Data Driven Engines

Logic vs Data

- Logic – the functionality of the object or **What the object does.**
- Data – describes attributes and configuration of the object or **What the object is.**

Logic vs Data

Giant cloud of ambiguity

What does “Data Driven Engine” mean?

- Use data files to determine behavior
  - Level files
  - Archetype files
  - Script files
  - Tweakables

What does it look like?

- Very little hardcoded constants.
- The only hardcoded file in the code is to run the configuration file.
- Objects are always created using files/database.
- No hardcoded “new”s for any game object

Why?

- Engineers are slow and expensive.
- Game designers are crazy.
- Need fast iteration.
- Need non-programmers to edit content.
- Rebuilding large projects takes a long time.
- Massive amounts of content.
- Remember principle #2 is Embrace change.

How?

- Need a generic and robust way to load data.
- This process is called **serialization.**
- The key is to encapsulate the variability of what data an object from the functionality of how data is loaded.
- Or: What the object needs (HP, position, etc.) from how the data is read (file, xml, database, etc.)

Serialization Pattern

- Special form of the visitor pattern
- Participants
  - Object that supports the serialize method or interface which defines how it serializes itself
  - The serializer or stream object that encapsulates file/database saving and loading
  - Operators that help to define how objects serialize

Serializers

```
class ISerializer
{
    //Only fundamental types.
    void ReadFloat(float&);
    void ReadInt(int&);
    void ReadString(string&);
};
//Concrete Serializers
class TextSerializer : public ISerializer
{
//...
class BinarySerializer : public ISerializer
{
```

Serialize Operators

```
 //Base Stream Operator
void StreamRead(ISerializer& stream,float& f)
{
    stream.ReadFloat(f);
}

//Extended serialization operators of compound types
void StreamRead(ISerializer& stream,Vec2& v)
{
    StreamRead(v.x);
    StreamRead(v.y);
}
```

Serialization

```
void GameObject::Serialize(ISerializer& stream)
{
    StreamRead(stream, HP);
    StreamRead(stream, Speed);
    StreamRead(stream, Armor);
    StreamRead(stream, SpriteFile);
};
```

Text Serialization

```
100
4.5
20
bigship.png
```

JSON Serialization

```
Object =
{
   HP : 100,
   Speed : 4.5,
   Armor : 20,
   Spritefile : “bigship.png”
}
```

Xml Serialization

```
<Object>
    <int name="HP">100</int>
    <float name="speed">4.5</float>
    <int name="armor">20</int>
    <string name="spritefile">bigship.png</string>
</Object>
```

Serialization Phases

- 1. Construct the object
- Constructed – object has been built from the factory but is not active.
- 2. Serialization and data setting
- The object can be serialized and then data attributes can be adjusted.
- 3. Initialize the object
- Object really comes into existence using all the serialized data

Data Driving
Game Object Creation

Use “new” Everywhere!

```
//awesome code
void MyRandomFunction()
{
    GameObject* pObj = new PlayerShip();
};
```

Use “new” Everywhere!

```
//awesome code
void MyRandomFunction()
{
    GameObject* Obj = new PlayerShip();
};
```

Bad!

Object Management

- Create, manage, and destroy your objects in a unified way.
- Do not just use “new” whenever you need an object.
- Each object should have a clear owner.
- For game objects use a factory.

Create with a Factory

```
GameObject* ObjectFactory::CreateObject(string type)
{
   switch (type)
   {
     case “PlayerShip”:  return new PlayerShip();
     case “EnemyShip”:   return new EnemyShip();
     case “Projectile”:  return new Projectile();
   }
   return NULL;
};
```

Factory Advantages

```
//Do not need includes
//What to create is now data (an string) so it can
//be stored
void MyRandomFunction()
{
    GameObject* pObj =
          GObjectFactory->BuildObject( "PlayerShip" );
};
```

Factory Advantages

```
GameObject* ObjectFactory::CreateObject(unsigned int ID)
{
    GameObject* newObj = BuildObject(ID);
    if( newObj )
    {
            //Single point of object management
            newObj->Intialize();
            ++this->NumberOfGameObjects;
            this->GameObjectList->Add( this );
    }
};
```

Factory Problems

```
#include “Ship.h”
#include “EnemyShip.h”

GameObject* ObjectFactory::CreateObject(string type)
{
      switch (type)
      {
        case “PlayerShip”: newObj = new PlayerShip();
        case “EnemeyShip”:  newObj = new EnemyShip();
        case “Projectile”:  newObj = new Projectile();
      }
```

