4
Creating GUIs

In this chapter, we will cover:

- Controlling the scrollbar with the mouse wheel
- Implementing custom mouse cursor icons
- Displaying a digital clock
- Displaying an analogue clock
- Displaying a compass to indicate direction player is facing
- A radar displaying the relative locations of other objects
- Image display for corresponding integers (e.g. hearts for number of lives left)
- Image display for corresponding floats and ranges (e.g. health bar red/green zones)
- Displaying a digital countdown timer
- Displaying a countdown timer graphically (5,4,3,2,1-blast off)
- Displaying a countdown timer graphically as a pie-chart style clock
- Creating a message that fades away
- Single object inventory – text display (key pickup)
- Single object icon inventory – icon display (key pickup)
- A general purpose Inventory PickUp class for inventory item pickup and display

Introduction

One element to the entertainment and enjoyment of most games is the quality of the visual elements, and an important part of this is the GUI (Graphical User Interface). GUI elements involve ways for the user to interact with the game (such as scroll wheels and cursors), and also ways for the game to present up to date information to the user (such as an inventory of what they are carrying, or the location of other objects in the game via a radar screen).

This chapter is filled with GUI recipes to give you a range of examples and ideas of creating game GUIs.

Controlling the scrollbar with the mouse wheel

For many users, there is nothing more natural than moving a scrollbar with the mouse wheel. However, this is not natively supported by Unity. In this recipe, we will add mouse wheel support to a vertical scrollbar**.**

How to do it...

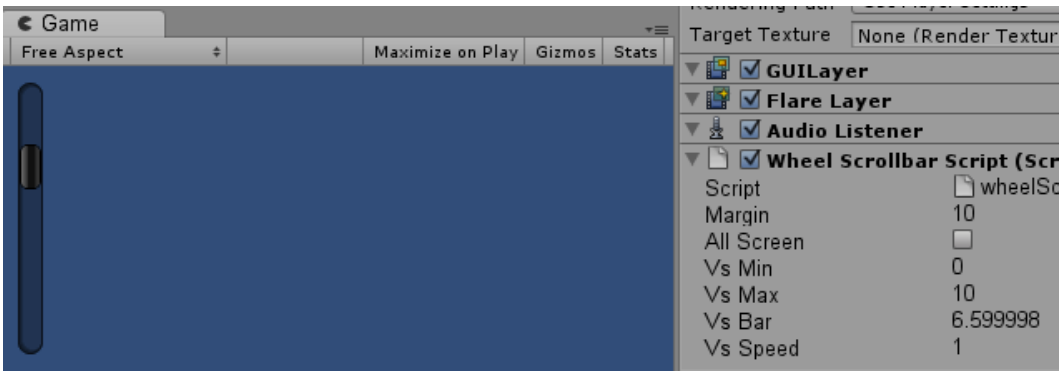
1 - In the Project window, create a new Java Script. Then, rename it *ScrollWheelScript* and open it in your editor.

2 - Add the code below to the top of the script:

```
var margin : int;
var allScreen : boolean = true;
var vsMin : float = 0.0;
var vsMax : float = 10.0;
var vsBar : float = 10.0;
var vsSpeed : float = 5;


function OnGUI () {

   var rect = Rect (margin, margin, 30, Screen.height -
(margin * 2));
   vsBar = GUI.VerticalScrollbar(rect, vsBar, 1.0,vsMax,
vsMin);
   var onArea : boolean;

   if(!allScreen && !rect.Contains(Input.mousePosition)){

   onArea = false;

   } else {

   onArea = true;

   }

var vsMove = Input.GetAxis("Mouse ScrollWheel") * vsSpeed;

if((vsBar + vsMove) > vsMin && onArea){
   vsBar += vsMove;

   }
```

3 - Apply the script to the scene's *Main Camera*.

4 - Test your Scene. You should be able to control the scrollbar with the mouse wheel.



0423_04_01.png

How it works...

Since the scrollbar's handle position is given by a variable (in our case, *vsBar*), all we needed to do was incrementing that variable according to the mouse scroll wheel input.

We have also added a few variables to enhance the scrollbar customization: margin, minimum and maximum values, scroll speed; and also a boolean variable to enable/disable the mouse wheel input when the mouse cursor is not directly on top of the scrollbar**.**

There's more...

In this recipe, the scrollbar height is determined by the application screen's height. However, there's no reason you couldn't add more variables to manually set its size and position.

See also

Implementing custom mouse cursor icons

Cursor icons are often used to indicate the nature of the interaction that can be done with the mouse. Zooming, for instance, might be illustrated by a magnifying glass. Shooting, on the other hand, is usually represented by a stylized target. In this recipe, we will learn how to implement custom mouse cursor icons to better illustrate your

gameplay -- or just to escape Windows and OSX default GUI.

Getting ready

If you need a set of textures to be used as cursor icons, please use the three image files available in the folder 04_02.

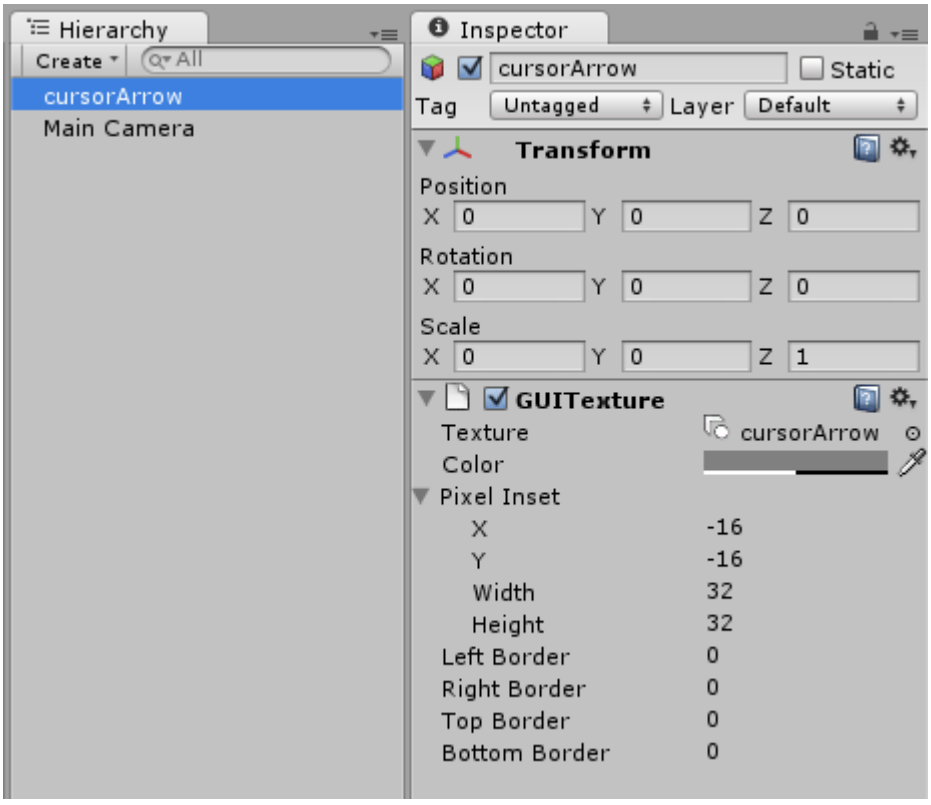
How to do it...

1 - Import the textures into your Unity Project.

2 - In the *Project* window, select the texture *cursorArrow*.

3- Create a new GUI texture through the *menu GameObject > Create Other > GUI texture*. Since you had the texture *cursorArrow* selected, the GUI Texture will use it.

4 – In the Transform component of the *Inspector*, change the GUI Texture's position to x: 0, y: 0, and Z: 0.



0423_04_02.png

5 – Create a new JavaScript and rename it *CursorScript*.

6 – Open the script in your editor and replace everything with the following code:

```
var iconArrow : Texture2D;
var arrowRegPoint : Vector2;
var iconZoom : Texture2D;
var zoomRegPoint : Vector2;
var iconTarget : Texture2D;
var targetRegPoint : Vector2;
private var mouseReg : Vector2;

function Start(){
   guiTexture.enabled = true;
      if(iconArrow){
         guiTexture.texture = iconArrow;
         mouseReg = arrowRegPoint;
         Screen.showCursor = false;
      }
}

function Update(){
   var mouseCoord = Input.mousePosition;
   var mouseTex = guiTexture.texture;
   guiTexture.pixelInset = Rect(mouseCoord.x - (mouseReg.x),
mouseCoord.y - (mouseReg.y),mouseTex.width,mouseTex.height);

   if (Input.GetKey (KeyCode.RightShift) || Input.GetKey
(KeyCode.LeftShift)){
      if(iconTarget){
         guiTexture.texture = iconTarget;
         mouseReg = targetRegPoint;
      }
   } else if(Input.GetMouseButton(1)){
      if(iconZoom){
         guiTexture.texture = iconZoom;
         mouseReg = zoomRegPoint;
      }
   } else {
      if(iconArrow){
         guiTexture.texture = iconArrow;
         mouseReg = arrowRegPoint;
      }
   }
}
```

7 – Save your script and apply it to the GUI Texture you have created by dragging it

