CE318: High-level Games Development

Lecture 6: GUIs, Menus, Particle Systems

Diego Perez

dperez@essex.ac.uk
Office 3A.527

2016/17

Outline

Outline

The Graphical User Interface

The **Graphical User Interface** (or simply **GUI**) in games is in charge of presenting the player the appropriate amount of relevant information to facilitate taking decisions. This could be presented in the form of menus or as a **Heads-Up Display** (or **HUD**).

A good design of the GUI system is essential for the success and playability of a game. The game could be unplayable if the information provided is too little, as the player might not know what to do, but also if it is too much, as this could confuse the player. The HUD is frequently used to simultaneously display several pieces of information including the main character's health, items, and an indication of game progression.

Both the GUI (and especially the HUD) can influence importantly in the **immersion** the player experiences when playing the game. In this lecture, we'll examine Unity's GUI, including HUD possibilities, menus, windows, panes and a few effects.

The Canvas (1/2)

The **Canvas** is the area that holds all UI elements. In Unity, it is included into the scene as a game object, with a Canvas component on it. All the UI elements are children game objects of this canvas. The Canvas area is shown as a rectangle in the *Scene View*, and it's a good idea to switch it to 2D view to operate with UI elements..

Draw order of elements: UI elements in the Canvas are drawn in the same order they appear in the Hierarchy. If two UI elements overlap, the later one will appear on top of the earlier one..

An important parameter of the canvas is the render mode, that can be set to:

- **Screen Space - Overlay:** UI elements are rendered on top of the screen. The canvas will fill the screen automatically, and it will resize if the screen dimensions change.
- **Screen Space - Camera:** UI elements are rendered at a certain distance from the camera, and camera settings (field of view, projection, etc.) affect how the elements are rendered.
- **World Space:** the Canvas behaves as any other object in the scene (useful for UIs that are meant to be a part of the world).

The Canvas (2/2)

Other parameters of the Canvas component are:.

Pixel Perfect: UI elements are adjusted to the nearest pixel.

Render Camera: Camera that renders the UI elements in *Screen Space - Camera* render mode.

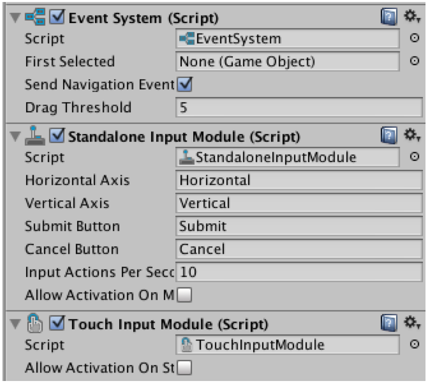
Plane Distance: Distance between Canvas and camera in *Screen Space - Camera* render mode.

Event Camera: Determines which camera should receive the event of the player clicking in UI elements in *World Space* render mode.

Sorting Layer and **Order in Layer:** control the render order of the canvas when compared to other renderers on the scene.

UI Events and Triggers (1/2)

The interactivity of a UI element placed on the screen is driven by **Events**. Whenever a Canvas is created on a scene, an *EventSystem* game object is automatically added to the scene. This game object has several components, including a Event System script and one or more input systems.



✓ Event System (Script)

Script	EventSystem
First Selected	None (Game Object)
Send Navigation Event	✓
Drag Threshold	5

✓ Standalone Input Module (Script)

Script	StandaloneInputModule
Horizontal Axis	Horizontal
Vertical Axis	Vertical
Submit Button	Submit
Cancel Button	Cancel
Input Actions Per Secc	10
Allow Activation On M	☐

✓ Touch Input Module (Script)

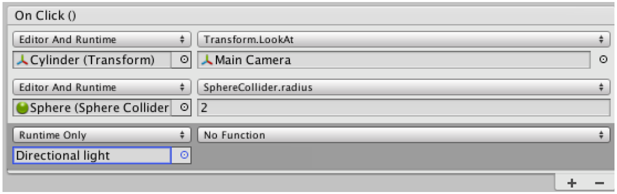
Script	TouchInputModule
Allow Activation On St	☐

The **First Selected** property allows to select (not to click!) a UI element when the scene starts. This is useful for non-pointer based UI Systems (game pads).

You can add an **Event Trigger** component to any game object to trigger an event when the user interacts with it. Most UI components respond to at least one event by default.

UI Events & Triggers (2/2) *

To catch a specific event, a list of event handlers can be associated to each UI element:



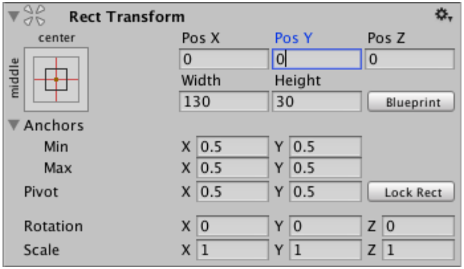
On Click ()

Editor And Runtime	Transform.LookAt
Cylinder (Transform)	Main Camera
Editor And Runtime	SphereCollider.radius
Sphere (Sphere Collider	2
Runtime Only	No Function
Directional light	

For each function you can specify:

- **Execution mode:** *Off* (disabled), *Runtime Only* (no events while in editor) or *Editor and Runtime*.
- **Object:** Object to manipulate.
- **Function:** Function to call when the event is triggered, on the object selected. The functions needs to be a `public void` with 0 or 1 parameters. You will be able to choose between available Unity functions or from your scripts.
- **Parameters:** An optional parameter that can be an `int`, `float`, `string`, `bool` or `Object`.

