Building Game-Based Trainers with the Delta3D Game Manager

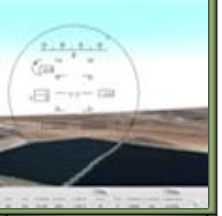
Curtiss Murphy
Alion Science and Technology
BMH Operation
cmmurphy@alionscience.com

Erik Johnson
Naval Postgraduate School
rejohnso@nps.edu

See Delta3D on the floor!

* NPS Delta3D - 775
* Alion - 701

I/ITSEC Tutorial
Dec 4, 2006






Tutorial Contents

- Introductions (Intro)
  - Introduce Game Engines
  - Introduce Delta3D
  - Tutorial-At-A-Glance
- Tutorial Parts
  - Part 0 – Delta3D Overview
  - Part 1 – Actors, Properties, & STAGE
  - Part 2 – Libraries & Actor Types
  - Part 3 – Game Manager & Game Actors
  - Part 4 – Game Manager Components
  - Part 5 – Messages

INTRODUCTION

Hint: No code ☺

Intro - Assumptions

- Assumptions
  - Gaming technologies are a valuable part of our training toolbox
  - Delta3D interests you because it is Open Source
  - You want to learn more about using Delta3D
- Constraints
  - Time limit – 90 minutes
  - Some topics covered briefly
  - Copy-paste of code to avoid typos
- Audience
  - Software developer or manager
  - Technical background
    - WARNING - lots of C++ code in this tutorial!!!






Intro - What is a Serious Game?

- Serious Game
  - Use of game technology for non-entertainment purposes (ex. Training)
- Why use Serious Games at all?
  - Experiential fidelity!
    - Immersiveness
    - Dynamic
    - Interactive
    - Engaging
    - Immediate Feedback
    - Simple Interface






Intro – What is a Game Engine?

- Visualization
  - Move around and ‘see’ the simulation – typically 3D
  - 3D Models (trucks, planes, tanks, soldiers)
  - 2D Textures (brick walls, satellite imagery, UI Icons)
  - Terrain (large or small, indoor or outdoor)
  - Shaders (detail mapping, specular highlights, bump maps)
- Behaviors
  - Moving and rotating in 3D space
  - Character Animation (walking, running)
  - Physics (collisions, gravity)
  - Weather (clouds, fog, sun rise and set)
  - Particle Effects (smoke, explosions)
- Misc
  - User Interface (Heads Up Display)
  - Input (joystick, keyboard, mouse)
  - Sound (voices, explosions, music, ambient)





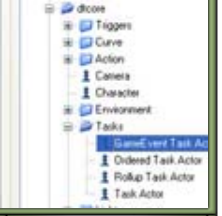

Intro – Tutorial at a Glance

- Part 0 – Overview of Delta3D
- Part 1 – Actors, Properties, & STAGE
  - Actor - building blocks of the world
- Part 2 – Libraries & Actor Types
  - Put the building blocks into a library
- Part 3 – Game Manager (GM) & Game Actors
  - Managing the world
- Part 4 – GM Components
  - Adding high level behaviors
- Part 5 – Messages
  - Communication is everything!

TUTORIAL
PART 0

Overview of Delta3D

Hint: No code! Relax your brains for now…






What is Delta3D?

- Delta3D
  - Open Source Gaming Engine == FREE
  - Government maintained - Naval Postgraduate School (NPS)
  - Active community involvement
    - www.delta3d.org
    - ~ 1000 registered users, 7500+ Forum Posts, 45+ Tutorials
    - Numerous companies and organizations involved
- What's it for?
  - Primarily for 3D visualization (such as Stealth Views)
    - 3D Models, 2D Textures, Input Devices, Audio, Physics, Weather, Terrain, Character Animation, Particle Effects, Graphics Shaders, User Interface (HUD)
  - Game-based training – especially Modeling & Simulation
    - Large Terrains (Terra Page, OpenFlight), HLA, After Action Review (AAR), Learning Management System (LMS), 3D Simulations
- Specifically geared to M&S community!



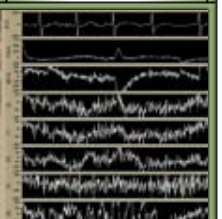



Legal Mumbo Jumbo

- Delta3D is licensed under LGPL
  - Lesser GNU Public License
    - http://www.gnu.org/copyleft/lesser.html
  - Non-viral in nature. Applications built with Delta3D may retain a proprietary license.
- STAGE is licensed under GPL
  - GNU Public License.
    - http://www.gnu.org/copyleft/gpl.html
  - Modifications directly to STAGE's UI code may *not* retain a proprietary license unless you purchase a QT license.





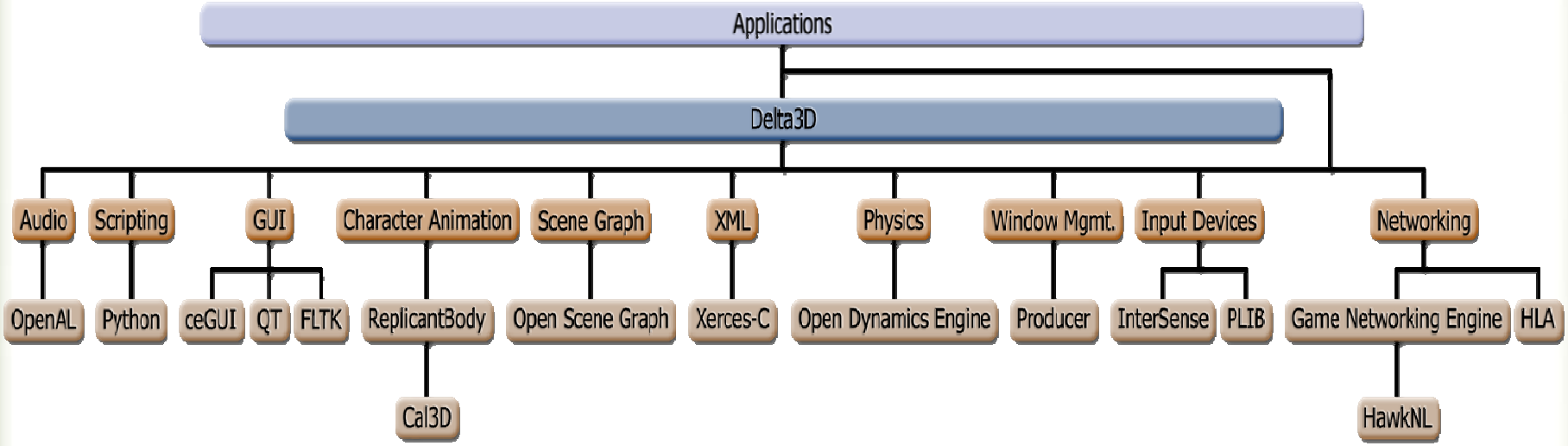
Features

- Cross platform
  - Windows XP, Linux, and Apple Mac OS X (unofficial)
- High level C++ API
  - Includes some Python bindings
- Dynamic Actor Layer
  - For creating and setting properties on Actors
- GameManager
  - Messaging framework for managing interaction among Actors
  - Base game
- After Action Review System
  - Record/playback and task tracking
- Graphics
  - Full support for OpenGL Shading Language
  - Supports many standard 3D file formats (.3ds, .flt, .osg, .ive, .obj, TERREX)
  - 3D content exporters for Max 7, Maya 6, and Blender 2.x

