ESCAPE FROM PERFECTION

ABSTRACT

PROJECT DOMAIN :

Game Development

PROJECT TITLE:

2D Puzzle Platformer “Escape from Reality”

OBJECTIVE :

The objective of this project is to create a thought-provoking 2D puzzle platformergame.The game includes ananimated 2D sprite which is controlled manually to dodge the obstacles and solve the puzzles embedded in the world.The game is won when all the puzzles are solved and the player remains alive.

EXISTING SYSTEM :

1) A Sprite that can be controlled using W, A, S, D, Space, CTRL keys.
2) Automated and animated enemy objects placed at intervals in the world.
3) Sound Effects corresponding to the scene.
4) Menu and options.

PROPOSED SYSTEM:

1) Hidden alternate paths.
2) Limited number of lives.
3) RESTART/CONTINUE button.
4) Automated scorekeeping.
5) Story mode.

FUTURE EXPANSION:

1) Network communication (Controller,more or less).
2) In-game purchases.
3) Level and difficulty select.
4) Cross-platform.

LANGUAGE: C#

CONTENTS

TOPICS **Page No.**

WHAT IS A 2D GAME?

2D games are typically Scrollers. The first Super Mario Bros. on NES fits one type of Scroller.

Your avatar is flat. You can move left, right, up, and down.

There are several types of 2D games,.. several styles. So beyond that flat avatar and those directions of movement, it is hard to find any other constant.

A 3D game usually suggest the incorporation of Depth. If you were to look into details on the Dimensions, Depth is the third Dimension. So the term holds up in reality as well.

There are several types of 3D games though, so Depth is not always a factor. Sometimes just the lighting and shadow on a sprite makes people label something 3D.

Now all this gets confusing when you have Aerodynamic games because your movements might be in the 3D range. However, these games use to scroll on systems like NES and in Arcades. So you were only able to move within three dimensions on the current screen/range and you were limited to that. Just as would happen with Super Mario Bros. you could not move backwards on the game or to a previous screen.

So, typically a 2D game is a scroller and you can only move Left and Right, or Up and Down, or Left, Right, Up, and Down. Hence, the keyboard arrow inputs are more than sufficient for a player to play the game.

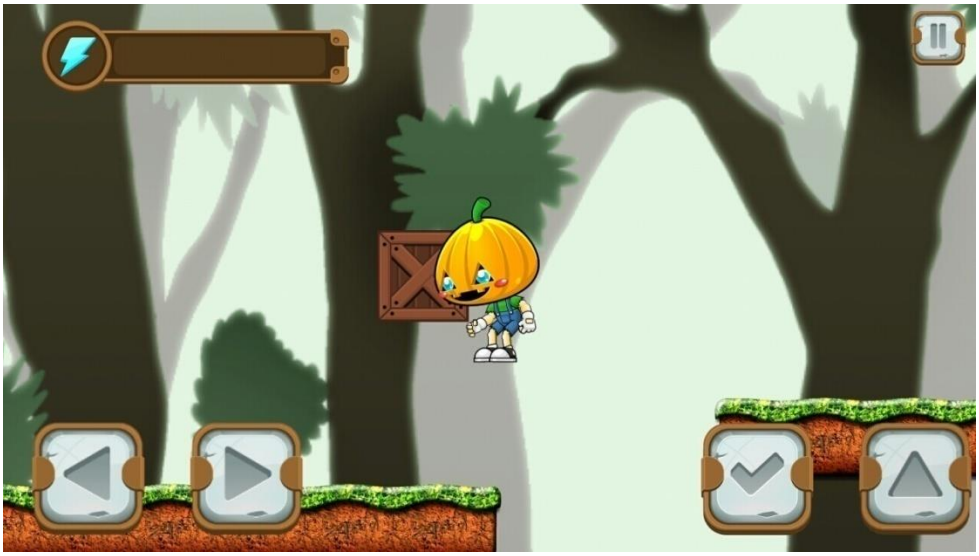
PUZZLE PLATFORM GAME

There are many genres for games and one of them is the puzzle platform. In such type of games, there are multiple suspended platforms on which the player runs around and tries to complete a task based on the assigned objectives.

Initially, the player is on one of the platforms. There can be various objectives specified for the player in order to complete the game or a level in the game such as collecting items or power ups, evading from various objects that can harm or kill the player, jumping from one platform to another to reach an end point of a level.

The player generally requires a lot of skill to time all of the movements as well as some luck in favor of him/her to survive in the most difficult levels of the game and emerge as a winner.

This is how a player moves around in puzzle platform games to actually finish a level or even the game:



UNITY GAME ENGINE

Unity is a cross-platform game engine developed by Unity Technologies and used to develop video games for PC, consoles, mobile devices and websites. First announced only for OS X, at Apple's Worldwide Developers Conference in 2005, it has since been extended to target 21 platforms. Nintendo provides free licenses of Unity 5 to all licensed Nintendo Developers along with their software development kits (SDKs) for the Wii U and Nintendo 3DS Family.

Five major versions of Unity have been released. At the 2006 WWDC show, Apple named Unity as the runner up for its Best Use of Mac OS X Graphics category.

FEATURES AND PORTABILITY OF UNITY

With an emphasis on portability, the engine targets the following APIs: Direct3D on Windows and Xbox 360; OpenGL on Mac, Linux, and Windows; OpenGL ES on Android and iOS; and proprietary APIs on video game consoles. Unity allows specification of texture compression and resolution settings for each platform that the game engine supports,[6] and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects.[10] Unity's graphics engine's platform diversity can provide a shader with multiple variants and a declarative fallback specification, allowing Unity to detect the best variant for the current video hardware and, if none are compatible, to fall back to an alternative shader that may sacrifice features for performance.[11]

Unity is notable for its ability to target games to multiple platforms. Within a project, developers have control over delivery to mobile devices, web browsers, desktops, and consoles.[6][12] Supported platforms include Android, Apple TV,[13] BlackBerry 10, iOS, Linux, Nintendo 3DS line,[14][15][16] OS X, PlayStation 4, PlayStation Vita, Unity Web Player (including Facebook[17]), Wii, Wii U, Windows Phone 8, Windows, Xbox 360, and Xbox One. It includes an asset server and Nvidia's PhysX physics engine. Unity Web Player is a browser plugin that is supported in Windows and OS X only,[18] which has been deprecated in favor of WebGL.[3] Unity is the default software development kit (SDK) for Nintendo's Wii U video game console platform, with a free copy included by Nintendo with each Wii

U developer license. Unity Technologies calls this bundling of a third-party SDK an "industry first".

FAMOUS GAMES DEVELOPED USING UNITY

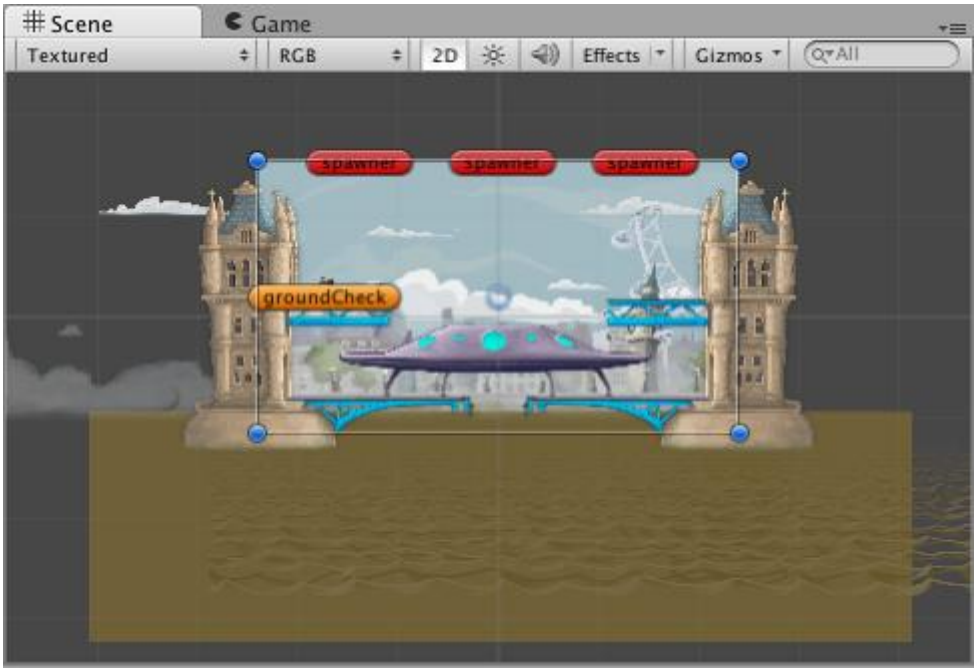
Unity is one of the most widely used platform by both novice developers as well as established developers as it is both developer-friendly as well as very efficient. So, there were games which were successful overnight and some took time to attract users. But, at the end of the day, many games developed on unity platform have been largely successful such as the Temple Run trilogy (Temple Run, Temple Run:Brave, Temple Run 2).

Some of the top games developed using the Unity game engine are:

- Kerbal Space Program
- Hearthstone: Heroes of Warcraft
- Wasteland 2
- Battlestar Galactica Online
- Rust
- Temple Run Trilogy
- Escape plan
- Satellite Reign
- Assassin’s Creed: Identity
- Deus Ex: The Fall

Gameplay in 2D

While famous for its 3D capabilities, Unity can also be used to create 2D games. The familiar functions of the editor are still available but with helpful additions to simplify 2D development.



Scene viewed in 2D mode

The most immediately noticeable feature is the *2D* view mode button in the toolbar of the Scene view. When 2D mode is enabled, an orthographic (ie, perspective-free) view will be set; the camera looks along the Z axis with the Y axis increasing upwards. This allows you to visualise the scene and place 2D objects easily.

For a full list of 2D components, how to switch between 2D and 3D mode, and the different 2D and 3D Mode settings, see 2D or 3D Projects.

2D Graphics

Graphic objects in 2D are known as **Sprites**. Sprites are essentially just standard textures but there are special techniques for combining and managing sprite textures for efficiency and convenience during development. Unity provides a built-in Sprite Editor to let you extract sprite graphics from a larger image. This allows you to edit a number of component images within a single texture in your image editor. You could

use this, for example, to keep the arms, legs and body of a character as separate elements within one image.

Sprites are rendered with a Sprite Renderer component rather than the Mesh Renderer used with 3D objects. You can add this to a GameObject via the Components menu (**Component > Rendering > Sprite Renderer** or alternatively, you can just create a GameObject directly with a Sprite Renderer already attached (menu: **GameObject > 2D Object > Sprite**).

In addition, you can use a Sprite Creator tool to make placeholder 2D images.

2D Physics

Unity has a separate physics engine for handling 2D physics so as to make use of optimizations only available with 2D. The components correspond to the standard 3D physics components such as Rigidbody, Box Collider and Hinge Joint, but with "2D" appended to the name. So, sprites can be equipped with Rigidbody 2D, Box Collider 2D and Hinge Joint 2D. Most 2D physics components are simply "flattened" versions of the 3D equivalents (eg, *Box Collider 2D* is a square while *Box Collider* is a cube) but there are a few exceptions.

For a full list of 2D Physics components, see 2D or 3D Projects. See the Physics section of the manual for further information about 2D physics concepts and components.

Sprites

Sprites are 2D Graphic objects. If you are used to working in 3D, **Sprites** are essentially just standard textures but there are special techniques for combining and managing sprite textures for efficiency and convenience during development.

