# DeepThink

## Object Model Refactor

Prepared for OpenSim-dev by Adam Frisby – Last Update on 11/12/2009

## Sponsorship provided by realXtend

Sponsorship of this project is provided by the realXtend Group. Their support in paying to fix broken areas of the OpenSim software is appreciated.

While the realXtend group has been instrumental in defining some use cases where OpenSim fails dramatically, the design of this proposal has been prepared by me (Adam Frisby) in the interests of improving OpenSim for supporting outside developers, including but not limited to the realXtend group.

Importantly, the proposal defined here should help all developers working with the OpenSimulator platform.

## Contents

# We begin with a humble proposal

*Wherein the current* `SceneObjectMess` *is transmogrified into something a fraction less horrible.*

## Foreword

Currently, there are several areas of the Scene Object description model employed by OpenSimulator where limitations are visible – to the core project, the internal formatting and layout is anything but programmer friendly. Manipulating Scene Objects from region modules is a recipe for disaster and is often best avoided.

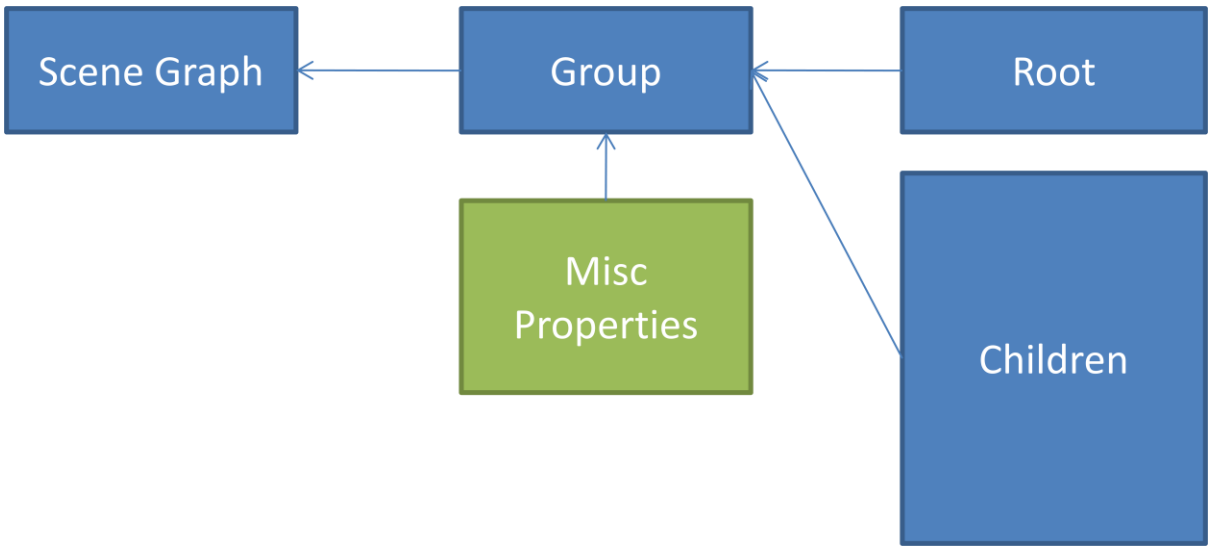
To outside projects, the Scene Object model is very tightly coupled to the LLUDP stack – possibly the last remaining area (except maybe Parcels) where such tight coupling is evident. The inability to extend the Scene Object Group limits functionality improvements by outside groups – since inheriting and extending the properties is not possible from a serialisation perspective.

It is possible to do mind you – realXtend for instance maintains a separate Scene Graph for their extended properties; but interfacing with Inventory, Region Storage, etc is difficult and problematic.

In addition to these – there is also a desire to support hierarchical linking which this proposal can support.

## The Proposal

Objects right now consist of two components – Part & Group. Group is the Entity within scene itself, and may be constructed from a number of sub-“parts”. Part generally contains the properties of the object in a visual sense, while group contains meta-data describing the collection as a whole.




0-1 Scene Objects "Today”

There are two major changes occurring as part of this proposal – first, `SceneObjectGroup` becomes depreciated in favour of `SceneObjectPart` containing other `SceneObjectPart`s. The Second is `SceneObjectPart` will gain the ability to attach in 'components'.

In the interests of getting this done with a minimal of breakage (or more correctly; avoiding everything breaking at once, and instead having a break-over-six-months), we will implement Components first; then move various clumps of SOG/SOP functionality into those components before commencing the merger of SOP+SOP=SO.

