Adding new Entity and Munition Types

to the Delta3D Stealth Viewer

Version 1.3

Sept 30, 2009

Prepared by:
Alion Science and Technology
Advanced Modeling and Simulation Technologies Operation
5365 Robin Hood Road, Suite 100
Norfolk, VA 23513
(757) 857-5670

Document Control Information

Revision	Revision History	Date
Ver 1.0	Initial Draft (Guthrie)	Sep 18, 2009
Ver 1.2	Reviews and updates (Murphy)	Sep 24, 2009
Ver 1.3	Minor wording changes (Guthrie)	Sept 30, 2009

Table of Contents

1 Background **1**

2 Referenced Documents **2**

3 Details **2**

3.1 HLA Mapping *2*
3.1.1 Attributes (HLA Objects) and Properties (Delta3D Actors) 2
3.1.2 Entity Type Inheritance ('extend') 2
3.1.3 Example of an HLA Entity Mapping 3
3.1.4 A Walkthrough of the M1A1 Entity Mapping 4
3.1.5 Complex attributes 4
3.1.6 Missiles 5

3.2 Munition Types Map *6*
3.2.1 Munition Actors and Munition Effects 6
3.2.2 Editing Munitions in STAGE 6
3.2.3 Creating a new Munition in STAGE 8

List of Figures

Figure 1 – Change Project Context **6**

Figure 2 – Open MunitionsTypeMap **7**

Figure 3 – Look at Global Actors **7**

Figure 4 – Munition Properties **8**

List of Tables

1 Background

The Delta3D Stealth Viewer and Simulation Core (SimCore) are often used in a networked environment using High Level Archictecture (HLA). In such an environment, each community has their own set of entity and munition types that will be supported. To support this, the Stealth Viewer and SimCore have a data driven mechanism for defining and mapping both entity and munition types. These changes can almost always be done without the need to modify the source code.

In an HLA or DIS network, entities and munition types are defined using a 48 bit encoded key referred to as an Entity Type or DIS ID. The term DIS ID is a carry over from the older Distributed Interactive Simulation (DIS) standard. A DIS ID has 7 fields that allows you to define an entity or munition type with a very high degree of precision. Because DIS ID's are hierarchical in nature, you are also able to define an entity or munition type generically. To do this, you simply use less precision and set unused fields to 0. A fully specificied DIS ID has the following fields:

KIND – DOMAIN – COUNTRY – CATEGORY – SUBCAT – SPECIFIC – EXTRA.

The definitions for entities and munitions are done via XML mapping files. At runtime, the Delta3D HLA layer uses the values in the mapping file to determine what an entity or detonation means and displays it as appropriate. It translates the HLA mapping into the appropriate actor type and converts the data directly into an entity actor update.

Adding a new type of entity requires the Entity Type code (DIS ID) plus a 3D visual model to draw in the Stealth Viewer. If you have them, the mapping also allows you to specify a separate model for the damaged and destroyed states. A separate mapping file exists for each HLA federation because their data formats vary somewhat.

Munitions are significantly different from entities. They are handled separately and have their own mapping file. Each munition has between one and three phases: the fire interaction, an entity showing the munition during flight, and the detonation interaction. Direct fire weapons usually only have a fire interaction that includes the firing entity, the munition type, and the target entity if one was hit. Indirect fire munitions such as missiles have fire interactions for the launch/creation of the missile but also may exist in the world long enough that they show up as an entity for the purpose of simulating the flight path of the munition. In that case, the entity portion works exactly like any other entity mapping. When the indirect munition explodes, a detonation interaction is sent.

In the Stealth Viewer, the fire and detonation interactions are mapped once and the Entity Type data is sent on to an internal munitions component. This has a separate map file that defines visual and audio effects associated with Entity Type codes on these interactions.

2 Referenced Documents

See relevant contractual statement of work.

3 Details

This section describes the various files and gives a step-by-step explanation of what to do to add new types of entities and munitions.

Note: Delta3D uses a project context data directory. This directory may be named anything, but for the purposes of this document, it will be referred to as "ProjectAssets" which is the most common named used in Simulation Core base applications.

