DESK WARS TECHNICAL DESIGN DOCUMENT

Authors:

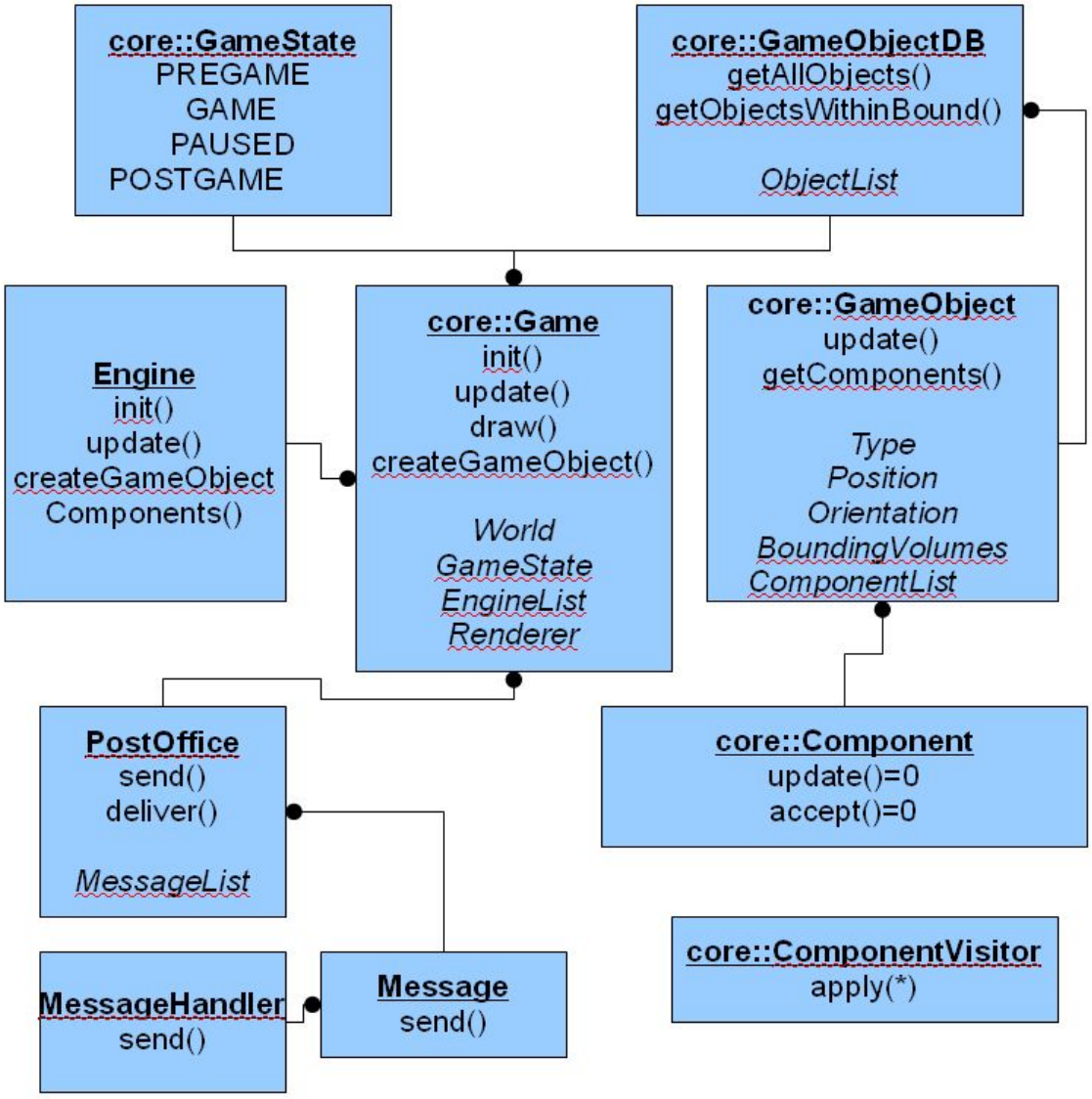
Alex Pecoraro, Ejaz Mudassir, and Katie Mazzoli

Creation Date: 10/19/2009

Table of Contents

Overview

The Desk Wars Game Engine Architecture is depicted by the diagram below. The core classes are the Game, Engine, GameObjectDB, GameObject, and Component classes.



