유니티3D 스크립팅 개관(Unity3D Scripting Overview)

http://unity3d.com/support/documentation/ScriptReference/index.html
한국어: 황규석(2010.02.07)

이것은 유니티 안에서 스크립트가 어떻게 작동하는가에 대한 짧은 개관이다.

유니티 내부의 스크립팅은 동작들(behaviours)이라 불리는 스크립트 오브젝트들을 게임 오브젝트들에 attach 시킴으로써 이루어진다(attaching custom script objects). 스크립트 오브젝트들 내부의 서로 다른 함수들은 어떤 이벤트들 상에서 호출된다. 가장 많이 사용되는 것들은 다음과 같다:

Update함수:
이 함수는 하나의 프레임이 렌더링되기 전에 호출된다. 여기는 물리 코드를 제외한 대부분의 게임 동작 코드가 실행되는 곳이다.

FixedUpdate함수:
모든 물리 타임 스텝마다 한번씩 호출된다. 이곳은 물리 기반의 게임 동작들이 실행되는 곳이다.

함수 외부 코드(Code outside any function)
오브젝트들이 로드될 때 함수 외부의 코드들이 실행된다. 이것은 스크립트의 상태를 초기화하는데 사용될 수 있다.

주의: 이 문서의 섹션들은 당신이 Javascript를 사용한다고 가정한다. C# 또는 Boo scripts 사용에 관한 것은 Writing scripts in C#을 참조하기 바란다.

당신은 또한 이벤트 핸들러들을 정의할 수 있다. 이 모든 것들은 On으로 시작되는 이름들을 가진다.(즉, OnCollisionEnter처럼). 미리 정의된 모든 이벤트 리스트를 보려면, MonoBehavior를 위한 문서를 참조하기 바란다.

Subsections

Common Operations
Keeping Track of Time
Accessing Other Components
Accessing Other Game Objects
Vectors
Member Variables & Global Variables
Instantiate
Coroutines & Yield
Writing Scripts in C#

The most important classes
Performance Optimization
Script compilation (Advanced)

Overview: Common Operations

대부분의 게임 오브젝트에 대한 조작은 게임오브젝트의 Transfom, 그리고/또는 Rigidbody를 통해서 이루어진다. 이것들은 behaviour script들 안에서 멤버변수 transform과 rigidbody를 통해 각각 액세스 될 수 있다. 그래서 만일 당신이 매 프레임 Y축으로 5도 만큼씩 회전시키길 원한다면 아래와 같이 코드를 작성할 수 있다:

```
function Update(){
transform.Rotate(0, 5, 0);
}
```

만일 당신이 하나의 오브젝트를 앞으로 움직이고 싶다면 다음처럼 쓸 수 있다:

```
function Update() {
transform.Translate(0, 0, 2);
}
```

Overview: Keeping Track of Time

Time 클래스는 deltaTime이라 불리는 매우 중요한 클래스 변수를 갖는다. 이 변수는 Update 또는 FixedUpdate 함수가 마지막으로 호출된 이후 경과한 시간의 양을 저장한다.(Update함수 내부인지 FixedUpdate함수 내부인지에 따라 다르다.)

그래서 앞의 예제를 frame rate에 종속되지 않고 일정한 속도로 회전하도록 만들기 위해
다음처럼 수정한다:

```
function Update() {
transform.Rotate(0, 5 * Time.deltaTime, 0);
}
```

오브젝트 이동:

```
function Update() {
transform.Translate(0, 0, 2 * Time.deltaTime);
}
```

만일 당신이 매 프레임마다 하나의 값에 더하거나 뺀다면 Time.deltaTime으로 곱해야만 한다. 당신이 Time.deltaTime을 곱할 때 당신은 "나는 매프레임 10미터를 움직이는 대신 매초 이 오브젝트를 10미터 움직이길 원하다."라고 표현하는 것이 된다. 이것은 당신의 게임이 frame rate에 독립적으로 실행될 뿐 아니라 모션을 위해 사용되는 유닛들을 이해하기 쉽게 만들기 때문에 좋다. (초당 10미터)

다른 예로, 만일 당신이 시간에 따라 빛의 범위를 증가시키고 싶다면, 다음 구문은 초당 2유닛 만큼씩 반경을 변경시킬 것이다.

```
function Update() {
light.range += 2.0 * Time.deltaTime;
}
```

forces를 통해 Rigidbody를 조작할 때는, 엔진이 이미 그것을 처리해 주므로 일반적으로 Time.deltaTime을 곱해줄 필요가 없다.