Mo' Features

- 3D audio rendering
- Character animation & scripting
- HLA & game-style client/server networking
- Learning Management System (LMS)
  - SCORM 2004 compliant
- Extensible terrain architecture
  - Runtime generation from DTED, continuous level of detail using SOARX, procedural vegetation placement, GeoTIFF satelite imagery support
- Tool suite
  - STAGE, Particle System Editor, 3D Model Viewer
- Unit tests (27,100+ lines)

Architecture








What's this tutorial all about?

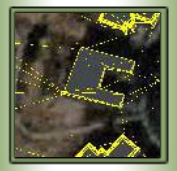
- Actors
- Actor Libraries
- STAGE
- GameManager

- Many other tutorials are available online at http://www.delta3d.org

TUTORIAL
PART 1

Actors, Properties, and STAGE

Hint: An app and XML, but still no code



Part 1 – Actors

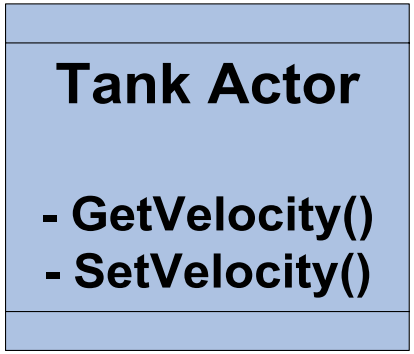
- What is an Actor?
  - An object in the world that we care about
  - Can be moving (a tank) or static (a tree)
  - Visible (building) or invisible (a task/objective)
  - Have behavior (guided missile)
  - With position (green eyed monster) or without (infinite light)
  - Almost anything can be an Actor
    - Player, enemies, weapon, terrain, sound, NPC character, explosion, vehicle, F18, chair
- Our symbol for Actor:






Part 1 – Properties

- What is a property?
  - Data about an Actor
    - Velocity, position, rotation, light color, engine sound, max volume, static mesh
    - Any piece of data you want available outside of your actor for reading (get) or writing (set)
- Our Actor again, with methods for Velocity:



- We'll revisit properties later, in Part 2

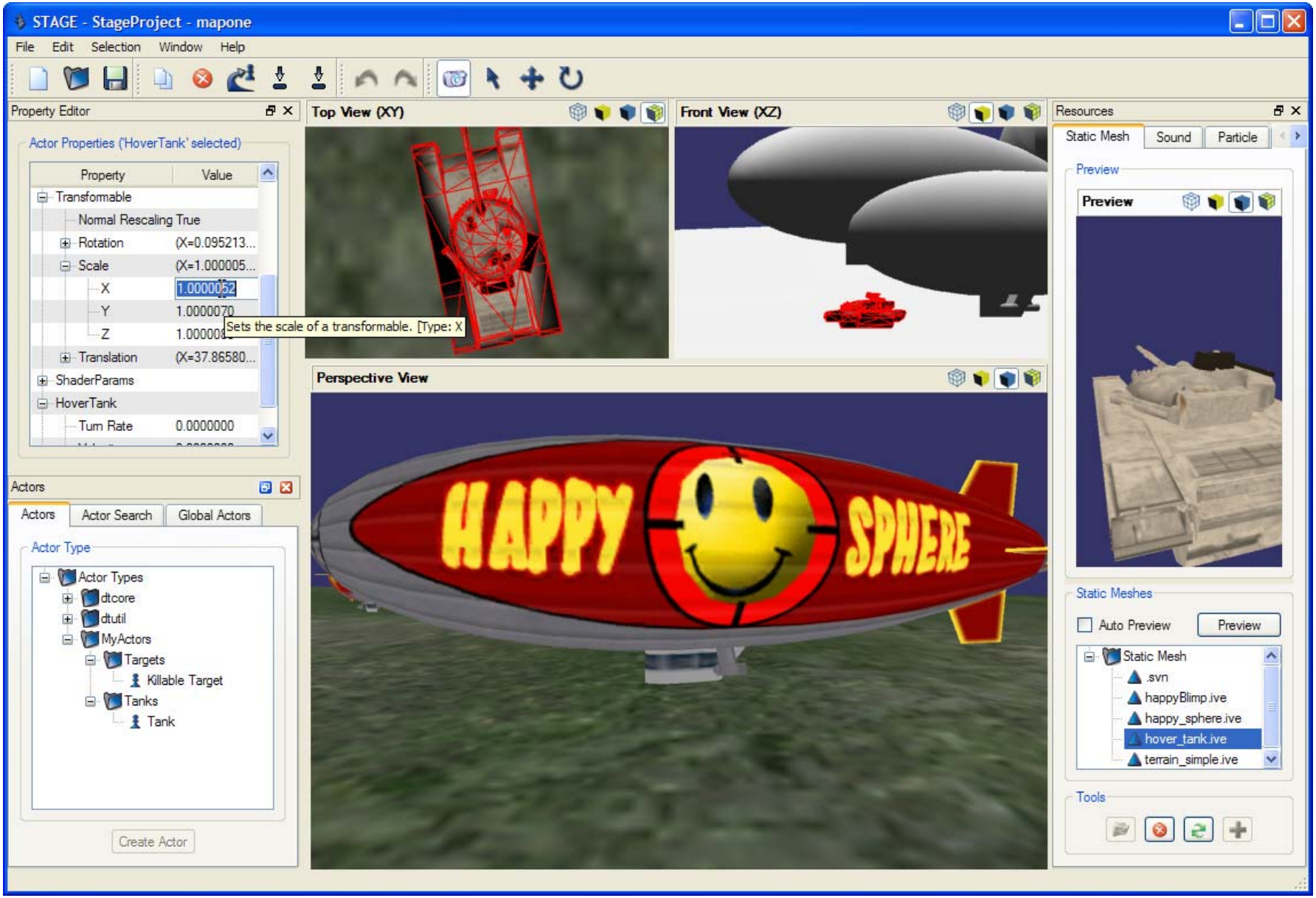



Part 1 – STAGE

- What is STAGE?
  - Simulation, Training, And Game Editor
  - A 3D map editor
    - Place Actors
    - Edit Properties
    - Load resources – meshes, sounds, …
  - Create multiple maps for one project
- As the saying goes,
  - "All the world is in STAGE, and all the men and women merely Actors."