The main class is the Game class. It provides the basic framework to implement the Desk Wars game. It has a list of instances of the Engine class. These Engine instances are subclasses of the Engine class, which is an abstract interface class. The subclasses of the Engine are used to provide sub-system specific functionality such as graphics, physics, or AI. The Game class also has an instance of the GameObjectDB, which contains all the GameObjects in the Game and provides functions for querying for specific objects or getting a list of all the objects. Instances of the GameObject class are used to represent any unique object in the Desk Wars world such as the desk, the objects on the desk, the player's units, the AI's units, etc. A GameObject in and of itself doesn't know how to do anything or be anything. It only has a few basic properties such as position, orientation, bounding volume(s), and a list of Components. A GameObject's frame to frame update logic comes from the Component's that it contains. The Component class is an abstract interface class that defines an interface for implementing custom GameObject behavior. The creation of

GameObjects and their corresponding Components is a team effort between the Game class and the Game's list of Engines.

Game Initialization

The Game class will be initialized by an XML configuration file. The configuration file will specify a list of Engines that the Game will create and initialize. The format of the XML configuration file is given below:

<Game>

    <Engine class="[Name of Engine class]" config="[path to configuration file for this Engine]" />

    ...

</Game>

The Game class will create an instance of the Engine and then call the Engine's init() function, passing to the newly created Engine the path to it's configuration file. The Engine init() function returns a Status object whose boolean member variable is set to true if the initialization is successful and false if it is not. The Status class also contains a std::stringstream that will allow the Engine to specify an error message if the initialization is not successful.

Main Game Loop

At the beginning of each Game::update() call the Game class will ask the PostOffice to deliver Messages to any instances of the MessageHandler's. Then it will call update() on the GameObjectDB, which will then call update() on each GameObject, which will then call update() on each of it's Components. Finally the Game will allow each Engine to update by calling update() on it's list of Engines. After the update stage has completed the Game will render the frame by passing the GameObjectDB to the Renderer. The algorithm is describing in pseudo code below:

Game::update()

    PostOffice::deliver()

    For each GameObject in the GameObjectDB

GameObject::update() – this funtion calls update() on the Components

For each Engine in the Game

Engine::update()

Game::draw()

Renderer::render()

Core Game Framework

This section describes the classes that make up the core framework of the Desk Wars Game.

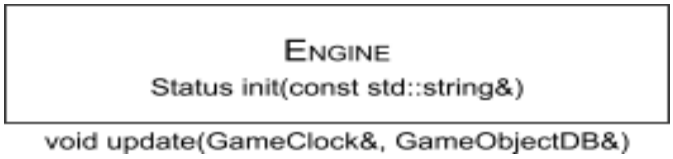
Game Class

The purpose of the Game class is to act as a GameObject factory and to manage the game initialization, the main game loop, consisting of an update and draw stage, and the game shutdown.

Methods:

- Status init(const std::string&)
  - The init function takes a string that specifies a path to a game configuration file. The game configuration file specifies a list of engines to instantiate and a path to each engine's configuration file. Returns a Status GameObject specifying the success or failure of the call and an error message if needed.
- GameObject* createGameObject()
  - This function is used to create a new GameObject. The newly created GameObject will then be passed to each of the Game's Engines so that they can each create Components for the GameObject via their createGameObjectComponents().
- void update()
  - This function calls the PostOffice's deliver() function to deliver any messages sent in the previous update loop. Then it calls the GameObjectDatabase's update

Engine Class



The Engine class is an abstract interface class that provides a common interface for implementation of custom Game subsystems. The Graphics, Physics, AI, and Gameplay subsystems will all be implemented as subclasses of the Engine class. The Engine class defines two pure virtual functions that must be overridden by subclasses, the init() and update() function. The init() function is called by the Game class after the Engine is constructed and it is passed a path to a file that contains custom configuration parameters. The format of the configuration file can be unique to the specific Engine class, but a ConfigFile class that implements a very simple param=value format is provided in the core framework if a unique format is not required by the Engine.

