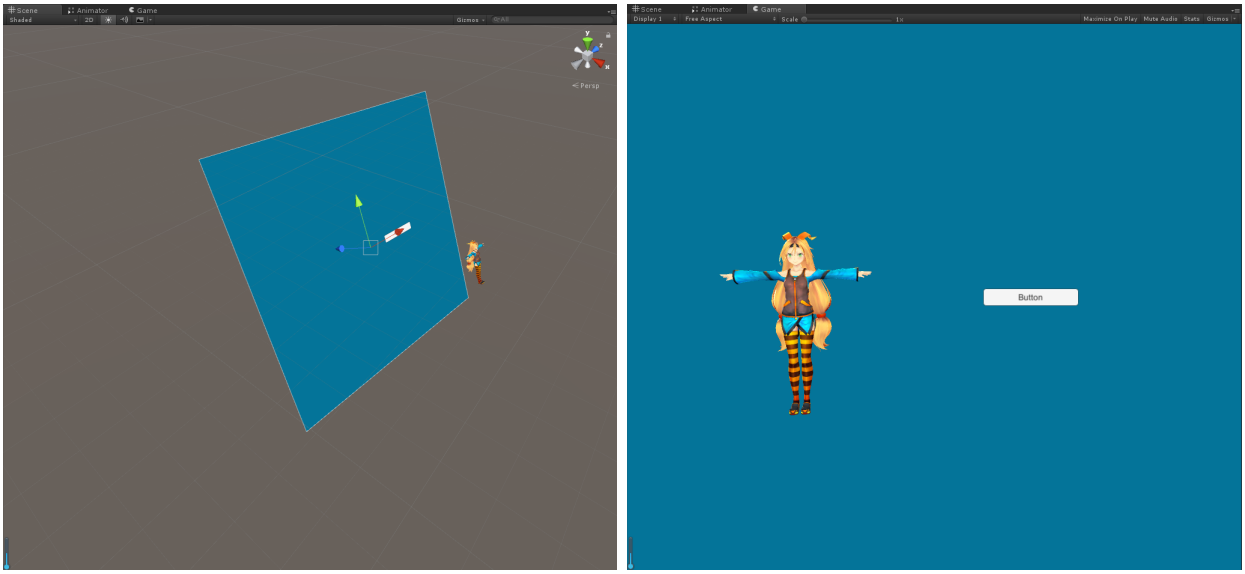


图 10.1: 摄像机模式-画布

它比 Screen Space-Overlay 模式的画布多了下面几个参数：

1. `Render Camera`: 渲染摄像机

2. `Plane Distance`: **画布距离摄像机的距离**

3. `Sorting Layer`: Sorting Layer 是 UGUI 专用的设置，**用来指示画布的深度**。可以通过点击该栏的选项，在下拉菜单中点击“Add Sorting Layer”按钮进入标签和层的设置界面，或者点击导航菜单->edit->Project Settings->Tags and Layers 进入该页面。

4. `Order in Layer`: **在相同的 Sort Layer 下的画布显示先后顺序**。数字越高，显示的优先级也就越高。

10.2.3 World Space -世界空间模式

World Space 即世界空间模式。在此模式下，**画布被视为与场景中其他普通游戏对象性质相同的类似于一张面片（Plane）的游戏物体**。

画布的尺寸可以**通过 RectTransform 设置**，所有的 UI 元素可能位于普通 3D 物体的前面或者后面显示。**当 UI 为场景的一部分时，可以使用这个模式**。

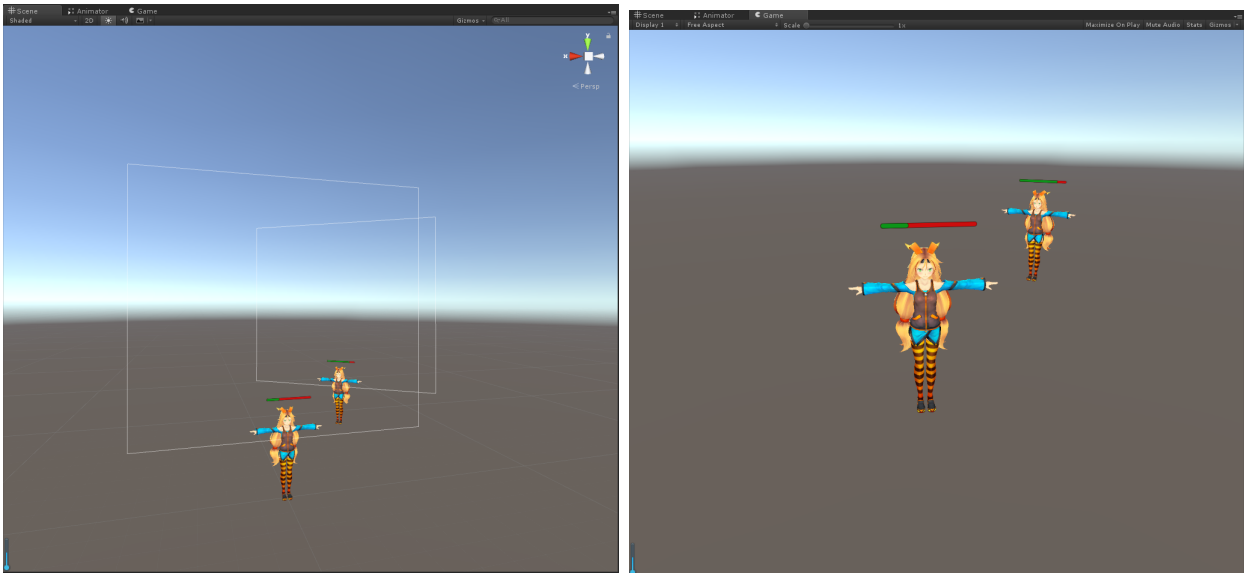


图 10.2: 世界空间模式- 画布

10.2.4 使用总结

表 10.1: 渲染模式使用场景说明

渲染模式	画布匹配屏幕?	摄像机?	像素对应	适应
覆盖-overlay 模式	是	不需要	可选	2D
摄像机-camera 模式	是	需要	可选	2D+3D
世界空间-world 模式	否	需要	不可选	3D

10.2.5 Canvas Scalar

https://blog.csdn.net/qq168213001/article/details/49744899

10.2.6 Layer

10.3 RectTransform

https://blog.csdn.net/jk823394954/article/details/53861539

https://blog.csdn.net/rickshaozhiheng/article/details/51569073

https://blog.csdn.net/serenahaven/article/details/78826851

核心看：https://blog.csdn.net/Happy_zailing/article/details/78835482

http://lib.csdn.net/article/unity3d/36875

RectTransform 继承自 Transform，又增加锚点、中心轴点等信息，**主要提供一个矩形的位置、尺寸、锚点和中心信息以及操作这些属性的方法**，同时提供多种基于父级 *RectTransform* 的缩放形式。

10.3.1 Pivot(中心)

Pivot 用来指示一个`RectTransform`（或者说是矩形）的中 (重) 心点。

10.3.2 锚点- 自适应屏幕

http://www.bubuko.com/infodetail-2384845.html

锚点（四个）由两个Vector2 的向量确定，这两个向量确定两个点，归一化坐标分别是Min和Max，**再由这两个点确定一个矩形，这个矩形的四个顶点就是锚点**。

在Hierarchy 下新建一个 Image，查看其Inspector。

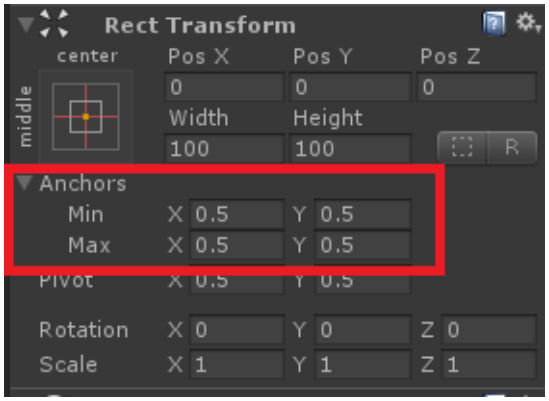


图 10.3: Anchor 属性

在 Min 的 x、y 值分别小于 Max 的 x、y 值时，Min 确定矩形**左下角**的归一化坐标，Max 确定矩形右上角的归一化坐标。

刚创建的 Image，其Anchor的默认值 为Min（0.5，0.5）和Max（0.5，0.5）。也就是说，Min和Max 重合了，四个锚点合并成一点。锚点在 Scene 中的表示如下：