Unity provides a placeholder Sprite Creator, a built-in Sprite Editor, a Sprite Renderer and a Sprite Packer

See ***Importing and Setting up Sprites*** below for information on setting up assets as **Sprites** in your Unity project.

Sprite Tools

Sprite Creator

Use the Sprite Creator to create placeholder sprites in your project, so you can carry on with development without having to source or wait for graphics.

Sprite Editor

The Sprite Editor lets you extract sprite graphics from a larger image and edit a number of component images within a single texture in your image editor. You could use this, for example, to keep the arms, legs and body of a character as separate elements within one image.

Sprite Renderer

Sprites are rendered with a Sprite Renderer component rather than the Mesh Renderer used with 3D objects. Use it to display images as **Sprites** for use in both 2D and 3D scenes.

Sprite Packer

Use Sprite Packer to opimize the use and performance of video memory by your project.

Importing and Setting Up Sprites

Sprites are a type of **Asset** in Unity projects. You can see them, ready to use, via the **Project View**.

There are two ways to bring **Sprites** into your project:

1. In your computer's Finder (Mac OS X) or File Explorer (Windows), place your image directly into your Unity project's **Assets**folder.
   Unity detects this and displays it in your project's **Project View**.
2. In Unity, go to **Assets>Import New Asset…** to bring up your computer's Finder (Mac OS X) or File Explorer (Windows).
   From there, select the image you want, and Unity puts it in the **Project View**.

See Importing Assets for more details on this and important information about organising your **Assets** folder.

Setting your Image as a Sprite

If your project mode is set to 2D, the image you import is automatically set as a **Sprite**.

However, if your project mode is set to 3D, your image is set as a **Texture**, so you need to change the asset's **Texture Type**:

1. Click on the asset to see its **Import Inspector**.
2. Set the **Texture Type** to **Sprite (2D and UI)**.

2D PHYSICS REFERENCE

Rigidbody 2D

A Rigidbody 2D component places a GameObject under the control of the physics engine. Many concepts familiar from the standard Rigidbody component carry over to Rigidbody 2D; the differences are that in 2D, objects can only move in the XY plane and can only rotate on an axis perpendicular to that plane.

How a Rigidbody 2D works

Usually, the Unity Editor's Transform component defines how a GameObject (and its child GameObjects) is positioned, rotated and scaled within the Scene. When it is changed, it updates other components, which may update things like where they render or where colliders are positioned. The 2D physics engine is able to move colliders and make them interact with each other, so a method is required for the physics engine to communicate this movement of colliders back to the Transform components. This movement and connection with colliders is what a Rigidbody 2D component is for.

The Rigidbody 2D component overrides the Transform and updates it to a position/rotation defined by the Rigidbody 2D. Note that while you can still override

the Rigidbody 2D by modifying the Transform component yourself (because Unity exposes all properties on all components), doing so will cause problems such as GameObjects passing through or into each other, and unpredictable movement.

Any Collider 2D component added to the same GameObject or child GameObject is implicitly attached to that Rigidbody 2D. When a Collider 2D is attached to the Rigidbody 2D, it moves with it. A Collider 2D should never be moved directly using the Transform or any collider offset; the Rigidbody 2D should be moved instead. This offers the best performance and ensures correct collision detection. Collider 2Ds attached to the same Rigidbody 2D won't collide with each other. This means you can create a set of colliders that act effectively as a single compound collider, all moving and rotating in sync with the Rigidbody 2D.

When designing a Scene, you are free to use a default Rigidbody 2D and start attaching colliders. These colliders allow any other colliders attached to different Rigidbody 2Ds to collide with each other.

Adding a **Rigidbody 2D** allows a sprite to move in a physically convincing way by applying forces from the scripting API. When the appropriate collider component is also attached to the sprite GameObject, it is affected by collisions with other moving GameObjects. Using physics simplifies many common gameplay mechanics and allows for realistic behavior with minimal coding.

Kinematic Rigidbody 2D components

The **Is Kinematic** setting switches off the physical behaviour of the Rigidbody 2D so that it will not react to gravity and collisions. This is typically used to keep a GameObject under non-physical script control most of the time, but then switch to physics in a particular situation. For example, a player might normally move by walking (better handled without physics) but then get catapulted into the air by an explosion or strike (better handled with physics). Physics can be used to create the catapulting effect if you switch off **Is Kinematic** just before applying a large force to the GameObject.

Collider 2D

Colliders define an approximate shape for an object that's used by the physics engine to determine collisions with other objects.

2D colliders all have names ending '2D'. A collider named without a '2D' ending is a 3D joint. Note that you can't mix 3D game objects and 2D colliders or 2D game objects and 3D colliders.

The collider types that can be used with Rigidbody 2D are:

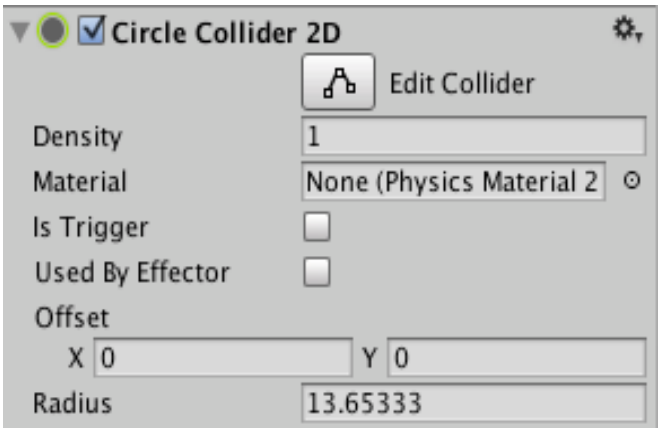
- Circle Collider 2D
- Box Collider 2D
- Edge Collider 2D
- Polygon Collider 2D

Use Auto Mass:

Note that a **Collider**'s mass determines the **Rigidbody**'s mass if you have **Use Auto Mass** selected. This option is good to use in conjunction with the **Buoyancy Effector** 2D.

Circle Collider 2D

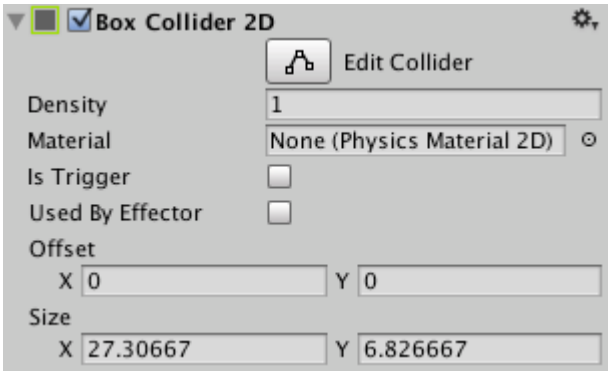
The **Circle Collider 2D** class is a collider for use with 2D physics. The collider's shape is a circle with a defined position and radius in the local coordinate space of a **Sprite**.



Property:	*Function:*
Density	Changing the density here affects the mass of the GameObject's associated Rigidbody 2D. If you set the value to zero, its associated Rigidbody 2D ignores the Circle Collider 2D for all mass calculations, including centre of mass calculations. Note that this option is only available if you have selected **Use Auto Mass** in an associated Rigidbody 2D.
Material	A physics material that determines properties of collisions, such as friction and bounce.
Is Trigger	Check this if you want the Circle Collider 2D to behave as a trigger.
Used by Effector	Check this if you want the Circle Collider 2D to be used by an attached Effector 2D.
Offset	The local offset of the Circle Collider 2D geometry.
Radius	Radius of the circle in local space units.

Box Collider 2D

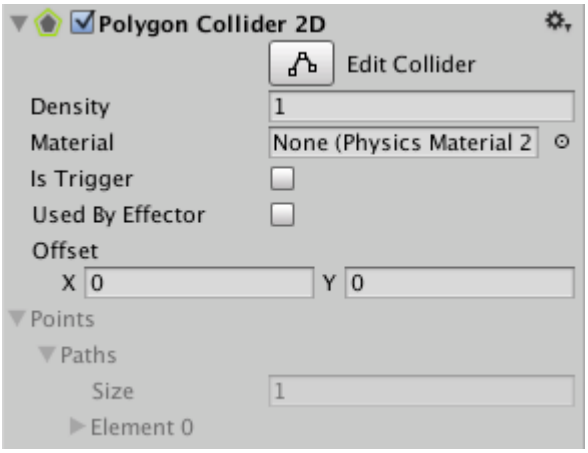
The **Box Collider 2D** component is a collider for use with 2D physics. Its shape is a rectangle with a defined position, width and height in the local coordinate space of a **Sprite**. Note that the rectangle is axis-aligned - that is, its edges are parallel to the x or y axes of local space.



Property:	***Function:***
Density	Changing the density here affects the mass of the GameObject's associated **Rigidbody 2D**. Set the value to zero and its associated **Rigidbody 2D** ignores the **Collider 2D** for all mass calculations, including centre of mass calculations. Note that this option is only available if you have selected **Use Auto Mass** in an associated **Rigidbody 2D** component.
Material	A physics Material that determines properties of collisions, such as friction and bounce.
Is Trigger	Check this box if you want the Box Collider 2D to behave as a trigger.
Used by Effector	Check this box if you want the Box Collider 2D to be used by an attached Effector 2D component.
Offset	Set the local offset of the Collider 2D geometry.
Size	Set the size of the box in local space units.

Polygon Collider 2D

The **Polygon Collider 2D** component is a collider for use with 2D physics. The collider's shape is defined by a freeform edge made of line segments, so you can adjust it to fit the shape of the **Sprite** graphic with great precision. Note that this collider's edge *must* completely enclose an area (unlike the similar Edge Collider 2D).



Property:	***Function:***
Density	Changing the density here affects the mass of the object's associated **Rigidbody**. Set the value to zero and its associated **Rigidbody** ignores the **Collider** for all mass calculations, including centre of mass calculations. **NOTE:** This option is only available if you have selected **Use Auto Mass** in an associated **Rigidbody**.
Material	A physics material that determines properties of collisions, such as friction and bounce.
Is Trigger	Does the collider behave as a trigger?
Used by Effector	Whether the collider is used by an attached effector or not.

Property:	*Function:*
Offset	The local offset of the collider geometry.
Points	Non-editable information about the complexity of the generated collider.

Details

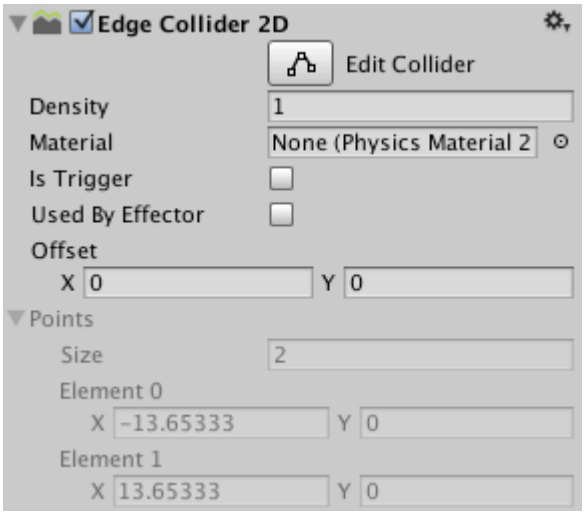
The collider can be edited manually but it is often more convenient to let Unity determine the shape automatically. You can do this by dragging a sprite asset from the Project view onto the Polygon Collider 2D component in the inspector.