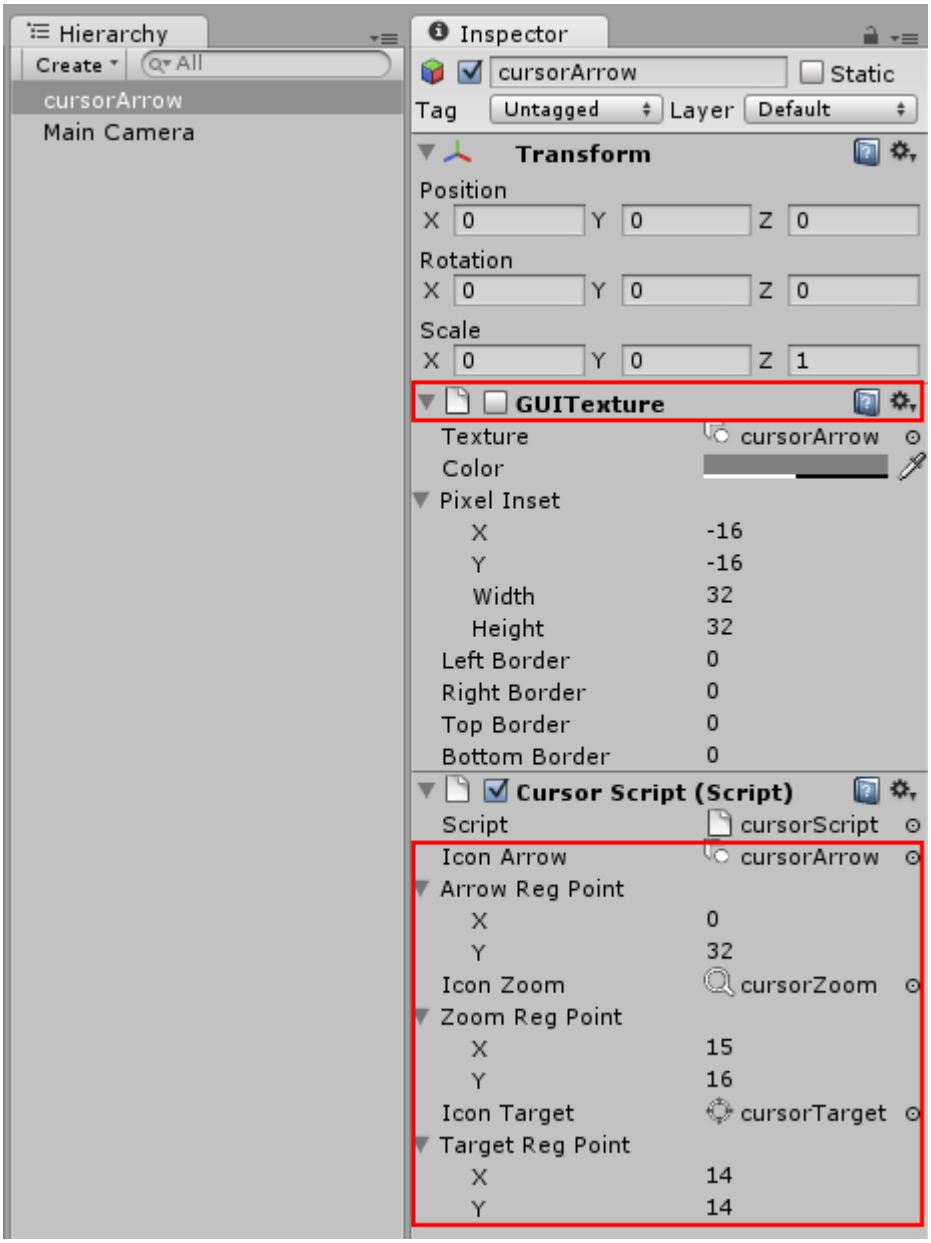
from the *Project* window to the GUI Texture game object in the *Hierarchy* window.

8 – In the *Hierarchy* window, select the GUI Texture and disable its *GUITexture* component.

9 – Drag the texture files *cursorArrow*, *cursorZoom* and *cursorTarget* to the fields Icon Arrow, Icon Zoom and Icon Target, respectively.

10 – Change the *Arrow Reg Point*, *Zoom Reg Point and Target Reg Point* fields to X:0 Y:32, X:15 Y:16, and X:14 Y:14, respectively.

> The *Reg Point* values are actually the pixel coordinates for the focus point of the cursor. When in doubt, open your image editor and check it.



0423_04_03.png

11 – Play your scene. Your OS default cursor should be replaced by the *cursorArrow*, changed to *cursorZoom* when the right mouse button is clicked and to *cursorTarget*

when any Shift key is down.

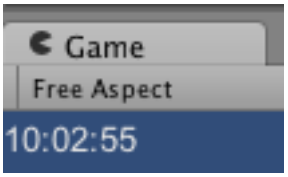
How it works...

The script updates the GUI Texture position based on the mouse cursor position and registration points for each cursor type. The texture map is chosen according to the user's action (pressing the shift key or clicking the right mouse button will change the texture). The script also hides the original mouse cursor, so it won't be on top of your custom one.

See also

Displaying a digital clock

Whether it be the real-world time, or an in-game clock, many games are enhanced by some form of clock or timer display. The most straightforward type of clock to display is a string composed of the integers for hours-minutes-seconds.



0423_04_04.png

Getting ready

How to do it...

1 – Attach the following script class to the Main Camera

```
// file: ClockDigital.cs
using UnityEngine;
using System.Collections;

using System;

public class ClockDigital : MonoBehaviour
{
   void OnGUI ()
   {
```

```
        DateTime time = DateTime.Now;
        string hour = LeadingZero( time.Hour );
        string minute = LeadingZero( time.Minute );
        string second = LeadingZero( time.Second );

        GUILayout.Label( hour + ":" + minute + ":" +  second);

    }


    /**
     * given an integer, return a 2-character string
     * adding a leading zero if required
     */
    string LeadingZero(int n)
    {
        return n.ToString().PadLeft(2, '0');
    }
}
```

How it works...

Importing the 'System' namespace allows the current time to be retrieved from the system from the DateTime class. The first line of the OnGUI() method retrieves the current time and stores it in a DateTime object *time*.

Integers for the hour, minute and second are extracted from the DateTime object, and converted into two-character strings with the LeadingZero() method. This method pads out single digit values with a leading zero if required.

Finally a string concatenating the hour, minute and second strings with a colon separator is displayed via a GUI Label.

There's more...

Some details you don't want to miss:

Converting to a 12-hour clock

If you prefer to display the hours in 12-hour format, this can be calculated by finding the modulo 12 value of the 24-hour clock integer stored in DateTime objects:

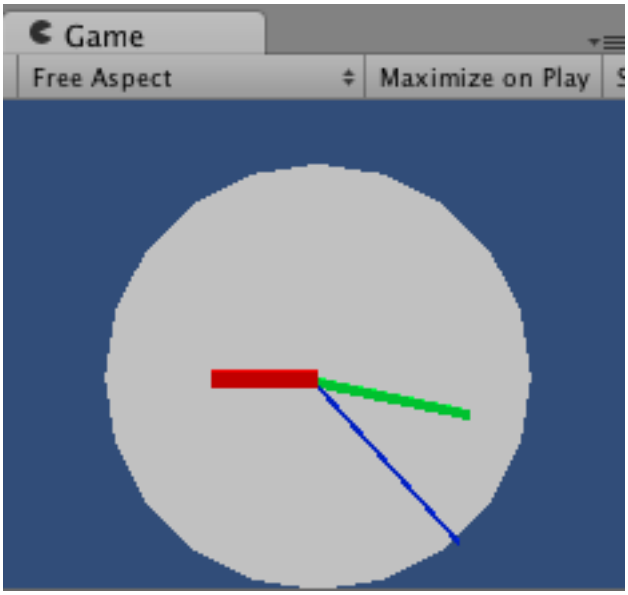
```
    int hour12 = time.Hour % 12;
```

See also

- Displaying an analogue clock

Displaying an analogue clock

Analogue clocks are often more visually appealing, and often worth the little extra effort in creating the GUI elements for the clock-face and hands.



0423_04_06.png

Getting ready

How to do it...

1 – Create a new scene, and add a directional light in the direction of the camera

2 – Create the clock face, this is a Cylinder with the following properties:

- Position (0, 0, 3)
- Rotation (90, 0, 0)
- Scale (13, 0.1, 13)

3 – Create three differently coloured or textured materials (e.g. red, green and blue), to be applied to the hands of the clock. Name these materials *m_seconds, m_minutes* and *m_hours.*

4 – Create a long, thin second hand, with the following properties and with one of

the three coloured materials applied:

- Position (0, 2.5, 0)
- Scale (0.1, 5, 1)

5 – Create a medium long, medium width minute hand, with the following properties and with one of the remaining two coloured materials applied:

- Position (0, 2, 1)
- Scale (0.25, 4, 1)

6 – Create a short, wide hour hand, with the following properties and with the last coloured materials applied:

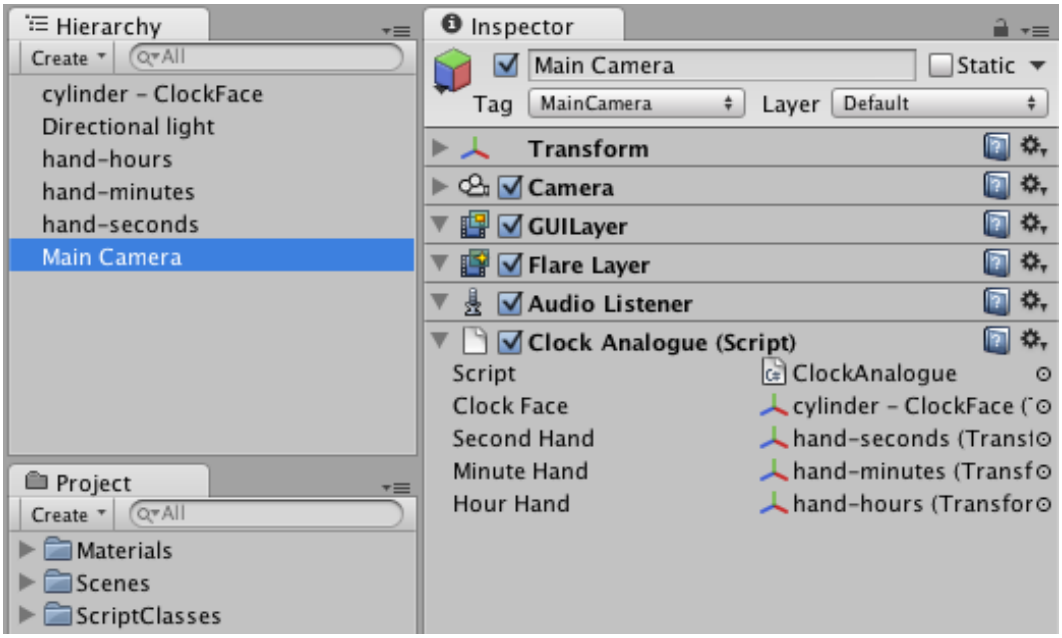
- Position (0, 1.5, 2)
- Scale (0.5, 3, 1)

