Technical Design Document

Date: February 18, 2010

Authors: Sela Davis
Chip Hilseberg
Jonathan Lobaugh
Eric Moreau
Nicholas Wilsey

Revision History

Date	Author	Description
1.20.2010	Jonathan Lobaugh	Initial document creation
2.1.2010	Eric Moreau	Added content to components section
2.10.2010	Jonathan Lobaugh	Task subsystem added
2.11.2010	Jonathan Lobaugh	Scene subsystem added
2.12.2010	Jonathan Lobaugh	Renderer component system added
2.13.2010	Jonathan Lobaugh	Component section extended
2.16.2010	Jonathan Lobaugh	Lua and Game Objects extended
2.17.2010	Jonathan Lobaugh	Collider component added and extended render section

Table of Contents

1. Summary

2. Document Scope

This document is intended to be used in the development and implementation of the Trigger Happy game. The document will detail the architecture of the multi-threaded engine, the core components, and gameplay elements that are to be used in the development cycle. It is also useful background reading for anyone involved in management or oversight of the Trigger Happy game.

3. Development Technology

Hardware	NVidia 9600 GT (x2)
	Intel Core2 Quad 2.66GHz
Software	Maya
	Visual Studio 2008
	SVN (gdd.unfuddle.com)
Operating System(s)	Windows Vista
	Windows 7
External Libraries	Lua

4. Production Technology Requirements

Hardware	NVidia 9 Series graphics card or greater Radeon HD2900 graphics card or greater
	Intel Pentium D 3GHz or Higher
Operating System(s)	Windows Vista
	Windows 7
Software	DirectX 10 or greater

5. System Architecture

The game engine design used in the development of Trigger Happy utilizes a new engine architecture based on multi-core component based processing. The system attempts to utilize the CPU's multi-core advancements to the their utmost performance. One of the main changes from most engines is the move from a hierarchical approach to game objects to a component based approach. The component based approach is based on the separation of functionality into individual components that are mostly independent of one another. The traditional game object is replaced with a aggregation of independent components, together which define the game objects behaviors and render characteristics.

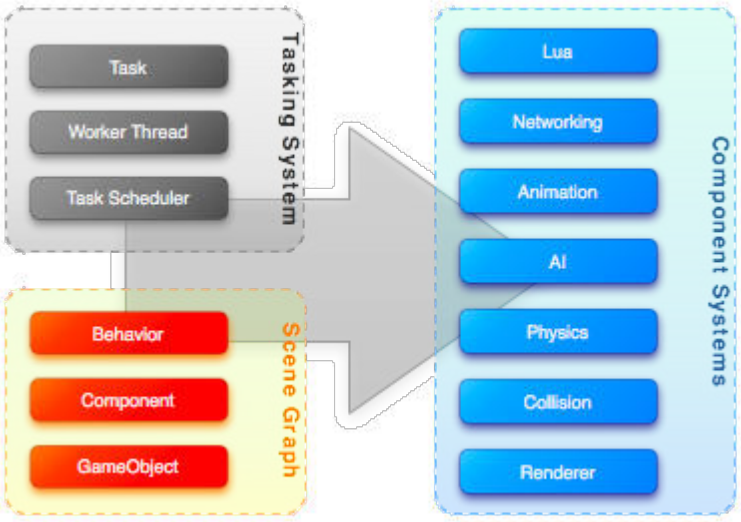


Figure 1: Singularity System Architecture Overview

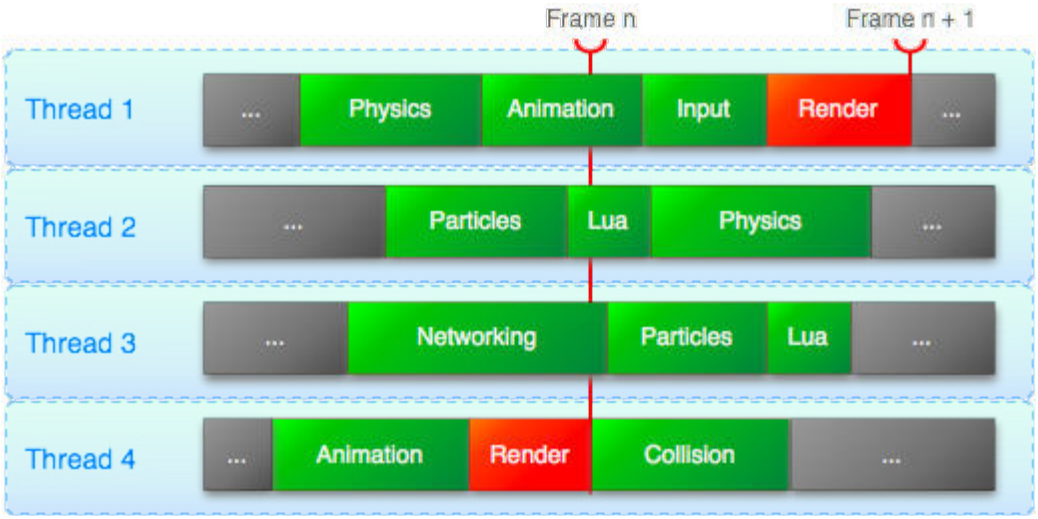


Figure 2 : Threading Task Scheduling Diagrams

5.1. Scene System

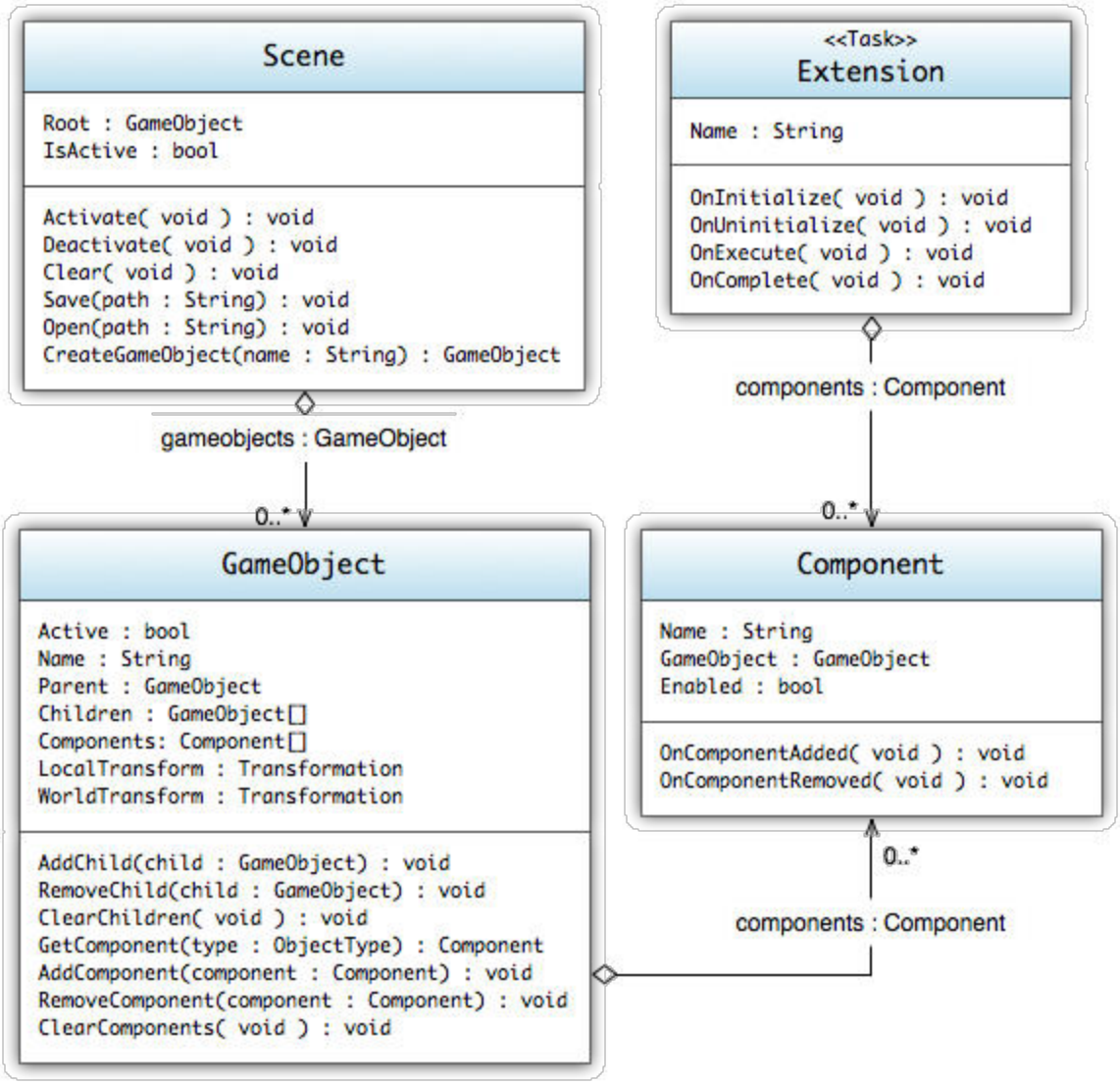


Figure ?? : Scene System Diagram

The scene system is the core of the engine and manages all of the objects that are to be placed within Trigger Happy. Basic scene culling and object partitioning is handled within this system via the SceneGraph system. The system allows customization of the storage and partitioning of the *GameObjects* by the inclusion of user definable *SceneGraph* classes.

5.1.1. GameObject

The *GameObject* are containers for all of the engine's components. All of the objects in Trigger Happy are compositions of other *GameObjects* and *Components*. *GameObjects* do not add any characteristics to the game by themselves; instead, they manage the attached Components which implement the actual functionality.

- **Data**
  - Active : *Boolean*
    - Details whether the *GameObject* is active within the *SceneGraph*
  - Name : *String*
    - The name of the object
  - Parent : *GameObject*
    - The parent of the game object
  - Children : *GameObject[]*
    - The children of the game object
  - Components : *Component[]*
    - The list of all the attached components
  - LocalTransform : *Transformation*
    - The local transformation of the *GameObject*, this transform describes the *GameObjects* position, rotation and scale relative to the parent *GameObject*
  - WorldTransform : *Transformation*
    - The world transformation of the *GameObject*, this transform describes the *GameObjects* absolute position, rotation and scale in the world.

- **Operations**
  - AddChild(child : *GameObject*) : *void*
    - Adds a child *GameObject* to current *GameObject*; thus setting the child's parent to the current *GameObject*
  - RemoveChild(child : *GameObject*) : *void*
    - Removes the child *GameObject* from the current *GameObject*; if the child is not found then nothing is removed
  - ClearChildren(*void*) : *void*
    - Removes all of the child *GameObjects* from the current *GameObject*
  - GetComponent(type : *ObjectType*) : *Component*
    - Retrieves the first instance of a component that matches the *ObjectType*
  - AddComponent(component : *Component*) : *void*
    - Adds a *Component* to the current *GameObject*
  - RemoveComponent(component : *Component*) : *void*
    - Removes the component from the current *GameObject*; if the component is not found then nothing is removed
  - ClearComponents(*void*) : *void*
    - removes all of the components from the current *GameObject*

5.1.2. Component

The core class for all components that attach to *GameObjects*. *Components* represent all of the functional additions that can be attached to *GameObjects*; examples include rendering, collisions, physics, etc. See the Component System section (???) for the current list of engine components that are available to the designers and the game development.

