# The Shaman Engines
# Technical Design Document

Hammertank Games
Tommy A. Brosman IV, Technical Director

# Table of Contents



# Major Revision Summary

**August 26, 2009 – Tommy Brosman**
Created initial document, blocked out the sections.

**August 28, 2009 – Tommy Brosman**
Revised engine summary.

**September 2, 2009 – Tommy Brosman**
Added more module descriptions.

**September 7, 2009 – Tommy Brosman**
Added coding standards. Started the section on the scenegraph.

**September 10, 2009 – Tommy Brosman**
Redesigned the way the scenegraph and ObjectManager work based on input from Chris Peters. Added SoundManager.

**September 11, 2009 – Tommy Brosman**
Added the rest of the module descriptions.

**September 14, 2009 – Tommy Brosman**
Added in a diagram of how bone hierarchies work, added a few more bits to the coding standards. Added the section on the GameLoop. Also added a table of contents and a footer (for legal purposes).

**September 25, 2009 – Tommy Brosman**
Added in the Tools and Pre-Built Libraries section.

**October 3, 2009 – Tommy Brosman**
Added versions for all tools and libraries.

**October 30, 2009 – Tommy Brosman**
Added info about SOIL in the tools and libraries section.

**November 13, 2009 – Tommy Brosman**
Added Alex's “Getting WxWidgets To Work” section.

**May 9, 2010 – Tommy Brosman**
Updated the tools/libraries list to reflect changes made up until our Gold build. Added the final gameloop/module list section. Updated the table of contents.

# Engine Summary

## Statement of Purpose

"The Shaman Engines" is a fast-paced mech shooter with high physics, animation, and graphical requirements. Like most modern games, its engine attempts to maximize efficiency by using multiple hardware threads. However, since the resources devoted to the game are somewhat limited (number of team members, development time, budget), the codebase must be approachable and easy to work with.

## Two Common Parallel Architectures

There are two major approaches to multithreaded game engines that are commonly used: the synchronous parallel model and the asynchronous parallel model [1]. The synchronous model has the advantage of being easy to understand, but it can only make a few major systems parallel. Basically, the engine branches to perform a few parallel tasks, then rejoins when it renders the results. No message system is required, and minimal locking is required since the modules are separated by concern, minimizing cross-cutting aspects.

The second model is based on message passing and update caching. The modules run in parallel, passing messages and storing update results in a cache without directly manipulating game objects. The main thread (usually the render thread) gets the latest cached result, applies it to the game objects, then renders. This approach is very non-intuitive and requires much agreement in terms of the messaging interface. However, it is much faster than the synchronous parallel model. Assuming that the modules are completely separate, virtually no locking is required.

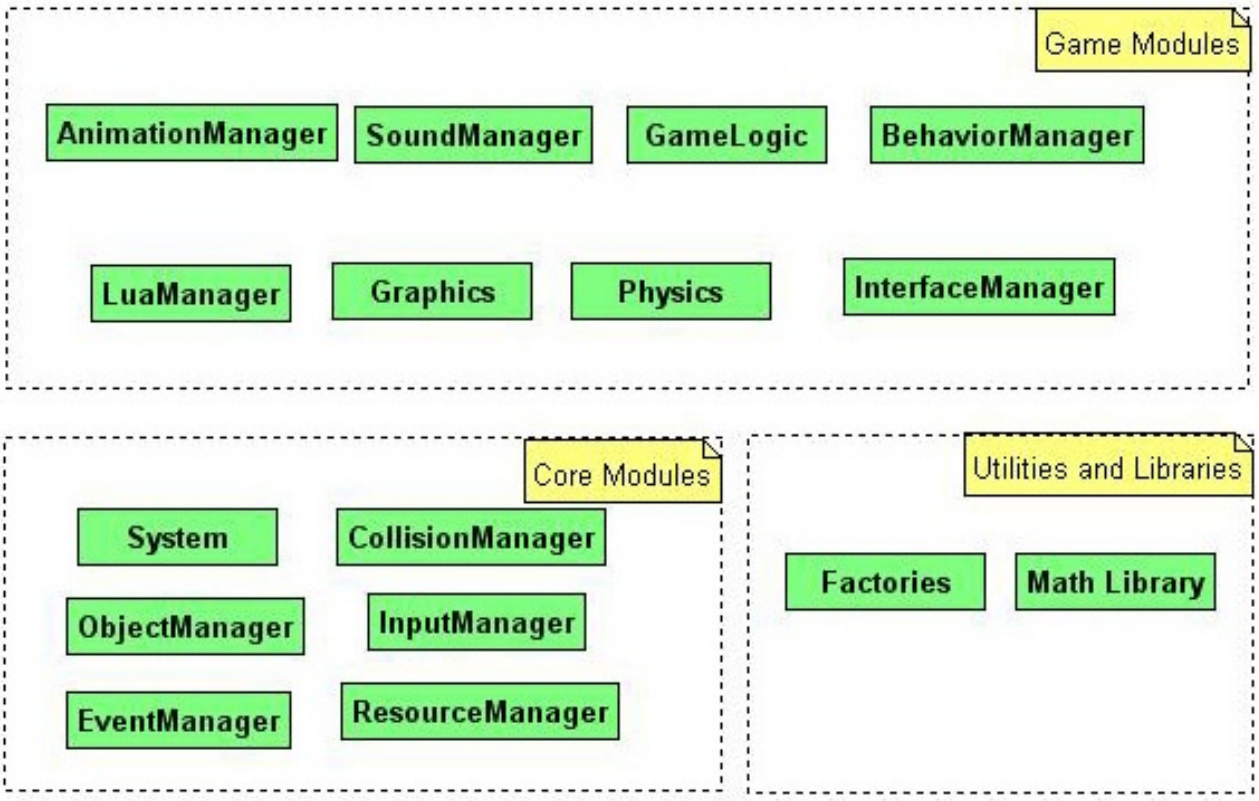
## Our Concurrency Model

For this engine, I propose a hybrid model. A set modules referred to as Core Modules make up a set of synchronized modules, with semaphores protecting access to shared data. The Core Modules hold the game objects, contain functions for launching worker threads, and other basic tasks. A second set of modules, the Game Modules, handle the game-specific tasks of the engine. Things like physics updates, collision updates, inverse kinematic solving, and graphics are all handled by these modules. The game modules are never accessed directly, instead processing a set of messages (once an update function is called). Because of this, they require little synchronization. By partitioning the game engine into synchronous and asynchronous sections, modules requiring more resources can operate on their own threads in an intuitive manner.

Further, our modular division of the various tasks in the game engine leads to high granularity and minimal cross-cutting between the high-risk concerns. This model lends itself to unit testing, which when applied properly, ultimately minimizes development time. The message system allows for iterative development to take place with very few modifications to the test plans.

Architecture Overview




Game Modules cannot be directly accessed, with the exception of their Update() function. They can directly access Core Modules as well as Utilities and libraries.

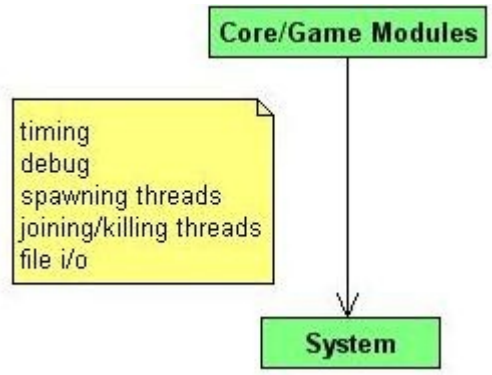
Core Modules can be directly accessed, and are highly synchronized. Usually they contain shared data used by other modules.

Utilities and Libraries are considered stateless entities, and require no synchronization.

# Core Modules

System




**Todo**

System needs to manage worker threads, maintain threadpool. Threads are no longer per-module.