7 – Attach the following script class to the Main Camera

```
// file: ClockAnalogue.cs
using UnityEngine;
using System.Collections;

using System;

public class ClockAnalogue : MonoBehaviour
{
   public Transform clockFace;
   public Transform secondHand;
   public Transform minuteHand;
   public Transform hourHand;

   Vector3 secondHandStartPosition;
   Vector3 minuteHandStartPosition;
   Vector3 hourHandStartPosition;

   void Start()
   {
      secondHandStartPosition = secondHand.position;
      minuteHandStartPosition = minuteHand.position;
      hourHandStartPosition = hourHand.position;
   }

   void LateUpdate ()
   {
      DateTime time = DateTime.Now;
      float seconds = (float)time.Second;
      float minutes = (float)time.Minute;
      float hours12 = (float)time.Hour % 12;
      float angleDegreesSeconds = -360 * (seconds/60);
```

```
        float angleDegreesMinutes = -360 * (minutes/60);
        float angleDegreesHours = -360 * (hours12 / 12);

        ZeroAllHands();
        RotateHand( secondHand, angleDegreesSeconds);
        RotateHand( minuteHand, angleDegreesMinutes);
        RotateHand( hourHand, angleDegreesHours);
    }

    // move all hands back to 12 o clock
    private void ZeroAllHands()
    {
        secondHand.rotation = Quaternion.identity;
        secondHand.position = secondHandStartPosition;
        minuteHand.rotation = Quaternion.identity;
        minuteHand.position = minuteHandStartPosition;
        hourHand.rotation = Quaternion.identity;
        hourHand.position = hourHandStartPosition;
    }

    private void RotateHand(Transform hand, float angleDegrees)
    {
        hand.RotateAround(
            clockFace.position, Vector3.forward, angleDegrees);
    }
}
```

8 – With the Main Camera selected in the Hierarchy, drag the clock face and the three clock hands into the Inspector for the four public Transform variables:



0423_04_05.png

How it works...

The clock-face is a very thin Cylinder object, and the three hands are stretched cubes. Each object at a different Z-axis position, so they are seen as in font of each other – the clock-face at the back, then the fat hour hand, then the middle sized minute hand, and closed to the viewer is the thin long second hand. Each LateUpdate() event all 3 hands are returned to their initial 12-o-clock position, and then rotated about the clock-face center by an angle proportionate to the seconds/minutes/hours of the DateTime object.

Method *RotateHand*() is called for each hand, which uses the *Transform.RotateAround*() method, rotating a GameObject about a point (the clock face center), for a given axis (Vector3.forward – the Z-axis the viewer in looking along), by a given angle in degrees (a proportion of 360 calculated repeatedly for each of the 3 hands). The *Start*() method caches the origininal (12-o-clock) position of the three hands when the game begins, so that each frame the hands can be returned to their (zero-degrees) starting position before being rotated the appropriate amount – this approach means that the clock is constantly repositioning itself to the accurate DateTime object, so differences in frame rates or small inaccuracies in co-routine timings do not resulting in cumulative errors.

Due to the LHS (left-hand-system) arrangement of the Z-axis in Unity, the rotation angles have to be negative – hence the negative 360 multiplied each time for the

calculations for each clock hand angle in degrees.

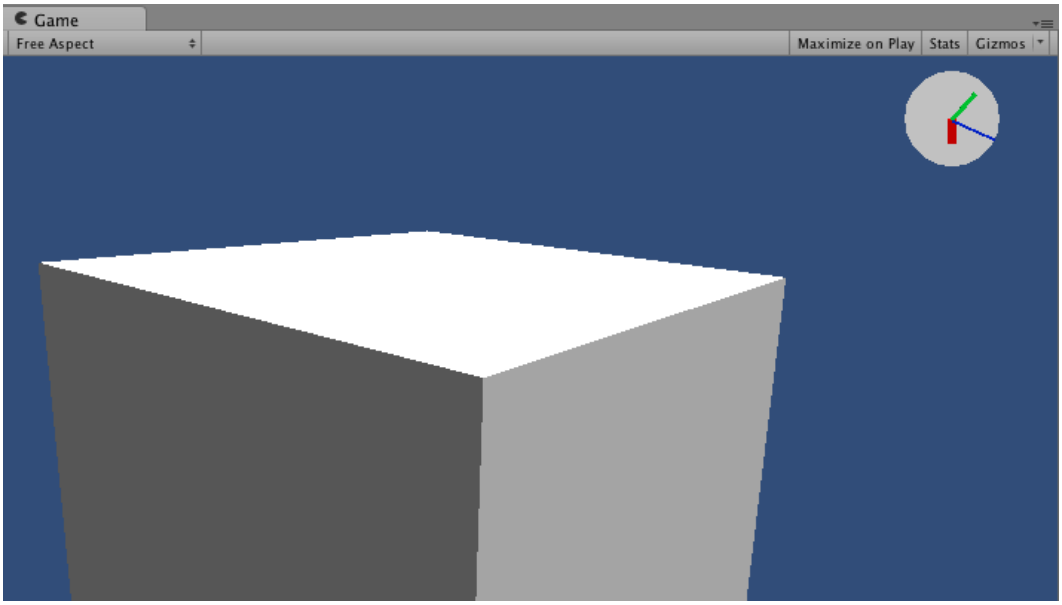
Each clock-hand has its original position arranged as the center of the clock face, and then half the length of the hand added to the Y-axis co-ordinate. For example, the clock face has a center of (0,0,0), and the second hand is a cube with a Y-scale of 5, so its center is (0, 2.5, 0). This means that each hand when rotated around the center of the clock-face moves correctly. The width of the hands is determined by the X-axis scaling, so the second hand is thin (X-s axis scale 0.1) and long (Y-axis scale 5), while the hour hand is wide (X-axis scale 0.5) and short (Y-axis scale 3). The wider short hands are further from the user (larger Z-axis positions), and the thinner long hands are nearer the user – so that a wide hour hand can be seen behind the thinner minute and second hands even when pointing to the same clock-face position.

There's more...

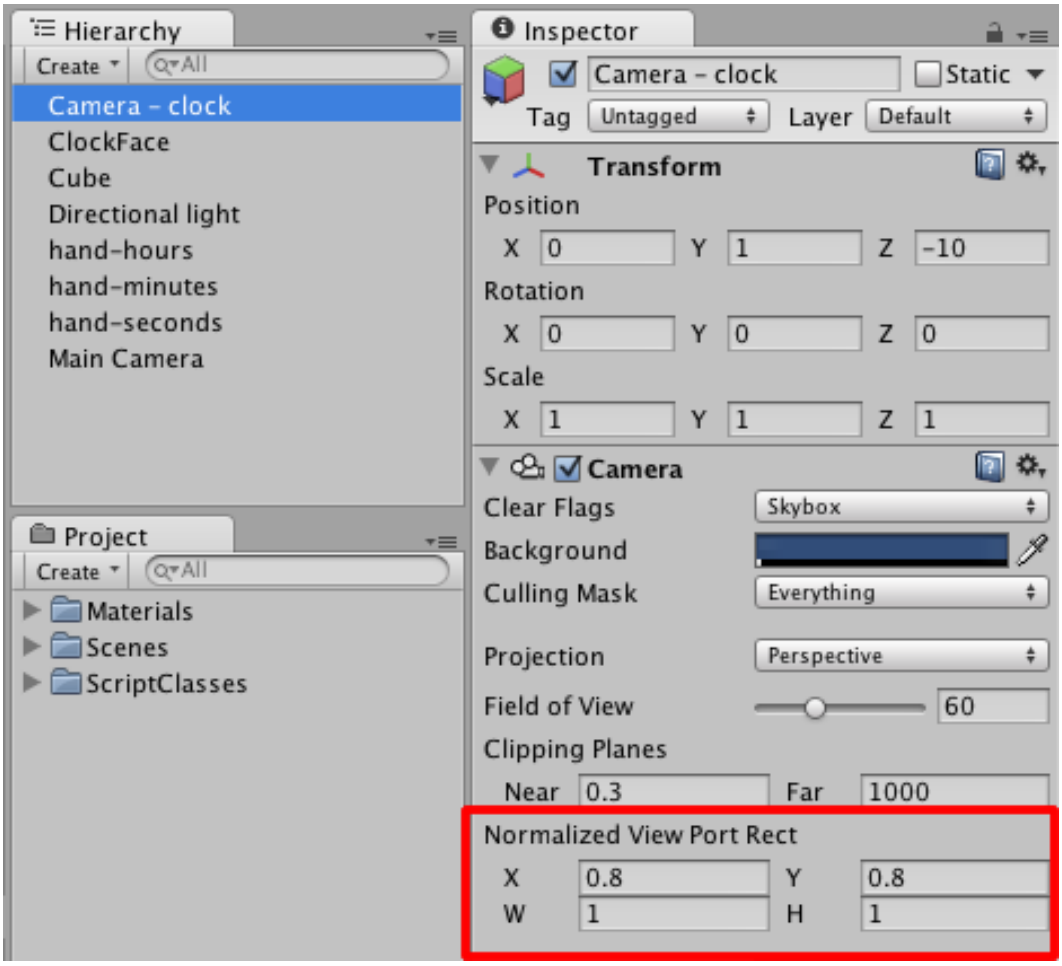
Some details you don't want to miss:

Clock taking up a small part of the screen

Usually a clock does not take up the whole screen, and one method of having a view of a GameObject only take up part of the screen is by having a second camera. These screenshots show a scene where the Main Camera shows a cube, but a second camera (*Camera – clock*) has been created to display in the top right of the screen. By default cameras display to a 'normalised viewport' of the whole screen (0,0)-(1,1). However, by setting the rectangle of a camera's normalised viewport to less than the whole screen, the desired effect can be created we show. Our second camera *Camera – clock* has a viewport rectangle of (0.8,0.8)-(1,1) which defines the 20% top right of the screen as we can see.



0423_04_07.png



0423_04_08.png

See also

- Displaying a digital clock

Displaying a compass to indicate direction player is facing

In games where players must refer to maps to confidently navigate large terrains, it can be very useful to display a compass in the GUI, presenting a real-time graphical

indication of the direction the player is facing. The screenshot shows the player's 3rd person controller looking North-West, in-between the cube and sphere. The black 'blip' circle on the compass indicates that the player is facing this compass direction.

0423_04_10.png

Getting ready

If you need a set of images for this recipe, you'll find two image files in the folder 04_05. One is a background image of a compass circle, and the other is a black circle image to represent the direction the player is facing.