Overview: Accessing Other Componets

컴포넌트들은 게임오브젝트에 attach 된다. 하나의 Renderer 컴포넌트를 게임오브젝트에 attach 하는 것은 그것을 스크린 상에 렌더링 되도록 만드는 것이고, Camera 를 attach 하는 것은 그것을 카메라 오브젝트 안으로 변환하는 것이다. 모든 스크립트들은 컴포넌트들이므로 그것들은 게임오브젝트들에 attach 될 수 있다.

대부분의 일반 컴포넌트들은 간단한 멤버변수들로 액세스 할 수 있다:

Component	Accessible as
Transform	transform
Rigidbody	rigidbody
Renderer	renderer
Camera	camera (only on camera objects)
Light	light (only on light objects)
Animation	animation
Collider	collider
... etc.	

미리 정의된 변수들의 전체 리스트는 Component, Behaviour, 그리고 MonoBehaviour 클래스에 대한 문서들을 참조하기 바란다. 만일 게임오브젝트가 당신이 fetch 하고픈 타입의 컴포넌트를 가지고 있지 않다면, 위 변수들은 null 로 셋팅될 것이다.

게임오브젝트에 attach 된 어떤 컴포넌트나 스크립트는 GetComponent 를 통해서 access 될 수 있다.

```
transform.Translate(0, 1, 0);
// is equivalent to
GetComponent(Transform).Translate(0, 1, 0);
```

transform 과 Transform 사이의 대/소문자 차이에 주의하라. 전자는 하나의 변수이다(소문자), 후자는 하나의 클래스 또는 스크립트의 이름이다(대문자). 이 case difference 는 당신에게 클래스 & 스크립명과 변수명을 구별하게 해준다.

우리가 배운 것을 적용해 이제 우리는 GetComponent 를 사용해 동일한 게임오브젝트에 attach 된 어떤 다른 스크립트 또는 builtin 컴포넌트들을 찾을 수 있다. 다음의 예제를 작동시키려면 DoSomething 함수를 가진 OtherScript 라는 스크립트가 필요하다. OtherScript 스크립트는 게임오브젝트에 attach 되어야 한다.

```
// This finds the script called OtherScript in the same game object
// and calls DoSomething on it.
function Update ()
{
otherScript = GetComponent(OtherScript);
otherScript.DoSomething();
}
```

Overview: Accessing Other Game Objects

대부분의 게임코드는 단 하나의 오브젝트만 다루지 않는다. 유니티 스크립팅 인터페이스는 다른 게임오브젝트들과 그 내부의 컴포넌트들을 찾아서 액세스하기 위한 다양한 방법들을 제공한다. 다음의 코드에서 우리는 씬 내의 게임오브젝트들에 attach 된 OtherScript.js 라는 이름의 스크립트가 있다고 가정한다.

```
var foo = 5;
function DoSomething ( param : String){
print(param + " with foo: " + foo);
}
```

1. Through inspector assignable references.

inspector 를 통해서 어떤 타입의 오브젝트에 변수들을 assign 할 수 있다:

```
// Translate the object dragged on the target slot
var target : Transform;
function Update () {
target.Translate(0, 1, 0);
}
```

또한 다른 오브젝트에 대한 참조들(references)을 inspector 에 노출 시킬 수 있다. inspector 의 타겟 슬롯 위로 다른 스크립트를 가지고 있는 게임 오브젝트를 드래그 할 수 있다:

```
// Set foo DoSomething on the target variable assigned in the inspector.
var target :
OtherScript;
function Update () {
// Set foo variable of the target object
target.foo = 2;
// Call do something on the target
target.DoSomething("Hello");
}
```

2. Located through the object hierarchy.

오브젝트의 Transform 컴포넌트를 통해 기존 오브젝트의 child 와 parent 오브젝트를 찾을 수 있다:

```
// Find the child "Hand" of the game object
// we attached the script to
transform.Find("Hand").Translate(0, 1, 0);
```

일단 hirerarchy 에서 transform 을 찾으면, 타 스크립트들에서 get 하기 위해 GetComponent 함수를 사용할 수 있다:

```
// Find the child named "Hand".
// On the OtherScript attached to it, set foo to 2.
transform.Find("Hand").GetComponent(OtherScript).foo = 2;

// Find the child named "Hand".
// Call DoSomething on the OtherScript attached to it.
transform.Find("Hand").GetComponent(OtherScript).DoSomething("Hello");

// Find the child named "Hand".
// Then apply a force to the rigidbody attached to the hand.
transform.Find("Hand").rigidbody.AddForce(0, 10, 0);
```

모든 child 들을 loop 할 수 있다:

```
// Moves all transform children 10 units upwards!
for (var child : Transform in transform) {
child.Translate(0, 1, 0);
}
```

더 자세한 정보는 Transform 클래스에 대한 문서를 참조하기 바란다.