- **Data**
  - Name : *String*

        - Contains the user-defined name of the *Component*
    - GameObject : *GameObject*
        - The GameObject that the *Component* is attached to
    - Enabled : *Boolean*
        - Describes whether or not the *Component* will be executed

- **Operations**
    - OnComponentAdded (*void*) : *void*
        - Called when the *Component* is added to a *GameObject*
    - OnComponentRemoved (*void*) : *void*
        - Called when the *Component* is removed from a *GameObject*

5.1.3. Extension

Extensions are the managers of components and execute the overall behavior that the custom *Component* is meant to embody. *Extensions* are also *Tasks* and as such are able to take advantage of the multicore tasking system. By utilizing the multicore tasking system, *Extensions* are able to partition their *Components* and execute each subset in parallel; greatly increasing the execution throughput of the system.

- **Data**
    - Name : *String*
        - Contains the user-defined name assigned to the *Extension*

- **Operations**
    - OnInitialize(*void*) : *void*
        - Called when the first *Extension* of this type is initially loaded into the scheduler
    - OnUninitialize (*void*) : *void*
        - Called when the last *Extension* of this type is unloaded from the scheduler
    - OnExecute(*void*) : *void*
        - Inherited from Task
    - OnComplete (*void*) : *void*
        - Inherited from Task

5.1.4. Scene

The Scene is the organizational construct that maintains and orders the *GameObjects*. Culling and other spacial operations will also be performed using the power of the *SceneGraph*. Finally the management of object locking is maintained by the *SceneGraph* in order to prevent systems from causing data collisions through the manipulation of the *GameObjects* within the *SceneGraph*.

- **Data**
    - Root : *GameObject*
        - Root *GameObject* that all other *GameObjects* are relative to. This is normally used as the base coordinate system for the game.

- **Operations**
  - BeginUpdate(*void*) : *void*
    - Sets up a differed rebuild of the *SceneGraph* tree structure. This is used to prevent a large adjustment of *GameObjects* from propagating on every change; speeding up the system significantly.
  - EndUpdate(force : *Boolean*) : *void*
    - Finalizes the differed update of the *SceneGraph* tree structure. If *force* is enabled, then the tree will rebuild whether or not the tree needs it or not.
  - Update(force : *Boolean*) : *void*
    - Instantly rebuilds the *SceneGraph* tree structure. If *force* is enabled, then the tree will rebuild whether or not the tree needs it or not.

5.2. Tasking System

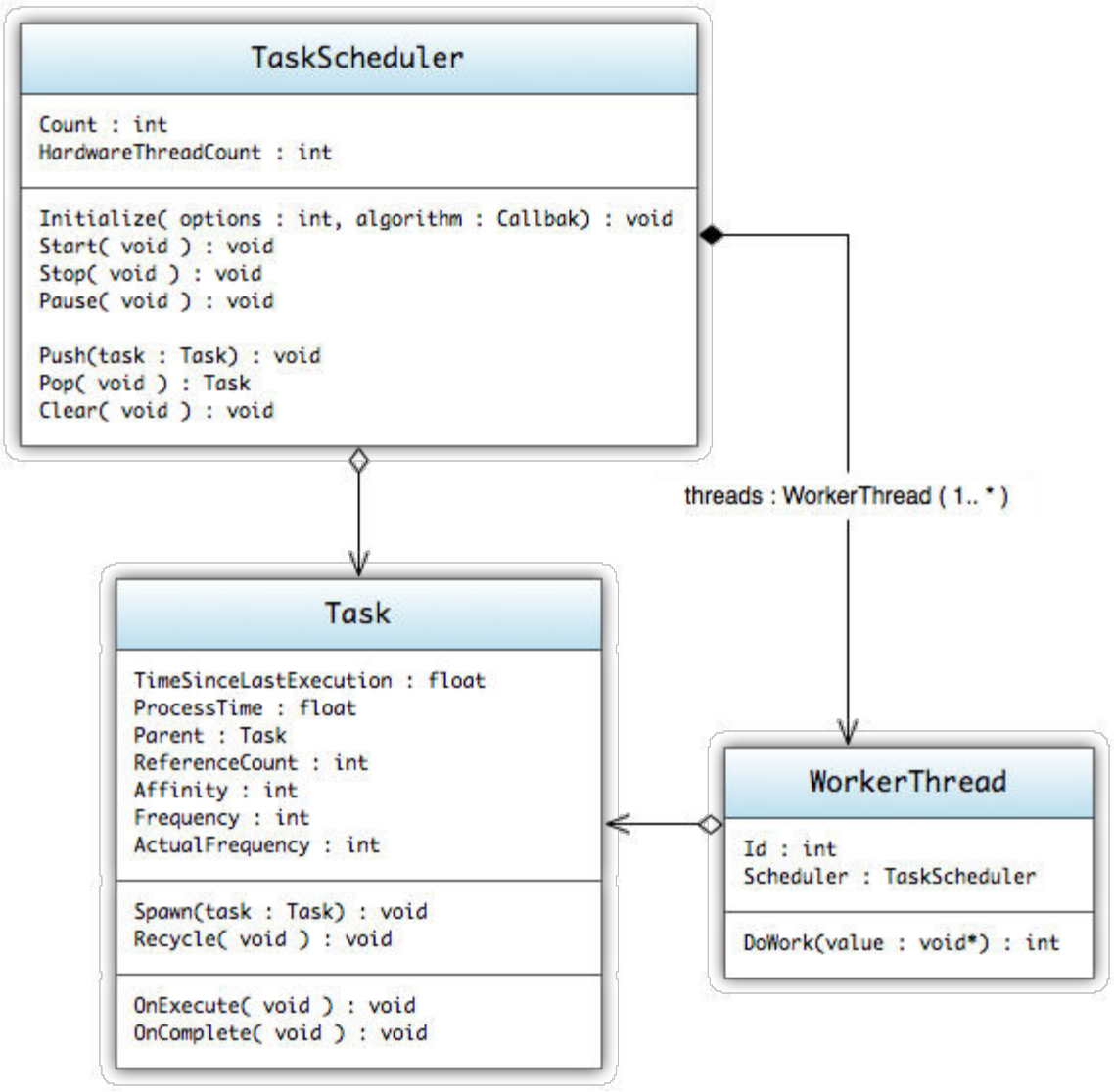


Figure 2: Task System UML Diagram

The tasking system is a hybrid of the two techniques used in the Intel Smoke demo; both of which utilizes the Intel Threaded Building Blocks library. To minimize the systems size and execution overhead, the tasking system uses a custom developed systems architecture that closely mirrors the functionality of the Intel Threaded Building Block. Instead of managing tasks on each worker thread the adjusted system architecture manages a centralized queue of tasks. Also, due to the system component based architecture, the standard game loop has been replaced with a dependency driven task execution pipeline.

The main purpose of the tasking system is to schedule tasks, taking into account dependencies, execution time, and task priority.

The task scheduler is intended for parallelizing computationally intensive work.

5.2.1. TaskScheduler

The task scheduler is the core of the tasking system, its primary function is to manages the scheduling of tasks according a weighting algorithm. The scheduler maintains an internal queue of the currently assign tasks and is structured as a dequeue When a task is requested the scheduler returns the next task according to the weighting algorithm.

Weighting Algorithm

The

1. The task whose lastly completed child was completed by this thread.
2. The task whose frequency deadline has been reached or will fail if another task is executed
3. A task with the highest affinity
4. A task with the shortest process time

- **Data**
  - Count : *int*
    - Number of *Tasks* currently on the scheduling queue
  - HardwareThreadCount : *int*
    - Number of cores/processors that are available on the current system

- **Operations**
  - Initialize(options : *int,* algorithm *: WeightingAlgorithmCallback*) : *void*
    - Initializes the task scheduler using the options. Allows the system to run in single or multi-threaded mode as well as scheduling behaviors.
  - Start(void) : void
    - Starts the scheduling system and spools up the appropriate *WorkerThread*
  - Stop(void) : void
    - Stops the scheduling system and shuts down the *WorkerThreads*
  - Pause(void) : void
    - Pauses the scheduling system and puts all of the *WorkerThread* into sleep mode
  - Push(task : Task) : void
    - Pushes the task onto the scheduling queue.
  - Pop(void) : Task
    - Pops the next task from the scheduling queue. Priority is determined by task frequency, last execution time, affinity and process time. Ideally tasks that have a frequency set will be moved to the start of the queue the closer they get to their deadline.
  - Clear(void) : void
    - Removes all of the currently queued tasks from the scheduling queue.

5.2.2. WorkerThread

The *WorkerThread* is the work horse of the tasking system. Depending on the options specified in the

initialization of the *TaskScheduler* the number of *WorkerThreads* can range from one up to the number of cores the system has.

- **Data**
  - Id : *int*
    - Identifier of the *WorkerThread* as assigned by the *TaskScheduler*
  - Scheduler : *TaskScheduler*
    - The scheduler which is currently handling the task management.

- **Operations**
  - DoWork(value : void*) : int
    - The main execution loop for the *WorkerThread*. This is where all of the *Task* Execution happens as well as the idling when no more tasks are present.

5.2.3. Task

The task is the primary development concept of the tasking system. Tasks are the unitized functional calls that allows the threading model to execute dependent on the processor core, thus allowing multicore programming without having to maintain the overhead of thread management.
The task offers a set of functions that allow the developer to manage child and parent relationships. Since tasks are **???**

- **Data**
  - TimeSinceLastExecution : *float*
    - Measurement, in seconds, of the elapsed time since the last execution of the task; if this is the initial run then the return value is 0.
  - ProcessTime : *float*
    - Measurement, in seconds, of the amount of time the task requires to complete. This value is averaged over the execution of the task.
  - Parent : *Task*
    - If the task was spawned from another task, this will be the value of the task's originator. Tasks that have not been spawned from a task will not have a parent assigned.
  - Scheduler : *TaskScheduler*
    - The scheduler which is currently handling the task management.
  - ReferenceCount : *int*
    - The number of tasks that have been spawned from this task. Before the task can complete every sub-task must complete, thus bringing the reference count to 0.
  - Affinity : *int*
    - The task's affinity within the current task scheduler. The lower the number the higher the priority in the scheduler and the task will run sooner.
  - Frequency : *float*
    - This defines, for the system, how often the task should be run. If no frequency is set, the system will treat the task as a filler when dealing with the scheduling algorithm.
  - ActualFrequency : *float*
    - Measurement of the tasks actual execution frequency. Due to the dynamic scheduling

and process times associated with tasks, the actual execution frequency may not be the same as the defined frequency.

- **Operations**
  - Spawn(task : *Task*) : *void*
    - Spawns a child task associated with the current task. Completion of the parent task is dependent on the completion of all of the spawned child tasks. The reference count of the task will increase by one for each spawned task.
  - Recycle(*void*) : *void*
    - Commands the task to push itself back onto the queue once it has finished execution and has been completed (all of its children have completed).
  - OnExecute(*void*) : *void*
    - This function is called when the task has been assigned to a *Worker Thread* and is executed. When deriving from *Task* this the primary function that is adjusted to accommodate the functional needs.
  - OnComplete(*void*) : *void*
    - This function is called when the task and all its children have completed and the task is finally pulled from the task queue. If the task has been recycled it will be reset and pushed back onto the scheduling queue.