The UI Rect Transform (1/3)



Rect Transform

center	Pos X	Pos Y	Pos Z
middle	0	0	0
	Width	Height	
	130	30	Blueprint
Anchors			
Min	X 0.5	Y 0.5	
Max	X 0.5	Y 0.5	
Pivot	X 0.5	Y 0.5	Lock Rect
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

The **UI Rect Transform** is a specific transform component for UI elements. It is used to specify the size, position and rotation of all UI elements. Canvas have their own Rect Transform, although it is editable only in the *World Space* render mode.

Scaling vs. Resizing: It is recommended not to scale UI elements, but resizing them instead, as this does not affect font sizes, border on sliced images, etc. However, scaling can be useful for animated effect or other special effects. A negative size turns an UI element invisible, while a negative scale flips it.

An UI element can be anchored to its parent if the parent has also a Rect Transform. This allows the element to re-position and re-size based on the parent's Rect Transform.

The UI Rect Transform (2/3)

Pos (X, Y, Z): Position of the rectangles pivot point relative to the anchors.

Width/Height: Width and height of the rectangle (to resize).

Anchors Min: The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle. 0,0 corresponds to anchoring to the lower left corner of the parent, while 1,1 corresponds to anchoring to the upper right corner of the parent.

Anchors Max: The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle. 0,0 anchors to the lower left corner of the parent, while 1,1 anchors to the upper right corner.

Pivot: Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

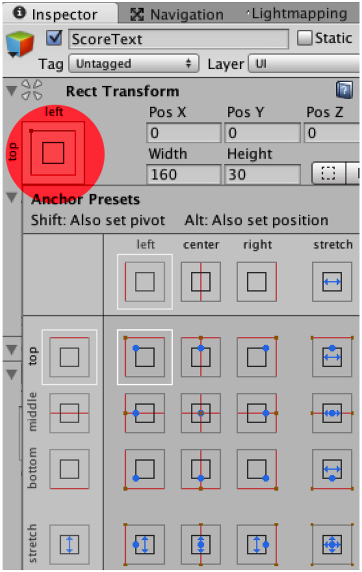
Rotation: Angle of rotation (in degrees) of the object around its pivot point along the X, Y and Z axis.

Scale: Scale factor applied to the object in the X, Y and Z dimensions.

The UI Rect Transform (3/3) *

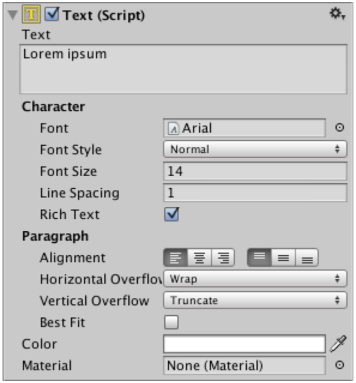
There are a few anchor presets available that correspond to the most used settings. These can be accessed by clicking on the button highlighted in the image in this slide.

By pressing *Shift* and *Alt* keys, you will be able to set pivot point and the position of the UI element, respectively.



The Visual Components - UI Text

The Text control displays a non-interactive piece of text to the user. This can be used to provide captions or labels for other GUI controls or to display instructions or other text.



Text (Script)

Text

Lorem ipsum

Character

Font	Arial
Font Style	Normal
Font Size	14
Line Spacing	1
Rich Text	✓

Paragraph

Alignment	
Horizontal Overflow	Wrap
Vertical Overflow	Truncate
Best Fit	☐
Color	⬜
Material	None (Material)

You can specify setting of the text such as font, colour, size, alignment, etc.

The Visual Components - UI Image

The Image control displays a non-interactive image to the user. This can be used for decoration, icons, etc, and the image can also be changed from a script to reflect changes in other controls. The image to display must be imported as a Sprite to work with the Image control.

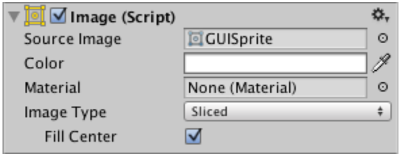


Image (Script)	✓
Source Image	GUISprite
Color	
Material	None (Material)
Image Type	Sliced
Fill Center	✓

The image type can be set to:

- *Simple:* The image scales to fit the control rectangle, unless *Preserve Active* is enabled.
- *Sliced:* The image is treated as a "nine-slice" sprite with borders.
- *Tiled:* The image is kept at its original size but is repeated as many times as necessary to fill the control rectangle.
- *Filled:* The image is displayed as with the Simple method but can also be made to grow gradually from an empty image to a completely filled one.

The Interaction Components - UI Button

The Button control responds to a click from the user and is used to initiate or confirm an action. The UI Button game object comes with an **Image** and a **Button** components, which allow to establish different settings such as different colours, sprites, image and fade duration.

- *Interactable:* Check if this buttons should accept user input.
- *Transition:* Determines if the button should be provided with different colours, sprites or animations when it is clicked, released, or the mouse is over it.
- *Navigation:* Allows to specify how to move between UI elements using the arrow keys.
- *OnClick():* list of functions that will be called when the button is clicked.

