SON GÜNCELLEME: 10 Ocak 2016

Bu kılavuz Süleyman Yasir KULA tarafından hazırlanmıştır/hazırlanmaktadır.

Bloguma gitmek için tıklayın: http://yasirkula.com
E-mail: yasirkula@yahoo.com

-- KOMUTLAR NASIL OKUNMALI --

Örneğin "*Instantiate(obje : GameObject, pozisyon : Vector3, rotasyon : Quaternion)*" (Teknik tabirle bu gösterime *Method Signature* denir) komutunu (metod) ele alalım. Burada görüldüğü üzere bu metod 3 parametre (argüman) almakta: *obje*, *pozisyon* ve *rotasyon*. Bu argümanların hemen sağında iki nokta işareti '*:*' bulunmakta ve onun da sağında bu argümanların alması gereken tür (type) yazmakta; yani sırasıyla: *GameObject*, *Vector3* ve *Quaternion*. Bu metodu kendi oyunumuzda kullanırken ise türünü yazmaya gerek yoktur (yazılmamalıdır da zaten); yani örneğin *Instantiate* metodu "*Instantiate(gameObject, transform.position, Vector3.zero);*" şeklinde kullanılabilir.

NOT: Bu kılavuzda yer alan herhangi bir örnek kodda eğer ki " " (Tırnak) işaretleri varsa, kodu Unity'e yapıştırdıktan sonra o tırnak işaretlerini her birini silin ve tekrar elle tırnak işareti girin. Çünkü MS Word'ün 'tırnak işareti' Unity'nin algıladığı 'tırnak işareti'nden biçimce farklı ve bu yüzden eğer tırnak işaretlerini kendiniz düzenlemezseniz Unity tarafından 'bilinmeyen karakter' hatası alırsınız. (EKLEME: Yaptığım birkaç ayar sonucu sanırım bunun üstesinden geldim. Artık böyle bir hata çıkacağını sanmıyorum ama yine de bu yazı burada kalsın.)

- # COMPONENTLER

Transform: Bir objenin 3 boyutlu uzaydaki konumunun, eğiminin ve boyutlandırmasının (Herhangi bir eksendeki boyut olarak uzama ya da kısalma miktarı)(Scale) depolandığı, tüm GameObject'lerde ortak olan Component

RigidBody: Fizik olaylarının bir objeye sağlanmasını sağlayan Component (örneğin yerçekimi)

Collider: Bir objenin temas edilebilir olan kısımlarını ayarlayan Component. Bu component'teki "*Is Trigger*" seçeneği işaretlenirse, herhangi bir obje; bu objeyle temas hâline geçince durmaz, içinden geçer. Fark budur. Ayrıca bu sefer "*Collision*" fonksiyonları yerine "*OnTrigger*" fonksiyonları kullanılır.

NOT: Eğer bir obje için Mesh Collider kullanıyorsan, bu obje normal şartlarda başka Mesh Collider'a sahip objelerle temas etmez. Ancak eğer iki Mesh Collider'ın da birbiriyle temas etmesini istiyorsan objenin Inspector panelinde Mesh Collider'ın altından "*Convex*"i seç.

AudioListener: Bir sahnede sadece 1 tane olmalıdır. Oyun mekanındaki tüm ses dosyaları, bu objenin yakınlığına, temasına vb. şeylerine göre, yani bu objeye göre duyulur, dinlenir. Bir başka deyişle AudioListener component'ine sahip olan obje, oyuncunun oyundaki kulağıdır.

Mesh Renderer: Objeyi sahnede çizdirmeye, yani onu görünür kılmaya yarayan Component. Eğer başındaki tik işareti kaldırılırsa obje görünmez olur.

Trail Renderer: Objenin hareketi esnasında arkasından bir şerit çıkmasını sağlar. Örneğin uçakların kanatlarından çıkan izi simüle etmede ya da "Tron Efsanesi" filmindeki, arkasından düz şerit çıkaran motosikletlerdeki gibi iz çıkarmada kullanılabilir. Materyal olarak genelde Partikül materyalleri kullanılır.

AudioSource: Bir objeyi ses kaynağı olarak yetkilendirir. Objenin ses çıkarabilmesi için bu component'e sahip olması gereklidir (aslında bu component'i kullanmadan da ses çalmaya yarayan bir metod var ancak genel anlamda ses çalmak için bu component'in kullanılmasını tavsiye ederim). AudioSource component'ine sahip bir obje birden çok ses çalabilir, kısıtlama yoktur.

Fixed Joint: İki objeyi birbirine bağlamaya yarar. Ancak iki obje arasında herhangi bir hareketli olay olmaz. Sadece ikisi bitişiktir. Objelerin ikisinin de RigidBody'e sahip olması gereklidir.

Hinge Joint: İki objeyi birbirine bağlar. Bu sefer ayrıca iki obje arasında, hareket esnasında gerçekçi hareketli bir bağlantı oluşur. Tıpkı dönüp duran bir zincirin halkaları gibi. Objelerin ikisinin de RigidBody'e sahip olması gereklidir.

- # DEĞİŞKENLER (VARIABLE)

public var: Başka scriptler tarafından ulaşılıp değiştirilebilir ve Inspector'da gözükebilir.

private var: Başka scriptler erişemez ve Inspector'da değeri gözükmez.

static var: Bu değişkenler özeldir ve bir objeden sahnede kaç tane olursa olsun bu değişkenden sadece 1 tane vardır. Örneğin Dusman adında bir class (script) olsun. Oyunun bir zorluk seviyesi olsun ve bu zorluk seviyesi tüm düşmanlar için geçerli olsun (public static var zorluk). Oyunda kaç tane düşman olursa olsun tüm düşmanlar için tek bir zorluk değişkeni bulunmakta, yani zorluğun düşmandan düşmana değişme göstermesi mümkün değil. Bu durumda bu statik değişkene ulaşmak için direkt "*zorluk*" diye değişkenin adı kullanılmaz da "*Dusman.zorluk*" diye, değişkenden önce class'ın ismi yazılır (statik değişkenleri çağırmada kural böyledir). Örnekten de görüleceği üzere statik değişkenler public veya private olabilir.

*****Bir değişkenin değerini Inspector'dan değiştirirsen o değişkenin değeri script'teki değil de Inspector'daki varsayılarak alınır. Ayrıca Inspector'dan değiştirdiğin bir değişkenin değeri sadece Inspector'da değişir, scriptin kendisindeki varsayılan değer değişmez! Bu da Unity'nin önemli özelliklerinden biridir.*****

*****Bir değişken adı küçük harfle başlar. Ayrıca örneğin "benimAdim" şeklindeki bir değişken adı Inspector'da "Benim Adim" şeklinde gözükür.*****

*****Bir fonksiyon adı büyük harfle başlar.*****

*****Bir class (Sınıf)(Script) (Birden çok fonksiyonun ve değişkenin depolandığı birimler de diyebiliriz.) adı büyük harfle başlar.**

- # DEĞİŞKEN TİPLERİ

NOT: Burada gösterdiğim değişken tipleri dışında daha bazilyon tane daha değişken türü bulunmakta. Bu değişkenleri kullana kullana öğreneceksiniz.

int: Tamsayı

float: Virgüllü sayı

double: float'tan daha büyük üst ve alt limite sahip olan ve daha küsüratlı olabilen virgüllü sayı

NOT: Javascript'te çalışırken virgüllü sayıları direkt kullanabilirsin (*0.5* gibi) ancak *C#*'ta çalışırken *0.5* yazarsan Unity bunu otomatik olarak *double* olarak algılar ve çoğu metod *double* sayılarla değil de *float* sayılarla çalıştığı için hata verir. Bunu düzeltmenin (*double* değeri *float*'a çevirmenin) yolu ise oldukça basittir: sayının sonuna " f " harfi konur (*0.5f* gibi)

boolean: True veya false değerini alan, en basit değişken tipi (*C#*'ta bu değişkenin ismi bool'dur.)

String: Yazı şeklinde değer alır; "*Merhaba Dünya!*" (**Tırnak işaretleri de var!**) gibi. '*int*', '*float*' gibi türlerin aksine bunun ilk harfi büyüktür: '*String*'

NOT: C#'ta bu tür string diye geçmektedir, yani Javascript'in aksine küçük harfle başlar.

GameObject: Herhangi bir oyun objesi

AudioClip: Ses Dosyası

Light: Sahne ışıkları için kullanılır.

ParticleEmitter: Partikül oluşturucular (emitter) için kullanılır.

Color: Renk depolamaya yarar. Değeri şöyle ayarlanabilir:

```
var renk : Color = Color(0.4, 0.5, 0.23, 0.8);
```

İçerideki 4 adet parametrenin ilk üçü rengin RGB'si (Yani kırmızılık, yeşillik, mavilik oranı. Bilgisayarda renkler genel anlamda bu 3 rengin farklı oranlarda karışmasıyla oluşturulur.) ve sonuncu parametre de rengin saydamlığı. Değeri 1 olursa tamamen opak, 0 olursa tamamen görünmez olur. Tüm değerler 0-1 arasında değer alır.

Dilenirse şöyle de renk tanımlanabilir:

```
var renk : Color = Color.red;
```

Vector3: Uzaydaki bir noktanın koordinatlarını depolamaya yarar. Uzayda noktalar x,y ve z koordinatlarından oluşur ve bu yüzden aslında bu değişken içinde 3 değer bulundurur:

'*Vector3(x,y,z)*' şeklinde. Ancak tanımlama yaparken sadece '*Vector3*' yazılır. Örnek bir kullanımı:

```
var konum : Vector3 = transform.position;
```

Ardından bu scriptin içinde; örneğin '*konum.x*' dersek bu o scriptin uygulandığı objenin '*x*' koordinatındaki konumunu verir.

Vector2: 2 boyutlu düzlemde bir nokta (Mesela ekranın kendisinde bir nokta) belirtir. x ve y koordinatlarından oluşur.

Transform: Bir GameObject'in Transform componentini depolar.

NOT1: Aslında bu, Transform componenti olan bir GameObject'i depolamaya yarar. Mesela '*Rigidbody*' türündeki bir değişken kullanırsak bunun anlamı Inspector'dan oraya bir GameObject sürüklediğimizde, sürüklediğimiz GameObject'te Rigidbody componentinin varlığının olmasını zorunlu kılmaktır. Yani eğer onda Rigidbody componenti yoksa değişkenin değeri olarak o objeyi atayamayız. Bunu pek çok component için yapabiliriz, mesela '*AudioSource*' componenti zorunluluğu da koyabiliriz vb.

NOT2: Bu tipte bir değişken depolamanın bir diğer kullanım alanı da script yazarken kolaylık sağlamasıdır. Mesela "*obje*" adında Transform tarzında bir değişkende direkt "*obje.position.x*" diyerek ilgili objenin *x* koordinatını bulabiliriz ancak eğer değişken türümüz GameObject olsaydı "*obje.transform.position.x*" yazmak zorunda kalırdık.

Texture2D: İki boyutlu resim dosyalarını depolamak için kullanılır.

GUISkin: Arayüz elemanlarıyla uğraşırken (Özellikle interaktif menü yapımında) Unity'nin hazır arayüz stili yerine kendi arayüz stillerimizi kullanabilmemiz için kullanmamız gereken değişken türü. Bu değişkene uygun stili proje panelinden "*Create – GUI Skin*" yoluyla oluşturabilirsin. Ardından "*OnGUI()*" fonksiyonunda "*GUI.skin = arayuzdegiskeni;*" komutuyla arayüze kendi oluşturduğun stili atayabilirsin.

KeyCode: Bir klavye tuşu depolamaya yarar. Daha sonra Input.GetKeyDown() ve benzeri komutlara parametre olarak atanabilir.

Touch: Mobil platformlarda ekrandaki bir parmağı özellikleriyle beraber depolamak için kullanılan bir Struct. Depoladığı özellikler ise "*position*", "*deltaPosition*", "*phase*" ve "*fingerId*"dir.

RaycastHit: Raycast kullanımında, raycast'ın bir nesneye çarparsa geri döndüreceği bilginin depolanacağı değişken türü.

Quaternion: Rotasyonu depolamaya yarar. Sanıldığının aksine objelerde gerçekçi bir rotasyon elde etmek için 4 boyutlu bir değişken türü, Quaternion kullanılır ve gerçekten karmaşık bir şeydir kendisi. Neyse ki işimizi kolaylaştırmak ve rotasyonlarla içli dışlı olmak için kullanımı rahat çeşitli Quaternion fonksiyonları mevcuttur.

Array: Bir ‘dizi’ oluşturmaya yarar. Diziler, içlerinde birden çok değeri depo edebilen özel değişkenlerdir.

ArrayList: Array’den daha gelişmiş özelliklere sahip olan bir dizi –ArrayList– oluşturmaya yarar. Bu listelerde sadece Object türünde değişkenler depolanabilir. ArrayList’ler hakkında daha detaylı bilgi: http://msdn2.microsoft.com/en-us/library/system.collections.arraylist_members%28VS.80%29.aspx

delegate: Normal bir değişken içerisinde bir veri tutmaya yararken bir delegate ise içerisinde bir (veya birden çok) fonksiyon tutmaya yarar. İsmi çok karmaşık bir veri türüymüş gibi gösterebilir ama aslında kullanımı basittir ve doğru kullanıldığında çok faydalı olabilir.

event: delegate'lere çok benzer ve bu veri türü de içerisinde fonksiyon(lar) tutar. Tek fark bir delegate'i tüm class'lar tetikleyebilirken (yani delegate'teki tüm fonksiyonların çalıştırılmasını sağlarken) bir event'i sadece içerisinde tanımlandığı script tetikleyebilir.

- # ÖNEMLİ FONKSİYON ADLARI

function Update(): Her bir stepte (frame, kare) gerçekleşen olayların yazıldığı fonksiyon.

function FixedUpdate(): Fizik olayları için (rigidbody Component'i ile yapılan olaylar) bu fonksiyon kullanılmalıdır. Update'ten tam olarak farkını anlamadım gerçi ama dediğim gibi, fiziksel olaylar buraya yazılmalı.

function LateUpdate(): Tüm objelerin *Update()* fonksiyonları çalıştırıldıktan sonra LateUpdate() fonksiyonları, yine her bir stepte çalıştırılır. Özellikle kameranın bir objeyi takip ettiği kodlar için kullanışlıdır (Araba yarışları vb.). Çünkü takip edilen obje *Update()*'in içerisinde hareket etmişse kamera objenin son konumuna göre ayarlanır ve bu sayede daha istenilen tarzda bir görüntü elde edilir.

function Awake(): *Start()* fonksiyonundan daha önce çalışan ve tıpkı onun gibi sadece bir kere çalıştırılan bir fonksiyon.

function Start(): Sadece script ilk çalıştırıldığında yapılacak olayların yazıldığı fonksiyon.

function OnCollisionEnter(): Scriptin uygulandığı obje herhangi başka bir objeyle temas (collision) yaşarsa gerçekleşir. Eğer '*function OnCollisionExit()*' yazılırsa olaylar temas kesilince, '*function OnCollisionStay()*' yazılırsa temasın olduğu her stepte gerçekleşir. Örnek bir kullanımı:

```
function OnCollisionEnter(temasEdilen : Collision)
{
	if(temasEdilen.gameObject.name == "zemin")
	{
		Debug.Log("Zemine vurdun!");
	}
}
```

function OnTriggerEnter(): Scriptin uygulandığı objenin Collider'inde "*Is Trigger*" seçeneği işaretliyse, Collider'ine bir obje temas edince bu komut çalışır. '*function OnTriggerExit()*' fonksiyonuyla temas kesilince, '*function OnTriggerStay()*' fonksiyonuyla da temas süresince yapılacakları ayarlayabilirsin. Bu fonksiyonun '*OnCollisionEnter()*'dan bir farkı, içerisine argüman girilecekse argümanın türünün "*Collision*" değil de "*Collider*" olmasıdır! Örnek bir kullanımı:

```
function OnTriggerEnter(temasEdilen : Collider)
{
	if(temasEdilen.gameObject.name == "Yukseltici Zemin")
	{
		rigidbody.AddForce(Vector3.up*10);
	}
	else if(temasEdilen.gameObject.name == "Zaman Yavaslatici Zemin")
	{
```

```
                Time.timeScale = 0.5;
        }
}

function OnTriggerExit(cikisYapilan : Collider)
{
        if(cikisYapilan.gameObject.name == "Zaman Yavaslatici Zemin")
        {
                Time.timeScale = 1.0;
        }
}

@script RequireComponent(Rigidbody)
```

Bu örnekte scriptin uygulandığı obje '*Yukseltici Zemin*' adlı GameObject'in triggerına girerse, içerde olduğu süre boyunca ona yukarı yönde 10 birimlik güç uygulanır. Eğer '*Zaman Yavaslatici Zemin*'in triggerine girerse zaman yavaşlar, yarı hızına düşer yani Slow-motion olur. Ardından '*OnTriggerExit()*' fonksiyonunda da eğer '*Zaman Yavaslatici Zemin*'den çıkarsak zamanı normal hâline döndürüyoruz ki sürekli Slow-motion modunda kalmasın oyun.