    … (or something like that)

Part 1 – STAGE (Demo)



TUTORIAL
PART 2

Actor Proxy, Actor Types and Actor Libraries

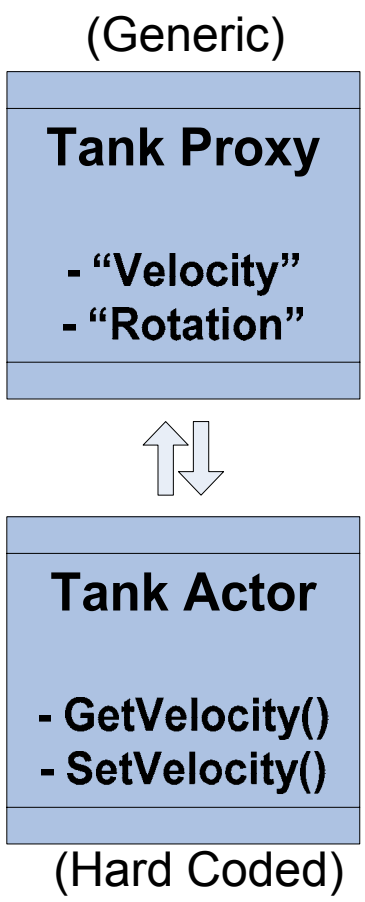
Hint: Yay! Actor Code!






Part 2 – Actor Proxy

- What is a proxy?
  - A substitute that can be used in place of something else
- What is an Actor Proxy?
  - An object that can be used in place of an Actor
  - The Proxy creates the actor
  - Generically exposes actor properties
    - Hard coded methods are not scalable
  - Provides consistent behavior across all objects
    - For example: properties, unique ID, name, class instance identifier, actor type, billboard icon, render mode, …
  - Proxies allow legacy Actor classes to be wrapped non-intrusively
- Revisit Actor Properties
  - Properties are generic - have a name & type
    - String, integer, float, vec2, vec3, vec4, colorRGB, enumeration, mesh resource, sound resource, …



Part 2 – Actor Proxy Snippet

- Header Snippet

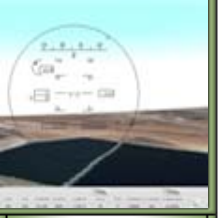
```
class TankActorProxy : public dtActors::GameMeshActorProxy
{
  public:
     virtual void BuildPropertyMap();

  protected:
     virtual void CreateActor();
};
```

- Class Snippet

```
void TankActorProxy::BuildPropertyMap()
{
   …

   AddProperty(new dtDAL::FloatActorProperty("Velocity","Velocity",
      dtDAL::MakeFunctor(actor, HoverTankActor::SetVelocity),
      dtDAL::MakeFunctorRet(actor, HoverTankActor::GetVelocity),
      "Sets/gets the hover tank's velocity.", "TankProperties"));
}
```






Part 2 – Actor Type

- Simple class that describes an Actor Class
  - Meta data about the Actor Type
    - Name – “CoolTank”
    - Description – “A really cool Tank, gosh!”
    - Category – “MyActors.Tanks” – enforces uniqueness and UI hierarchy in STAGE
    - Parent Type – allows use of InstanceOf(), if this Actor is a subclass of another Actor
      - Ex OrderedTaskActor is subclass of TaskActor
- Snippet

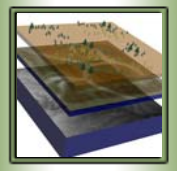
1) As Local (if not used elsewhere)

```
new dtDAL::ActorType (“Tank", “MyActors.Tanks", “A cool Tank!");
```

2) As Static member (use when referenced elsewhere)

```
dtCore::RefPtr<dtDAL::ActorType> ActorsRegistry::TANK_ACTOR_TYPE(
   new dtDAL::ActorType( "Tank","MyActors.Tanks","A cool tank!" ) );
```



Part 2 – Actor Libraries

- A collection of your Actor Classes
  - Allows you to group your actor classes
  - Allows STAGE to find and load your actors dynamically (without restarting!)
  - Allows you to use Actor Types in a map
- **All** Actor Proxies Must Be In A Library!!!
  - No exceptions! Not even for Delta3D!
- Snippet

1) As Local (if not used elsewhere)

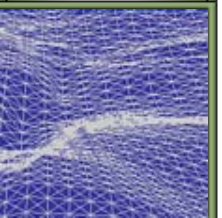
```
mActorFactory->RegisterType<TankActorProxy>(
        new dtDAL::ActorType (…));
```

2) As Static (use when referenced elsewhere)

```
mActorFactory->RegisterType<TankActorProxy>(
        TANK_ACTOR_TYPE.get() );
```





Part 2 – CODE TIME 1

Visual Studio – "ProjectAlpha"

1. Build a new Tank Actor
    1. Add Properties to BuildPropertyMap()
        1. Velocity
        2. Turn Rate
2. Create an Actor Library
    1. RegisterType in RegisterActorTypes()
3. Load our Library in STAGE

TUTORIAL
PART 3

The Game Manager (GM) and Game Actors

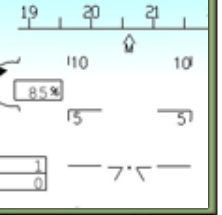
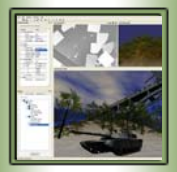
Hint: Theory and Code