3.1 HLA Mapping

The HLA Mapping files are normally in subdirectories of "ProjectAssets/Federations". The XML schema file is in that directory and is named "HLAMapping.xsd". The one used for the NTF federation is in "ProjectAssets/Federations/ntf/NTFMapping.xml". A mapping file has a number of xml sections: <header>, <libraries>, <ddm>, <objects>, and <interactions>. New entities will go in the <objects> section. The term "objects" refers to HLA objects, not delta3d objects or actors.

Do define a new entity, you will need to know the HLA Object Type and the DIS ID. Those 2 fields create a unique mapping to a Delta3D Actor Type. Note that many different entities may be mapped to the same Delta3D Actor Type (ex MissileActor). HLA object types may vary depending on your federation, but often look something like this: "BaseEntity.PhysicalEntity.Platform.GroundVehicle." The Delta3D Actor Type will typically look something like this: "Entity.PlatformWithPhysics."

3.1.1 Attributes (HLA Objects) and Properties (Delta3D Actors)

When an entity is sent over the network, each piece is sent as a stand alone piece of data such as WorldLocation, Orientation, and VelocityVector. On the HLA side, these pieces of data are called attributes whereas on the Delta3D side, they are called Actor Properties. Each entity actually has many attributes and properties.

3.1.2 Entity Type Inheritance ('extend')

To fully define the mapping from an HLA object to a Delta3D Actor typically requires about 10-20 different HLA attribute to Actor property mappings. Fortunately, most entities share the same set of HLA Object attributes. Therefore, the mapping file allows entity definitions to inherit their object to actor mappings via an abstract mapping. The abstract mappings are defined at the top of the <objects> section. Abstract mappings have the <abstract/> tag inside them instead of an HLA object type and an actor type. The final entity definition will 'extend' from an abstract mapping. Note that abstract mappings can extend another abstract mapping, allowing you to create a nice set of mappings for sharing values.

Typically, most HLA mapping files will start with an abstract mapping for a basic entity: <object name="BaseEntity">. This defines a basic set of parameters used by all entity objects such as velocity, orientation, and dead reckoning algorithm. It mostly just has a list of <attrToProp> definitions which are inherited in extended mappings. The "BaseEntity" mapping is often extended into "PhysicalEntity" which is then extended by "Platform," "Lifeform," "Missile," and "SurfaceVessel." "Platform" is further extended to "GroundVehicle," "Submersible," and "Aircraft." "Lifeform" is further extended into "Human."

Almost every entity mapping will extend one of these abstract mappings. Not only do the base mappings provide hooks to send over the properties for each of the different HLA object types, but also they define internal property values such as the domain and whether the entity flies. If your federation uses Distributed Data Management (DDM), then you can also define your DDM space on the abstract mapping. It is important to extend from the correct abstract mapping or you will end up duplicating a lot of properties.

3.1.3 Example of an HLA Entity Mapping

Here is a mapping for an M1A1 tank:

```
<object extends="GroundVehicle" name="M1A1">
   <objectClass>BaseEntity.PhysicalEntity.Platform.GroundVehicle</objectClass>
   <actorType>Entity.PlatformWithPhysics</actorType>
   <remoteOnly>true</remoteOnly>
   <disEntityEnum>
      <kind>1</kind>
      <domain>1</domain>
      <country>225</country>
      <category>1</category>
      <subcategory>1</subcategory>
      <!-- This is generic for different versions of the M1A1 -->
      <specific>0</specific>
      <extra>0</extra>
   </disEntityEnum>
   <attrToProp>
      <gameName>Non-damaged actor</gameName>
      <gameDataType>StaticMeshes</gameDataType>
      <default>StaticMeshes:m1a1:m1a1_desert_good.ive</default>
   </attrToProp>
   <attrToProp>
      <gameName>Damaged actor</gameName>
      <gameDataType>StaticMeshes</gameDataType>
      <default>StaticMeshes:m1a1:m1a1_desert_damaged.ive</default>
   </attrToProp>
   <attrToProp>
```

```
        <gameName>Destroyed actor</gameName>
        <gameDataType>StaticMeshes</gameDataType>
        <default>StaticMeshes:m1a1:m1a1_desert_destroyed.ive</default>
      </attrToProp>
      <attrToProp>
        <gameName>ShaderGroup</gameName>
        <gameDataType>STRING</gameDataType>
        <default>SimpleVehicleShaderGroup</default>
      </attrToProp>
    </object>
```

3.1.4 A Walkthrough of the M1A1 Entity Mapping

For the 'M1A1' entry, the entity extends the "GroundVehicle" mapping. It uses the HLA object type "BaseEntity.PhysicalEntity.Platform.GroundVehicle" which is the object type defined in the FOM. The <actorType> tag specifies the Delta3D actor type to use. All vehicles should use "Entity.PlatformWithPhysics." All Human mappings should use "Entity.HumanWithPhysics". Missiles, Torpedos, and most other munitions should use "Munitions.Missile." The names for the object to actor type mappings have similar names, so it can be easy to confuse them. Note also that it has the disEntityEnum, which is the Entity Type code that associates the mapping with the data on the network.