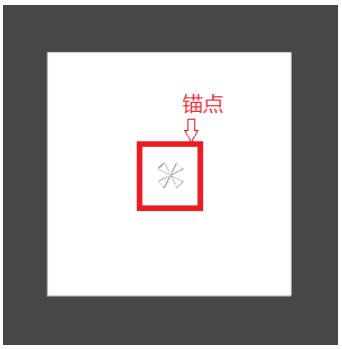


图 10.4: 锚点初始位置

将 Min 和 Max 的值分别改为（0.4，0.4）和（0.5，0.5）。可以看见四个锚点已经分开了。

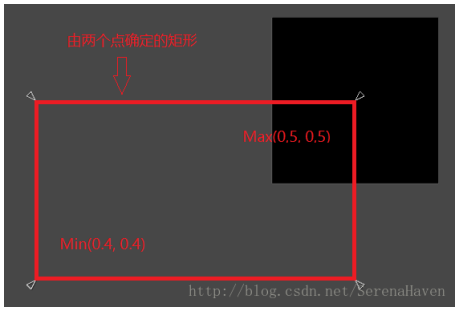


图 10.5: min Max 位置、确定矩形

需要注意 在不同的 Anchor 设置下，控制该 RectTransform 的变量是不同的。

比如设置成全部居中（默认）时，属性里包含熟悉的用来描述位置的`PosX`、`PosY`和`PosZ`，以及用来描述尺寸的`Width`和`Height`；

切换成全部拉伸时，属性就变成了`Left`、`Top`、`Right`、`Bottom` 和`PosZ`，**前四个属性**用来描述该 RectTransform **分别离父级各边的距离**，PosZ 用来描述该 RectTransform 在世界空间的 Z 坐标

锚点类型

- 位置类型 左上角、中心等
- 拉伸类型 纵向拉伸适配、横向、整体

锚点在一块的时候

- Anchor 是打在父级窗体上的
- Anchor 的位置在父级窗体上的标记方式是按照百分比记录的，单位（百分比）
- Anchor 的`Min(RectTransform.anchorMin)` `Max(RectTransform.anchorMax)` 的信息保持一致
- 子物件的坐标系为纵向 Y, 横向 X, 并且以`Anchor` 为原点，自身坐标用中心轴点`Pivot` 表示
- 子物件的 Pivot 与 Anchor 位置始终保持不变，单位（像素）

锚点单向（横或者纵）分开的时候

- 分开的部分 (拉伸方向) 与父级窗体保持一致变化，单位（百分比）
- 与相对方向则绝对保持，单位（像素）

锚点双向分开的时候

- 双向都与父级窗体保持一致的变化，单位（百分比）
- 上-top、下-bottom、左、右边距绝对保持，单位（像素）

anchorMax、anchorMin `anchorMin.x` 表示锚点在x 轴的起始点位置，`anchorMax.x` 表示锚点在 x 轴的终点位置，取值0~1，表示**百分比值**，该值乘以父窗口的`width` 值就是实际锚点相对于父窗口 x 轴的位置。y 轴与 x 轴同理。

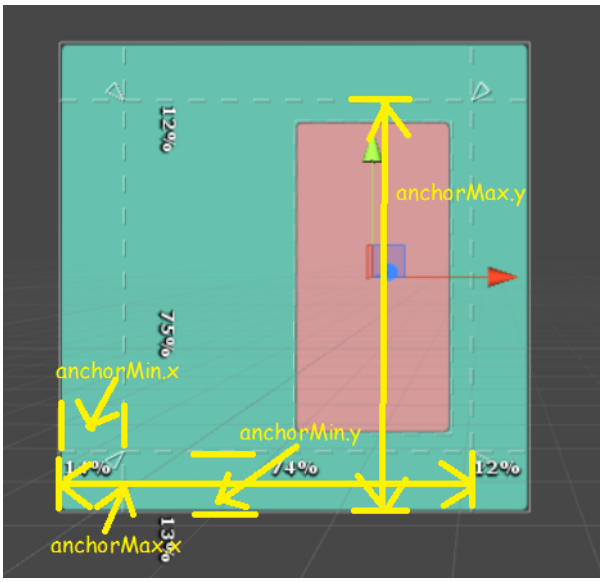


图 10.6: Anchor.Min 与 Anchor.Max

这个值确定了锚点相对于父窗口的位置，是**真正决定锚点位置的值**

offsetMax 和 offsetMin 属性

锚点分开时 在锚点分开的状态下：锚点其实是四个钉子，分为左上，左下，右下及右上四个，每个空间在 UI 模型中都是一个矩形，也有左上，左下，右下及右上四个顶点，那么锚点的每个钉子可以关联一个点，即左上————左上；左下————左下；右下————右下；右上————右上。这样进行绑定。

`offsetMax` 是 RectTransform **右上角相对于右上 Anchor 的距离**；

`offsetMin` 是 RectTransform **左下角相对于左下 Anchor 的距离**。

`offset` 可以认为是以像素为单位。

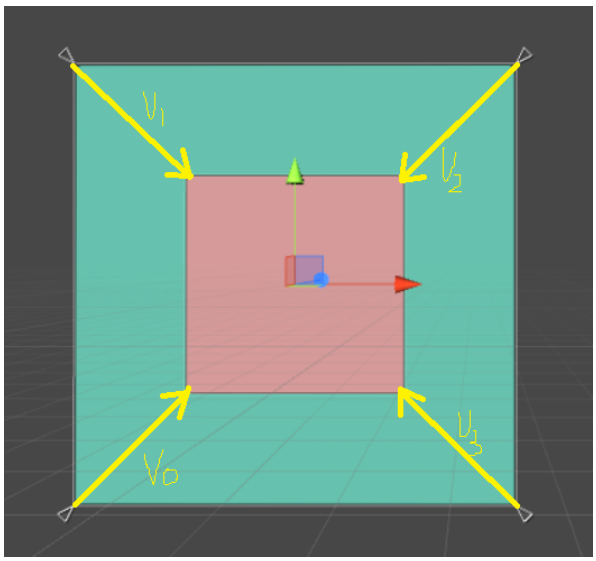


图 10.7: 锚点在一起时 Offset 求取向量示例

锚点在一处时　锚点 offset 计算如下：

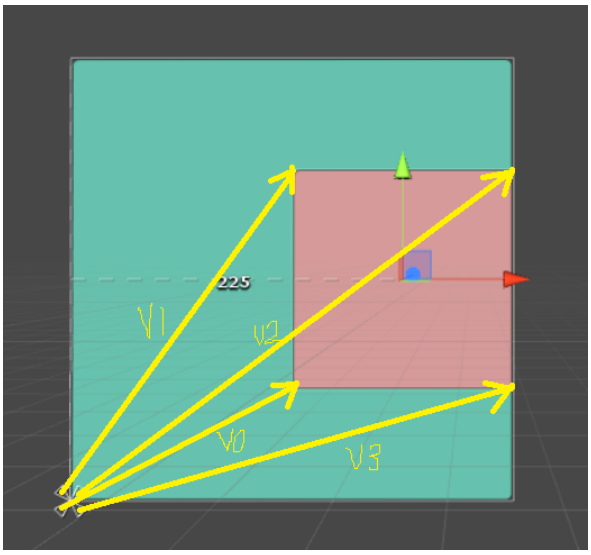


图 10.8: 锚点分开时 Offset 求取向量示例

求取　首先计算锚点的每个钉子到其对应的顶点矢量值，分别记作`v0`，`v1`，`v2`，`v3`，入上图。

然后比较四个向量的`x` 值，将`x` 的最大值赋给`offsetMax.x`，将`x` 的最小值赋给`offsetMin.x`；`y` 的值同理。

anchoredPosition

锚点在一处时　anchorPosition 就是 **从锚点**到本物体的**轴心**（Pivot）的**向量值**.

锚点分开时

10.3.3 sizeDelta

sizeDelta 是offsetMax-offsetMin 的结果。在锚点全部重合的情况下，它的值就是面板上的（Width，Height）。

在锚点完全不重合的情况下，它是相对于父矩形的尺寸。

一个常见的错误是，当 RectTransform 的锚点**并非全部重合时**，使用 sizeDelta 作为这个 RectTransform 的尺寸。此时拿到的结果一般来说并非预期的结果。

10.3.4 RectTransform.rect

RectTransform.rect 的各值如图所示。

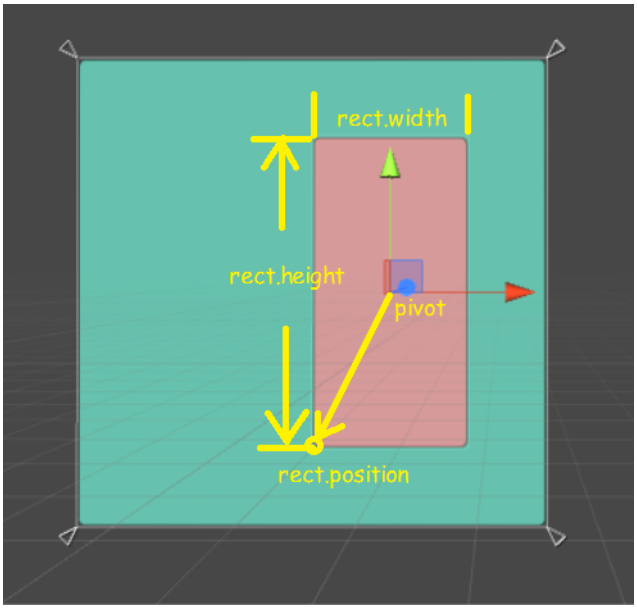


图 10.9: RectTransform rect 属性

10.3.5 示例

```
GameObject webText = new GameObject("webText");
webText.AddComponent<UnityEngine.UI.Text>();
webText.GetComponent<UnityEngine.UI.Text>().text = "";
webText.GetComponent<RectTransform>().anchorMin = new Vector2(0, 0);
webText.GetComponent<RectTransform>().anchorMax = new Vector2(1, 1);
webText.GetComponent<RectTransform>().sizeDelta = new Vector2(0, 0);
```

```
webText.GetComponent<RectTransform>().anchoredPosition = new Vector2(0, 0);
webText.transform.localPosition = new Vector3(0,0,0);
webText.transform.SetParent(webObj.transform, false);
```