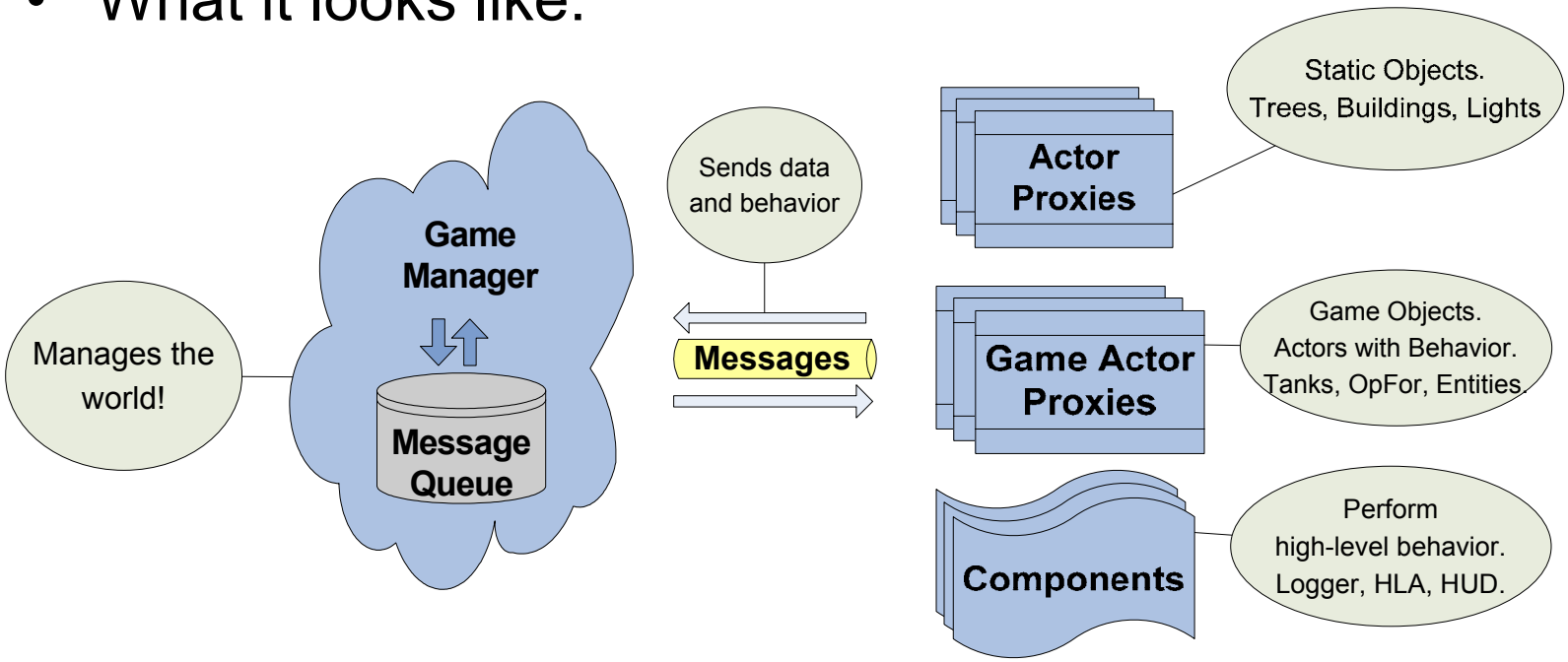
Part 3 – Actor Review – What now?

- What are we going to do with our Actors?
  - We can define actor classes
  - Make actor types and libraries
  - Load them into STAGE
  - Create instances in our map
- What now? How will our Actors:
  - Get into the world?
  - Perform behavior each tick?
  - Interact with each other?
  - Send themselves across a network?
  - Or save state for playback or after action review?