NOT: Collider'de '*Is Trigger*' seçince ne olduğunu merak ediyor olabilirsiniz. Haklısınız da, heralde sadece fonksiyonların isimleri değişmiyordur. Eğer '*Is Trigger*' seçeneğini seçerseniz Collider'i olan bir obje '*Is Trigger*' olan objeyle temas edince normal temasların aksine durmaz, objenin Collider'inin içinden sanki hiçbir şey yokmuş gibi geçer ve yoluna devam eder. Eğer '*Is Trigger*' seçeneği seçili olmazsa iki obje temas edince hareket hâlindeki obje ötekinin içinden geçemez, ona toslar.

function OnMouseDown(): Scriptin uygulandığı bir GUI objesine ya da bir objenin Collider'ine mouse ile tıklandığı anda gerçekleşir. Eğer '*function OnMouseDrag()*' yazılırsa mouse ile basılı tutulduğunda gerçekleşir. Veya '*function OnMouseUp()*' yazılırsa mouse ile tıklanma kesildiği anda olaylar gerçekleşir.

function OnMouseEnter(): Scriptin uygulandığı bir GUI objesinin veya bir objenin Collider'inin üzerine mouse ile gelindiğinde gerçekleşir. Eğer '*function OnMouseExit()*' yazılırsa mouse ilgili objenin, GUI'nın üzerinden çekilince olaylar gerçekleşir. Veya '*function OnMouseOver()*' yazılırsa mouse ilgili objenin, GUI'nin üzerinde olduğu her step'te ilgili olaylar gerçekleşir.

function OnDrawGizmos(): Scene panelinde temsili şekiller çizmeye yarar. Örneğin Point Light'taki lamba sembolü, Collider'daki dikdörtgen gibi semboller ve şekiller bunların arasındadır. Oyuna herhangi bir etkisi yoktur, bu gizmo'lar sadece Scene panelinde gözükür ve oyun yapımcısına görsel anlamda kolaylık sağlar.

function OnDrawGizmosSelected(): Üstteki fonksiyondan tek farkı, bu fonksiyonda çizdirilen gizmo'lar sadece obje Scene panelinde seçiliyse görünür ancak üstteki OnDrawGizmos() fonksiyonunda gizmo'lar obje seçili olmasa da görünür durumdadır.

function OnBecameVisible(): Obje herhangi bir kamera tarafından görüldüğü zaman gerçekleşir. Bu kameralara Scene panelindeki gezmeye yarayan kamera da dahildir. Bu

özellik performans açısından faydalı olabilmektedir. Bunun için OnBecameInvisible() fonksiyonuyla beraber kullanımı önemlidir.

function OnBecameInvisible(): Obje hiçbir kamera tarafından görülmediği zaman gerçekleşir. Örneğin bir objedeki bir scripte sadece obje kameraların en az biri tarafından görüldüğü zaman ihtiyaç duyuluyorsa, o scriptte şöyle bir kod yazılarak performansta artış sağlanabilir:

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

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

function OnLevelWasLoaded(): Oyun sırasında *Application.LoadLevel()* komutuyla yeni bir Scene yüklendiğinde çalıştırılır. Ancak oyun ilk başladığı anda çalışmaz. Dediğim gibi, bölümün elle değiştirilmesiyle çalıştırılır.

function OnEnable(): Script "*enabled = true;*" komutuyla aktif edildiğinde çalıştırılır.

function OnDisable(): Script "*enabled = false;*" komutuyla deaktif edildiğinde çalıştırılır.

function OnDestroy(): Script veya GameObject "*Destroy()*" komutuyla yok edildiği zaman, yok olma işlemi bitmeden önce gerçekleştirilir.

function OnApplicationQuit(): Oyun kapatılmadan hemen önce çalıştırılır.

function Reset(): Bir component'in sağında yer alan dişli ikona tıklayıp Reset seçeneğini seçersen o component'teki Reset fonksiyonu çalışır.

function OnApplicationFocus(focus : boolean): Çok kullanışlı bir fonksiyon. Eğer oyun açıkken kullanıcı Windows tuşuyla masaüstüne dönerse ya da bir uygulama açarsa vb., yani kısaca başka bir Windows işlemine focus yaparsa ve ardından tekrar oyuna geri dönerse çeşitli şeyler yapmak için birebir (Örneğin oyunu otomatik olarak pause etmek). Alttaki komutta kullanıcı ne zaman oyundan başka bir işle meşgul olursa oyun otomatik olarak duraklatılır ve kullanıcı oyuna geri dönünce oyun eski hızıyla kaldığı yerden devam eder:

```
private var oyunHizi : float = Time.timeScale;
function OnApplicationFocus( odaklanma : boolean ) {
	if( !odaklanma ) /* Kullanıcı başka işlerle meşgul olmaya başladıysa */
	{
		oyunHizi = Time.timeScale;
		Time.timeScale = 0;
	}
	else /* Kullanıcı oyuna geri döndüyse */
	{
		Time.timeScale = oyunHizi;
	}
```

```
}
```

- # ÇEŞİTLİ ÖNEMLİ METODLAR (KOMUTLAR)
- ## OBJEYİ KONUMLANDIRMA - TRANSFORM

transform.Translate(xDegisimi : float, yDegisimi : float, zDegisimi : float): Bir objenin konumunu (Inspector panelindeki *Transform* Componentinde yer alan *Position* değerlerini) değiştirmeye yarar.

transform.Rotate(xDegisimi : float, yDegisimi : float, zDegisimi : float): Bir objenin eğimini, yani rotation'ını (Inspector panelindeki *Transform* Componentinde yer alan *Rotation* değerlerini) değiştirmeye yarar.

transform.eulerAngles.x, transform.eulerAngles.y, transform.eulerAngles.z: Bir objenin herhangi bir eksendeki rotation açısını verir. Eğer obje bir başka objenin child'ı ise genelde bu değerler Transform component'indeki değerlerle aynı çıkmaz çünkü bu değişkenler global uzaydaki eğimi vermektedir. Bu değerler kesinlikle teker teker değiştirilmemelidir. Ancak onun yerine bir objenin açısını şöyle değiştirebilirsin: *transform.eulerAngles=Vector3(x,y,z)*

transform.localEulerAngles.x, transform.localEulerAngles.y, transform.localEulerAngles.z: Bir objenin herhangi bir eksendeki rotation açısını verir. Bu değişkenler daima Transform componentindeki rotasyon değerlerini döndürür (Objenin hiyerarşisinden bağımsız bir şekilde).

transform.TransformDirection(Vector3.right): Objenin kendine has (Local) X eksenindeki (Bunu objeyi seçince Scene panelinde kırmızı ok olarak görebilirsin.) sağ yönü depolamaya yarar. Mesela ne işe yarar? Şöyle bir örnek vereyim:

```
rigidbody.AddForce(transform.TransformDirection(Vector3.right)*20)
```

Bu kodun uygulandığı objede güç objenin kendi X ekseninde sağa doğru uygulanır. Yani tam olarak Scene panelindeki objenin kendi kırmızı oku yönünde uygulanır. Peki şöyle deseydik ne olurdu:

```
rigidbody.AddForce(Vector3.right*20)
```

Böyle yapsaydık gücün uygulanacağı yön Scene panelinin sağ üstündeki kırmızı ok yönünde olurdu, yani evrensel (World Space) sağ yönde olurdu. Bunu daha iyi anlamak için bir objeyi '*E*' tuşuyla kırmızı yönde bir miktar çevir. Ardından '*W*' tuşuna bastığında göreceksin ki objenin kırmızı okunun yönü değişmiş ama Scene panelinin sağ üstündeki kırmızı okun yönü hiç değişmemiş. Çünkü evrensel sağ yönü önceden belirlenmiş sabit bir şeydir ve objenin yönüne göre vb. değişmez.

transform.LookAt(hedefObje : Transform): Scriptin uygulandığı objenin, kodun içerisindeki 'hedefobje' isimli objeye bakacak şekilde rotation değerlerinin otomatik olarak ayarlanmasını sağlar. Örneğin bir spot lambanın sürekli oyuncuya doğru bakmasını ayarlamak için kullanılabilir. Ancak kullanım alanı oldukça geniştir. ÇOK ÖNEMLİ bir not: '*Hedef Obje*'nin

bir ‘*Transform*’ olması lazım. Bunun için ‘*Transform*’ olarak tipi belirlenmiş bir variable (değişken) tanımlayıp onu ilgili oyun objesine eşitlemek yeterlidir.

Input.mousePosition : Vector3 : Mousenin ekrandaki (Ekran 2 boyutlu bir düzlemdir.) konumunu verir, değeri sadece okunabilir. Eğer mouse ekranın en sol üstündeyse değeri ’ *(0,0,0)* ’, ekranın en sağ altındaysa değeri, ekranın genişliği ve yüksekliği olan ‘*(Screen.width,Screen.height, 0)*’dır. Görüldüğü gibi farenin Z konumu daima 0'dır.

- **OBJE – COMPONENT'LERLE İLGİLİ KOMUTLAR**

GameObject.Find("obje" : String): "obje" isimli objeyi bulmaya yarar. Bu kodu örneğin o objeyi bulup sonradan kullanmak için bir değişkende depolamak için kullanabilirsin.

GameObject.FindWithTag("Tag Adı" : String): Girilen tag'a sahip bir objeyi bulmaya yarar.

gameObject.name : String : Bir objenin ismini verir. Genelde bir objeyi bir değişkende saklıyorsak o değişken vasıtasıyla ilgili objenin adını bulmak için kullanılır.

*.enabled: Bir componentin adı girilip yazılırsa o componentin aktifliğini değiştirmeye yarar. Eğer değeri 'true'ya eşitlenirse component aktif olur, 'false'ye eşitlenirse component etkisiz hâle gelir.

renderer.enabled : boolean : Kodun uygulandığı objeyi, eğer değeri '*false*' yapılırsa görünmez yapmaya; değeri '*true*' yapılırsa tekrar görünür yapmaya yarar.

renderer.material.color : Color : Kodun uygulandığı objeye uygulanmış olan materyalin ana rengini, hâliyle objenin rengini değiştirmeye yarar. Değeri örneğin '*Color.red*' veya '*Color.green*' yapılabilir.

transform.parent : Transform : İlgili objenin Parent objesini seçmeye yarar. Mesela bir evin kapısına bunu uygularsan büyük olasılıkla evin kendisini GameObject olarak elde etmiş olursun.

transform.root : Transform : Eğer bir obje başka bir objenin Child objesiyse ve bunun Parent objesi de daha başka bir objenin Child objesiyse, yani karmaşık sayılabilecek bir Parent hiyerarşisi varsa kodun uygulandığı objenin en üst düzeyde Parent'ı olan objeyi seçmeye yarar. Yani tüm bu hiyerarşideki ata Parent objeyi seçer. Ardından "*transform.root.rigidbody.enabled = false;*" gibi o ata objede istediğin değişikliği yapabilirsin.

SendMessage("Fonksiyon Adı" : String, Eğer varsa argümanlar): Bir objedeki, ismi girilen fonksiyonu; tercihe bağlı olarak değeri girilmiş argümanlarla (İlgili argümanların fonksiyonda tanımlı olması lazım.) çağırmaya yarar.

SendMessageUpwards("Fonksiyon Adı" : String, Eğer varsa argümanlar): *SendMessage*'dan tek farkı, bu komut kullanılınca scriptin yazıldığı objenin dışında ayrıca o objenin tüm ata objeleri için de *SendMessage* komutu çalıştırılır, eğer onlarda da "*Fonksiyon Adı*" adında bir fonksiyon varsa bu fonksiyon onlarda da çalıştırılır.

BroadcastMessage("Fonksiyon Adı" : String, Eğer varsa argümanlar): *SendMessage*'dan tek farkı, bu komut kullanılınca scriptin yazıldığı objenin dışında ayrıca o objenin tüm child objeleri için de *SendMessage* komutu çalıştırılır, eğer onlarda da "*Fonksiyon Adı*" adında bir fonksiyon varsa bu fonksiyon onlarda da çalıştırılır.

objeadi.GetComponent("Component Adı" : String): İsmi girilen objenin ismi girilen componentini seçmeye yarar. Bu component hazır bir bileşen de olabilir, o objeye atanmış herhangi bir script kod parçası da.

objeadi.GetComponentInChildren("Component Adı" : String): İsmi girilen objeden başlamak üzere, ismi girilen componenti seçmeye yarar. Eğer objenin kendisinde component yoksa hiyerarşik bir sıraya göre child objelerine de bakılır ve hiçbirinde bu component yoksa *null* döndürülür.

objeadi.AddComponent("Component Adı" : String): İlgili objeye, ilgili component'i; eğer yoksa eklemeye yarar. Ancak eğer o component zaten ilgili objede mevcutsa herhangi bir şey gerçekleşmez.

Instantiate(obje : GameObject, pozisyon : Vector3, rotasyon : Quaternion): Girilen pozisyonda girilen rotasyona sahip bir '*obje*' objesi oluşturur.

GameObject.CreatePrimitive(PrimitiveType sekil): Unity'nin hazır basit şekillerinden oluşturmaya (*GameObject -> Create Other* menüsü altındaki geometrik şekiller) yarar. Oluşturulan şeklin türünü "*sekil*" parametresi belirler. Değeri *PrimitiveType.Plane* (2 boyutlu düzlem), *PrimitiveType.Cube* (küp), *PrimitiveType.Sphere* (küre), *PrimitiveType.Capsule* (uçları yumuşatılmış silindir vari olan kapsül şekli) ve *PrimitiveType.Cylinder* (silindir) olabilir. Değeri mantıken bir değişkene atılmalıdır ve daha sonra bu değişken vasıtasıyla oluşturulan şeklin konumu, rotasyonu ve boyutu ayarlanmalıdır.

Destroy(obje : GameObject): Bir objeyi yok etmeye yarar. Örnek bir kullanımı:

```
function Start()
{
        Destroy(gameObject.Find("masa"), 5);
}
```

Bu kodla script ilk uygulandığında "*masa*" isimli bir obje bulunup 5 saniye sonra yok edilir. Eğer anında yok edilmesi isteniyorsa '*, 5*' kısmı; yani virgül ve virgülden sonrası silinir. Eğer '*Destroy(gameObject)*' yazılırsa kodun uygulandığı obje yok edilir.

- **FİZİK – RIGIDBODY İLE İLGİLİ KOMUTLAR**

rigidbody.AddForce(gucMiktari : Vector3): Scriptin uygulandığı objede rigidbody varsa, Translate'in aksine, fizik unsurlarını yok saymadan, o objeye belirli bir yönde istediğimiz güçte bir kuvvet uygulamak için kullanılır. Örneğin "*rigidbody.AddForce(Vector3.up * 100)*" komutu objeye alttan 100 birimlik güç uygular. Bu komutla uygulanan gücün yönü global koordinat sistemine göredir. Yani objenin rotasyonuna bakmaksının Vector3.right sürekli Scene panelinin sağ üstündeki x eksenini (kırmızı eksen) ifade eder.

rigidbody.AddRelativeForce(gucMiktari : Vector3): rigidbody.AddForce()'tan tek farkı, gücün uygulandığı objenin rotasyonunun gücün yönünde etkin rol oynamasıdır. Yani uygulanan gücün yönü global değil de local (yerel) koordinat sistemine göre belirlenir.

rigidbody.velocity(gucMiktari : Vector3): Objeye uygulanan normal gücün üzerine (Relative) güç uygulamak yerine (AddForce komutu) objedeki toplam uygulanan gücü ayarlamaya yarar. Tıpkı *eulerAngles* fonksiyonuyla objenin rotasyonunun anında ve kesin değişmesi gibi. Mesela karakteri zıplatırken kullanılabilir.

rigidbody.AddTorque(torkMiktari : Vector3): Objeyi fizik motoru vasıtasıyla döndürmeye (tork vererek) yarar. Uygulanan tork global koordinat sistemine göredir.

rigidbody.AddRelativeTorque(torkMiktari : Vector3): Girilen vektör objenin local (yerel) koordinat sistemine göre yorumlanır, yani objenin rotasyonu torkun yönünde etkilidir.