You can edit the polygon's shape by pressing the Edit Collider button in the Inspector. You can exit collider edit mode by pressing the Edit Collider button again. While in edit mode, you can move an existing vertex by dragging when the mouse is over that vertex. If you shift-drag while the mouse is over an edge then a new vertex will be created at the mouse location. You can remove a vertex by holding down the ctrl/cmd key while clicking on it.

Note that you can hide the outline of the 2D move gizmo while editing the collider - just click the foldout arrow on the Sprite Renderer component in the Inspector to collapse it.

Edge Collider 2D

The **Edge Collider 2D** component is a collider for use with 2D physics. The collider's shape is defined by a freeform edge made of line segments, so you can adjust it to fit the shape of the **Sprite** graphic with great precision. Note that this collider's edge need not completely enclose an area (unlike the similar Polygon Collider 2D) and can be as simple as a straight line or an L-shape.



Property:	***Function:***
Density	Changing the density here affects the mass of the object's associated **Rigidbody**. Set the value to zero and its associated **Rigidbody** ignores the **Collider** for all mass calculations, including centre of mass calculations. **NOTE:** This option is only available if you have selected **Use Auto Mass** in an associated **Rigidbody**.
Material	A physics material that determines properties of collisions, such as friction and bounce.
Is Trigger	Does the collider behave as a trigger?
Used by Effector	Whether the collider is used by an attached effector or not.
Offset	The local offset of the collider geometry.
Points	Non-editable information about the complexity of the generated collider.

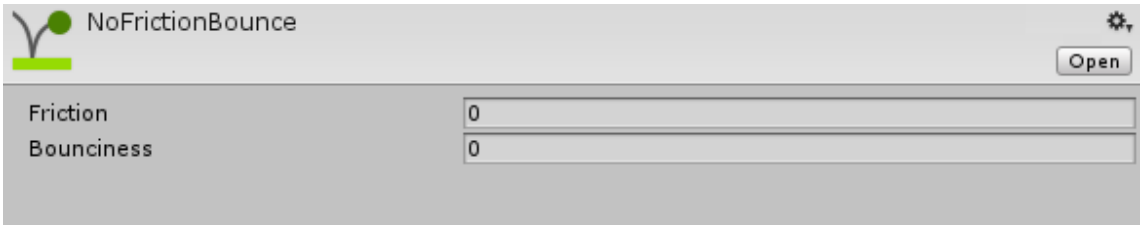
Details

You can edit the polyline directly by holding down the shift key as you move the mouse over an edge or vertex in the Scene view. You can move an existing vertex by shift-dragging when the mouse is over that vertex. If you shift-drag while the mouse is over an edge then a new vertex will be created at the mouse location. You can remove a vertex by holding down the ctrl/cmd key while clicking on it.

Note that you can hide the outline of the 2D move gizmo while editing the collider - just click the foldout arrow on the Sprite Renderer component in the Inspector to collapse it.

Physics Material 2D

A **Physics Material 2D** is used to adjust the friction and bounce that occurs between 2D physics objects when they collide. You can create a Physics Material 2D from the Assets menu (**Assets > Create > Physics2D Material**).



Property:	***Function:***
Friction	Coefficient of friction for this collider.
Bounciness	The degree to which collisions rebound from the surface. A value of 0 indicates no bounce while a value of 1 indicates a perfect bounce with no loss of energy.

Details

To use a Physics Material 2D, simply drag it onto an object with a 2D collider attached or drag it to the collider component in the inspector. Note that for 3D

physics, the equivalent asset is referred to as a *Physic Material* (ie, no S at the end of *physic*) - it is important in scripting not to get the two spellings confused.

SCRIPTING

Scripting is an essential ingredient in all games. Even the simplest game needs scripts, to respond to input from the player and arrange for events in the gameplay to happen when they should. Beyond that, scripts can be used to create graphical effects, control the physical behaviour of objects or even implement a custom AI system for characters in the game.

Scripting is a skill that takes some time and effort to learn. The intention of this section is not to teach you how to write script code from scratch, but rather to explain the main concepts that apply to scripting in Unity.

Creating and Using Scripts

The behavior of GameObjects is controlled by the **Components** that are attached to them. Although Unity's built-in Components can be very versatile, you will soon find you need to go beyond what they can provide to implement your own gameplay features. Unity allows you to create your own Components using **scripts**. These allow you to trigger game events, modify Component properties over time and respond to user input in any way you like.

Unity supports two programming languages natively:

- **C#** (pronounced C-sharp), an industry-standard language similar to Java or C++;
- **UnityScript**, a language designed specifically for use with Unity and modelled after JavaScript;

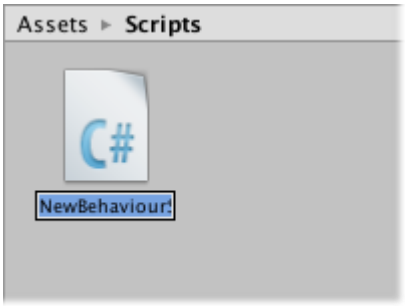
  In addition to these, many other .NET languages can be used with Unity if they can compile a compatible DLL - see here for further details.

Learning the art of programming and the use of these particular languages is beyond the scope of this introduction. However, there are many books, tutorials and other resources for learning how to program with Unity. See the Learning section of our website for further details.

Creating Scripts

Unlike most other assets, scripts are usually created within Unity directly. You can create a new script from the Create menu at the top left of the Project panel or by selecting **Assets > Create > C# Script** (or JavaScript) from the main menu.

The new script will be created in whichever folder you have selected in the Project panel. The new script file's name will be selected, prompting you to enter a new name.



It is a good idea to enter the name of the new script at this point rather than editing it later. The name that you enter will be used to create the initial text inside the file, as described below.

Anatomy of a Script file

When you double-click a script asset in Unity, it will be opened in a text editor. By default, Unity will use MonoDevelop, but you can select any editor you like from the External Tools panel in Unity's preferences.

The initial contents of the file will look something like this:

```
using UnityEngine;

using System.Collections;

public class MainPlayer : MonoBehaviour {
```

```
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
    }
}
```

A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called **MonoBehaviour**. You can think of a class as a kind of blueprint for creating a new Component type that can be attached to GameObjects. Each time you attach a script component to a GameObject, it creates a new instance of the object defined by the blueprint. The name of the class is taken from the name you supplied when the file was created. The class name and file name must be the same to enable the script component to be attached to a GameObject.

The main things to note, however, are the two functions defined inside the class. The **Update** function is the place to put code that will handle the frame update for the GameObject. This might include movement, triggering actions and responding to user input, basically anything that needs to be handled over time during gameplay. To enable the Update function to do its work, it is often useful to be able to set up variables, read preferences and make connections with other GameObjects before any game action takes place. The **Start** function will be called by Unity before gameplay begins (ie, before the Update function is called for the first time) and is an ideal place to do any initialization.

Note to experienced programmers: you may be surprised that initialization of an object is not done using a constructor function. This is because the construction of objects is handled by the editor and does not take place at the start of gameplay as you might expect. If you attempt to define a constructor for a script component, it will interfere with the normal operation of Unity and can cause major problems with the project.

A UnityScript script works a bit differently to C# script:

```
#pragma strict

function Start () {

}

function Update () {

}
```

Here, the Start and Update functions have the same meaning but the class is not explicitly declared. The script itself is assumed to define the class; it will implicitly derive from MonoBehaviour and take its name from the filename of the script asset.

Controlling a GameObject

As noted above, a script only defines a blueprint for a Component and so none of its code will be activated until an instance of the script is attached to a GameObject. You can attach a script by dragging the script asset to a GameObject in the hierarchy panel or to the inspector of the GameObject that is currently selected. There is also a Scripts submenu on the Component menu which will contain all the scripts available in the project, including those you have created yourself. The script instance looks much like any other Component in the Inspector:

Once attached, the script will start working when you press Play and run the game. You can check this by adding the following code in the Start function:-

```
// Use this for initialization

void Start () {

   Debug.Log("I am alive!");

}
```

Debug.Log is a simple command that just prints a message to Unity's console output. If you press Play now, you should see the message at the bottom of the main Unity editor window and in the Console window (menu: **Window > Console**).

Variables and the Inspector

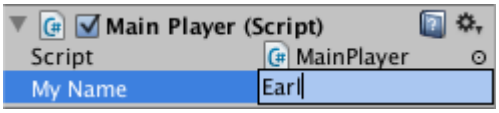
When creating a script, you are essentially creating your own new type of component that can be attached to Game Objects just like any other component.

Just like other Components often have properties that are editable in the inspector, you can allow values in your script to be edited from the Inspector too.

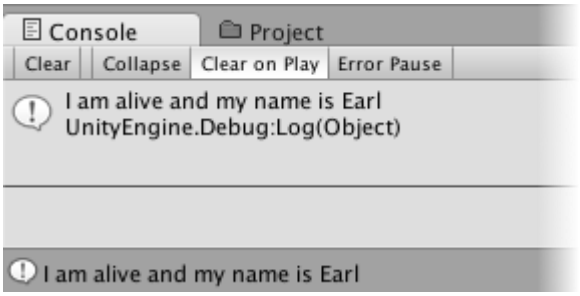
```
using UnityEngine;

using System.Collections;

public class MainPlayer : MonoBehaviour {

   public string myName;
```

```
    // Use this for initialization

    void Start () {

        Debug.Log("I am alive and my name is " + myName);

    }

    // Update is called once per frame

    void Update () {

    }

}
```

This code creates an editable field in the Inspector labelled “My Name”.



Unity creates the Inspector label by introducing a space wherever a capital letter occurs in the variable name. However, this is purely for display purposes and you should always use the variable name within your code. If you edit the name and then press Play, you will see that the message includes the text you entered.



In C#, you must declare a variable as public to see it in the Inspector. In UnityScript, variables are public by default unless you specify that they should be private:

```
#pragma strict

private var invisibleVar: int;

function Start () {

}
```

Unity will actually let you change the value of a script's variables while the game is running. This is very useful for seeing the effects of changes directly without having to stop and restart. When gameplay ends, the values of the variables will be reset to whatever they were before you pressed Play. This ensures that you are free to tweak your object's settings without fear of doing any permanent damage.

Controlling GameObjects Using Components

In the Unity editor, you make changes to Component properties using the Inspector. So, for example, changes to the position values of the Transform Component will result in a change to the GameObject's position. Similarly, you can change the color of a Renderer's material or the mass of a Rigidbody with a corresponding effect on the appearance or behavior of the GameObject. For the most part, scripting is also about modifying Component properties to manipulate GameObjects. The difference, though, is that a script can vary a property's value gradually over time or in response to input from the user. By changing, creating and destroying objects at the right time, any kind of gameplay can be implemented.

Accessing Components

The simplest and most common case is where a script needs access to other Components attached to the same GameObject. As mentioned in the Introduction section, a Component is actually an instance of a class so the first step is to get a reference to the Component instance you want to work with. This is done with the GetComponent function. Typically, you want to assign the Component object to a variable, which is done in C# using the following syntax:

```
void Start () {

    Rigidbody rb = GetComponent<Rigidbody>();

}
```

In UnityScript, the syntax is subtly different:

```
function Start () {

    var rb = GetComponent.<Rigidbody>();

}
```

Once you have a reference to a Component instance, you can set the values of its properties much as you would in the Inspector:

```
void Start () {

    Rigidbody rb = GetComponent<Rigidbody>();

    // Change the mass of the object's Rigidbody.

    rb.mass = 10f;

}
```

An extra feature that is not available in the Inspector is the possibility of calling functions on Component instances:

```
void Start () {

   Rigidbody rb = GetComponent<Rigidbody>();

   // Add a force to the Rigidbody.

   rb.AddForce(Vector3.up * 10f);

}
```

Note also that there is no reason why you can't have more than one custom script attached to the same object. If you need to access one script from another, you can use GetComponent as usual and just use the name of the script class (or the filename) to specify the Component type you want.