6. Components

A Component architecture is quite different from the usual hierarchical structure of most game engines. Instead of creating a player class that derives from classes that implement animation, rendering or physics functionality,

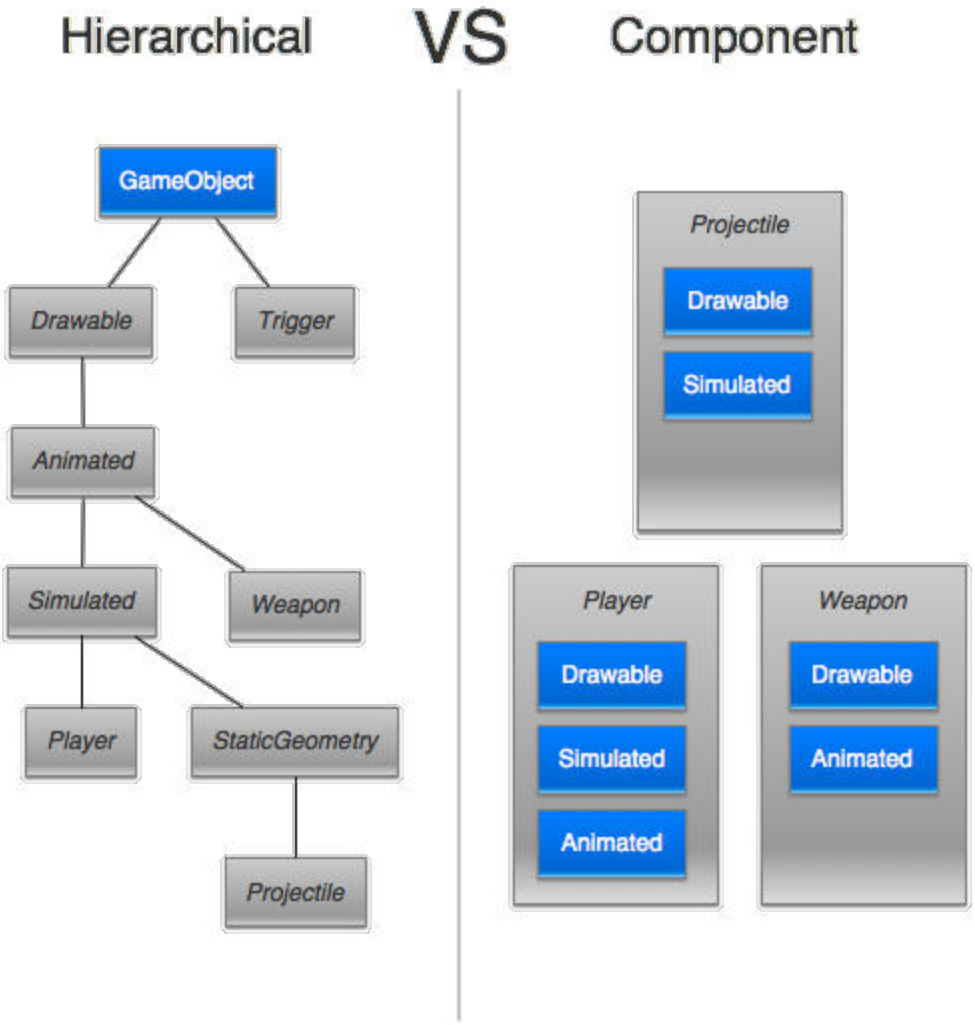


Figure ?? : Hierarchical Architecture vs. Component-based Architecture

The concept of the component architecture is to move away from a hierarchical inclusive functional model to an aggregation of functional components. What this means for object design is a generic model of creating functional units, each of which asks "independently" of each other, but together generate complex game behaviors.

In Singularity, there exists only one Object from which all game objects are created from. However, unlike the hierarchical method, we do not derive from the class. Instead we build up the class by adding components and behaviors. Components add functional utility such as animation or model rendering; behaviors are scriptable and allow developers to extend the functionality of the game

object.

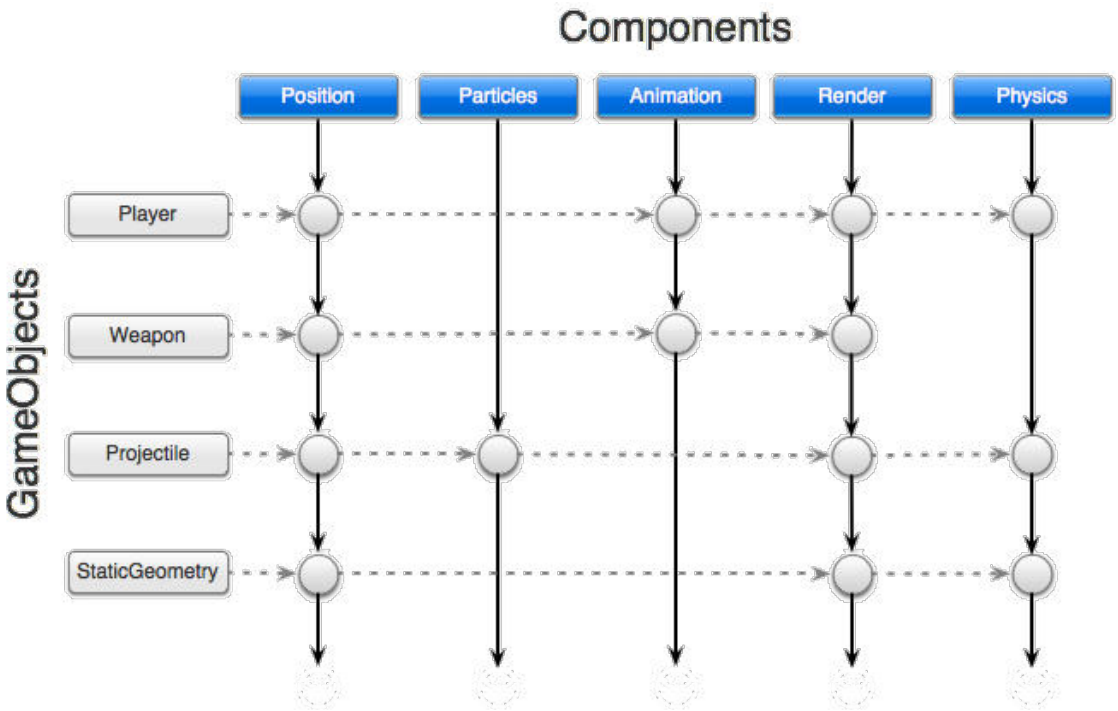


Figure ?? : Object Composition using Components

asdfasdf

6.1. Rendering Subsystem

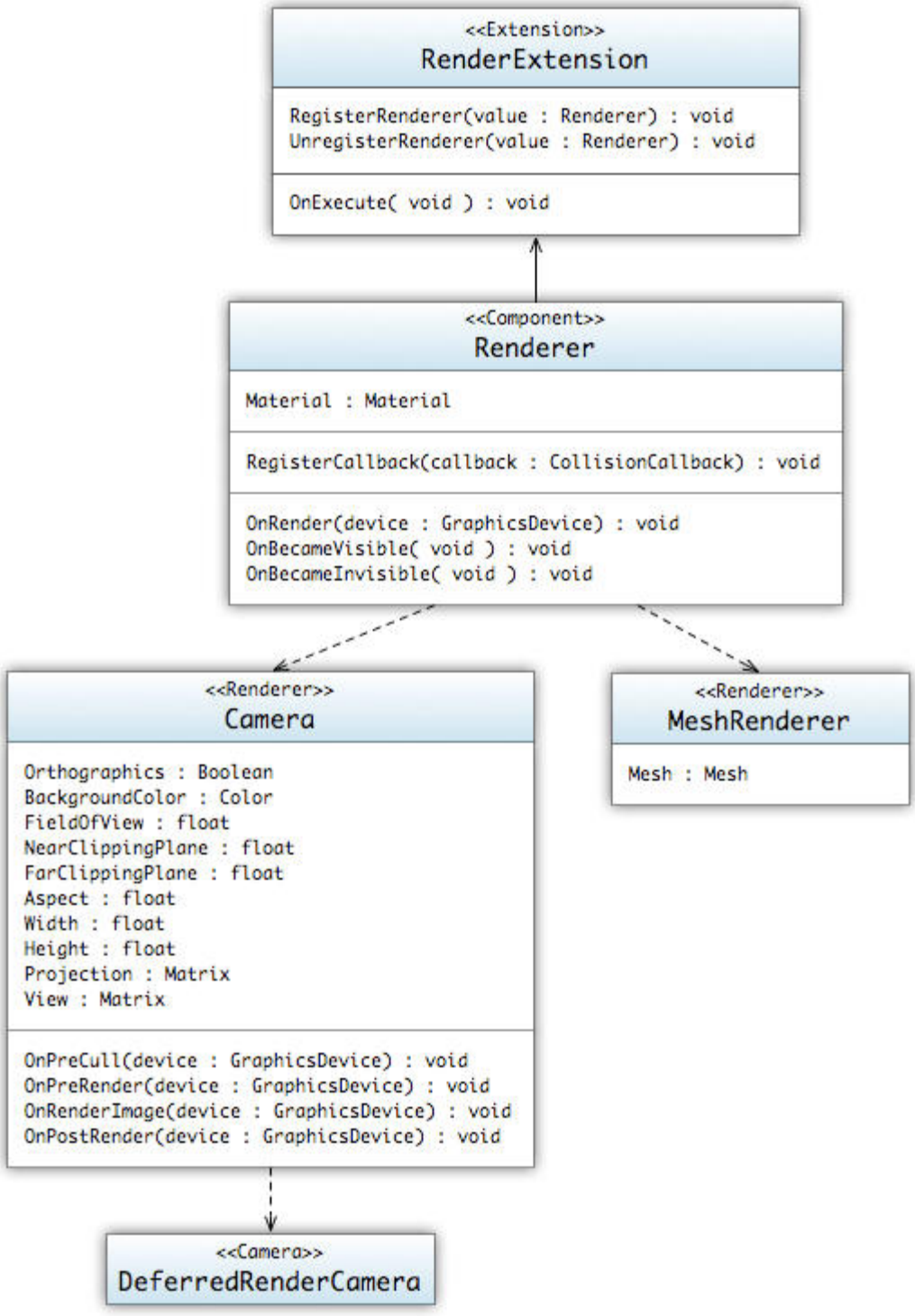


Figure ?? : Render UML Diagram

The graphics renderer is the component interface through which calls to the graphics hardware are made. It manages all aspects of the rendering and is the sole means through which one may request

objects be drawn to the screen. Since the Singularity engine is component based, the rendering pipeline is implemented as a collection of different components. Handling of all drawing calls is dealt with through the use of both the *Camera* component and the *MeshRenderer*. Though both are *Renderers*, each component handles different aspects of the rendering pipeline. The *Camera* primarily handles the setup and processing of the viewport and resulted images from rendering. The *MeshRenderer* handles the object based rendering.

6.1.1. Mesh

Represents a collection of vertices and indices that represent a 3D/2D object.

- **Data**
  - BoundingVolume : *BoundingVolume*
    - A bounding object that contains all of the mesh's polygons.
  - Vertices : *VertexBuffer*
    - The mesh's *VertexBuffer*
  - Indices : *IndexBuffer*
    - The mesh's *IndexBuffers*

- **Operations**
  - Clear(*void*) : *void*
    - Clears out the vertex and index buffers of the mesh resulting in an empty mesh
  - RecalculateBounds(*void*) : *void*
    - Recalculates the bounding volume of the mesh
  - SetVertices(declaration : *VertexDeclaration*, data : *void[]*, length : *int*) : *void*
    - Creates or modifies the *VertexBuffer* with the provided data
  - SetIndices(data : *int[]*, length : *int*) : *void*
    - Creates or modifies the *IndexBuffer* with the provided data

6.1.2. Material

Materials are the core of the rending pipeline. Materials represent effects and as such control how objects are drawn and represented on the screen.

