Unity3D note

ianaesthetic

October 12, 2017

1 开始： Roll a Ball

1.1 创建GameObject

在Unity3D中，首先是对于 scene 的创建，并将其保存在asset中进行管理。所有的 object 都将在已存在的 scene 中进行创建。每一个 object 都具有 attribute， 可以用 inspector 查看。引擎中提供 primitives 以供使用。 最简单对于 object 的着色方法为使用material 作为 attribute，然后调节 material 中的颜色实现。 material 所创建的数据将需要保存在 asset 中。

1.2 player 的移动

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

public class Player_Controller : MonoBehaviour {
    public float speed;

    private Rigidbody rb;

    void start() {
        rb = GetComponent<Rigidbody>();
    }

    void FixedUpdate() {
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        Vector3 movement = (moveHorizontal, 0.0f, moveVertical);
        rb.AddForce(movement * speed);
    }
}
```

speed 作为一个 public 元素可以直接在引擎界面中进行调节，可以更好的控制速度。 FixedUpdate 表示其更新不受frame控制而是间隔固定的一段时间更新, 适用于包含物理状态的改动比如在RigidBody上的改动。

1.3 camera 设置

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

public class Camera_Controller : MonoBehaviour {
    private float offset;

    void start() {
        offset = transform.position — player.transform.position;
    }

    void LateUpdate() {
        transform.position = player.transform.position + offset;
```

```
    }
}
```

为了保证 camera 位置的正确性，要使用 LateUpdate 保证更新是在 player 的位置绝对确定后进行的。

1.4 设置游戏场景边界

在本例中，直接简单的设计4堵墙即可。

1.5 创建可拾取物品

引入 prefab 的概念，相当于 object 的 template。 在创建完后保存在 asset 中可以再提取创建可以保证所有的 attribute 修改可以直接在 prefab 中实现。关于 rotation script (仅针对欧拉角）：

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

public class Camera_Controller : MonoBehaviour {
    public Vector3 rotation;

    void update() {
        Transform.Rotate(rotation * Time.deltatime);
    }
}
```

使用一个大的 Pick Ups 来代表所有可拾取物的集合，在对于物件进行 duplicate 时， 需要俯视视角并从 local 调整为 global。

1.6 碰撞机制

在unity的碰撞机制中，有 dynamic object 和 static object 之分。在碰撞的过程中，只有 dynamic object 会计算碰撞和反弹，而 dynamic object 需要同时具备 collider 和 rigid body 这两种性质。collider 可以非常方便的用来作为触发条件，当被设定为 trigger 时，object 将只会计算是否碰撞而不会计算物理效果。在本例中， 我们需要设定为「当 player 碰到了其他的 collider 时， 当这个 collider 所属的 object 被标上了 Pick Up 的 tag时，不显示这个 object」。tag 需要自行创建，并且和script中使用的完全一致。 script中新增函数如下：

```
void OnTriggerEnter(collider other) {
    if(other.gameObject.CompareTag("Pick Up")) {
        other.gameObject.SetActivate(false);
    }
}
```

注意， OnTriggerEnter() 是一个可供编辑的借口，表示「探测到别的collider时」的函数，函数名字不能变化。

1.7 简单的UI设计

设计计数和显示游戏完成。在创立 Text 时，一定会出现 canvas 和 Eventsystem 两个新的 object ， 同时 Text 一定是 canvas 的子 object。 在将其移动时，注意 shift 和 alt 的可选功能。

在 Player_Controller 中插入 script 时， 注意要使用「using UnityEngine.UI」。增加的代码如下：

```
public Text countText;
public Text winText;

void SetContext{
    countText.text = "Count: " + count.ToString();
    if(count > number_of_item) {
        winText = "You Win!";
    }

    void Start() {
        ...
        count = 0;
        winText.text = "";
    }

    void FixedUpdate() {
        ...
        ++ count;
        SetContext();
    }
}
```

1.8 建立游戏程序

2 继续教程：Space shooter

2.1 基础设置

游戏的分辨率调整位于在scene一栏的下面。 可以固定游戏的分辨率。

2.1.1 玩家对象

Texture 作为一个 attribute 时会自动添加一种 material ， 也可以手动添加 material 并手动材质。

2.1.2 照相机和光照

光照在此游戏中有三种， Main Light 模拟最主要的光照来源， Fill Light 模拟环境光， Rim Light 用于突出背光面边界。

2.1.3 背景设置

没啥好说的。

2.1.4 玩家移动

在非完全模拟现实的情况下，可以直接通过设定速率。在设定玩家可以涉及的范围时，可以按照面向对象的原则单独设计一个新的类型「Boundary」并在告知系统，方便调整。 「Mathf」用于调用数学函数。使用 rb 而非 transformation 来作为计算结果是因为 transform 的所有的结果是在所有的调整结束后进行调整的， 而 rb 中的相当于就是在此次过程中的直接更新。

```
[System.Serializable]
public class Boundary {
    public float xMin, xMax, zMin, zMax;
}

void FixedUpdate() {
    ...
    rb.velocity = movement * speed;
    rb.position = new Vector3(
        Mathf.Clamp(rb.position.x, xMin, xMax),
        0.0f,
        Mathf.Clamp(rb.position.z, zMin, zMax)
    );
    rb.rotation = Quaternion.Euler(0.0f, 0.0f, rb.velocity.x * tumble)
}
```

2.1.5 创建子弹

创建子弹一类的可细分类可以先创建出一个逻辑类再将贴图等具体区分信息加上去。shader 中的 additive 的效果为 黑色变为透明然后增强其他颜色。创建一个 mover 脚本以在他出现时就自动移动。