If you attempt to retrieve a Component that hasn't actually been added to the GameObject then GetComponent will return null; you will get a null reference error at runtime if you try to change any values on a null object.

Accessing Other Objects

Although they sometimes operate in isolation, it is common for scripts to keep track of other objects. For example, a pursuing enemy might need to know the position of the player. Unity provides a number of different ways to retrieve other objects, each appropriate to certain situations.

Linking Objects with Variables

The most straightforward way to find a related GameObject is to add a public GameObject variable to the script:

```
public class Enemy : MonoBehaviour {

   public GameObject player;

   // Other variables and functions...

}
```

This variable will be visible in the Inspector like any other:

You can now drag an object from the scene or Hierarchy panel onto this variable to assign it. The GetComponent function and Component access variables are available for this object as with any other, so you can use code like the following:

```
public class Enemy : MonoBehaviour {

   public GameObject player;

   void Start() {

      // Start the enemy ten units behind the player character.

      transform.position = player.transform.position - Vector3.forward * 10f;

   }

}
```

Additionally, if declare a public variable of a Component type in your script, you can drag any GameObject that has that Component attached onto it. This will access the Component directly rather than the GameObject itself.

```
public Transform playerTransform;
```

Linking objects together with variables is most useful when you are dealing with individual objects that have permanent connections. You can use an array variable to link several objects of the same type, but the connections must still be made in the Unity editor rather than at runtime. It is often convenient to locate objects at runtime and Unity provides two basic ways to do this, as described below.

Finding Child Objects

Sometimes, a game scene will make use of a number of objects of the same type, such as enemies, waypoints and obstacles. These may need to be tracked by a particular script that supervises or reacts to them (eg, all waypoints may need to be available to a pathfinding script). Using variables to link these objects is a possibility but it will make the design process tedious if each new waypoint has to be dragged to a variable on a script. Likewise, if a waypoint is deleted then it is a nuisance to have to remove the variable reference to the missing object. In cases like this, it is often better to manage a set of objects by making them all children of one parent object. The child objects can be retreived using the parent's Transform Component (since all GameObjects implicitly have a Transform):

```
using UnityEngine;

public class WaypointManager : MonoBehaviour {

    public Transform[] waypoints;

    void Start() {
```

```
        waypoints = new Transform[transform.childCount];

        int i = 0;

        foreach (Transform t in transform) {

            waypoints[i++] = t;

        }

    }

}
```

You can also locate a specific child object by name using the Transform.Find function:

```
transform.Find("Gun");
```

This can be useful when an object has a child that can be added and removed during gameplay. A weapon that can be picked up and put down is a good example of this.

Finding Objects by Name or Tag

It is always possible to locate GameObjects anywhere in the scene hierarchy as long as you have some information to identify them. Individual objects can be retrieved by name using the GameObject.Find function:

```
GameObject player;

void Start() {
```

```
    player = GameObject.Find("MainHeroCharacter");

}
```

An object or a collection of objects can also be located by their tag using the GameObject.FindWithTag and GameObject.FindGameObjectsWithTag functions:

```
GameObject player;

GameObject[] enemies;

void Start() {

    player = GameObject.FindWithTag("Player");

    enemies = GameObject.FindGameObjectsWithTag("Enemy");

}
```

Event Functions

A script in Unity is not like the traditional idea of a program where the code runs continuously in a loop until it completes its task. Instead, Unity passes control to a script intermittently by calling certain functions that are declared within it. Once a function has finished executing, control is passed back to Unity. These functions are known as event functions since they are activated by Unity in response to events that occur during gameplay. Unity uses a naming scheme to identify which function to call for a particular event. For example, you will already have seen the Update function (called before a frame update occurs) and the Start function (called just before the object's first frame update). Many more event functions are available in Unity; the full list can be found in the script reference page for the MonoBehaviour class along

with details of their usage. The following are some of the most common and important events.

Regular Update Events

A game is rather like an animation where the animation frames are generated on the fly. A key concept in games programming is that of making changes to position, state and behavior of objects in the game just before each frame is rendered.
The Update function is the main place for this kind of code in Unity. Update is called before the frame is rendered and also before animations are calculated.

```
void Update() {

   float distance = speed * Time.deltaTime * Input.GetAxis("Horizontal");

   transform.Translate(Vector3.right * distance);

}
```

The physics engine also updates in discrete time steps in a similar way to the frame rendering. A separate event function called FixedUpdate is called just before each physics update. Since the physics updates and frame updates do not occur with the same frequency, you will get more accurate results from physics code if you place it in the FixedUpdate function rather than Update.

```
void FixedUpdate() {

   Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");

   rigidbody.AddForce(force);

}
```

It is also useful sometimes to be able to make additional changes at a point after the Update and FixedUpdate functions have been called for all objects in the scene and after all animations have been calculated. An example is where a camera should remain trained on a target object; the adjustment to the camera's orientation must be made after the target object has moved. Another example is where the script code should override the effect of an animation (say, to make the character's head look towards a target object in the scene). The LateUpdate function can be used for these kinds of situations.

```
void LateUpdate() {

   Camera.main.transform.LookAt(target.transform);

}
```

Initialization Events

It is often useful to be able to call initialization code in advance of any updates that occur during gameplay. The Start function is called before the first frame or physics update on an object. The Awake function is called for each object in the scene at the time when the scene loads. Note that although the various objects' Start and Awake functions are called in arbitrary order, all the Awakes will have finished before the first Start is called. This means that code in a Start function can make use of other initializations previously carried out in the Awake phase.

GUI events

Unity has a system for rendering GUI controls over the main action in the scene and responding to clicks on these controls. This code is handled somewhat differently from the normal frame update and so it should be placed in the OnGUI function, which will be called periodically.

```
void OnGUI() {

   GUI.Label(labelRect, "Game Over");

}
```

You can also detect mouse events that occur over a GameObject as it appears in the scene. This can be used for targeting weapons or displaying information about the character currently under the mouse pointer. A set of OnMouseXXX event functions (eg, OnMouseOver, OnMouseDown) is available to allow a script to react to user actions with the mouse. For example, if the mouse button is pressed while the pointer is over a particular object then an OnMouseDown function in that object's script will be called if it exists.

Physics events

The physics engine will report collisions against an object by calling event functions on that object's script.
The OnCollisionEnter, OnCollisionStay and OnCollisionExit functions will be called as contact is made, held and broken. The
corresponding OnTriggerEnter, OnTriggerStay and OnTriggerExit functions will be called when the object's collider is configured as a Trigger (ie, a collider that simply detects when something enters it rather than reacting physically). These functions may be called several times in succession if more than one contact is detected during the physics update and so a parameter is passed to the function giving details of the collision (position, identity of the incoming object, etc).

```
void OnCollisionEnter(otherObj: Collision) {

   if (otherObj.tag == "Arrow") {

      ApplyDamage(10);
```

```
    }

}
```

Time and Framerate Management

The Update function allows you to monitor inputs and other events regularly from a script and take appropriate action. For example, you might move a character when the “forward” key is pressed. An important thing to remember when handling time-based actions like this is that the game’s framerate is not constant and neither is the length of time between Update function calls.

As an example of this, consider the task of moving an object forward gradually, one frame at a time. It might seem at first that you could just shift the object by a fixed distance each frame:

```
//C# script example

using UnityEngine;

using System.Collections;

public class ExampleScript : MonoBehaviour {

    public float distancePerFrame;

    void Update() {

        transform.Translate(0, 0, distancePerFrame);

    }
```

```
}
```

However, given that the frame time is not constant, the object will appear to move at an irregular speed. If the frame time is 10 milliseconds then the object will step forward by *distancePerFrame* one hundred times per second. But if the frame time increases to 25 milliseconds (due to CPU load, say) then it will only step forward forty times a second and therefore cover less distance. The solution is to scale the size of the movement by the frame time which you can read from the Time.deltaTimeproperty:

```
//C# script example

using UnityEngine;

using System.Collections;

public class ExampleScript : MonoBehaviour {

    public float distancePerSecond;



    void Update() {

        transform.Translate(0, 0, distancePerSecond * Time.deltaTime);

    }

}
```

Note that the movement is now given as *distancePerSecond* rather than *distancePerFrame*. As the framerate changes, the size of the movement step will change accordingly and so the object's speed will be constant.

Fixed Timestep

Unlike the main frame update, Unity's physics system *does* work to a fixed timestep, which is important for the accuracy and consistency of the simulation. At the start of the physics update, Unity sets an "alarm" by adding the fixed timestep value onto the time when the last physics update ended. The physics system will then perform calculations until the alarm goes off.

You can change the size of the fixed timestep from the Time Manager and you can read it from a script using the Time.fixedDeltaTime property. Note that a lower value for the timestep will result in more frequent physics updates and more precise simulation but at the cost of greater CPU load. You probably won't need to change the default fixed timestep unless you are placing high demands on the physics engine.

Maximum Allowed Timestep

The fixed timestep keeps the physical simulation accurate in real time but it can cause problems in cases where the game makes heavy use of physics and the gameplay framerate has also become low (due to a large number of objects in play, say). The main frame update processing has to be "squeezed" in between the regular physics updates and if there is a lot of processing to do then several physics updates can take place during a single frame. Since the frame time, positions of objects and other properties are frozen at the start of the frame, the graphics can get out of sync with the more frequently updated physics.

Naturally, there is only so much CPU power available but Unity has an option to let you effectively slow down physics time to let the frame processing catch up.
The *Maximum Allowed Timestep* setting (in the Time Manager) puts a limit on the amount of time Unity will spend processing physics and FixedUpdate calls during a given frame update. If a frame update takes longer than *Maximum Allowed Timestep* to process, the physics engine will "stop time" and let the frame processing catch up. Once the frame update has finished, the physics will resume as though no

time has passed since it was stopped. The result of this is that rigidbodies will not move perfectly in real time as they usually do but will be slowed slightly. However, the physics "clock" will still track them as though they were moving normally. The slowing of physics time is usually not noticeable and is an acceptable trade-off against gameplay performance.

Time Scale

For special effects, such as "bullet-time", it is sometimes useful to slow the passage of game time so that animations and script responses happen at a reduced rate. Furthermore, you may sometimes want to freeze game time completely, as when the game is paused. Unity has a *Time Scale* property that controls how fast game time proceeds relative to real time. If the scale is set to 1.0 then game time matches real time. A value of 2.0 makes time pass twice as quickly in Unity (ie, the action will be speeded-up) while a value of 0.5 will slow gameplay down to half speed. A value of zero will make time "stop" completely. Note that the time scale doesn't actually slow execution but simply changes the time step reported to the Update and FixedUpdate functions via Time.deltaTime and Time.fixedDeltaTime. The Update function is likely to be called more often than usual when game time is slowed down but the *deltaTime* step reported each frame will simply be reduced. Other script functions are not affected by the time scale so you can, for example, display a GUI with normal interaction when the game is paused.