- **Data**
  - PassCount : *int*
    - The number of passes associated with this material.

- **Operations**
  - SetVariable(key : *String*, value : *Object*) : *void*
    - Sets the materials parameter named *key* with the value provided.

6.1.3. GraphicsDevice

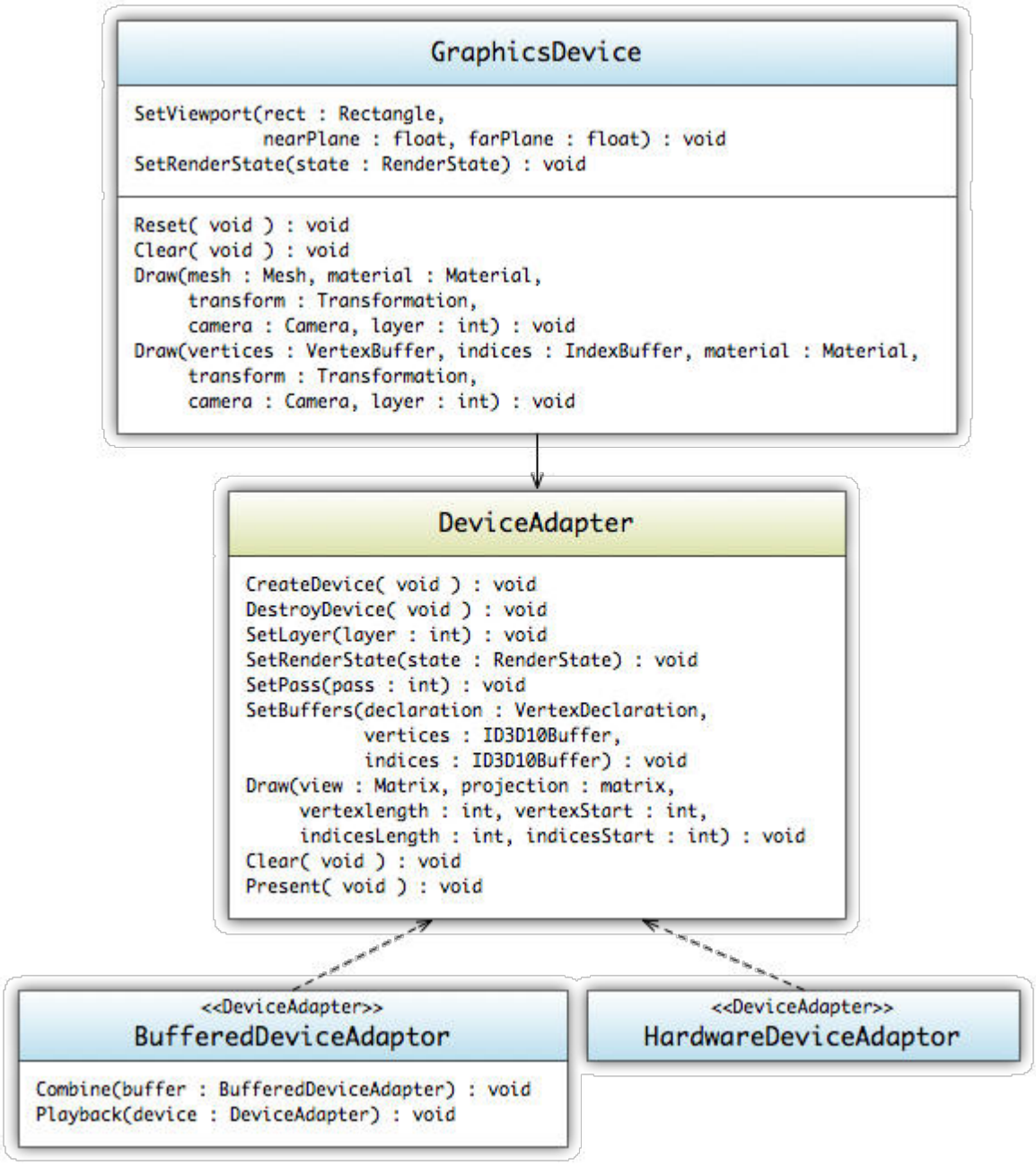


Figure ?? : Graphics UML Diagram

The *GraphicsDevice* used in Singularity has a different structure than most graphic devices in other engines. Most engines spend approximately 50-80% of their time in the render state. Since the majority of the execution time will be used on rendering, it is in our best interest to spread this

execution across the cores. However, due to the nature of the graphics hardware we are unable make render calls across threads without locking each call. With Singularity's multi-core system this slowdown would cause great hiccups in the execution process. To alleviate this issue the graphics device has two primary modes. One mode is a direct line to the underlying hardware device, the other is a recording system that both optimizes the render calls and is thread safe. The recording system allows the render of all of the visible objects using the power of the multi-threaded system without having to block between render calls. Once the recording has been completed, the *RenderExtension* spools up the *HardwareRendererExtension* which will playback the recorded rendering calls.

Figure ?? : Example of Threaded Rendering

6.1.4 DeviceAdapter

The *DeviceAdapter* is a base class for the graphics rendering pipeline. The class's main purpose to provide an interface layer between the objects and the device. In Singularity's case, there are two *DeviceAdapter* provided with the engine. One is the *HardwareDeviceAdapter* which is a DirectX 10 wrapper for the device calls. The other *DeviceAdapter* is the *BufferedDeviceAdapter*, this adapter is a call recorder, its main purpose is to allow the recording of draw calls for a playback at a later time.

- **Operations**
  - CreateDevice(*void*) : *void*
    - Creates and initialize the *DeviceAdapter*.

- DestroyDevice(*void*) : *void*
  - Destroys and releases and resources used by the *DeviceAdapter*.
- SetLayer(layer : *int*) : *void*
  - Sets the layer that the *Draw* call will occur on; Layers are a form of draw ordering.
- SetRenderState(state : RenderState) : void
  - Sets the current *RenderState* to the specified settings.
- SetBuffers(declaration : VertexDeclaration, vertices : ID3D10Buffer, indices : ID3D10Buffer) : void
  - Sets the draw buffers that are to be used when drawing; the indices are not required to be able to draw. If this is the case, then all of the vertices buffer must contain all of the vertices in draw order to be able to draw.
- Draw(view : Matrix, projection : Matrix, vertexLength : int, vertexStart : int, indicesLenth : int, indicesStart : int) : void
  - Draws the vertices/indices to the resources(screen/recorder) used.
- Clear(void) : void
  - Clears the resource(screen/recorder) buffer of the drawn objects.
- Present(void) : void
  - Finalizes the resource(screen/recorder) buffer and draws out the objects.

6.1.5 HardwareDeviceAdapter

The *HardwareDeviceAdapter* is just a wrapper around the DirectX 10 device calls. It provides an easier to use interface for the developers by abstracting away all of the device setup and management of render states.

6.1.6 BufferedDeviceAdapter

The *BufferedDeviceAdapter* is a recording class that takes all of the draw calls and stores them into a tree structure. The purpose for the adapter is the need for threaded rendering and means of which to organize and optimize the draw calls made by GameObjects.

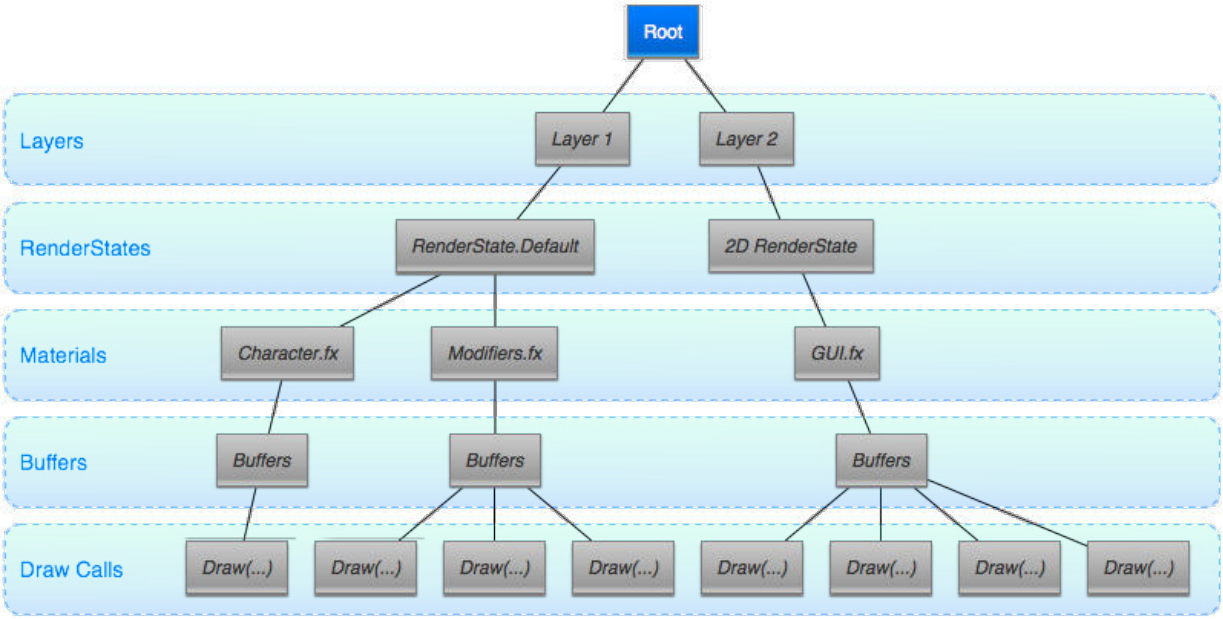


Figure ?? : Example of the recording tree structure used in the BufferedDeviceAdapter

Each instance of the device manages their own copy of the tree structure. When the final render task is completed, the trees are combined and played back using the *HardwareDeviceAdapter*. The technique allows the other threads to start working on other tasks and builds the *HardwareDeviceAdapter* a tree with an optimized render pipeline, speeding device rendering considerably.

- **Operations**
  - Combine(buffer : *BufferedDeviceAdapter*) : *void*
    - Combines the two *BufferedDeviceAdapter* Recordings into the parent. Due to the tree and depth nature of the recording tree, combination is a quick union of the recording trees.
  - Playback(device : *DeviceAdapter*) : *void*
    - Plays back the recorded render calls to the device. Most cases this will be the *HardwareDeviceAdapter*, however, if a file recording system was created it could be passed into record the draw calls to a file.

6.1.7. Renderer

The *Renderer* is the base *Component* that is used to render images to the screen. Though the class itself is abstract, the *Renderer* is the only means of a *Component* to be given the objects to be able to draw to the hardware device. At the current iteration the only classes that implement *Renderer* is the *Camera* and *MeshRenderer*. The *Camera* is used to setup the viewport and manage both post-render and pre-render processes, where as the *MeshRenderer* is used to draw out the objects to the device.

- **Data**
  - Material : *Material*
    - The *Material* that will be used to render out objects.