With that in mind, I present:

## Phase 1: Implementing Components

Components are C# Types implementing the `IComponent` interface – `IComponent` carries a pair of property-method definitions and can be defined here simply as:

```
public interface IComponent
{
    Type BaseType { get; }
    ComponentState State { get; }
}
```

`BaseType` is the recognised type of the component – for instance there may be multiple modules registering components for a hypothetical `IMeshComponent`. Each SceneObject may only have a single component of each type registered.

`State` is a version of the current state of the component. ComponentState will internally be a form of Dictionary, to the user it will appear to be something like `Set(string,object)`, `Get<T>(string,T)` – all Objects stored *must* be `[Serializable]`. State will be stored with any serialisation of the parent object, indexed by the BaseType's type name, this will be restored via the Factory described below.

Within SceneObject the additional definitions appear like this:

```
public class SceneObject
{
  public T Get<T>();
  public bool TryGet<T>(out T);
  public Set<T>(Object component);
  public bool Contains<T>();
  ...
}
```

In addition, a Factory class is required to implement the `SceneObject` constructor. `SceneObject`'s will be required to construct via the Factory (and *not elsewhere* as they can be presently), this will be to allow Region Modules to implement components at time of construction.

```
public class ObjectFactory {
    public SceneObject Create(ObjectState state);
    public SceneObject Create(String name, Vector3 Position, ...);
    ...
}
```

The factory class is a relatively thin wrapper around the `SceneObject` containing two primary constructors – the first constructor is to restore from a previously serialised state. This state contains all the core properties of the object, plus a collection of `ComponentState` objects associated with it. Upon constructing a class using this constructor, the Factory will proceed to fire an event – roughly summarised as this:

```
public event OnObjectConstructed(SceneObject obj,
                                 Dictionary<Type,ComponentState> componentData);
```

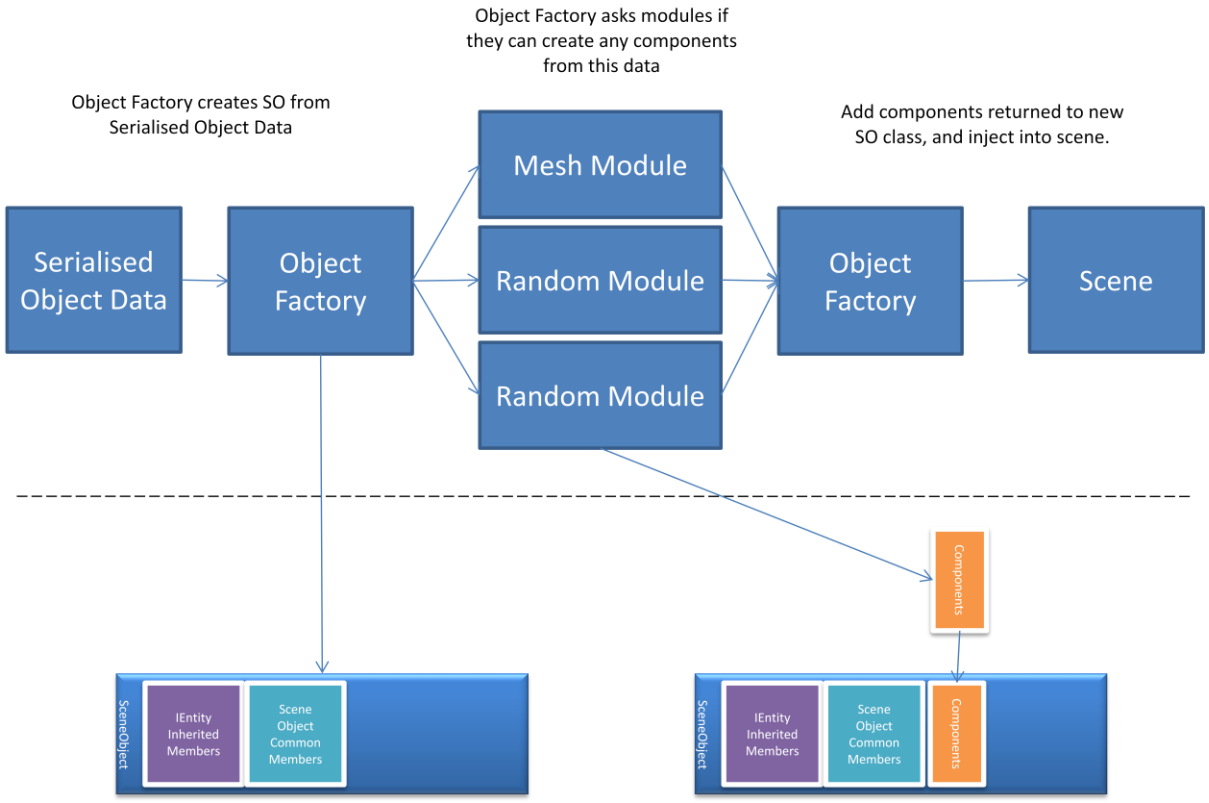
Every module listening for this event may scan `componentData` for data relevant to their module, and substantiate new components from this data. The method of this substantiation is left up to the module in question, but I imagine it will either be via de-serialisation or via standard constructor.