The Time Manager has a property to let you set the time scale globally but it is generally more useful to set the value from a script using the Time.timeScale property:

```
//C# script example

using UnityEngine;

using System.Collections;

public class ExampleScript : MonoBehaviour {
```

```
    void Pause() {

        Time.timeScale = 0;

    }

    void Resume() {

        Time.timeScale = 1;

    }

}
```

Capture Framerate

A very special case of time management is where you want to record gameplay as a video. Since the task of saving screen images takes considerable time, the usual framerate of the game will be drastically reduced if you attempt to do this during normal gameplay. This will result in a video that doesn't reflect the true performance of the game.

Fortunately, Unity provides a Capture Framerate property that lets you get around this problem. When the property's value is set to anything other than zero, game time will be slowed and the frame updates will be issued at precise regular intervals. The interval between frames is equal to 1 / Time.captureFramerate, so if the value is set to 5.0 then updates occur every fifth of a second. With the demands on framerate effectively reduced, you have time in the Update function to save screenshots or take other actions:

```
//C# script example

using UnityEngine;
```

```
using System.Collections;

public class ExampleScript : MonoBehaviour {

    // Capture frames as a screenshot sequence. Images are

    // stored as PNG files in a folder - these can be combined into

    // a movie using image utility software (eg, QuickTime Pro).

    // The folder to contain our screenshots.

    // If the folder exists we will append numbers to create an empty folder.

    string folder = "ScreenshotFolder";

    int frameRate = 25;

    void Start () {

        // Set the playback framerate (real time will not relate to game time after this).

        Time.captureFramerate = frameRate;

        // Create the folder

        System.IO.Directory.CreateDirectory(folder);

    }

    void Update () {
```

```
        // Append filename to folder name (format is '0005 shot.png"')

        string name = string.Format("{0}/{1:D04} shot.png", folder, Time.frameCount );

        // Capture the screenshot to the specified file.

        Application.CaptureScreenshot(name);

    }

}
```

Although the video recorded using this technique typically looks very good, the game can be hard to play when slowed-down drastically. You may need to experiment with the value of Time.captureFramerate to allow ample recording time without unduly complicating the task of the test player.

Creating and Destroying GameObjects

Some games keep a constant number of objects in the scene, but it is very common for characters, treasures and other object to be created and removed during gameplay. In Unity, a GameObject can be created using the Instantiate function which makes a new copy of an existing object:

```
public GameObject enemy;

void Start() {

    for (int i = 0; i < 5; i++) {

        Instantiate(enemy);

    }
```

```
}
```

Note that the object from which the copy is made doesn't have to be present in the scene. It is more common to use a prefab dragged to a public variable from the Project panel in the editor. Also, instantiating a GameObject will copy all the Components present on the original.

There is also a Destroy function that will destroy an object after the frame update has finished or optionally after a short time delay:

```
void OnCollisionEnter(Collision otherObj) {

   if (otherObj.gameObject.tag == "Missile") {

      Destroy(gameObject,.5f);

   }

}
```

Note that the Destroy function can destroy individual components without affecting the GameObject itself. A common mistake is to write something like:

```
 Destroy(this);
```

…which will actually just destroy the script component that calls it rather than destroying the GameObject the script is attached to.

Important Classes

These are some of the most important classes you'll be using when scripting with Unity. They cover some of the core areas of Unity's scriptable systems and provide a good starting point for looking up which functions and events are available.

Class:	***Description:***
MonoBehaviour	The base class for all new Unity scripts, the MonoBehaviour reference provides you with a list of all the functions and events that are available to standard scripts attached to Game Objects. Start here if you're looking for any kind of interaction or control over individual objects in your game.
Transform	Every Game Object has a position, rotation and scale in space (whether 3D or 2D), and this is represented by the Transform component. As well as providing this information, the transform component has many helpful functions which can be used to move, scale, rotate, reparent and manipulate objects, as well as converting coordinates from one space to another.
Rigidbody / Rigidbody2D	For most gameplay elements, the physics engine provides the easiest set of tools for moving objects around, detecting triggers and collisions, and applying forces. The Rigidbody class (or its 2D equivalent, Rigidbody2D) provides all the properties and functions you'll need to play with velocity, mass, drag, force, torque, collision and more.

UI

The UI system allows you to create user interfaces fast and intuitively. This is an introduction to the major features of Unity's UI system.

Canvas

The **Canvas** is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.

Creating a new UI element, such as an Image using the menu **GameObject > UI > Image**, automatically creates a Canvas, if there isn't already a Canvas in the scene. The UI element is created as a child to this Canvas.

The Canvas area is shown as a rectangle in the Scene View. This makes it easy to position UI elements without needing to have the Game View visible at all times.

Canvas uses the EventSystem object to help the Messaging System.

Draw order of elements

UI elements in the Canvas are drawn in the same order they appear in the Hierarchy. The first child is drawn first, the second child next, and so on. If two UI elements overlap, the later one will appear on top of the earlier one.

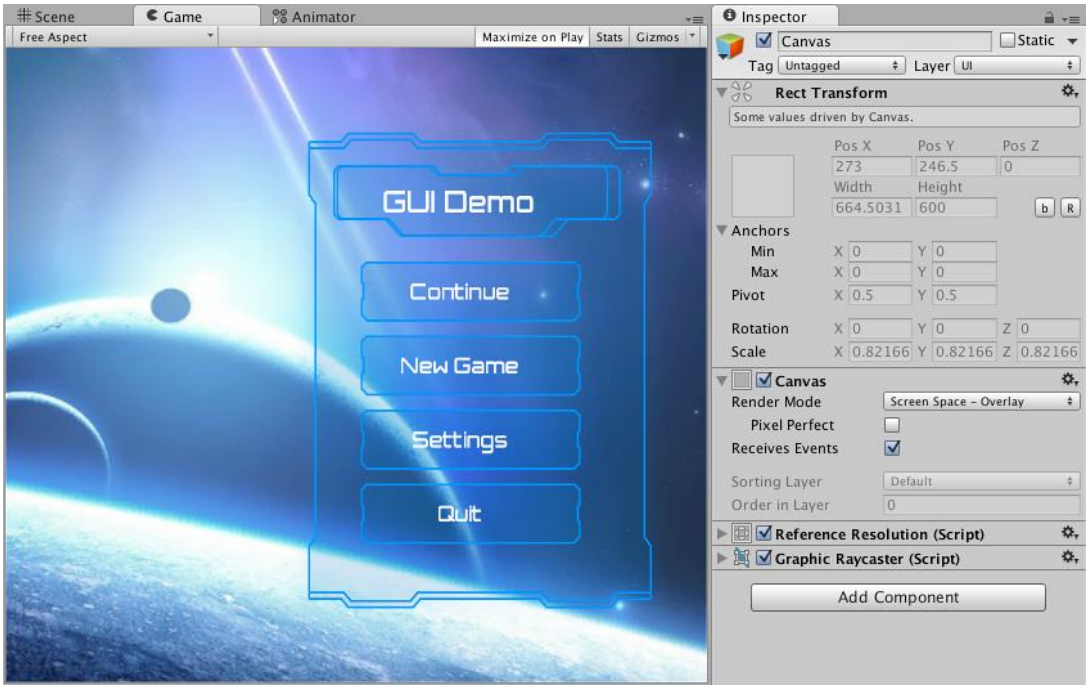
To change which element appear on top of other elements, simply reorder the elements in the Hierarchy by dragging them. The order can also be controlled from scripting by using these methods on the Transform component: SetAsFirstSibling, SetAsLastSibling, and SetSiblingIndex.

Render Modes

The Canvas has a **Render Mode** setting which can be used to make it render in screen space or world space.

Screen Space - Overlay

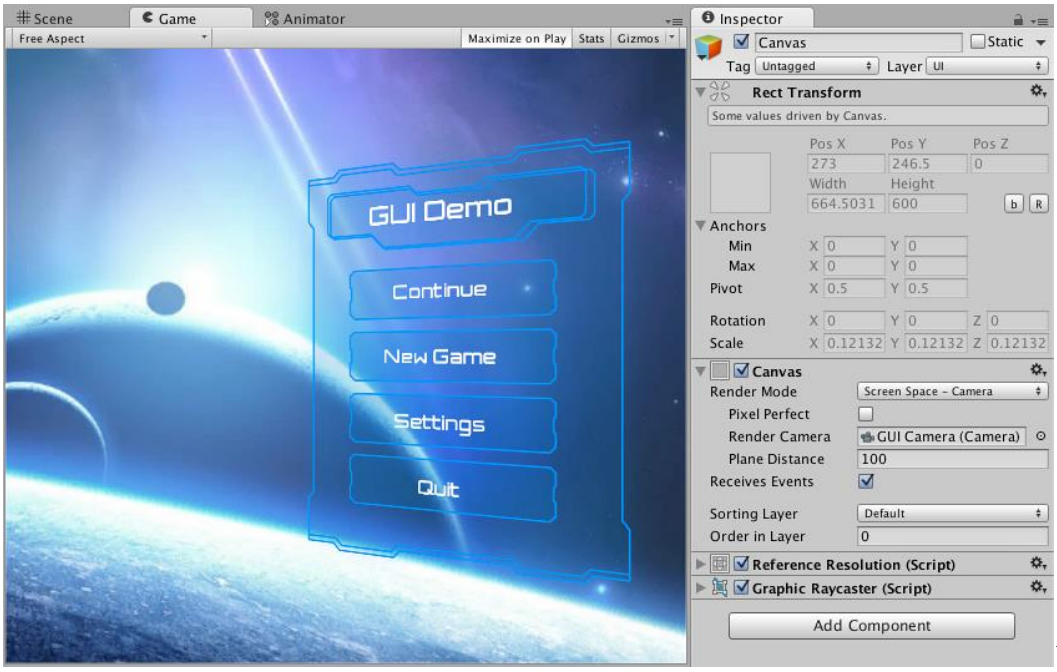
This render mode places UI elements on the screen rendered on top of the scene. If the screen is resized or changes resolution, the Canvas will automatically change size to match this.



UI in screen space overlay canvas

Screen Space - Camera

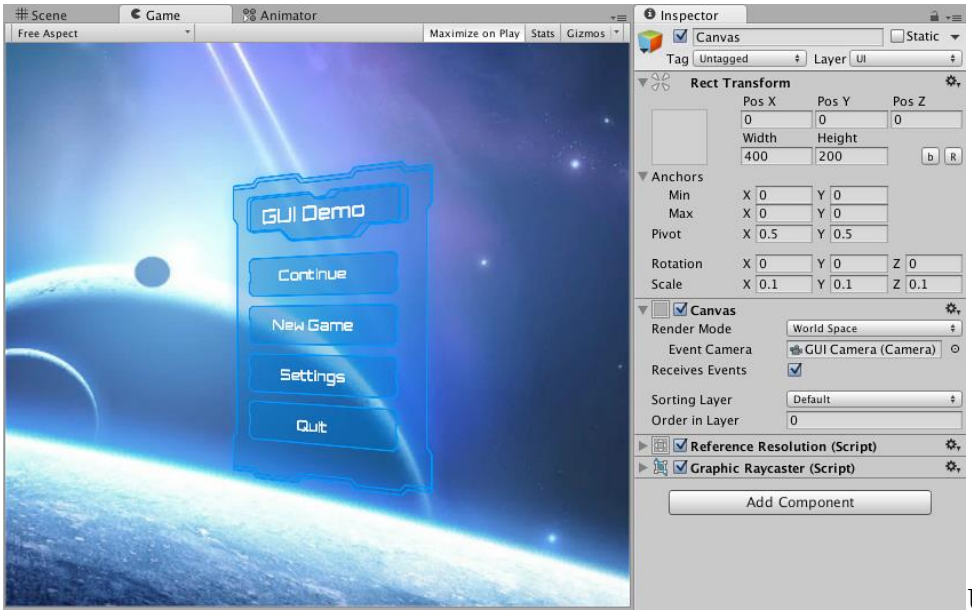
This is similar to **Screen Space - Overlay**, but in this render mode, the Canvas is placed a given distance in front of a specified **Camera**. The UI elements are rendered by this camera, which means that the Camera settings affect the appearance of the UI. If the Camera is set to **Perspective**, the UI elements will be rendered with perspective, and the amount of perspective distortion can be controlled by the Camera **Field of View**. If the screen is resized or changes resolution, or the camera frustrum changes, the Canvas will automatically change size to match as well.