- **Operations**
  - OnRender(device : *GraphicsDevice*) : *void*
    - Called when the *RenderExtension* is executed to render the screen and the object is not culled. The *GraphicsDevice* provided hides whether the rendering will be buffered or directly drawn to the screen.
  - OnBecameVisible(*void*) : *void*
    - Called when the object was originally culled from rendering but has now been allowed to render.
  - OnBecameInvisible(*void*) : *void*
    - Called when the object was originally rendered but has now been culled from rendering.

6.1.8. MeshRenderer

The *MeshRenderer* is the primary *Component* for rendering a Mesh to the screen.

- **Data**
  - Mesh: *Mesh*
    - The *Mesh* that will be used to render out objects.

- **Operations**
  - OnRender(device : *GraphicsDevice*) : *void*
    - Inherited from *Renderer; c*alled when the *RenderExtension* is executed to render the screen and the object is not culled.
  - OnBecameVisible(*void*) : *void*
    - Inherited from *Renderer;* called when the object was originally culled from rendering but has now been allowed to render.
  - OnBecameInvisible(*void*) : *void*
    - Inherited from *Renderer;* called when the object was originally rendered but has now been culled from rendering.

6.1.9. Camera

The *Camera* is a *Component* through which the player views the world. A screen space point is defined in pixels. The bottom-left of the screen is (0,0); the right-top is (*Width*, *Height*). A viewport space point is normalized and relative to the camera. The bottom-left of the *Camera* is (0,0); the top-right is (1,1).

- **Data**
  - Orthographic: *Boolean*
    - Whether or not to use orthographic projection.
  - BackgroundColor : *Color*
    - The background color of the *Camera*.
  - FieldOfView : *float*
    - The field of view of the camera in degrees.

- NearClippingPlane : *float*
  - The near clipping plane distance.
- FarClippingPlane : *float*
  - The far clipping plane distance.
- Aspect : *float*
  - The aspect ratio (width divided by height).
- Width : *int*
  - How wide is the camera in pixels
- Height : *int*
  - How tall is the camera in pixels
- Projection : *Matrix*
  - Matrix that transforms from world space to camera space
- View : *Matrix*
  - Matrix that transforms the model space to world space

- **Operations**
  - OnRender(device : *GraphicsDevice*) : *void*
    - Inherited from *Renderer; c*alled when the *RenderExtension* is executed.
  - OnPreCull(device : *GraphicsDevice*) : *void*
    - Called before a *Camera* culls the scene.
  - OnPreRender(device : *GraphicsDevice*) : *void*
    - Called before a *Camera* starts rendering the scene.
  - OnRenderImagel(device : *GraphicsDevice*) : *void*
    - Called after all rendering is complete to render image
  - OnPostRender(device : *GraphicsDevice*) : *void*
    - Called after a *Camera* has finished rendering the scene.

6.1.10. DeferredRenderCamera

ERIC THIS IS YOU

6.1.11. RendererExtension

The *RenderExtension* is the primary task used to render *Meshes* to the render screen. This extension spawns a multitude of subtasks to render out the *Meshes* and *Cameras* using the *BufferedDeviceAdapter*; once the tasks have completed, the *BufferedDeviceAdapters* are combined and played back using the *HardwareDeviceAdapter*.

Note: Most tasks will register themselves as parent dependencies of the RendererExtension, this makes the render task move to the end of the scheduling process.

- **Operations**
  - OnExecute(void) : void
    - Inherited from *Task;* Spawns the a multitude of sub-tasks to render the *Meshes* to the screen.

6.2. Input Subsystem

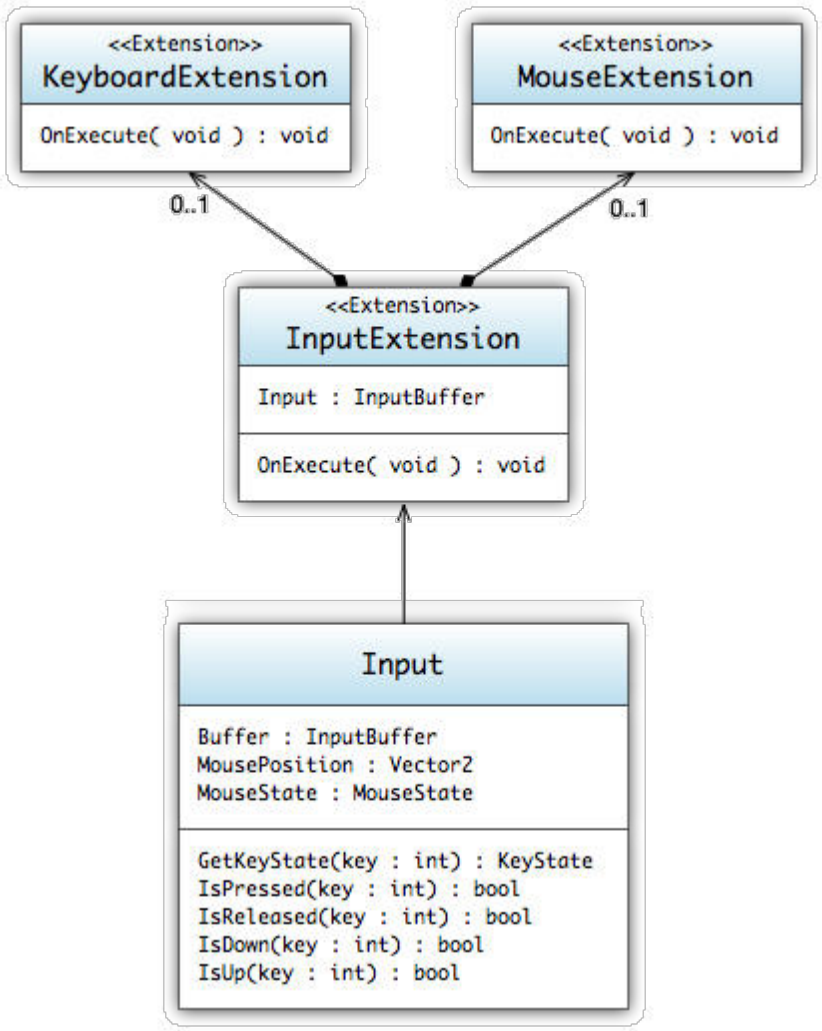


Figure ?? : Input UML Diagrams

The input system runs at approximately 30hz and gathers all of the inputs attached to the system. The system also provides intermediary calculations to provide easy interfaces for interacting with both the keyboard and mouse devices.

Note: Any tasks that require input feedback from the user inputs must register a dependency with the InputExtension.

6.2.1. InputBuffer

Manages and stores all of the input state.

- **Data**
  - Buffer : *InputBuffer*

        - The current buffered state of the input devices.
    - KeyStates : *KeyState[]*
        - The current states of all of the keys.
    - MousePosition : *Vector2*
        - The Current position of the mouse relative to the windows client area.
    - MouseState : *MouseState*
        - The current state of all of the mouse buttons.

6.2.2. Input

The *Input* class is a static thread-safe class that allows access to the device's state. The class is updated through the *InputExtension* and manages a double buffered system to prevent threading issues.

- **Data**
    - MousePosition : *Vector2*
        - Current position of the mouse relative to the window's client area.
    - MouseState : *MouseState*
        - Current state of the mouse; this constitutes which buttons are pressed, and where the scroll wheel is located

- **Operations**
    - GetKeyState(key : *int*) : *KeyState*
        - Gets the key's current state.
    - IsPressed(key : *int*) : *boolean*
        - Returns whether the key has been pressed; press constitutes an up event and then a subsequent down event.
    - IsReleased(key : *int*) : *boolean*
        - Returns whether the key has been released; release constitutes a down event and then a subsequent up event.
    - IsUp(key : *int*) : *boolean*
        - Returns whether the key is in the up state.
    - IsDown(key : *int*) : *boolean*
        - Returns whether the key is in the down state.

6.2.3. InputExtension

The *InputExtension* is the primary task used to gather the input state. This extension spawns both the *MouseExtension* and *KeyboardExtension* and manages the input state double buffer

- **Data**
    - Input : *InputBuffer*
        - The current input state that has been gathered.

- **Operations**
    - OnExecute(void) : void
        - Inherited from *Task*; Spawns the two other tasks, *KeyboardExtension* and

MouseExtension, and swaps buffers.

6.2.4. KeyboardExtension

The *KeyboardExtension* handles all of the calls to DirectInput keyboard device and updates the input state.

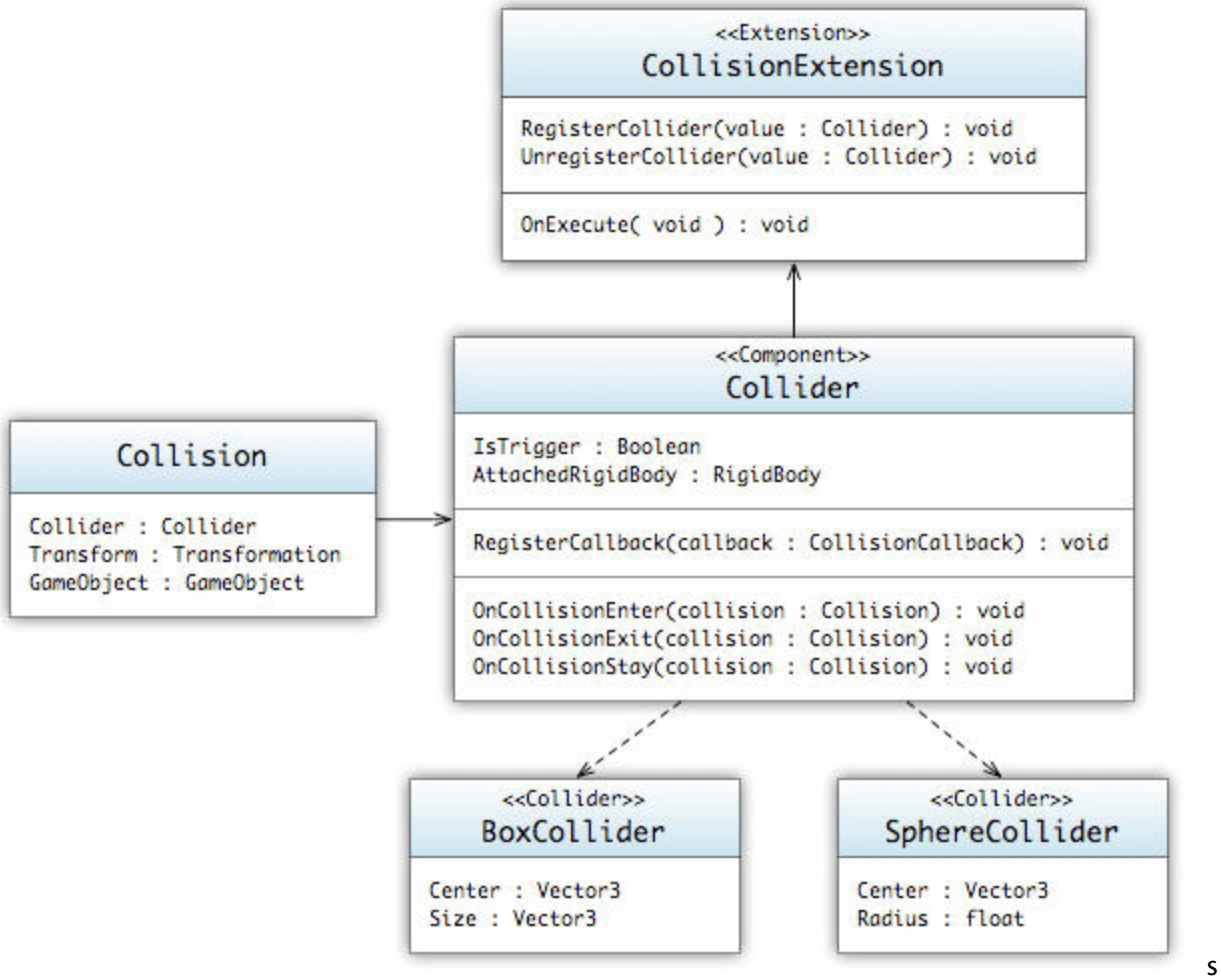
- **Operations**
  - OnExecute(*void*) : *void*
    - Inherited from *Task*; Queries and updates the input state.