Renderer Class

The renderer class is an abstract class that defines an interface for rendering the GameObjects to the screen. It's render() function is a pure virtual function that must be overridden by an concrete implementation classes. The render() function's arguments are the GameObjectDB and the GameClock

PostOffice, Messages, MessageHandler Classes

Separate subsystems such as the AI, Graphics, or Physics communicate with each other via Messages (more specifically subclasses of Message) and the PostOffice. In order for the Message to be useful a Message class must also define a subclass of the MessageHandler class that is specific to the new Message. Anyone that wants to receive the new message would register an instance of the MessageHandler class with the PostOffice's registerMessageHandler() function.

GameObjectDB

The GameObjectDB class is a spatially organized database of the GameObjects. It defines functions that can be used to query for GameObect's filtered by type and location. The query functions defined by the GameObjectDB are described below.

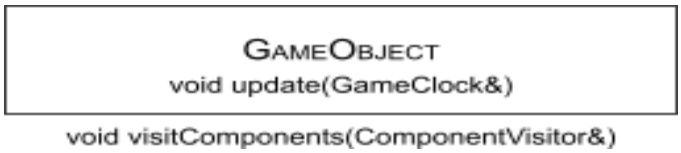
Methods:

- **size_t getAllGameObjects**(const GameObjectFilter&, GameObjectList&)
  - Arguments:
    - const GameObjectFilter& (in) – an instance of ObjectTypeFilter that is responsible for filtering out GameObject's based on their type (note: type is determined by the class of Component's contained in the GameObject's Component list).
    - GameObjectList& (in/out) – a list of GameObjects that are found and make it past the filter.
  - Description:
    - Finds all the GameObjects in the game that pass the ObjectTypeFilter.
    - Returns the number of GameObjects found and stores pointers to those GameObjects in the list.
- **size_t getGameObjectsWithinBounds**(const BoundingVolume&, const GameObjectFilter&, GameObjectList&)
  - Arguments:
    - const BoundingVolume& (in) – a bounding volume that specifies the 3d area that a GameObject must be inside of in order to be returned.
    - const GameObjectFilter& (in) - an instance of ObjectTypeFilter that is responsible for filtering out GameObject's based on their type (note: type is determined by the class of Component's contained in the GameObject's Component list).
    - GameObjectList& (in/out) – a list of GameObjects that are found and make it past the filter.
  - Description:

- Finds all the GameObjeccts whose bounds are intersected by the specified bounding volume and that pass the ObjectTypeFilter and stores pointers to those GameObjects in the GameObjectList.

- Returns the number of GameObjects found.

GameObject Class



The GameObject class purpose is to represent any object that needs to be updated on a frame to frame basis, such as one of the player's units or a weapon. A GameObject has an id, type, position, orientation, and bounding volume(s) attributes. The GameObject's id is provided to it by the Game class when it is created. It's type is determined by the kinds of Component's that are attached to it. The GameObject's update() function gets as input a reference to the GameClock which contains the current update time and the previous update time. It then calls each of it's Component's update() function passing to it a reference to itself and the GameClock.

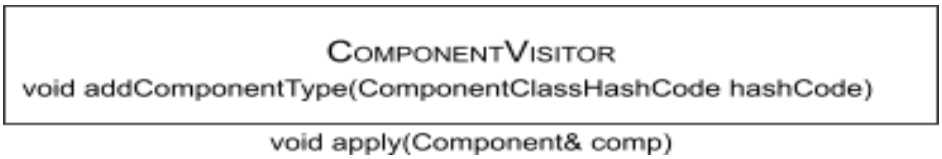
Custom attributes and update logic can be given to a GameObject by attaching a subclass of the Component class to the GameObject. For example, a GraphicsComponent could be attached to the GameObject in order to provide a visual representation for the GameObject. Furthermore, a PhysicsComponent could be attached to a GameObject to store physics related data such as the linear and rotational velocity of the GameObject. Finally, an AIComponent can be attached to a GameObject in order to give the GameObject some intelligence and allow it to navigate around the desk world.