Physics.Raycast(baslangicKonumu : Vector3, yon : Vector3, raycastHitDegiskeni (İsteğe bağlı) : RaycastHit, raycastUzunlugu : float): Başlangıç konumundan ilgili yöne ilgili uzunlukta bir raycast ışını yollar ve bu raycast ışını bir objenin collider'ine temas ederse fonksiyon "*true*" değerini döndürür. Eğer raycast ışını bir şeye çarpmazsa fonksiyonun döndürdüğü değer "*false*" olur. Ayrıca isteğe bağlı olarak bir değişken oluşturup tipini "*RaycastHit*" yaparsan ve o değişkenin adını metoddaki ilgili kısma yazarsan raycast ışınının temas ettiği obje hakkında çok detaylı bilgiler alabilirsin. Örneğin objeyle raycast'in başlangıç noktası arasındaki uzaklık, objenin adı vb. Ayrıca objenin "*gameObject*"i sayesinde onun üzerinde istediğin değişikliği yapabilirsin. Örnek bir kullanımı:

```
function FixedUpdate () {
	var vurus : RaycastHit;
	if(Physics.Raycast(transform.position, -Vector3.up, vurus, 5))
	{
		if(vurus.collider.gameObject.name == "Zemin")
		{
			rigidbody.AddForce(Vector3.up*1000*Time.deltaTime);
		}
	}
}
```

Bu kod sayesinde, kodun uygulandığı objeden aşağı yönde 5 metrelik bir raycast ışını yollanır ve eğer raycast ışını bir objenin *"collider"*iyle temas yaparsa "*vurus*" isimli "*RaycastHit*" türündeki değişken sayesinde o temas yapılan objenin adına bakılır ve objenin adı eğer "*Zemin*" ise kodun uygulandığı objenin rigidbody'sine yukarı yönde saniyede 1000'lik bir güç, raycast'in zeminle teması kesilene kadar uygulanır. Böylece obje zemine yaklaşınca script sayesinde üzerine yukarı yönde güç uygulanarak yükselir, bir miktar yükselip zeminle raycast işlemi artık yapılamayınca yerçekiminin etkisiyle tekrar aşağı düşmeye başlar ve bu sonsuza kadar tekrarlanır. Böylece güzel bir görüntü oluşur. Ancak objenin rigidbody component'indeki "*Mass*" isimli ağırlığı temsil eden değerine bağlı olarak '*1000*' değeri artırılabilir ya da azaltılabilir.

Physics.Linecast(baslangicKonumu : Vector3, bitisKonumu : Vector3, raycastHitDegiskeni : RaycastHit): *Physics.Raycast*'e çok benzer. Ondan farkı ise temas olup olmadığına bakan ışının bu komutta *baslangicKonumu* ile *bitisKonumu* arasında oluşturulması, yani başlangıç konumunun haricinde bitiş konumunun da belli olması ve haliyle ışının uzunluğuna gerek duyulmamasıdır.

Physics.IgnoreCollision(collider1 : Collider, collider2 : Collider, durum : boolean): Fizik motorunun "*collider1*" değişkeninde depolanmış obje ile "*collider2*" değişkeninde depolanmış obje arasındaki temasları ihmal etmesini (Eğer "*durum*" *true* yapılırsa) sağlar, yani bu 2 obje arasında herhangi bir fiziksel olay gerçekleşmez. Üçüncü parametresi "*false*" yapılırsa bu 2 obje arasındaki olası etkileşimlerde fizik motoru tekrar devreye girer. Örneğin "*Physics.IgnoreCollision(collider, kursunObjesi.collider, true);*" komutu bir kurşunun fırlatıldığı silahla ya da havadaki başka bir kurşunla temas edip istenmeyen görüntülere engel olması gibi şeyler için birebirdir.

Physics.OverlapSphere(kureKonum : Vector3, kureYaricap : float): Girilen *Vector3* konumunu merkez kabul eden ve *kureYaricap* kadar yarıçapa sahip zahiri bir kürenin içinde kalan veya onunla temas eden, *Collider* componentine sahip tüm GameObjectleri *Collider[]* türünde bir *array* olarak döndürür. Patlama gibi fiziki unsurlar için oldukça kullanışlıdır.

rigidbody.AddExplosionForce(gucMiktari : float, patlamaKonumu : Vector3, patlamaYaricapi : float, dikeyEkstraGuc : float): Gerçekçi patlama olayları için kullanılır. Obje eğer girilen *Vector3* şeklindeki *patlamaKonumu*'nda oluşan, *patlamaYaricapi* kadar yarıçapa sahip *gucMiktari* şiddetindeki bir patlamanın menzilindeyse Unity ona uygun bir doğrultuda uygun miktarda bir güç (*Force*) uygular. *dikeyEkstraGuc* değişkeni ise ne kadar büyürse patlama menzilindeki obje o kadar havaya fırlatılır, eğer değeri 0 yapılırsa dikey doğrultuda ekstra bir güç uygulanmaz. Örneğin:

```
/* (0,0,0) konumundaki, 10 metre yarıçaplı güçlü bir patlamanın scriptin yazıldığı objeye (Eğer obje menzildeyse) güç uygulamasını sağlar. Ayrıca objeye bir miktar ekstra dikey güç uygular. */

function Start() {
        rigidbody.AddExplosionForce( 15.0, Vector3.zero, 10.0, 2.0 );
}
```

rigidbody.MovePosition(yeniKonum : Vector3): Rigidbody'e sahip objeyi yeniKonum'a hareket ettirir ve bu esnada fizik olaylarını dikkate alır. Örneğin eğer yol üzerinde bir başka fizik objesi varsa o obje ile fiziksel bir etkileşim gerçekleşir.

rigidbody.MoveRotation(yeniEgim : Quaternion): Rigidbody'e sahip objenin eğimini değiştirir ve bu esnada fizik olaylarını dikkate alır. Örneğin obje bir küreyse ve kürenin tam üzerinde rigidbody'e sahip bir başka obje dengede duruyor ise, küreyi bu fonksiyon ile döndürünce kürenin üzerindeki objenin dengesi bozulur ve obje aşağı düşer.

RaycastHit Değişkeni.distance : float : Raycast için eğer bir RaycastHit değişkeni belirlenmişse, onun ismi ve ardından "*.distance*" yazılır. Raycast eğer bir objenin collider'iyle temas yaptıysa; o objeyle raycast'ın başlangıç noktası arasındaki uzaklığı verir.

RaycastHit Değişkeni.point : Vector3 : Raycast eğer bir objeye temas etmişse bu temasın 3 boyutlu uzayda tam olarak hangi koordinatlarda gerçekleştiğini depolar.

RaycastHit Değişkeni.collider : Collider : Raycast eğer bir objeye temas etmişse o temas edilen objenin çeşitli ayarlarını değiştirmek için kullanılır. Örneğin raycast'ın temas ettiği objeye "*rigidbody*" component'i eklemek için şöyle bir kod yazılır (RaycastHit değişkeninin adının '*vurus*' olduğu varsayılmaktadır.) :

```
vurus.collider.gameObject.AddComponent(Rigidbody);
```

- **EĞİMLE İLGİLİ (QUATERNION) KOMUTLAR**

Quaternion.FromToRotation(birinciDogrultu : Vector3, ikinciDogrultu : Vector3): **birinciDogrultu ile ikinciDogrultu arasında yer alan rotasyonu döndürür. Örneğin bir objenin tepesinin (y ekseninin, bir başka deyişle Vector3.up değişkeninin) sürekli Main Camera'nın baktığı yöne doğru bakması için şu basit kod kullanılabilir:**

```
function Update()
{
        transform.rotation = Quaternion.FromToRotation( Vector3.up,
camera.main.transform.forward );
}
```

Quaternion.AngleAxis(dereceMiktari : float, yon : Vector3): ***yonVektoru*'nun etrafında *dereceMiktari* kadar döndürülmüş bir rotasyon döndürür. Örneğin "*Quaternion.AngleAxis(30, Vector3.up)*" komutu *eulerAngles*'ı (0,30,0) olan bir *Quaternion* (rotasyon) döndürür.**

Quaternion.LookRotation(dogrultu : Vector3): **Bir doğrultu yönünde bakan bir rotasyon döndürür. Bir objenin sürekli başka bir objeye bakması için birebirdir. Örneğin:**

```
var dogrultu : Vector3;
var hedef : Transform;
function Update() {
        dogrultu = hedef.position – transform.position;
        transform.rotation = Quaternion.LookRotation( dogrultu );
}
```

- **OYUNCU İLE KLAVYE-MOUSE ETKİLEŞİMİ (INPUT)**

Input.GetAxis("Axis Adı" : String): İçerisine "*Horizontal*" ya da "*Vertical*" yazılır. Örneğin "*Vertical*" yazılmışsa, yukarı ok tuşuna basıldığında değeri 1, geri ok tuşuna basıldığında -1, hiçbirine basılmadığında 0 olur. Ancak sol veya sağ ok tuşuna basarsan bir şey olmaz. Olması için içinde "*Vertical*" yerine "*Horizontal*" yazmalıdır. Hareket scriptlerinde kullanılabilecek çok pratik bir koddur.

NOT: *Input.GetAxis()* komutunda yumuşatma işlemi uygulanmaktadır. Yani klavyede yukarı ok tuşuna basınca "*Input.GetAxis("Vertical");*" direkt 1 değerini döndürmez, onun yerine her bir frame'de azar azar artarak (0.1, 0.23 vb. şeklinde) kısa bir sürede 1 döndürülür. Yukarı ok tuşu bırakılınca da aynı şekilde hemen 0 döndürülmez, azar azar azalarak 0'a yaklaşılınır.

Input.GetAxisRaw("Axis Adı" : String): *Input.GetAxis()*'ten tek farkı bu komutta yumuşatma işlemi uygulanmamasıdır. Yani mesela "*Input.GetAxisRaw("Vertical");*" komutu, yukarı ok tuşuna basıldığı anda 1 döndürür.

Input.GetButton("Önceden Belirtilmiş Tuşun Özel Adı" : String): İlgili tuşa basılı tutulup tutulmadığını test eder. Eğer '*Input.GetButtonDown*' kodu kullanılırsa sadece tuşa basıldığı ilk anda olaylar gerçekleşir. '*Input.GetButtonUp*' kodu kullanılırsa da olaylar tuştan elimizi çekince gerçekleşir. Örnek bir tuş adı: '*Fire1*', mousenin sol tuşu veya *CTRL* tuşu yerine geçer.

Buraya girilebilecek tuş isimleri ve onların hangi butonlara ayarlandığı '*Edit – Project Settings – Input*' yoluyla açılan Inspector penceresinde '*Axes*' seçeneği altında görünür.

Input.GetKey(tusKodu : KeyCode): Bunun '*Input.GetButton()*'dan farkı, bu scriptte kullanabileceğimiz tuşların '*Input*' kısmında önceden tanımlanmış olan tuşlarla sınırlı olmaması. Bu script ile klavyedeki tüm tuşlar için işlem yapılabilir. İşlem, ilgili tuşa basılı tutulduğu sürece gerçekleşir. Eğer '*Input.GetKeyDown*' kullansaydık tuşa bastığımız anda, '*Input.GetKeyUp*' kullansaydık tuştan elimizi çektiğimiz anda gerçekleşirdi.

İçerisine girilebilecek '*KeyCode.Escape*' argümanı klavyedeki ESC tuşunu ifade eder. Onun yerine herhangi bir tuş da kullanılabilir; "*KeyCode.Backspace*" "*KeyCode.Keypad7*" "*KeyCode.UpArrow*" "*KeyCode.F1*" "*KeyCode.Home*" "*KeyCode.R*" "*KeyCode.T*" "*KeyCode.LeftControl*" "*KeyCode.Mouse1*" vb. pek çok tuş gibi…

Input.GetMouseButton(mouseTuşu : int): Mousenin bir tuşuna basılı tutulup tutulmadığını kontrol eder. Eğer '*Input.GetMouseButtonDown*' kodu kullanılırsa sadece ilgili tuşa basıldığında, '*Input.GetMouseButtonUp*' kullanılırsa da sadece o tuştan el çekilince ilgili olaylar gerçekleşir. 3 adet tanımlı tuş bulunur. Eğer değeri '0' girilirse sol mouse tuşu, '1' girilirse sağ mouse tuşu, '2' girilirse de orta mouse tuşu ele alınır.

Input.anyKeyDown : boolean: Mevcut karede (frame) herhangi bir mouse ya da klavye tuşuna basılıp basılmadığını döndürür. Daha önceki bir kareden basılı tutulan bir tuş true döndürmez, tuşa mevcut frame'de basılması gereklidir.

Input.anyKey : boolean: Herhangi bir mouse ya da klavye tuşuna basılıp basılmadığını döndürür. Daha önceki bir kareden basılı tutulan bir tuş da true döndürür, tuşa tam mevcut frame'de basılması gerekli değildir.

- **SES – MÜZİK KOMUTLARI**

audio.PlayOneShot(sesDosyasi : AudioClip): Bir ses dosyasını, tek bir kere çalmayı sağlar.

audio.Play(): Eğer ki objenin '*Audio Source*' componentine bir ses dosyası hâli hazırda atanmışa bu komut o ses dosyasını çalmaya yarar. Veya şöyle de kullanılabilir:

```
GameObject.Find("Araba").GetComponent(AudioSource).Play();
```

Bu komutla "*Araba*" objesinin *Audio Source* componentine atanmış olan sesi çalabilirsin.

audio.Pause(): Scriptin yazıldığı objede çalınan ses dosyasını duraklatmaya yarar. Daha sonra '*audio.Play()*' komutuyla çalmaya devam edilebilinir.

audio.Stop(): Scriptin yazıldığı objede çalınan ses dosyasını durdurmaya yarar.

AudioSource.PlayClipAtPoint(sesDosyasi : AudioClip, calinacagiKonum : Vector3): Ses dosyasını, uzayda herhangi bir konumda çaldırmaya yarar. Bu kodun güzel yanı, kodun uygulandığı objenin "*Audio Source*" Component'ine sahip olmak zorunda olmamasıdır. Çünkü bu kod gerçekleştirildiğinde sesi çaldıracak, geçici yeni bir obje, belirtilen konumlarda oluşturulur ve sesi çalıp bitirdikten sonra otomatikman kendisini yok eder.

- **ZAMANLA (SÜRE)(HIZ) İLGİLİ KOMUTLAR**

Time.deltaTime : float : Bir step yerine bir saniyeyle işlem yapılmasını sağlar. Örneğin; '*transform.Translate(Vector3.forward*5*Time.deltaTime)*' koduyla kodun uygulandığı objenin 1 saniyede 5 metre ileri gitmesi sağlanır.

Time.time : float : Oyun başladıktan bu yana geçen toplam süreyi saniye cinsinden sayar. Değeri değiştirilemez, sadece okunabilir.

yield WaitForSeconds(saniyeCinsindenBeklenecekSureMiktari : float): Kodun geri kalanının yapılmasından önce belirtilen süre kadar beklenilmesini sağlar.

Time.timeScale : float : Değeri 0 ile 1 arasında olmak zorundadır. Yaptığı şey zamanı yavaşlatmaktır. Yani slow-motion yapmanıza olanak tanır. Ayrıca eğer değeri 0 yapılırsa zaman tamamen durur, yani oyun "Pause" olur. Ardından değeri 1 yapılırsa zaman normale döner.

- ## ANDROID & iOS KOMUTLARI

NOT: Burada yazılan komutların üstünde basitçe geçeceğim çünkü bu komutları daha ayrıntılı bir şekilde ders olarak anlattım. Derse gitmek için tıklayın: http://yasirkula.com/2013/07/17/unity-ile-androide-uygulama-gelistirmek-1-dokunmatik-ekran-entegrasyonu/

Input.touches : Touch[]: Ekrandaki tüm parmakları *Touch* türünde depolayan bir değişken.

Touch.position : Vector2: Parmağın ekrandaki hangi koordinata dokunduğunu döndürür.

Touch.deltaPosition : Vector2: Eğer parmak ekranda hareket ettiyse bu hareketin yatayda ve dikeyde kaç pixel olduğunu döndürür.

Touch.phase : TouchPhase: Parmağın durumunu depolar. Daha açık konuşmak gerekirse parmağın ekrana yeni mi dokunduğunu (*TouchPhase.Began*), yoksa ekranda basılı halde hareket etmekte olan bir parmak olduğunu mu (*TouchPhase.Moved*), ekranda basılı halde duran ama hareket etmeyen bir parmak olduğunu mu (*TouchPhase.Stationary*) ya da ekrandan yeni kaldırılan bir parmak olduğunu mu (*TouchPhase.Ended*) depolar. Ekrana aynı anda beşten fazla parmağın dokunması gibi nadir olaylarda ise değeri *TouchPhase.Canceled* olur.

Touch.fingerId : int: O parmağa has olan bir değişken döndürür.