3. Located by name or Tag.

GameObject.FindWithTag 와 GameObject.FindGameObjectsWithTag 를 사용해서 어떤 tag 들로 게임오브젝트들을 찾을 수 있다. 이름으로 찾으려면 GameObject.Find 를 사용하기 바란다.

```
function Start ()
{
// By name
var go = GameObject.Find("SomeGuy");
go.transform.Translate(0, 1, 0);

// By tag
var player = GameObject.FindWithTag("Player");
player.transform.Translate(0, 1, 0);

}
```

게임오브젝트를 찾은 결과 상에서 어떤 스크립트나 컴포넌트를 get 하기 위해서 GetComponent 를 사용할 수 있다.

```
function Start ()
{
// By name
var go = GameObject.Find("SomeGuy");
go.GetComponent(OtherScript).DoSomething();

// By tag
var player = GameObject.FindWithTag("Player");
player.GetComponent(OtherScript).DoSomething();
}
```

메인 카메라 같은 특별한 오브젝트들은 Camera.main 을 사용하는 shorts cuts 를 가진다.

4. Passed as parameters.

어떤 이벤트 메시지들은 이벤트에 대한 자세한 정보를 담는다. 예를 들어, trigger 이벤트들은 핸들러 함수에 충돌 오브젝트의 Collider 컴포넌트를 넘겨 준다.

OnTriggerStay 함수는 collider 에 대한 참조를 제공해 준다. collider 로부터 그것에 attach 된 rigidbody 를 얻을 수 있다.

```
function OnTriggerStay( other : Collider ) {
// If the other collider also has a rigidbody
// apply a force to it!
if (other.rigidbody)
{
other.rigidbody.AddForce(0, 2, 0);
}
}
```

또는 collider 로 동일한 게임오브젝트에 attach 된 어떤 다른 컴포넌트를 get 할 수 있다.

```
function OnTriggerStay( other : Collider ) {
// If the other collider has a OtherScript attached
// call DoSomething on it.
// Most of the time colliders won't have this script attached,
// so we need to check first to avoid null reference exceptions.
if (other.GetComponent(OtherScript)) {
other.GetComponent(OtherScript).DoSomething();
}
}
```

위 샘플에서 other 변수 뒤에 접미사를 붙임으로서 충돌 오브젝트 안의 다른 컴포넌트를 액세스 할 수 있음에 주의하라.

5. All scripts of one Type

Object.FindObjectsOfType 을 사용해 한 클래스의 어떤 오브젝트나 스크립트명를 찾거나, Object.FindObjectOfType 을 사용해 한 타입의 첫번째 오브젝트를 찾기 바란다.

```
function Start ()
{
```

```
// Find the OtherScript which is attached to any game object in the scene.
var other : OtherScript = FindObjectOfType(OtherScript);
other.DoSomething();
}
```

Overview: Vectors

유니티는 모든 3D 벡터를 표현하기 위해 Vector3 클래스를 사용한다. 3D 벡터의 각 성분들은 x, y, z 멤버 변수들을 통해 액세스 될 수 있다.

```
var aPosition : Vector3;
aPosition.x = 1;
aPosition.y = 1;
aPosition.z = 1;
```

당신은 또한 모든 성분들을 한번에 초기화하기 위해 Vector3 생성자를 사용할 수 있다.

```
var aPosition = Vector3(1, 1, 1);
```

Vector3 은 또한 어떤 common 값들을 상수로 정의할 수 있다.

```
var direction = Vector3.up; // the same as Vector3(0, 1, 0);
```

벡터 상에서의 연산들은 다음 방식으로 액세스 된다.

```
someVector.Normalize();
```

또한 여러 개의 벡터들에 대한 연산들은 Vector3 클래스 함수들을 사용한다.

```
theDistance = Vector3.Distance(oneVector, otherVector);
```

(자바스크립트에게 알려 주기 위해 함수의 이름 앞에 "Vector3." 을 써주어야 하는 것에 주의하라. 이것은 모든 클래스 함수들에 적용된다.)