10.3.6 FramDebug

查看渲染的先后顺序

`windows->FrameDebug`

10.4 按钮

10.4.1 RayCast Target-点击事件的获取原理

参考文献：https://blog.csdn.net/liujunjie612/article/details/55097789

UGUI 会遍历屏幕中所有 RaycastTarget 是 true 的 UI，接着就会发射线，并且排序找到玩家最先触发的那个 UI，在抛出事件给逻辑层去响应。

团队多人在开发游戏界面，很多时候都是复制黏贴，比如上一个图片是**需要响应** Raycast-Target，然后 ctrl+d 以后复制出来的也就带了这个属性，很可能新复制出来的图片是**不需要响应的**，开发人员**又没有取消勾选掉**，这就出问题了。

所以 RaycastTarget 如果被勾选的过多的话，效率必然会低。

10.4.2 原始 Button

10.4.3 Image 等 -添加 button 组件

- `create -> UI -> Image`
- `Inspirit -> Add Component -> button`

10.4.4 添加事件处理脚本

- 书写脚本并添加到 Button gameObject 上

- 如果是 Button 组件的话直接在 button 组件上添加，如果是 Image 则添加 button 组件后再添加
- 添加脚本对象到`onClick()` 部分：`+ -> gameObject` 拖进来 `-> 选择脚本中的具体函数`

10.5 文本- Text

10.5.1 添加文字阴影 -shadow 组件

```
addComponent -> shadow
```

10.5.2 添加文子边框 -outline 组件

```
addComponent -> outline
```

10.6 图片- ImageView

10.7 选中标记- Toggle

Toggle 基本

Toggle Group

选项栏设定 将 panel 拖入 toggle 中的`value changed` 部分

预设 确定默认打开哪个 panel，然后将其`IsOn` 勾选，其余取消勾选

10.8 滚动区域、滚动条

10.9 其他工具条

10.10 布局- Layout

- 具体页面下创建空物体 `GameObject`
- 其次在`GameObject` 下添加组件 `-> grid layout group`
- 最后在这个`GameObject` 下创建出各种 Image 组件，然后这些组件将会以`grid layout` 的布局进行自动调整

10.10.1 grid layout group

- 调整`cell size` 进行调整子物件的大小
- `cell size` 的改变只影响子组件的第一层，既最下面一层

10.10.2 horizontal layout group

10.10.3 vertical layout group

第十一章　物理

11.1　流程

- RigidBody：创建，以完成受力接收。
- Physical Material：创建，以完成多种力的添加。
- Material：拖入材质球。

11.2　刚体

11.3　碰撞器

11.4　物理材质

11.5　触发器

11.6　射线

11.7　关节

第十二章　动画

12.1　流程

- 在要动态显示的物体的父节点上创建 **Animation**
  - 编辑 Animation
    - 在脚本中获取该父亲节点上的 Animator 组件
      - 播放动画

12.2　iTween 动画用法

第十三章　粒子系统

第十四章　着色器

入门参考网址：https://blog.csdn.net/ring0hx/article/details/46440037

14.1　Shader 版本

Shader 1.0（DirectX8.0）、Shader 2.0（DirectX9.0b）、Shader 3.0（DirectX9.0c）、Shader 4.0（DirectX10）、Shader 4.1（DirectX10.1）和 Shader 5（DirectX11）。

14.2 结构

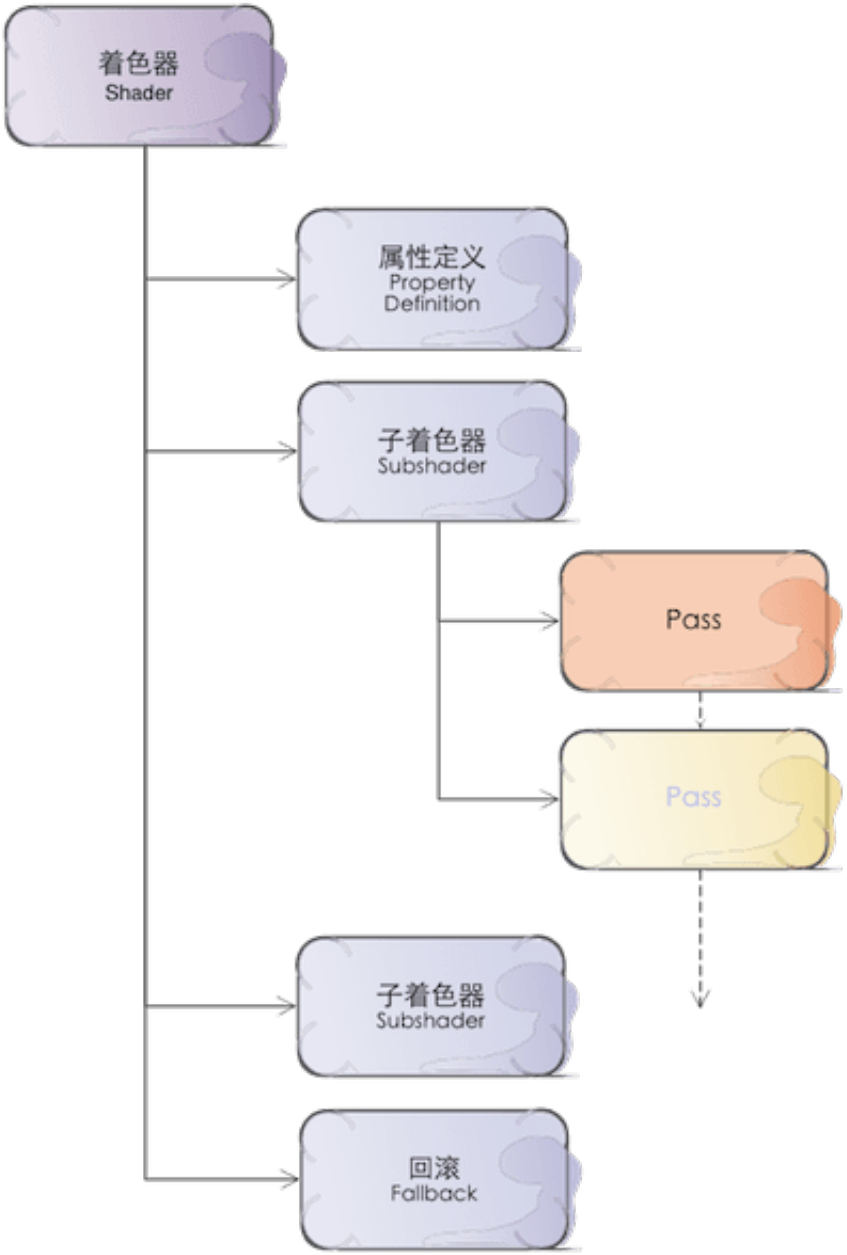


图 14.1: 着色器结构

- **属性定义（Property Definition）：**定义 Shader 的输入，并且绑定到编辑器上。
- **子着色器（SubShader）：**一个 Shader 可以有多个子着色器。这些子着色器互不相干且**只有一个会在最终的平台运行**。编写多个的目的是解决兼容性问题。*Unity* 会自己选择兼

容终端平台的 *Shader* 运行。

- **回滚（Fallback）**：如果子着色器在终端平台上都无法运行，那么使用 Fallback 指定的备用 Shader，**俗称备胎**。
- **Pass**：一个 *Pass* 就是一次绘制。**对于表面着色器，只能有一个 Pass**，所以不存在 Pass 节。**顶点片段着色器可以有多个 Pass**。多次 Pass 可以实现很多特殊效果，例如当人物被环境遮挡时还可以看到人物轮廓就可以用多 Pass 来实现。
- **Cg 代码**:每个 Pass 中都可以包含自定义的 Cg 代码,**从 CGPROGRAM 开始到 ENDCG 结束**。

14.3 Shader2.0

可以实现编程。

https://www.cnblogs.com/lixiang-share/p/5025662.html

大概流程

- MeshRender 将 CPU 数据传入至 GPU
  - 然后传递给顶点着色器
    - 顶点着色器处理完成后，将自己的输出数据传递给片段着色器处理

定义顶点着色器入口函数 `#pragma vertex _vertexFunctionName`

定义片段着色器入口函数 `#pragma fragment _fragmentFunctionName`

定义平面着色器入口函数 `#pragma surface _surfaceFunctionName`

常用语义 POSITION **语义**等相当于告诉渲染引擎，这个变量是代表什么含义。

- `POSITION` 获取模型顶点信息。
- `NORMAL` 获取法线信息。
- `TEXCOORD n` 第 (n) 张贴图的 uv 坐标
- `COLOR n` 第 n 个定点色。`float4`
- `TANGENT` 获取切线信息