How to do it...

1 – Create a new scene, and add a directional light

2 – Create a new Terrain

3 – Import the built-in Unity Package *Character Controller*

4 – Add a 3rd person character controller to your scene at position (0, 1, 0)

5 – Create a cube in front (NORTH) of the 3rd person controller at (0, 1, 5)

6 – Create a sphere to the left (WEST) of the 3rd person controller at (-5, 1, 0)

7 – Attach the following script class to the Main Camera

```
// Compass.cs
using UnityEngine;
using System.Collections;
```

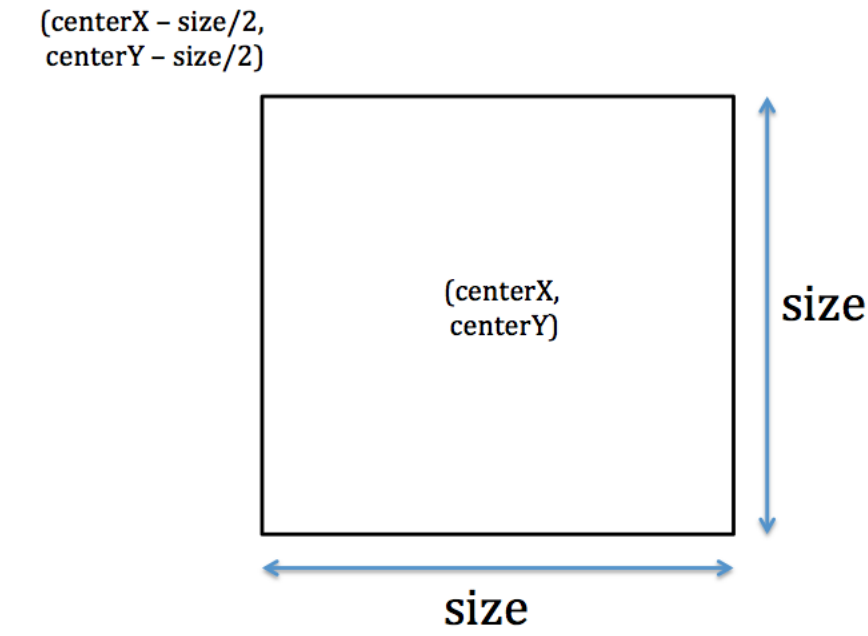
```
public class Compass : MonoBehaviour
{
   public Transform playerController;
   public Texture compassBackground;
   public Texture playerBlip;

   private void OnGUI()
   {
      // background displaying top left in square of 128
pixels
      Rect compassBackgroundRect = new Rect(0,0, 128, 128);

   GUI.DrawTexture(compassBackgroundRect,compassBackground);
      GUI.DrawTexture(CalcPlayerBlipTextureRect(),
playerBlip);
   }

   private Rect CalcPlayerBlipTextureRect()
   {
      // subtract 90, so North (0 degrees) is UP
      float angleDegrees = playerController.eulerAngles.y -
90;
      float angleRadians = angleDegrees * Mathf.Deg2Rad;

      // calculate (x,y) position given angle
      // blip distance from center is 16 pixels
      float blipX = 16 * Mathf.Cos(angleRadians);
      float blipY = 16 * Mathf.Sin(angleRadians);

      // offset blip position relative to compass center
(64,64)
      blipX += 64;
      blipY += 64;

      // stetch blip image to display in 10x10 pixel square
      return new Rect(blipX - 5, blipY - 5, 10, 10);
   }
}
```

8 – With the Main Camera selected in the Hierarchy, drag the 3rd person controller, the compass background image, and the image for the player's direction 'blip' into the Inspector for the three public variables.

How it works...

The Compass class needs 3 public variables, the first is a reference to the player's 3rd person controller, the other 2 are images for the compass background image (usually a circle of some kind with the compass letters displayed) and another image to indicate which direction the player is facings on the compass background.

The OnGUI() method is called every frame, and each frame the compass background needs to be displayed, and then the image indicating the player's direction. The position of that 'blip' image is calculated and returned by the method *CalcPlayerBlipTextureRect*(). Both images are displayed using the DrawTexture() method of the GUI class, which expects a Rect and the texture object. Many recipes result in us knowing the center co-ordinates of a square or rectangle, and its size/width/height. Therefore (as the figure shows) such calculations in this recipe are straightforward to calculate.




```
Rect textureRect = new Rect(centerX – size/2, centerY – size/2, size, size);
GUI.DrawTexture(textureRect, texture);
```

0423_04_09.png

This recipe makes use of the 'yaw' angle of rotation, which is rotation about the Y-axis – i.e. the direction a character controller is facing. This can be found in the 'y' component of a GameObject's eulerAngles component of its transform. You can imagine looking from above, down at the character controller, and seeing what direction they are facing – this is just what we are trying to display graphically with the compass. In mathematics, an angle of zero indicates an 'east' direction, to correct that we need to subtract 90 degrees from the yaw Euler angle. The angle is then converted into radians, since that is required for the Unity trigonometry methods. We then multiply these Sin() and Cos()

results by the distance we want the blip to be displayed form the center of the compass circle (in our code example this is 16 pixels). We add 64 pixels to these (x,y) co-ordinates results since that is the center of the compass background image. These final values for (blipX, blipY) are the position on screen we wish the center of player's directional blip image to be displayed. Knowing the center of the blip image, and its width and height (both 10 pixels) allows the Rect to be created for use by the DrawTexture() method called from OnGUI().

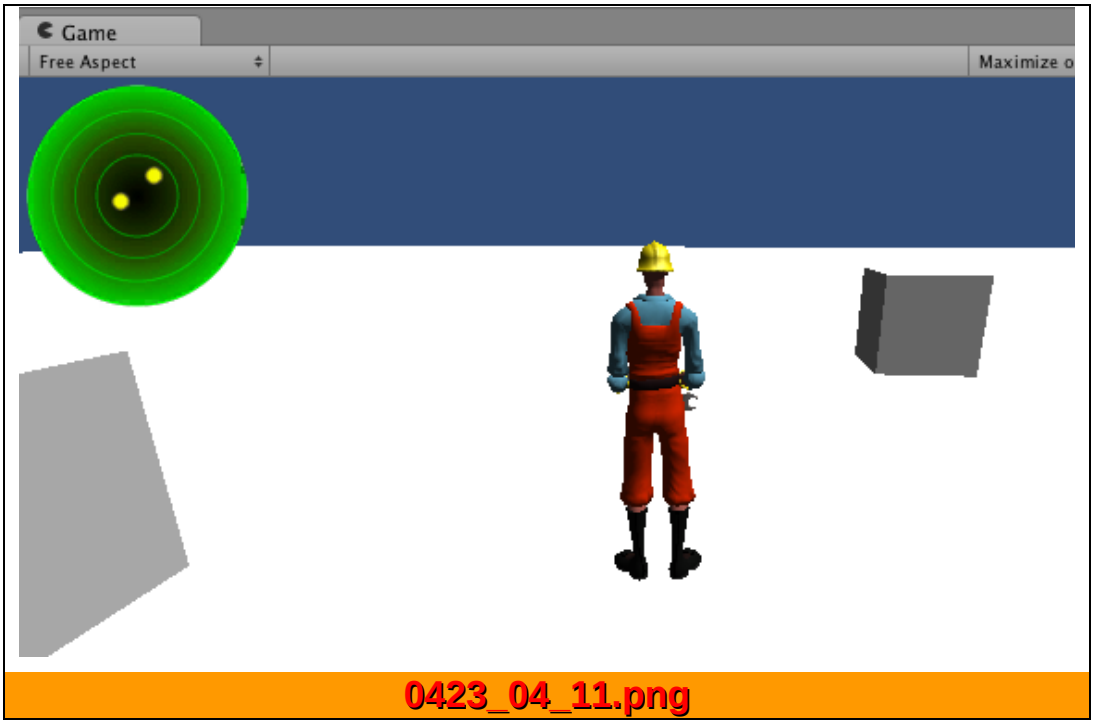
See also

- A radar displaying the relative locations of other objects

A radar displaying the relative locations of other objects

A radar displays the locations of other objects relative to the player, usually based on a circular display, where the center represents the player, and each graphical 'blip' indicates how far away, and in what relative direction the object is to the player. Sophisticated radar displays will display different categories of object with different coloured 'blip' icons.

In the screenshot we can see 2 yellow circle 'blips', indicating the relative position of the 2 objects tagged 'cube' near the player. The green circle radar background image gives the impression of an aircraft control tower radar.



0423_04_11.png

Getting ready

If you need a set of images for this recipe, you'll find two image files in the folder 04_06. One is a background image of green radar circles, and the other is a yellow circle image to represent objects on the radar.

How to do it...