또한 벡터들을 다루기 위해 일반 수학 연산자들을 사용할 수 있다.

```
combined = vector1 + vector2;
```

사용 가능한 연산들과 속성들의 전체 리스트는 Vector3 클래스 문서를 참조하기 바란다.

Overview: Member Variables & Global Variables

함수 외부에서 정의된 변수는 하나의 멤버변수로 정의된다. 멤버변수들은 유니티 내의 inspector 를 통해 액세스될 수 있다. 또한 멤버 변수에 저장되는 어떤 값은 프로젝트와 함께 자동으로 저장된다.

```
var memberVariable = 0.0;
```

위의 변수는 inspector 안에서 "Member Variable"이라는 하나의 수치속성(numeric property)으로서 보여진다.

만일 변수 타입을 컴포넌트 타입(즉 Transform, Rigidbody, collider, 어떤 스크립트명 등)으로 셋팅한다면, inspector 안에서 게임오브젝트들을 값 위로 드래깅함으로써 그것들을 셋팅할 수 있다.

```
var enemy : Transform;

function Update()
{
    if ( Vector3.Distance( enemy.position, transform.position ) < 10 ) {
        print("I sense the enemy is near!");
    }
}
```

또한 private 멤버 변수들을 생성할 수 있다. Private 멤버 변수들은 스크립트 외부에서는 보이지 않아야 하는 state 를 저장하는데 유용하다. Private 멤버 변수들은 disk 에 저장되지 않으며 inspector 에서 edit 할 수 없다. 그것들은 debug 모드일 때 inspector 안에서 볼 수 있다. 이것은 실시간 updating debugger 처럼 private 변수들을 사용할 수 있게 해준다.

```
private var lastCollider : Collider;

function OnCollisionEnter( collisionInfo : Collision ) {
    lastCollider = collisionInfo.other;
}
```

전역변수

static 키워드를 사용해 전역 변수들을 생성할 수 있다.

아래 스크립트는 someGlobal 이라는 전역 변수를 생성한다.

```
// The static variable in a script named 'TheScriptName.js'
static var someGlobal = 5;

// You can access it from inside the script like normal variables:
print(someGlobal);
someGlobal = 1;
```

다른 스크립트에서 그것을 액세스하기 위해서는 스크립트명, .(dot), 전역변수명을 사용할 필요가 있다.

```
print(TheScriptName.someGlobal);
TheScriptName.someGlobal = 10;
```

Overview: Instantiate

Instantiate 은 한 오브젝트를 복제(duplicate)한다. 모든 attach 된 스크립트들과 전체 hierarchy 를 포함한다. 그것은 당신이 예상하는 그대로 reference 들을 유지한다: 복제된 hierarchy 의 외부 오브젝트들에 대한 reference 들은 untouched 상태로 남고, 내부 오브젝트들에 대한 reference 들은 복제된 오브젝트에 매핑될 것이다.

Instantiate 은 믿기지 않을 만큼 빠르고 용도도 매우 많다. 유니티를 최대로 활용하기 위해서는 필수적이다.

아래의 간단한 예제 스크립트는 collider 를 가진 rigidbody 에 attach 되었을 때 자신을 destroy 하고 대신 explosion 오브젝트를 spawn 할 것이다.

```
var explosion : Transform;

// When a collision happens destroy ourselves
// and spawn an explosion prefab instead
function OnCollisionEnter ()
```

```
{
    Destroy (gameObject);

    var theClonedExplosion : Transform;
    theClonedExplosion = Instantiate(explosion, transform.position, transform.rotation);
}
```

Instantiate 은 흔히 Prefabs 과의 결합(conjunction)에 사용된다.

Overview: Coroutines & Yield

게임코드를 작성할 때, 하나의 이벤트 시퀀스를 스크립트해 줄 필요를 자주 느끼게 된다. 그것은 다음 코드와 같을 것이다.

```
private var state = 0;

function Update()
{
    if (state == 0) {
      // do step 0
      state = 1;
      return;
    }
    if (state == 1) {
      // do step 1
      state = 2;
      return;
    }
    //
...
}
```

yield 문을 사용하는 것이 좀더 편리할 때가 종종 있다. yield 문은 return 의 특수한 한 종류이고, next time 에 호출될 때 yield 문 다음 라인부터 함수가 계속됨을 보장해 준다.

```
while(true) {
    // do step 0
    yield; // wait for one frame
    // do step 1
    yield;
    // wait for one frame
    // ...
```

```
}
```

당신은 또한 어떤 이벤트가 발생할 때까지 Update 함수의 실행을 지연시키기 위해 특별한 값들을 yield 문에 전달할 수 있다.

```
// do something
yield WaitForSeconds(5.0); // wait for 5 seconds
// do something
more...
```

당신은 coroutine 들을 stack 하고 chain 할 수 있다.