**Responsibilities**

Manages threads
Used for debugging
Handles errors (instead of exceptions)
Used when reading/writing files

**Methods/Attributes**

```
public:
  void WriteDebugMsg(string msg);
  // asynch, bypasses mutex, writing synch
  void ThrowError(string msg);
  void KillAllThreads();
  void KillModuleThreads(Module::Type moduleType);

  // dumps an object as a set of bytes
  template<typename T> vector<Byte>  DumpObject<T>(T *obj);
  vector<string> ViewDebugMessages();

private:
  Semaphore                       systemSem;
  multimap<Module::Type, Thread> threads;
  vector<string>                  debugMessages;
  bool                            debugOn;
```

**Interactions**

ResourceManager and other modules needing file I/O will go through System
All other modules can use System for error handling
System can access the EventManager log
LuaManager uses System for saving stackdumps/etc

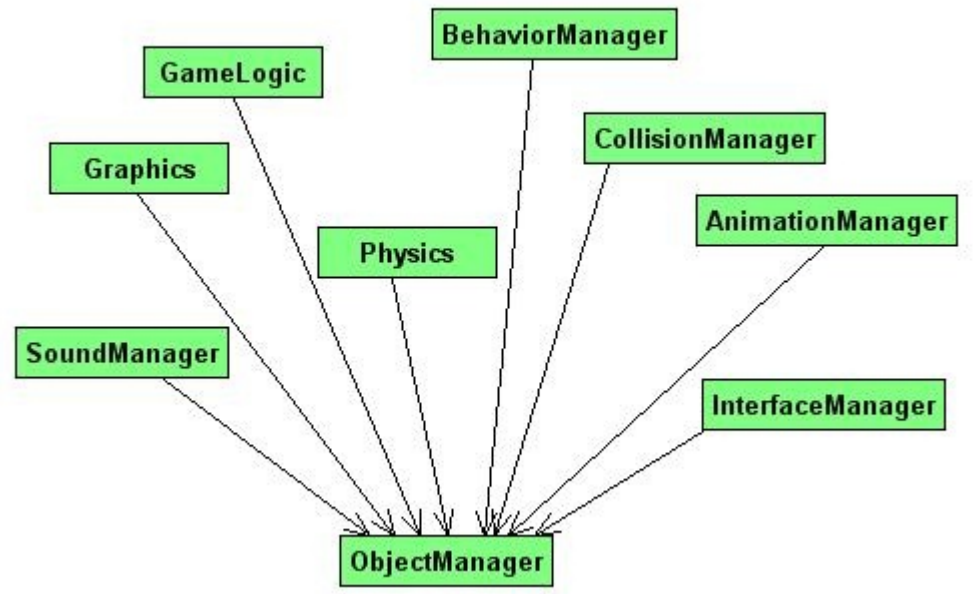
**Test Plan Outline**

Tests for thread creation/joining
  Create thread stress test
  Kill thread stress test
Tests with forced race conditions for all methods
  Trace system semaphore, make sure it's creating expected results
Tests for error handling system
  Do certain errors require different actions?
  If so, should the ThrowError call be extended for more parameters?
Test to compare output of DumpObject against other dump methods

ObjectManager




**Responsibilities**

holds the camera
holds objects
synchronizes object access

**Methods/Attributes**

```
  public:
    Update(); // unused
    Camera& GetCamera(); // synchronized if needed
    GameObject* GetRoot(); // returns objectList["ROOT"]
    // returns objectList[objName]
    GameObject* GetObject(std::string objName);
    // get the whole game object list at once
    unordered_map<GameObject *> GetGameObjectList();
    // adds the object by its name
    void AddObject(GameObject *obj);
    // deletes an object by its name, the removes references to it
    void DeleteObject(std::string objName);

    // accessors for other objects would go here (full-list accessors)

  private:
    Camera camera;
    // where the objects are actually stored
    unordered_map<std::string, GameObject*> objectList;
    // lists of static objects that are not game objects
    std::vector<Light> lightList;
    std::vector<LevelObject> levelObjectList;
```

```
    // kept here, updated by the InterfaceManager, drawn by Graphics
    std::vector<InterfaceObject *> interfaceObjectList;
```

**Interactions**

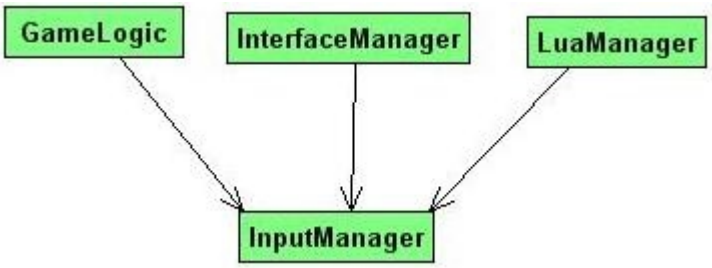
Accessed by almost all game modules, but usually for short periods of time

**Test Plan Outline**

Save and load for all objects contained
  compare results before/after load

InputManager




**Responsibilities**

Sets up functionality for input polling.
Holds callbacks for input
Holds and updates keystates
Checks to see in input devices are still connected

**Methods/Attributes**

```
public:
  void   UpdateMouse();
  // the input here is a GLFW enum
  void   KeyInput(int key, int state);
  // the input here is a Key enum
  bool   GetKey(Key key);
  bool   GetTriggerKey(Key key);
  Vector GetMousePos();
```

**Interactions**

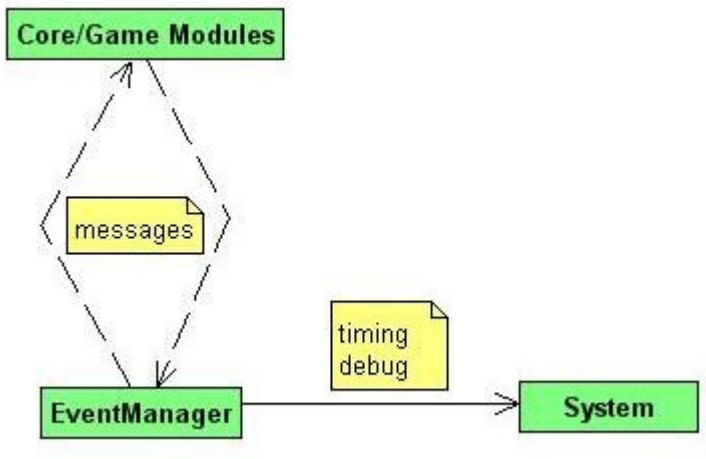
Accessed by GameLogic directly
Accessed by InterfaceManager directly
Accessed by LuaManager directly

**Test Plan Outline**

Test when keyboard and mouse are not present //  does game recognize
Test to make sure that input is being detected properly for both normal and triggered keys
Test mouse in different resolutions

EventManager




**Other Objects**

CLIENT_ID - probably an unsigned int
Message (defined in objects)

**Responsibilities**

store events that can be pushed or polled
synchronizing event publishing
registering/unregistering listeners and senders
locks everything on register/unregister; these are strictly synchronus

**Methods/Attributes**

```
public:
  // %strict_synch
  // registers an object and generates an outgoing queue for that object
  CLIENT_ID RegisterObject(std::string objName);

  // %strict_synch
  // unregisters an object and removes its outgoing queue for that object
  void UnregisterObject(CLIENT_ID id);

  // %partial_synch: inputQueue
  // %function_sync
  // publishes a message to the inputQueue
  void Publish(Message msg);

  // %partial_synch: inputQueue, outputQueues
  // %function_sync
  // if the inputQueue is not empty, publish all messages to the outputQueues
  // according to their recipient ID
  void Update();

  // %partial_synch: outputQueues[recieveID]
  // get the number of messages in the queue for a specific client
  unsigned GetMessageCount(CLIENT_ID recieveID);
```

```
    // %partial_synch: outputQueues[recieveID]
    // get a single message, causes an error if there are no messages in the queue
    Message PollMessage(CLIENT_ID recieveID);

  private:
    deque<Message> inputQueue;
    map<CLIENT_ID, vector<Message> > outputQueues; // holds all the output queues
    bimap<CLIENT_ID, string> nameTable; // keeps object names around for debugging
```