6.2.5. MouseExtension

The *MouseExtension* handles all of the calls to DirectInput mouse device and updates the input state.

- **Operations**
  - OnExecute(*void*) : *void*
    - Inherited from *Task*; Queries and updates the input state.

6.3. Collision Subsystem



s

Figure ?? : Collision UML Diagram

The collision system is a collection of customizable *Collision* components that allow *GameObjects* to interact. Due to the inherent response that most physics systems have, *Collision* components are loosely tied to the physics subsystem's *RigidBody* component. Without the *RigidBody* attached the *Collider* has no means of influencing the *GameObject's* position. In this case, where a *Collider* has no companion *RigidBody*, the *Collider* acts as a trigger to a collision and has no interaction on the *GameObject*.

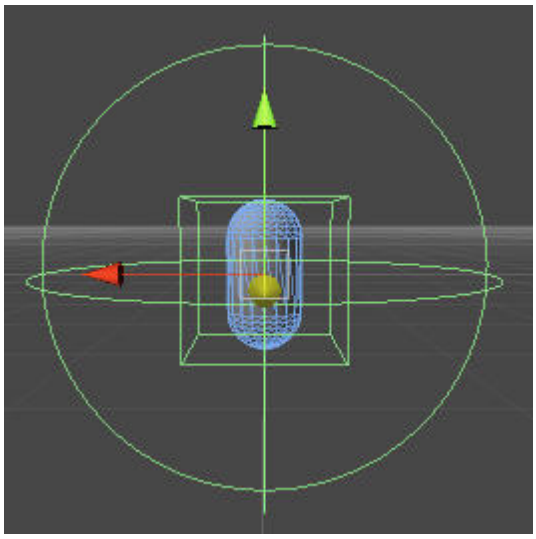


Figure ?? : Trigger and Physics Collider Combined

6.3.1. Collision

The *Collision* object is the primary data object used to track a collision between two *Colliders*.

- **Data**
  - Collider : *Collider*
    - The *Collider* that the current collision occurred with
  - Transform : *Transformation*
    - The world *Transformation* of the *GameObject* that the *Collider* collided with.
  - GameObject : *GameObject*
    - The *GameObject* that the *Collider* collided with.

6.3.2. Collider

A base class of all *Colliders*. *Colliders* are the means of which the system is able to manage and track the collisions between *GameObjects*.

Note: In order for two or more objects to collide, they both must have some form of a Collider(box, sphere, etc.) attached to them.

- **Data**
  - IsTrigger : *Boolean*
    - Whether or not the *Collider* is set to act as a trigger.
  - AttachedRigidBody : *RigidBody*
    - The *RigidBody* that the *Collider* is partnered with.

- **Operations**
  - RegisterCallback(callback : *ColliderCallback*) : *void*
    - Registers a callback that will be called when a collision is detected.

- OnCollisionEnter(collision : *Collision*) : *void*
  - Called when a the collision between two *Colliders* has just started to happen and the *Colliders* have "Entered" each others bounding volume.
- OnCollisionExit(collision : *Collision*) : *void*
  - Call when a collision between two *Colliders* has just ended and the *Colliders* have "Exited" each others bounding volume. This can only be called if a previous *OnCollisionEnter* has been called.
- OnCollisionStay(collision : *Collision*) : *void*
  - Called when a collision between two *Colliders* is still occurring and the *Colliders* are occupying each others bounding volume. This can only be called if a previous *OnCollisionEnter* has been called.

6.3.3. BoxCollider

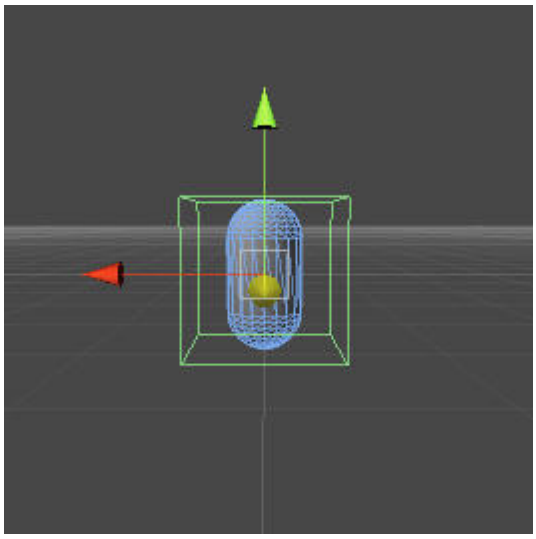


Figure ?? : Box Collider Diagram

The *BoxCollider* is a simple bounding box collision component.

- **Data**
  - Center : *Vector3*
    - The center of the box, measured in the *GameObject's* local space.
  - Size : *Vector3*
    - The size of the box, measured in the *GameObject's* local space.

6.3.4. SphereCollider

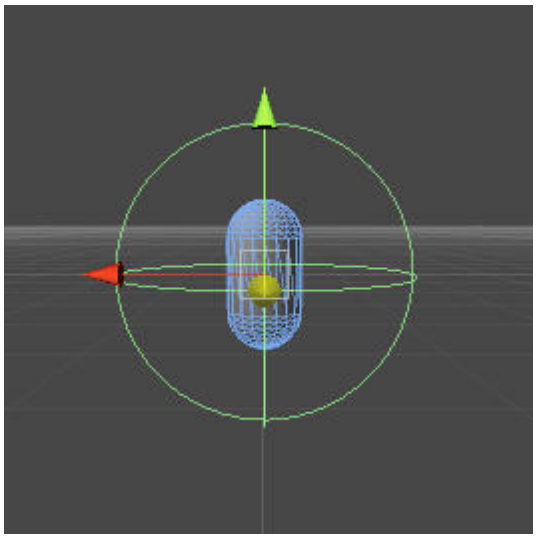


Figure ?? : Sphere Collider Diagram

The Sphere Collider is a simple bounding sphere collision component.

- **Data**
  - Center : *Vector3*
    - The center of the sphere, measured in the *GameObject's* local space.
  - Radius : *float*
    - The radius of the sphere, measure in the *GameObject's* local space

6.3.5. CollisionExtension

The *CollisionExtension* is the primary task that splits up and executes the collision tests for all of the registers *Colliders*. The extension uses the calls the *Scene* object to create subsets of *Colliders* that have no dependences, once this is completed each subset can be queued up as a task and run independently.

- **Operations**
  - OnExecute(void) : void
    - Inherited from *Task*; Spawns the a multitude of sub-tasks to check the collisions of all the *Colliders*.

6.4. Physics Subsystem

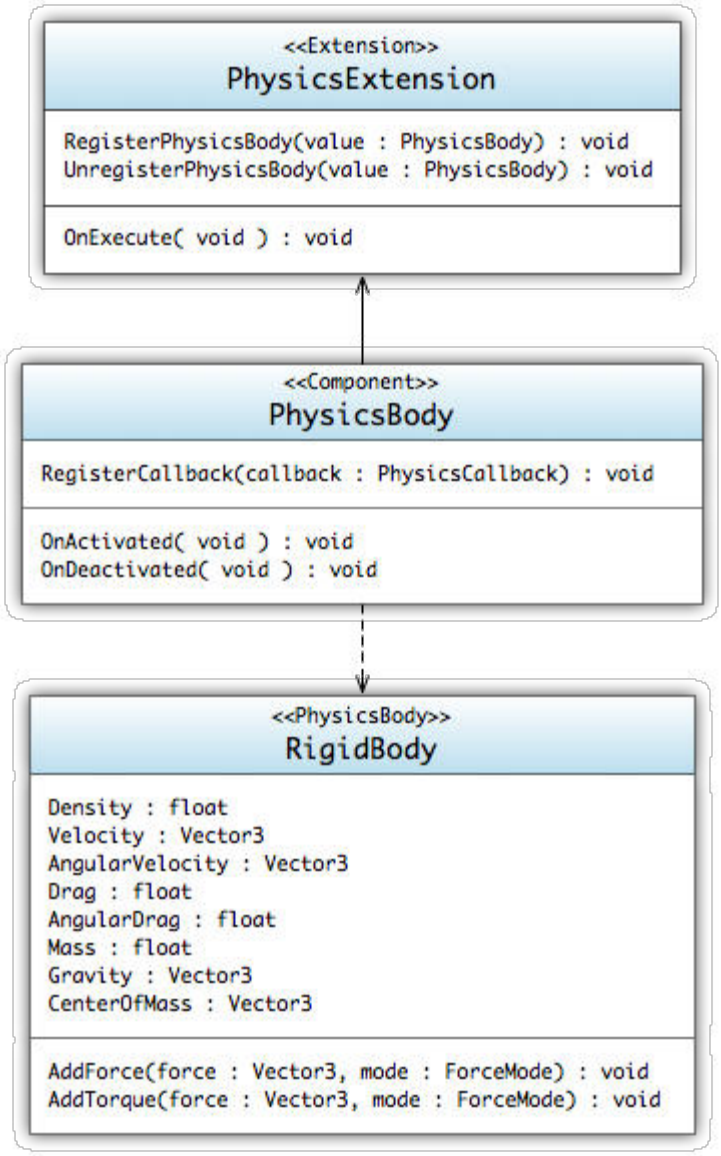


Figure ?? : Physics UML Diagram

6.4.1. PhysicsBody

Base class for all physics components.

- **Operations**
  - RegisterCallback(callback : *PhysicsCallback*) : *void*
    - Registers a method callback that will be called when the *PhysicsBody* is executed

6.4.2. RigidBody

Controls the GameObjects position through basic physics simulations.

- **Data**
  - Velocity : *Vector3*
    - The velocity vector of the *RigidBody*.
  - AngularVelocity : *Vector3*
    - The angular velocity vector of the *RigidBody*.
  - Drag : *float*
    - The drag associated with the *RigidBody*.
  - AngularDrag : *float*
    - The angular drag associated with the *RigidBody*.
  - Mass : *float*
    - The mass of the *RigidBody*.
  - Gravity : *Vector3*
    - The gravity vector of the *RigidBody*. If no gravity is wanted then set to Vector3(0,0,0).
  - CenterOfMass : *Vector3*
    - The center of mass relative to the transform's origin.

- **Operations**
  - AddForce(force : Vector3, mode : ForceMode) : void
    - Adds a force to the *RigidBody*.
  - AddTorque(force: Vector3, mode : ForceMode) : void
    - Adds a torque force to the *RigidBody*.
  - AddForceAtPosition(force : Vector3, position : Vector3, mode : ForceMode) : void
    - Applies a force at the specified position; the force will invoke a torque force as well on the object.
  - AddExplosionForce(force : float, position : Vector3, radius : float, mode : ForceMode) : void
    - Applies a force to the *RigidBody* that simulates the explosive force.