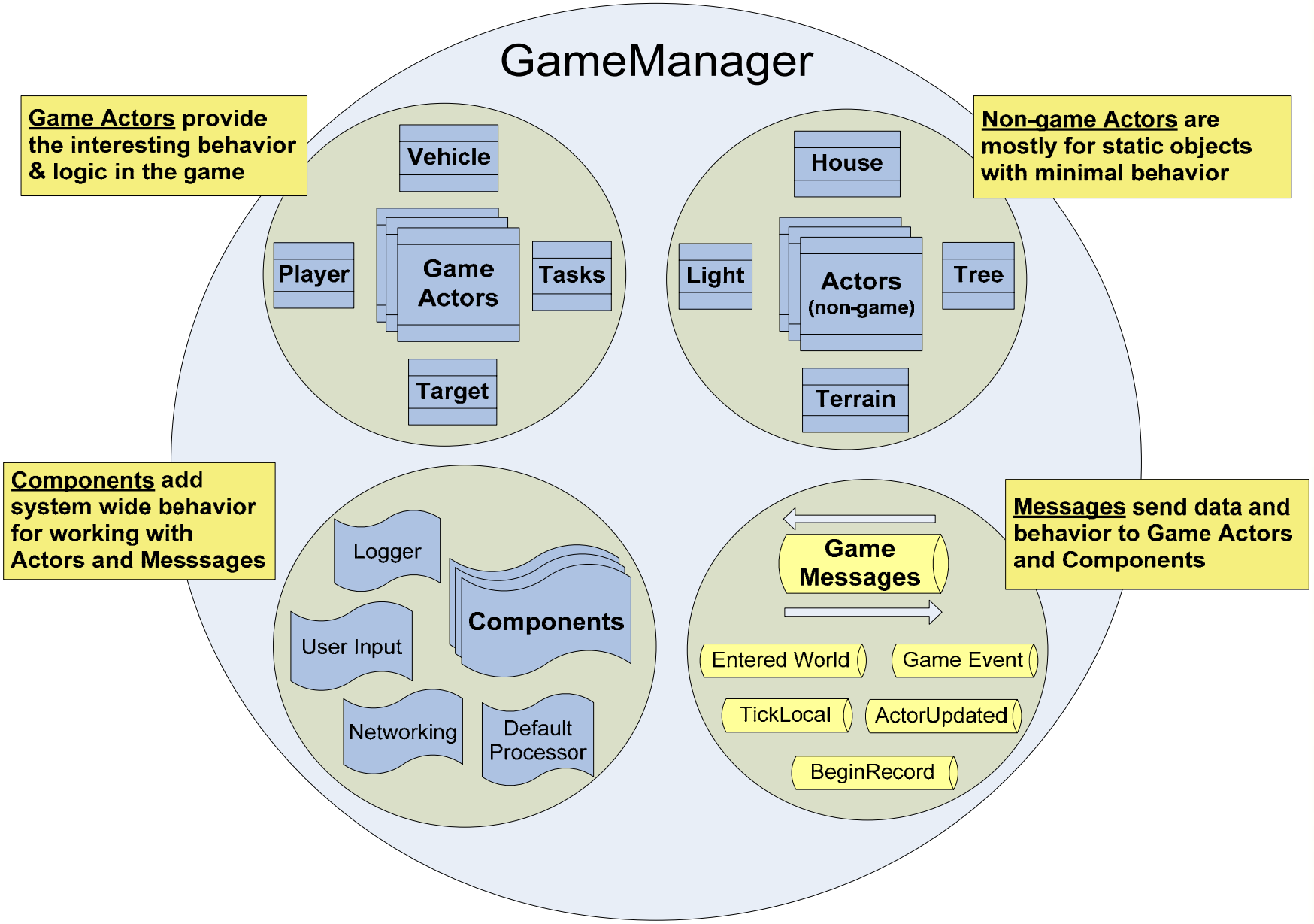




Part 3 – Enter the Game Manager

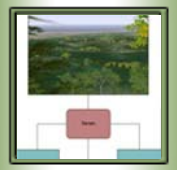
- The Game Manager (GM)
  - Manages all Actor Proxies in our app
  - Works with maps (loads objects into the scene)
  - Routes messages (to/from actors or components)
  - Controls time
  - Works with major systems (components)
- What it looks like:



Part 3 – A Different Perspective

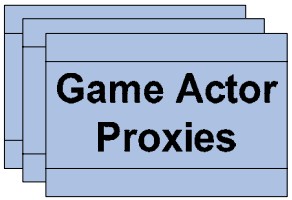






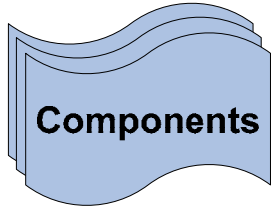

Part 3 – New Concepts

- Game Actor
  - A special type of Actor (subclass) that works well with the Game Manager
  - Understands ‘tick’ & ‘remote’ vs ‘local’ ownership
  - Works with messages
    - Can send and update itself via messages



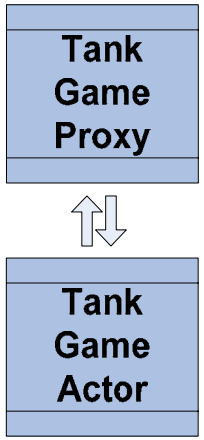
- Game Message
  - Special way of sending behavior or data around the system – to/from actors and components
  - Supports message parameters
  - Can serialize itself

- GM Component (covered in Part 5)
  - High level behavior – more abstract than an actor
  - Usually one instance of each component
  - Behavior like networking, logging & playback, HUD & UI, keyboard/mouse input, dead reckoning



Part 3 – Revisit Our Tank

- Revisit our Tank Actor
  - Need to register for Tick messages
    - RegisterForMessages(which_message, which_invokable_method) on Proxy
    - Message – dtGame::MessageType::TICK_LOCAL
    - Invokable - predefined methods for ticks
      - dtGame::GameActorProxy::TICK_LOCAL_INVOKABLE



- Implement TickLocal() behavior

```
void TankActor::TickLocal(const Message& tickMessage)
{
   // determine direction & velocity
   // move the tank – use direction, speed & ground clamp
   // Notice that tick is actually a message!
}
```

- Implement TickRemote() behavior

```
void TankActor::TickRemote(const Message& tickMessage)
{
   // move the tank – use direction, speed & ground clamp
}
```



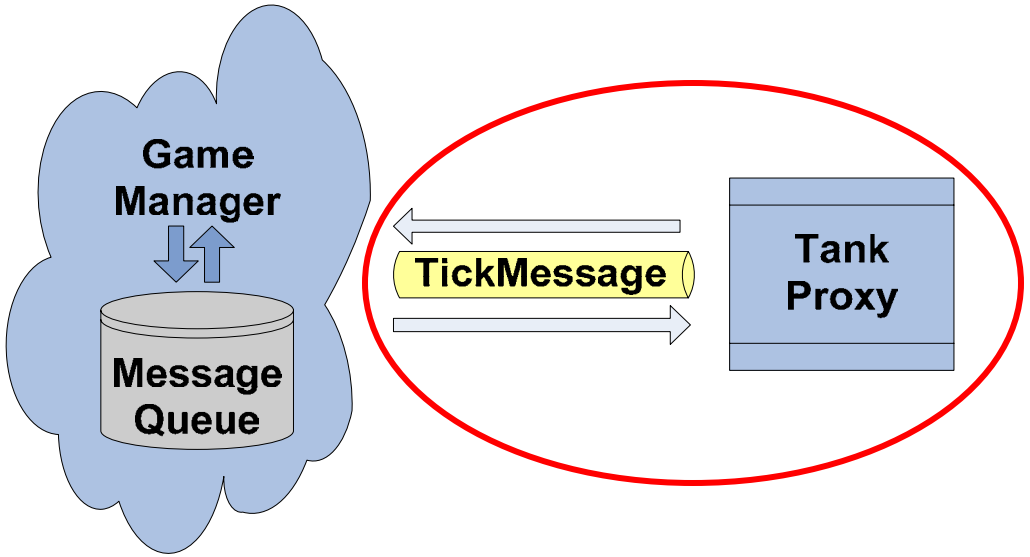


Tasks (9 of 11)
FireFighterMission
AcquireGear
Don fire suit
Activate SCBA
Acquire fire hos
SafelyEnterEngine
Throw primary
Throw secondar
Trip fuel valve

Part 3 – CODE TIME 2

Visual Studio – "ProjectBeta"

1. Register for TICK_LOCAL & TICK_REMOTE Messages in TankActorProxy::OnEnteredWorld
2. Implement TickLocal()
3. Implement TickRemote()