- SV_POSITION 表示经过 MVP 矩阵已经转化到屏幕坐标的位置
- SV_TARGET 输出到哪个 Render Target 上

常用坐标系

- 模型坐标系：也叫物体坐标系，3D 建模的时候每个模型都是在自己的坐标系下建立的，如果一个人物模型脚底是 (0,0,0) 点的话它的身上其它点的坐标都是相对脚底这个原点的。
- 世界坐标系：我们场景是一个世界，有自己的原点，模型放置到场景中后模型上的每个顶点就有了一个新的世界坐标。这个坐标可以通过模型矩阵 × 模型上顶点的模型坐标得到。
- 视图坐标系：又叫观察坐标系，是以观察者（相机）为原点的坐标系。场景中的物体只有被相机观察到才会绘制到屏幕上，相机可以设置视口大小和裁剪平面来控制可视范围，这些都是相对相机来说的，所以需要把世界坐标转换到视图坐标系来方便处理。
- 投影坐标系：场景是 3D 的，但是最终绘制到屏幕上是 2D，投影坐标系完成这个降维的工作，投影变换后 3D 的坐标就变成 2D 的坐标了。投影有平行投影和透视投影两种，可以在 Unity 的相机上设置。
- 屏幕坐标系: 最终绘制到屏幕上的坐标。屏幕的左下角为原点。

常用矩阵表示

- UNITY_MATRIX_MVP：当前模型 -> 视图 -> 投影矩阵。（注：模型矩阵为本地-> 世界）
- UNITY_MATRIX_MV：当前模型 -> 视图矩阵
- UNITY_MATRIX_V：当前视图矩阵
- UNITY_MATRIX_P：当前投影矩阵
- UNITY_MATRIX_VP：当前视图 -> 投影矩阵
- UNITY_MATRIX_T_MV：转置模型 -> 视图矩阵
- UNITY_MATRIX_IT_MV:逆转置模型 -> 视矩阵, 用于将法线从ObjectSpace 旋转到WorldSpace。为什么法线变化不能和位置变换一样用UNITY_MATRIX_MV 呢？一是因为法线是 3 维的向量而- UNITY_MATRIX_MV 是一个 4x4 矩阵，二是因为法线是向量，我们只希望对它旋转，但是在进行空间变换的时候，如果发生非等比缩放，方向会发生偏移。
- UNITY_MATRIX_TEXTURE0 to UNITY_MATRIX_TEXTURE3：纹理变换矩阵

14.4 材质、贴图、纹理

材质 Material 包含贴图 Map，贴图包含纹理 Texture。

纹理 ->

是最基本的数据输入单位，游戏领域基本上都用的是位图。常见格式有 PNG，TGA，BMP，TIFF 此外还有程序化生成的纹理 Procedural Texture。在内存中通常表示为二维像素数组。

贴图 ->

英语 Map 其实包含了另一层含义就是“映射”。其功能就是把 **纹理**通过 UV 坐标 映射 到 *3D 物体表面*。

贴图包含了**除了纹理以外其他很多信息**，比方说 **UV 坐标、贴图输入输出控制**等等。一张图便能说明其之间的关系

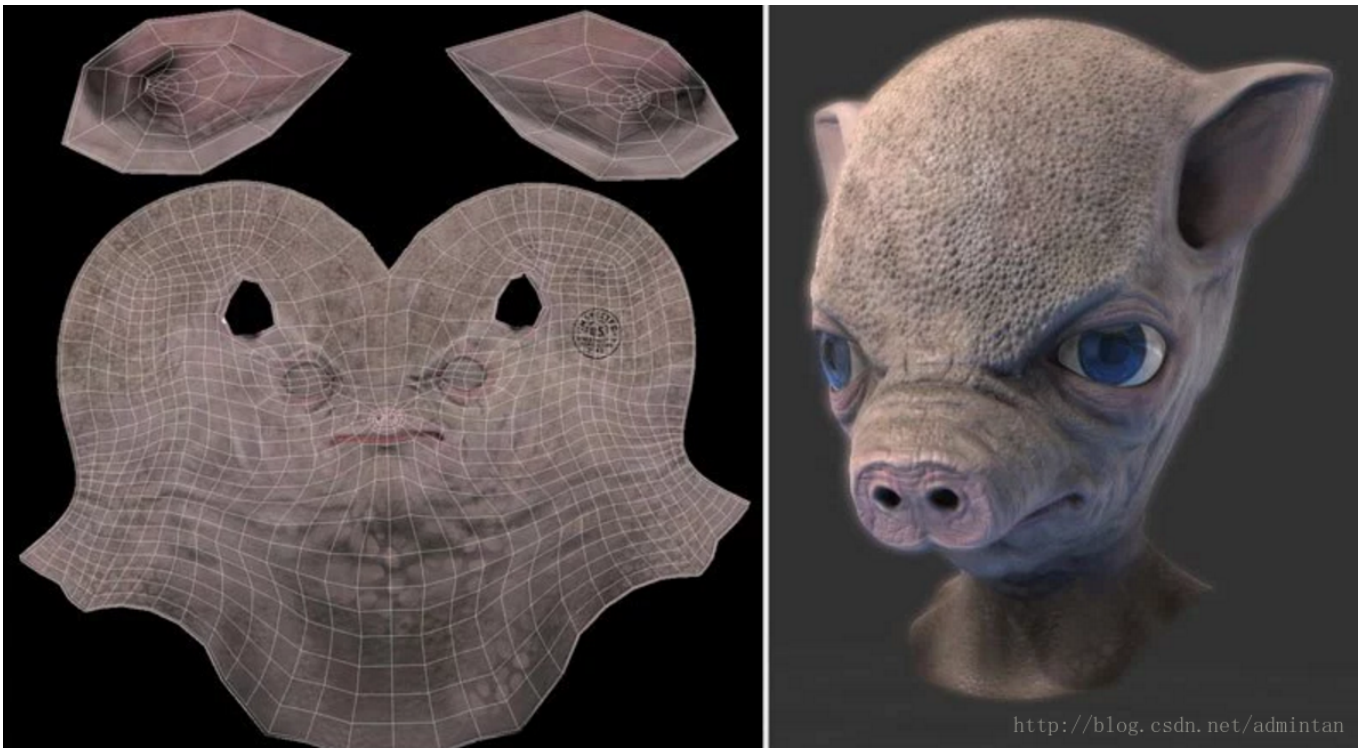


图 14.2: 纹理与贴图

材质 ->

本质就是一个**数据集**，主要功能就是**给渲染器***提供数据和光照算法*。

贴图就是其中**数据的一部分**,根据用途不同,贴图也会被分成不同的类型,比方说 Diffuse Map，

`Specular Map`，`Normal Map` 和 `Gloss Map` 等等。

另外一个重要部分就是**光照模型 Shader**，用以实现不同的渲染效果。贴图种类繁多：我做个不完全总结

Diffuse Map 漫反射贴图/也被称作反照率贴图 albedo map，存储了物体相应部分漫反射颜色

Normal Map 法线贴图本质上存储的是被 RGB 值编码的法向量表现凹凸，**比如一些凹凸不平的表面，光影在表面产生实时变化，常用来低精度多边形表现高精度多边形细节，比如在高多边形下生成 normal map 在匹配给低多边形模型是一种常见的降低性能要求的做法**

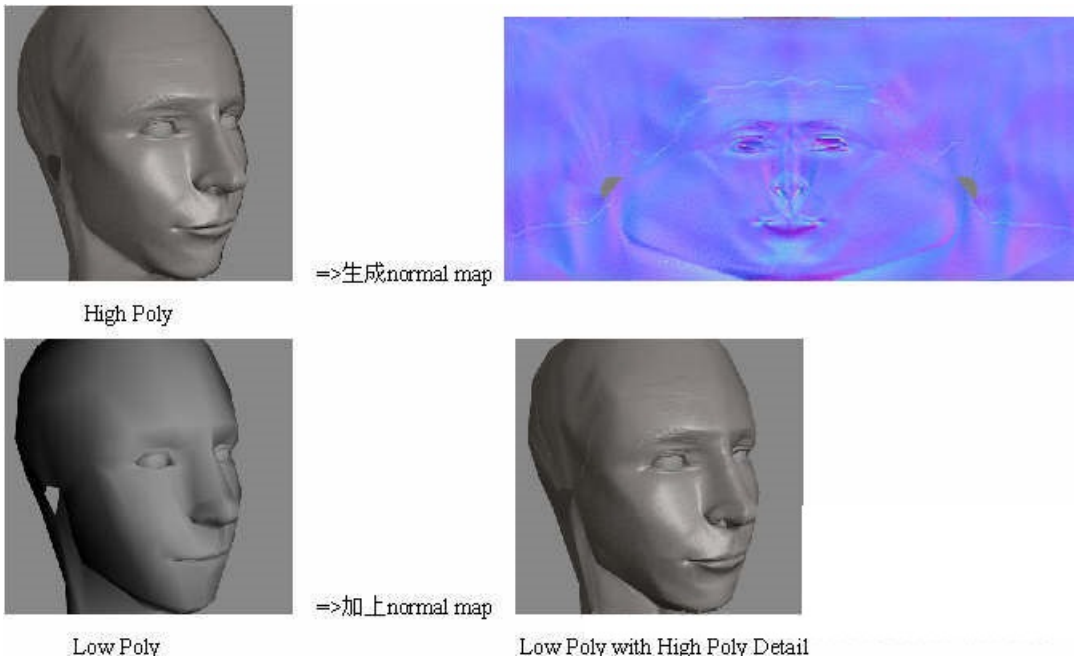


图 14.3: 法线贴图演示

建议参考：https://blog.csdn.net/cywater2000/article/details/749341#comments

Specular Map 高光贴图表现质感高光区域大小可真实反映材质区别

Gloss Map 光泽贴图，每个纹理元素上描述光泽程度

总体上说他们都是要被 shader 加工的原材料.

