G.O.D.

G.O.D. (God of Duality)

Team Choreopower - Christian Arca, Mark Aversa, Aaron Cooper, Ryan Michaels, Eujain Ting

January 21, 2009

Last Updated: February 10, 2009

G.O.D.

TDD

Technology Overview

The base of the game engine uses Ogre3D. Ogre3D provides a full set of libraries for graphics rendering, model & animation loading, and input management. Ogre will allow for easy to manage cameras and scenes.

To manage the code base, art, and documentation, an SVN server has been setup. Members are expected to make small and frequent updates for code. This frequent updating creates a better revision database.

Coding Practices

File Structure

To help organize files, most of the development is done in a Visual Studio solution. Classes are organized into subfolders depending on the category they belong to. For example, all entities, players enemies, exist under the entities folder. Each class is to be written in a separate file. That file must contain a description of the functionality of the class. If a class or a function in a class is getting too large, it should be broken up into sensible sub parts.

Code Standards

- File names will correspond to the class they contain
- All member variables must follow the standard "mClassVar"
- Keep functions short and concise. If a function becomes more than 100 lines of code, it should be broken up.
- Do not clutter high level classes such as the main game class. If it is getting clutter, that code should be put into new classes

- NO SINGLETONS
- Curly braces start at the end of the line, not on a new line. Ex:

  ```
  void function() {

  }
  ```

- Make function names clear to what they do. A longer descriptive name is better than a shorter cryptic name.
- Cleanup all memory in decontructors

Main Game Structure

Ogre Application & The Game

The application is setup and initialized in a separate class using Ogre. The game is then added as a listener to the Ogre engine. Ogre calls a function at the beginning and end of the frame for every Ogre frame listener. The game has a main update where all major managers are called to update. Those include level, player manager, enemy manager, etc…

Game Services

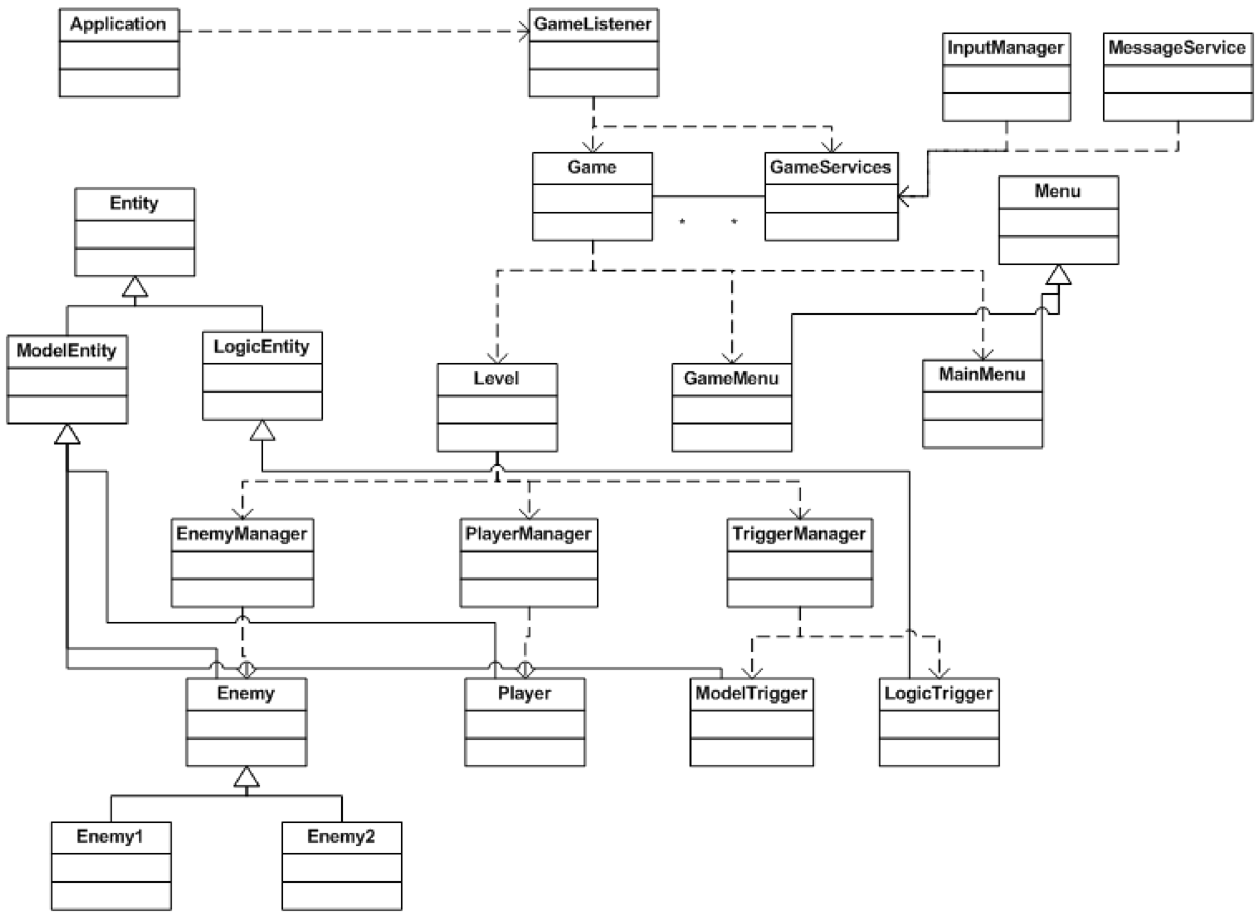
There is a class called game services that should be injected into every class in the game. Game services holds objects that generally need to be accessed by almost every class. Those objects include the Ogre root object, input manager, and message service. The Ogre root object holds access to all scene data, cameras, and viewports. The input manager is a wrapper class around the OIS input libraries to fit the specific needs of the game. It can be queried to find the state of the keyboard, mouse, and joysticks. The input manager sends out messages when a button state is changed.

