Analytics

> Read the code in the Examples Folder.

Introduction

Analytics attempts to record and summarise the data you need from game-play to manage your relationship with your players. It is useful to know when your players get stuck or give up at a particular level. Some packages support many platforms, some only a few. They all report results in different ways.

Networks

Unity Analytics

Unity Analytics is the easiest to implement for a Unity3D project. Also, it supports the same broad range of target platforms as Unity itself.

Preparation

Install the Askowl-Analytics package. If not using Unity Analytics you will also need to install the .unitypackage file from the network site.

Drag an instance of ***Askowl/Analytics/Prefabs/AnalyticsXXX*** where ***XXX*** matches the name of the analytics networks you intend to use.

Accessing a Single Network

If you choose to cache the analytics reference, do so in `Start`. The connection between the interface and concrete API are in `Awake`.

```
public class MyComponent : MonoBehaviour {
  private Analytics log;

  private IEnumerator Start() {
    log = Analytics.Instance;
  }
```C

## API
### Event(name, action, result, more...)
`Event` is the core function that writes a custom event to the analytics network. It has three fixed parameters and an unlimited number of additional ones.

* `name`: is the name of the event. For most analytics systems it has special meaning. Often it is set to the subsystem, area of code or scene name.
* `action`: Defines what the game or code was doing at the time.
* `result`: Is the result of the action. *Started*, *Completed*, *Success* and *Failure* are typical examples.

It makes sense to create functions in code associated with the name. Name it after the action and provide the result and additional parameters.

```C#
class Character : MonoBehaviour {
  private void LogRoleChange(string result, params object[] more) {
    log("Character", "Role Change", result, more);
  }

  private void Update() {
    if (roleChanged and error) LogRoleChange("Failure", "Reason:", reason);
  }
}
```

Error(name, message, more...)

`Error` is a custom event with an action of ***Error*** and where the result is a message.

More(list...)

`More` is a helper function that takes a variable length parameter list and returns a string with comma separated text representations. Use `More` when the analytics network does not support a dictionary of additional parameters.

ToDictionary(list...)

`ToDictionary` is another helper method. This time it returns a dictionary with `string` keys and `object` values. Since the list passed in is an array of objects, some interpretation is necessary. If a key ends in a colon, the next item in the list becomes the value. Otherwise, the value is empty, and the next item will be the next key.

```
var log = Decoupled.Analytics.Instance;
log.Event("State","Election","Started","Region:", "Townsville", "Open",
"Prepared", "Population:", 429344);
```

Gender

Set the player gender in the analytics network. `Authentication` triggers the event.

```
   public override string Gender {
     set {
       switch (value) {
         case "Male":
           Analytics.SetUserGender(UnityEngine.Analytics.Gender.Male);
           break;
         case "Female":

Analytics.SetUserGender(UnityEngine.Analytics.Gender.Female);
           break;
         default:

Analytics.SetUserGender(UnityEngine.Analytics.Gender.Unknown);
           break;
       }
     }
   }
```

BirthYear

Set the player birth year in the analytics network. `Authentication` triggers the event.

```
   public override int BirthYear {
     set { Analytics.SetUserBirthYear(value); }
   }
```

Adding an Analytics Network

Let's say you are going to create an interface to Fabric.

Firstly we must find out whether the Fabric Unity package exists. Install Fabric and look for a directory unique to it. Create a script in an ***Editor*** directory called ***DetectFabric.cs***.

```
  using UnityEditor;
  using Askowl;

  [InitializeOnLoad]
  public sealed class DetectFabric : DefineSymbols {
    static DetectFabric() {
      bool usable = HasFolder("Fabric");
      AddOrRemoveDefines(addDefines: usable, named: "FabricAnalytics");
    }
}
```

Next, create a C# script ***AnalyticsFabric.cs***. It is an unusual case where the script file holds two classes.

```
#if FabricAnalytics
  using Decoupled;

  public sealed class AnalyticsFabric : Singleton<AnalyticsFabric> {
    private void Awake() { Analytics.Register<AnalyticsFabricService>();
}
  }

  public sealed class AnalyticsFabricService : Analytics {
    public virtual void Error(string name, string message, params
object[] more) {
      // Only override if a distinct error condition is required
      // Event(name: name, action: "Error", result: message, more:
more);
    }

    public override void Event(string  name, string action, string
result, params object[] more) {
      // call the Fabric method to record a custom event.
      // Use *More* or *ToDictionary* to record the additional
information.
    }

    public override string Gender {
      set {
        switch (value) {
          case "Male": /* tell Fabric */ break;
          case "Female": /* tell Fabric */ break;
          case "Other": /* tell Fabric */ break;
          case "Unknown": /* tell Fabric */ break;
        }
      }
    }

```

```
    public override int BirthYear {
      set { /* tell Fabric */ }
    }
  }
}
#endif
```

This script has a singleton MonoBehaviour whose sole task is to register the Fabric API. That API class exists in the same file because it is never needed elsewhere. Unless the editor script can find the Fabric package, No API registration occurs. The preprocessor takes care of that for us.

Create an empty GameObject and rename it to ***Fabric Analytics***. Attach ***AnalyticsFabric.cs*** as a component. Drag the resulting GameObject into a prefabs folder for reuse.

```
```