14.5 光照模型

参考：https://blog.csdn.net/admintan/article/details/53913624

14.5.1 Phone 光照模型

真实世界中的光照效果抽象为三种独立的光照效果的叠加。

$$Color = Ambient + Diffuse + Specular$$

环境光-Ambient 此为模拟环境中的整体光照水平，是间接反射光的粗略估计，间接反射的光使阴影部分不会变成全黑关于环境光还有个事实，1 某个可以独立分析的局部场合的环境光强和能够进入这个地方的光的强度有关。

$$Ambient = K_a * GlobalAmbient$$

漫反射光-Diffuse 模拟直接光源在表面均匀的向各个方向反射，能够逼近真实光源照射到哑光表面的反射。比如在阳光下，由于路面粗糙的性质，我们发现从任意一个角度观察路面，亮度都是差不多。

$$Diffuse = K_d * lightColor * dot(L, N)$$

镜面反射光-Specular 模拟在光滑表面会看到的光亮高光。会出现在光源的直接反射方向。镜子、金属等表面光亮的物体会有镜面反射光。镜面反射光同时与物体表面朝向、光线方向、视点位置有关。如图：

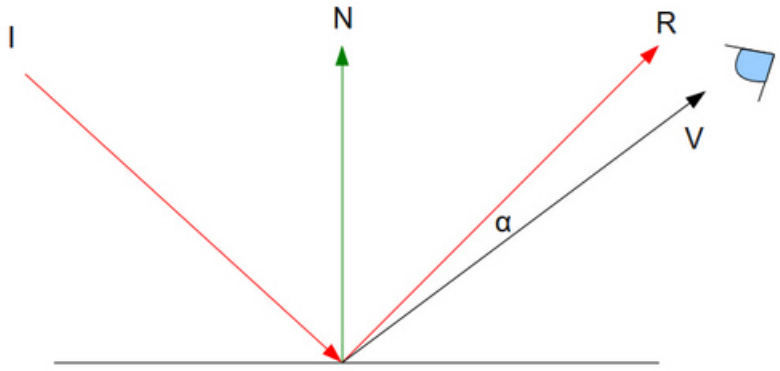


图 14.4: 高光演示

I 是入射光，N 是表面法线，R 是反射光线，V 是从物体上的目标观察点指向视点的向量，a 是 V 和 R 的夹角。

我们可以判断出一个规律，夹角 a 越小，即视线与反射方向的偏离越小，则目标点的光强越大, 计算公式为：

$$Specular = K_s * lightColor * (dot(V, R))^{shininess}$$

- Ks 为物体对于反射光线的衰减系数
- Shininess 为高光指数, 高光指数反映了物体表面的光泽程度。
  - Shininess 越大，反射光越集中，当偏离反射方向时，光线衰减的越厉害，只有当视线方向与反射光线方向非常接近时才能看到镜面反射的高光现象，此时，镜面反射光将会在反射方向附近形成亮且小的光斑；
  - Shininess 越小，表示物体越粗糙，反射光分散，观察到的光斑区域小，强度弱。

总结 至于更多的光照模型结合了更多的物理光学等信息 在模拟单种材质例如塑料 合金石膏陶瓷等等上要好于 Phone 式模型的效果但是基本上属于 Phone 式模型的扩充

第十五章　资源管理- AssetBundle

15.1　工作流程

AssetBundle 的使用流程

- 1- 创建 AssetBundle
  - 2- 上传到 Server
    - 3- 游戏运行时根据需要下载 AssetBundle 文件
      - 4- 解析加载 Assets
        - 5- 使用完后释放内存等资源

15.2　创建

通过编译管线 BuildPipeline 来创建 AssetBundle 文件，总共有三种方法, 具体如下所示。

15.2.1　BuildAssetBundle

该 API 将编辑器中的任意类型的 Assets 打包成一个 AssetBundle，适用于**对单个大规模场景的细分**。

- 名称：**BuildPipeline.BuildAssetBundle**
- 参数-1：**mainAsset** : Object
- 参数-2：**assets** : Object[]
- 参数-3：**pathName** : string
- 参数-4：**options** : BuildAssetBundleOptions
- 参数-5：**targetPlatform** : BuildTarget = BuildTarget.WebPlayer
- 返回值：**bool**

BuildAssetBundleOptions

CompleteAssets

CollectDependencies

DisableWriteTypeTree

DeterministicAssetBundle

UncompressedAssetBundle

AssetBundle 之间的依赖 如果游戏中的某个资源被多个资源引用（例如游戏中的 Material），单独创建 AssetBundle 会使多个 AssetBundle 都包含被引用的资源（这里跟 flash 编译选项中的链接选项有些像），从而导致资源变大，这里可以通过指定 AssetBundle 之间的依赖关系来减少最终 AssetBundle 文件的大小（把 AssetBundle 解耦）。

具体方法是在创建 AssetBundle 之前调用 BuildPipeline.PushAssetDependencies 和 BuildPipeline.PopAssetDependencies 来创建 AssetBundle 之间的依赖关系，它的用法就是一个栈，后压入栈中的元素依赖栈内的元素。

15.2.2 BuildStreamedSceneAssetBundle

该 API 将一个或多个场景中的资源及其所有依赖以流加载的方式打包成 AssetBundle，一般适用于**多单个或多个场景进行集中打包**

- 名称：**BuildPipeline.BuildStreamedSceneAssetBundle**
- 参数-1：**level** : string[]
- 参数-2：**locationPath** : string
- 参数-3：**target** : BuildTarget
- 返回值：**String**

15.2.3 BuildAssetBundleExplicitAssetNames

该 API 功能与 a 相同，但创建的时候可以为每个 Object 指定一个自定义的名字。（一般不太常用）

- 名称：**BuildPipeline.BuildAssetBundleExplicitAssetNames**
- 参数-1：**assets**：Object[]
- 参数-2：**assetNames**：string[]
- 参数-3：**pathName**：string
- 参数-4：**options**：BuildAssetBundleOptions
- 参数-5：**targetPlatform**：BuildTarget
- 返回值：**bool**

15.3 使用

15.4 卸载

15.5 内存模型

15.6 其他

第十六章　Editor 扩展

Unity 编辑器扩展是扩展 Unity 菜单功能，也可以说是自定义 Unity 菜单，以此来便利我们能够更快捷地开发游戏。

16.1　流程

1. 在 Asset 文件夹下**创建一个文件夹**`Editor`，如果已经存在则忽略此步
2. 在该`Editor` 文件夹下，**创建一个**`C# 脚本`，无需继承于任何类
3. **引用**`UnityEditor` **命名空间**
4. **写静态方法**既菜单的功能,类似于`[SerilizedFiled]` 下的东西, 与`[Menuitem("Netease/xx")]` 一一对应。
5. **给该方法上添加**`[Menuitem("")]` 特性

16.2　在编辑器上增加一个 MenuItem

16.3　创建一个对话框

16.4　扩展 Inspector 面板

16.5　编辑器插件常用函数

16.5.1　资源导入回调函数

当导入资源到 Unity 项目中的某个资源文件夹下时，当 Unity Editor 获得焦点后，会在加载完资源后，先为其创建.meta 文件，然后再触发该回调函数。

```
public class TestBundleNameAndTexture : UnityEditor.AssetPostprocessor
{
    static void OnPostprocessAllAssets( // 这个函数必须为静态的，其他可以不是!
        string[] importedAssets,
        string[] deletedAssets,
        string[] movedAssets,
        string[] movedFromAssetPaths)
    {
        foreach (var path in importedAssets)
        {
            DirectoryInfo dir = new DirectoryInfo(path);
            Debug.Log(dir.Parent.FullName);
            BundelNameCreater.Proc(dir.Parent.FullName);
        }
    }

}
```

16.6 一些常用的 Inspector 属性设置

16.7 参考

https://blog.csdn.net/puppet_master/article/details/51012298

http://blog.csdn.net/asd237241291/article/details/38235091

http://blog.sina.com.cn/s/blog_471132920101n8cr.html

第十七章　跨平台发布 apk

17.1　流程

- 安装 JavaSDK、Android Studio 并在 SDK manager 里添加对应的 API 包
- 在 unity 中的edit 选项下的preferences, 并选中External Tools 选项,配置JDK 和Android SDK 安装位置。
- 在 unity 中的File -> Build Settings 中，添加需要添加的场景，并选择对应的平台（Android，IOS）等
- 在 unity 中的Build Settings 中的Player Settings 设置以下几个重要内容。

    1. Company Name
    2. Product Name
    3. Default Icon :192×192
    4. Default Orientation
    5. Other Settings -> Identification : 修改为com.netease(Or Other).TestName(Or Other)