**Interactions**

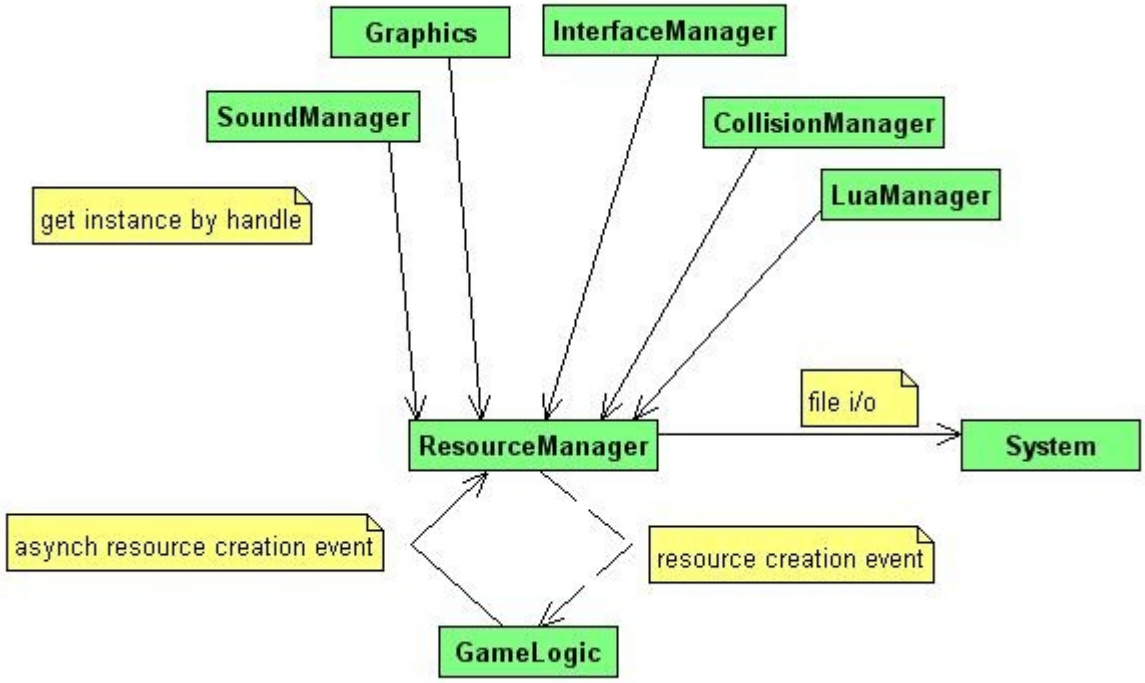
Game Modules send and receive messages through the EventManager to communicate

**Test Plan Outline**

Speed test for publishing a message
Speed test for polling a message
Stress test for publishing as many messages as possible through multiple senders
Stress test for registration functions
Stress test with multiple objects polling/publishing in parallel
Test of message system integrity for all speed/stress tests
  does the data received match the data sent?

ResourceManager




**Todo**

More detail on what kind of resources the ResourceManager handles, how it parses them after opening the file through the system.

**Responsibilities**

Loading resources form input stream or file.
Caching resources for later so that secondary calls return same instance.

**Methods/Attributes**

```
LoadResource // gets resource from stream or file unless already exists, then
             // sends a message back to the requester when loaded (async)
GetResourceHandle // gets a resource handle, loads the resource if not in memory
ReleaseResourceHandle // frees current reference but doesn't free memory
DeleteResource // only deletes the resource if there are no
GetPointerFromHandle // gets the actual object from the Handle
```

**Interactions**

Game Modules (usually GameLogic) call LoadResource to load a resource asynchronously.
Game Modules call GetPointerFromHandle read only
Game Modules call ReleaseResource to not reference it anymore.
The ResourceManager does file i/o through System
The ResourceManager sends creation events to the GameLogic

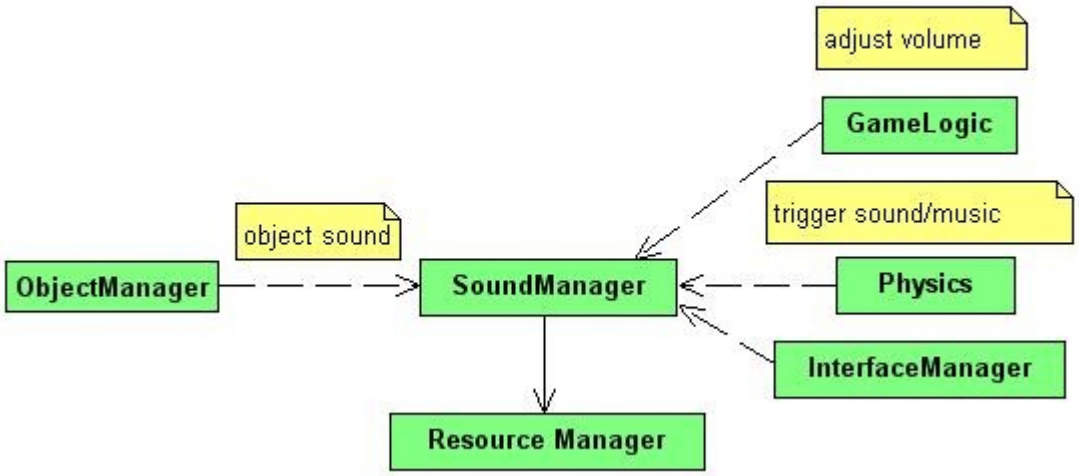
**Test Plan Outline**

Stress test a lot of resources being loaded and many object using the same resource.
Try and Delete the pointer that is returned to the Module and make sure it doesn't fail.
Try and get resources that don't exist and make sure ResourceManager handles appropriately.

# Game Modules

SoundManager




**Implementation Notes**

using FMOD, and using the C interface instead of the C++ interface

**Responsibilities**

manage channels (done through fmod priorities)
play, pause, and stop sounds
volume control

**Objects Used**

```
//sound clip
//a new instance of the sound that has a position, currently playing sounds
//Fmod handles master channel volume and priorities
Clip( Handle of sound, Handle of parent object )
```

**Methods/Attributes**

```
public:
  void Update( void );
  void Add( Clip );
  void Remove( Clip );
  void SetVolume( int Volume );
  int GetVolume( );
private:
  vector<Clip> clips; //instances of our currently playing sounds
  float masterVolume; //master volume, between 0 and 1
  //volume group, System, Voice, SFX, or Music
  ChannelGroup[4] channelGroups;
  //real volume = masterVolume * groupVolume
  float[4] groupVolume;
```