UI in screen space camera canvas

World Space

In this render mode, the Canvas will behave as any other object in the scene. The size of the Canvas can be set manually using its Rect Transform, and UI elements will render in front of or behind other objects in the scene based on 3D placement. This is useful for UIs that are meant to be a part of the world. This is also known as a "diegetic interface".



UI in world space canvas

Interaction Components

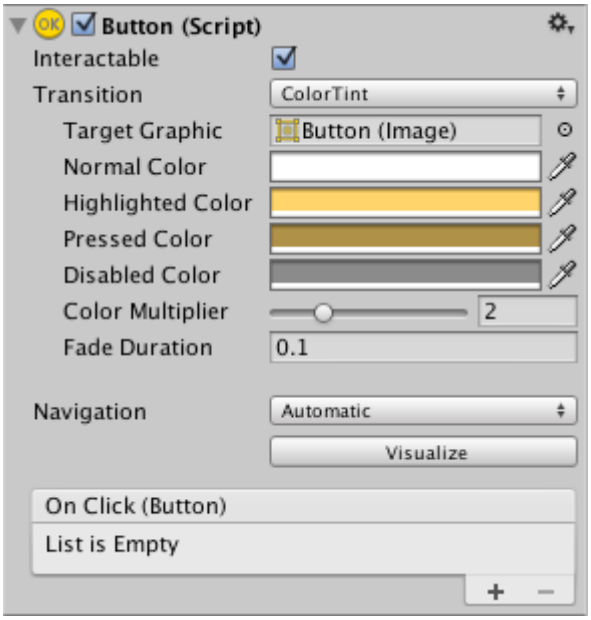
The interaction components in the UI system handle interaction, such as mouse or touch events and interaction using a keyboard or controller.

- Selectable Base Class
- Button
- Toggle
- Toggle Group
- Slider
- Scrollbar
- Scroll Rect
- InputField

Button

The **Button** control responds to a click from the user and is used to initiate or confirm an action. Familiar examples include the *Submit* and *Cancel* buttons used on web forms.

A Button.



Properties

Property:	*Function:*
Interactable	Will this component will accept input? See Interactable.
Transition	Properties that determine the way the control responds visually to user actions. See Transition Options.
Navigation	Properties that determine the sequence of controls. See Navigation Options.

Events

Property:	*Function:*
On Click	A UnityEvent that is invoked when when a user clicks the button and releases it.

Details

The button is designed to initiate an action when the user clicks and releases it. If the mouse is moved off the button control before the click is released, the action does not take place.

The button has a single event called *On Click* that responds when the user completes a click. Typical use cases include:

- Confirming a decision (eg, starting gameplay or saving a game)
- Moving to a sub-menu in a GUI
- Cancelling an action in progress (eg, downloading a new scene)

ESCAPE FROM PERFECTION

INTRODUCTION

Escape from perfection is a 2D puzzle platform game containing 4 levels. The player has to move around and collect power-ups, dodge from enemy objects like the zombies and boulders, make logical use of the crates available and reach the time machine with sufficient power to reach/unlock the next level.

The home screen of the game on a desktop

Some of the notable facts about the game are:

- The game was developed using the Unity Game Engine.
- The game related codes were written in C#.
- The game has button UI to interact with the player.
- Each game situation has a different background music.
- The game has 4 levels.
- The game was downloaded for more than 100 times from the Google PlayStore with an average rating of 5.0(on a scale of 5).

FEASIBILITY STUDY

A game feasibility study is a formal project proposal used to secure internal or external funding and resources for a game development project. It is designed to assess the business and technical potential/problems of the proposed project i.e. can we make it, if so can we make at a profitable level. After the study, the game project is either further developed or cancelled. As a game feasibility study takes time (and therefore, money) to do correctly, it should only be developed for promising game concepts.

The game feasibility study includes the following features:

- Revised game concept
- Market analysis
- Technical analysis
- Legal analysis
- Cost and revenue projections

Revised game concept:

The objective of this game project is to create a thought-provoking 2D puzzle platformer game.The game includes an animated 2D sprite which is controlled manually to dodge the obstacles and solve the puzzles embedded in the world.The game is won when all the puzzles are solved and the player remains alive.

Market analysis:

Target market: This game project has already proposed to develop the game to work in both PC's and Android devices. The game mainly focuses on the age group 10 to 25 Years. Thus, the target market is huge.

Technical analysis:

Major development tasks:

- Design four levels of most innovative puzzles
- Cross Platform (minimum in two forms)
- Develop the Player and Enemy scripts

- Testing and Deployment

Estimated resources:

- Software: Open source Unity 3D software
- Hardware: Any Computer which can run the Unity game engine.(A general PC can easily run Unity without much difficulty)
- Manpower

Estimated Schedule:

Being a 2D game with small complexities, the project has been completed within the stipulated time during the summer break.

Legal Analysis:

No rights have been violated in the game and all the online resources have been credited in the games CREDITS column.

Cost and revenue projections:

No cost involved for the team members.

Resource cost:

Already available

Suggested Retail/online Price:

The game is not for sale as of now and there might not be any charges in the future as well.

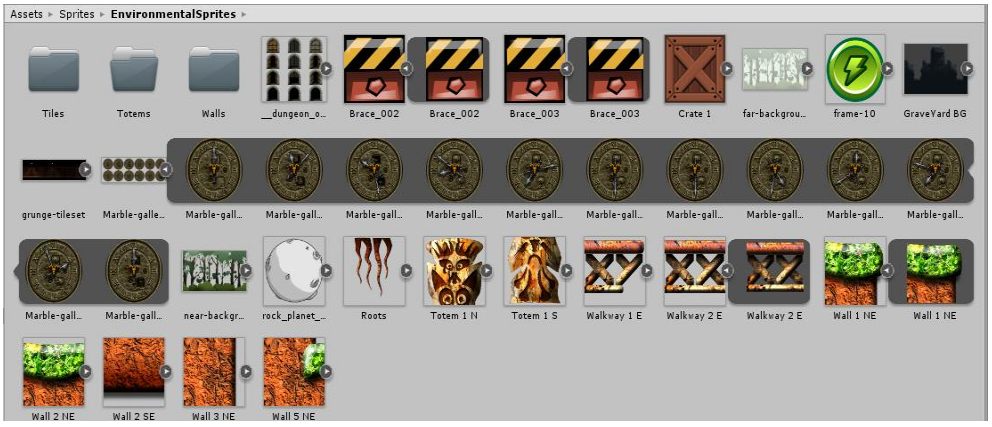
DESIGN OF THE GAME

Being a puzzle platform game, the game has to have increasing difficulty with new unlocked levels. Also the game objects had to look good for the player to get attracted to the game.

Game Objects

The game consists of many game objects such as:

- The Player
- Platforms/Ground
- Moving platform
- Enemy Objects like Zombie and Boulder
- Helping crates/Boxes
- Power-ups
- Time machine
- And many more in-depth objects which cannot be seen by the player.



These are some of the sprites of the game objects which are collectively seen by the player on each and every frame while playing the game.

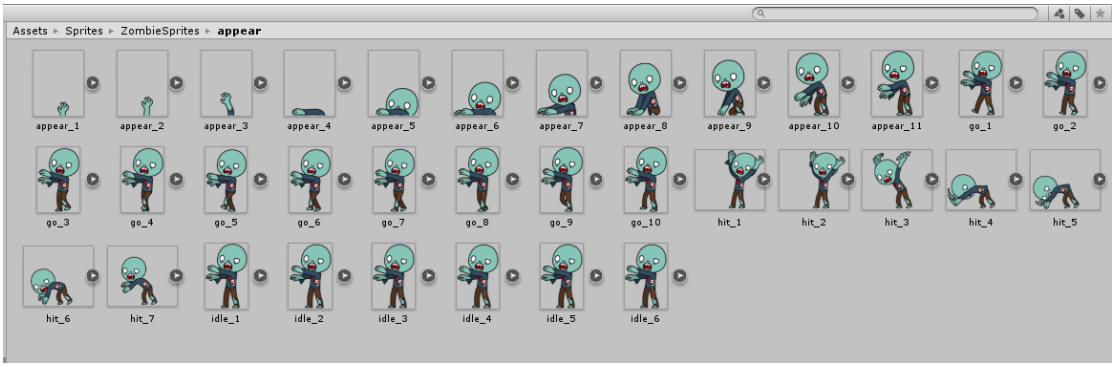
All the sprites of the Game Objects have either been taken from online free sites or have been created using Adobe Photoshop. Credits have been mentioned for the ones who have made the sprites available online.

- The GameObjects wear a look which is similar to the characters of Halloween.

This is because the games concept is related to the survival of a child who is lost in the wild.

- The boy(player) bears a pumpkin head, there is a zombie here and there trying to kill the child, the fullmoon is haunting the child and also the background theme is a graveyard in one of the levels.
- The most important thing any of the game are the sprites of the GameObjects. In this game, the most important character, i.e., the player character has 38 different positions and there is a corresponding graphic for the player object. This differentiation for each position is very important for the character to move along.

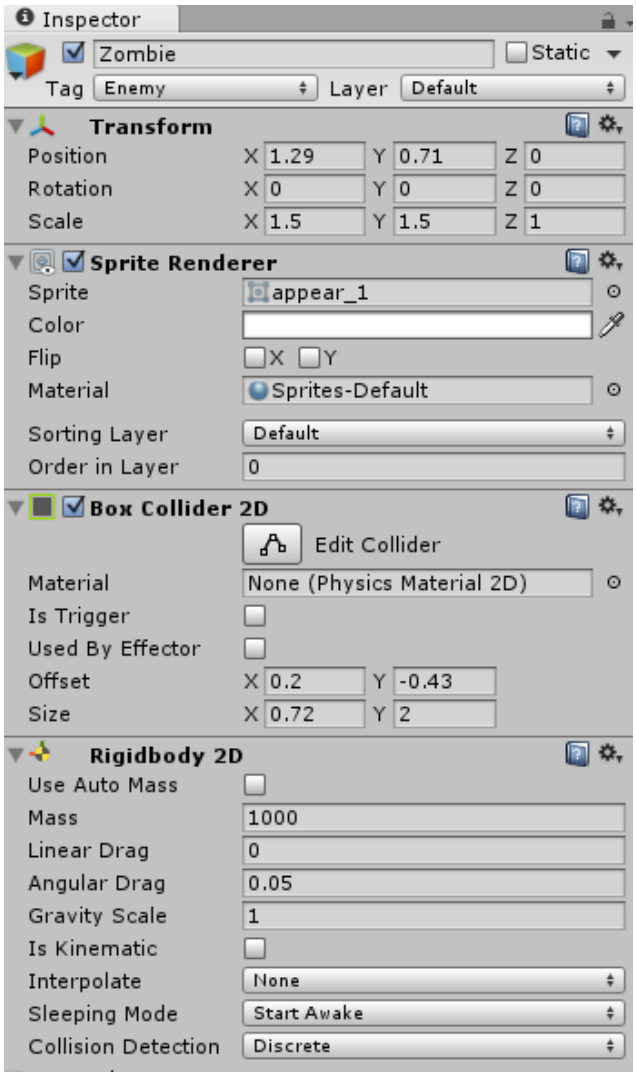
The zombie object has 34 different animations for various actions like spawning, running and changing direction.