Since components are written as C# Types, they can be a class or an interface; components may contain events, functional members or other 'live' elements without breaking syntax. A sample Mesh component may look like the following.

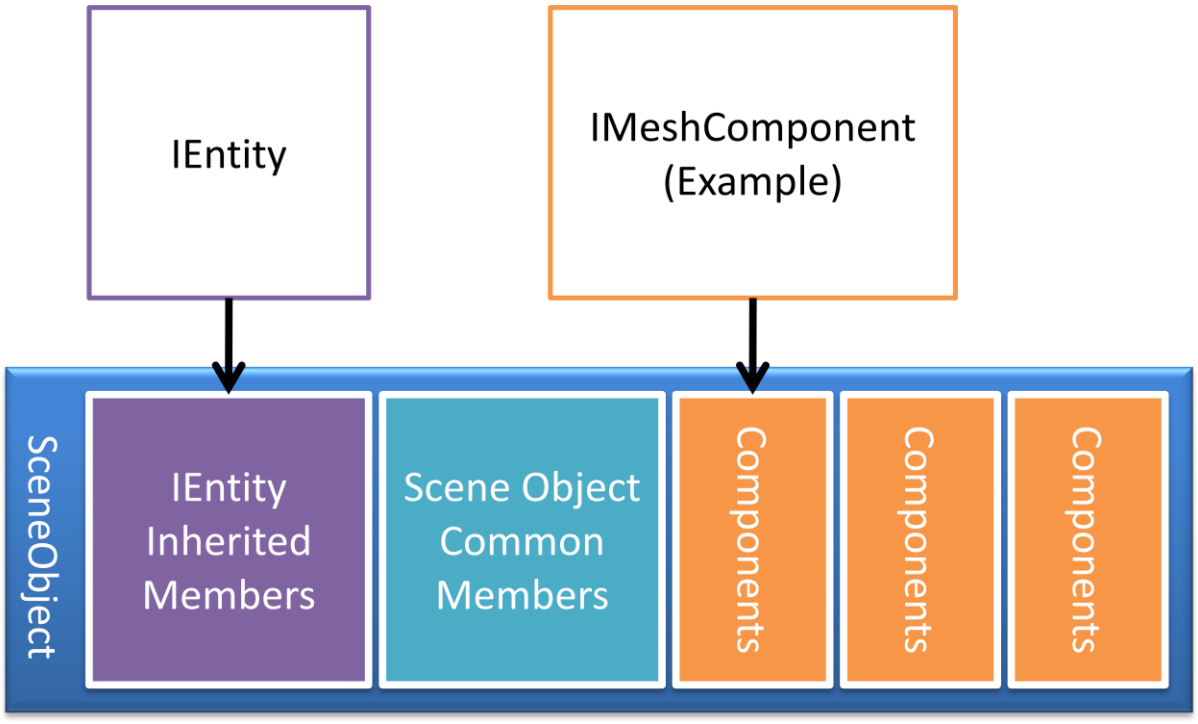
```
public class MeshComponent : IMeshComponent, IComponent {
    public UUID AssetID { get; set; }
    public MeshData CollisionMesh { get; set; }

    public event MeshChangedDelegate OnMeshChanged;

    Type BaseType {
        get {
            return typeof(IMeshComponent);
        }
    }

    ComponentState State {
        get {
            cs = new ComponentState();

            cs.Set("asset",AssetID);

            return cs;
        }
    }
}
```

This is fairly straight forward naive implementation of a component with a single stored property (collision mesh may be built via the AssetID.)