이 예제는 Do 를 실행하지만 호출한 즉시 계속해서 다음이 실행될 것이다.

```
Do ();
print ("This is printed immediately");

function Do ()
{
    print("Do now");

    yield WaitForSeconds (2);

    print("Do 2 seconds later");
}
```

이 예제는 Do 를 실행하고, 실행을 계속하기 전에 Do 가 끝날 때까지 기다릴 것이다.

```
// chain the coroutine
yield StartCoroutine("Do");
print("Also after 2 seconds");

print ("This is after the Do coroutine has finished execution");

function Do ()
{
    print("Do now");

    yield WaitForSeconds (2);

    print("Do 2 seconds later");
}
```

이벤트 핸들러들 역시 coroutine 이 될 수 있다.

Update 나 FixedUpdate 함수 안에서는 yield 를 사용할 수 없다는 것에 주의하라, 하지만 당신은 StartCoroutine 을 하나의 함수를 시작하기 위해 사용할 수 있다.

yield 사용에 대한 더 많은 정보는 YieldInstruction, WaitForSeconds, WaitForFixedUpdate, Coroutine, MonoBehaviour.StartCoroutine 을 참조하기 바란다.

Overview: Writing Scripts in C#

syntax 는 별개로 하고 C#이나 Boo 로 스크립트를 작성할 때 몇 가지 차이점이 있다.

가장 주목할 만한 것들은:

1. MonoBehaviour 로 부터의 상속

모든 동작(behaviour) 스크립트는 MonoBehaviour 로부터 상속을 받아야 한다(직접 또는 간접적으로). 이것은 자바스크립트 안에서는 자동적으로 일어나지만, C#이나 Boo 스크립트들에서는 명시적이어야 한다. 만일 당신이 유니티에서 Asset -> Create -> C Sharp/Boo Script 메뉴를 통해 스크립트를 생성한다면, 생성된 템플릿은 이미 필요한 정의를 포함하고 있을 것이다.

```
public class NewBehaviourScript : MonoBehaviour {...} // C#

class NewBehaviourScript (MonoBehaviour): ... # Boo
```

2. 초기화를 위해서는 Awake 나 Startfunction 을 사용하라.

자바스크립트에서 함수들 외부에 놓는 것들을, C# 또는 Boo 에서는 Awake 나 Start 함수 내부에 놓는다.

Awake 와 Start 의 차이점은 Awake 는 씬이 로드될 때 실행되고, Start 는 Update 나 FixedUpdate 가 처음 호출되기 직전에 호출된다. 모든 Awake 함수들은 Start 함수들이 호출되기 이전에 호출된다.

3. 클래스명은 파일명과 일치해야 한다.

자바스크립에서, 클래스명은 암묵적으로 스크립트 파일명으로 세팅된다(minus the file extension). C#과 Boo 에서는 수동으로(manually) 해야만 한다.

4. C#에서 Coroutine 들은 다른 sytax 를 갖는다.

Coroutine 은 IEnumerator 타입을 리턴해야 하고 단순히 yield ...; 가 아닌 yield return ...; 를 사용해야 한다.

```
using System.Collections;
using UnityEngine;
public class NewBehaviourScript : MonoBehaviour {
    // C# coroutine
    IEnumerator SomeCoroutine () {
        // Wait for one frame
        yield return 0;

       // Wait for two seconds
       yield return new WaitForSeconds (2);
   }
}
```

5. namespace 를 사용하지 말라.

유니티는 아직 namespace 안에 당신의 스크립트들을 놓는 것을 지원하지 않는다. 이 제한은 차후 버전에서는 제거 될 것이다.(역자: 2.6 버전에서는 가능한 듯)

6. 오직 멤버 변수들만 serialized 되고 Inspector 에 보여진다.

private 과 protected 멤버 변수들은 Expert Mode 일 때 만 보여질 것이다. Property 들은 serialize 되지 않거나, 또는 Inspector 에서 보이지 않을 것이다.

7. 생성자 사용을 피하라.