6.4.2. PhysicsExtension

The *PhysicsExtension* is the primary task used to simulate the physics on *RigidBody* components. With the help of the *Scene* object this extension spawns a number of subtasks to simulate the physics.

- **Operations**
  - OnExecute(void) : void
    - Inherited from *Task*; Spawns the a number of sub-tasks to simulate the physics.

6.5. Animation Subsystem

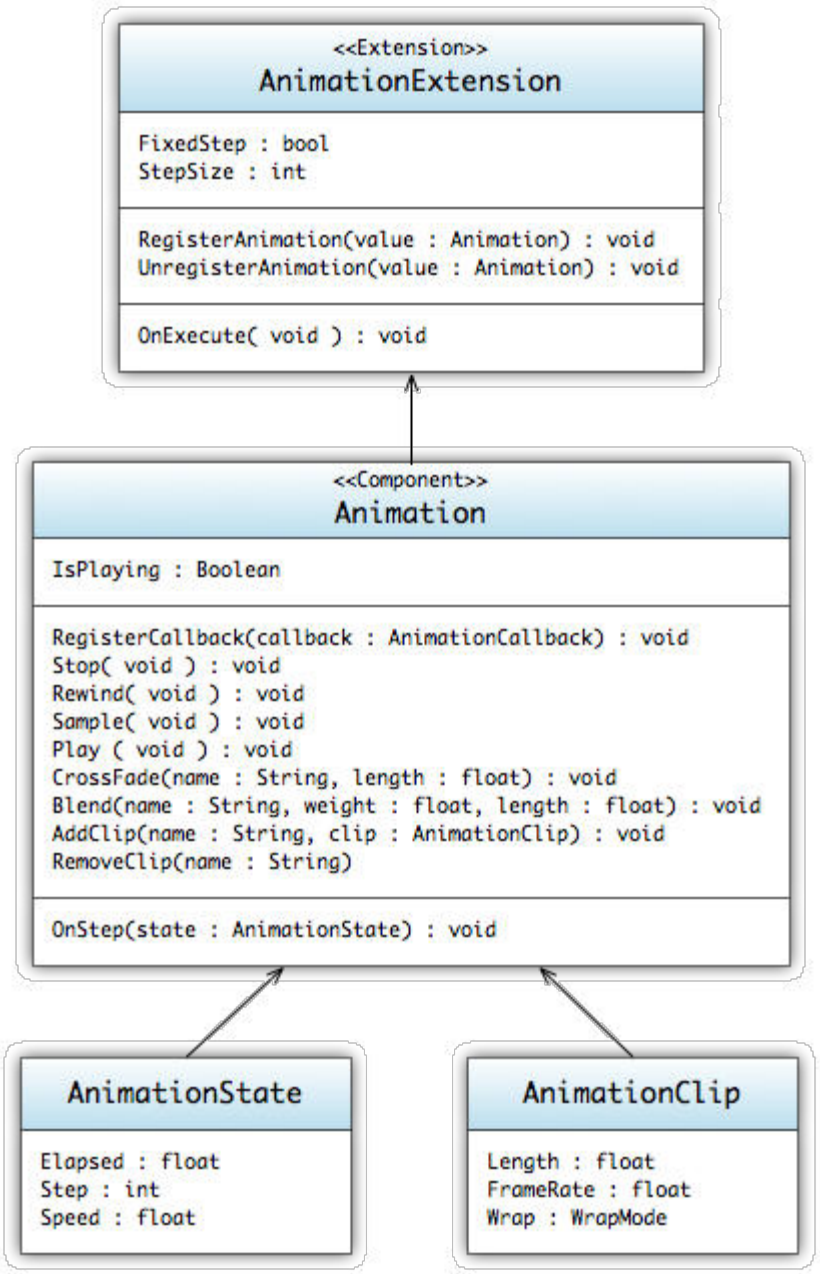


Figure ?? : Animation UML Diagrams

The animation subsystem is a composition of classes that utilize the built in animation formats, weighted keyframes, of the FBX file format. Although animation is primarily controlled via the loaded in approach, via models, the *Animation* component allows for uses to attach a callback to either calculate pre-processing or to actually animate the *GameObject*. The core object, the *AnimationClip*, is an Object used to store and playback the animation sequence defined by the model format. Due to its nature, the Component can be extended to add inverse-kinematics or any other animation systems

if the game development requires it.

Note: Since most animation sequences are directly tied to the render system, the AnimationExtension registers itself as a parent dependency of the RenderExtension

6.5.1. AnimationClip

Stores the keyframes for the animation clip. The keyframes are loaded from the FBX model definition.

- **Data**
  - Length : *float*
    - Length of the animation clip in seconds.
  - FrameRate : *float*
    - Frame rate at which keyframes are sampled.
  - Wrap : *WrapMode*
    - Sets the default wrap mode used in the animation state.

6.5.2. Animation

The *Animation* component is used to play back *AnimationClips*. The animation system is weight based and supports animation blending, additive animations, animation mixing, and full control over all aspects of the animation playback.

- **Data**
  - Clip : *AnimationClip*
    - The default animation clip
  - IsPlaying : *Boolean*
    - Whether or not an *AnimationClip* playing

- **Operations**
  - RegisterCallback(callback : *AnimationCallback*) : *void*
    - Registers a callback to be called when the *Animation* component is executed
  - Stop(*void*) : *void*
    - Stops all playing animations that were started.
  - Rewind(name : *String*) : *void*
    - Rewinds the specified *AnimationClip*
  - Sample(*void*) : *void*
    - Samples the animation at the current state.
  - Play(name : *String*) : *void*
    - Plays the specified *Animation*, if no name is provided then the default animation is started.
  - CrossFade(name : String, length : float) : void
    - Fades the current animations with the specified *Animation* over a period of time.
  - Blend(name : *String*, weight : *float*, length : *float*) : *void*
    - Blends the specified *Animation* into the currently running animations.

- AddClip(name : *String*, clip : *AnimationClip*) : *void*
  - Adds an *AnimationClip* to the *Animation*.
- RemoveClip(name : *String*) : *void*
  - Removes an *AnimationClip* from the *Animation*.

6.5.3. AnimationExtension

The *AnimationExtension* is the primary task used to step the animations through the animation clips. Due to the independent nature of animations, the animation processing is instantly split into as many as 10 separate running tasks to execute the animation sequences in parallel.

- **Operations**
  - OnExecute(void) : void
    - Inherited from *Task*; Spawns the *n* number of animation tasks.

6.6. Networking Subsystem

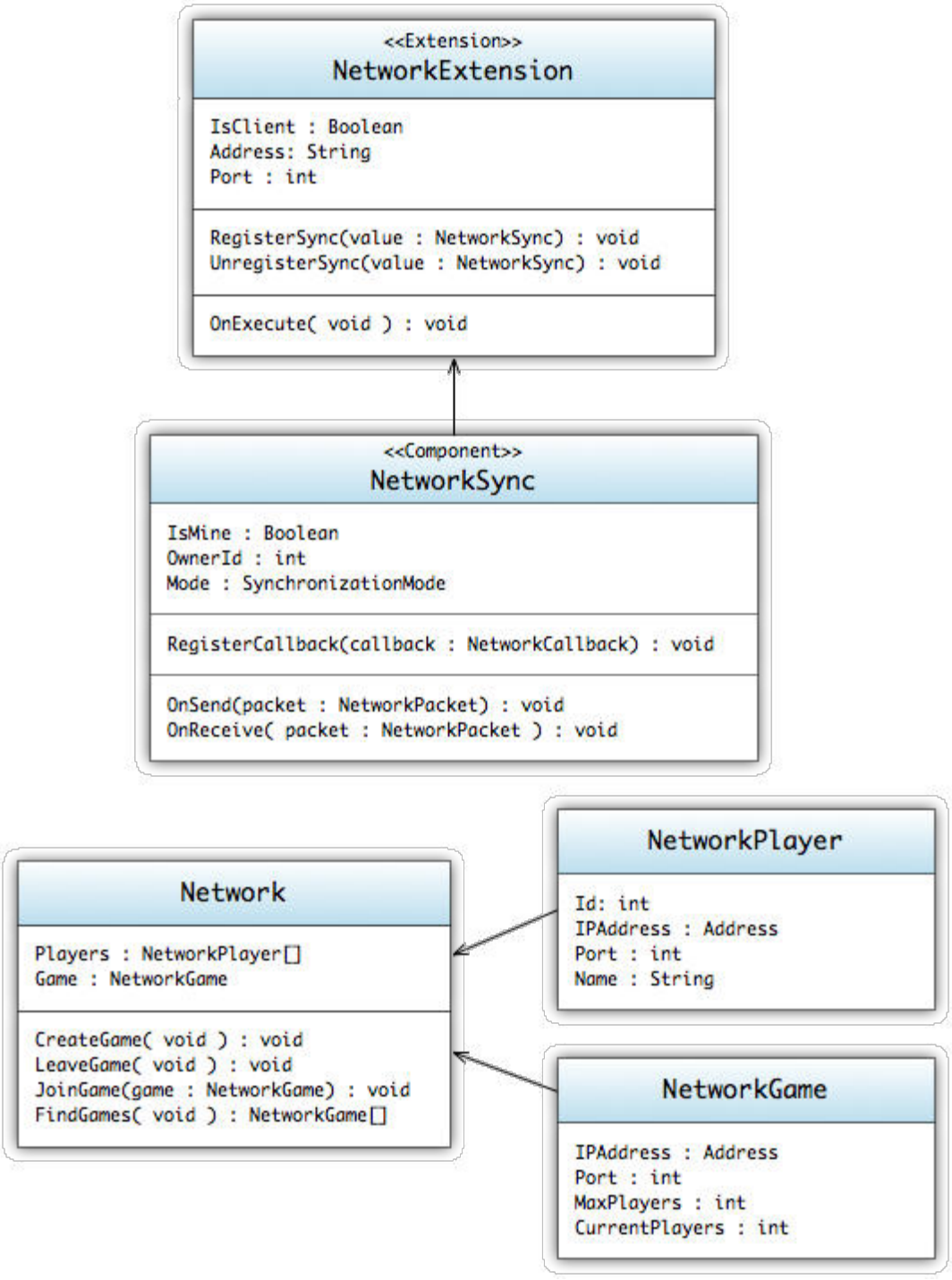


Figure ?? : Network UML Diagram

The networking system runs on multiple multicast networks to distribute and maintain synchronization among all of the connected clients. When the engine starts up it attaches to the "game" network. This network used to publish open games and other higher level commands

between clients. The information provided on this network is what is used in the game lobby so that users can choose which games they would like to join. Once joined, or if the player creates a game, a secondary multicast network is created to manage only information pertinent to that instance of a game. The *NetworkSync* works on this game dependent network and utilizes the small multicast network to synchronize objects between all of the clients. If a player leaves the update is register back on the "game" network and the game is updated; if the last player leaves, then the game is register as destroyed and the network shutdown and unregistered from the list of available games.

6.6.1. Network

The *Network* class is an interface to the networking communication system. The class provides information and methods to allow the player to join, leave, and create games.

- **Data**
  - Players : *NetworkPlayer[]*
    - The current players in currently running game; if no game has been joined, then the list is empty
  - Game : *NetworkGame*
    - The currently running game; if no game has been joined, then return NULL.