To get access to a specific type of Component that is attached to a GameObject the GameObject::visitComponents() function is used. The visitComponents() function is passed an instance of the ComponentVisitor class.

Component Class

The component class as stated previously provides a mechanism for specifying subsystem specific attributes and logic to a particular GameObject. Custom update logic can be added by overriding the Component's update() function. The update function receives as input a reference to the GameObject that the Component is attached to and the GameClock.

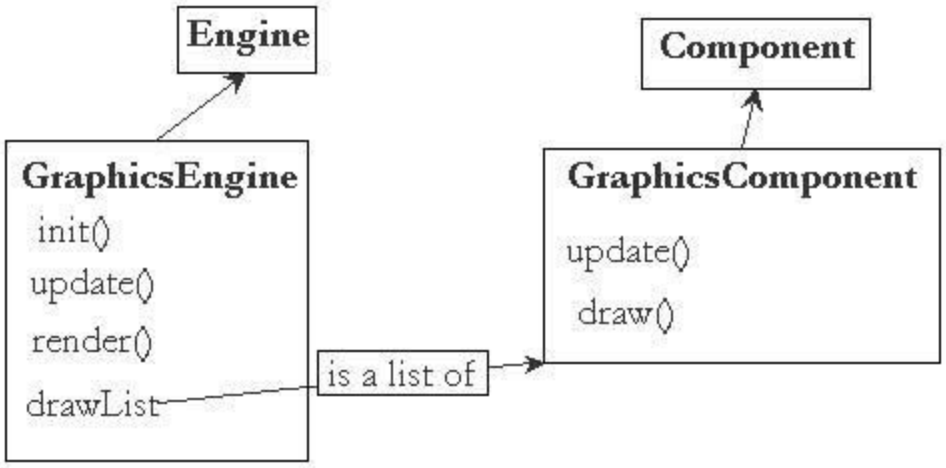
ComponentVisitor



The ComponentVisitor class is an abstract class that defines overridable apply() functions for each subclass of the Component class. The addComponentType() function is used to specify the list of Component subclasses that the ComponentVisitor should try to visit.

Graphics System

This section describes how the Desk Wards Graphics System works and the classes that are used to implement it. The diagram below shows the basic framework.



The Graphics system is implemented with primarily two classes, the GraphicsEngine class and the GraphicsComponent class. The GraphicsEngine will be a subclass of the Engine and the Renderer class and will thus implement the Engine's update() and the Renderer's render () function.

Graphics System Initialization

The intialization and configuration of the Graphics Engine will be driven by a configuration file. The purpose of the configuration file is to specify the settings for the graphics. The format of the config file is as the following:

TBD

Rendering Algorithm

The basic algorithm that the Desk Wars GraphicsEngine will use to render the GameObjects is as follows:

GraphicsEngine::update()

GameObjectDB::getAllObjectsWithinBounds() where the bounding volume is the view frustum of the camera, the ObjectTypeFilter will filter by type GraphicsComponent.

Separate the objects into groups

The player's units in one group

The enemy's units in the second group

The non-team GameObject's in the third group

For each GameObject in the enemy unit group

If it can be seen by one of the player's units then add it to the draw list

Add the player's units to the draw list

Add the non-team GameObject to the draw list

GraphicsEngine::render()

For each GraphicsComponent in the draw list

GraphicsComponent::draw()