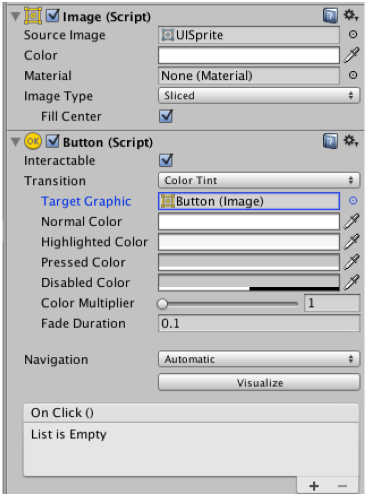


Image (Script)

Source Image	UISprite
Color	
Material	None (Material)
Image Type	Sliced
Fill Center	✓

Button (Script)

Interactable	✓
Transition	Color Tint
Target Graphic	Button (Image)
Normal Color	
Highlighted Color	
Pressed Color	
Disabled Color	
Color Multiplier	1
Fade Duration	0.1
Navigation	Automatic

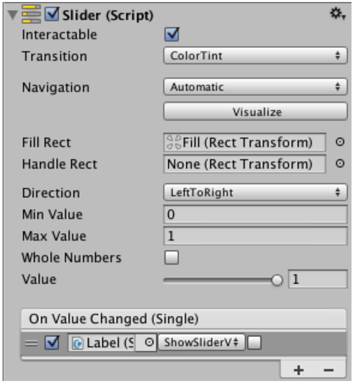
On Click ()

List is Empty

The Interaction Components - UI Slider

The **Slider** control allows the user to select a numeric value from a predetermined range by dragging the mouse. The UI Slider game object comes with a **Slider** component. It comes with setting such as Interactable, Transition, Navigation and list of functions as in the button case, plus:

- Direction: *Left to Right*, *Right To Left*, *Up to Bottom*, *Bottom to Up*.
- Min and Max Values for the slider.
- Whole numbers: avoid decimal point values.
- Value: current and default value for the slider.


Slider (Script)

Interactable	✓
Transition	ColorTint
Navigation	Automatic
	Visualize
Fill Rect	Fill (Rect Transform)
Handle Rect	None (Rect Transform)
Direction	LeftToRight
Min Value	0
Max Value	1
Whole Numbers	☐
Value	1

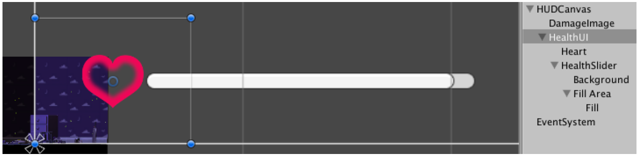
On Value Changed (Single)

✓ Label (S ShowSliderV ☐

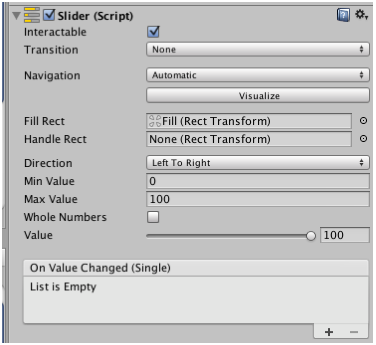
Additionally, it's possible to specify a background and several properties for the bar and the handle of the slider. The Slider game object comes with 3 children objects for this.

Example: A health bar display (1/3)

This example consists of an UI Image (Heart) and a UI Slider (Health Slider), which contains a background and a fill area:



HealthSlider (Slider Script):



Slider (Script)

Interactable	✓
Transition	None
Navigation	Automatic
	Visualize
Fill Rect	Fill (Rect Transform)
Handle Rect	None (Rect Transform)
Direction	Left To Right
Min Value	0
Max Value	100
Whole Numbers	☐
Value	100

On Value Changed (Single)

List is Empty

Example: A health bar display (2/3)

Background Image:

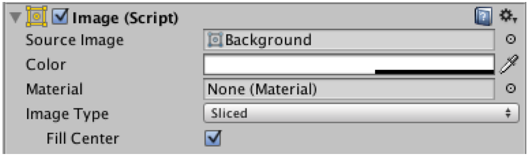


Image (Script)

Source Image	Background
Color	
Material	None (Material)
Image Type	Sliced
Fill Center	✓

Fill Image:

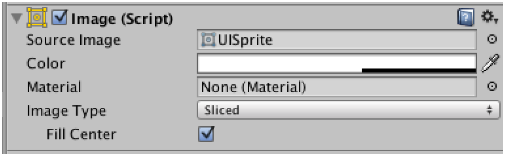
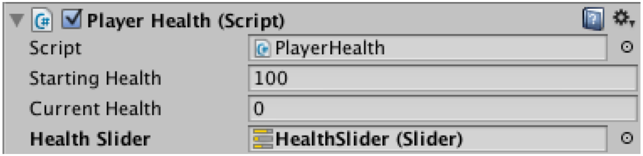


Image (Script)

Source Image	UISprite
Color	
Material	None (Material)
Image Type	Sliced
Fill Center	✓

Example: A health bar display (3/3)

Changing the health value and the slider from a script is simple:


Player Health (Script)

Script	PlayerHealth
Starting Health	100
Current Health	0
Health Slider	HealthSlider (Slider)

```
public class PlayerHealth : MonoBehaviour
{
    public int startingHealth = 100;
    public int currentHealth;
    public Slider healthSlider;

    void Awake (){
        currentHealth = startingHealth;
    }

    //...

    public void TakeDamage (int amount){
        currentHealth -= amount;
        healthSlider.value = currentHealth;
    }
}
```

Outline

Fading In and Out (1/3)