```
void Start() {
    ...
    rb.velocity = transform.forward * speed;
}
```

2.1.6 发射子弹

子弹发射是被设定为：每当玩家按下于设定的按键并且按键频率小于开火频率时会复制一个子弹从设定好的位置（位置有一个逻辑 gameObject）的 perfab 。 作为玩家对象脚本的一部分。

```
public gameObject shoot;
public Transform shootSpawn;
public float fireRate;

private float nextTime;

void Start() {
    ...
    nextTime = 0;
}

void Update() {
    if(Input.GetButton("Fire1") && nextTime < Time.time) {
        nextTime = Time.time + fireRate;
        Instantiate(shoot, shootSpawn.position, shootSpawn.rotation);
    }
}
```

2.1.7 设置边界

本游戏采用的思路是设置一个 BoundaryBox，所有离开此 BoundaryBox collider 的对象全部被删除。

```
void OnTriggerExit(Collider  other) {
    destroy(other.gameObject);
}
```

2.1.8 陨石的设置

自转除了上一个游戏的设置方法外，还可以直接使用 rb.angularVelocity。同时也要设定碰撞检测于相应的动画（作为新的对象进行创建）。注意，动画所建立的游戏对象需要按照其使用时间设置Destroy函数, 碰撞在删除对象时不用在意顺序，所有的删除都是在所有操作进行玩之后按照 marker 进行的。

```
\\Astroid Collider

public gameObject explosionPlayer, explosionAstroid;

void Start() {
    ...
    rb.angularVelocity = Random.InsideUnitSphere * tumble;
```

```
}

void OnTriggerEnter(Collider other) {
    if(other.gameObject.tag != "BoundaryBox") {
        Destroy(other.gameObject);
        Destroy(gameObject);
        Instantiate(explosionAstroid, transform.position, Quaternion.identity);
        if(other.tag == "Player") {
            Instantiate(explosionPlayer, other.transform.position. Quaternion.identity);
        }
    }
}

\\Delete

public float lifetime;

void Start() {
    Destroy(gameObject, lifetime);
}
```

2.1.9 释放陨石

Coroutine的引入：在一个Update()中， 如果有一个循环的操作需要跨越多帧，就需要使用Coroutine函数。函数的返回类型为IEnumerator, 如果每次循环之间的操作没有帧数或者时间的间隔，那么就返回「null」， 否则返回「WaitForSeconds」,「WaitForFraps」等。 陨石的释放需要一个单独的逻辑游戏控制对象，并再次之上附上脚本。

```
    public gameObject hazard;
    public float waitPlayer, waitHazard, waitWave;
    public int hazardCount;
    void start() {
        StartCoroutine(SpawnAstroid());
    }

    IEnumerator SpawnAstroid() {
        yield return WaitForSeconds(waitPlayer);
        while(true) {
            for(int i = 0; i < (int)Random.Range(0, (float)hazardCount); ++ i) {
                Instantiate(hazard, RANDOM_POSITION, Quaternion.identity);
                yield return WaitForSeconds(waitHazard);
            }
            yield return WaitForSeconds(waitWave);
        }
    }
```

2.1.10 ui

GUIText 也可以用来显示UI，而含有这个component的object的transform的xy范围限制在[0,1]之间。可以用pixel来做细致的调整。在本例中，

GameController中的加分，胜利的判断函数需要在Astroid脚本中得到引用。对于每一个prefab，要找到public的函数要先找到需要引用的函数所属脚本的所属对象，而找到这个对象的方法是通过tag实现的。再次注意Start要大写第一个字母。

```
void Start() {
    GameObject gameObjectController = GameObject.FindWithTag("GameController");
    if(gameObjectController != null) {
        gameController = gameObjectController.GetComponent<GameController>();
    }
    else {
        Debug.Log("No GameController!");
    }
}
```

在函数体的设计中，注意由于分数是astroid的固有属性，所以在设计AddScore函数时是带有参数的。对于按键触发的行为，不同于GetButton而是GetKey-Down.R。

2.1.11 创建多种astroid

由于逻辑结构的设定，我们只需要把美术部分更换然后调整collider就可以了。在设计的时候非常重要的一点：同种类的通用逻辑可以用一个父对象来承载，而具体的美术细节挂在下面方便调换。 注意脚本中数组的使用方式。

```
    public GameObject[] hazards;
    ...
    hazard = hazards[Random.Range(0, c)];
```

2.1.12 背景滚动

原理是复制一个新的背景图案然后不断重复。强调transform.forward和Vetor3.forward的区别。简单来说transform.forward表示的是沿着transform之后的坐标轴的z轴前进，而Vector3.forward表示沿着世界坐标系的z轴前进。然后在这个背景上有粒子效果，可以单独调整粒子的施放位置，单个例子的速度，旋转，大小变化。由于是预设值所以不做过多涉及。使用Mathf.Repeat的时候，注意背景图像的大小调整以保证又一边的长度和镜头一致，并保证图像拼接紧密。复制后的相对位置是以原长方形的长和宽作为1的基准。

```
void Update() {
    float newPosition = Mathf.Repeat(Time.time * speed, length);
    transform.position = startPosition + Vector3.forward * newPosition;
}
```