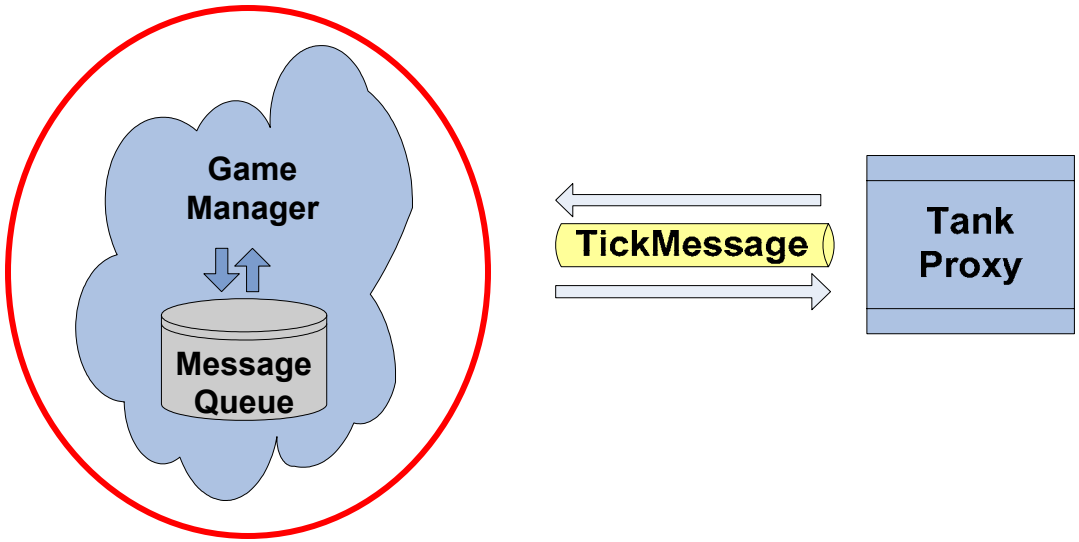
Part 3 – Game Start

- How do we make a running app?
  - Need to load & initialize our objects, including our map
  - Need to create our Components Game Manager & run it!
- Enter the Game Start executable
  - Self contained executable for running a Delta3D game
  - Loads your custom GameEntryPoint class
    - Part of your Actor Library
    - Is your new “main()”
  - Only need to create one class to start everything!
- GameEntryPoint - methods (1 required):
  - Initialize() – command line params & set paths
  - CreateGameManager() – Create your GM.
  - **OnStartup()** – Final chance to init before main game loop. Create components, register Message types, move the Camera, load your map.
- Note – GameEntryPoint encapsulates dtABC::Application

Part 3 – CODE TIME 3

Visual Studio – "ProjectBeta"

1. Create MyGameEntryPoint
    1. Load our STAGE map
        1. Set the project context
        2. Tell the game manager to change map
    2. Attach our Camera
        1. Find our tank proxy using GM.FindActorsByName()
        2. Add the camera to the tank actor with AddChild()
    3. Add a FlyMotionModel to our camera (optional)
2. Run our first app!



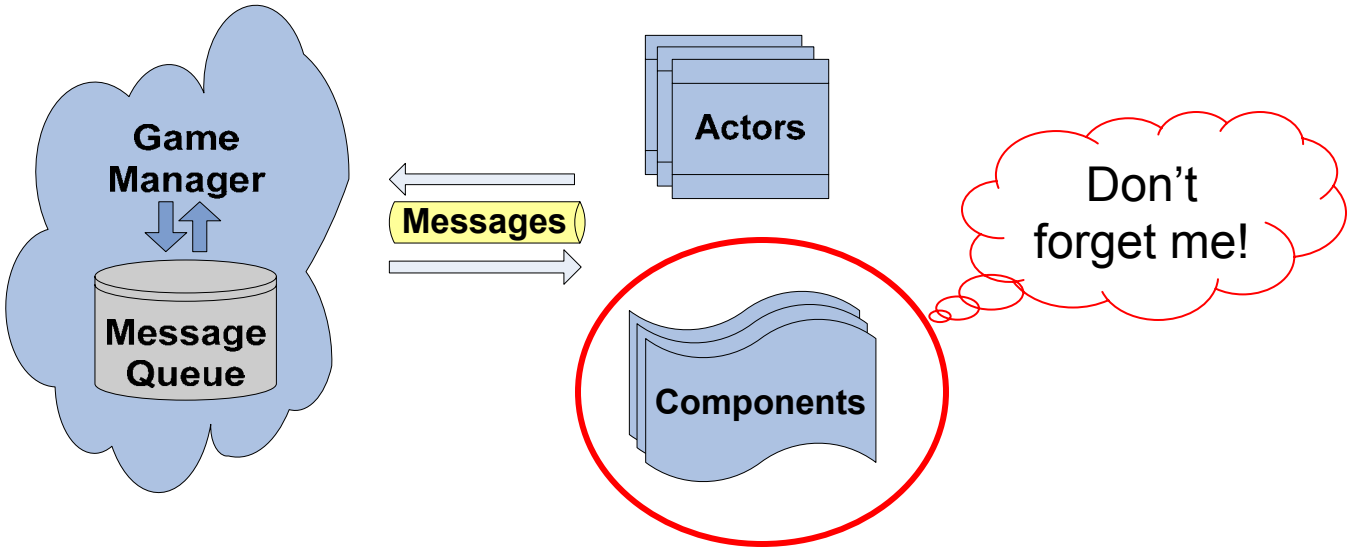
TUTORIAL
PART 4

Game Manager (GM) Components

Hint: Theory and Code

Part 4 – Review

- What do we have so far?
  - Game Manager
  - Game Entry Point
  - Game Actors
  - We have a map, a library, an executable, …
- What are we missing?



Part 4 – New Concepts

- GMComponent base class
  - High level behavior – more abstract than an actor
  - Use a component for system level behavior such as: networking, logging & playback, HUD & UI, keyboard/mouse input, dead reckoning
  - Receives ALL messages – without registering
    - Actors only receive messages if they register for them
    - Components get messages BEFORE actors
  - One instance of each component
  - Two important methods to override
    - ProcessMessage() – handle local messages
    - DispatchNetworkMessage() – handle network messages
  - Component Priority – affects message handling order - some components need to know first!






Part 4 - Provided GM Components

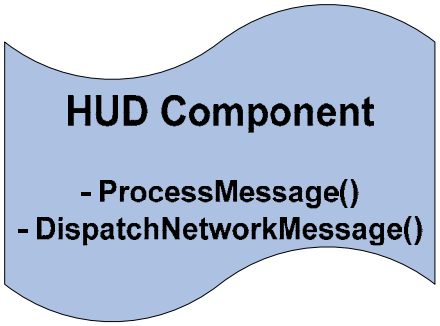
- **DefaultMessageProcessor** – actor updates, creation, & entered world (required)
- **BaseInputComponent** – base behavior for handling keyboard/mouse
- **DeadReckoningComponent** – smooth movement for remote actors, reduce net traffic
- **TaskComponent** – task tracking for objectives/score and LMS connectivity
- **LogController** (client) & **ServerLoggerComponent** (server) – logging & playback behavior for AAR
- **RulesComponent** – baseline behavior for routing ProcessMessage vs DispatchNetworkMessage
- **HLAComponent** – connectivity and interface to HLA. Converts Game messages to HLA traffic





Part 4 – Let's make one

- Build your own component
  - Single entry for handling messages – improves efficiency and facilitates optimization
  - Can manage groups of actors
  - Can perform global processing
  - Examples – HUD, Input, Network, …