1 – Create a new scene, and add a directional light

2 – Create a new Terrain

3 – Import the built-in Unity Package *Character Controller*

4 – Add a 3rd person character controller to your scene at position (0, 1, 0)

5 – Create a cube in front of the 3rd person controller at (0, 1, 5), and give this the tag 'cube'

6 – Create a second cube to the left of the 3rd person controller at (-5, 1, 0) and give this the tag 'cube'

7 – Attach the following script class to the Main Camera

```
// radar.cs
using UnityEngine;
using System.Collections;

public class Radar : MonoBehaviour
{
   const float MAX_DISTANCE = 20f;
   const int RADAR_SIZE = 128;

   public Transform playerController;
   public Texture radarBackground;
   public Texture targetBlip;

   private void OnGUI()
   {
      // background displaying top left in square of 128
pixels
      Rect radarBackgroundRect = new Rect(0,0, RADAR_SIZE,
RADAR_SIZE);
      GUI.DrawTexture(radarBackgroundRect,radarBackground);

      // find all 'cube' tagged objects
      GameObject[] cubeGOArray =
GameObject.FindGameObjectsWithTag("cube");

      // draw blips for all within distance
      Vector3 playerPos = playerController.transform.position;
      foreach (GameObject cubeGO in cubeGOArray)
      {
         Vector3 targetPos = cubeGO.transform.position;
         float distanceToTarget =
Vector3.Distance(targetPos,playerPos);
         if( (distanceToTarget <= MAX_DISTANCE) )
            DrawBlip(playerPos, targetPos, distanceToTarget);
      }
   }

   private void DrawBlip(Vector3 playerPos, Vector3 targetPos,
float distanceToTarget)
   {
      // distance from target to player
      float dx =  targetPos.x - playerPos.x;
      float dz =  targetPos.z - playerPos.z;

      // find angle from player to target
      float angleToTarget = Mathf.Atan2(dx,dz) *
Mathf.Rad2Deg;

      // direction player facing
      float anglePlayer = playerController.eulerAngles.y;
```

```
      // subtract player angle, to get relative angle to
object
      // subtract 90
      // (so 0 degrees (same direction as player) is UP)
      float angleRadarDegrees =  angleToTarget - anglePlayer -
90;

      // calculate (x,y) position given angle and distance
      float normalisedDistance = distanceToTarget /
MAX_DISTANCE;
      float angleRadians = angleRadarDegrees * Mathf.Deg2Rad;
      float blipX = normalisedDistance *
Mathf.Cos(angleRadians);
      float blipY = normalisedDistance *
Mathf.Sin(angleRadians);

      // scale blip position according to radar size
      blipX *= RADAR_SIZE/2;
      blipY *= RADAR_SIZE/2;

      // offset blip position relative to radar center (64,64)
      blipX += RADAR_SIZE/2;
      blipY += RADAR_SIZE/2;

      // draw target texture at calculated location
      Rect blipRect = new Rect(blipX - 5, blipY - 5, 10, 10);
      GUI.DrawTexture(blipRect, targetBlip);
   }
}
```

8 – With the Main Camera selected in the Hierarchy, drag the 3rd person controller, the radar background image, and the image for the cubes yellow 'blip' into the Inspector for the three public variables.

How it works...

Two constants are defined, *MAX_DISTANCE* to specify the maximum distance (in Unity 'units') objects are to be detected, and *RADAR_SIZE* to specify the size (in pixels) of the top left square the radar will occupy. The Radar class needs 3 public variables, the first is a reference to the player's 3rd person controller, the other 2 are images for the radar background image (usually a circle of some kind) and another image to indicate object locations on the radar background.

The OnGUI() method first displays the radar background image. An array of GameObjects with the desired tag is retrieved, and iterated through. For each target GameObject, if its distance to the character controller is within *MAX_DISTANCE* then method DrawBlip() is called, with arguments of the Vector3 positions of the player's

character controller and the target GameObject.

Method DrawBlip() finds the X and Z distances between target and player, and uses them to calculate the angle from the target to the player (using Unity's Atan2() inverse trigonometric function). The direction the player's character is facing is retrieved from its Y-axis Eurler angle (just as in the compass recipe). The player's direction angle is subtracted from the angle between target and player, since a radar displays the **relative** angle from the direction the player is facing, to the target object. As usual, 90 degrees need to be subtracted from the final angle, since we want zero degrees to be displayed as upwards in our GUI.

A *noramlisedDistance* is calculated, which will always range between 0 and 1, by dividing the distance in pixels of the target from the player, by *MAX_DISTANCE*. The angle is converted into radians, since that is required for the Unity trigonometry methods. We then multiply these Sin() and Cos() results by the distance we want the blip to be displayed form the center of the radar circle – i.e. *noramlisedDistance*. These coordinates are then converted to display in the top left of the screen in a square of *RADAR_SIZE* pixels. Knowing the center of the blip image, and its width and height (both 10 pixels) allows the Rect to be created for use by the DrawTexture() method called from OnGUI().

There's more...

Some details you don't want to miss:

Different coloured 'blips' for different objects

Adding more public Texture variable, allows for different icons to represent different categories of object on the radar. A Texture parameter can be added to the DrawBlip() method declaration, and that texture can be used in the GUI.DrawTexture() statement. This allows DrawBlip() to be called from different loops, each for objects tagged with a different string.

See also

- Displaying a compass to indicate direction player is facing

Image display for corresponding integers (e.g. hearts for number of lives left)

While the first prototype of a user interface often simply displays game parameters as

strings, engaging interfaces usually require customized graphical representations. A straightforward technique is to display an icon for each value of an integer – e.g. a stickman or heart or ship for each life left, a bullet or magazine for ammo etc.

Getting ready

in folder 04_07you'll find some you'll find an image of a stick man.

How to do it...

1 – Attach the following script class to the Main Camera

```
// file: PlayerGUI.cs
using UnityEngine;
using System.Collections;

public class PlayerGUI : MonoBehaviour
{
   public Texture heartImage;
   private int livesLeft = 5;

   private void OnGUI()
   {
      Rect r = new Rect(0,0,Screen.width, Screen.height);
      GUILayout.BeginArea(r);
      GUILayout.BeginHorizontal();

      ImagesForInteger(livesLeft, heartImage);

      GUILayout.FlexibleSpace();
      GUILayout.EndHorizontal();
      GUILayout.EndArea();
   }

   private void ImagesForInteger(int total, Texture icon)
   {
      for(int i=0; i < total; i++)
      {
         GUILayout.Label(icon);
      }
   }
}
```

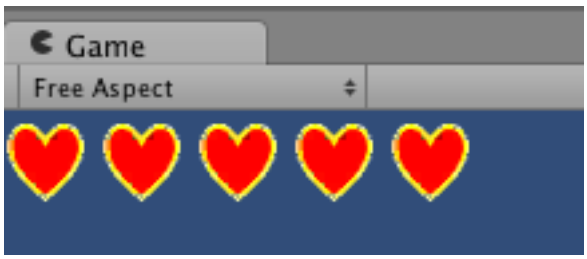
2 – With the Main Camera selected in the Hierarchy, drag the heart image into the Inspector for the public variable.

How it works...

A public Texture variable will store a heart image. A private *livesLeft* integer variable stores the game parameter to be graphically represented.

The OnGUI() method sets up a basic GUILayout Area that will display contents horizontally, and left aligned (via an Area, Begin-End Horizontal, and a FlexibleSpace after the contents are displayed). A call is made to method ImagesForInteger(), passing in the integer *livesLeft* and the heart Texture.

Method ImagesForInteger() is a straightforward 'for' loop, that repeatedly displays the provided image argument (via GUILayout.Label()) as many times as defined by the integer total argument.



0423_04_12.png

There's more...

Some details you don't want to miss:

GUILayout vs. GUI

You may prefer to use GUI rather than GUILayout – this saves having to set up an Area and Horizontal and FlexibleSpace calls, however, it requires maintaining a horizontal pixel position as part of the loop, making the ImagesForInteger() a little more complicated.

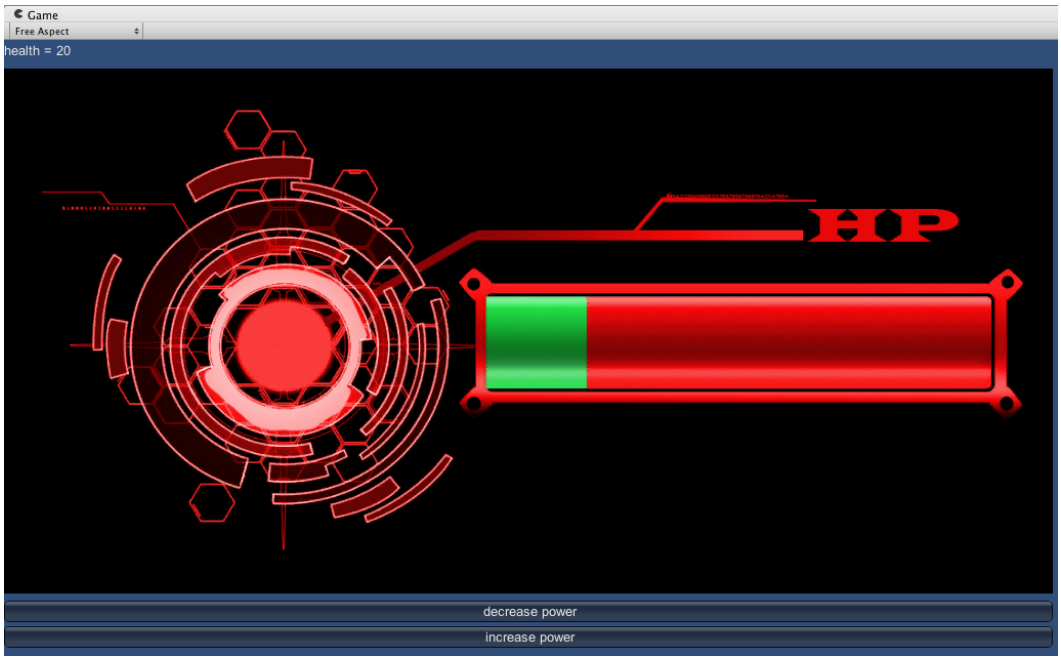
See also

- Image display for corresponding floats and ranges (e.g. health bar red/green zones)

Image display for corresponding floats and ranges (e.g. health bar red/green zones)

As the previous recipe illustrates, displaying images for integers with one-to-one

correspondence is straightforward. However, sometimes there may be a small number of images, to help highlight the 'zone' of a numeric variable. Examples might include health points, with green (very healthy) and red (little health remaining). Whether values are represented as floats (e.g. 0.0 – 1.0) or ranges within larger numbers (e.g. 0 – 50), a general solution is to identify the sub-range (zone) of the numeric value and display the appropriate corresponding image.



0423_04_13.png

Getting ready

in folder 04_08 you'll find some you'll find a series of PNG images.

How to do it...