Note that <specific> and <extra> both have the value "0." This means that they are wild cards, so this mapping would, for example, match entities with codes of "1 1 225 1 1 0 0" and "1 1 225 1 1 2 3." A mapping can still be defined for something more specific such as "1 1 225 1 1 4 4." When matching the Entity Type code, it will always pick the most specific mapping. Just to be clear, if an entity came over with code "1 1 225 1 1 4 3" and two mappings existed for "1 1 225 1 1 0 0" and "1 1 225 1 1 4 0", the second mapping would match because it more exactly matches the actual entity.

The mapping example above has three 3D models defined for the non-damaged, damaged, and destroyed states. It uses the Delta3D resource descriptor format which uses colons instead of slashes for the path. This is for cross platform safety and to conceptually distinguish resources from actual file paths.

The "ShaderGroup" property mapping is necessary so it can do lighting correctly, among other things. The "SimpleVehicleShaderGroup" can be used on any object. More specific shader groups exist to support reflectivity, underwater effects, and other lighting properties for specific 3D models, but many new entities can just use the default simple one.

3.1.5 Complex attributes

Consider the <remoteOnly> option. It defaults to false. The "remoteOnly" option means that the mapping is only valid for incoming or remotely simulated entities. If you are primarily using the Stealth Viewer and do not have any other SimCore based

applications, then you should always set the “remoteOnly” option to true. Since it defaults to false, you will need to do make sure to add this entry or you may get errors if you reuse an Actor Type.

Consider the <localOnly> option. It also defaults to false. The “localOnly” option allows you to setup an entity with dual hats – one actor type for when the entity is simulating locally and a second actor type for incoming remote instances of the entity. This can be used for example with the Player’s position – locally, we want it to be the Stealth Actor, but for remote entities, we might want to map it to a human entity. This setting is only valid for locally simulated entities that are being published. Again, if you are only using the Stealth Viewer, then this value should always be false since the Stealth Viewer doesn't simulate entities.

3.1.6 Missiles

Missiles have their own abstract mapping called, ‘Missile’. The abstract mapping defines a smoke trail and flames and the shader. Since missiles move very fast, they may sometimes have simpler effects. Missiles also don't need a damaged model for obvious reasons.

```
<object name="Missile AT 6C Spiral" extends="Missile">
  <objectClass>BaseEntity.PhysicalEntity.Munition</objectClass>
  <actorType>Munitions.Missile</actorType>
  <remoteOnly>true</remoteOnly>
  <disEntityEnum>
    <kind>2</kind>
    <domain>2</domain>
    <country>222</country>
    <category>1</category>
    <subcategory>8</subcategory>
    <specific>0</specific>
    <extra>0</extra>
  </disEntityEnum>
  <attrToProp>
    <gameName>Non-damaged actor</gameName>
    <gameDataType>StaticMeshes</gameDataType>
    <default>StaticMeshes:hydra_70:hydra_70.ive</default>
  </attrToProp>
</object>
```

To add a new class of long-duration munitions such as Torpedo, you would create a new Abstract mapping just as with the Missile. Then, for each specific type of torpedo, you would have an entry just like the AT 6C Spiral.

3.2 Munition Types Map

This section explains how to map munitions. Like entities, munitions have an Entity Type code. Past that, munitions are entirely different. As explained above, munitions typically have between one and three phases: the initial fire interaction, a simulated munition entity (ex missile/torpedo), and the final detonation interaction. For the interaction phases, munitions have complex data properties for explosion particles, impact sounds, and burning and light effects. There is too much data to define in the HLA mapping. Consequently, munitions have their own map file called the Munition Types Map. This file is found in 'ProjectAssets/maps/MunitionTypesMap.xml.'