Input.deviceOrientation : DeviceOrientation: Telefonun hangi pozisyonda tutulduğunu belirtir. Yani cihazın yere dik ve düzgün bir şekilde (HOME butonu aşağıda olacak şekilde) mi (*DeviceOrientation.Portrait*), yere dik ama tepetaklak bir şekilde (HOME butonu yukarıda olacak şekilde) mi (*DeviceOrientation.PortraitUpsideDown*), yere dik ve sola yatırılmış bir şekilde mi (*DeviceOrientation.LandscapeLeft*), yere dik ve sağa yatırılmış bir şekilde mi (*DeviceOrientation.LandscapeRight*), yere paralel ve ekranı yukarı bakacak şekilde mi (*DeviceOrientation.FaceUp*) yoksa yere paralel ve ekranı aşağı bakacak şekilde mi (*DeviceOrientation.FaceDown*) tutulduğunu depolar. Eğer ki cihaz abuk subuk bir eğimle tutuluyorsa o zaman bu değişken "*DeviceOrientation.Unknown*" değerini alır.

Input.acceleration : Vector3: Telefonun hareket sensörünün telefonun mevcut rotasyonuyla ilgili hesapladığı değeri döndürür.

Handheld.Vibrate(): Telefonu titretmeye yarar.

Screen.orientation : ScreenOrientation: Oyunun düz ekran mı yoksa yatay ekran mı oynandığını döndürür ve değeri değiştirilerek yatay ekranla düz ekran arasında geçiş yapılabilir. Varsayılan olarak değeri *ScreenOrientation.Portrait*'tir ve bu düz ekran anlamındadır. Bu değer *ScreenOrientation.LandscapeLeft* ile değiştirilerek oyun sola yatık yatay ekran olarak, *ScreenOrientation.LandscapeRight* ile değiştirilerek de sağa yatık yatay ekran olarak oynanabilir. Veya *ScreenOrientation.PortraitUpsideDown* ile değiştirilerek düz ama tepetaklak ekran olarak oynanabilir oyun. Son olarak da, değeri

eğer *ScreenOrientation.AutoRotation* ile değiştirilirse ekranı döndürdükçe oyun ekranı da otomatik olarak yatay ekrana ya da dikey ekrana geçiş yapar.

SystemInfo.deviceModel: Cihazın modelini döndürür (iPhone 3.1 gibi).

Application.genuine: Uygulamanın korsan olup olmadığını döndürür. AppStore'daki ücretli uygulamaları kırıp ücretsiz olarak sunan korsanlara karşı kullanılan bir tedbirdir. Eğer Application.genuineCheckAvailable komutu *true* döndürüyorsa ve Application.genuine komutu komutu da *false* döndürüyorsa o zaman oyununuzun kullanıcının o an kullanmakta olduğu versiyonu büyük olasılıkla korsanlar tarafından kırılmış versiyonudur ve bu durumda çeşitli tedbirler almak isteyebilirsiniz. Bu komutu kullanırken işlemciye çok yük biner ve bu yüzden *Update()* gibi bir fonksiyonda kullanılmamalıdır!

- **WEBCAM KOMUTLARI**

WebCamTexture.devices : WebCamDevice[]: Donanıma bağlı olan tüm webcam'leri, kameraları döndürür.

webcam.name : String: Girilen webcam cihazının okunabilir, özgün ismini verir. Buradaki "*webcam*" değişkeni *WebCamDevice* türündedir.

webcam.isFrontFacing : boolean: Girilen webcam cihazının ön kamera olup olmadığını depolar. Yani eğer kamera kullanıcıya bakıyorsa değeri *true*, arkaya bakıyorsa *false*'dir. Buradaki "*webcam*" değişkeni *WebCamDevice* türündedir.

WebCamTexture(cihazAdi : String): İsmi girilen webcamin görüntüsünü anlık depolayacak olan bir texture oluşturmaya yarar. Bu bir constructor'dır ve değeri *WebCamTexture* türünde bir değişkene aktarılmalıdır. Ardından bu WebCamTexture herhangi bir materyale atanarak o materyale sahip objelerin kameranın görüntüsünü render etmesi sağlanabilir. Eğer ki metodun içi boş bırakılırsa (*WebCamTexture()* şeklinde) cihaza bağlı ilk webcamin görüntüsünü depolayan bir *WebCamTexture* döndürür. *cihazAdi*'na atanabilecek kamera isimleri için *WebCamTexture.devices* ve *webcam.name* komutları kullanılabilir.

webcamGoruntusu.Play(): Girilen *webcamGoruntusu*'nün bağlı olduğu kameranın anlık görüntü vermeye başlamasını sağlar, yani kamerayı çalıştırır. *webcamGoruntusu* değişkeni *WebCamTexture* türünde olmalıdır.

webcamGoruntusu.Pause(): Girilen *webcamGoruntusu*'nün bağlı olduğu kameranın anlık görüntü vermeyi duraklatmasını sağlar. Webcam sonradan *webcamGoruntusu.Play()* ile çalışmaya devam ettirilebilir. *webcamGoruntusu* değişkeni *WebCamTexture* türünde olmalıdır.

webcamGoruntusu.Stop(): Girilen *webcamGoruntusu*'nün bağlı olduğu kameranın anlık görüntü vermeyi kesmesini sağlar. Webcam sonradan *webcamGoruntusu.Play()* ile tekrar çalıştırılabilir. *webcamGoruntusu* değişkeni *WebCamTexture* türünde olmalıdır.

webcamGoruntusu.isPlaying : boolean: Girilen *webcamGoruntusu*'nün bağlı olduğu kameranın o anda çalışıp çalışmadığını kontrol eder. *webcamGoruntusu* değişkeni *WebCamTexture* türünde olmalıdır.

webcamGoruntusu.videoRotationAngle : int: Her kameranın kendine has bir çekim açısı olabilir. Yani çektiği görüntüyü düzleştirmek için kendi etrafında 90, 270 veya başka bir derece döndürüyor olabilir. Bu değişken de bu çekim açısını depoluyor. Bunun kullanım alanı ise çok önemli. Çünkü bu değişkeni kullanarak kameranın çekim açısına bakmaksızın görüntünün objeye düzgün aktarılması sağlanabilir. Bunun için yapılması gereken şey ise scripte şu kodu eklemek:

```
var webcamGoruntusu : WebCamTexture;
var anaRotasyon : Quaternion;

function Start () {
        webcamGoruntusu = new WebCamTexture();
```

```
        renderer.material.mainTexture = webcamGoruntusu;
        anaRotasyon = transform.rotation;
        webcamGoruntusu.Play();
}

function Update () {
        transform.rotation = anaRotasyon * Quaternion.AngleAxis(
webcamGoruntusu.videoRotationAngle, Vector3.up );
}
```

- **SAVE – LOAD SİSTEMİ**

NOT: Buradaki PlayerPrefs komutlarıyla yapılan kayıtlar Windows'ta Registry Editor'de "*HKCU\Software\[şirket ismi]\[ürün ismi]*" konumuna, MAC OS X'te ise " ~/Library/Preferences" konumundaki "*unity.[şirket ismi].[ürün ismi].plist*" adlı bir dosyaya kaydedilmektedir. Eğer oyun *WebPlayer* üzerinden çalıştırılıyorsa o zaman Windows'ta "*%APPDATA%\Unity\WebPlayerPrefs*" konumunda, MAC OS X'te ise "*~/Library/Preferences/Unity/WebPlayerPrefs*" konumunda kayıtlar yapılır.

PlayerPrefs.SetInt("Anahtar İçin İsim" : String, deger : int): Bir *int* değişkenin değerini depolamaya yarar. Buradaki "*Anahtar İçin İsim*", değişkenin ismiyle aynı olmak zorunda değildir. "*Anahtar İçin İsim*" ismi, kaydettiğimiz bu *int* değere sonradan ulaşmak için gereklidir.

PlayerPrefs.GetInt("Anahtar İçin İsim" : String): Daha önce kaydedilmiş bir *int* değere ulaşmak için kullanılır. Eğer girilen "*Anahtar İçin İsim*" isminde bir anahtar yoksa varsayılan olarak 0 döndürülür.

PlayerPrefs.SetFloat("Anahtar İçin İsim" : String, deger : float): Bir *float* değişkenin değerini depolamaya yarar.

PlayerPrefs.GetFloat("Anahtar İçin İsim" : String): Daha önce kaydedilmiş bir *float* değere ulaşmak için kullanılır. Eğer girilen "*Anahtar İçin İsim*" isminde bir anahtar yoksa varsayılan olarak 0.0F döndürülür.

PlayerPrefs.SetString("Anahtar İçin İsim" : String, "Değer" : String): Bir *String* değişkenin değerini depolamaya yarar.

PlayerPrefs.GetString("Anahtar İçin İsim" : String): Daha önce kaydedilmiş bir *String* değere ulaşmak için kullanılır. Eğer girilen "*Anahtar İçin İsim*" isminde bir anahtar yoksa varsayılan olarak "" döndürülür.

PlayerPrefs.HasKey("Anahtar İsmi" : String): Girilen "*Anahtar İsmi*"ne sahip bir anahtarın olup olmadığını, *boolean* cinsinden döndürür.

PlayerPrefs.DeleteKey("Anahtar İsmi" : String): Girilen "*Anahtar İsmi*"ne sahip anahtarı ve değerini silmeye yarar.

PlayerPrefs.DeleteAll(): Şimdiye kadar oluşturulmuş tüm anahtarları ve değerlerini silmeye yarar.

- **DEBUG (KONSOL) KOMUTLARI**

Debug.Log("Yazı" : String): Debug konsoluna ilgili String yazıyı yazdırır. print("Yazı") komutu da aynı işi yapar.

Debug.DrawRay(baslangicKonum : Vector3, dogrultu : Vector3, renk : Color): Sadece oyunu tasarlarken "Scene" penceresinde gözüken, hayali bir çizgi çizer. Önce çizginin başlangıç noktası, sonra yönü ve uzunluğu, sonra da rengi belirlenir.

Debug.Break(): Oyunu test ederken pause etmeye yarar. Oyuna devam etmek için elle 'Play' tuşuna basılmalı.

- **GUI (ARAYÜZ) KOMUTLARI**

NOT1: Burada yazılan scriptlerin hepsi '*OnGUI()*' fonksiyonunda işe yarar.
NOT2: Bazı *GUILayout* komutlarının direkt *GUI* şeklinde olan versiyonlarını yazmadım çünkü genel mantık tıpkı *GUILayout.Button* ve *GUI.Button* komutlarında olduğu gibi işliyor. Bu yüzden heralde ilgili *GUILayout* komutlarını kendiniz de *Button* örneğine bakarak *GUI* komutuna çevirebilirsiniz.

GUI.skin : GUISkin : Eğer Proje panelinde "*Create – GUI Skin*" yoluyla veya dışarıdan import ederek kişiselleştirilmiş bir GUI Skin oluşturmuşsanız ve bunu '*GUISkin*' tipinde bir değişkene aktarmışsanız o kişiselleştirilmiş arayüzü ilgili '*OnGUI()*' fonksiyonunda kullanmanıza yarar:

```
var arayuz : GUISkin;

function OnGUI()
{
	GUI.skin = arayuz;
}
```

GUI.color : Color : Arayüz elemanlarının temel rengini belirler (Butonlardaki yazı renkleri, arkaplan renkleri vb.). Değeri '*Color.white*', '*Color.red*' gibi tanımlı bir renge ayarlanabilir.

GUI.enabled : boolean : Eğer değeri *false* yapılırsa bu koddan sonraki tüm GUI elemanları, bunun (*GUI.enabled*) değeri tekrar *true* yapılana kadar tıklanamaz hâle gelir (Eğer tıklanabilir bir GUI elementiyse) ve yarı saydamlaşır (Eğer GUI elementini çevreleyen bir çerçeve varmışsa.). Örneğin:

```
function OnGUI()
{
	GUILayout.BeginVertical(); /*GUI elemanlarını dikey olarak yerleştiriyorum.*/
	GUILayout.Button("Merhaba");
	GUI.enabled = false; /*Bu satırdan sonraki GUI elemanları aktif olmayacak, ta ki GUI.enabled tekrar true yapılana kadar.*/
	GUILayout.Button("Buna tıklayamazsın!!"); /*Bu butona tıklanamaz, ayrıca buton aktif olmadığından dolayı yarı saydamdır.*/
	GUI.enabled = true; /*Artık bundan sonraki GUI elemanları ise aktif olacak.*/
	GUILayout.Button("Ama buna tıklayabilirsin.");
	GUILayout.EndVertical();
}
```

GUILayout.BeginArea(Rect(baslangicXDeğeri : float, baslangicYDegeri : float, genislik : float, yukseklik : float)): Normal şartlarda GUI elemanları sen bu scripti yazmazsan (0,0) koordinatlarında oluşmaya başlar, yani ekranın en sol üstünde. Ancak bu kötü görünüme engel olmak için, mesela GUI elemanlarının (Butonlar, kaydırılabilir çubuklar vb.) ekranın orta kısmı ile sol kısmının tam ortasından başlamasını istiyorsan bu script ile bu GUI elemanlarını bir dikdörtgen alan içinde oluşturulmaya zorlarsın. İlk iki parametre ile

dikdörtgenin ekrandaki başlangıç X ve Y değerlerini belirlerken sonraki parametrelerle dikdörtgenin genişliğini ve yüksekliğini belirlersin. Örnek bir kullanımı:

```
public var genislik : float = 400;
public var yukseklik : float = 300;

function OnGUI()
{
    var baslangicX = ((Screen.width * 0.5) - (genislik * 0.5));
    var baslangicY = ((Screen.height * 0.5) - (yukseklik * 0.5));
    GUILayout.BeginArea(Rect(baslangicX,baslangicY, genislik, yukseklik));
    ...
    ...
    ...
    GUILayout.EndArea();
}
```

Burada yapılan işlemle GUI elemanları, "*genislik*" ve "*yukseklik*"i girilen bir dikdörtgenin içine çizilir ve bu dikdörtgen de ekranın çözünürlüğü, oranı vb. ne olursa olsun daima ekranın tam ortasına ortalanır.

GUILayout.EndArea(): *GUILayout.BeginArea* ile açılmış bir *GUILayout* alanını kapatmaya yarar. Böylece kullanılacak yeni *GUILayout* komutları artık bu *Layout*'un içerisine yerleştirilmez.

GUI.BeginGroup(Rect(x : float, y : float, genislik : float, yukseklik : float)): Bu komut *GUI.EndGroup()* ile kapatılmalıdır. *X* ve *Y* değerleri grubun ekrandaki başlangıç konumunu ifade ederken *genislik* ve *yukseklik* değerleri de grubun kapsadığı genişlik ve yükseklik değerlerini ifade eder. Bu komut ile *GUI.EndGroup()* arasında yer alan tüm GUI elemanları (GUILayout elemanları değil! GUI ve GUILayout farklı işleyen 2 ayrı sistem.) için artık (0,0) olan başlangıç koordinatı oyun ekranının en sol üstünü temsil etmez de *GUI.BeginGroup* ile açılmış grupta yer alan (X,Y) değerini temsil eder. Yani mesela *GUI.BeginGroup(Rect(100, 200, 500, 500))* ile bir grup açtıktan sonra *GUI.Box(Rect(0,0,100,100), "Merhaba")* komutunu kullanırsak, üzerinde "*Merhaba*" yazan ilgili kutucuk ekranın (0,0) koordinatlarında değil de (100,200) koordinatlarında çizilir.

NOT: *GUI.BeginGroup*'taki yükseklik, genişlik gibi değerlerle belirlenmiş alanın dışına taşan GUI elemanlarının taşan kısımları çizilmez, ortaya garip bir görüntü çıkar. O yüzden GUI elemanlarını alanın dışına taşırmamaya özen gösterin.

GUILayout.BeginHorizontal(): GUILayout fonksiyonlarıyla oluşturulacak arayüz elemanlarının birbirinin sağına doğru dizilmesini sağlar. İşi bitince GUILayout.EndHorizontal() komutuyla kapatılmalıdır.

GUILayout.BeginVertical(): GUILayout fonksiyonlarıyla oluşturulacak arayüz elemanlarının birbirinin altına doğru dizilmesini sağlar. İşi bitince GUILayout.EndVertical() komutuyla kapatılmalıdır.

GUILayout.Width(genislik : float): İçine yazılan arayüz elemanının genişliğini elle belirlemeye yarar.

GUILayout.Height(yukseklik : float): İçine yazılan arayüz elemanının yüksekliğini elle belirlemeye yarar:

```
function OnGUI()
{
	GUILayout.Box("Merhaba", GUILayout.Width(150), GUILayout.Height(70));
}
```

GUILayout.Space(boslukMiktari : float): İki GUI elemanı arasında *boslukMiktari* kadar pixellik boşluk bırakır.