- Two main methods (both optional):

```
void ProcessMessage(const Message& message)
{
}

void DispatchNetworkMessage(const Message& message)
{
}
```





Part 4 – HUD Component

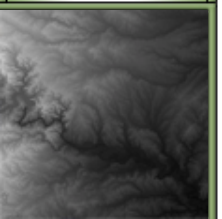
- Our Task - build a HUD component
  - Built using Crazy Eddie's UI tool
  - Add UI controls to display
    - Simulation Time
    - Last Message that was sent
    - Total # of messages sent by the GM
  - Handle messages in ProcessMessage
    - Listen for TICK_LOCAL & all others…
    - Note - no networking messages needed
  - Advanced Note – add our CEGUI UI drawable in OnAddedToGM() to ensure scene is ready

HUD Component

- ProcessMessage()
- UpdateSimTime()
- UpdateNumMessages()
- UpdateLastMessageName()



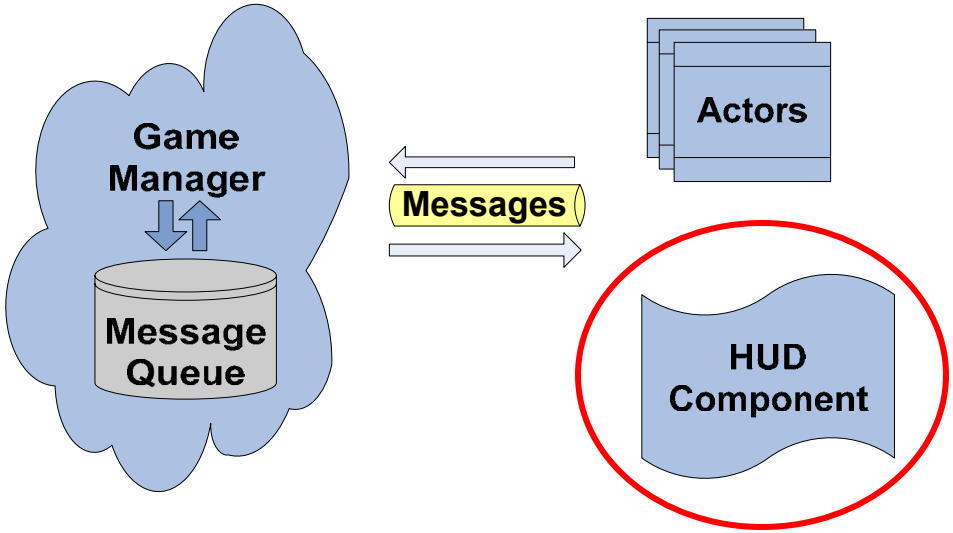


Part 4 – CODE TIME 4

Visual Studio – "ProjectGamma"

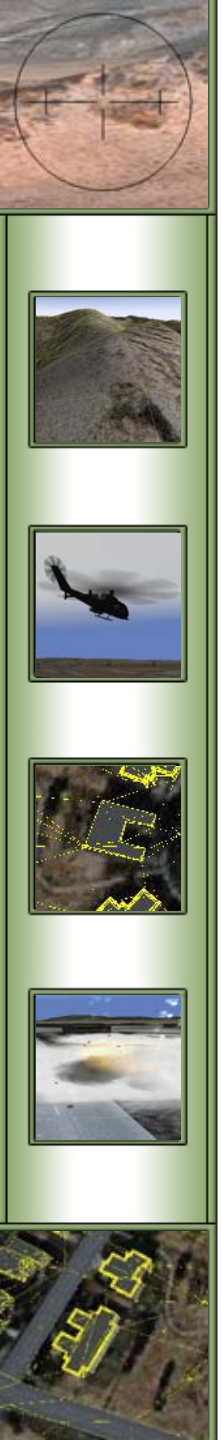
1. Create our HUD Component
    1. Find controls for Sim Time, Last Message, & Num Messages
    2. Implement ProcessMessage()
        1. Handle TICK_LOCAL – update sim time
        2. Handle TICK_REMOTE - do nothing
        3. Handle all others – update LastMsg and NumMsgs
2. Add our component to the Game Manager
3. Run our app!



TUTORIAL
PART 5

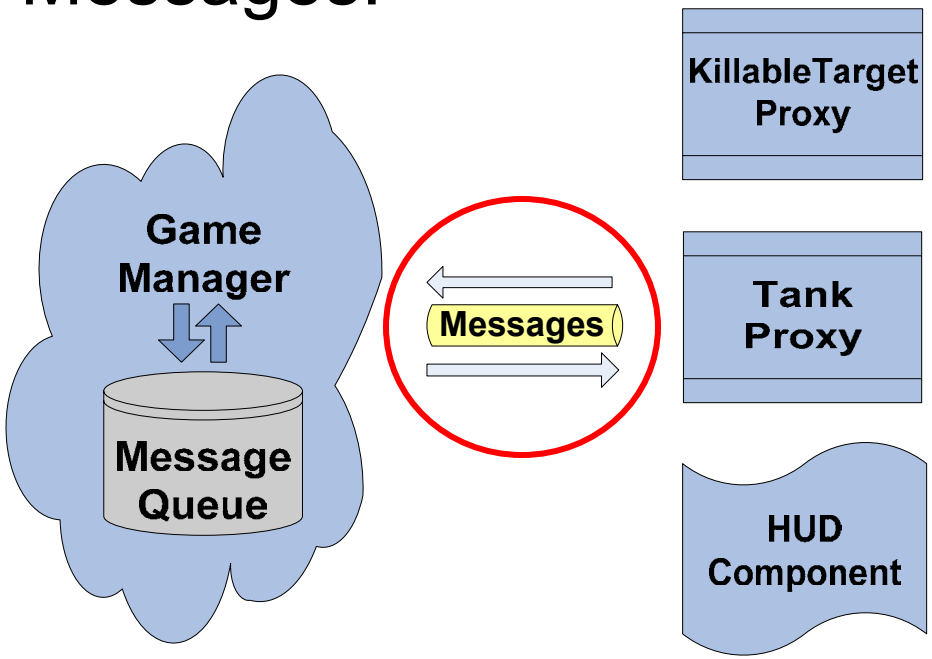
Messages

Hint: Theory and Code



Part 5 – Review

- What have we done!
  - Game Actors
  - Game Entry Point
  - Game Manager
  - GM Component
- Time for Messages!