17.2　Apk 安装常见错误

http://mumu.163.com/2017/03/30/25905_680657.html

第十八章　调试技巧

18.1　以父类为基点

在 Inspector 中查看是否存在父类脚本`[SerializedField]` 的变量，这样方便对空间进行查找，并且添加新的**控制**

第十九章　GPU 相关

大部分时间在**知其然不知其所以然**。该部分将从 *GPU* 讲到游戏引擎再到游戏逐层介绍，以打通任督 2 脉。

介绍绘画流程的 video：https://v.youku.com/v_show/id_XNjY3MTY4NjAw.html

本节参考：https://blog.csdn.net/admintan/article/details/53861781

19.1　最底层——GPU/硬件原理

19.1.1　硬件基础

ALU-Arithmetic Logic Unit 算数逻辑单元　->

整数算术运算（加、减，有时还包括乘和除，不过成本较高）、位逻辑运算（与、或、非、异或）、移位运算（将一个字向左或向右移位或浮动特定位，而无符号延伸），移位可被认为是乘以 2 或除以 2。ALU 可以说是计算机处理器的核心部件之一。

Cache-缓存 SRAM　->

SRAM 叫静态内存，“静态”指的是当我们将一笔数据写入 SRAM 后，除非重新写入新数据或关闭电源，否则写入的数据保持不变。

由于 CPU 的速度比内存和硬盘的速度要快得多，所以在存取数据时会使 CPU 等待，影响计算机的速度。SRAM 的存取速度比其它内存和硬盘都要快，所以它被用作电脑的高速缓存 (Cache)。

19.1.2　CPU with GPU 异同

参考：https://www.jianshu.com/p/fae645d70e0e

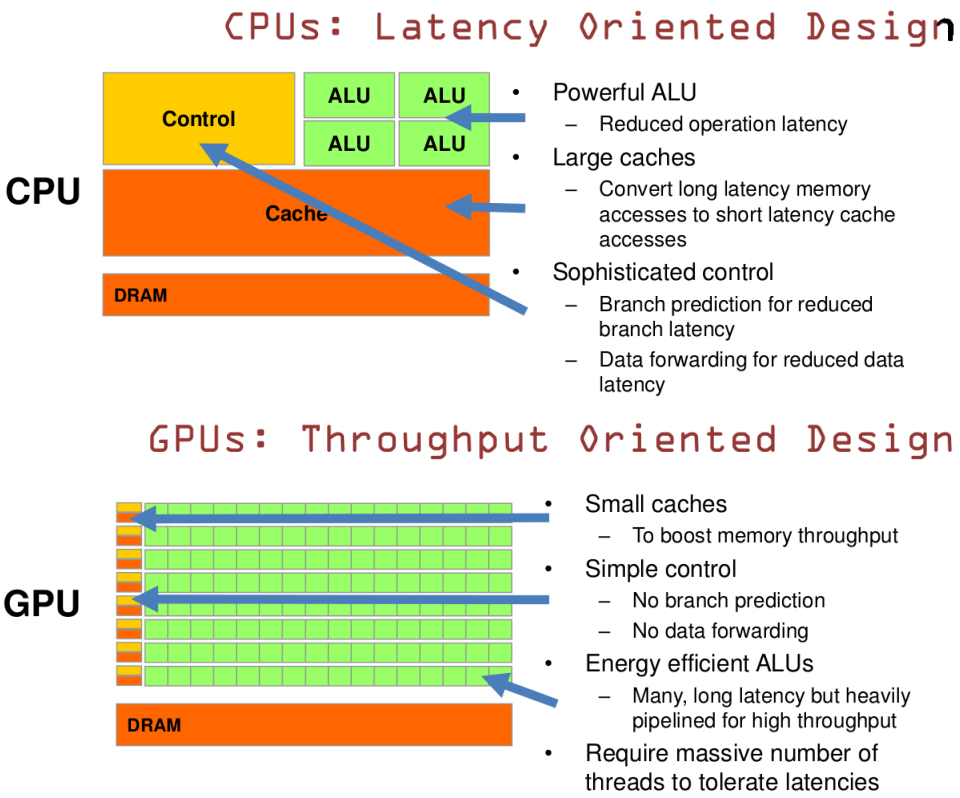


图 19.1: CPU with GPU

CPU -> 量小但能力强

- ALU **部分会很强大**，可以在很少的时钟周期内完成算数计算。对于一个 64bit 双精度的 CPU 来说浮点加法和乘法只需要 1-3 个时钟周期。
- **大的 Cache** 也将延时降低很多, 结合了现在的各种高级调节技术比如超线程、多核等技术 CPU 对于复杂逻辑的运算能力得到了极大提升。

GPU -> 量大但能力弱

- ALU 的**数量会非常大、功能会更少、能耗很低、cache 就会很小**，这样带来的好处就是针对大吞吐量的需要简单计算的数据来说，处理效率就高了非常多。

  如果有很多线程需要访问同一个相同的数据，*缓存会合并这些访问，然后再去访问 DRAM*（因为需要访问的数据保存在 DRAM 中而不是 Cache 里面），**获取数据后 Cache 会转发这个数据给对应的线程，这个时候是数据转发的角色**。但是由于需要访问 DRAM，自然会带来延时的问题。

- GPU 的控制单元（左边黄色区域块）**可以把多个的访问合并**成少的访问。
- GPU 的虽然有 DRAM 延时，**却有非常多的 ALU 和非常多线程**. 为了平衡内存延时的问

题，我们可以中充分利用多的 ALU 的特性达到一个非常大的吞吐量的效果。尽可能多的分配多的线程. 通常来看 GPU ALU 会有非常重的 pipeline 就是因为这样。

结论 ->

我们可以先得出一个简单结论:对显卡来说-**更适合做高并行，高数据密度，简单逻辑的运算**。

19.1.3 GPU 架构

4D 向量和 4+1 ->

3D 物件的成像過程中，VS（Vertex Shader，**顶点着色引擎**）PS（Pixel Shader，**像素著色引擎**或片段着色器）**最主要的作用就是运算坐标（XYZW）@（RGBA）**。

此时数据的**基本单位是 scalar（标量）**，1 个单位的变量操作，为 1D 标量简称 1D。而跟标量相對的就是 vector（向量），**向量是由多個标量构成**。*例如每個周期可执行 4 個向量平行运算，就称为 4D 向量架构*。若 GPU 指令发射口只有 1 个，卻可執行 4 個数据的平行运算，这就是 SIMD(单指令多数据流) 架构。