GUILayout.Button("Butonun Adı" : String): Üzerinde "*Butonun Adı*" yazan tıklanabilir bir GUI butonu oluşturur. Butonun genişliği, oluşturulacağı yer vb. '*GUILayout.BeginArea()*'ya bağlı olarak değişebilir, eğer yoksa (0,0) koordinatlarında oluşturulur. Örnek:

```
function OnGUI()
{
	if(GUILayout.Button("Bana tikla!"))
	{
		Debug.Log("Butona tikladin :)");
	}
}
```

GUILayout.RepeatButton("Butonun Adı" : String): Bu fonksiyonun *GUILayout.Button*'dan tek farkı; *GUILayout.Button*'da butona her bastığımızda olaylar sadece 1 kez gerçekleşirken bu fonksiyonda butona basılı tuttuğumuz sürece olay gerçekleşir.

GUILayout.Box("Kutunun adı" : String): Üzerinde "*Kutunun adı*" yazan bir GUI kutusu oluşturur. Kutuların butonlardan farkı; kutulara tıklanamaz ancak butonlara tıklanabilir. Ayrıca kutular görsel olarak da butonlardan biraz farklıdır. GUI.Box fonksiyonu da ayrıca mevcuttur.

GUILayout.Label("Yazı" : String): Ekrana "*Yazı*" yazdırır. Bu GUI elemanının buton ya da kutu gibi bir görseli yoktur, tıpkı *GUIText* gibi sadece ekrana girdiğin yazıyı yazar. Ya da *GUILayout.Label(gorsel)* ile *Texture2D* tipinde bir görseli (*gorsel*) ekrana çizdirebilirsin. Bu görsel çizdirme işlemi diğer GUI elemanları için de aynen geçerlidir. GUI.Label fonksiyonu da ayrıca mevcuttur.

GUILayout.TextField(duzenlenecekString : String): Bir String türünden değişkenin değerini arayüz elemanı vasıtasıyla, girdi olarak düzenlememizi/girmemizi sağlar. Çok kullanışlı özellik, mesela online bir oyunda kullanıcı adı girmek için kullanılabilir. Dilenirse 2. bir argüman vasıtasıyla girilen yazının olabilecek maksimum uzunluğu belirlenebilir. Örnek bir kullanımı için *GUILayout.PasswordField* fonksiyonunun tanıtıldığı kısma bakabilirsin. GUI.TextField fonksiyonu da ayrıca mevcuttur.

GUILayout.TextArea(duzenlenecekString : String): Bu fonksiyonun *GUILayout.TextField*'dan farkı; *GUILayout.TextField'*da string sadece tek bir satırdan

oluşabilirken *TextArea*'da ise Enter tuşuyla yeni bir satıra geçebiliyoruz. GUI.TextArea fonksiyonu da ayrıca mevcuttur.

GUI.DrawTexture(Rect(x : float, y : float, genislik : float, yukseklik : float), textureDosyasi : Texture): Girilen X ve Y koordinatlarında, girilen genişlik ve yükseklik değerlerine sahip bir dikdörtgen alanın içerisine bir textureDosyası (Resim dosyası) çizer.

GUILayout.PasswordField(duzenlenecekParola : String, "*"[0] : char): Bir String türünde olan duzenlenecekParola'yı arayüz elemanı vasıtasıyla, klavyeden girdi olarak düzenlememizi/girmemizi sağlar. Parola * şeklinde görünür. Bunu değiştirmek için komuttaki * işaretini başka bir işaretle değiştirmek yeterlidir. Oradaki [0] da ne diyebilirsiniz ki ben de demiştim zaten. Onun görevi şu: Unity PasswordField fonksiyonunda parola yerine gösterilecek işaretin (*) bir char olmasını istiyor, bir String değil.! Buradaki [0] ise "*" stringindeki ilk elemanı (Çünkü dizilerin elemanları 0'dan başlar ve 1 2 … diye devam eder.) char olarak döndürmeye yarar. Örneğin "Merhaba"[2] deseydim bu sefer yazıdaki 2+1=3'üncü harfi, yani 'r' harfini char olarak döndürecek ve şifreyi bu harfle gizleyecekti (Çok enteresan duracağına eminim :)). Örnek:

```
function OnGUI()
{
	GUILayout.BeginVertical();
	GUILayout.Label("Nickinizi girin: ");
	nick=GUILayout.TextField(nick, 25, GUILayout.Width(120)); /* Maksimum 25 karakter uzunluğunda*/
	GUILayout.Label("Şifrenizi girin: ");
	sifre=GUILayout.PasswordField(sifre, "*"[0], 15, GUILayout.Width(120)); /*Maksimum 15 karakter uzunluğunda*/
	GUILayout.EndVertical();
}
```

GUI.Button(Rect(0, 0, 200, 100), "Butonun Adı" : String): Ekranın en sol üstünde (0,0 koordinatlarında) 200 pixel genişliğinde ve 100 pixel yüksekliğinde bir buton çizer. Bu komutta *GUILayout.Button*'dan farklı olarak butonun konumunu ve boyutlarını tamamen kendimiz belirliyoruz.

GUI.RepeatButton(Rect(0, 0, 200, 100), "Butonun Adı" : String): Bu fonksiyonun *GUI.Button*'dan tek farkı; *GUI.Button*'da butona her bastığımızda olaylar sadece 1 kez gerçekleşirken bu fonksiyonda butona basılı tuttuğumuz sürece olay gerçekleşir.

GUILayout.FlexibleSpace(): Normal şartlarda *GUILayout* elemanları belirli bir alanın içerisine otomatik olarak yerleştirilirler, yani her bir *GUILayout* elemanının ekrandaki konumunu tasarımcı tek tek girmez (Zaten üstteki *GUILayout* fonksiyonlarında da hiç konum bilgisi bulunmamakta.). *GUILayout.FlexibleSpace()* komutu ise, geçerli *GUILayout* için öngörülmüş, belirli bir yerdeki tüm alanı (*GUILayout.BeginArea()* fonksiyonuyla belirlenmiş bölgeyi. Zaten eğer *GUILayout.BeginArea()* fonksiyonunu kullanmamışsanız *GUILayout.FlexibleSpace()* komutu işe yaramaz.) doldurmaya yarar.

Normalde GUI elemanları oluşturduğumuz bir GUI alanının (*GUILayout.BeginArea()* ile oluşturuyoruz.) en sol üstünden itibaren dizilmeye başlar. Ama biz butonların ilgili alanın sol üstüne değil de en sağ altına dayalı olmasını istiyoruz diyelim:

```
function OnGUI()
{
	GUILayout.BeginArea(Rect(0, 0, Screen.width, Screen.height)); /*GUI elemanlarının çizileceği alanı tüm oyun ekranı olarak belirliyoruz.*/
	GUILayout.BeginHorizontal(); /*Önce GUI elemanlarını sağa dayamak için, yatay dizilime başlıyoruz.*/
	GUILayout.FlexibleSpace(); /*Üstteki komutla yatay dizilime başladığımızdan bu komutla ekranın solunda mümkün olduğunca boşluk kalmasını ayarlıyoruz.*/
	GUILayout.BeginVertical(); /*Şimdi de GUI elemanlarını aşağıya dayamak için dikey dizilime başlıyoruz.*/
	GUILayout.FlexibleSpace(); /*Ekranın üstünde mümkün olduğunca boşluk kalması, yani GUI elemanlarının aşağıya dayatılması için yukarıda bir esnek boşluk oluşturuyoruz. (Yani bu boşluğun ne kadar olacağı belli değildir. Biz yeni GUI elemanları ekledikçe tüm GUI elemanlarını en aşağıya dayamak için boşluğun miktarı Unity tarafından sürekli hesaplanır.*/
	GUILayout.Button("İlk buton"); /*Örnek olması için birkaç GUI elemanı oluşturuyorum.*/
	GUILayout.Box("Buraya da bir kutucuk koyalım :)");
	GUILayout.Button("Son buton");
	GUILayout.EndVertical(); /*Artık tüm GUI elemanlarımı yerleştirdim. O yüzden bir sorun çıkmaması için tüm yerleştirme komutlarını sırayla kapatıyorum.*/
	GUILayout.EndHorizontal();
	GUILayout.EndArea(); /*En son GUI alanını kapatmayı unutma!*/
}
```

Bu örnekte de olduğu gibi *FlexibleSpace()* komutunun yeri önemlidir. Biz her yeni dizilim metodunun başında bu komutu kullandık. Yani Unity'e dedik ki "Unity! Sen önce bir bu GUI elemanlarının ne kadar yer kaplayacağını hesapla. Sonra bu alanın dışında kalan boşluğu en sona değil de en başa ekle!". Örneğin *BeginHorizontal()*'ın altındaki *FlexibleSpace()* komutunu oradan alıp *EndHorizontal()*'ın üstüne koyarsan göreceksin ki bu sefer butonlar ekranın en sağ altına değil de en sol altına dizildi, çünkü boşluğu yatayda GUI elemanlarının soluna değil de sağına yerleştirdik. Veya hem *BeginHorizontal()*'ın altına hem de *EndHorizontal()*'ın üstüne birer *FlexibleSpace()* yazsaydık bu sefer butonlar orta-aşağıya dayanırdı çünkü hem soldan hem de sağdan esnek boşluk bırakmış olurduk. Bu konunun hâlâ anlaşılmaması normaldir. Biraz karışık duran ama aslında çok basit olan bir şey çünkü. Ben yine de bir örnek daha vereyim:

```
var yukseklik : int = 400;
function OnGUI() {
	GUILayout.BeginArea(Rect(0, 0, 200, yukseklik));
	GUILayout.BeginVertical();

	GUILayout.Box("Üstteki kutu", GUILayout.Height(100));
	GUILayout.FlexibleSpace();
	GUILayout.Box("Alttaki kutu", GUILayout.Height(100));
```

```
        GUILayout.EndVertical();
        GUILayout.EndArea();
}
```

Bu kodu yazınca göreceksin ki "*Üstteki kutu*" en üstte ve "*Alttaki kutu*" da onun 200 pixel altında, çünkü *BeginArea* ile oluşturduğumuz alanın yüksekliği zaten 400. Şimdi burada *FlexibleSpace()* komutu aslında aynen şuna eşit: *GUILayout.Space(200)*. Neden? Çünkü ilk kutu en üstte 100 pixel bir yükseklik kaplıyor ve arada *FlexibleSpace* var ve aşağıdaki kutu da aşağıda 100 pixel yer kaplıyor. E zaten bizim oluşturduğumuz alan 400 pixel yüksekliğinde olduğundan *FlexibleSpace* otomatik olarak 200 pixel yer kaplıyor. Peki neden biz direkt *GUILayout.Space(200)* demedik? Aslında bunun güzel 2 nedeni var: Birincisi, biz *FlexibleSpace* deyince aradaki boşluğu matematiksel olarak hesaplama zahmetinden kurtuluyoruz, bunu Unity kendisi otomatik olarak yapıyor (Hiç şimdiye kadar *FlexibleSpace()*'in içerisinde herhangi bir değer gördün mü!), ikinci faydası ise şu ki biz '*yukseklik*' değişkeninin değerini değiştirdiğimizde *FlexibleSpace()*'in kaplayacağı alan da yine otomatik olarak yeniden hesaplanıp ona göre değişiyor. Yani gidip de *GUILayout.Space()*'in miktarını da kendimiz yeniden hesaplamak zorunda kalmıyoruz. Ve hatta işte tam bu yüzden bunun adı Türkçe adı "esnek bölge". Çünkü burada bahsettiğim gibi durumdan duruma göre esneyip gerilebiliyor.

GUILayout.BeginScrollView(): Bunun pek çok zaman gördüğümüz en yaygın örneği, bir oyunu veya programı yüklerken karşılaştığımız çok uzun olan ve bizim direkt "Kabul ediyorum (I accept/agree …)" diyerek geçiştirdiğimiz Kullanıcı Sözleşmeleri'dir. Tıpkı onun gibi, içinde bir yazı ve yazının sağında da onu aşağı yukarı kaydırmaya yarıyan bir scrollbar içeren bir sistemdir bu. Kullanımı biraz farklı olduğu için örnekle izah edeyim:

```
private var scrollbarKonumu : Vector2;
function OnGUI() {
        scrollbarKonumu = GUILayout.BeginScrollView(scrollbarKonumu,
GUILayout.Width(200), GUILayout.Height(50));
        /*Buradan itibaren yazdıklarımız bu 'Kullanıcı Sözleşmesi' gibi olan şeyin içine
yazılıyor; ta ki onu GUILayout.EndScrollView() ile kapatana kadar. Bu alanın içine istersek
text girebileceğimiz gibi istersek buton ya da resim bile girebiliriz.*/
        GUILayout.Label("Bu bir lisans sözleşmesidir …\n\n **PEK ÇOK SATIR SONRA**
\n\nBöylece sözleşmenin sonuna geldik :)"); /*Label ile çerçevesiz sade bir yazı yazıyoruz
sözleşmeye.*/
        GUILayout.EndScrollView();
}
```

Burada da gördüğün gibi öncelikle dışarıda bir Vector2 değişkeni oluşturacaksın (*private* olması zorunlu değil.) ve onu sürekli *BeginScrollView*'e eşitleyeceksin. *BeginScrollView*'in içindeyse ilk argüman olarak yine *Vector2* değişkenini kullanacak, ardından 'Kullanıcı Sözleşmesi' alanının boyutlarını gireceğiz. ***Burada sürekli 'Kullanıcı Sözleşmesi' diyorum ama bu alan sadece sözleşme için kullanmak için değil, istediğin her şeyde kullanılabilir. Benim böyle dememin sebebi ise neyden bahsettiğimin daha rahat anlaşılmasını istemem.*** Burada şöyle bir durum var: Eğer *ScrollView*'in içine onun alabileceğinden fazla bir şey girmezsek, yani tüm yazı zaten scrollbar'a gerek duymadan tek bir seferde görüntülenebiliyorsa, normal şartlarda Unity de orada bir scrollbar oluşturmaz. Ancak yazı tek seferde görüntülenemeyecek / sığamayacak kadar büyük olursa o zaman bir scrollbar

oluşturur. Ancak dilersek Unity'i her zaman scrollbar göstermeye zorlayabiliriz. Bunun için yapmamız gereken değişiklik şu (Üstteki koddaki örneğe göre gösteriyorum):

```
scrollbarKonumu = GUILayout.BeginScrollView(scrollBarKonumu, false, true,
GUILayout.Width(200), GUILayout.Height(50));
```

Bir başka kullanışlı örnek daha vereyim. Mesela bir kullanıcı sözleşmesi olsun ve kullanıcı onun sağındaki scrollbarı aşağı kaydırmadığı sürece oyunu başlatan buton inaktif olsun. Böylece kullanıcıyı okumasa bile sözleşme ile temasa geçirmiş olacağız:

```
private var scrollBarKonumu : Vector2;
function OnGUI() {
	GUILayout.BeginVertical();
	scrollBarKonumu = GUILayout.BeginScrollView(scrollBarKonumu, false, true,
GUILayout.Width(200), GUILayout.Height(50));
	GUILayout.Label("Dikkat! Bu sözleşmeyi imzalarsanız eviniz, tapunuz, mobilyanız
karşılıksız bizim olacaktır!! O yüzden lütfen sözleşmeyi okumayın ki biz de sizden kolayca
faydalanabilelim. Şaka yapıyoruz, evinize vb. ihtiyacımız yok. Siz sadece sözleşmeyi okuyun
işte…");
	GUILayout.EndScrollView();
	GUILayout.Space(20);
	if(scrollBarKonumu.y < 80)
	{
		GUI.enabled = false;
	}
	if(GUILayout.Button("Okudum, kabulüm!"))
	{
		Application.LoadLevel("Oyun");
	}
	GUI.enabled = true;
	GUILayout.EndVertical();
}
```

Eğer bu kodu yazarsan bahsettiğim gibi bir sistem kurmuş olacaksın. Burada bir şey oldukça önemli; o da *scrollBarKonumu.y < 80*'deki 80 değerinin (*scrollBarKonumu.y* sağdaki scrollbarın yazıyı aşağı doğru kaç pixel indirdiğini gösterir. Scrollbar aşağı kaydıkça bu değer artar.) göz kararı verilmiş olması. Çünkü yazıdan yazıya ve *ScrollView* alanının boyutuna göre scrollbarın inebileceği maksimum miktar değişir ve maksimum miktarı bize veren bir fonksiyon da yok. O yüzden bunu deneme-yanılma yoluyla kendin bulmalısın.

GUIUtility.RotateAroundPivot(dereceMiktari : float, referansNokta : Vector2): Bu komuttan sonra yazılan GUI elemanları, Referans Nokta'ya göre 'Derece' miktarında döndürülür. Eğer bir GUI elemanını tam olduğu yerde döndürmek istiyorsan Referans Nokta olarak o GUI elemanının tam merkezindeki noktasının koordinatlarını *Vector2* olarak vermelisin.