3.2.1 Munition Actors and Munition Effects

Defining new munition types can be fairly complicated. To make this process easier, munitions are broken up into 2 pieces: the munition type itself (MunitionTypesActor) and the graphical effects (MunitionEffectInfoActor). The MunitionTypeActor defines the munition itself (such as Grenade, 50 Cal Machine Gun, or Large IED) and has properties such as name, DIS id/Entity Type, Damage Type, and Effect.

The graphical effects, particles, lighting, and sounds for each detonation are defined by the MunitionEffectInfoActor. The Effects actor has a lot of properties, but each effect is typically shared by many different munitions. For instance, you may have mappings for a 50 cal machine gun, 240G, 249, M16, etc, but may only have a single Effects actor for all of them that shows tracers, has an impact sound, and a particle effect for hitting dirt. As another example, consider the 'Large Explosion'. This effect might be used for a dozen different munitions from 500 lb bombs to torpedos.

3.2.2 Editing Munitions in STAGE

The mapping file is made to be edited with the STAGE tool from Delta3D. Below are the steps to using STAGE to open and edit this file. First, run STAGE, then go to File->Change Project... The first time you run it, it should do that automatically.

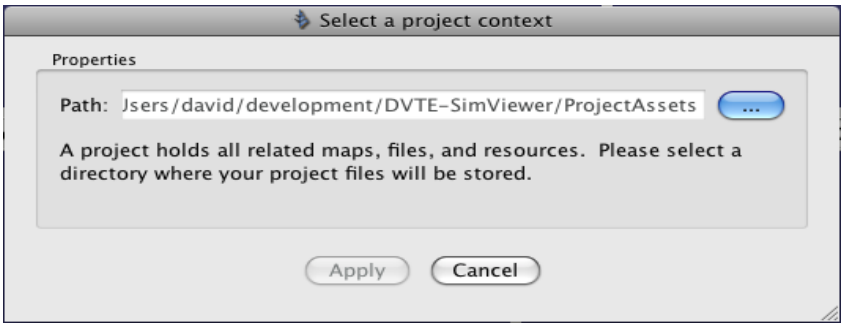

Figure 1 – Change Project Context

Select the ProjectAssets Directory and hit Apply.

Then go to File->Open Map

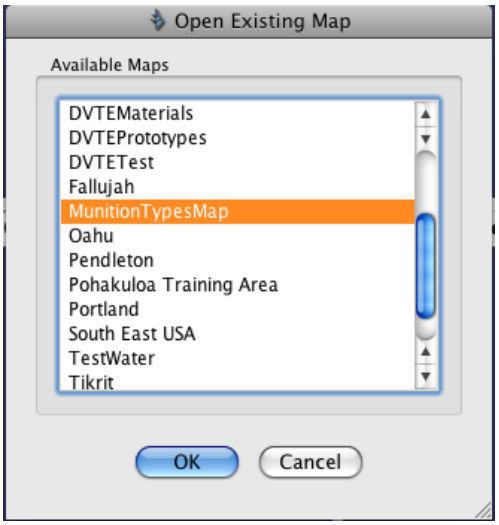

Figure 2 – Open MunitionsTypeMap

Pick the MunitionTypesMap. Go to the "Search" window, then the "Global Actors" tab.