运算单元计算机制 ->

以 GPU 的矩阵加法为例：

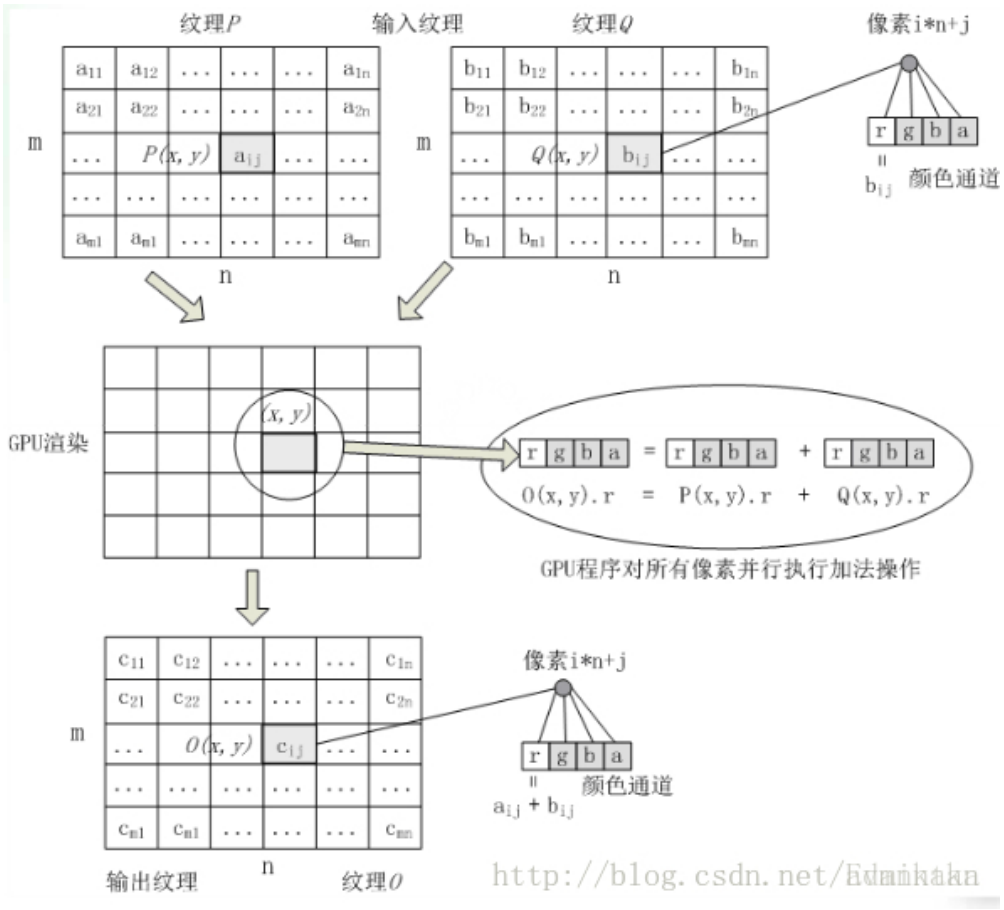


图 19.2: 运算单元计算机制

NVIDIA ->

NVIDIA 架构如下

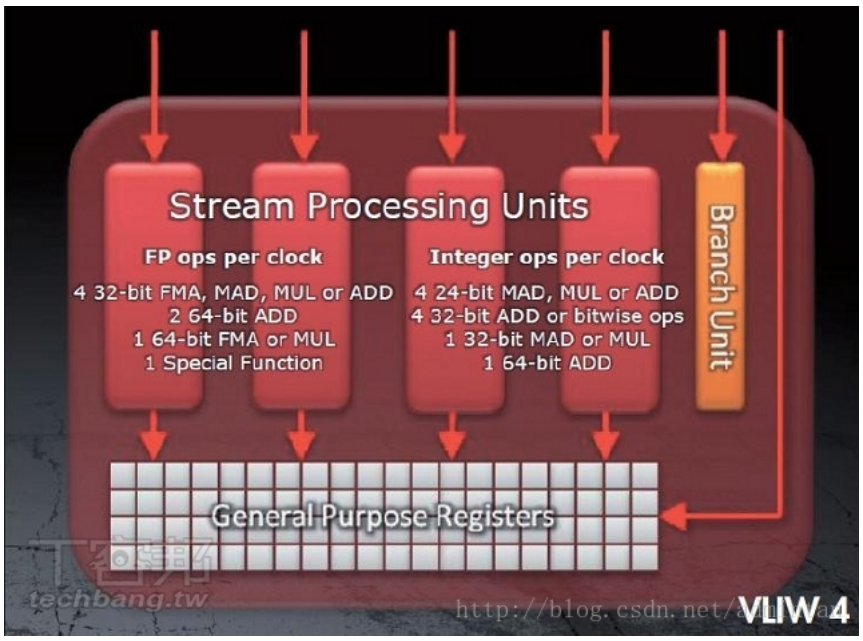


图 19.3: NVIDIA 架构演示

标注的是Stream Processing(流处理器) 数量,NVIDIA 的流处理器每个都具有完整的 ALU(可以理解为数学、逻辑等运算)。NVIDIA 从 G80 以后采用全标量设计，**所有运算全都转为标量计算。但是这么做一旦遇到 4D 矢量运算时，就需要 4 次运算才能完成**，所以 NVIDIA 显卡的 Shader 频率几乎比核心频率高一倍，就是为了弥补这个缺点。NV 的流处理器都具有完整的 ALU 功能，所以每个流处理器消耗的晶体管数量较多，成本较高。在加上现在的 CUDA 功能所以晶体管数量大幅多于 AMD-ATI。

AMD/ATI ->

AMD 架构如下

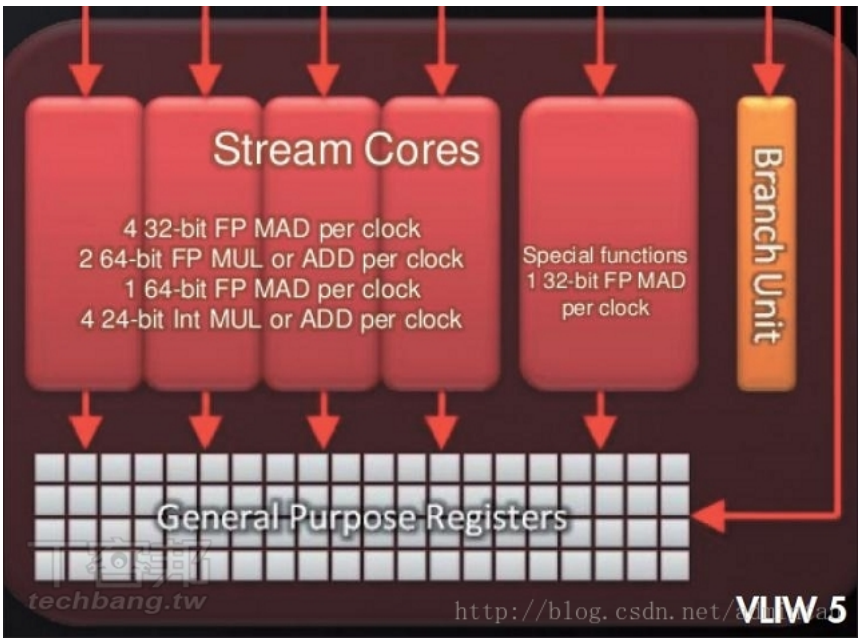


图 19.4: AMD 架构演示

标注的是 Stream Processing Units(统一渲染单元) 数量，也可以叫流处理器单元。AMD-ATI 从 RV670 以后，流处理器是 **5 个固定的统一渲染单元为一组**，4D 矢量 +1D 标量组合。**其中 4 个只能进行 MADD(乘加) 运算，1 个可以进行超运算 (函数等运算)**。*因为是 5 个固定为一组，不能拆分，所以遇到纯标量运算时就会有 4 个 SPU 处于闲置状态而无法加入其它 SP 组合协助运算*。但换句话说如果分配得当让每个 SPU 都充分工作，那么 AMD 显卡的效率可是非常高的。**这也是玩家公认 A 卡驱动提升性能比 N 卡要高**，但也就是这个原因导致 A 卡驱动设计难度非常高，游戏想要为 A 卡优化的难度也一样很高。

19.2 更高层-硬件流程

19.2.1 数据存储转换

资源信息和指令信息由**硬盘**经过 CPU 调度传输到**内存**中转，再传输进**显存**中备用.

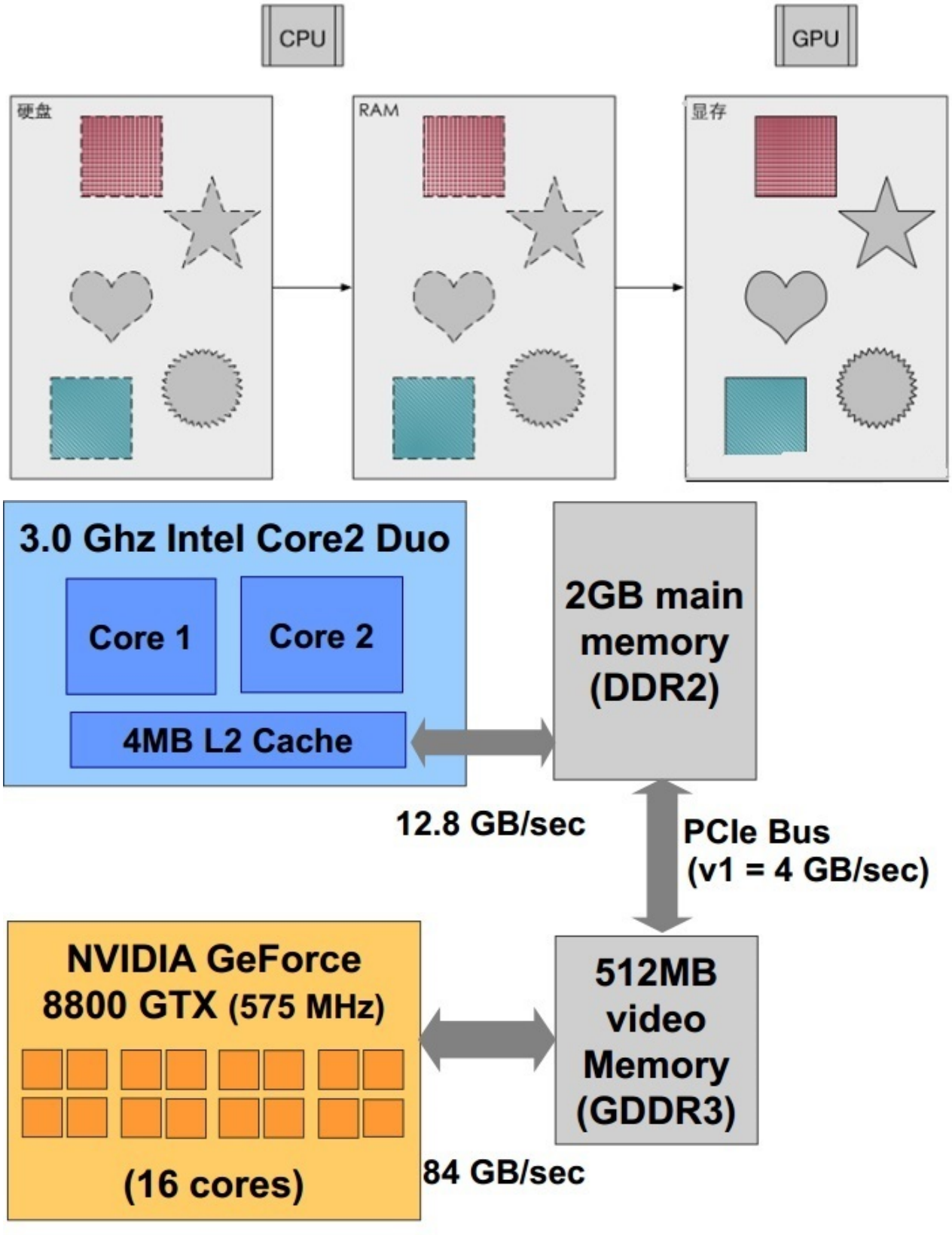


图 19.5: 资源转移

由图中的典型的带宽可见 **GPU 和显存之间的内部带宽**要比单纯的**系统总线带宽**要高很多。所以典型情况下**渲染资源和渲染指令都被加载进显存**, 所以 GPU 在渲染过程中 **只需向显存调度**