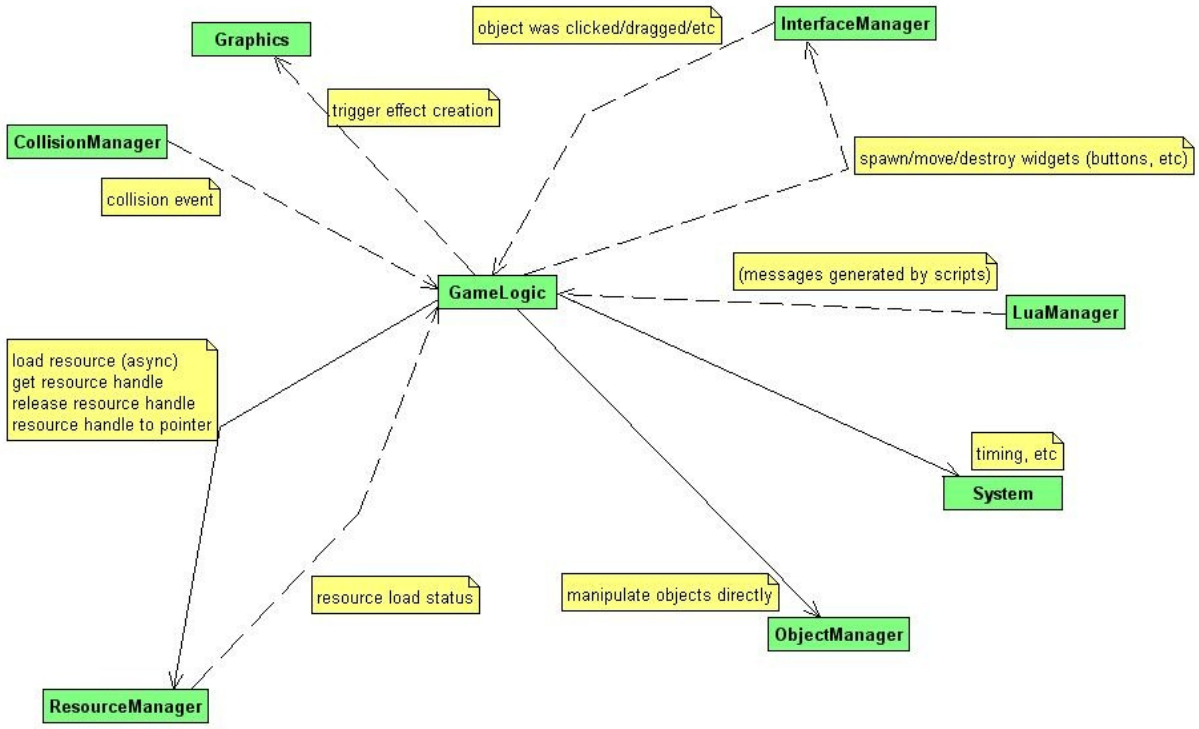
 

**Interactions**

ObjectManager - calls directly to get position of sound source
GameLogic - sends messages to change music track and play sfx
ResourceManager - calls directly to create a new instance of a sound

**Test Plan Outline**

Stress Test -play more sounds than we have channels
Stress Test -attempt to play many streaming sounds at once
Configuration Test -stress test with different numbers of hardware/software channels




**Responsibilities**

Controls the logic and flow of the game
Polls input from the InputManager and applies actions by sending appropriate messages
Directly manipulates objects based on game state/input

**Methods/Attributes**

```
public:
  void Update();
private:
  // lots of game-specific functionality in here
```

**Interactions**

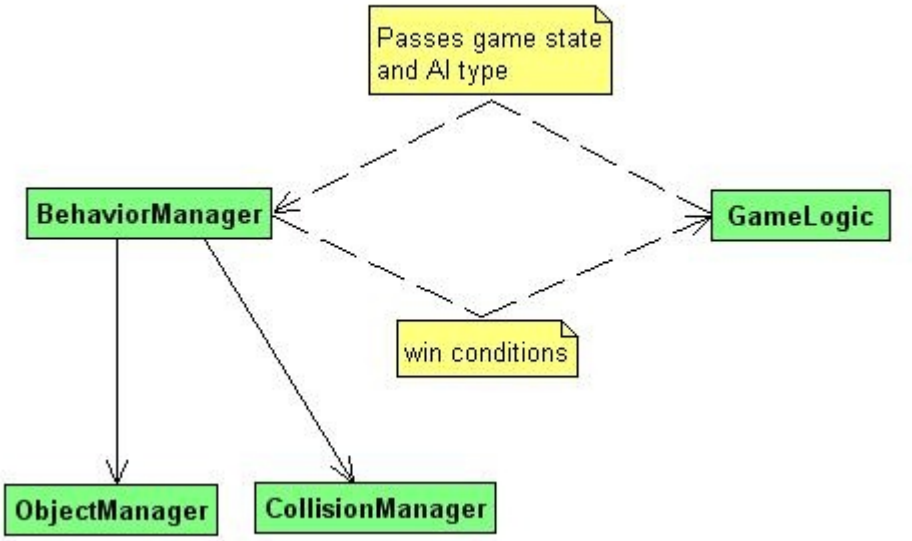
Sends spawn/move/destroy widget events to the InterfaceManager
Gets state changes from the InterfaceManager (object clicked/dragged/etc)
Gets messages generated by scripts from the LuaManager
Calls timing, debug, and other types of functions from System
Manipulates GameObjects directly in the ObjectManager
Calls functions in the ResourceManager when constructing objects (normally done from factories)
Gets load finished events from the ResourceManager
Gets collision events from the CollisionManager
Sends messages to Graphics for constructing effects and adding them to the scenegraph

**Test Plan Outline**

Make the AI play against the AI to test (most of) the game logic automatically

BehaviorManager




**Responsibilities**

Chooses which objects to run AI on
Manages swarms (stupid squads that share a worldState and mirror each other's actions)
Sets AI type
Keeps track of game state
Does A* calculations
Calculates passability/moveablility

**Objects Used**

```
  //property of the world
  WorldProperty( enum KeySymbol property, union value 4 byte value )
  //world state
  typedef worldProperty* worldState
  //Goal
  Goal( enum GoalType goal, worldState goal state, int number of conditions, float
priority)
  //Actions
  Action( enum ActionType action, float cost, worldState preconditions, worldState
effects, int number preconditions, int
number effects )
    //checks any precondition not in worldState
    bool checkProceduralPreconditions();
    void Activate(); //does action and applies effects to world state
  //Behavior
  Behavior( WorldState, vector<actions>, vector<goals> )
    void Update();
    //FSM containing two states, (move to node) and (animate)
```

**Methods/Attributes**

```
  public:
  void Update( void );
  void SetPriorities();


  private:
    unordered map objects //a tr1 hash table storing AI type, Behavior, and other
components for aiEntities
```

**Interactions**

GameLogic
- Receives game state info and AI type
- Sends win condition updates

CollisionManager
- Getting shot
- Various procedural preconditions

**Test Plan Outline**

Stress Test - how much AI can we run

Stress Test - swarms
- How many swarms?
- How many per swarm?
- Comparative - Is it even faster than non-swarm AI?

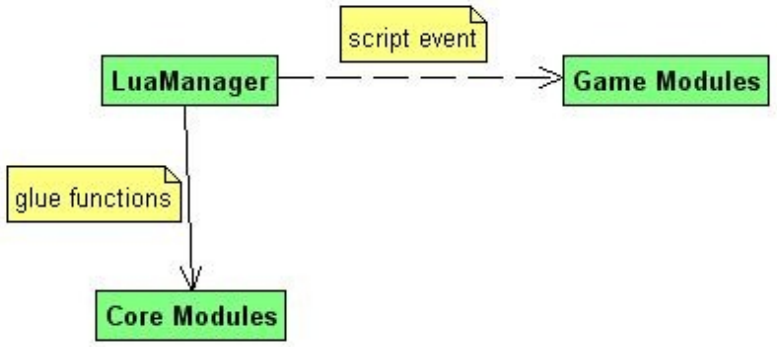
Stress Test - too many actions
- Comparative - many preconditions, few preconditions

Stress Test - too many goals

Helper tags - draw basic Behavior info above aiEntities

LuaManager




**Responsibilities**

holds the Lua state
contains some debugging functionality

**Methods/Attributes**

```
public:
  void Update(); // unused, part of base module functionality though
private:
  lua_Lstate *Lstate;
```

**Interactions**

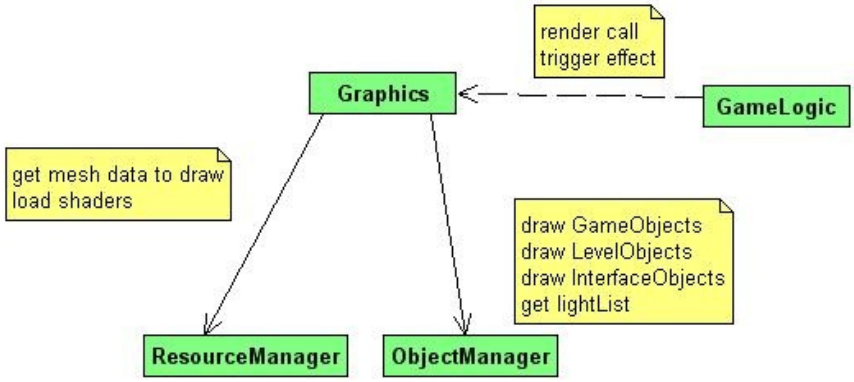
Calls functions in the Core Modules directly (using glue functions built with SwIG)
Sends/recieves messages to/from Game Modules
Recieves "run function" messages from modules
Uses the System module directly to load .lua files into memory