## Diagrams & References for Phase 1




0-1 Scene Object Construction via Factory

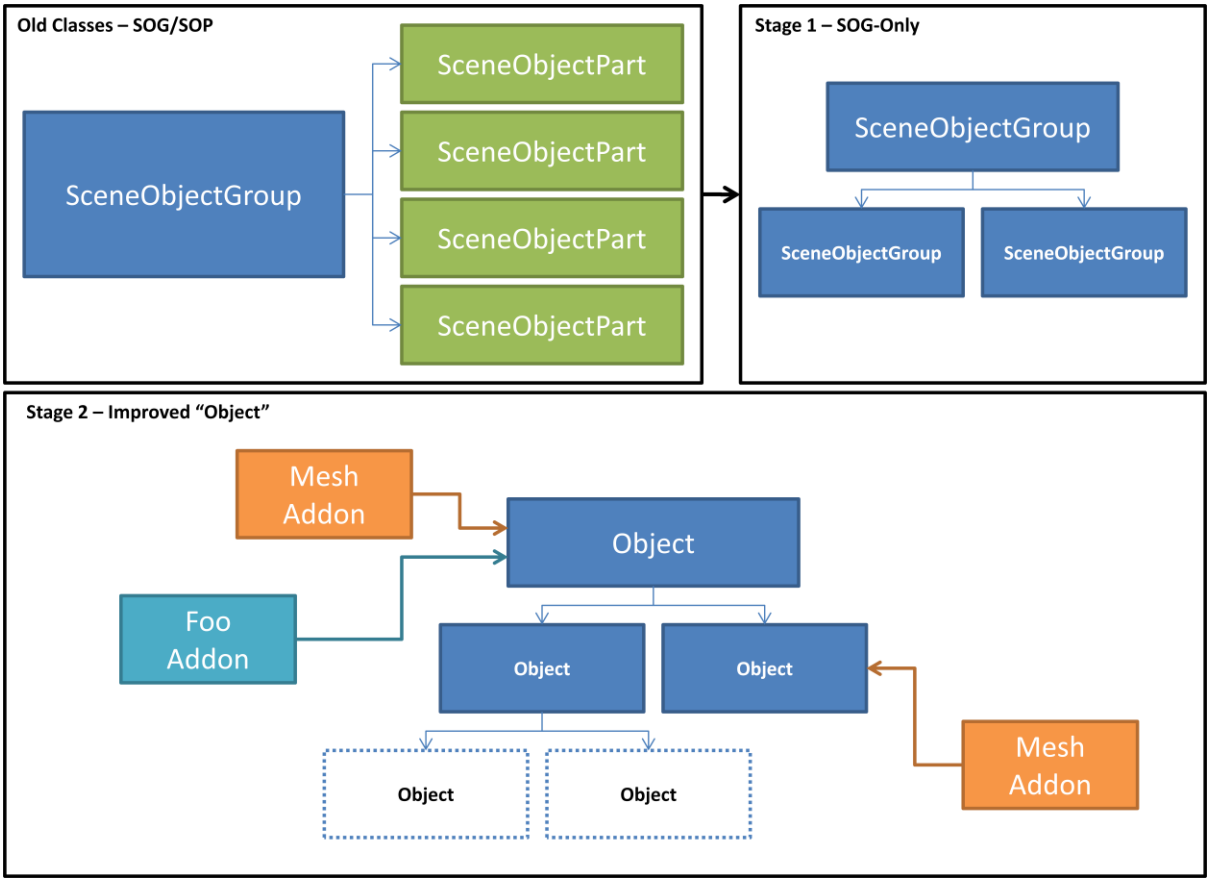



0-2 Scene Object "Layout"

## Phase 2: Merger of SceneObjectGroup & SceneObjectPart

The next step is to refactor the current descriptions of SceneObject into something more manageable. There are two goals to this change – the first is to clean up SceneObjectGroup and Part such that they are accessible to programmers.

For a reference of how this will look in the end – take a look at the `IObject` interface defined for the MRM Scripting API. To see a more basic outline of the process involved in phase two – the following diagram is an overview.




0-1 Improved Scene Object

As you can see from the above diagram the stages in which this conversion will take place. This is a fairly simple move architecturally – however will require a large number of updates across the codebase. Once completed, the SceneObject should be vastly simpler and easier to manipulate for end developers.

The main differences between the proposed IObject and the MRM version is that the IObject in core will be a much smaller class – moving as much "platform specific" properties into Components as possible. For example, the Primitive Shape Data will become a component allowing for alternate shape data models to exist. In this particular case, the collision mesh could be implemented as a type of component, allowing Mesh support (or alternate geometry) to be implemented by other forms of "Object".

## Last Minute Change

After discussions re: the above – it was decided to allow Components to be embedded into a class inheriting from both the component and the base Object class. These would be something such as PrimObject (Object + PrimComponent), MeshObject (Object+MeshComponent) – and would still allow the attachment.

## Feedback

This proposal represents a first-stage design of the new Object Model, I would appreciate feedback from all developers concerned as to the design's soundness – and any particular use cases I may be inadvertently limiting through these changes.

Comments on this proposal should be directed to the "opensim-dev" mailing list as appropriate.