Factory Problems

```
#include "Ship.h"
#include "EnemyShip.h"
#include "Asteriod.h"
#include "SuperMissle.h"
#include "Carrier.h"
#include "Base.h"

GameObject* ObjectFactory::CreateObject(string type)
{
    switch (type)
    {
        case "PlayerShip":  return new PlayerShip();
```

Distributed Factories

```
GameObject* ObjectFactory::BuildObject(string type)
{
      GameObject* newObj = NULL;
      switch (type)
      {
         case “PlayerShip”: newObj = new PlayerShip();
         case “EnemeyShip”:  newObj = new EnemyShip();
         case “Projectile”:  newObj = new Projectile();
      }
      //return the object for initialization
      return newObj;
};
```

Distributed Factories

```
GameObject* ObjectFactory::BuildObject(string type)
{
    GameObject* newObj = NULL;


    newObj = CreatorMap[type]->Create();


    //return the object for initialization
    return newObj;
};
```

Distributed Factories

```
class GOCreator
{
    virtual GameObject* Create();
    virtual ~GOCreator(){};
};
//elsewhere
class CreateShip : public GOCreator
{
    virtual GameObject * Create()
    {
        return new Ship();
    }
};
```

Creator Registration

```
void GameLogic::RegisterObjects()
{
  GObjectFactory->AddCreator( "Ship", new ShipCreator() );

  //Templates!
  GObjectFactory->AddCreator( "Ship" new TCreator<Ship>() );

  //Macros!
  RegisterCreator( Ship );

};
```

Bringing it together

- Now combine serialization and the data driven factory together.
- Add to the data source what creator it should use.
- The factory serializes the object when it is created.

Add type information to data file

```
PlayerShip
100
4.5
20
bigship.png
```

Add type information to JSON

```
PlayerShip :
{
  HP : 100,
  Speed : 4.5,
  Armor : 20,
  Spritefile : “bigship.png”
}
```

Add type information to data file

```
<PlayerShip>
    <int name="HP">100</int>
    <float name="speed">4.5</float>
    <int name="armor">20</int>
    <string name="spritefile">bigship.png</string>
</PlayerShip>
```

Factory with Serialization

```
GameObject* ObjectFactory::BuildObject(string filename)
{
    FileStream stream(filename);
    string objectType;
    StreamRead(stream, objectType);
    GameObject* newObj = NULL;
    newObj = CreatorMap[objectType]->Create();
    newObj->Serialize(reader);
    //return the object for initialization
    return newObj;
};
```

Wait!

- But we are using a component based engine!
- A game object is just a collection of components.
- So lets also data drive **composition.**

Composition Factory

- In a component based engine all objects are just a collection of components.
- The factory has a list of creators for components.
- It can then use the data source to determine what components are on the composition and their attributes.

Composition Factory

```
GOC* ObjectFactory::BuildObject(string file)
{
  Stream stream(filename);
  GOC * gameObject = new GOC();
  while(stream.IsGood())
  {
    StreamRead(stream,componentName);
    Component * component = CompCreators[componentName]->Create();
    component->Serialize( stream );
    gameObject->AddComponent( componentName , component );
  }
  //return the object for initialization
  return gameObject;
}
```

Determine Components Text

```
Model
BigGuy.bin
Guy
100
4.5
20
```

Determine Components JSON

```
GameObject :
{
   Model :
   {
        ModelFile = “BigGuy.bin”,
   }
   Guy :
   {
        HP : 100,
        Speed : 4.5,
        Armor : 20,
   }
}
```

Determine Components Xml

```xml
<Object>
    <Component type="Model">
        <string name="ModelFile">BigGuy.bin</string>
    </Component>
    <Component type="Guy">
        <int name="HP">100</int>
        <float name="speed">4.5</float>
        <int name="armor">20</int>
    </Component>
</Object>
```

Archetypes

- An archetype is a prototype or the original model (blueprint, recipe, etc.) for an object.
- The factory uses the archetype to build the object and then run time data is modified as needed. (such as position).
  - Object = Car
  - Archetype = Gray Model 2 BMW
  - Instance = Bill's BMW, that BMW on the corner, etc.

Archetype Problems

- What data do you want to have changed per object?
  - Position, Scale, Rotation?
- Can archetype override everything?

Data Driven Factory

- The true power of the factory is when it is data driven.
- Systems register their component creators to the factory.
- Objects are created through archetypes which describe what components are on a composition and their attributes.
- Run time data is modified as needed.