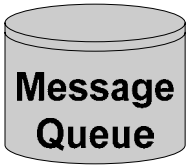







Part 5 – Messages

- Why do we need messages?
  - Generic way to send behavior and data.
  - Decouples actors and components
  - Expandable – you can create custom messages and parameters
  - Serializable - network, logging
  - Trappable by Game Actors or Components
  - Managed by the GM
- When should we use them?
  - To convey property changes – “but wait, we never did that” – you’re right! We will now!
  - To send requests across a network (i.e. LoggerController)
  - To cause events to happen
  - To ensure loose coupling








Part 5 – New Concepts

- Message Type
  - Just like an Actor Type – generically defines a type of message
- Message Class
  - Many messages have their own class with parameters (like actor properties)
  - You can reuse a class for multiple Types (e.g. have the same parameters, just different meaning) *
- Message Factory
  - Just like the actor registry. Tells the GM what message types exist.
  - Use the factory to create new messages
  - Register multiple types using the same Message class *
- Even Tick is a message!
  - Messages can be used lots of ways. The GM can handle lots of them per frame (one unit test sends 5000 actor update messages with 38 properties each in < 2 seconds).
  - Limit your processing in Tick, especially if lots of objects
    - E.g. ground clamping or dead reckoning

* Lots of message classes are reused – ex. Tick Local and Tick Remote. Also Actor Updated, Actor Created, and Actor Deleted.






Part 5 – Game Events

- Game Events
  - A simple string identifier for some significant event
  - Ex: “Tank Fired”, “Speed Boost”, “ToggleEngine”
  - Register events on dtDAL::GameEventManager
    - new dtDAL::GameEvent(“ToggleEngine”)
    - dtDAL::GameEventManager::GetInstance().AddEvent(…);
  - Look up with GameEventManager.FindEvent(“name”)
  - Game Events can be defined in STAGE
- GameEventMessage Class
  - A special message for sending game events
  - Has a parameter for the event.

```
dtCore::RefPtr<dtGame::GameEventMessage> eventMsg;
GetGameManager()->GetMessageFactory().CreateMessage(
        dtGame::MessageType::INFO_GAME_EVENT, eventMsg);

eventMsg->SetGameEvent(…);
GetGameManager()->SendMessage(*eventMsg);
```






Part 5 – Input Component

- Our Task - build an Input component
  - Create & Register some Game Events
  - Handle when the key is pressed
  - Fire a GameEvent for
    - “Toggle Engine”
    - “Speed Boost”
    - “Tank Fired”
  - Add our Component to the Game Manager
- Handle GameEvents on our Tank
  - Register for INFO_GAME_EVENT using the default ProcessMessage() invokable
  - React to “Toggle Engine” & “Speed Boost” GameEvents in ProcessMessage()






Part 5 – CODE TIME 5

Visual Studio – "ProjectDelta"

1. Create our Input Component
   1. Create GameEvents

      "ToggleEngine", "SpeedBoost", & "TankFired"
   2. Implement FireGameEvent(&event) - Create & send the GameEventMessage
   3. Implement HandleKeyPressed() & call FireGameEvent(…)
2. Modify our TankActor
   1. Register for INFO_GAME_EVENT in the proxy.OnEnteredWorld()
   2. Implement ProcessMessage() on the Actor – check for GameEventMessage
      1. Check the message name
         1. For 'ToggleEngine" – flip the mIsEngineRunning flag and enable mDust appropriately
         2. For 'SpeedBoost' – add -5.0f to the velocity – negative because our model is backwards, hah!!
3. Add our component to the Game Manager
4. Run our app!






Part 5 – Creating Custom Messages

- Goal – We want to send a message when the tank targets a blimp (aka KillableTargetActor). The message should have a parameter identifying the new target.
- * Note – Creating custom message types is harder than anything else we have covered.
- Our Task – Build a TargetChangedMessage
  - Create a new Message Type (like ActorType)
  - Create a new Message Class (like an Actor)
  - Add an actor Id parameter
  - Register our message type with the GM MessageFactory
  - Send the message from the TankActor

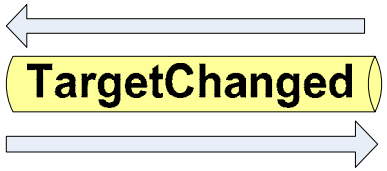





Part 5 – Message for you, sir!

- How to make a new message class?

```
class TargetChangedMessage : public Message
{
}
```



- How to add parameters?

```
TargetChangedMessage::TargetChangedMessage()
{
   AddParameter(new ActorMessageParameter
        (“NewTargetUniqueId"));
}
```

- Add getter and setter

```
void SetNewTargetUniqueId( const dtCore::UniqueId& uniqueId );

const dtCore::UniqueId& GetNewTargetUniqueId() const;
```



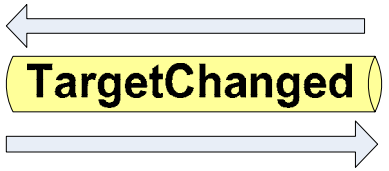



Part 5 – Message Types

- Make our Message Type

```
const TutorialMessageType TARGET_CHANGED(
    “Target Changed”, “Info",
    “Sent by the tank when the target changes.", 1025);
```

- Register our Message Type




```
dtGame::MessageFactory::RegisterMessageType
   < TargetChangedMessage >(TARGET_CHANGED);
```

- Message Type Id
  - Uniquely identifies message type
  - Consistent via code
  - Start yours at 1025
    - MessageType::USER_DEFINED_MESSAGE_TYPE + 1
    - Do this in the implementation for your custom MessageType

Final Demo and Questions

THE END

Building Game-Based Trainers with the Delta3D Game Manager

Thank you for attending our tutorial!

Please come to our booths to learn more about Delta3D!

Part 5 – CODE TIME 6

Visual Studio – "ProjectDelta"

1. Add KillableTargetActor.cpp and KillableTargetActor.h to the project
2. Create our Message Class
   1. Call AddParameter() in our message constructor
   2. Look at the Set and Get for the target Id.
3. Create new Message Type
   1. Create a static message type just like we did for our ActorType – "TANK_TARGET_CHANGED"
   2. Implement RegisterMessageTypes and add our type to the message factory
4. From GameEntryPoint – call TutorialMessageType::RegisterMessageTypes
5. Send new Message from Hover Tank in FireTargetChangedMessage()
   1. Create the TargetChangedMessage like we did for a game event message
   2. Set the NewTargetUniqueId() with mCurrentTargetId
   3. Set the about actor id to this actor's GetUniqueId()
   4. Send the message
6. Run the App!