- **MATEMATİKSEL KOMUTLAR**

Random.Range(minimumDeger : float, maksimumDeger : float): İçine girilen minimum ve maksimum değerlerin arasında rasgele bir değer elde eder. Pek çok yerde kullanılabilir. Ancak önemli husus, eğer girdiğiniz değerlerin biri '*float*' ise diğeri de '*float*' olmalı, '*int*' ise diğeri de '*int*' olmalı ve minimum asla maksimumdan büyük olmamalı. Ayrıca eğer girilen değerler '*int*' ise döndürülecek değer minimum değere eşit olabilirken maksimum değere eşit olmaz. Yani minimum değer bu rasgele işlemine dahilken maksimum değer değildir. Ama girilen değerler '*float*' ise döndürülen değer maksimum değere de eşit olabilir.

Random.value : float: "*Random.Range(0.0f, 1.0f);*" komutu ile tamamen aynıdır, 0.0 ile 1.0 arasında (Bu 2 değer de dahil olmak üzere) bir sayı döndürür.

Vector3.Distance(birinciKonum : Vector3, ikinciKonum : Vector3): İki nokta arasındaki uzaklığın hesaplanmasına yarar. Konumların herhangi biri için bir değişken kullanılacaksa değişkenin tipi '*Transform*' olmak zorundadır. Örnek bir kullanım:

```
var obje : Transform;
function Update(){
	var mesafe : float = Vector3.Distance(obje.position, transform.position);
	if(mesafe <= 50)
	{
		Destroy(gameObject); /*Scriptin uygulandığı obje yok edilir.*/
	}
}
```

Vector3.Normalize(): Girilen *Vector3* türünden vektörün *x,y* ve *z* değerlerini, vektörün uzunluğu 1 olacak şekilde düzenler. Örneğin:

```
var vektor : Vector3 = Vector3( 1, 2, 2 ); // Uzunluğu 3 birim olan bir vektör
vektor.Normalize(); /* Artık vektor'un değeri ( 1/3, 2/3, 2/3 ) oldu. Böylece uzunluğu da 1 birim oldu. */
```

Vector3.normalized : Vector3: *Vector3.Normalize()*'den tek farkı, bu fonksiyon girilen vektörün değerini değiştirmez, onun yerine yeni bir vektör döndürür. Örneğin:

```
var vektor1 = Vector3( 1, 2, 2 );
var vektor2 : Vector3;
vektor2 = vector1.normalized; /* Artık vektor2'nin değeri ( 1/3, 2/3, 2/3 ) olurken vektor1'in değeri (1, 2, 2 ) olarak kaldı. */
```

Mathf.Clamp(floatBirSayi : float, minimumDeger : float, maksimumDeger : float): Bir değişkenin değerinin, bir minimum ve bir maksimum olmak üzere 2 değerin arasında olmasını garantiler. Örneğin:

```
function Update() {
	transform.Translate(Vector3(Mathf.Clamp(transform.Translate,-15,15),0,0));
```

```
}
```

Bu kodla yapılan, eğer kodun uygulandığı objenin x koordinatındaki konumu -15'den küçük olursa otomatik olarak -15, 15'den büyük olursa otomatik olarak 15 yapmaktır.

Mathf.Clamp01(floatBirSayi : float): Bu komut *Mathf.Clamp(Float bir sayı, 0, 1)* ile tamamen aynı şeyi ifade eder. İçine girilen sayının 0 ile 1'in dışına çıkmamasını sağlar.

Mathf.Lerp(birinciDeger : float, ikinciDeger : float, sayi : float): Öncelikle içine girilen "*sayı*" parametresinin değeri eğer 0'dan küçükse Unity onu otomatik olarak 0 yaparken 1'den büyükse de otomatik olarak 1 yapar. Ardından başta karmaşık gibi duran ama çok basit olan bu scriptte mantık şöyle işler:

Scriptin içine girilen "*sayı*"nın değeri ne kadar 0'a yaklaşırsa scriptin döndüreceği değer o kadar "*Birinci Değer*"e yaklaşır, "*sayı*" 0 olursa script "*Birinci Değer*"in kendisini döndürür. "*sayı*"nın değeri 1'e ne kadar yaklaşırsa scriptin döndüreceği değer de "*İkinci Değer*"e o kadar yaklaşır, "*sayı*" 1 olursa script "*İkinci Değer*"i döndürür. Eğer "*sayı*" 0.5 olursa yaptığım açıklamalardan da çıkarılabileceği üzere "*Birinci Değer*" ile "*İkinci Değer*"in aritmetik ortalamasını döndürür. Mesela

```
Mathf.Lerp(10, 50, 0.7);
```

Bu scriptte "*sayı*" 0.7, yani döndürülecek değer 50'ye daha yakın olacak ve iki sayının aritmetik ortalamasından da (30) büyük olacak. Peki burada döndürülecek sayı nasıl bulunuyor? Şöyle ki, biz "*sayı*"yı 0.7 girince demiş oluyoruz ki "*Unity! Bana döndüreceğin sayı 10 ile 50 arasında olsun ve bu döndüreceğin değer 10 sayısına 7 birim uzaktaysa 50 sayısına 3 birim uzakta olsun.*" ve Unity de bundan yola çıkarak şu işlemi yapıyor:

1. *(50-10) * 0.7 = 28*
2. *10 + 28 = 38*

Ve böylece sonuç olarak **38** döndürüyor.

Mathf.InverseLerp(birinciDeger : float, ikinciDeger : float, sayi : float): *Mathf.Lerp*'in tam tersini yapar. Yani mesela biz *Mathf.Lerp(10, 50, 0.7)* deyince nasıl 38 bulmuşsak; *Mathf.InverseLerp(10, 50, 38)* yazarsak da bu sefer 0.7 buluruz. Yani *sayı*'nın *Birinci Değer* ile *İkinci Değer* arasındaki oransal olarak konumunu buluruz. Buradan da anlaşılacağı üzere döndürülen değer bir float'tır ve değeri 0 ile 1 arasındadır.

Mathf.LerpAngle(birinciDeger : float, ikinciDeger : float, sayi : float): *Mathf.Lerp* ile aynı görevi yapan bir fonksiyondur. Ondan tek farkı; bu fonksiyon özellikle derece cinsinden açılarla işlem yapmak için olduğu için, eğer girilen değerlerden herhangi biri 360'dan büyükse, o değer 360'dan küçük olana kadar ondan otomatik olarak 360 çıkarılır. Ancak negatif bir değere 360 eklenmez (Aslında negatif değerlerle mümkünse pek uğraşmayın çünkü onlarla oluşan sonuçlar karmaşık olabiliyor.). Bu fonksiyonun yaygın bir kullanımı:

```
function Update() {
        transform.eulerAngles = Vector3(0, Mathf.LerpAngle(Birinci Açı, İkinci Açı,
Time.time), 0);
}
```

Bu fonksiyon sayesinde oyun başladıktan sonra 1 saniye içerisinde obje *y* ekseni etrafında *Birinci Açı*'dan *İkinci Açı*'ya doğru rotasyon yapar. Bunu mesela 2 saniyede yapmak için *Time.time*0.5* yazabiliriz. Burada mesela *Mathf.LerpAngle(20,500, Time.time)* yazarsak *İkinci Açı* otomatik olarak 500-360=140'a ayarlanır ve obje 1 saniyede *y* ekseninde 20 dereceden 140 dereceye döner. Ancak aynı şeyi *Mathf.Lerp(20,500, Time.time)* diye yazsaydık bu sefer 500'den 360 çıkarılmayacaktı ve obje önce 20 dereceden 360 dereceye dönecek, sonra Unity açıyı otomatik olarak 0'a çevirecek (Bunu Unity bir fonksiyon kullanmadan kendisi sürekli yapar.) ve sonra obje 140 dereceye kadar dönüp duracaktı. Yani 1 saniyede 120 derece değil de 480 derece dönecekti. İşte *Lerp* ile *LerpAngle* arasındaki fark bu.

Mathf.PingPong(birinciDeger : float, ikinciDeger : float): İçine girilen *birinci değer*in 0 ile *İkinci değer* arasında kalmasını sağlar. Ancak *Mathf.Clamp* veya Mathf.Lerp fonksiyonlarından bariz farkı vardır. O da şudur ki; *Birinci değer*, *İkinci değer*'den büyük olmaya başladığında bu sefer döndürülen sonuç *İkinci değer*'den 0'a doğru azalmaya başlar ve değer 0'a ulaşınca tekrar *İkinci değer*'e doğru yükselmeye başlar. İşte bu yüzden adı **PingPong**'tur. Yine bu yüzden *İkinci değer* 0'dan küçük olamaz. Daha iyi anlaman için birkaç örnek vereyim:

Mathf.PingPong(1,3) bize direkt 1 döndürür çünkü 1 sayısı zaten 0 ile 3 arasında.
Mathf.PingPong(4,3) bize 2 döndürür. Neden? Çünkü tıpkı PingPong gibi, değer 3'ten büyükse, 3'ten 0'a doğru geri gelmeye başlıyordu. Peki 4 sayısı 3'ten kaç fazla: 1. O zaman döndürülecek değer de 3-1=2 olur.
Mathf.PingPong(7,3) bize 1 döndürür. Tekrar neden? İşlemi adım adım ilerletelim: Önce sayı 3'e gelene kadar 3 oldu. Sonra sayı geriye doğru bir 3 daha gitti ve 0'a geldi; bu arada değeri 6 oldu. Şimdi sayı tekrar 3'e doğru hareket edecek ama kaç birim: 7-6=1 birim. Sonuç da 0+1=1 olarak döndürülür.

Bu olayı canlı olarak daha rahat görmen için dinamik bir örnek:

```
function Update() {
        print(Mathf.PingPong(Time.time, 3)); /*Console ekranına, 0 ile 3 arasında gidip gelen değerler döndürür.*/
}
```

Mathf.Repeat(birinciDeger : float, ikinciDeger : float): Aslında *Mathf.PingPong*'un neredeyse aynısı, sadece bir fark var aralarında. O da şu: *PingPong* fonksiyonunda *Birinci değer İkinci değeri* aşınca *İkinci değer*'den 0'a geri yönelme varken *Repeat* fonksiyonunda geri yönelme diye bir şey yok. *Birinci değer İkinci değer*'den büyük olunca direkt 0'a geri döner ve yine *İkinci değer*'e doğru yükselmeye başlar. Yani mesela *Mathf.PingPong(4,3)=2* iken *Mathf.Repeat(4,3)* ise 1'e eşittir. Aslında bu fonksiyon '*Birinci değer % İkinci Değer*' ifadesiyle neredeyse tamamen aynı işi yapıyor, yani bu fonksiyonu kullanmana pek gerek de yok. Ama Unity kılavuzunda bir farktan bahsediliyor ki bu farkı ben nedense göremedim :).

Mathf.Abs(birSayi : float): İçine girilen sayının mutlak değerini (Absolute) döndürür. Yani negatif bir sayı girersen onun pozitifini verir. (Mesela -12 girersen 12, -3.5 girersen 3.5, 6 girersen 6 döndürür.)

Mathf.Sign(birSayi : float): İçine girilen sayının işaretini döndürür. Eğer sayı 0'a eşit ya da ondan büyükse 1 döndürürken sayı 0'dan küçükse de -1 döndürür.

Mathf.ClosestPowerOfTwo(birSayi : int): İçine girilen sayıya en yakın olan 2'nin kuvvetini bulur. 2'nin kuvvetleri sırayla 1-2-4-8-16-32-64-128-256-512-1024-… diye devam eder ve bu komutun içindeki sayı bunların hangisine en yakınsa o sayı döndürülür. (Gereksiz bir komut sanki :))

Mathf.IsPowerOfTwo(birSayi : int): İçine girilen sayının 2'nin kuvveti olup olmadığını *true* ya da *false* olarak döndürür. Bence gereksiz bir şeydir çünkü '*(sayi % 2) == 0*' ifadesi de aynı işi görmektedir.

Mathf.NextPowerOfTwo(birSayi : int): İçine girilen sayıdan büyük olan ve 2'nin kuvveti olan ilk sayıyı verir. Sayı zaten 2'nin kuvveti ise sayının kendisini verir.

Mathf.Approximately(birSayi : float, baskaBirSayi : float): İçine girilen 2 *float* değerin aynı olup olmadığını, *true* ya da *false* olarak döndürür. Bunu özel kılan şey ise şu ki float'larda noktadan sonrası çok net olmadığı için mesela *1.0 == 10.0/10.0* işlemi *true* döndürülmezken (ki aslında bu *true* bir ifadedir.) *Mathf.Approximately(1.0, 10.0/10.0)* yazdığımızda bu sefer *true* döndürülür.

Mathf.PI : float : Pi sayısını (π) ifade eder. Değeri değiştirilemez.

Mathf.Infinity : float : +∞ (Sonsuz) sayısını ifade eder. Değeri değiştirilemez.

Mathf.NegativeInfinity : float : -∞ (Eksi sonsuz) sayısını ifade eder. Değeri değiştirilemez.

Mathf.Deg2Rad : float : Derece cinsinden bir açıyı radyan cinsine çevirir (Yani dereceyi **2π/360** ile çarpar.). Mesela 30 derece π/6 radyana (Unity buradaki π'yi ifade etmek yerine onu kendi PI değeri ile (3.1415…) çarpar. Yani 3.14/6 ≅ 0.5'e yakın bir değer döndürür.) eşittir. Bu bir fonksiyon değildir, bu yüzden bir fonksiyon gibi kullanılamaz. **Bu sadece 2*Mathf.PI/360 demenin bir başka yoludur**. Ve mesela 60 dereceyi radyana çevireceksen **60*Mathf.Deg2Rad** şeklinde kullanırız.

Mathf.Rad2Deg : float : Radyan cinsinden bir açıyı derece cinsine çevirir. **Bu sadece 360/2*Mathf.PI demenin bir başka yoludur**. Kullanımı tıpkı *Deg2Rad*'daki gibidir. (Mesela *0.5*Mathf.Rad2Deg* gibi..)

Mathf.Epsilon : float : 0'a olabilecek en yakın *float* tarzındaki sayıdır (Tabi bu da 'sonsuz' sayı gibi; işlemlerde kolaylık sağlaması için kullanılan, gerçekte var olmayan bir sayıdır. Lise 4 matematiğinde görülen lim $x \to 0^+$ (Sıfıra sağdan gelirken limit) gibi bir şeydir.). Bunu Unity kendisi *Mathf.Approximately* fonksiyonunda kullanır. Bu sayının oyun tasarımcısı tarafından kullanılmasına gerek olduğunu sanmıyorum.

Mathf.DeltaAngle(birSayi : float, baskaBirSayi : float): Derece cinsinden 2 açının arasındaki en küçük açıyı bulur. Mesela açılardan biri 360'tan büyükse de, 360'tan küçük olana kadar ondan otomatik olarak 360 çıkarır (Açıyı bir değişkende saklıyorsan değişkenin değerini değiştirmez, merak etme.). Burada açıların konumu önemlidir. Lisede trigonometride görüldüğü gibi burada da negatif sonuçlar döndürülebilir. Mesela *Mathf.DeltaAngle(10,350)* yazarsan -20 döndürür. Çünkü birim çemberde açılar saat yönünde negatif değer alırken saat

yönünün tersinde pozitif değerler alıyordu. Aynen öyle de, 10 dereceyi ve 350 dereceyi birim çemberde çizersen aralarındaki en küçük açının 10'dan 350'ye saat yönünde -20 derece olduğunu rahatça görebilirsin. Yok ama eğer *Mathf.DeltaAngle(350,10)* deseydik bu sefer +20 döndürürdü. Eğer ki sonucun sürekli pozitif çıkmasını istiyorsan *Mathf.Abs(Mathf.DeltaAngle(Float, Float))* şeklinde yazabilirsin.

Mathf.Cos(birSayi : float): İçine float olarak girilen radyan cinsinden açının cosinüs'ünü verir.

Mathf.Sin(birSayi : float): İçine float olarak girilen radyan cinsinden açının sinüs'ünü verir.

Mathf.Tan(birSayi : float): İçine float olarak girilen radyan cinsinden açının tanjant'ını verir.

Mathf.Acos(birSayi : float): İçine girilen değerin radyan cinsinden arccosinüs'ünü verir.

Mathf.Asin(birSayi : float): İçine girilen değerin radyan cinsinden arcsinüs'ünü verir.

Mathf.Atan(birSayi : float): İçine girilen değerin radyan cinsinden arctanjant'ını verir.

Mathf.Exp(birSayi : float): Doğal logaritmada da kullanılan '**e**' sayısının 'Float'ıncı kuvvetini, yani $\mathbf{e}^{\text{Float değer}}$ sonucunu verir. **e**'nin normal değeri ise yaklaşık 2.718282…'dir.

Mathf.Pow(birinciDeger : float, ikinciDeger : float): birinciDeger'in ikinciDeger'inci kuvvetini döndürür. Yani $\mathbf{birinciDeger}^{\text{ikinciDeger}}$ işlemini yapar.