Levels

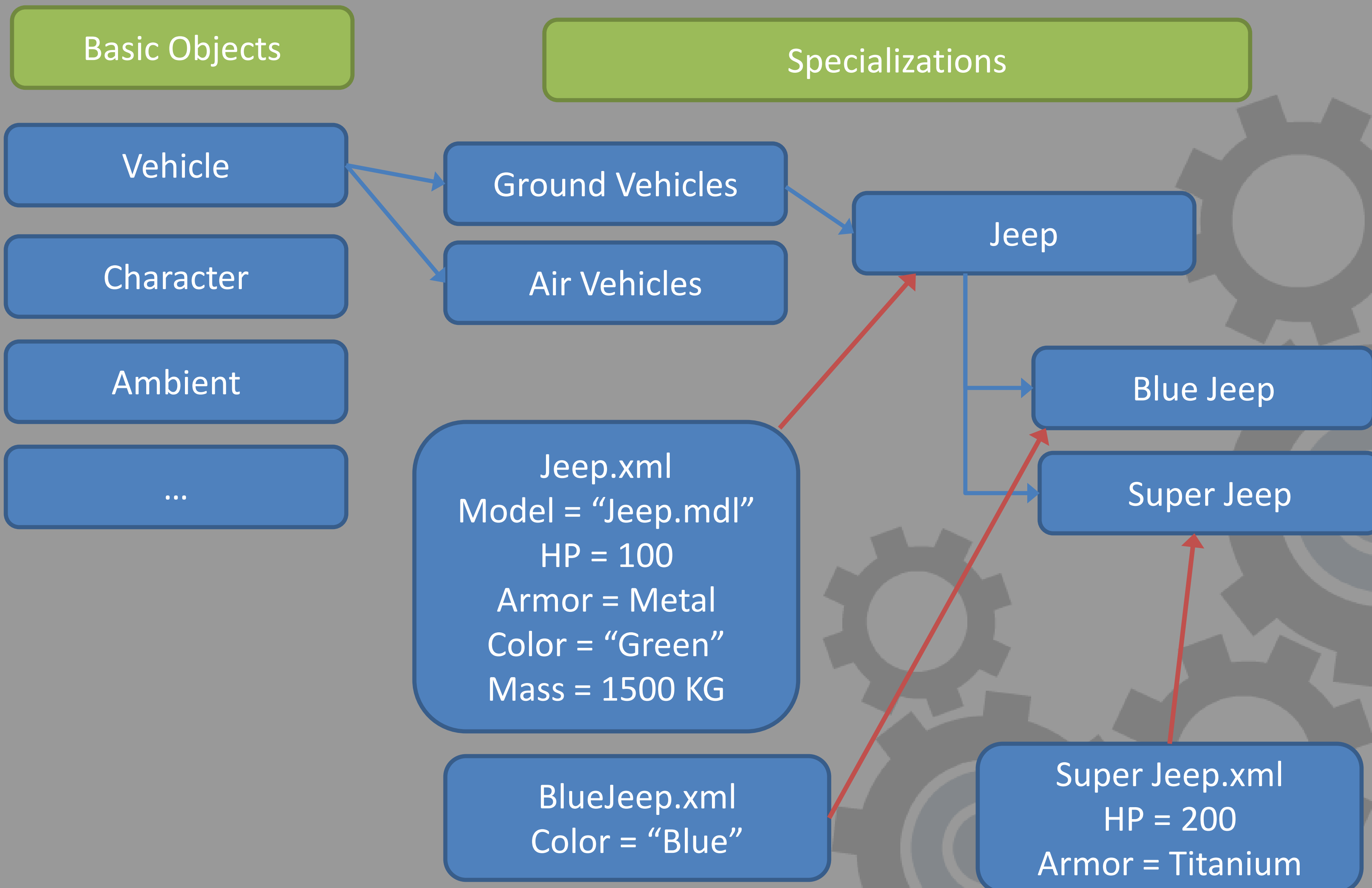
- In a level you will want to place an object multiple times.
- Use archetypes to alias out the objects so their properties can be adjusted.
- The loader then overrides the position, rotation, or whatever else.

Level File

```
Camera.txt
0 0
0
Wall.txt
320 -180
0
Wall.txt
-320 -180
0
Ground.txt
0 -280
0
```

Data Inheritance

Data Inheritance



Data Inheritance

- Create an inheritance hierarchy with your data files.
- Each data file has a tag that gives its parent's name.
- Each data file only contains data that is new or different from its parent.
- Pulling all the data together is done automatically, usually at build time.

Extensions

- Different serializers (xml, JSON, binary, binary in memory, lua, etc)
- Do not always load from a file cache in memory
- Configuration objects vs serializations

Questions?