GraphicsComponent

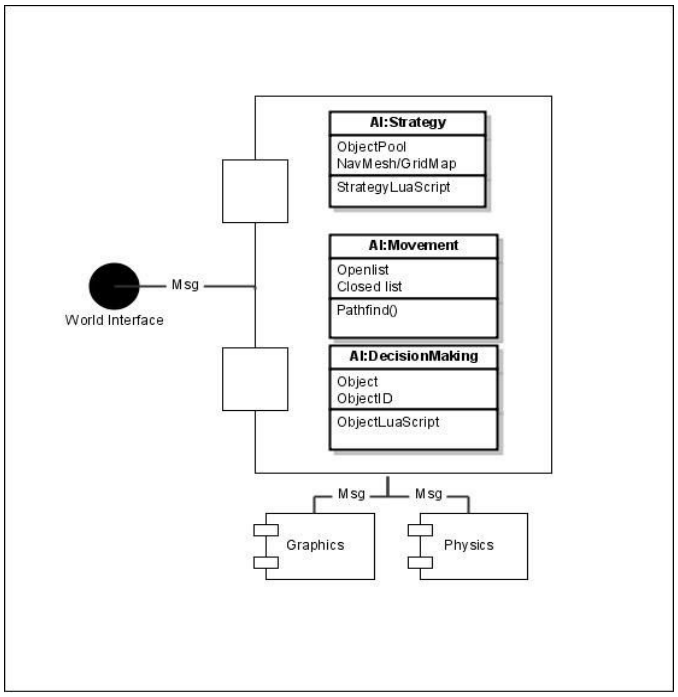
GraphicsComponentList

The GraphicsComponent is both a subclass of Component and an abstract class that defines an interface for drawing visual objects on the screen via the pure virtual function draw(). It

is utilized by the GraphicsEngine during the GraphicsEngine::update() and GraphicsEngine::render() functions. In addition, the GraphicsComponent can be composed with other GraphicsComponent's in a tree structure.

Behavior System

The behavior system is implemented with three main components. The first component is more relevant to group behavior or tactical planning of the AI team. The second and third components are correlated, as they are individual character behavior. First is the decision-making component, which is a rule-based decision, made by each individual object according to the characters they inherit. Second Component is responsible for individual path finding. The diagram shows the basic layout of the AI engine.



AI System Initialization

The initialization of the AI engine is by taking the parameters from the world interface at the beginning of the level. These include the grid map the object pool etc. The AI engine is also responsible for the initialization of the LUA interface.

AIEngine:Update()

GameObjectDB:getAllObjects(), The ObjectTypeFilter will filter by Character.

Analyze the AI list and run tactical analysis code for planning of the next move.

Run Lua script of AI Commander (Group behavior)

Separate the objects into groups

The player's units

The AI units

The Environmental GameObjects

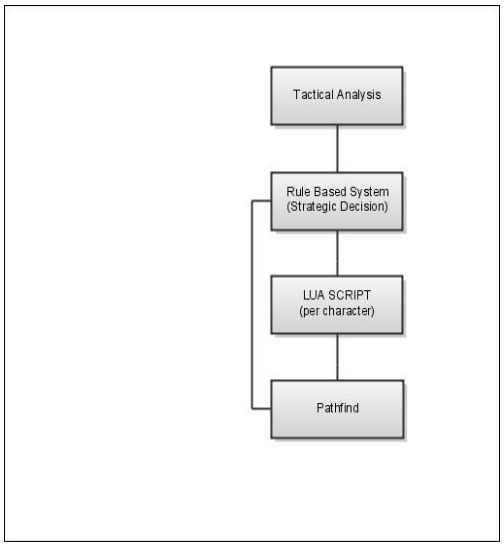
For Each GameObject

Run LUA script

Check for messages for pathfind.

Run Pathfind if GameObjectID found in Msg.