渲染指令,避免了和系统总线频繁 IO。

19.2.2 进入渲染预备状态

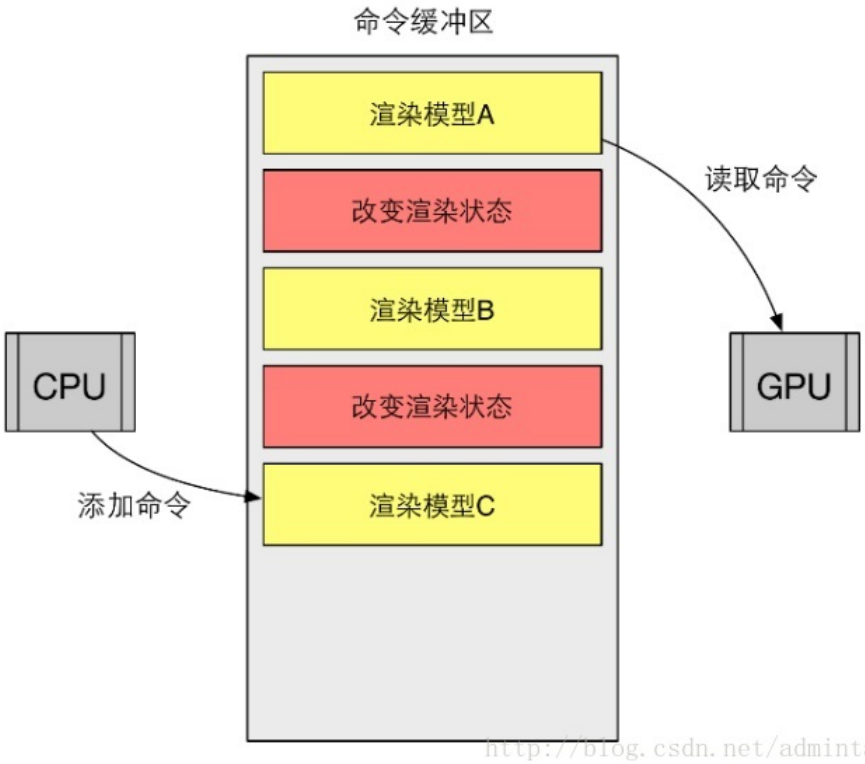


图 19.6: 渲染预备状态

此时 GPU 的**显存越大则依靠 CPU 的加载额外渲染指令的需要就越少**，显存中的信息一般包括：显存和内存一样用于存储 GPU 处理过后的数据，在显存中有几种不同的储存区域，用于储存不同阶段需要的数据。

1. **顶点缓冲区**：用于储存从内存中传递过来的顶点数据。
2. **索引缓冲区**：用于储存每个顶点的索引值，我们可以根据索引来使用相应的顶点
3. **纹理缓冲区**：用于储存从内存中传递过来的纹理数据
4. **深度缓冲区**：用于存储每个像素的深度信息
5. **模板缓冲区**：用于存储像素的模板值，且模板缓冲区域深度缓冲区公用一片内存。
6. **颜色缓冲区**：用于储存像素的颜色数据

19.3 软件

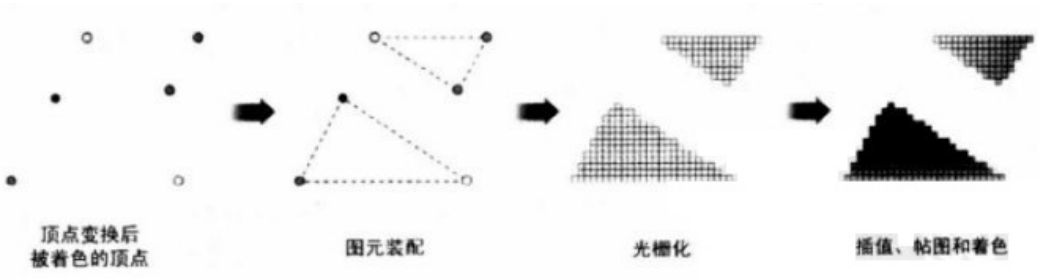
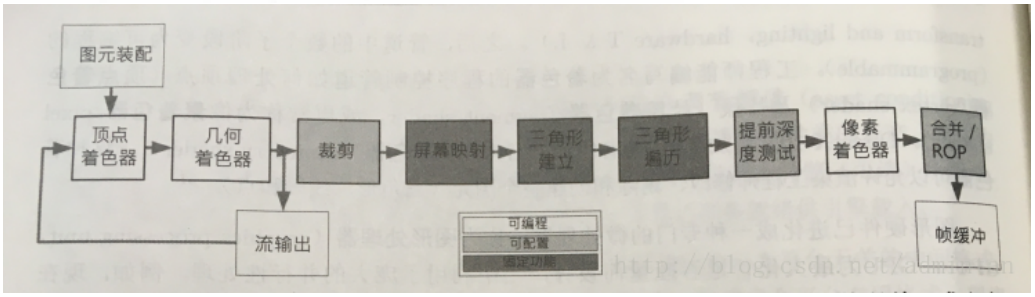


图 1-6 形象化图形流水线



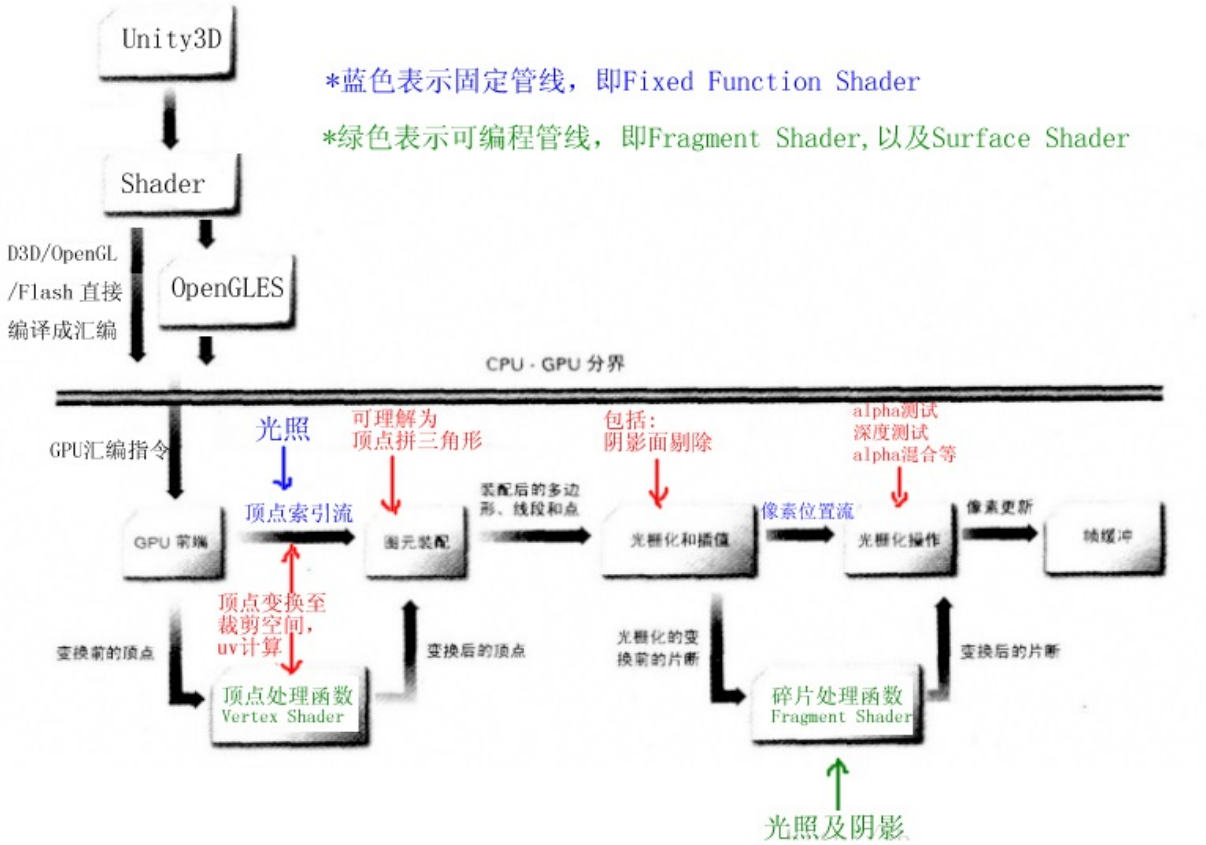


图 19.7: Shader 流程 (1,4)