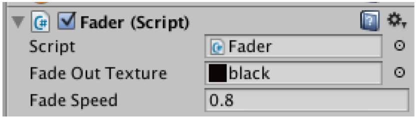
We'll see an example of how to fade in and out a texture. First, we'll create an script for managing this effect. We can assign it to any game object in the scene (i.e. game manager, camera, etc.).

```
using UnityEngine;
using System.Collections;

public class Fader : MonoBehaviour {

   //texture that will overlay the screen.
   public Texture2D fadeOutTexture;

   //fading speed.
   public float fadeSpeed = 0.8f;

   //the texture's alpha value (0 to 1).
   private float alpha = 1.0f;

   //order in the draw's hierarchy (highest priority).
   private int drawDepth = -1000;

   //fade direction: in = -1 (towards texture not visible),
   //                out = 1 (towards texture visible).
   private int fadeDir = -1;

   //...
}
```

Fading In and Out (2/3)

```
    //...
    void OnGUI() {
       //Modify the alpha value gradually and
       // use deltaTime to talk in seconds.
       alpha += fadeDir * fadeSpeed * Time.deltaTime;

       //Set colour of our texture. Keep the color the same and
       // change the alpha channel.
       GUI.color = new Color (GUI.color.r, GUI.color.g, GUI.color.b,
            alpha);
       GUI.depth = drawDepth;
       Rect dimension = new Rect (0, 0, Screen.width, Screen.height);
       GUI.DrawTexture (dimension, fadeOutTexture);
    }

    public float BeginFade(int direction){
       fadeDir = direction;
       return 1.0f/fadeSpeed;
    }

    void OnLevelLoaded(){
       BeginFade (-1);
    }
}
```

We can set any texture for the fade in/out effect. A black texture would work pretty well.


Fader (Script)

Script	Fader
Fade Out Texture	black
Fade Speed	0.8

Fading In and Out (3/3)

To start the fader, we can call a function that starts the following coroutine:

```
IEnumerator ChangeLevel()
{
   Fader fader = GameObject.Find("MainCamera").GetComponent<Fader>();
   float fadeTime = fader.BeginFade(1);
   yield return new WaitForSeconds(fadeTime);
   SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex+1);
}
```

`public static void LoadScene(int index)` and `public static void LoadScene(string name)` load a scene by its name or index.

Before you can load a scene you have to add it to the list of scenes used in the game. Use *File* → *Build Settings* in Unity and add the scenes you need to the scene list there.

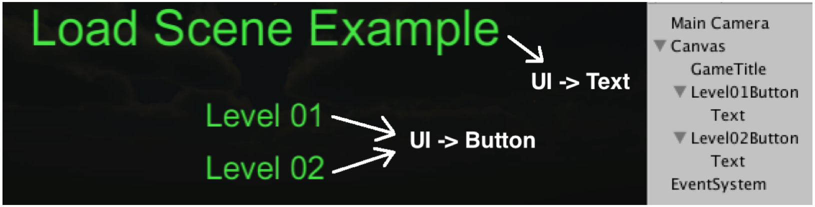
Calling `Application.LoadScene` will update:

- `SceneManager.GetActiveScene().buildIndex` (scene index loaded).
- `SceneManager.GetActiveScene().name` (the name of the level that was last loaded).

When a new scene is loaded, all game objects from the current scene are **destroyed**.

Creating a Simple Menu (1/3)

A menu to select levels (scenes) to play can be placed in an **independent scene**. This is an example of a menu with two levels to load, and the *Hierarchy View* of the scene:

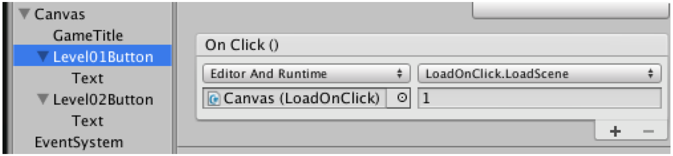


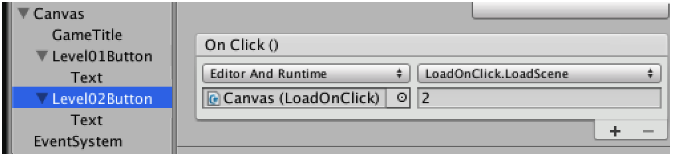
The Canvas object has a script (*LoadOnClick.cs*) component, with the following code:

```
using UnityEngine;
using System.Collections;

public class LoadOnClick : MonoBehaviour {

  public void LoadScene(int levelIdx)
  {
    SceneManager.LoadScene (levelIdx);
  }
}
```

Creating a Simple Menu (2/3)

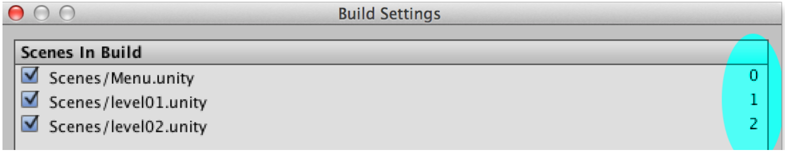
Each one of the buttons has a *Button (Script)* component, that holds the list of functions to call when this button is clicked. As LoadOnClick has a `public void` function with one parameter (*LoadScene*), this function appears in the available methods to call when the object to hold the trigger is the canvas. Each button passes a different value for the parameter, depending on the level to load:





Creating a Simple Menu (3/3)

The parameter passed to the function corresponds to the order in which the scenes are loaded in the build settings (*File* → *Build Settings*).