어떤 값을 생성자에서 초기화하지 말라. 대신 Awake 나 Start 를 사용하라. 유니티는 edit mode 에서 조차도 생성자를 자동으로 invoke 한다. 이것은 종종 스크립트를 컴파일한 후 즉시 발생하는데, 생성자는 스크립트의 defalut value 들을 retrieve 하기 위해서 invoked 될 필요가 있기 때문이다. 생성자는 예기치 않은 순간에 호출될 수도 있을 뿐만 아니라, prefabs 나 inactive 게임오브젝트들을 위해서 호출될 수도 있다.

예를 들어 생성자를 사용하는 singleton 패턴의 경우에 위험한 결과를 야기할 수 있고, random null reference exceptions 로 이어질 수 있다.

그러므로 만일 당신이 예를들어 singleton 패턴을 구현하고 싶다면 생성자 대신 Awake 를 사용하라. 실제로 MonoBehaviour 로부터 상속 받은 클래스의 생성자 안에 어떤 코드를 넣을 이유는 없다.

Overview: The most important classes

Global functions accessable in javascript or the base class in C#

Moving / Rotating objects around

Animation System

Rigid bodies

FPS or Third person character controller

Overview: Performance Optimization

1. **Static Typing** 을 사용하라

자바스크립트를 사용할 때 가장 중요한 최적화는 dynamic typing 대신 static typing 을 사용하는 것이다. 유니티는 당신이 어떤 작업을 해 줄 필요없이 자바스크립트 constructs 를 statically typed code 로 변환해주는 자바스크립트 type inference 라는 테크닉을 사용한다.

```
var foo = 5;
```

위의 예제에서 foo 는 자동으로 integer 값으로 추론이 된다. 그러므로 유니티는 비싼 dynamic name variable lookups 없이 많은 컴파일 타임 최적화를 적용할 수 있다. 이것이 왜 유니티의 자바스크립트가 다른 자바스크립트 구현들 보다 약 20 배 더 빠른지에 대한 이유 중 하나다.

유일한 문제는 때때로 type 추론이 모두 적용되지 않아서 유니티가 dynamic typing 으로 되돌아 가는 것이다. Dynamic typing 으로 돌아 가면 자바스크립트 코드 작성은 더 간단하다. 하지만 또한 코드가 더 느르게 실행된다.

몇가지 예제를 보자.

```
function Start ()
{
var foo = GetComponent(MyScript);
foo.DoSomething();
}
```

여기서 foo 는 동적으로 타입이 정해지므로 DoSomething 함수 호출은 필요 이상으로 오래 걸린다 – 왜냐하면 foo 의 타입은 unknown 이기 때문에 그것이 DoSomething 함수를 지원하는지를 확인해야 하고 만일 그렇다면 그 함수를 호출한다.

```
function Start ()
{
var foo : MyScript = GetComponent(MyScript);
foo.DoSomething();
}
```

여기서 우리는 foo 를 강제로 특정 타입으로 만들었다. 당신은 좀더 나은 퍼포먼스를 얻을 것이다.

2. #pragma strict 를 사용하라

이제 문제는 당신이 dynamic typing 을 사용할 때 보통 알아 차리지 못한다는 것이다. 이것을 구제해 주는 것이 #pragma strict 이다! 간단히 스크립트의 맨 위에 #pragma strict 를 추가하면 유니티는 그 스크립트에서 dynamic typing 을 disable 시키고, 당신으로 하여금 static typing 을 사용하도록 강제할 것이다. 타입을 알지 못할 때마다, 유니티는 컴파일 에러를 리포트 할 것이다. 그래서 이 경우에서는 컴파일 시 foo 에서 에러가 날 것이다.

```
#pragma strict
function Start ()
{
var foo = GetComponent(MyScript);
foo.DoSomething();
}
```

3. Cache component lookups

또 다른 최적화는 컴포넌트의 캐싱이다. 이 최적화는 약간의 코딩이 요구되고 항상 가치가 있는 것은 아니다. 하지만 만일 당신의 스크립트가 정말로 자주 사용되고 당신이 최적화의 마지막을 얻을 필요가 있다면, 이것은 매우 좋은 최적화가 될 수 있다.

GetComponent 나 액세스 변수를 통해 하나의 컴포넌트를 액세스할 때마다, 유니티는 게임오브젝트로부터 올바른 컴포넌트를 찾아야 한다. 이때 private 변수에 컴포넌트에 대한 참조를 캐싱함으로써 쉽게 저장될 수 있다.