1 – Attach the following script class to the Main Camera

```
// file: HealthBar.cs
using UnityEngine;
using System.Collections;

public class HealthBar : MonoBehaviour
{
    const int MAX_HEALTH = 100;
```

```
   public Texture2D bar00;
   public Texture2D bar10;
   public Texture2D bar20;
   public Texture2D bar30;
   public Texture2D bar40;
   public Texture2D bar50;
   public Texture2D bar60;
   public Texture2D bar70;
   public Texture2D bar80;
   public Texture2D bar90;
   public Texture2D bar100;

   private int healthPoints = MAX_HEALTH;

   private void OnGUI()
   {
      GUILayout.Label("health = " + healthPoints);
      float normalisedHealth = (float)healthPoints /
MAX_HEALTH;
      GUILayout.Label( HealthBarImage(normalisedHealth) );

      bool decButtonClicked = GUILayout.Button("decrease
power");
      bool incButtonClicked = GUILayout.Button("increase
power");

      if( decButtonClicked )
         healthPoints -= 5;

      if( incButtonClicked )
         healthPoints += 5;
    }

   private Texture2D HealthBarImage(float health)
   {
      if( health > 0.9 ){  return bar100; }
      else if( health > 0.8 ){   return bar90;  }
      else if( health > 0.7 ){   return bar80;  }
      else if( health > 0.6 ){   return bar70;  }
      else if( health > 0.5 ){   return bar60;  }
      else if( health > 0.4 ){   return bar50;  }
      else if( health > 0.3 ){   return bar40;  }
      else if( health > 0.2 ){   return bar30;  }
      else if( health > 0.1 ){   return bar20;  }
      else if( health > 0 ){  return bar10;  }
      else{ return bar00;  }
   }
}
```

2 – With the Main Camera selected in the Hierarchy, drag each of the images into the Inspector for the corresponding public variable.

How it works...

A private *healthPoints* integer variable stores the game parameter to be graphically represented. A constant *MAX_HEALTH* defines the value of *healthPoints* that indicates full health. The 11 public image variables store different versions of the health bar image, the correspond to different health ranges – for example bar00 corresponds to zero health remaining, so is an image with warning red colouring.

The core is this recipe is that a method is called that returns a Texture image object corresponding to the current value of *healthPoints*. A normalised value is created (which will always range between 0.0 and 1.0) by dividing *healthPoints* by *MAX_HEALTH*. This normalised float is passed to method HealthBarImage() which returns an image; the image is displayed via a Label() method call.

Method HealthBarImage () uses a sequence of 'if' statements to decide which image to return. The provided health value is tested in a sequence of decreasing boundary values – 0.9 tested first, then 0.8 and so on. Ranges are implied by these values – so if the normalised health value is from 0.9 to 1.0, the image in variable bar100 will be displayed.

There's more...

By **normalising** values into the range 0.0 – 1.0, methods can be reused in many different circumstances. So whether the health points for one game are from 0 – 50, or 0 – 10, or 0 – 500, the same code can be used, by always normalising values by dividing them by their maximum value to get the range 0.0 – 1.0.

If ranges corresponding to images are always the same size, then an even more general purpose method could be created that takes as input a normalised value, and an array of images, and determines the ranges depending on the size of the array.

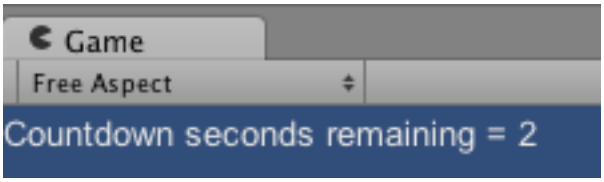
See also

- Image display for corresponding integers (e.g. hearts for number of lives left)
- Displaying a countdown timer graphically (5,4,3,2,1-blast off)
- Displaying a countdown timer graphically as a pie-chart style clock

Displaying a digital countdown timer

Many games involve players completing tasks or getting bonus points within a set time.

This recipe shows how to create a basic countdown timer.



0423_04_14.png

Getting ready

How to do it...

1 – Attach the following script class to the Main Camera

```
// file: CountdownTimer.cs
using UnityEngine;
using System.Collections;

public class CountdownTimer : MonoBehaviour {
   private int countdownTimerDelay;
   private float countdownTimerStartTime;

   void Awake(){
      CountdownTimerReset( 3 );
   }

   void OnGUI(){
      int secondsLeft = CountdownTimerSecondsRemaining();

      if( secondsLeft > -1)
         GUILayout.Label("Countdown seconds remaining = " +
secondsLeft );
      else
         GUILayout.Label("countdown has finished");
   }

   void CountdownTimerReset(int delayInSeconds){
      countdownTimerDelay = delayInSeconds;
      countdownTimerStartTime = Time.time;
   }

   int CountdownTimerSecondsRemaining(){
      int elapsedSeconds = (int)(Time.time -
countdownTimerStartTime);
      int secondsLeft = (countdownTimerDelay -
elapsedSeconds);
      return secondsLeft;
   }
```

```
}
```

2 – With the Main Camera selected in the Hierarchy, drag each of the images into the Inspector for the corresponding public variable.

How it works...

Time.time returns the number of seconds since the application started as a float. By storing the time that the timer was started in *countdownTimerStartTime*, and comparing it with Time.time, the number of seconds since the timer started can be calculated.

Method Awake() resets the timer with a total of 3, by calling method CountdownTimerReset(). This method stores the current time into instance variable *countdownTimerStartTime*, and sets the countdown timer total delay to the provided argument, by storing it in instance variable *countdownTimerDelay*.

Method OnGUI() checks the seconds remaining for the timer variables by assigning *secondsLeft* to the value returned by method CountdownTimerSecondsRemaining(). If seconds left is zero or more, the number of seconds remaining is displayed, otherwise a message stating that the timer has finished is displayed.

Method CountdownTimerSecondsRemaining() finds the elapsed seconds since the timer started, and subtracts this from the countdown timer total delay in instance variable *countdownTimerDelay*, to calculate and return the number of seconds remaining in the countdown.

There's more...

Some details you don't want to miss:

A count-up timer …

If you want the player to see seconds counting upwards, just rename method CountdownTimerSecondsRemaining() to something like TimerSeconds() and return the value of *elapsedSeconds* instead of *secondsLeft*.

See also

- Displaying a countdown timer graphically (5,4,3,2,1-blast off)

Displaying a countdown timer graphically (5,4,3,2,1-

blast off)

Timers can have more effect when displayed graphically. This recipe illustrates a simple way to display a rocketship style countdown, using images for each second (5, 4, 3, 2, 1) and a final image for 'blast off'.

0423_04_15.png

Getting ready

If you need a set of images for this recipe, you'll find a series of PNG images in the folder 04_10

How to do it...

1 – Attach the following script class to the Main Camera

```
// file: CountdownGaphical.cs
using UnityEngine;
using System.Collections;

public class CountdownGaphical : MonoBehaviour {
   public Texture2D imageDigit1;
   public Texture2D imageDigit2;
   public Texture2D imageDigit3;
   public Texture2D imageDigit4;
   public Texture2D imageDigit5;
   public Texture2D imageBlastOffText;

   private int countdownTimerDelay;
   private float countdownTimerStartTime;

   void Awake(){
      CountdownTimerReset( 5 );
```

```
    }

    void OnGUI(){
        GUILayout.Label( CountdownTimerImage() );
    }

    void CountdownTimerReset(int delayInSeconds){
        countdownTimerDelay = delayInSeconds;
        countdownTimerStartTime = Time.time;
    }

    int CountdownTimerSecondsRemaining(){
        int elapsedSeconds = (int)(Time.time -
countdownTimerStartTime);
        int secondsLeft = (countdownTimerDelay -
elapsedSeconds);
        return secondsLeft;
    }

    Texture2D CountdownTimerImage(){
        switch( CountdownTimerSecondsRemaining() ){
        case 5:
            return imageDigit5;
        case 4:
            return imageDigit4;
        case 3:
            return imageDigit3;
        case 2:
            return imageDigit2;
        case 1:
            return imageDigit1;
        default:
            return imageBlastOffText;
        }
    }
}
```

2 – With the Main Camera selected in the Hierarchy, drag each of the images into the Inspector for the corresponding public variable.

How it works...

This recipe builds on the previous one. The countdown timer algorithm is just the same. In this recipe the OnGUI() method displays a Label with the image returned from method CountdownTimerImage(). This method uses a 'switch' statement to return the corresponding image to the current number of seconds remaining. Six images are public variables, and so can be set via the inspector.

See also

- Displaying a digital countdown timer
- Image display for corresponding floats and ranges (e.g. health bar red/green zones)

Displaying a countdown timer graphically as a pie-chart style clock

Another way to display a timer graphically is as a kind of clock, so the player can see the time running out. This example shows elapsed time as the red portion of the circle…

0423_04_16.png

Getting ready

in folder 04_11 you'll find some you'll find a series of pie-chart PNG images.

How to do it...

1 – Attach the following script class to the Main Camera

```
// file: CoundownClock.cs
using UnityEngine;
using System.Collections;

public class CoundownClock : MonoBehaviour
{
   public int timerTotalSeconds = 5;

   public Texture2D time0;
   public Texture2D time10;
   public Texture2D time20;
   public Texture2D time30;
   public Texture2D time40;
   public Texture2D time50;
   public Texture2D time60;
```

```
   public Texture2D time70;
   public Texture2D time80;
   public Texture2D time90;
   public Texture2D time100;

   private int countdownTimerDelay;
   private float countdownTimerStartTime;

   private void Awake(){
      CountdownTimerReset( timerTotalSeconds );
   }

   private void CountdownTimerReset(int delayInSeconds){
      countdownTimerDelay = delayInSeconds;
      countdownTimerStartTime = Time.time;
   }

   private void OnGUI(){
      float propertionRemaining = (CountdownSecondsLeftFloat()
/ timerTotalSeconds);
      GUILayout.Label( TimeRemainingImage(propertionRemaining)
);
    }

   private float CountdownSecondsLeftFloat(){
      float elapsedSeconds = Time.time -
countdownTimerStartTime;
      float secondsLeft = countdownTimerDelay -
elapsedSeconds;
      return secondsLeft;
   }

   private Texture2D TimeRemainingImage(float
propertionRemaining)
   {
      if( propertionRemaining > 0.9 ){ return time100;   }
      else if( propertionRemaining > 0.8 ){  return time90; }
      else if( propertionRemaining > 0.7 ){  return time80; }
      else if( propertionRemaining > 0.6 ){  return time70; }
      else if( propertionRemaining > 0.5 ){  return time60; }
      else if( propertionRemaining > 0.4 ){  return time50; }
      else if( propertionRemaining > 0.3 ){  return time40; }
      else if( propertionRemaining > 0.2 ){  return time30; }
      else if( propertionRemaining > 0.1 ){  return time20; }
      else if( propertionRemaining > 0 ){ return time10; }
      else{ return time0;  }
   }
}
```

2 – With the Main Camera selected in the Hierarchy, drag each of the images into the Inspector for the corresponding public variable.

How it works...