Messaging Service

The messaging service is the heart of the events system in the game. An instance of the messaging service will be held in game services so all object can access it. The service will be responsible for registering and sending events throughout the system. To send out an event, any class calls the dispatch message function. The function has only one parameter, the message object. All messages will inherit from a base message interface. A message object will contain all the data that needs to be sent and will act as an id for the message.

To receive events, objects will register (become listeners) with the messaging service for a specific message type. When an object registers, it must specify a message type it is looking for and function pointer to call when the event happens. When the event occurs, the messaging service will call the specified function pointer.

UML Diagram



Class Descriptions

Application
It is responsible for initializing the Ogre engine and registering the game listener with Ogre.

Game Listener
This is merely a connection between the Ogre update loop and the game. Responsible for initializing and updating the game.

Game Services
Acting as a holder for global like objects, game services replaces the need for singletons. The game services object should contain only a few objects that need to be accessed by a majority of classes. Game services should be past through the constructors of all object in the game.

Input Manager
A wrapper around the OIS input libraries, the input manager handles initializing the input devices and sending out event message relating to the change in device states.

Message Service
This is the event management system. Objects will send and listen for messages through this system. To create a message, objects will call a send message function passing a specific message. All objects listening for the message type will receive it. To register for a message, object can register as a listener for a specific message. When objects register, they pass a function pointer to call when the event happens. Objects can dynamically stop listening for messages as well.

Game
Starting point for all game activities, this class initializes the ogre root object, the game services, scene manager, and the level. This class should not be bloated with game code. Only large systems should be initialized and updated here.

Player Manager
Handles the initializing and updating of players.

Enemy Manager

Responsible for spawning, updating and keeping track of enemies. The enemy manager should be keeping track and guiding enemies.

Trigger Manager

Keeps track of all puzzle triggers in the level. Holds the states of puzzles and keeps track of what puzzle and active and inactive.

Level Manager

Responsible for loading the level file, initializing the doodads and puzzles throughout the level. Level will contain all of the managers and will be responsible for starting the player, enemy and trigger managers.

Entity

Base class that all in game entities should inherit from. Takes care of basic ogre initialization and model loading for all entities.