간단히 다음을

```
function Update () {
transform.Translate(0, 0, 5);
}
```

아래처럼 바꾼다

```
private var myTransform : Transform;
function Awake () {
myTransform = transform;
}

function Update () {
myTransform.Translate(0, 0, 5);
}
```

후자의 코드는 유니티가 매프레임 게임오브젝트에서 transform 컴포넌트를 찾을 필요가 없기 때문에 상당히 더 빠르게 실행될 것이다. transform 나 다른 shorthand 속성 대신 GetComponent 를 사용하는 scripted 컴포넌트들에 모두 똑같이 적용된다.

4. Use Builtin arrays

Builtin arrays 는 빠르다, 매우 빠르다. 그러니 그것들을 사용하라.

비록 ArrayList 나 Array 클래스들은 쉽게 성분들을 추가할 수 있으므로 사용하기 더 쉽지만 그것들은 거의 동한한 속도를 가지지 않는다. Builtin Arrays 는 고정된 size 를 가지지만 대부분의 경우 사전에 maximum size 를 알고 있기에 나중에 단지 그것을 채울 수가 있다. Builtin arrays 의 가장 좋은 점은 어떤 extra type information 이나 오버헤드 없이 하나의 tightly packed buffer 에 구조체 데이터 타입들을 직접 embed 하는 것이다. 그러므로 모든 것이 메모리에서 linear 한 것처럼 chache 상에서의 iterating 이 매우 쉽다.

```
private var positions : Vector3[];
function
Awake () {
positions = new Vector3[100];
for (var i=0;i<100;i++)
positions[i] = Vector3.zero;
```

```
}
```

5. 필요없다면 함수를 호출하지 말라

모든 최적화에서 가장 간단하고 최고인 것은 덜 실행하는 것이다. 예를 들면, 멀리 떨어져 있는 적을 잠들게 하는 것은 대부분 acceptable 하다. 플레이어가 가까이 오기 전까지 아무것도 하지 않는 것이다. 이런 상황을 handling 하는 느린 방법은 아래와 같을 것이다:

```
function Update ()
{
// Early out if the player is too far away.
if (Vector3.Distance(transform.position, target.position) > 100)
return;
perform
real work work...
}
```

이것은 좋은 방법이 아니다. 왜냐하면 유니티는 매프레임 update 함수를 호출해야하기 때문이다. 좀더 나은 솔루션은 플레이어가 가까이 올 때까지 behaviour 를 disable 시키는 것이다. 이것에는 3 가지 방법이 있다:

1. OnBecameVisible 과 OnBecameInvisible 을 사용하라. 이 call back 들은 렌더링 시스템에 들어 있다. 어떤 카메라가 그 오브젝트를 볼 수 있는 순간, OnBecameVisible 은 호출되고, 어떤 카메라도 그것을 더 이상 보지 않을 때 OnBecameInvisible 이 호출 될 것이다. 이것은 몇가지 경우에 유용하다, 하지만 종종 AI 상에서는 유용하지 않다. 왜냐하면 카메라를 enemy 들로부터 돌리자 마자 disable 되기 때문이다다.

```
function OnBecameVisible () {
enabled = true;
}

function OnBecameInvisible ()
{
enabled = false;
}
```

2. triggers 를 사용하라. 하나의 간단한 sphere trigger 는 놀라운 일을 할 수 있다. 원하는 영향 범위에서 벗어 날 때 OnTriggerEnter/Exit calls 를 얻는다.

```
function OnTriggerEnter (c : Collider)
{
if (c.CompareTag("Player"))
```

```
enabled = true;
}

function OnTriggerExit (c : Collider)
{
if (c.CompareTag("Player"))
enabled = false;
}
```

3. Coroutines를 사용하라. Update함수에서의 문제는 매프레임 호출된다는 것이다. 플레이어와의 거리 체크는 5초마다 수행할 수도 있을 것이다. 이것은 처리시간을 많이 save 해준다.

Overview: Script compilation (Advanced)

유니티는 모든 스크립트들을 .NET dll 파일들로 컴파일 한다. .dll 파일들은 런타임에 JIT(Just-In-Time) compilation 될 것이다.

이것은 믿기 어려울 만큼 빠르게 스크립트를 실행할 수 있게 해준다. 그것은 전통적인 자바스크립트 보다 20 배 더 빠르고 native C++ 코드보다 약 50% 더 느리다. 당신이 스크립트를 저장할 때 유니티는 당신의 모든 스크립트를 컴파일하고 잠깐의 시간이 걸린다. 만일 유니티가 여전히 컴파일 중이라면 당신은 유니티의 메인 윈도우의 오른쪽 구석에 있는 spinning progress 아이콘을 볼 수 있다.