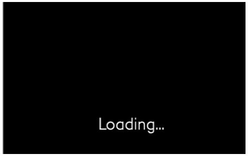

Build Settings

Scenes In Build

✓	Scenes/Menu.unity	0
✓	Scenes/level01.unity	1
✓	Scenes/level02.unity	2

To add a loading screen, we can add an image with a black background colour and a text that indicates that the scene is loading. Set as a public variable of the same script, we can just set it to active when the event is captured:

```
public GameObject loadingImage;

public void LoadScene(int levelIdx)
{
    loadingImage.SetActive (true);
    SceneManager.LoadScene (levelIdx);
}
```



sceneLoaded and delegates

It is possible to add a **delegate** to the event `SceneManager.sceneLoaded` to detect when a scene has been loaded. A *delegate* is just a function that captures an event triggered by Unity.

```
using UnityEngine;
using System.Collections;

public class ChangeMusic : MonoBehaviour {
   public AudioClip level2Music;
   private AudioSource source;

   void Awake () {
      source = GetComponent<AudioSource>();
      SceneManager.sceneLoaded = mySceneLoaded;
   }

   void mySceneLoaded(){
      int level = SceneManager.GetActiveScene().buildIndex;
      if (level == 2){
         source.clip = level2Music;
         source.Play ();
      }
   }
}
```

Loading Additively

It is possible to add all the objects from a scene into the current scene by using a different loading mode: *SceneManagement.LoadSceneMode mode = LoadSceneMode.Additive*:

```
SceneManager.LoadScene (sceneIndex, LoadSceneMode.Additive);
SceneManager.LoadScene (sceneName, LoadSceneMode.Additive);
```

This call loads a level additively. Unlike *LoadSceneMode.Single* (default), it **does not** destroy objects in the current level. Objects from the new level are added to the current scene. This is useful for creating continuous virtual worlds, where more content is loaded in as you walk through the environment.

The following script can be added to an object in the scene where the objects from another scene are going to be loaded in:

```
using UnityEngine;
using System.Collections;

public class LoadAdditive : MonoBehaviour {

  public void LoadAddOnClick(int levelIdx)
  {
    SceneManager.LoadScene(levelIdx, LoadSceneMode.Additive);
  }
}
```

TimeScale - Pausing the game

A way to pause a game in Unity is via modifying the `float Time.timeScale` variable. *timeScale* is the scale at which the time is passing. It can also be used for slow motion effects.

When timeScale is:

- 1.0 the time is passing as fast as real-time.
- 0.5 the time is passing $2\times$ slower than real-time.
- 0 the game is basically **paused** (if all your functions are frame rate independent).

Also:

- *FixedUpdate* functions will not be called when timeScale is set to 0.
- If you lower timeScale it is recommended to also lower *Time.fixedDeltaTime* (physics timer) by the same amount.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        //Switches Time.timeScale between 0 & 1 if Escape is pressed.
        Time.timeScale = Time.timeScale == 0 ? 1 : 0;
    }
}
```

Outline

Billboarding (1/2)

Billboarding is a technique where 2D textures are drawn onto 3D rectangles (quads) placed in the scene. Usually, the quads are rotated towards the camera such that they will always face the viewer. This allows one to create the illusion of a model albeit at a much reduced cost. A common usage is trees in the background or clouds in the sky. There are different types of billboards:

- Spherical: Full rotation towards camera.
- Cylindrical: Rotation around one axis only (typically *Up*).
- Non-rotating (interleaved): Two quads forming 90 degrees to one another.

It is common to use billboards in the distance and replace them with models once the gamer gets sufficiently close. This is a common approach in many "level of detail" systems.

Billboarding (2/2)

Some textures work much better for billboarding that others:

- With trees, for instance, the trunk should be round as this prevents the gamer from noticing the rotation.
- Some simple trees do not look good at all with non-rotating billboards because the two planes are too obvious. A more complex and realistic tree overcomes this issue.
- Textures with transparent backgrounds are better. Use shaders.

With billboarding, one can draw a lot more objects than would otherwise be possible. In your game you can try to combine billboarding with models to ensure the gameplay experience is consistent.

Particle Systems (1/2)

Sprites (in 2D games) and meshes (in 3D) are ideal ways of depicting "solid" objects in a game. There are other entities in games, however, that are fluid and intangible in nature and consequently difficult to portray using meshes or sprites. **Particle Systems** is a graphics approach to capture the behaviour of these entities, such as moving liquids, smoke, clouds, flames and magic spells.

A particle system is composed by a collection of billboards (hence, they have a texture) or particles, where each one:

- is rotated towards the camera.
- has a starting random position, within a region of space shaped like a sphere, hemisphere, cone, box or any arbitrary mesh.
- begins its life when it is generated or *emitted* by its particle system.
- has a *lifetime*. When the time is up, it is removed from the system.
- has a velocity vector that determines the direction and distance the particle moves with each frame update.

Particle Systems (2/2)

The system's *emission rate* indicates roughly how many particles are emitted **per second**, although the exact times of emission are randomized slightly. The choice of emission rate and average particle lifetime determine the number of particles in the **stable** state (i.e., where emission and particle death are happening at the same rate) and how long the system takes to reach that state.

Particles are typically **faded** in (when they are created) and out (when destroyed) for smoother transitions, modifying the alpha component of its color. During its lifetime, it can undergo various changes (such as velocity, color, size, rotation) and be affected by forces as wind or gravity.

Particle Systems can be created in Unity as an object (*Create* → *Particle System*) or as a component for an object. When an object with a particle system is selected, the *Scene View* will contain a small *Particle Effect* panel with some simple controls that are useful for visualising changes you make to the systems settings. In this panel you can set the speed at which the particle simulation is played.

Varying Properties over Time

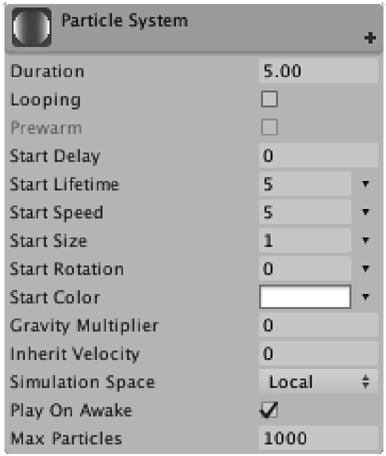
Many of the numeric properties of particles or even the whole system can be varied over time. Unity provides several different methods of specifying how the variation will happen:

- *Constant:* The property's value is fixed throughout its lifetime.
- *Curve:* The value is specified by a curve/graph.
- *Random between two constants:* Two constant values define the upper and lower bounds for the value; the actual value varies randomly over time between those bounds.
- *Random between two curves:* Two curves define the upper and lower bounds of the the value at a given point in its lifetime; the current value varies randomly between those bounds.

For color properties, such as Color over lifetime, there are two separate options:

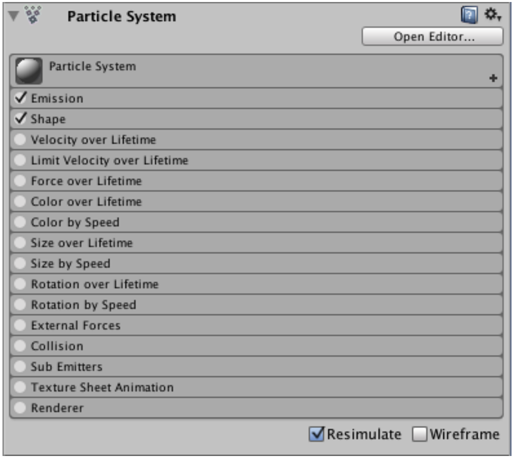
- *Gradient:* The color value is taken from a gradient.
- *Random between two gradients:* Two gradients define upper and lower bounds on the color value at a given time; the value used is a randomly weighted average of the two bound colors.

The Particle System Component



Particle System

Duration	5.00
Looping	☐
Prewarm	☐
Start Delay	0
Start Lifetime	5
Start Speed	5
Start Size	1
Start Rotation	0
Start Color	⬜
Gravity Multiplier	0
Inherit Velocity	0
Simulation Space	Local
Play On Awake	✓
Max Particles	1000