Positioning of GameObjects in the Scene

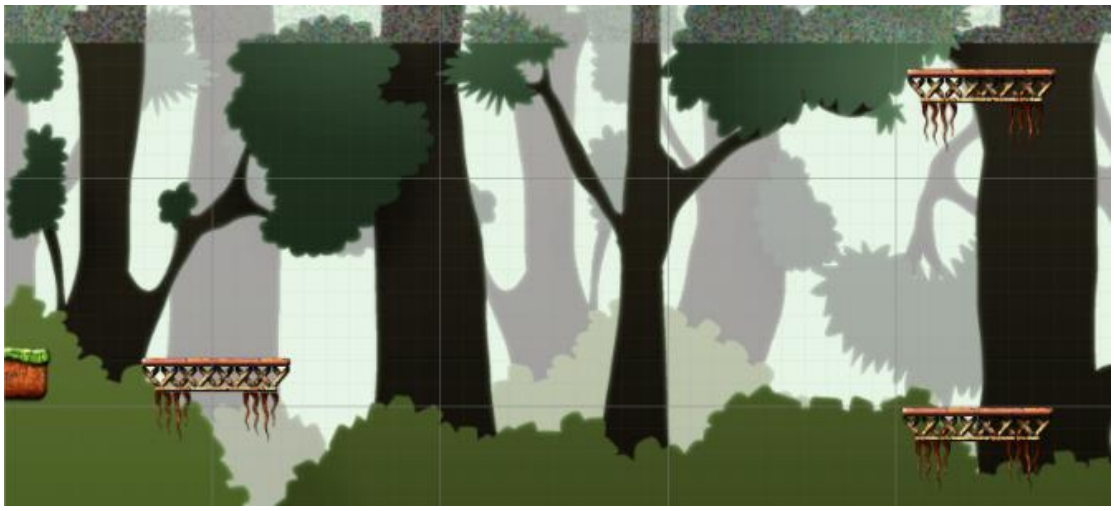
It is very important to follow proper dynamics and position various game objects at the exact locations so that it is possible for the player to perform the tasks.

In order to have proper positioning of each object, we use the Transform values of each object. The movement of each object in the game must be synchronized right through the level. This will be helpful even if the player starts moving after waiting for some time.



This is one of the window that contains all the attribute values of a GameObject which on this instance is the Zombie. The transform block takes care of the positioning, orientation and the scale of the GameObject. The Rigidbody 2D block takes care of the physics part for any kind of motion of the GameObjects.

The synchrnisation part is one of the most important one which involves few calculations regarding timing. We can see the working of various game objects simultaneously but without causing any king of imbalance in the gameplay. For example, there is one situation in the level 4 of the game where 3 of the moving platforms must be synchronized with each other so that the player can jump over them and reach the end of the level.



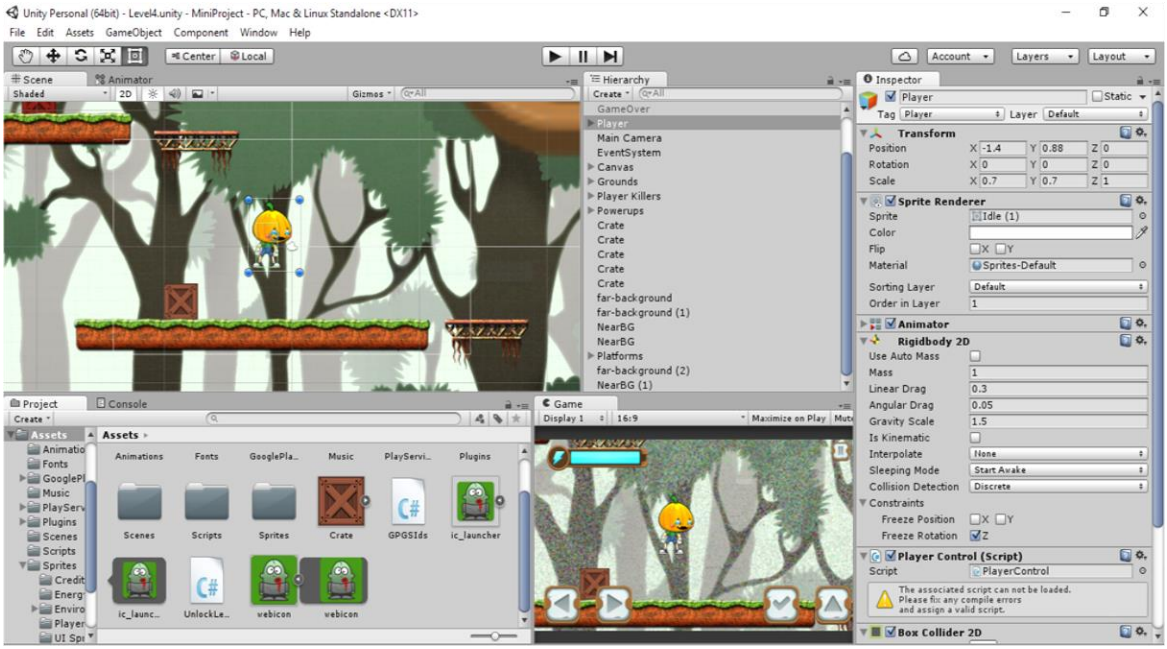
The 3 moving platforms move so that they come close to each other and the player can jump over it.

IMPLEMENTATION

Developing the Player character

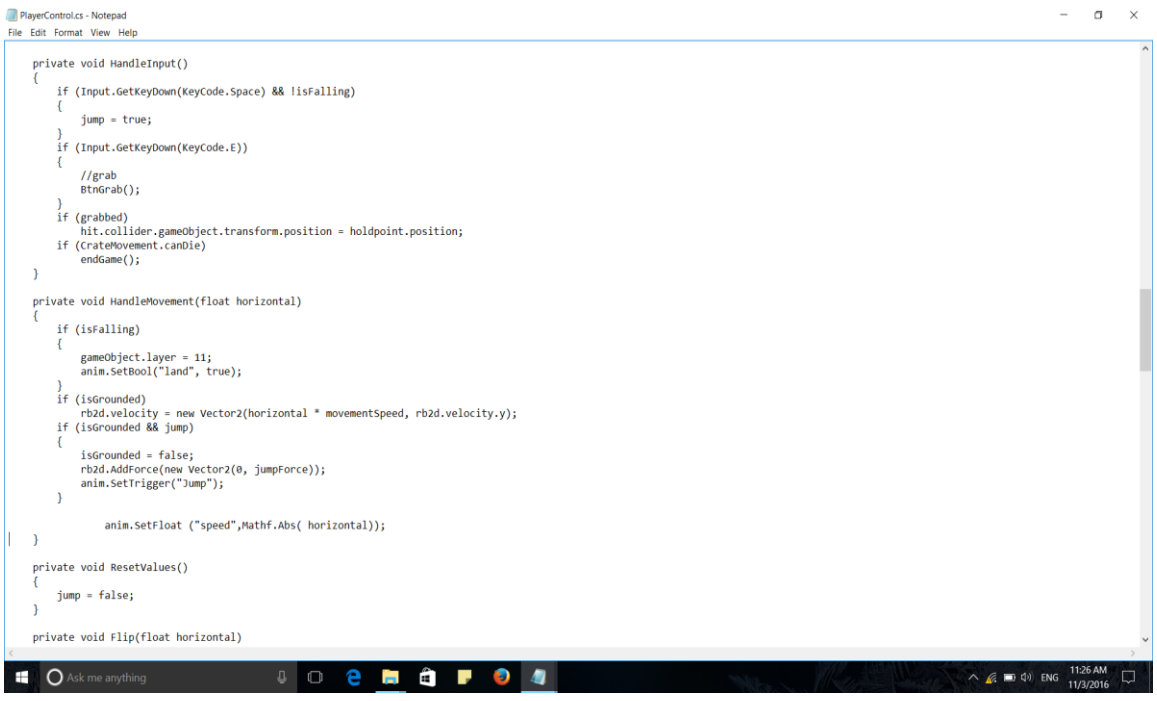
In the game, the player is basically someone who can move around and can jump some good distance. The player can also lift objects like crates, collect some power-ups, and also dies when comes in contact with the enemy objects or falls down from the platforms. So, the script for the player had to be written accordingly for each and every act and the corresponding sprites had to be allocated.

The player also has a few physics related features like speed, gravity, jump-force, and mass which limit the player's capabilities from being a superman.



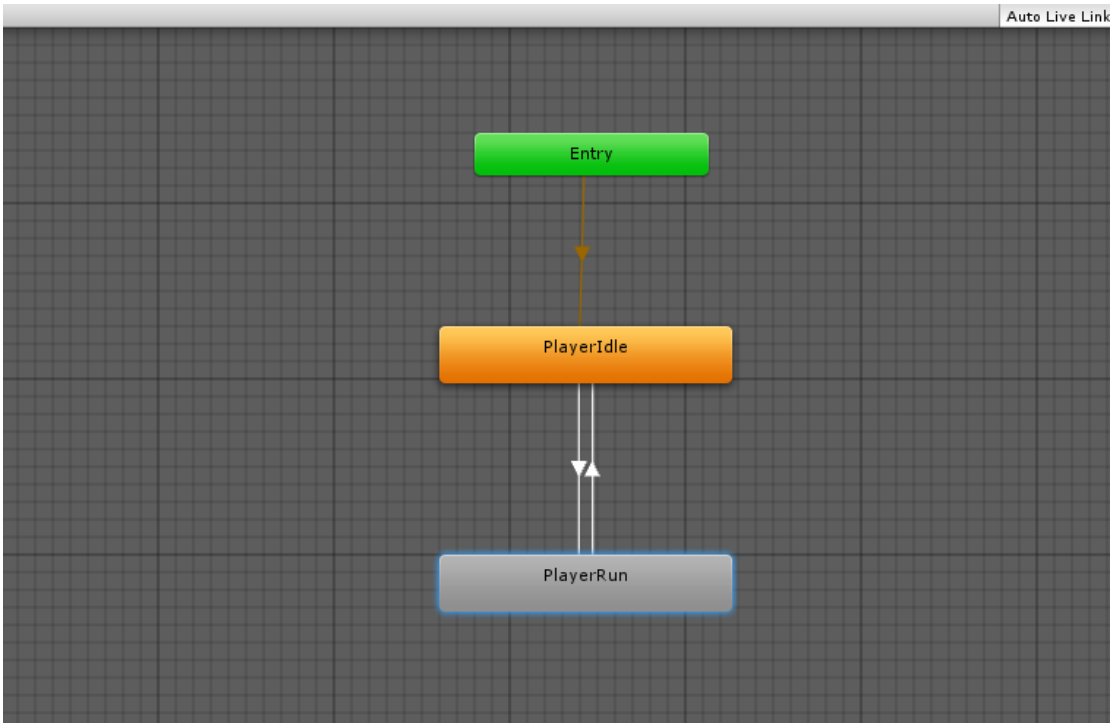
The above picture shows the basic view of how the player looks like while developing its attributes.