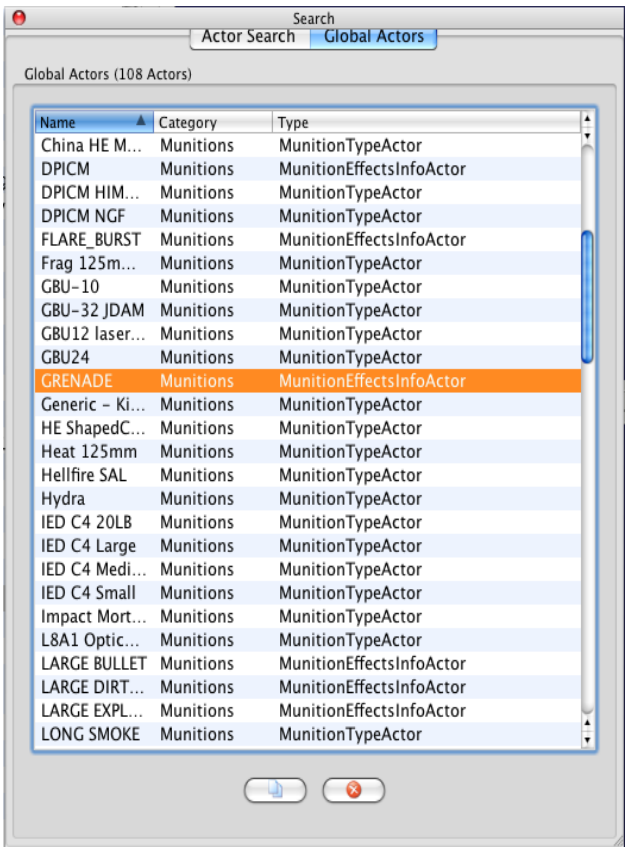

Figure 3 – Look at Global Actors

You can then look at the different Type and Effect actors. Go to the Property Editor after you select one to see the properties.

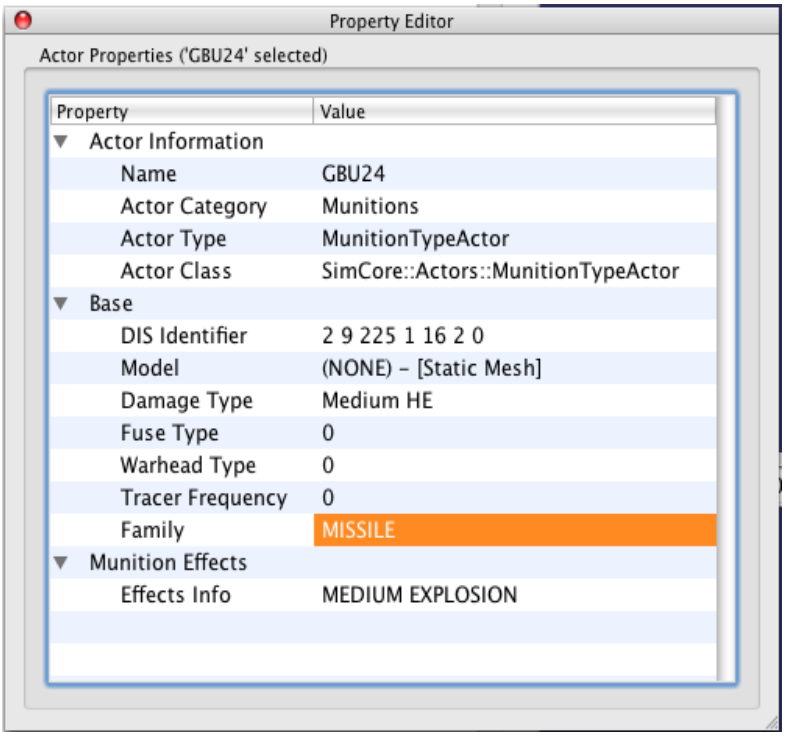

Figure 4 – Munition Properties

The DIS Identifier property is like the one in the HLA mapping. You enter in one number at a time with spaces, and 0's are wild cards as in the HLA mapping. For the Stealth Viewer the Damage, Fuse, and Warehead types don't matter. The "Tracer Frequency" is for direct fire weapons that use tracers, such as a 50 cal machine gun. The most important property is the Effects Info, which references a MunitionEffectInfoActor. In most cases, it is best to pick one of the existing MunitionEffectInfoActors for a new munition type and then save the map.

That will make the Fire and Detonation interactions work correctly.

3.2.3 Creating a new Munition in STAGE

To create a new munition, find a munition that is similar to the new one. Then, use the 'Duplicate' button on the Edit menu to make a copy of the munition. Change the DIS Identifier, Model, and other properties as desired and then save the map.

If you need to add a new effect, you will need sound effects and particle systems that you will have to import into STAGE, and there are a significant number of properties that have to be set. That process is outside the scope of this document.

Appendix A: Glossary

Acronym	Definition
API	Application Programming Interface
DDM	Distributed Data Management
FOM	Federated Object Model
GM	Game Manager
HTML	Hypertext Markup Language
HLA	High Level Architecture
NPS	Naval Postgraduate School
PC	Personal Computer
SDD	Software Design Document
SRD	Software Requirements Document
SOW	Statement of Work
STAGE	Simulation, Training, and Game Editor, a tool that is part of Delta3D
UI	User Interface
XML	eXtensible Markup Language