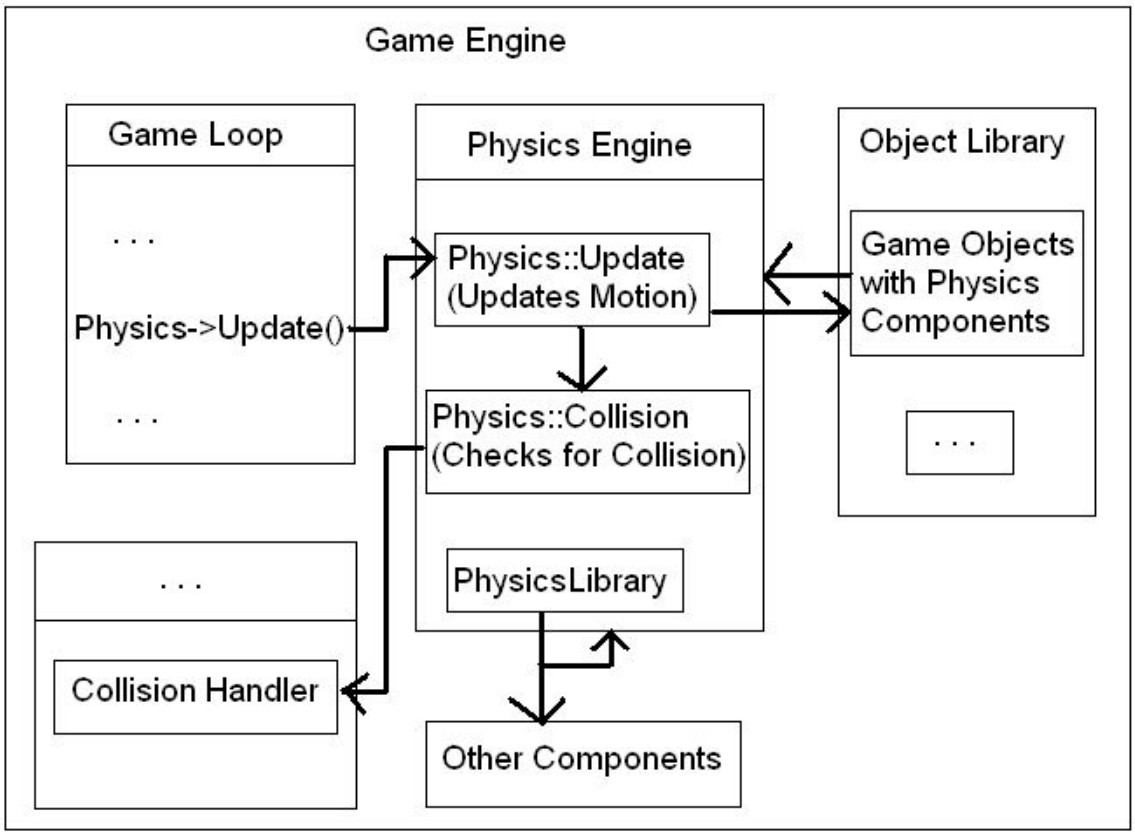
If needed Re-trace to AI Commander Script.



The following figure describes the AIEngine loop. This shows the different subset of the AIEngine and the sequence the each section is executed.

Physics System

The Physics Engine's main purpose is to update the game objects' positions and orientations in the game world based on their current parameters. Physics is also responsible for checking for collisions, and passing information on any collisions that occur to the collisions handler. Finally, the Physics Engine is where our game's math libraries are located, including special functions needed for the game that are not included in a standard math library.



PHYSICS UPDATE

The update function is the function that will be called by the game loop. This function's main purpose is to update the position orientation, and velocity of all game objects that are affected by physics. This will include updates based on gravity and momentum.