This recipe is a combination of the basic countdown timer, and the display of images for floats and ranges. Key aspects of this recipe include:

- The total number of seconds needs to be known by OnGUI(), so that a normalised proportion of time remaining can be calculated (in the range 0.0 – 1.0)
- The seconds left needs to be returned as a float, so the clock will work fine for short time intervals or long ones, so the method CountdownSecondsLeftFloat() returns the remaining seconds as a float rather than int as in the previous recipes
- Method TimeRemainingImage() takes in a float argument in the range 0.0 – 1.0, and uses 'if' statements to return the image corresponding to the proportion of the total seconds remaining in the countdown

There's more...

Some details you don't want to miss:

Creating your own images with a spreadsheet

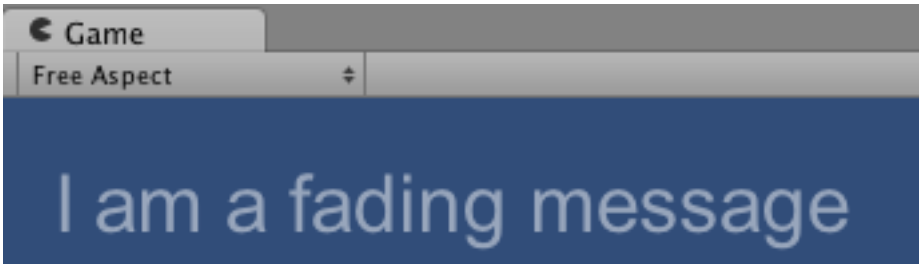
The pie-chart clock style images were created using a spreadsheet, and just changing the two values. Stacked bar charts are another useful spreadsheet graph for timer or health graphical images…

See also

- Displaying a digital countdown timer
- Displaying a countdown timer graphically (5,4,3,2,1-blast off)
- Image display for corresponding floats and ranges (e.g. health bar red/green zones)

Creating a message that fades away

Often we want to show the user a message briefly, and then have that message disappear. A nice way to do this is to have the message fade away, and they Destroy() itself when is it no longer visible …



0423_04_17.png

Getting ready

How to do it...

1 – Create a new GUIText object, position it and change the font size and text as desired

2 – Attach the following script class to the this GUIText object

```
// file: FadingMessage
using UnityEngine;
using System.Collections;

public class FadingMessage : MonoBehaviour {
   const int TOTAL_FRAMES = 100;

   private float alpha = 1.0f;
   private float alphaStep = 1.0f / TOTAL_FRAMES;
   private float deathAlpha;
   private float r;
   private float g;
   private float b;

   private void Awake() {
      deathAlpha = 2 * alphaStep;;
      Color startColor = guiText.material.color;
      alpha = startColor.a;

      r = startColor.r;
      g = startColor.g;
      b = startColor.b;
   }

   private void Update() {
      alpha -= alphaStep;

      if( alpha < deathAlpha)
         Destroy(gameObject);
```

```
        Color newColor = new Color(r, g, b, alpha);
        guiText.material.color = newColor;
    }
}
```

How it works...

The constant *TOTAL_FRAMES* defines the number of frames the image is to be visible for. By diving 1 by this value, the amount to decrement the GUText's alpha by can be calculated, which is stored in variable *alphaStep*. To overcome possible floating point errors the variable *deathAlpha* is set to be twice this amount in the Awake() method.

The Color component of the Material of a GUIText object is made up of 4 components: red, green, blue and alpha. The Awake() method stored these starting values into variables *r, g, b, alpha*.

The Update() method subtracts the value of *alphaStep* from the *alpha* variable each frame, and creates a new Color object to be applied to the Material of the GUIText object to which this script class is a component. Once the alpha transparency gets below the variable *deathAlpha* the GameObject destroys itself.

There's more...

Some details you don't want to miss:

Timed fading message

By adding a countdown timer, rather than the less reliable number of frames, a more accurate number of seconds can be defined. The alpha value can be set to the normalised (0.0 – 1.0) proportion of the time remaining.

Several messages

By creating a prefab, and placing a copy of the GUIText object with this script attached, messages can be easily created as required by the main game manager. Assuming a public variable *fadingMessagePrefab* has been assigned a value of the prefab via the Inspector, then the following method could be called whenever a fading message is required, which creates an instance of the GUIText prefab, and sets its text property to whatever message is passed in to the method:

```
    private void CreateMessage(string message)
    {
        GameObject fadingMessageGO =
(GameObject)Instantiate(fadingMessagePrefab);
```

```
        fadingMessageGO.guiText.text = message;
    }
```

Reuse single message

Another strategy is to have a single GUIText object always present in the scene, but not always visible. Each time a message is to be displayed to the user, the new text is set and the alpha is set back to 1 (or its desired starting value), and it will start fading again.

See also

Single object inventory – text display (key pickup)

Many games involve the player pickup up items. Perhaps the most common is the need to pick up a key to be able to pass through a doorway.

0423_04_18.png

Getting ready

In folder 04_13 you'll find some key images, and a criss-cross image to apply to the terrain.

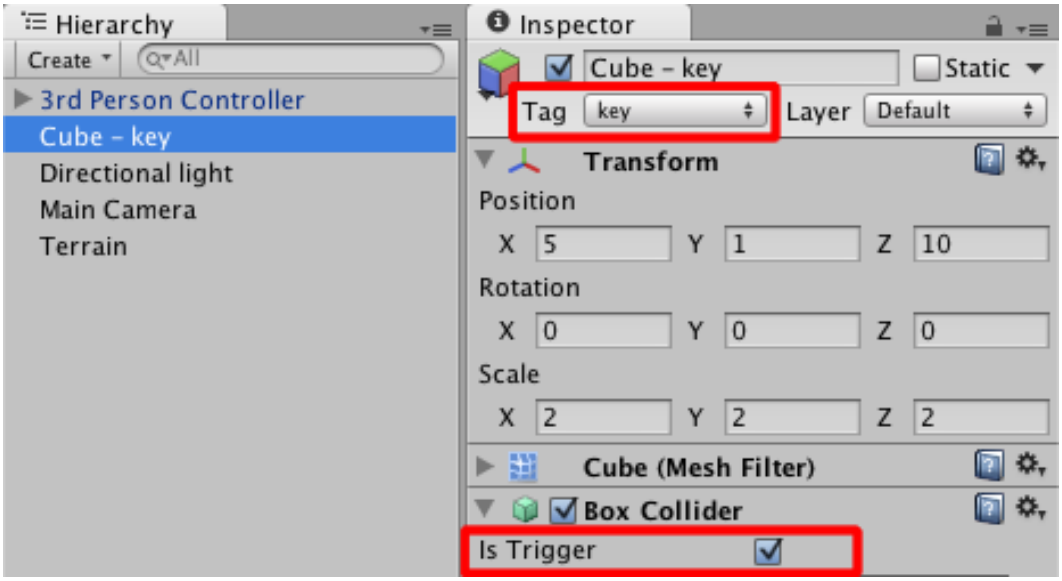
How to do it...

1 – Create a new scene, and add a directional light

2 – Create a new Terrain, and apply a texture to it

3 – Create a cube named *Cube–key* at position (0, 1, 5) with scale (2, 2, 2)

4 – Tag *Cube–key* with the string 'key', and tick its 'Is Trigger' checkbox



0423_04_19.png

5 – Add a key image to *Cube–key*

6 – Import the built-in Unity Package *Character Controller* and add a 3rd person character controller to your scene at position (0, 1, 0)

7 – Attach the following script class to your character controller:

```
// file: PlayerInventory
using UnityEngine;
using System.Collections;

public class PlayerInventory : MonoBehaviour
{
   private bool isCarryingKey = false;

   private void OnGUI()
   {
      string keyMessage = "(bag is empty)";
      if( isCarryingKey )
```

```
        {
            keyMessage = "carrying: [ key ]";
        }

        GUILayout.Label ( keyMessage );
    }

    private void OnTriggerEnter(Collider hitCollider)
    {
        if( "key" == hitCollider.tag )
        {
            isCarryingKey = true;
            Destroy ( hitCollider.gameObject );
        }
    }
}
```

How it works...

The bool variable *isCarryingKey* represents whether or not the player is carrying the key at any point in time.

In the OnGUI() method the contents of string keyMessage is displayed via a GUILayout.Label(). The default value of this string tells the user that the players bag is empty, but an 'if' statement tests the value of *isCarryingKey*, and if that is true then the message is changed to inform that the player is carrying a key.

The OnTriggerEnter() method tests the string 'tag' of any object the player's character controller collides with that has its 'Is Trigger' checkbox set to true.

Each time the player's character controller collides with any object that has its 'Is Trigger' set to true an OnTriggerEnter()event message is sent to both objects involved in the collision.

The OnTriggerEnter() message is passed a parameter that is the Collider component inside the object just collided with. Our character controller's OnTriggerEnter() method tests the 'tag' string of the object collided with to see if has the value 'key'.

Since the cube *Cube–key* we created has its trigger set, and has the tag 'key', then the 'if' statement inside this method will detect a collision with *Cube–key* and does two actions: it sets the bool variable *isCarryingKey* to true, and it destroys the game object it has just collided with (in this case *Cube–key*).

There's more...

Some details you don't want to miss:

Boolean variables often referred to as 'flags'

The use of a bool (true/false) variable to represent whether some feature of the game state is true or false is very common. Programmers often refer to these variables as 'flags'. So programmers might refer to variable *isCarryingKey* as the **key carrying flag**.

See also

- Single object icon inventory – icon display (key pickup)
- A general purpose Inventory PickUp class for inventory item pickup and display

Single object icon inventory – icon display (key pickup)

The previous recipe communicated whether or not the player was carrying a key via a text string on screen. The use of graphical icons often results in a more engaging GUI…

Getting ready

In folder 04_14 you'll find a key icon image, and an empty inventory icon image. In folder 04_13 you'll find some key images, and a criss-cross image to apply to the terrain.

How to do it...