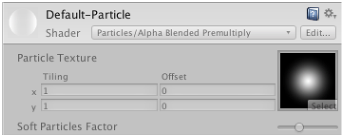
Particle System

Open Editor...

Particle System

- ✓ Emission
- ✓ Shape
- Velocity over Lifetime
- Limit Velocity over Lifetime
- Force over Lifetime
- Color over Lifetime
- Color by Speed
- Size over Lifetime
- Size by Speed
- Rotation over Lifetime
- Rotation by Speed
- External Forces
- Collision
- Sub Emitters
- Texture Sheet Animation
- Renderer

✓ Resimulate ☐ Wireframe



Default-Particle

Shader: Particles/Alpha Blended Premultiply — Edit...

Particle Texture

	Tiling	Offset
x	1	0
y	1	0

Select

Soft Particles Factor

The Particle System Component - Global Settings

Duration: The length of time the **system** will run.

Looping: If the system will start again at the end of its duration time.

Start Delay: Delay in seconds before the system starts emitting once enabled.

Start Lifetime: The initial lifetime for particles.

Start Speed: The initial speed of each particle in the appropriate direction.

Start Size / Rotation / Colour: The initial size / rotation / colour of each particle.

3D Start Size: 3D size of the particle at start.

Gravity Multiplier: Scales the physics manager's gravity value (0 : no gravity).

Inherit Velocity: If particles start with the particle system object's velocity

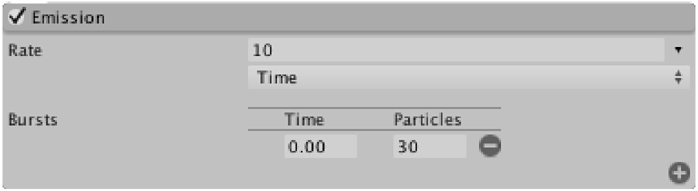
Simulation Space: Should particles be animated in the parent objects local space (and therefore move with the object) or in world space?

Play on Awake: If the system starts automatically when the object is created.

Max Particles: The maximum number of particles in the system at once. Older particles will be removed when the limit is reached.

The Particle System Component - Modules (1/12) *

Emission Module:



✓ Emission

Rate: 10 ▾ / Time

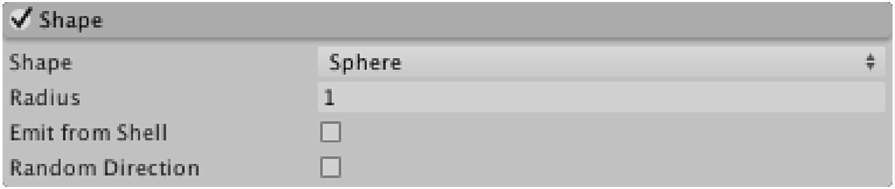
Bursts:

Time	Particles
0.00	30

Regulates the emission of particles from the system. It can be set as number of particles per time or per distance (number of particles released per unit of distance moved by the parent object). Bursts are used to add bursts of extra particles that appear at specific times.

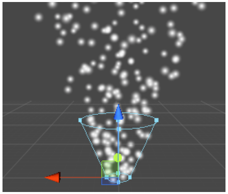
The Particle System Component - Modules (2/12) *

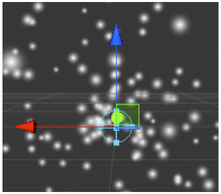
Shape Module:

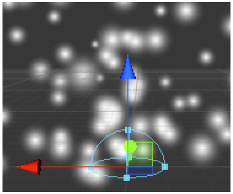

✓ Shape

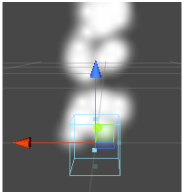
Shape	Sphere
Radius	1
Emit from Shell	☐
Random Direction	☐

Indicates the shape of the emission volume and where the particles are launched from within it. The shape can be a *Cone* (which angle and radius can be set), a *Sphere* (radius), a *Hemisphere* (radius), a *Box* (x, y and z dimensions), a *Mesh* (the proper mesh can be set) or others.



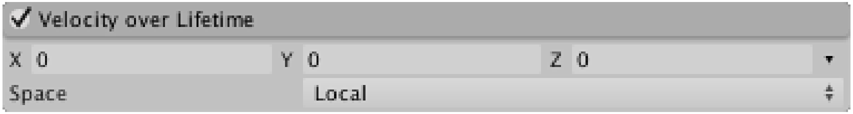






The Particle System Component - Modules (3/12) *

Velocity over lifetime:

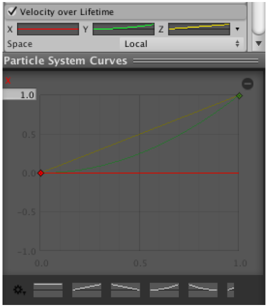


✓ Velocity over Lifetime

X 0 Y 0 Z 0

Space Local

Acceleration can be applied to change the velocities of particles over their lifetime. As all changes over time, it is possible to specify how this variation will happen. This is an example of variation over a curve:



✓ Velocity over Lifetime

X Y Z

Space Local

The Particle System Component - Modules (4/12) *

Other modules over Lifetime:

- **Limit Velocity Over Lifetime:** This module controls how the speed of particles is reduced over their lifetime.
- **Force Over Lifetime:** Particles can be accelerated by forces (wind, attraction, etc) that are specified in this module.
- **Color Over Lifetime:** This module specifies how a particles color and transparency vary over time.
- **Size Over Lifetime:** Many effects involve a particle changing size according to a curve, which can be set in this module.
- **Rotation Over Lifetime:** This module allows to arrange for particles to rotate as they move.

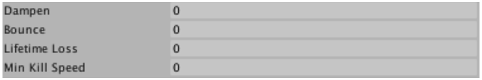
Modules by speed, that change according to its speed in distance units per second:

- **Color By Speed:** Changes the color of a particle
- **Size By Speed:** Changes the size of a particle.
- **Rotation By Speed:** Changes the rotation of a particle.

The Particle System Component - Modules (5/12)

Collision Module:

This module controls the way particles collide with solid objects in the scene. Collisions can be set to two different modes (World or Planes, see first property of the module). Five settings are common:



Dampen	0
Bounce	0
Lifetime Loss	0
Min Kill Speed	0

Dampen: The fraction of a particle's speed lost after a collision.

Bounce: Fraction of a particle's speed that rebounds after a collision.

Lifetime Loss: Fraction of a particle's total lifetime lost if it collides.

Min Kill Speed: Particles travelling below this speed after a collision will be removed from the system.

Send Collision Messages: If enabled, particle collisions can be detected from scripts by the *OnParticleCollision* function.

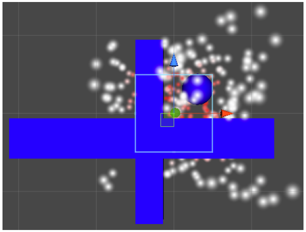
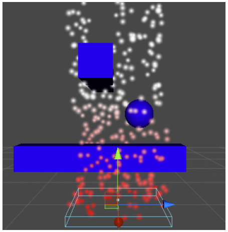
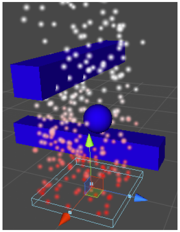
The *Dampen* and *Bounce* properties are useful when the particles represent solid objects. *Lifetime Loss* and *Min Kill Speed* can help to reduce the effects of residual particles following a collision.

The Particle System Component - Modules (6/12)