**Test Plan Outline**

attempt to call Lua functions from GameLogic
attempt to call game module functions from a Lua function
attempt to create and pass objects between Lua and C++ to make sure SWIG is working correctly

Graphics




**Effect class**

can be a light, particle system, or shader
lights refer to light objects in ObjectList

**Responsibilities**

render models
render particles/effects
manage OpenGL state changes (keep track of them)
manage shaders (Effects)
render interface buffer

**Methods/Attributes**

```
public:
  //Updates OpenGL and calls the Render functions on non occluded objects
  Update(???);
private:
  RenderModels(???); //Renders objects and lights
  RenderEffects(???); //Renders particle effects and other effects
  RenderInterface(???); //Renders the interface and hud

  List of OpenGL States and their status;
  List of Shaders;
```

**Interactions**

ObjectManager

**Test Plan Outline**

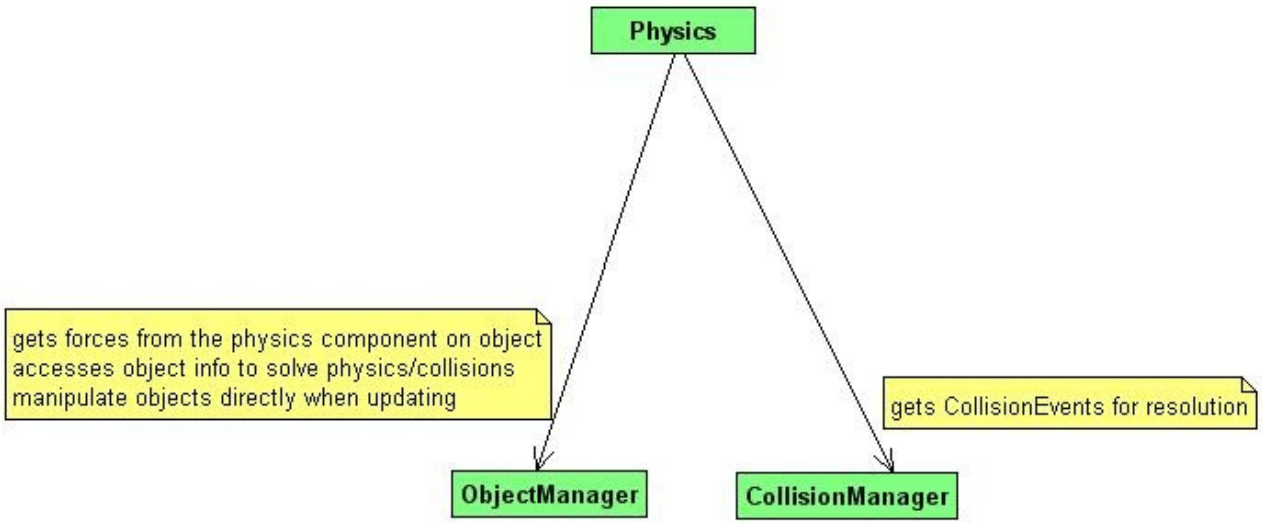
Stress tests for

maximum number of effects

complex models

Physics




**Responsibilities**

integrators
constraint solver
collision resolution

**Methods/Attributes**

```
public:
  Update();
private:
  ResolveCollisions();
  // the part that handles constraints, motors, etc
  // fills the force cache
  CalculateForces();
  IntegrateForces();
  IntegrateVelocity();
```

**Interactions**

Gets CollisionEvents from the CollisionManager
Updates the PhysicsComponent on the GameObjects

**Test Plan Outline**

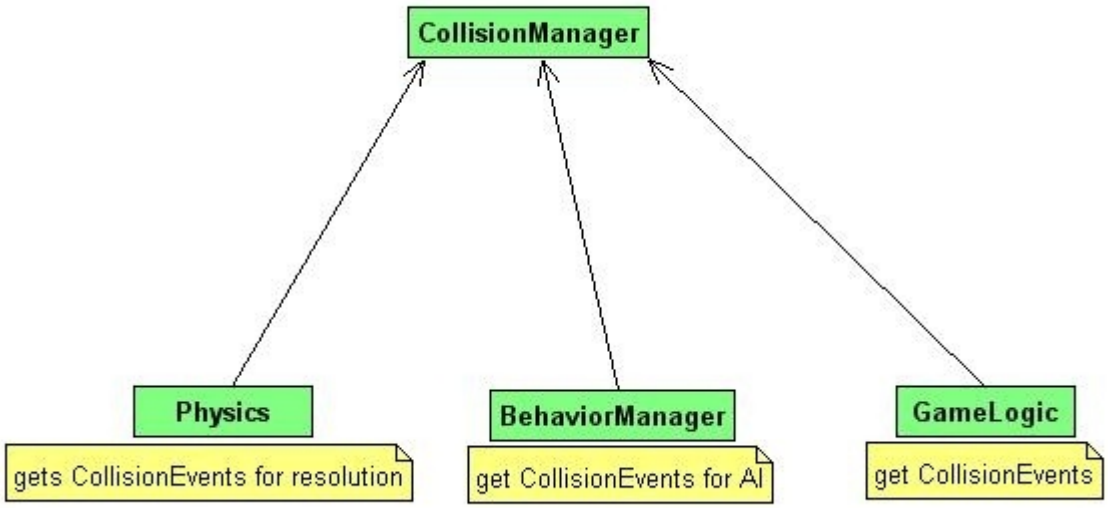
Speed and stress tests for
  collision solving
  constraint solving
  integrators
Interactive tests for collision
  irregular models
    many/few polys

inconsistently sized polys/regions
concave/convex models
Integrator accuracy tests
output to gnuplot (use existing testbed)
check against falling with drag, linear ODE, and oscillating 2nd order ODE
check equations with explicit spacial dependencies
check equations with explicit time dependencies
Constraint solving test
limbs
trees of objects (a hand with fingers for example)
check for penetration during solving, use it to guage accuracy
interactive mixed physics tests
constrained systems of irregular objects colliding

CollisionManager




**Responsibilities**

Holds a list of collisions with times attached
  Possibly per-frame, though caching older collisons might make sense
Collision events are generated, but not resolved through the CollisionManager
  Collisions are resolved by Physics/etc, and marked resolved/processed
Each collidable object has a CollisionComp with different levels of collision geometry
  The collision geometry might be updated by the CollisionManager

**Methods/Attributes**

```
public:
  void Update();
  // synchronized accessors
  std::vector<CollisionEvent> GetCollisions();
  std::vector<CollisionEvent> GetCollisionCache();

private:
  void BroadPhase();
  void NarrowPhase();
  // all previous collisions from the last frame with their resolution info
  std::vector<CollisionEvent> collisionCache;
  // collision events generated by the NarrowPhase
  std::vector<CollisionEvent> collisions;
```

**Interactions**

Physics accesses the CollisionManager when updating
GameLogic may check to see if a bullet hit something, etc when updating
CollisionManager accesses both the ObjectManager and the ResourceManager
  gets both collision and model geometry