1 – Create a new scene, and add a directional light

2 – Create a new Terrain, and apply a texture to it

3 – Create a cube named *Cube–key* at position (0, 1, 5) with scale (2, 2, 2)

4 – Tag *Cube–key* with the string 'key', and tick its 'Is Trigger' checkbox

5 – Add a key image to *Cube–key*

6 – Import the built-in Unity Package *Character Controller* and add a 3rd person character controller to your scene at position (0, 1, 0)

7 – Attach the following script class to your character controller:

```
// file: PlayerInventoryIcon.cs
using UnityEngine;
using System.Collections;

public class PlayerInventoryIcon : MonoBehaviour
```

```
{
   public Texture keyIcon;
   public Texture emptyIcon;

   private bool isCarryingKey = false;

   private void OnGUI()
   {
      if( isCarryingKey )
         GUILayout.Label( keyIcon );
      else
         GUILayout.Label( emptyIcon );
   }

   private void OnTriggerEnter(Collider hitCollider)
   {
      if( "key" == hitCollider.tag )
      {
         isCarryingKey = true;
         Destroy ( hitCollider.gameObject );
      }
   }
}
```

8 – With the 3rd person controller selected in the Hierarchy, drag the key icon and empty icon images into the Inspector for the corresponding public variables.

How it works...

Two public Texture variables hold the icons for a key and an empty inventory. Method OnGUI() uses an 'if' statement to test the value of *isCarryingKey*. if it is true then a key icon image is displayed, if not, then an icon representing an empt inventory is displayed.

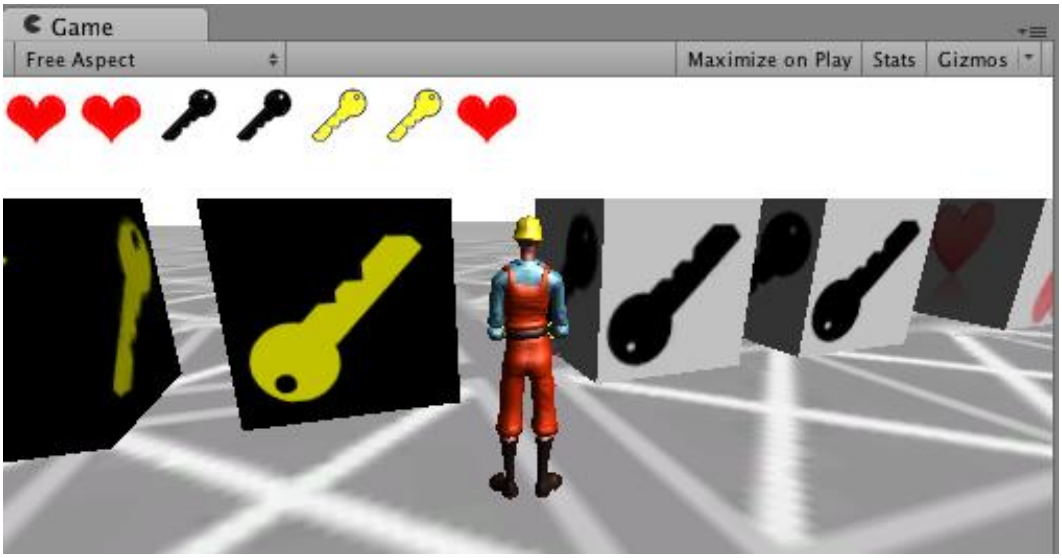
0423_04_20.png

See also

- Single object inventory – text display (key pickup)
- A general purpose Inventory PickUp class for inventory item pickup and display

A general purpose Inventory PickUP class for inventory item pickup and display

There are often several different kinds of items players are expected to pickup and collect during a game (keys, extra lives, items that score points etc.). A general PickUp class to represent the properties for each pickup item can be very useful, and the GUI display of inventory items can be made straightforward using a Csharp List<> of such objects.



0423_04_21.png

Getting ready

In folder 04_15 you'll find selection of images and icons for keys and hearts.

How to do it...

1 – Create a new scene, and add a directional light

2 – Create a new Terrain, and apply a texture to it

3 – Create the following script class:

```
// file: PickUp.cs
using UnityEngine;
using System.Collections;

public class PickUp : MonoBehaviour {
   public enum PickUpCategory{
      KEY, HEALTH, SCORE
   }

   public Texture icon;
   public int points;
   public string fitsLockTag;
   public PickUpCategory catgegory;
}
```

4 – Import the built-in Unity Package *Character Controller* and add a 3rd person character controller to your scene at position (0, 1, 0)

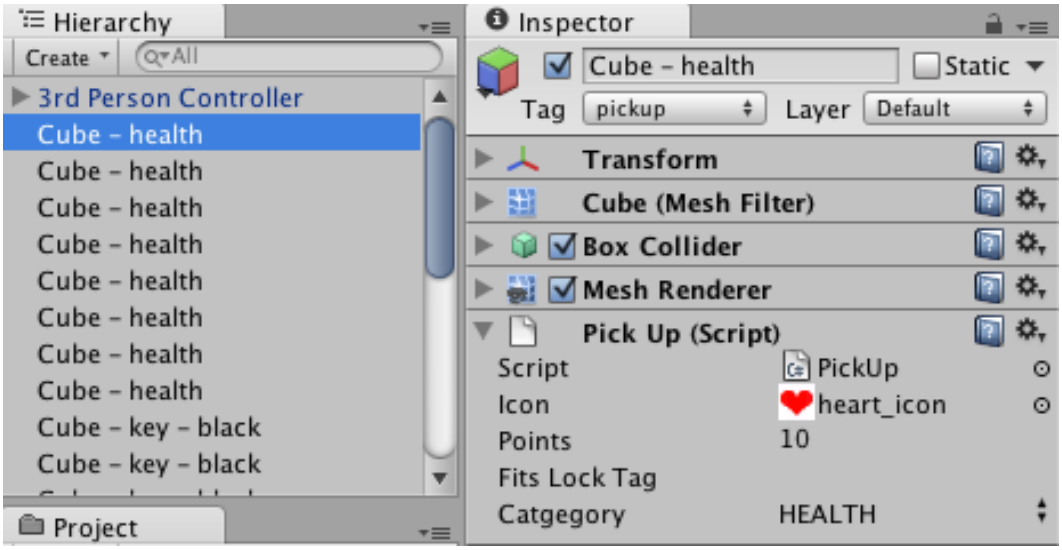
5 – Attach the following script class to your character controller:

```
// file: GeneralInventory.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class GeneralInventory : MonoBehaviour {
   const int ICON_HEIGHT = 32;
   private List<PickUp> inventory = new List<PickUp>();

   private void OnGUI(){
      // restrict display to left of screen
      Rect r = new Rect(0,0,Screen.width/2, ICON_HEIGHT);
      GUILayout.BeginArea(r);
      GUILayout.BeginHorizontal();

      DisplayInventory();

      GUILayout.FlexibleSpace();
      GUILayout.EndHorizontal();
      GUILayout.EndArea();
   }

   private void DisplayInventory(){
      foreach (PickUp item in inventory)
          GUILayout.Label( item.icon );
   }

   private void OnTriggerEnter(Collider hitCollider){
      if( "pickup" == hitCollider.tag ){
         PickUp item = hitCollider.GetComponent<PickUp>();
         inventory.Add( item );
         Destroy ( hitCollider.gameObject );
      }
   }
}
```

6 – Create a cube named *Cube – health* at position (0, 1, 5) with scale (2, 2, 2), and do the following:

- Tag this objet with ‘pickup’
- tick its ‘Is Trigger’ checkbox
- add script class *PickUp* as a component of this object

- drag the heart icon into the Inspector for the corresponding public variable
- choose HEALTH form the dropdown *Category* list
- add the heart image to the material for this object



0423_04_22.png

7 – Make several duplicates of *Cube – health*, and change some of them to have coloured key images and icons as appropriate – so that there are several different items for the player to pickup in your game

How it works...

The PickUp class has no methods, but it declares several useful public variables. It also declares a public 'enum' *Category* defining three possible categories of pickups: KEY, HEALTH and SCORE. Each object of the class PickUp can define an image icon, an integer number of points, a string (for key items, the tag of locks that such a key would fit), and which of the enum categories each PickUp item belongs to.

GameObjects can have a scripted object of the class PickUp added as a component. In this recipe cubes with images to illustrate what they are have been used, but the GameObjects the player meets in the game world could be interactive 3D objects or whatever. Each GameObject with a PickUp component needs to have its properties set appropriately, so yellow key objects should have a yellow key icon assigned, and be set

as being in the category KEY, and have the string tag set for locks it can open.

The player's character controller has the GeneralInventory script added. This is a relatively straightforward GUI script that has two main functions: first, it maintains a List<> of PickUp objects, representing items the player is 'carrying' at any point in time, and it displays the icons for each item via the OnGUI() method. Second, it detected collisions with pickup GameObjects via the OnTriggerEnter() method, and objects tagged 'pickup' that are collided with, are added to the inventory list.

There's more...

Some details you don't want to miss:

Not all pickups have to be displayed

By having different categories of pickups, the actions for collisions can be different. So perhaps for HEALTH pickups the *points* value is added to the player's health, and the object Destroyed, rather than being added to the inventory. This would simply require 'if' or 'case' statements inside the OnTriggerEnter() method to decide what to do once an object tagged 'pickup' has been collided with.

Removing items from the List<>

Events may occur (e.g. open a door for a key being carried) that result in an item needing to be removed from the inventory List<>. For example, if a yellow door were colloided with, and the player was carrying a key that could open such doors, then that item should be removed from the inventory List<> and the door be opened. To implement this you would need to add an 'if' test inside OnTriggerEnter() to detected collision with the item tagged 'yellowDoor':

```
if( "yellowDoor" == hitCollider.tag )
   OpenDoor(hitCollider.gameObject);
```

Method OpenDoor() would need to identify which item (if any) in the inventory can open such a door, and if found, then that item should be removed from the List<> and the door be opened by the appropriate method.

```
private void OpenDoor(GameObject doorGO){
   // search for key to open the tag of doorGO
   int colorKeyIndex = FindItemIndex(doorGO.tag);
   if( colorKeyIndex > -1 ){
      // remove key item
      inventory.RemoveAt( colorKeyIndex );

      // now open the door ...
```

```
      doorGO.animation.Play ("open");
   }
}

private int FindItemIndex(string doorTag){
   for (int i = 0; i < inventory.Count; i++){
      PickUp item = inventory[i];
      if( item.fitsLockTag == doorTag )
         return i;
   }
}
```

See also

- Single object inventory – text display (key pickup)
- A general purpose Inventory GUI class for inventory item pickup and display