스크립트 컴파일은 4 단계로 수행된다:

1.“Standard Assets”, “Pro Standard Assets” 또는 “Plugins’ 안의 모든 스크립트들이 먼저 컴파일 된다.

이 폴더들 안의 스크립트들은 폴더 밖의 다른 스크립트들을 직접 액세스 할 수 없다.

클래스나 변수들을 참조할 수 없지만, GameObject.SendMessage 를 통해서 통신은 할 수 있다.

2. “Standard Assets/Editor”, “Pro Standard Assets/Editor” 또는 “Plugins/Editor”안의 모든 스크립트들이 먼저 컴파일된다.

만일 당신이 UnityEditor namespace 를 사용하고 싶다면 당신의 스크립트들을 이 폴더들에 넣어야 한다. 예를들면 메뉴 아이템 추가나 custom wizards 를 작성하기 위해서는 당신의 스크립트들을 이 폴더들 안에 넣어야 한다.

이 스크립들은 앞의 그룹으로부터 스크립트들을 액세스 할 수 있다.

3. "Editor" 안의 모든 스크립트들이 그다음 컴파일 된다.

만일 당신이 UnityEditor namespace 를 사용하기 원한다면 당신의 스크립트들을 이 폴더들에 넣어야 한다. 예를 들면 menu 아이템들을 추가하거나 custom wizards 를 작성하기 위해서는 당신의 스크립트들을 이 폴더들에 넣어야 한다.

이 스크립트들은 앞 그룹의 모든 스크립트들을 액세스 할 수 있다. 하지만 그들은 마지막 그룹 안의 스크립트들을 액세스 할 수 없다.

이것은 editor 코드를 작성할 때 문제가 될 수 있다. 왜냐하면 당신은 자주 마지막 그룹 안의 스크립트들을 edit 하고 싶어지기 때문이다. 여기에는 두개의 솔루션이 있다: 1. 다른 스크립트를 "Plugins"폴더로 이동한다. 2. 자바스크립트의 dynamic typing 의 장점을 취한다. 자바스크립트 내에서 당신은 그것과 함께 작동하는 클래스의 type 을 알 필요가 없다. GetComponent 를 사용할 때 type 대신 string 을 사용할 수 있다. 또한 string 을 취하는 SendMessage 를 사용할 수 있다.

4. 다른 모든 스크립트들이 마지막에 컴파일 된다.

위 폴더에 있지 않은 모든 스크립트들이 마지막으로 컴파일 된다.

이 단계에서 컴파일 되는 모든 스크립트들은 첫번째 그룹("Standard Assets", "Pro StandardAssets' 또는 "Plugins")에 있는 모든 스크립트들을 액세스한다. 이것은 서로 다른 스크립트 언어들이 상호연동될 수 있도록 해준다. 예를 들면, 만일 C#스크립트를 사용하는 Javascript 를 만들고 싶으면: "Standard Assets"폴더 안에 C#스크립를 놓고 Javascript 는 "Standard Asets"폴더 외부에 놓는다. 자바스크립트는 이제 C#스크립트를 직접 참조할 수 있다.

첫번째 그룹 안에 놓인 스크립트들이 컴파일 되면 세번째 그룹도 재컴파일 되어야 하므로 컴파일에 시간이 더 걸릴 것이다. 따라서 컴파일 시간을 단축 시키고 싶다면, 덜 바뀌는 스크립트들을 첫번째 그룹으로 옮기고 많이 바뀌는 것은 4 번째 그룹으로 옮겨야 한다.

Conditional compilation against the Unity version.

유니티 2.6 버전부터 C# 전처리기 정의가 추가되었다. 이 정의는 사용중인 유니티의 버전을 identify 하고 특정 features 를 조건적으로 액세스할 수 있게 해준다. 예를 들면:

```
// Specific version define including the minor revision
#if
UNITY_2_6_0
// Use Unity 2.6.0 specific feature
#endif

// Specific version define not including the minor revision
#if UNITY_2_6
// Use Unity 2.6.x specific feature
#endif
```

이 코드는 유니티의 특정버전에서만 가능한 game feature들을 enable 시키는데 사용될 수 있다. 이 정의는 오직 2.6버전부터 제공된다는 것에 주의하라. 앞으로의 유니티 버전들은 당신의 스크립트의 버전을 identify하기 위한 적절한 정의를 제공할 것이다.