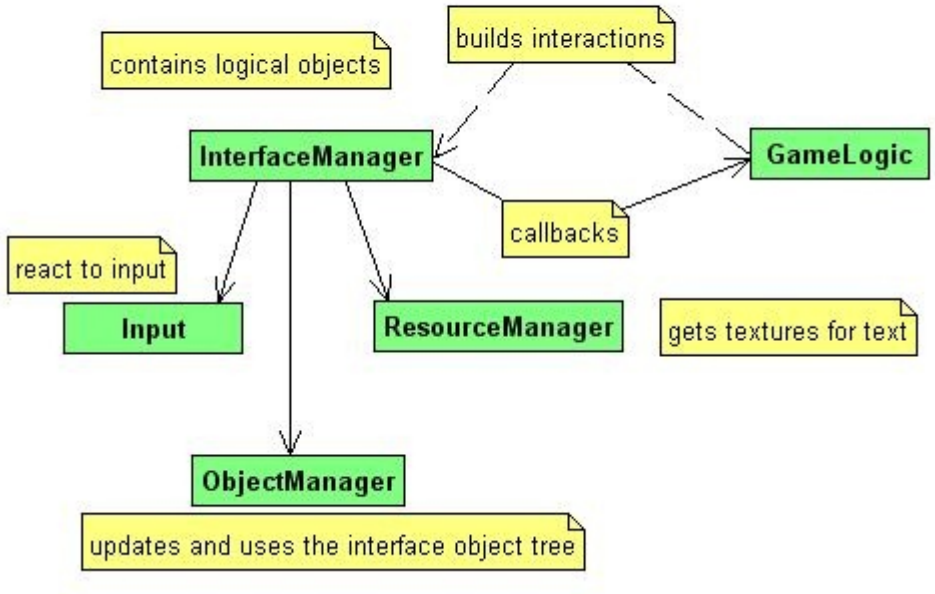
 

**Test Plan Outline**

Test collision geometry generation
Test convex object collisions
Stress test for lots of different objects colliding
Test for objects with large triangles colliding with objects that have small triangles

InterfaceManager




**Responsibilities**

Update interface
Handles the interaction
Handles and creates the gui

Update
Read events
Instantiate objects
Check objects in the graph for clicks/button presses
Modify the GUI based on input (open menus, etc)
Send events generated by GUI interactions
Render the GUI to a texture (done inside of Interface, doesn't use Graphics module)

**Methods/Attributes**

```
public:
  void Update(void);

  // Creates a interface object for the gui which are handled by the object
  // manager
  void CreateInterfaceObject(objType type);
```

**Interactions**

Takes spawn/move/destroy widget events from GameLogic

Sends state changes to GameLogic
Accesses the InputManager
Creates interface objects that the ObjectManager handles

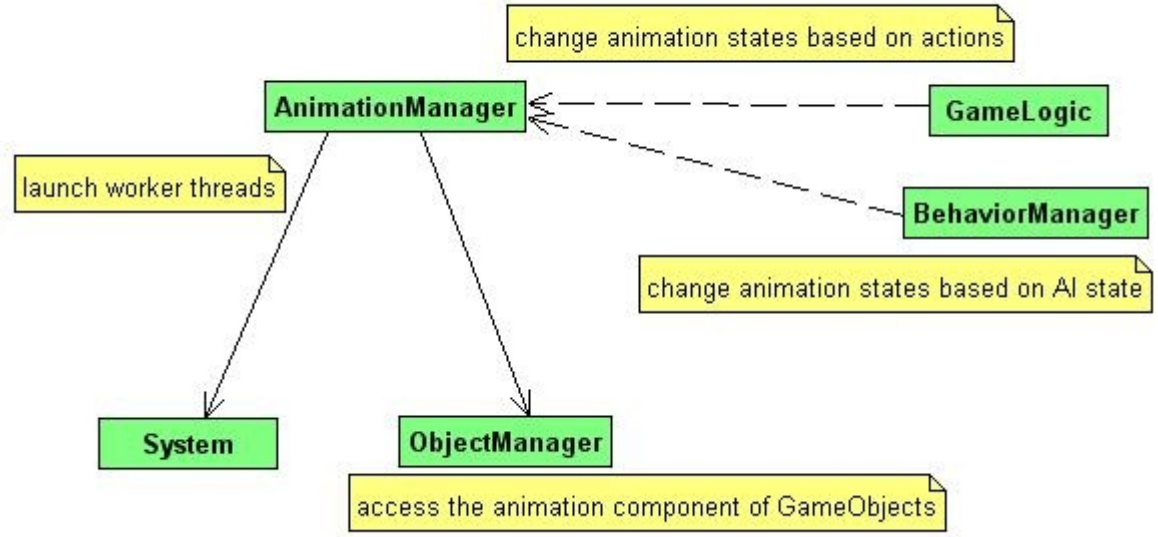
**Test Plan Outline**

Test text input
  random input stress test
Test GUI
  focus stress test
  add/remove child stress test
  child depth stress test

// Although the ObjectManager probably does this
Test interface object creation/destruction

AnimationManager




**Responsibilities**

Uses the Animation component of an object like a state machine
  Animation component is attached to top level object and holds the bones for an object
Updates the animation of a bone system (articulates joints, etc) based on keyframes
Possibly solves inverse kinematics if not using keyframes
Applies animation updates either using the physics module, or directly

**Methods/Attributes**

```
public:
  void Update();

private:
  void GetAnimationUpdate();
  void ApplyAnimation();
```

**Interactions**

GameLogic and BehaviorManager send state change messages to the AnimationManager
  the AnimationManager modifies the AnimationComponent's state based on its messages
AnimationManager launches threads from System
AnimationManager uses the AnimationComponent on GameObjects as a state machine
  the component also holds the bones, their mappings, the keyframes, and the matrix buffer

**Test Plan Outline**

Tests for different animations
Tests for keyframing
  Tests for whatever keyframing mechanism is used
Tests for properly importing keyframes from .fbx files (this is a ResourceManager test)

# Common Objects

## GameObject

note on HANDLE:
  is a string for now

**Methods/Attributes**

```
  Quaternion     rotation;
  Vector         position;
  unordered_map<std::string, Component> components;
```

**Component Types**
  collision information
  model (holds HANDLE to resource)
  physics (holds velocity/etc)

**Test Plan Outline**
  Creation/deletion stress tests with memory leak checking

## Factory (abstract)

note on HANDLE:
  is a string for now

**Responsibilities**
  loads an xml file through System
    xml files = resources? If so, load through ResourceManager
  using ticpp parses the xml into an object
    GameObjects factory creates objects and puts them into the ObjectManager
  has an interface that can be called by lua scripts
  Factory also makes sure everything has a unique name (at least for GameObjects)
  some factories may call other factories
  component factory
  patterns are lua scripts

**Methods/Attributes**

```
GameObjectFactory
  public:
    HANDLE CreateGameObject(std::string script);
  private:
```

**Interactions**
  calls other factories for different components
  calls scripts from the LuaManager

called by GameLogic
adds things to the ObjectManager in some cases

**Test Plan Outline**
stress test from multiple threads
have a way to save the ObjectManager so that results of running scripts can be compared
object creation/destruction: memory leak check

## Module (abstract template)

**Responsibilities**
is inherited by all modules
contains functions for registering itself with the EventManager
is also a singleton interface

**Methods/Attributes**

```
class Module (is a template class)
  public:
    T& Instance();
    virtual void Init();
    virtual void Update();
    virtual void Kill();
  protected:
    void RegisterWith(EventManager& eventManager, std::string moduleName);
    T instance;
```

**Interactions**
Calls a registration function on the EventManager if needed (ex. EventManager won't register with EventManager)

**Test Plan Outline**
no need for testing

# The Gameloop

The basic gameloop has a simple structure: update core modules, update background game modules, update non-rendering game modules, update rendering game module.

```
GameLoop()
  // core modules
  System.Update()           // update any timing/debug/etc stuff
  EventManager.Update()     // the update here is largely unused
  InputManager.Update()     // need to know the current keystates
  ResourceManager.Update()  // sends/receives load status msgs
  ObjectManager.Update()    // processes any messages
  CollisionManager.Update() // needed for proper physics/etc


  // background game modules
  LuaManager.Update()       // Lua runs through interrupts
  SoundManager.Update()     // sound runs in the background
  InterfaceManager.Update() // draws to texture, generates logic
  GameLogic.Update()        // reacts to logic, sends messages
  BehaviorManager.Update()  // reacts to messages
  AnimationManager.Update() // anim states altered by BehaviorManager
  Physics.Update()          // corrects animation
  Graphics.Update()         // finally, the scene is rendered
```

# The Scenegraph

## Scene “Graph”

Our game does not require a true scenegraph, but refers to the separate object lists as the scenegraph to prevent confusion with individual lists. GameObjects are any object that will be moving in the scene and may have components. LevelObjects are part of the level, and typically have some sort of visibility information (terrain is organized in a BSP tree, etc). The Light list and the Camera object are used mainly by the Graphics module with some exceptions.

