Notes - Unity - Basic Usage

- Dr Nick Hayward

A guide to basic use of the Unity IDE, game assets, game objects &c.

Contents

Intro

As we start to design and develop Unity based games, there are various concepts that need to be considered. For example,

- asset management and usage
- GameObjects and Prefabs
- cameras and projections
- physics
- materials

Asset management

A common part of game development with Unity is management and correct use of assets. For example, this might include C# scripts, music, textures, prefabs, models &c.

Each of these may be stored in the game's ***Assets*** directory, as shown in the IDE's ***Project*** pane.

As we add such assets, perhaps by dragging and dropping an existing set of folders, we'll see icons for each asset in the ***Project*** pane. We may also modify the view to smaller icons or a list.

We may also prefer to organise such assets into sub-directories for each of the above defined concepts. For example, a separate folder for `models`, `scripts` &c.

scenes and saving

Unity saves current development work in scenes. It's important to correctly define such scenes as Unity does not offer a default autosave feature for the current project.

So, we may start by defining an initial game *scene*, which we'll add to a new folder in the `Assets` of the game.

```
.
|-- Assets
|   |-- Scenes
```

We can also define an appropriate name for this new scene, e.g. `MainScene` or perhaps `GameScene` &c.

GameObjects and Prefabs

To build many games, we'll need a main character with a defined object.

objects

For 3D games, We might create or import a **FBX** file for the required model. This file includes the required metadata to create the 3D model, plus any associated animations.

Some object images also include extra information specific to Unity, although they are saved with a standard image file extension. For example, such images may include metadata for surface depths. Such information defines required lighting without the extra overhead of specific geometry. This is known as a **normal map**.

n.b. in some instances, Unity may perceive such image files as regular images - use the **Fix Now** option on import to define as a **normal map**...

Once an image asset has been correctly added to the project, we may then drag it into the current game *scene*. This will create an instance of a model for the game object.

The *Hierarchy* pane will also be updated to show this new game object.

customise objects

We may also customise these game objects by adding any required **Components**.

For example, we might add a given lighting component to help define and illuminate our current game object. Likewise, we can add additional components for sounds, and define any custom scripts as well.

prefabs

Prefabs allow a developer to easily duplicate existing GameObjects in a game scene.

Such objects are identified by a distinct blue colour in the *Hierarchy* pane, and this identifies a connection to other objects.

All instances of a prefab are also inter-connected. So, if we change the *prefab*, the connected instances will also be modified. In effect, change the original, and the copies are modified as well.

However, any instances of a prefab may be customised as necessary to fit the current game. Unity offers three valuable management options for Prefabs and instances.

- select - select and highlight prefab object relative to current instances
- revert - revert an instance to the state of the original prefab
- open - opens the model file in the original design tool, e.g. Blender...

Cameras and Projections

Camera usage is a key part, along with defined physics, in allowing a player to control a given game object.

To get started with camera setup, we need to select a given game object in the *Hierarchy* pane, and then use the *Inspector* tools to update its position.

For a 3D object, we may modify its position relative to the X, Y, and Z axes.

Likewise, we may also modify its rotation relative to the same axes.

Scale is also available as part of these *Transform* options.

As we modify such values, we may find that the relative position of the game object is either incorrect or inappropriate for the current scene. For example, the object might appear too close to the scene view.

We may fix this issue, for example, either by effectively pushing the object back in the scene, or through modification of the **camera**.

initial camera usage

To fix this type of issue, we may select the *Main Camera* in the *Hierarchy* pane.

The *camera component* will now become available in the *Inspector*.

There are many options available for the camera. However, we may initially focus upon *Projection* and *Field of View*.

Projection offers two options, including *Perspective* and *Orthographic*. Either option will affect the render scene appearance of the game.

For example, *Perspective* is akin to the view of the eye, which will perceive close objects as larger. However, an *Orthographic* projection will discard such depth information. In effect, the relative size of an object will remain the same regardless of perceived depth.

So, which do we normally choose? For most games, we may use the following rules of thumb

- 3D game - commonly uses a *Perspective* projection
- 2D game - *Orthographic*

After selecting the required *Projection*, we may consider the size for the *Field of View*, perhaps adding a value of *17.55*.

Physics

As we add motion, for example, to game objects, we may find that we defy expected best practices for physics usage. i.e. we start breaking the expected laws of physics for motion such as flying a plane.

So, we might consider adding physics for collision detection.

basic collision detection

To add collision detection to a game object in Unity, we need to consider applicable physics.

For example, if a game object does not have a defined *RigidBody* component, Unity will consider an object as a *static collider*. i.e. the object is assumed to be static, and non-moving in the scene.

This is beneficial to rendering and processing the game, so it's important to define appropriate physics for game objects.

For moving objects, therefore, we may use Unity's physics engine by adding a *RigidBody* component.

RigidBody options

A *RigidBody* component presents numerous config options, including common options such as *Is Kinematic* and *Use Gravity*.

Selection of either option will depend upon the type of game being developed, and the specific use of physics relative to the selected game object.

colliders

For many games, we will also need to consider colliders for various game objects.

Colliders provide listeners for collisions within the game, and allow customised responses.

Unity provides various collider types, which we may use with basic shapes. For example, we might add a *box collider* and *sphere collider*. We may also consider a *mesh collider* relative to a defined 3D model.

n.b. colliders may be shown with a green outline in a Unity scene.

After adding a required collider for the selected game object, we may then customise its position relative to the x, y, and z axes, and perhaps update the radius for a *sphere collider*.

References

- Unity - https://unity.com/
- Unity documentation - https://docs.unity3d.com/Manual/index.html
- Unity tutorials - https://unity3d.com/learn/tutorials