Coding the Player scripts



```
private void HandleInput()
{
    if (Input.GetKeyDown(KeyCode.Space) && !isFalling)
    {
        jump = true;
    }
    if (Input.GetKeyDown(KeyCode.E))
    {
        //grab
        BtnGrab();
    }
    if (grabbed)
        hit.collider.gameObject.transform.position = holdpoint.position;
    if (CrateMovement.canDie)
        endGame();
}

private void HandleMovement(float horizontal)
{
    if (isFalling)
    {
        gameObject.layer = 11;
        anim.SetBool("land", true);
    }
    if (isGrounded)
        rb2d.velocity = new Vector2(horizontal * movementSpeed, rb2d.velocity.y);
    if (isGrounded && jump)
    {
        isGrounded = false;
        rb2d.AddForce(new Vector2(0, jumpForce));
        anim.SetTrigger("Jump");
    }

            anim.SetFloat ("speed",Mathf.Abs( horizontal));
}

private void ResetValues()
{
    jump = false;
}

private void Flip(float horizontal)
```

The above code shows how the player responds to the various inputs given to the game by the user. It is generic for both touch and keyboard.



The player is in any of the 3 states which are entry, PlayerIdle and PlayerRun states. Based on the currunt state of the player object, the corresponding code is executed as well as the related background music is also played. The player can die either in PlayerIdle state or the PlayerRun state.

Developing Enemy characters

There are 2 enemy characters, namely, the Zombie and the Boulder which are spawned when the player approaches them. While the boulder can move only in one direction and dies after falling down, the zombie can run around the particular platform to kill the player.

These 2 enemy objects are difficult to evade from as the player can’t jump over them due to their larger size.

The boulder starts rolling just when the player comes closer to it. This demands the player to use his reflexes to escape from the boulder or else, he will be rolled over.

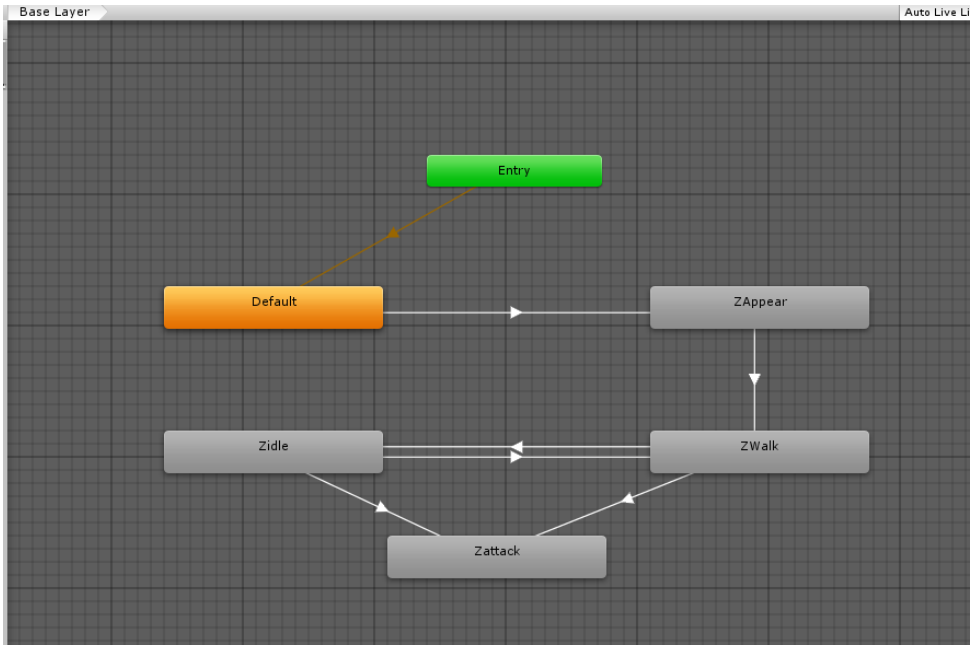
Below is a script which describes the Boulder’s actions at various stages in a piece of code.

Boulder.cs

Boulder ▸ onSpawn ()

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class Boulder : MonoBehaviour
5 {
6     [SerializeField]
7     private Transform player;
8     [SerializeField]
9     private float enemySpeed, spawnDistance;
10    private bool spawned;
11    private Rigidbody2D rb;
12
13    void Start()
14    {
15        rb = GetComponent<Rigidbody2D>();
16    }
17    void FixedUpdate()
18    {
19        if (!GameManager.Instance.Paused)
20        {
21            if (onSpawn() && !spawned)
22            {
23                rb.velocity = new Vector2(-enemySpeed, rb.velocity.y);
24                spawned = true;
25            }
26        }
27    }
28    private void OnCollisionEnter2D(Collision2D collision)
29    {
30        if (collision.gameObject.name == "Player") enemySpeed = 0;
31    }
32    private bool onSpawn()
33    {
34        if (Mathf.Abs(transform.position.x - player.position.x) < spawnDistance &&
35            Mathf.Abs(transform.position.y - player.position.y) < 1)
36            return true;
37        return false;
38    }
39 }
```

The zombie is a bit different from the boulder. The zombie, after spawning, moves around the platform and the only way for the player to stop it is to throw a crate in front of him. The zombie doubles its speed when the player is in front of it.



Other aspects of the game

The UI of the game is one of the most important things to focus. There are a lot of buttons on offer for the user to control the game like pause and quit etc. As soon as you see the UI of the game, you get the first impression for it. There are many good looking buttons which make the look of the game even better and can give a good feel to the users. The buttons used in the game are shown below:

One of the other important feature for a game is to have an efficient loading screen which can show the amount of game loaded to the user. This game has a loading screen which displays the loaded percentage on a white strip with black background.

loadingBarManager.cs

No selection

```
1 using UnityEngine;
2 using System.Collections;
3 using UnityEngine.UI;
4
5 public class loadingBarManager : MonoBehaviour
6 {
7     public static float percentLoaded;
8
9     void Start ()
10    {
11        percentLoaded = 0f;
12    }
13
14    void Update ()
15    {
16        if (percentLoaded < 0 || percentLoaded > 100) return;
17        gameObject.GetComponent<Image>().fillAmount = percentLoaded / 100f;
18    }
19 }
20
```

TESTING THE GAME

This chapter includes some test case for the game to check if the game works properly in various situations. We are giving six test examples for six different situation here.

Test Case 1:

Test Case : This test will check if the animation for characters is working correctly.

Test Procedure : Import a character model with animation in unity. Place character on the scene. Run the game.

Expected Result : Animation works perfectly in the environment.

Actual Result : Animation is not working.

Comment : Need to check character configuration on inspector window. The appropriate animation was not selected. Select it.

Conditional Test : Again run scene.

Expected Result : Animation is working now.

Actual Result : Yes, it is working.

Test Case 2:

Test Case : This test will check if the interaction between object and script working correctly.

Test Procedure : Add scripts of interaction in the objects that we want to interact with each other. Run scene.

Expected Result : Objects are interacting.

Actual Result : Run time exception

Comment : Need to add checking in the scripts for the objects that have a particular script.

Conditional Test : Again run scene.

Expected Result : Interaction is working now.

Actual Result : Interaction is working now.

Test Case 3:

Test Case : This test will check if the enemy object initiates and follows the main character, when main character is near to them.

Test Procedure : Import a main character and enemy object model with animation in unity. Place characters on the scene. Run the game.

Expected Result : Enemy object initiates and follows the main character when main character is near to them.

Actual Result : Working Perfectly.

Comment : Enemy object can recognize main character's presence.

Test Case 4:

Test Case : This test will check if the level flow works along with loading screen.

Test Procedure : Run the initial scene and choose the flow of the game and check the loading screen appearance.

Expected Result : Synchronization in between loading screen, main screen and four levels.

Actual Result : Loading screen scene is not running at the end of the second level.

Comment : Need to add Loading screen scene at the end of second level and then connect it with third level.

Conditional Test : Again run scene.

Expected Result : Flow of scenes is working perfectly.

Actual Result : Flow of scenes is working perfectly.

Test Case 5:

Test Case : This test will check if the buttons in the game working correctly.

Test Procedure : Run the game to check all buttons working status.

Expected Result : All buttons are working as per the requirements.

Actual Result : There is no back button to come back to Main menu from About.

Comment : Add a button to come back to Main menu from About.

Conditional Test : Again run game.

Expected Result : All buttons are working as per the requirements.

Actual Result : All buttons are working as per the requirements.

Test Case 6:

Test Case : This test will check if the Leaderboards and Achievements are working correctly.

Test Procedure : Run the game. Try to accomplish specified achievements and complete the last level to check leaderboards.

Expected Result : Animation works perfectly in the environment.

Actual Result : Animation is not working.

CONCLUSION AND FUTURE WORK

In the game Escape From Perfection, we have implemented a game environment that includes 2D viewing and objects with a moving camera, platforms, enemy objects, achievements and leaderboards. To enhance the game environment we have also implemented advanced features such as parallaxing, collision detection and sound effects using different Unity libraries available. The key feature of our game are enemy objects, hindering items, energy bars and background music makes the game more challenging but yet interesting and enjoyable to play with.

In this section we summarize the experience gained our project team during development of “Escape From Perfection”.

8.1: The Obstacles

1. Working with unity game engine completely a new experience for us. Normally, we are working with different programming languages, query languages, web development, etc.

2. We adopted to user interface by video tutorials, text tutorials given in the tools section of Unity.

3. Knowledge about properties, inbuilt classes and packages and also as script is completely written in C#, we learned the basics of C# through online course..

4. As this is a logical game we had to think innovatively for designing the levels to make them interesting as well as frustrating ;)

8.2: The Achievements

1. Now, we have vivid idea about the game engines and game development.

2. How to generate sprites and how it is animated.

3. Had a chance to work as a Designer, Developer, Tester, Presenter

4. Team Work

5. Develop communication skills

6. Creative thinking

8.3 Future Work

- Level Extension.
- Resolving bugs reported by the users
- Set difficulty modes
- Introduce new features
- Reduce game size
- Promote Android App of the game (presently with 100+ downloads)
- Publishing in iOS store and windows store

BIBILOGRAPHY

Books Used:

Unity 5 Game Optimization.

Unity Multiplayer Games.

Learning C# by Developing Games With Unity3D Beginners Guide.

Sprites :

https://docs.unity3d.com/Manual/SpriteEditor.html

http://www.adventurecreator.org/tutorials/using-sprites-2d-backgrounds-unity-2d

http://picturetopeople.org

http://bevouliin.com

Sprites be Irmirx and Julien Jorge

Communication:

https://www.teamviewer.com/en/ (for interacting and files sharing)

Background music:

https://creativecommons.org/ -> Damnnation by Jens Kiilstofte

References:

Mastering Unity 2D Game Development – by Ashley Godbold(Author)

.Unity: Developing Your First Game with Unity and C# Adam Tuliper