Mathf.Sqrt(birSayi : float): İçine girilen sayının karekökünü bulur.

Mathf.Log(): İki farklı kullanımı vardır: *Mathf.Log(a : float, b : float)* ile $\mathbf{log_a(b)}$'yi bulursun. İkinci kullanımı ise *Mathf.Log(a : float)* şeklindedir. Bunda ise doğal logaritma kullanılır, yani $\mathbf{log_e(a)}$ ya da bir başka deyişle **ln(a)** bulunur.

Mathf.Log10(birSayi : float): *Mathf.Log(10,* birSayi : float *)* ile tamamen aynı şeydir, onu kısayoludur. 10 tabanında logaritma almaya yarar (En meşhur logaritma).

Mathf.Max(En az 2 sayı (Aralarında virgül konulmalı) : float): Girilen sayılar arasından <u>en büyük</u> olan sayıyı döndürür. Burada dikkat edilmesi gereken husus; sayılardan biri *int* ise diğerleri de *int,* biri *float* ise diğerleri de *float* olmak zorundadır.

Mathf.Min(En az 2 sayı (Aralarında virgül konulmalı) : float): Girilen sayılar arasından <u>en küçük</u> olan sayıyı döndürür. Burada dikkat edilmesi gereken husus; sayılardan biri *int* ise diğerleri de *int,* biri *float* ise diğerleri de *float* olmak zorundadır.

Mathf.Round(birSayi : float): İçine girilen sayıyı en yakın tamsayıya yuvarlar. Özel durum olarak; eğer girilen sayının sonu ".*5*" ile biterse, o sayıya en yakın çift tamsayı döndürülür.

Mathf.RoundToInt(birSayi : float): *Mathf.Round*'dan tek farkı, bu fonksiyonda döndürülen değerin *float* tipinde değil de *int* tipinde olmasıdır. Bu farkın bize etkisi ise; bir değişkene bu fonksiyonun döndürdüğü değeri atayacaksak, hata almamak için değişkenin türüne göre bu fonksiyonu ya da *Mathf.Round* fonksiyonunu kullanırız.

Mathf.Ceil(birSayi : float): İçine girilen sayıyı -eğer zaten bir tamsayıya eşit değilse- yukarı yuvarlamaya yarar. Mesela içine 5.3 girersen 6, 9.0 girersen 9, 2.04 girersen 3, -4.6 girersen -4 döndürür.

Mathf.CeilToInt(birSayi : float): Bunun *Mathf.Ceil*'dan tek farkı; bu komut *int* türünden sonuç döndürürken *Mathf.Ceil* ise *float* cinsinden sonuç döndürür.

Mathf.Floor(birSayi : float): İçine girilen sayıyı –eğer zaten bir tamsayıya eşit değilse- aşağı yuvarlamaya yarar. Mesela 5.3 girersen 5, 9.0 girersen 9, 2.78 girersen 2, -4.6 girersen -5 döndürür.

Mathf.FloorToInt(birSayi : float): Bunun *Mathf.Floor*'dan tek farkı; bu komut *int* türünden sonuç döndürürken *Mathf.Floor* ise *float* cinsinden sonuç döndürür.

- **UYGULAMAYLA İLGİLİ KOMUTLAR**

Application.LoadLevel("Geçilecek Bölümün (Scene) Adı" : String): Başka bir scene'e (bölüm) geçmeye yarar. Ancak bu kodun işe yaraması için geçiş yapılacak bölümün "*File – Build Settings*"deki "*Scenes In Build*" kısmına eklenmesi gereklidir.

Application.Quit(): Oyunu sonlandırmaya yarar.

Application.OpenURL(http://yasirkula.com/ : String): İlgili web sayfasını açmaya yarar. Eğer Web Player'da kullanılırsa oyunun oynandığı sayfayı yönlendirir, normal bir '*.exe*' oyunda kullanılırsa siteyi varsayılan tarayıcıda açar.

Application.loadedLevelName : String : En son yüklenen scene'in adını depolar ki normal şartlarda bu oyuncunun o anda bulunduğu bölümdür. Mevcut bölümün hangisi olduğunu test edip ona göre işlem yapmak için ideal bir scripttir. Değeri sadece okunabilir, değiştirilemez.

Application.platform : RuntimePlatform : Oyunun oynandığı cihazın – platformun adını depolar. Değeri, oynanılan platforma göre genelde "*RuntimePlatform.WindowsPlayer*" "*RuntimePlatform.WindowsWebPlayer*" "*RuntimePlatform.Android*" "*RuntimePlatform.IPhonePlayer*" "*RuntimePlatform.FlashPlayer*" "*RuntimePlatform.OSXPlayer*" "*RuntimePlatform.OSXWebPlayer*" değerlerinden birini alır. (Bunlardan başka birkaç değer daha var.)

Application.CaptureScreenshot("Oluşturulacak resim dosyasının adı.png" : String, boyut : int): Ekranın resmini çekmeye ve onu oyunun olduğu klasöre kaydetmeye yarar. Eğer "*boyut*"un değeri 1 yapılırsa çekilen resmin boyutu oyun ekranının boyutuyla aynı olur. Resmin boyutu oyunun oynandığı ekranın boyutuyla doğru orantılıdır. Eğer "*boyut*"u yükseltirsen resmin çözünürlüğü de artar ve hatta bu esnada grafik kalitesinden hiçbir şey kaybolmaz. Mesela bu değeri 4 yaparsan ekranın çözünürlüğünün 4 katı boyutlarında bir resim kaydedilir ve normal oyuna göre çok daha kaliteli bir görsele sahip olur. Yani resim çekerken oyundaki görsellerden daha kaliteli görseller elde edebilirsin bu sayede. Ancak değer ne kadar büyürse resmin çekilme süresi ve boyutu bir o kadar uzar.

Application.dataPath : String : Oyunun yüklü olduğu dizini döndürür.

Application.persistentDataPath : String : Oyunun save dosyalarını kaydetmek için ideal bir konum döndürür. Windows'ta bu "Belgelerim" olurken mobil cihazlarda telefonun hafıza kartı olur.

- **ANİMASYONLA İLGİLİ KOMUTLAR**

animation.Play("Animasyon Adı" : String): Objede ismi girilen animasyonun oynatılmasını sağlar. Bu animasyonlar önceden tanımlı olmalı. Bir objedeki varolan animasyonları onu seçince Inspector'daki *Animation* componentinin altındaki *Animations* sekmesinden görebilirsin. Objenin animasyonlarını tanımlamak içinse (Unity'de oluşturulmamış, başka bir programda oluşturulmuş animasyonları) onu Asset olarak ekleyince Proje panelinden seçip sağdaki *Animations* kısmının altından "Split Animations" seçeneğini seçeceksin ve altındaki kutucukta animasyonlara isim girip onların oynatıldığı kare aralığını gireceksin.

animation.Stop(): Objede çalışmakta olan tüm animasyonları durdurmaya yarar. Veyahutta "*animation.Stop("Animasyon Adı" : String);*" komutu kullanılarak sadece belli bir animasyonun durdurulması da sağlanabilir.

animation.CrossFade("Animasyon Adı" : String): İsmi girilen animasyonu oynatmaya yarar. "*animation.Play()*"den farklı olarak ismi girilen animasyonu direkt geçiş yapılmaz, mevcut animasyondan o animasyona yumuşak bir geçiş yapılır. Böylece animasyonlar arası keskin hareket değişikliklerinin önüne geçilmiş olur. Çok kullanışlı bir özellik yani.

animation.Blend("Animasyon Adı" : String, hiz : int): İsmi girilen animasyonla mevcut oynanmakta olan animasyonu birleştirir. Örneğin karakterin ileri ve sağa gitmek için 2 ayrı animasyonu olsun. Karakter sağ ileri giderken bu 2 animasyonu kombine etmek için bu komut kullanılabilir. Ancak öncesinde "*animation.SyncLayer()*" kullanımı önemlidir. Eğer "*hiz*" değeri 1 yapılırsa yeni animasyon gerçek hızıyla mevcut animasyona dahil olur. Ama eğer "*hiz*" değeri 0 yapılırsa ismi girilen animasyonun bu kombine animasyonla beraber oynaması durdurulur.

animation.SyncLayer(layerNumarasi : int): Birden çok animasyon "*animation.Blend()*" ile kombine edilecekse bunların eş zamanlı, düzgün bir şekilde beraber oynamatılmasını sağlar. Bunun nasıl başarıldığı pek önemli değil, önemli olan işe yaraması. *Start()* fonksiyonunda kullanımı önerilir. Eğer bir animasyonun layer'ı "*animation["Animasyonun Adı" : String].layer = layer : int;*" komutu ile değiştirilmezse, her animasyonun varsayılan başlangıç layer'ı 0'dır.

animasyonAdi.normalizedTime : float: Bir animasyonun en baştan değil de ortadan ya da başka bir konumdan başlamasını sağlar. Değeri 0 yapılırsa animasyon en baştan, 1 yapılırsa en sondan (Aynı şey), 0.5 yapılırsa ortadan başlar.

- ## DOSYA YÖNETİMİ (FILE MANAGEMENT)

NOT1: Klasörlerle çalışırken Directory.Exists(), dosyalarla çalışırken File.Exists() metodları vasıtasıyla öncelikle üzerinde işlem yaptığınız dosyanın gerçekten var olup olmadığını kontrol edin ve böylece olası hataların önüne geçin.

NOT2: Buradaki metodlarda geçen *konum* parametresinin değeri klasörlerle çalışırken örneğin "*D:\Wordpress Dersler*" iken dosyalarla çalışırkense örneğin "*D:\Wordpress Dersler\Ders.pdf*" şeklinde olmalıdır!

Directory.Exists(konum : String): İçine bir klasöre giden yol (path) girilir ve metod bu klasörün var olup olmadığını döndürür. Eğer belirtilen yolda gerçekten ilgili klasör varsa *true*, yoksa *false* döndürür.

Directory.CreateDirectory(konum : String): Konumu girilen klasörü oluşturmaya yarar.

Directory.Delete(konum : String, herSeySilinsin : boolean): Konumu girilen klasörü silmeye yarar. Eğer *herSeySilinsin true* ise klasörün içeriği de silinirken *false* ise klasörün içeriği silinmez.

Directory.Move(eskiKonum : String, yeniKonum : String): Bir klasörü *eskiKonum*'dan *yeniKonum*'a taşır.

Directory.GetDirectories(konum : String): Konumu girilen klasörün içerisindeki tüm alt klasörlere giden yolları bir String array'inde depolar ve döndürür.

Directory.GetFiles(konum : String): Konumu girilen klasörün içindeki tüm dosyalara giden yolları bir String array'inde depolar ve döndürür.

File.Exists(konum : String): İçine bir dosyaya giden yol (path) girilir ve metod dosyanın var olup olmadığını boolean cinsinden döndürür.

File.WriteAllText(konum : String, icerik : String): Girilen konumda girilen *icerik*'e sahip bir dosya oluşturur.

File.AppendAllText(konum : String, icerik : String): Girilen konumdaki dosyanın mevcut içeriğinin en sonuna *icerik*'i ekler.

File.ReadAllText(konum : String): Konumu girilen dosyayı açar ve içeriğini *String* cinsinden döndürür.

File.Delete(konum : String): Konumu girilen dosyayı siler.

- **DİĞER KOMUTLAR (KATEGORİLENDİRİLMEMİŞ)**

color.a : float : Bir renk değişkeninin '*Alpha*'sını, yani saydamlığını değiştirir. Değeri 0-1 arasında olmalıdır.

color.r – color.g – color.b : float : Bir renk değişkeninin RGB (Kırmızı-Yeşil-Mavi) değerlerini ayrı ayrı değiştirmeye yarar. Değerleri 0-1 arasında olmalıdır.

gameObject.SetActiveRecursively(true : boolean): Scriptin yazılı olduğu GameObject'i ve onun varsa tüm child objelerini aktif eder. Yani "*gameObject.active = true;*" komutunu obje ve onun tüm child'ları için gerçekleştirir.

arrayAdi.RemoveAt(1 : int): Bir dizinin (**array**) 2. elemanını (Çünkü dizilerde numaralandırma 0'dan başlar ve bu da dizinin 1. elemanına denk gelir. Yani numarası 1 olan elemanı da dizinin 2. elemanına denk gelmektedir.) silmeye yarar. Bu elemandan sonra başka elemanlar varsa her birinin numarası, aradaki bu boşluğu doldurmak için, 1 düşer. Örneğin:

```
function Start()
{
	var dizi : Array = ["Süleyman", "Yasir", "Kula"]; /*3 elemanlı diziyi oluşturdum.*/
	print(dizi[0]); /*Konsola "Süleyman" yazdırır.*/
	print(dizi[1]); /*Konsola "Yasir" yazdırır.*/
	print(dizi[2]); /*Konsola "Kula" yazdırır.*/
	dizi.RemoveAt(1); /*Dizinin 1+1=2. elemanı olan "Yasir"i diziden atıyoruz.*/
	print(dizi[0]); /*Konsola "Süleyman" yazdırır.*/
	print(dizi[1]); /*Konsola "Kula" yazdırır. Çünkü "Yasir" diziden atıldı ve "Yasir"den sonraki elemanların numaraları birer azaldı.*/
	print(dizi[2]); /*Hata verir çünkü artık diziden "Yasir" gittiği için dizi 2 elemanlı ve bu yüzden dizide 2+1=3'üncü eleman diye bir şey kalmadı.*/
}
```

arrayAdi.Add(birSey : object): İsmi verilen arrayin en son elemanından sonra yeni bir eleman oluşturur ve ona birSey'in değerini verir. (birSey yerine istersen "merhaba" gibi bir String veya başka bir obje değeri girebilirsin.)

arrayAdi.length : int : İsmi verilen arrayin kaç elemandan oluştuğunu söyler.

arrayListAdi.Count : int : İsmi verilen ArrayList'in (Bunlar normal *Array*'lerden biraz daha farklıdır.) kaç elemandan oluştuğunu söyler.

arrayListAdi.IndexOf(birSey : object): İsmi verilen ArrayList'in içerisindeki '*birSey*' elemanının (*Object* türünde olmalıdır.) arraylist'in kaçıncı elemanı olduğunu bulur ve *int* olarak geri döndürür. Eğer arraylist'te '*birSey*'in değeri yoksa -1 döndürür.

collider.ClosestPointOnBounds(birVektor : Vector3): Bir objenin, girilen *Vector3* şeklindeki birVektor noktasına olan en yakın noktasını *Vector3* şeklinde döndürmeye yarar.

Screen.width : int: Oyun ekranının pixel cinsinden genişliğini verir.

Screen.height : int: Oyun ekranının pixel cinsinden yüksekliğini verir.

Camera.pixelWidth : float: Girilen kameranın pixel cinsinden genişliğini verir. Tek kameralı oyunlarda değeri genelde Screen.width ile aynıdır. Ancak bir kamera örneğin minimap görevi görüyorsa; bir başka deyişle oyun ekranının sadece bir kısmını kapsıyorsa o zaman bu değişkenin değeri Screen.width'in değerinden farklı olur ve minimapın pixel cinsinden genişliğini döndürür.

Camera.pixelHeight : float: Girilen kameranın pixel cinsinden yüksekliğini verir. (Örnek kullanımı: *Camera.main.pixelHeight*)

Camera.ViewportPointToRay(pozisyon : Vector3): Aslında gerçekten karmaşık sayılabilecek bir komut. Kameradan başlayan ve ekranın (*pozisyon.x, pozisyon.y*) koordinatlarından geçen bir ray döndürür. Bu yüzden pozisyon vektörünün *z* argümanı önemsizdir, ihmal edilir. Daha sonra bu ray ile *Raycast* yapılabilir. Ekranın en sağında *pozisyon.x* 1, en üstünde *pozisyon.y* değişkenleri 1 değerini alır. Bu döndürülen ray'i kullanarak oluşturulan *Raycast*'ı oluşturmanın uzun yolu ise "*Physics.Raycast(Camera.main.transform.position, (Camera.main.ViewportToWorldPoint(pozisyon) - Camera.main.transform.position));*" yapmaktır. Kısa yol ise "*Physics.Raycast(kamera.ViewportPointToRay(pozisyon));*"dur. Örneğin kameranın görüş alanının tam ortasından çıkan ve kameranın baktığı yön doğrultusunda giden bir raycast oluşturmak için şu komut kullanılabilir:

```
var vurus : RaycastHit;
Physics.Raycast( Camera.main.ViewportPointToRay( Vector3( 0.5, 0.5, 1.0 ) ), vurus );
```

Camera.ScreenPointToRay(pozisyon : Vector3): Anladığım kadarıyla ViewportPointToRay(pozisyon) komutundan sadece bir farkı var: Viewport komutunda ekranın en sağ üstü (1,1) iken bu komutta ekranın en sağ üstü (Camera.pixelWidth, Camera.pixelHeight). Yani kameranın verdiği görüntünün pixel cinsinden eni ve boyu.