## Orientation Representation

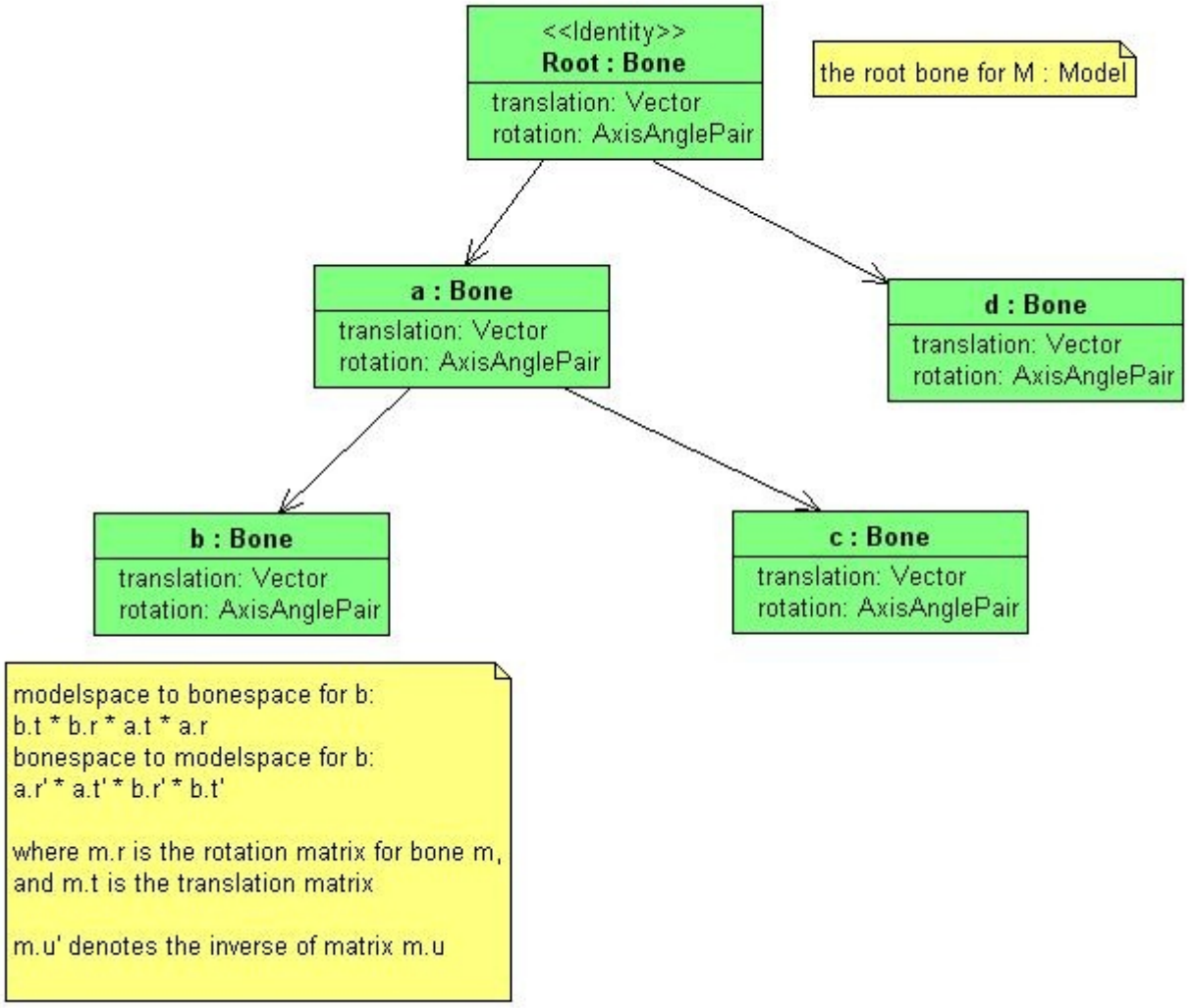
To simplify calculations while rendering, collision detection, physics, and any other stage of the gameloop that requires calculating transformations, the position of an object relative to the world is always represented using a single rigid body transformation. All scaling is done at design time to limit all orientation to translations and rotations only (a smaller, more manageable subset of generalized affine transformations [2]).

The same orientation representation is also used in bone hierarchies. The rotation is stored as a quaternion, while the translation is stored as a vector. When it comes time to calculate the bone buffer, the quaternion vector pairs are converted to matrices.

## Spatial Partitioning

Since most of our game takes place on the ground, spatial partitioning for GameObjects will use a quad tree. To optimize level rendering, a BSP tree will be used for the level itself, and will most likely be calculated at design time in some sort of tool.




Animation will use bones, and will not be done using the scenegraph hierarchy. Instead, a hierarchy of bones will be used to manipulate a mesh. The bones will most likely reside in the Animation component for the GameObject they operate on. Each bone will have the same translation vector, quaternion scheme that the scenegraph objects use. In terms of our game, typically each vertex will have influence from only one bone.

To save time, the resulting matrices are cached. Each cached matrix for a given bone represents the bone-to-modelspace transformation instead of the bone-to-parent transformation given by the quaternion-vector pair on the bone itself. These are updated if an object's limbs are moved during a gameloop (a flag is set, stating that the matrix buffer has out of date information). During render time, the skin is applied using the vertex shader. The same matrices are also used during collision detection, AI, and other things requiring world coordinates.

# Libraries and Pre-Built Tools

## Game Libraries

| | |
|---|---|
| OpenGL | Rendering.<br>Version: OpenGL 2.0 with GLSL shader model 2.0 (at minimum, optimized for 4.0) |
| GLEW | OpenGL extension management.<br>Version: glew 1.5.1 |
| GLFW | Originally used for window management/input, now only used for the high-res timer.<br>Version: glfw 2.6 |
| SDL | Window management, input, resolution changing.<br>Version: SDL 1.2.14 |
| FMOD | Sound.<br>Version: fmod 3.75 |
| Lua | Scripting.<br>Version: Lua 5.1.4 (August 2008) |
| Boost | Containers, synchronization (threads, hash tables, etc).<br>Version: Boost 1.37.0 or later (up to Boost 1.40.0 works so far) |
| TinyXML | XML serialization.<br>Version: TinyXML 2.5.3 |
| SOIL | Image/texture loading.<br>Version: July 7, 2008 release. http://www.lonesock.net/soil.html |

## Tool Libraries

| | |
|---|---|
| Autodesk FBX SDK | Used for building tools to generate model/scene files from FBX files.<br>Version: FBX SDK 2010.0 and 2010.2 |
| wxWidgets | GUI for tools.<br>Version: wxWidgets 2.8.10 |
| Visual Leak Detector | Better output for Visual Studio's built-in leak checking. Used in Debug.<br>Version: VLD 1.0 (August 5, 2005) |

## Tools

| | |
|---|---|
| SWIG | Generating Lua glue functions. |

| | |
|---|---|
| | Version: swig 1.3.36 |
| Autodesk 3DS MAX | Animation and modelling.<br>Version: Autodesk 3DS MAX 2010 (12.0) |
| GIMP, Photoshop | Textures, logos, etc.<br>Version: GIMP 2.0 and above. Photoshop Elements. |
| MS Visual Studio 2005 | IDE/compiler.<br>Version: Microsoft Visual Studio 2005 (IDE Version 8.0.50727.762)<br>Command line compiler version 14.00.50727.762 |
| Cygwin Bash | Automation for various tasks. Mainly used for building the installer.<br>Version: GNU bash, version 3.2.x or later for Cygwin |
| NSIS | The installer compiler.<br>Version: Nullsoft Scriptable Install System 2.24 |
| AMD CodeAnalyst | Used to identify performance bottlenecks.<br>Version: AMD CodeAnalyst version 2.9 |

## Old Libraries and Tools (not used anymore)

| | |
|---|---|
| TiCpp | XML serialization. Dropped because it is broken under MSVC 2005.<br>Version: ticpp 2.5.3 |
| Cygwin Make | Scripted builds. Dropped after Makefiles were not properly maintained.<br>Version: GNU Make 3.81 or later for Cygwin |

# Final Game Loop and Module List

## Module List

**Core Modules**

System
EventManager
InputManager
ResourceManager
CollisionManager
ObjectManager

**Game Modules**

AnimationManager
Physics
Graphics
InterfaceManager*
SoundManager*
GameLogic*
BehaviorManager*
SpawnManager*
LuaManager*

* denotes Game Modules with extra functions exposed that are used exclusively by game logic and behavior code, or by calls from objects explicitly related to its functionality. No * denotes a "pure" Game Module interface, exposing only Init(), Update(), and Kill() functions.