- **Operations**
  - CreateGame(*void*) : *void*
    - Creates a new game and publishes it to the game network.
  - LeaveGame(*void*) : *void*
    - Leaves the joined game.
  - JoinGame(game : *NetworkGame*) : *void*
    - Joins a game; updating the number of players and initializing the in-game network.
  - FindGames(void) : *NetworkGame[]*
    - Finds currently open games on the game network.

6.6.2. NetworkPlayer

The *NetworkPlayer* encapsulates the information needed to identify and track a networked player.

- **Data**
  - IPAddress : *Address*
    - The IP address of the networked player.
  - Port : *int*
    - The port of the networked player.
  - Name : *String*
    - Friendly name or username of the player.

6.6.3. NetworkGame

The *NetworkGame* encapsulates the information needed to identify and track a running networked game.

- **Data**
  - IPAddress : *Address*
    - The IP address of the network game.
  - Port : *int*
    - The port of the networked game.
  - MaxPlayers : *int*
    - The max number of players allowed in the game.
  - CurrentPlayers : *int*
    - The current number of players joined in the game

6.6.4. NetworkSync

The *NetworkSync* is a component that when attached to a *GameObject* will sync the information from other components and transform information to other clients of the game. There are two modes of synchronization, Reliable(TCP) and Unreliable(UDP). Depending on what mode is selected will determine the protocol that is used to transfer the information between clients.

- **Data**
  - IsMine : *Boolean*
    - Whether the object being synchronized is the responsibility of the current client.
  - OwnerId : *int*
    - The id of the *NetworkPlayer* who has responsibility of the object.
  - Mode : *SynchronizationMode*
    - This sets which protocol(UDP/TCP) the network uses to synchronize the information between clients.

- **Operations**
  - RegisterCallback(callback : *NetworkCallback*) : *void*
    - Registers a callback that is called whenever a synchronization happens between the client and host.
  - OnSend(packet : *NetworkPacket*) : *void*
    - Called when a synchronization packet is sent out.
  - OnReceive(packet : *NetworkPacket*) : *void*
    - Called when a synchronization packet is received.

6.6.5. NetworkExtension

The *NetworkExtension* manages the connection between both the "Game" network and the game instance networks. When run, the extension will spawn of packet builder tasks that go and collect the synchronization information needed to be sent. Once the packet data is collected, the system will bundle that up and send it out to the other game clients.

- **Operations**
  - OnExecute(void) : void
    - Inherited from *Task*; Collects the synchronization data and then packages it up and

sends it out to game clients.

6.7. Lua Scripting

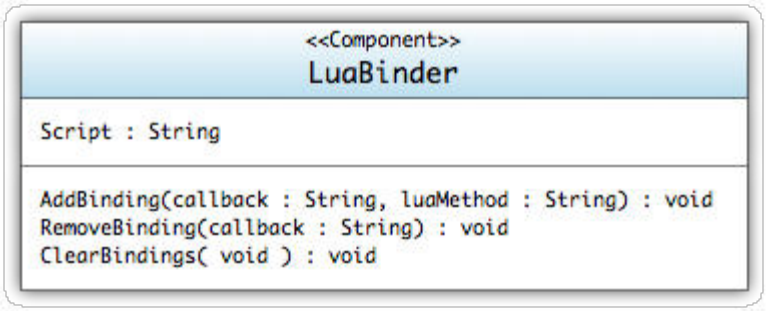


Figure ?? : Lua UML Diagram

The Lua script interpreter is one of the main ways for designers/developers modify object behaviors. Instead of building off of the *Extension/Component* architecture of most subsystems the lua subsystem uses a single *Component*, the *LuaBinder*. The main purpose of the *LuaBinder* is to manage the bindings between *Component* callbacks and the lua script methods.

6.7.1. LuaBinder

The *LuaBinder* is a *Component* used to bind Lua snippets to the *GameObject* and *Component* Method callbacks. Execution of the Lua script is run immediately and is given access to the objects scope. Management of the *LuaState* is held through global variables and management across all *LuaBinders*.

- **Data**
  - Script : *String*
    - Lua Script file to reference

- **Operations**
  - AddBinding(callback : *String*, luaMethod : *String*) : *void*
    - Adds the callback binding to the *LuaBinder*; when the callback is called, the method will be redirected to the lua scripted method.
  - RemoveBindings(callback : *String*) : *void*
    - Removes the callback binding
  - ClearBindings(*void*) : *void*
    - Clears all of the callback bindings.

6.8. Particles Subsystem

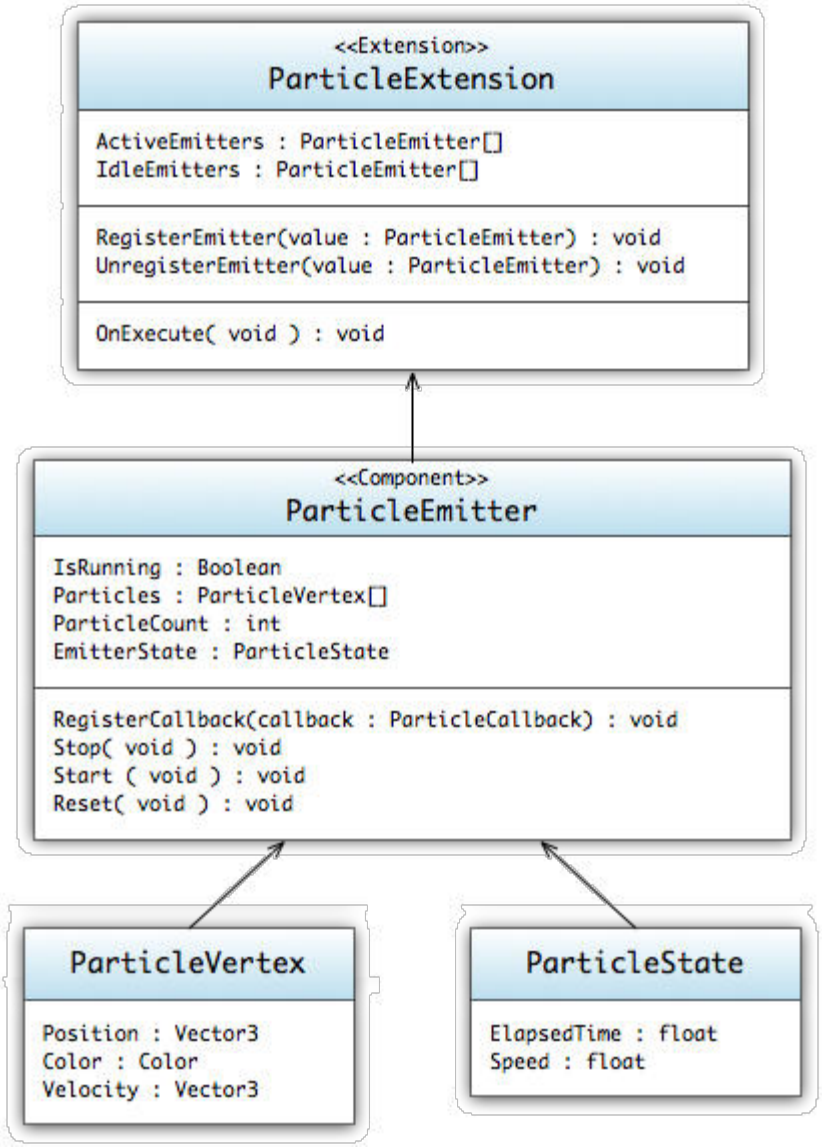


Figure ?? : Particle UML Diagram

The particles subsystem is a composition of classes that will render particle effects to the screen. The subsystem will be managed by a *ParticleExtension* which handles all particle emitters created, running, and idling through the game. The *ParticleExtension* will have a series of *ParticleSubTasks* that will divide up the tasks for the *TastExtension*. *ParticleEmitters* will be created and attached to *GameObject*s and from there will handle the rendering of the particles. The emitters will store *ParticleVertexs* and *ParticleState*s to assist in keeping track of the progress of the particle rendering.

6.8.1. ParticleState

The *ParticleState* will be responsible for keeping track of the time along the emitter's lifespan. The

timer's data will translate into the *ParticleVertex* itself via particle age.

- **Data**
  - ElapsedTime : *float*
    - Amount of time that has passed since the emitter has started.
  - Speed : *float*
    - The rate at which the the timer ticks

6.8.2. ParticleVertex

The *ParticleVertex* will hold all the possible data that a particle will need to render. This data is passed into the *VertexBuffer* and to the *GraphicsDevice*. Shaders will take the data in from the vertices and render them out to the screen. *ParticleVertex* inherits from *VertexDeclaration* so all calls from there are pulled into *ParticleVertex*.

- **Data**
  - Position : *Vector3*
    - The starting position of the vector
  - Color : *Color*
    - Color of the vertex
  - Velocity : *Vector3*
    - The velocity that alters the position of vertex over time.

6.8.3. ParticleEmitter

Used to render *ParticleVertexes*. The emitter will keep track of a vertex buffer which will house all of the *ParticleVertexes* to be rendered. Emitters attach themselves to *GameObjects* as components so they can be recognized and rendered by the renderer. Emitters will need to keep track of their overall lifetime and translate that information to the particles.

- **Datas**
  - IsRunning : *Boolean*
    - Whether or not the *ParticleEmitter* is currently running.
  - Particles : *ParticleVertex*[]
    - Array of particle vertices that will be passed into the vertex buffer
  - ParticleCount : *int*
    - The max number of particles the emitter can hold
  - EmitterState : *ParticleState*
    - Timer object that keeps track of the emitter's lifespan

- **Operations**
  - RegisterCallback(callback : *ParticleCallback*) : *void*
    - Registers a callback event whenever the particle emitter component needs to be rendered.
  - Stop(void) : void
    - Stops the *ParticleEmitter* from moving to the next display state.

- Start(void) : void
  - Starts the *ParticleEmitter*.
- Reset(void) : void
  - Resets the *ParticleEmitter* to its initial state.

6.8.4. ParticleExtension

The *ParticleExtension* is the workhorse of the particle subsystem. It is required to manage all active and idling emitters. When a call is made for a *ParticleEmitter* to be created, it is the extension's job to create this object. It is also the emitter's responsibility to kill any active emitters when their lifespan has passed. When an emitter's lifespan reaches its max, it is taken out of the active queue and placed into an idle queue to be used later. This helps with reducing the need to create new emitters as a game progresses on.

- **Data**
  - ActiveEmitters : *ParticleEmitter[]*
    - Full list of emitters that are currently active in the system
  - IdleEmitters : *ParticleEmitter[]*
    - List of emitters that have gone through the active system and are currently waiting to be used again

- **Operations**
  - OnExecute(*void*) : *void*
    - Inherited from *Task*; Executes and spawns particle sub-tasks.

6.9. Audio Subsystem

6.9.1. AudioClip

6.9.2. AudioSource

6.9.3. AudioListener

6.9.4. AudioExtension

6.9.5. ??SelaAudioExtension??

6.10. GUI Subsystem

6.10.1. Panel

6.10.2. Label

6.10.3. Button

6.10.4. Render2DExtension

6.11. Content Subsystem

6.12 Game Specific Components

6.12.1 HealthManager

6.12.2 ResourceManager

6.12.3 GameManager

7. Game Objects

7.1. Modifier

7.2. Weapon

7.3. Projectiles

7.4. Geometry

7.5. Player

7.6. Level