COLLISION CHECK

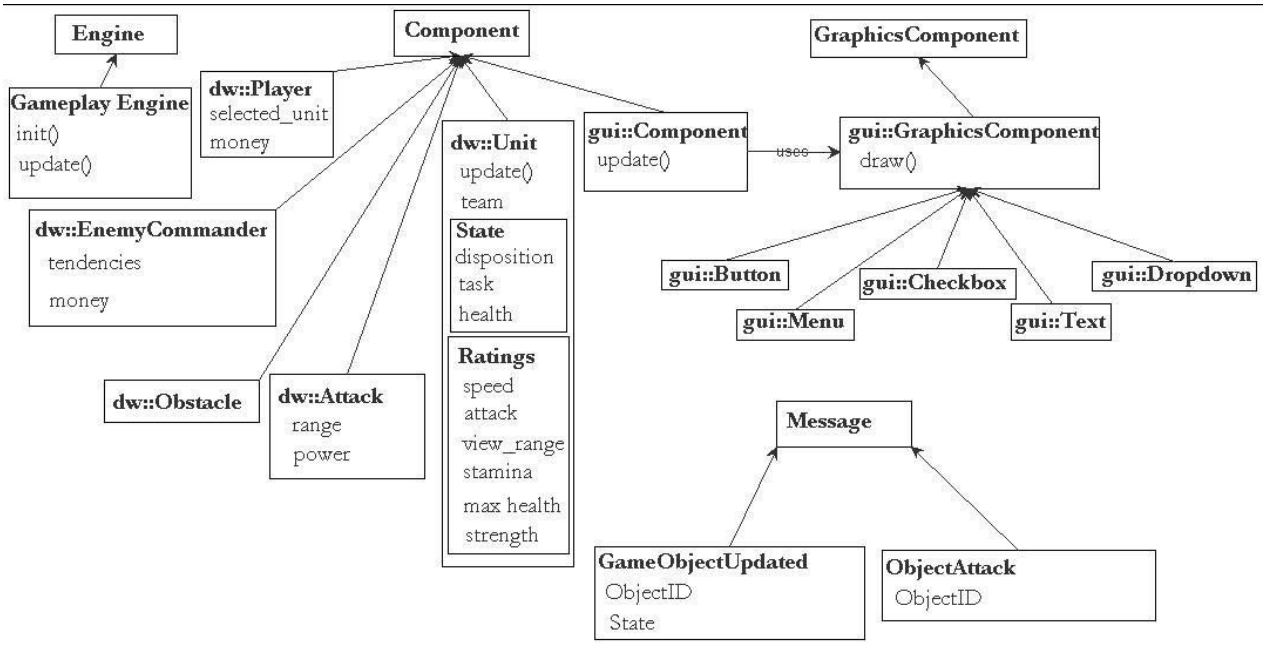
This component of the physics engine compares the game's objects and terrain to determine if there are any collisions. All collisions are reported to a collision handler, which is provided with information such as which objects have collided, and any other relevant details.

PHYSICS/MATH LIBRARY

The physics/math library is a specialized library that is used mainly by the Physics Engine. This is where any special math functions that are needed are defined. This library may also contain functions for use by other parts of the Game Engine. One such example of this would be initial position of a projectile fired by a unit, so that the projectile originates from the weapon, rather than the center of the unit's mass.

GAMEPLAY SYSTEM

This section describes the framework for implementation of the Desk Wars gameplay. The Desk Wars gameplay will be managed by a class called GamePlay, which will be a subclass of the Engine class. It will be responsible for creating the gameplay specific Component subclasses. Additionally, it is responsible for creation and management of the Desk Wars GUI. A basic overview of the classes responsible for implementing the gameplay of Desk Wars is shown below.



There are several component subclasses which are created and managed by the Gameplay Engine, including the Player, EnemyCommander, Obstacle, Attack, Unit, GUI Component, and GUI Graphics Components.

GamePlay Engine

This class is responsible for managing the gameplay logic of Desk Wars. It's init() function takes as input a path to a configuration file that specifies the following information:

- Game config data
    - Startup money for player
    - Map sequence list with reward money
- List of maps with following information for each
    - Enemy commander ID
    - Startup money for Enemy commander

- List of Enemy Commanders with following information for each
    - Tendencies
- List of GameObject configurations
    - Path to file that specifies the specific GameObject configuration
- List of unit types with the following information for each
    - Ratings
    - Cost
    - GameObject ID
    - Attack GameObject ID (GameObject used for attack action)
- Rest of this is TBD

The Gameplay class' manages the gameplay logic of DeskWars via it's message handlers and it's update() function. Its message handlers deal with the creation of new GameObject's, updating of unit state, etc. The update() function handles the flow of the game from startup GUI, to in game control, to pause GUI, to end of game GUI.

Unit Class

The Unit class is a subclass of Component. It implements functions that are common to all units such as:

setDestination(x, y) – this function sets the state of the unit to MOVING and its destination location to the position (x, y). The AI system is responsible for figuring out the exact path to take to arrive at the destination.

fire(x, y) – this function causes ObjectAttack message to be sent for this Unit. The ObjectAttack message will be handled by the Gameplay engine.