## Main

The following functions run in parallel, then wait until all three threads reach the barrier.

```
void RunPhysics()
{
  while(System::Instance().isRunning)
  {
    if(!System::Instance().Pause())
    { Physics::Instance().Update(); }

    updateSyncBarrier.wait();
  }
}

void RunCollisionManager()
{
  while(System::Instance().isRunning)
  {
    if(!System::Instance().Pause())
    { CollisionManager::Instance().Update(); }
```

```
    updateSyncBarrier.wait();
  }
}

void GameLoop()
{
  while(System::Instance().isRunning)
  {
    System::Instance().BeginFrame();

    // core modules
    System::Instance().Update();
    EventManager::Instance().Update();
    InputManager::Instance().Update();
    ResourceManager::Instance().Update();
    ObjectManager::Instance().Update();

    // game modules
    LuaManager::Instance().Update();
    InterfaceManager::Instance().Update();
    SoundManager::Instance().Update();
    GameLogic::Instance().Update();

    if(System::Instance().Pause() == false )
    {
      BehaviorManager::Instance().Update();
      AnimationManager::Instance().Update();
      SpawnManager::Instance().Update();
    }

    // graphics
    Graphics::Instance().Update();

    updateSyncBarrier.wait();

    System::Instance().EndFrame();
  }
}
```

# Getting WxWidgets To Work

Using 2.8.10 version of wxWidgets

Installed on C:

open up wxWidgets-2.8.10/build/msw/wx

In "C/C++" -> "Code Generation" use "Multi-threaded Debug (/MTd)"

Under "Build" -> "Batch Build"

Build only "Release" and "Debug" for all projects under "Configuration"

Keep rebuilding ontill all projects have been built successfully (projects can fail
to build because they sometimes use the same libraries)

To use wxWidgets in a project:

"C/C++" -> "General" -> "Additional Include Directories"

add C:\wxWidgets-2.8.10\include // Needed for general includes

add C:\wxWidgets-2.8.10\include\msvc // This is needed because setup.h is in here for some reason

"Linker" -> "General" -> "Additional Library Directories"

add C:\wxWidgets-2.8.10\lib\vc_lib

"Linker" -> "Input" -> "Additonal Dependencies"

add comctl32.lib

add rpcrt4.lib

"Linker" -> "Input" -> "Ignore Specific Library"

add LIBC // If Compiling in Debug

add MSVCRT // If Compiling in Debug

add LIBCD // If Compiling in Debug

add LIBCMTD // If Compiling in Debug

add LIBCMT // If Compiling in Debug

You have to recompile parts of wxWidgets to get wxGLCanvas working.
In C:\wxWidgets-2.8.10\build\msw\

Open wx and rebuild wxregex after changing all #define wxUSE_GLCANVAS 0 to 1 in both setup.h

# Coding Standards

## Filenames

Lowercase with underscores, usually representing the library or object it contains.
Example: GameObject is in game_object.h and game_object.cpp

## Type/Object Names

Enumerated types, typedef'd types, and classes are all camel-cased with the first letter capitalized.
Example: the state machine class is StateMachine
Example: the matrix class is Matrix

## Functions and Function Members

Functions and function members follow the same convention as object names.
Example: `GameObject::ApplyVelocity()`

## Data and Data Members

Data objects (class instances, variables, class members, etc) are camel-cased with the first letter lowercase.
Example: `GameObject::canRender`

## Preventing Duplicate Includes

Say I have a file named mesh.h in the graphics/ subdirectory. Around the header file, the following should be added to prevent duplicate includes:

```
#ifndef GRAPHICS_MESH_H
#define GRAPHICS_MESH_H

// header stuff here

#endif /* GRAPHICS_MESH_H */
```

Using `#define`'s instead of `#pragma`'s gives extra information that can be used for double-checking which files have been included. Note that the subdirectory is included in the `#define`. This resolves the problem of duplicate filenames. If somehow there were a file named graphics_mesh.h in the top-level source directory, two underscores could be used between directory and filenames (but having odd filenames like that is bad anyway). As with .cpp files, a newline should be inserted at the end of the file. It's an old convention, but still a good one to follow, especially if you want to look at pre-processed output.

## File Headers

Example:

```
/******************************************************************************/
/*!
@brief      SLEEP Engine
@file       lua_link.h
@author     Tommy A. Brosman IV
@par        dp email: tbrosman\@digipen.edu
@par        alt email: drakoniis\@gmail.com
@date       September 2, 2007

@brief
  Adapted from the LuaLink class (also by me) used in the DigiPen game
  NullSpace by Hopeless Productions.

*/
/******************************************************************************/
```

File headers are a must, so that authorship can be shown. All authors for the file should be listed, with a new set of author/dp email/alt email tags for each. Note that the alt email is an optional field. Also, the line length does not extend past 80 characters. In Crimson Editor, Tools->Preferences->Visual, "2nd column marker at" checked with its value set to 80. This is an old standard dealing with the width of the console. Also, it makes code look pretty.

These headers should appear in both the .h and .cpp files, even if the information is the same and only the file extension in filename is changed.

## Function Headers

Example:

```
/******************************************************************************/
/*!

  Transform a point by the pre-computed matrix values (these must be computed
  first for the transformation to work).

  @param pntB
    the point to transform

  @return
    a transformed copy of the original point

*/
/******************************************************************************/
```

Use function headers as needed. Show authorship for individual functions if necessary.

## Comments

Use, but don't overuse. The 80-column rule applies, so multi-line comments can be done the C++ way:

```
// this is the first line
// this is the second line
```

Or the C way:

```
/*
* this is the first line
* this is the second line
*/
```

## Syntax

```
// comment above the code, not after

void UseAnsiBrackets()
{
  // not K&R
}

while("Use ANSI brackets everywhere")
{
  // even in loops
}

  // indent with two spaces, no tabs...
  // or cheat and use Crimson Editor to change it
  // from tabs to spaces later (what I do)

{
  SomeMenu(string _name, int _time); // in this case, it is to keep from reusing
                                     // a member-name as a parameter (even though
                                     // that is perfectly legal)
}
```

Cut down on variable usage, reuse functions, keep your code readable, etc. Write code that won't require any extra explanation. Remember to put a newline at the end of both .h and .cpp files (just in case you want to view pre-processed output, and also to keep compilers from throwing warnings).

Last but not least, `((++*no ^ --*clever), !!code)`

## Makefiles, Scripts

Lua scripts, Makefiles, etc should also show authorship equivalent to that shown in the C++ files.

# Bibliography

[1] Mönkkönen, Ville. “Multithreaded Game Engine Architectures”.
http://www.gamasutra.com/features/20060906/monkkonen_01.shtml.

[2] Van Den Bergen, Gino. “Collision Detection In Interactive 3D Environments”.
Morgan Kaufmann Publishers, 2004. pg 20.

[3] Erleben, Sporring, Henriksen, Dohlmann. “Physics Based Animation”.
Charles River Media, Inc, 2005. pg 22.