Collision Module - World Mode:

In the **World** collision mode particles collide with **any** collider in the scene. The processor overhead for this option is high, although it is more realistic. Two more settings are available in this mode:

- **Collides With**: Particles will only collide with objects on the selected layers.
- **Collision Quality**: This affects how strict collisions are (at lower quality levels, particles may sometimes pass through colliders).

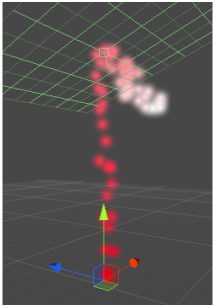




The Particle System Component - Modules (7/12) *

Collision Module - Plane Mode:

In the **Plane** collision mode, a list of transforms (typically empty GameObjects) can be added, as a set of planes to the scene that the particles will collide with. These planes extend infinitely in the objects' local XZ planes with the positive Y axis indicating the planes' normal vectors (although they can be rotated). Settings for this mode:

- **Planes:** An expandable list of Transforms that define collision planes. **Important:** planes should be added with the + sign, that creates an empty game object and assigns it as a plane.
- **Visualization:** Selects whether the collision plane gizmos will be shown in the scene view as wire-frame grids or solid planes.
- **Scale Plane:** Size of planes used for visualization.
- **Particle Radius:** Approximate size of a particle, used to avoid clipping with collision planes.

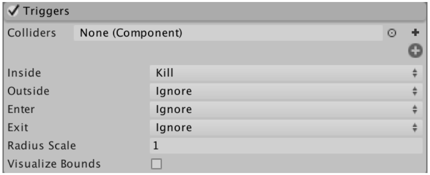


The Particle System Component - Modules (8/12)

Triggers:

Allows you to define a collider (i.e. a cube) that works as a trigger for the Particle System. Effects can be defined for particles when they are *inside*, *outside*, *enter* and/or *exit* the trigger. Options are:

- Ignore: nothing happens.
- Kill: kills the particle.
- Callback: custom function.



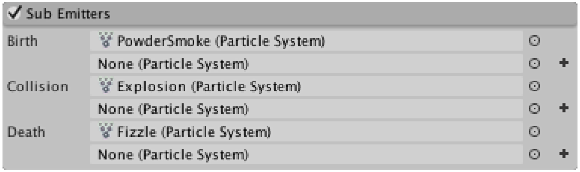
✓ Triggers

Colliders	None (Component)
Inside	Kill
Outside	Ignore
Enter	Ignore
Exit	Ignore
Radius Scale	1
Visualize Bounds	☐

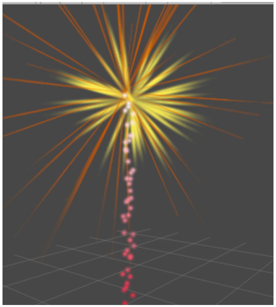
The Particle System Component - Modules (9/12)

Sub-emitters:

Sub Emitters are additional particle systems that are created at a particle's position at certain stages of its lifetime. The sub-emitters are just ordinary particle system objects created in the scene or from prefabs. The stages of the particles' lifetime with these secondary systems can be created are *Birth*, *Collision* and *Death*. Two sub-emitters can be added for each phase using the '+' arrows at the right of the inspector.


✓ Sub Emitters

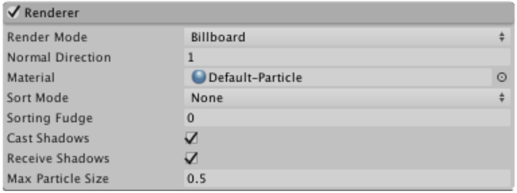
Birth	PowderSmoke (Particle System)
	None (Particle System)
Collision	Explosion (Particle System)
	None (Particle System)
Death	Fizzle (Particle System)
	None (Particle System)



The Particle System Component - Modules (10/12)

Renderer:

This module's settings determine how a particle's image or mesh is transformed, shaded and overdrawn by other particles.



✓ Renderer	
Render Mode	Billboard
Normal Direction	1
Material	Default-Particle
Sort Mode	None
Sorting Fudge	0
Cast Shadows	✓
Receive Shadows	✓
Max Particle Size	0.5

The Render Mode determines how the rendered image is produced from the graphic image:

- **Billboard:** particle always faces the camera.
- **Stretched Billboard:** faces the camera but with scaling applied.
- **Horizontal Billboard:** particle plane is parallel to the XZ "floor" plane.
- **Vertical Billboard:** particle is upright on the world Y axis but turns to face the camera.
- **Mesh:** particle is rendered from a 3D mesh instead of a texture.

The Particle System Component - Modules (11/12)

Renderer common properties:

- **Material:** Material used to render the particle.
- **Sort Mode:** The order in which particles are drawn (and therefore overlaid). The possible values are By Distance (ie, from camera), Youngest First and Oldest First.
- **Sorting Fudge:** Bias of particle sort ordering. Lower values increase the relative chance that particles will be drawn over other transparent objects, including particles from other systems.
- **Cast Shadows**: Should the particle cast shadows on other objects? Only opaque materials cast shadows.
- **Receive Shadows:** Should shadows be cast onto particles? Only opaque materials can receive shadows.
- **Max Particle Size**: The largest particle size (regardless of other settings), expressed as a fraction of viewport size.

The Particle System Component - Modules (12/12)

Stretched Billboard mode - only properties:

- **Camera Scale:** Stretching applied in proportion to camera movement.
- **Speed Scale:** Stretching applied in proportion to a particle's speed.
- **Length Scale:** Stretching applied in proportion to a particle's length.

Billboard modes - only property:

- **Normal Direction:** Bias of lighting normals used for the particle graphics. A value of 1.0 points the normals at the camera, while a value of 0.0 points them towards the centre of the screen.

Mesh mode - only property:

- **Mesh:** One or more meshes used to render particles.

Outline

Lab Preview

We’ll finish the terrain assignment adding a menu, fade-in/out effects, and particle systems.

Next lecture: Persistence and animations.