Screen.lockCursor : boolean: Mouse'nin ekranda gözükmemesi ve olduğu yerden kımıldamaması, böylece oyunu oynarken yanlışlıkla başka yerlere tıklayarak oyun ekranının aktifliğini kaybetmemesi için güzel bir komut. Eğer "*Screen.lockCursor = true;*" yapılırsa mouse kilitlenir, "*Screen.lockCursor = false;*" yapılırsa kilit açılır.

NOT: Bu özellik maalesef Unity editör ekranında çalışmamaktadır.

transform.DetachChildren(): Objenin eğer varsa tüm child objelerini child'lık durumundan çıkarır, yani hiyerarşiyi kırar. Örneğin normalde *Destroy(gameObject)* komutu bir objeyi child objeleriyle beraber yok etmeye yarar ancak eğer child objelerin yok edilmesini istemiyorsak şöyle bir çözüm yolu bulunmaktadır:

```
transform.DetachChildren();
Destroy( gameObject );
```

transform.IsChildOf(birBaskaTransform : Transform): Bir objenin başka bir objenin child objesi olup olmadığını döndürür (Eğer child ise *true*, değilse *false* döndürür.). Örneğin:

```
/* obje1'in, scriptin atandığı objenin bir child'ı olup olmadığına bakan bir script */
```

```
var obje1 : GameObject;
function Start() {
	if( obje1.transform.IsChildOf( transform ) ) {
		print( "obje1, bu objenin bir child objesi!" );
	}
	else {
		print( "obje1, bu objenin bir child objesi değil!" );
	}
}
```

Invoke("Metod İsmi" : String, saniyeMiktari : float): İsmi girilen metodu (Kullanıcı tarafından oluşturulmuş bir fonksiyonu) girilen *saniyeMiktari* kadar saniye geçtikten sonra çalıştırmaya yarar. Örneğin:

```
function Start() {
	Invoke( "BeniKlonla", 1.5 ); /* Script çalıştırılmaya başlandıktan 1.5 saniye sonra BeniKlonla fonksiyonu çağrılarak (0,0,0) pozisyonunda objenin bir klonu oluşturuluyor. */
}
function BeniKlonla()
{
	Instantiate( gameObject, Vector3.zero, Vector3.zero );
}
```

CancelInvoke("Metod İsmi" : String): Kodun yazıldığı scriptte "*Metod İsmi*" isimli metod *Invoke* edilmişse bu *Invoke* işlemini iptal eder. Eğer "*Metod İsmi*" boş bırakılırsa (*CancelInvoke()* şeklinde) mevcut scriptte çalıştırılmış olan tüm *Invoke* işlemlerini iptal eder (*Invoke* edilmiş metodların isimlerine bakmaksızın).

Resources.Load("Assetin adı" : String): Project panelinde eğer herhangi bir yerde bir *Resources* klasörü varsa, onun içindeki bir asset ile işlem yapmaya yarar. Dilenirse içerisinde 2. bir parametre olarak assetin ne türünde olduğu belirtilebilir. Böylece örneğin 2 farklı Resources klasöründe aynı isimli 2 farklı asset varsa, Unity'e hangisinden bahsettiğimizi anlatabiliriz:

```
guiTexture.texture = Resources.Load("dusman", Texture2D);
var dusman : GameObject = Instantiate(Resources.Load("dusman", GameObject));
```

@script RequireComponent(AudioSource): Scriptin uygulandığı GameObject'te scriptin çalışması için "Audio Source" Component'inin bulunmasını zorunlu kılar. Eğer yoksa kendi otomatik olarak ekler. Mesela scriptin yazıldığı objede bir ses dosyası çalınacaksa 'Audio Source' componenti gerekli olduğundan bu componentin varlığından emin olmaya yarar. Genelde scriptin en sonuna yazılır bu kod ve en önemli özelliği, bu kodun sonuna ";" (Noktalı virgül) işareti konulmaz. Çünkü bu bir JavaScript kodu değil, Unity motorunun kendine has bir kodudur.

#pragma strict: Scriptte katı kuralların uygulanmasını sağlar. Yani mesela bir değişken tanımlarken onun **türünü de yazmak zorunda olmanızı** sağlar. Bence güzel bir özellik. Bu kodun sonuna noktalı virgül "**;**" konulmaz. Bu komut ayrıca zorunlu kıldığı bu özellik sayesinde performans artışı da sağlar.

• BAŞKA ÖNEMLİ BİLGİLER

String Olmayan Bir Değişkenin Bir String İçerisinde Kullanımı

Örneğin bir 'int' değişkenini bir yazı içinde direk göstermeye kalkarsanız hata alırsınız. Bunun yerine şöyle bir kullanım yapılırsa problem çözülür:

```
var sayi : int = 45;
var yazi : String = ""+sayi;
```

Burada yaptığımız şey *'yazi=sayi'* demek yerine (Böyle yazarsanız hata alırsınız.) '*yazi=""+sayi*' demek oldu. Burada dilersek iki tırnak işareti arasına ekstra bir yazı girebiliriz. Mesela '*yazi="Sayinin Degeri: "+sayi;*' gibi.

Bir Objede Herhangi Belirli Bir Component Var mı Yok mu Kontrol Etmek

```
if(objem.rigidbody){
…
}
```

Böyle bir kullanımda eğer '*objem*' objesinin '*rigidbody*' component'i varsa '*if*'in içindeki yapılacak şeyler çalıştırılır. Eğer biz '*objem*'de '*rigidbody*' yoksa bir şeyler yapmak istiyorsak en başa bir '**!**'(**Ünlem İşareti**) koyabiliriz, yani: '*if(!objem.rigidbody)*' şeklinde.

Bölme Yerine Çarpma Kullanmak (Garip Bir Şey)

Bir değişkenin değerinin yarısını bulmak için "*degisken/2*" yazılır kısa yoldan ancak gariptir ki "*degisken * 0.5*" yazarsan Unity (Daha doğrusu bilgisayarın kendisi, yani oyun motoruyla alakalı bir şey değil bu.) sonucu daha az CPU harcayarak buluyor(muş). Çünkü bilgisayar çarpma işlemini bölme işlemine göre daha rahat yapıyor(muş). Sebebini bilmiyorum ama bu bilgiyi birden çok yerde gördüm ve madem öyle biz de çarpma işlemini kullanırız o halde :)

'this' Komutu

Bazen bir kullanıcı tanımlı fonksiyonun içerisinde *this.birDegiskenAdi* şeklinde bir kullanıma rastlayabilirsin. Bu aslında çok basit bir şey, bu yüzden bunu ufak bir örnekle anlatayım:

```
var yazi : String = "bir";
function Start()
{
	Degistir("iki");
}
function Degistir(yazi : String)
{
	this.yazi = yazi;
	print(yazi); /*Console ekranına "iki" yazdırır.*/
}
```

Burada da gördüğün gibi kullanıcı tanımlı bir fonksiyon olan *Degistir*'in içinde *this.yazi = yazi* komutunu kullandım. Peki neden? Çünkü *Degistir* fonksiyonuna gelen argümanın ismi ne?: *yazi* – Peki bizde de *yazi* diye bir değişken zaten var mı? Var. İşte burada biz bu 2 farklı şeyin değer ataması sırasında (Değişken ile argüman) birbirine karışmaması için *this.yazi* ile ***yazi* değişkenini** ifade ediyoruz (*this* böyle durumlarda her zaman değişkeni ifade etmeye yarar.). Ardından *this.yazi = yazi* diyerek *yazi* değişkeninin değerini *yazi* argümanının değerine eşitliyoruz. Eğer orada *this* kullanmasaydık ortaya *yazi=yazi* gibi saçma bir ifade çıkacaktı ve Unity bize uyarı verecekti ve yapmak istediğimiz şey de gerçekleşemeyecekti. Peki bu işlemi şöyle yapamaz mıydık:

```
...
function Degistir(birYazi : String)
{
	yazi = birYazi;
	print(yazi); /*Console ekranına "iki" yazdırır.*/
}
```

Yapabilirdik tabi ki. Peki ben bunca satır boyunca niçin *this*'in ne işe yaradığını anlatmakla uğraştım? Çünkü bunun kullanımı özellikle yabancı kimseler tarafından çok yaygın ve artık böyle bir *this* kullanımıyla karşılaştığında aslında *this*'in orada değişkeni ifade etmek için kullanıldığını bilmiş olacaksın.

Tek Basamaklı Bir Sayıyı (Yani Bir Rakamı) İki Basamakta Göstermek (Saatlerde Olduğu Gibi)

Oyununda bir sayaç olduğunu düşün. Eğer ki bu sayacı output edersen (ekranda gösterirsen) ve saniye 10'dan küçükse, yani mesela 6 ise ekranda saniye '*06*' olarak değil de sadece '*6*' olarak gözükür. Ama bu güzel durmaz çünkü bir sayaç ' *2:06* ' şeklinde gözükür normalde, ' *2:6* ' gibi garip bir şekilde değil. Bunu ayarlamak için basit bir komut var:

```
saniye.ToString( "D2" );
```

Bu sayede Unity '*saniye*' değişkeninin değerini her zaman (saniye 3 veya daha çok basamaklı olmadığı sürece) 2 basamaklı olarak gösterir. Bunu mesela "*D3*" yapsaydım 3 basamakta gösterirdi, ' *2:006* ' gibi. Örnek bir kullanımı:

```
guiText.text = "" + dakika + ":" + saniye.ToString( "D2" );
```

Son olarak, buradaki *ToString()* fonksiyonunun işlevi ise '*saniye*' değişkeninin değerini o satırdaki komuta has *String* türüne çevirmektir. Aynı şekilde isteseydim ' *dakika.ToString()* ' de yazabilirdim ama parantez içine bir parametre girmediğim için buna gerek yok.

Bu komutun float'lar için olan bir başka güzel versiyonu da mevcut. Örneğin bir virgüllü sayının virgülden sonra sadece 3 basamağını göstermek istiyorsak yazmamız gereken kod şu:

```
floatSayi.ToString( "F3" );
```

Performans Canavarı Kodlar Oluşturmak

Burada anlatılan yöntem Unity'nin direkt Unity'nin dökümanından aldığım bir şey ve gerçekten çok ilginç. Eğer büyük bir oyun projesinde çalışıyorsan tavsiye edilen bir yöntem. Şu örnekten yola çıkayım:

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

Bu basit kodun yaptığı şey objeyi her karede 5 metre ileri götürmek. Bu kodu şöyle değiştirirsek biraz daha fazla kod yazmış oluyoruz ancak çok daha fazla performans alıyoruz:

```
private var transformComponenti : Transform;
function Awake()
{
   transformComponenti = transform;
}
function Update()
{
   transformComponenti.Translate(0, 0, 5 * Time.deltaTime);
}
```

Burada görüldüğü üzere objenin *Transform component*'ini *Awake()* fonksiyonunun içinde bir değişkene (Değişkenin *private* olması zorunlu değil. Ancak *Inspector*'da gözükmesinin hiçbir anlamı olmadığı için *private* tavsiye edilir.) atıyoruz (*Awake* fonksiyonunun *Start* fonksiyonundan farkları; objedeki scriptin tikli olup olmasına bakmazsızın, *Awake* fonksiyonunun obje oluşturulduğu anda tek seferlik çalıştırılması ve de ne olursa olsun *Awake* fonksiyonunun her zaman *Start* fonksiyonundan önce çalıştırılmasıdır.). Ardından ne zaman *Transform componen*ti ile iş yapacak olsak bu değişkeni kullanıyoruz. Bu sayede Unity her bir karede objenin Transform componentini tekrar ve tekrar aramak yerine bu componentin yerini zaten depolamış olan değişkeni kullanarak büyük bir performans kazancı sağlıyor. Bir başka örnek:

```
private var theRigidbody : Rigidbody;
function Awake()
{
   theRigidbody = rigidbody;
}
function FixedUpdate()
{
   theRigidbody.AddForce(Vector3.up * Time.deltaTime);
}
@script RequireComponent(Rigidbody)
```

Inspector'u Debug Modunda Çalıştırmak

Çok basit ama az bilinen bir özellik. Inspector panelindeki "*Inspector*" yazısına sağ tıklayıp "*Debug*"u seçersen Debug moduna geçiş yaparsın. Bu moddayken objelerin rotasyonları *EulerAngles* cinsinden değil de *Quaternion* cinsinden gözükür ve scriptlerde yer alan *private*

değişkenlerin değerleri de Inspector'da, gri renkli şekilde görünür (ama değiştirilemez). Ancak *static* değişkenlerin değerleri ne olursa olsun Inspector'da görünmez.

Mevcut Andaki Saat, Dakika ve Saniye Değerlerine Ulaşmak

Çok çeşitli sebeplerden dolayı oyun sırasında saatin kaç olduğuna bakıp ona göre işlem yapmak isteyebilirsiniz (Örneğin saat geceyarısını gösteriyorsa akşam, öğleni gösteriyorsa gündüz atmosferi olan bir yarış oyunu için). Zaman değerlerine ulaşmak için *Javascript*'te şu kodu kullanabilirsin:

```
public static var saat : int;
public static var dakika : int;
public static var saniye : int;
function Update () {
        var zaman : System.DateTime = System.DateTime.Now;
        saat = zaman.Hour;
        dakika = zaman.Minute;
        saniye = zaman.Second;
}
```

Veya *C#* kullanacağım diyorsan aşağıdaki kodu kullanabilirsin:

```
using UnityEngine;
using System.DateTime;

public class Zaman : MonoBehaviour {
        public static int saat, dakika, saniye;
        void Update() {
                DateTime zaman = DateTime.Now;
                saat = zaman.Hour;
                dakika = zaman.Minute;
                saniye = zaman.Second;
        }
}
```

Bu kodu "*Zaman*" adında yeni bir scripte yapıştırıp bu scripti istediğin GameObject'e ata. Ardından herhangi bir scriptte "*Zaman.saat*", "*Zaman.dakika*", "*Zaman.saniye*" değişkenleriyle direkt mevcut zamana ulaşabilirsin. Ancak dikkat et: Eğer *C#* kodunu kullandıysa bu değişkenlere direkt yoldan sadece *C#* scriptlerinden, *Javascript* kodunu kullandıysan direkt olarak sadece *Javascript* scriptlerinden ulaşabilirsin. Yani *C#*'taki bir *static* değişkene *Javascript*'ten direkt ulaşamazsın!

Public Bir Sayının Inspector'da Alabileceği Değere Alt ve Üst Limit Belirlemek

Diyelim ki elimizde **sayi** adında **public** bir **int** var ve bu sayının değerinin **Inspector**'dan sadece -100 ile 100 arasında verilebilmesini istiyoruz. O zaman değişken tanımlamamızı şöyle yapıyoruz:

```
@Range(-100, 100)
public var sayi : int;
```

C# kullanıyorsan şöyle yapacaksın:

```
[Range(-100, 100)]
public int sayi = 0;
```

Yalnız dikkat et: bu değişkene scriptteki fonksiyonlarda -100 ile 100 limiti dışında da değerler verebilirsin ve bu değer otomatik olarak -100 ile 100 arasına dönmez! Yani **limit koyma işlemi sadece Inspector'dan değer verirken geçerlidir.**

C# Scriptindeki Bir private Değişkeni Inspector'da Göstermek/Düzenlemek

Normalde private değişkenler Inspector'da gözükmez. Ama C#'ta bir private değişkenin bir üst satırına şu kodu eklersen o değişken Inspector'da görünür ve değeri değiştirilebilir:

```
@SerializeField
```

C# için:

```
[SerializeField]
```

Scriptteki Bir Fonksiyonu Inspector'dan Çağırmak

Eğer scriptindeki static olmayan bir fonksiyonu belirsiz bir sebepten ötürü Inspector'dan çağırabilmek istiyorsan fonksiyonun bir üstüne şu kodu yazabilirsin:

```
@ContextMenu( "Aciklayici Ufacik Bir Yazi" )
```

Eğer böyle yaparsan Inspector'da o component'in sağındaki dişli ikona tıklayınca "*Aciklayici Ufacik Bir Yazi*" isminde bir buton görünür ve bu butona tıklayınca fonksiyon editörde çalıştırılır. Kodun **C#** hali için ise önceki örneklerde olduğu gibi @ işaretini silip kalan kısmı [ve] işaretleri içine alabilirsin.

Niçin böyle bir şey isteyeyim diyebilirsin. Örneğin **Transform** component'inde "*Reset Position*" butonu var; ona tıklarsan pozisyon sıfırlanıyor. Kendi component'inde de buna benzer bir işlev yapmak istersen işte o zaman ContextMenu işine yarar.