SharpMedia Scene Design

© 2007 Žiga Osolin – SharpMedia project, document version: DRAFT

Table Of Contents

[About 3](#_Toc210922382)

[Goals 3](#_Toc210922383)

[Scene Graph 4](#_Toc210922384)

[SceneNode 4](#_Toc210922385)

[Scene Component 5](#_Toc210922386)

[Scene Manager 5](#_Toc210922387)

[Rendering 7](#_Toc210922388)

[Camera 7](#_Toc210922389)

[LOD 7](#_Toc210922390)

[RenderingProgram 7](#_Toc210922391)

[Material 7](#_Toc210922392)

[Lights and Lighting 7](#_Toc210922393)

[ShaderSuppliedVariable, ShaderSuppliedConstant, ShaderSuppliedTexture 8](#_Toc210922394)

[Shadowing 8](#_Toc210922395)

[Transformation 8](#_Toc210922396)

[Rendering Data 9](#_Toc210922397)

[Custom rendering 9](#_Toc210922398)

[Occlusion Culling 9](#_Toc210922399)

[Rendering order 10](#_Toc210922400)

[Batching 10](#_Toc210922401)

[Examples 10](#_Toc210922402)

[Query 11](#_Toc210922403)

[Animation 12](#_Toc210922404)

[Scene Node Animations 12](#_Toc210922405)

[Morph Animation 12](#_Toc210922406)

[Skeletal Animation 12](#_Toc210922407)

[Particle System 12](#_Toc210922408)

[Physics 13](#_Toc210922409)

[StaticObject 13](#_Toc210922410)

[RigidBody 13](#_Toc210922411)

[Fluid 13](#_Toc210922412)

[Cloth 13](#_Toc210922413)

[DeformableBody 13](#_Toc210922414)

# About

Scene library is a sort of a system connector. This is the place where graphics, physics, sound and other facilities are connected. It implements shared parts of system, higher level concepts of enumerated sub-systems and »smart« decisions based on knowledge of all system. It is also responsible for sub-system synhronization, data holding and loading, runtime (rendering) optimisations and so on.

# Goals

The Scene library must be easy to use. This is probably one of the most complex library and the API that will probably be most usable to the engine user. This makes Scene design even more important. Scene links and provides high level concepts to the following libraries:

* Graphics API and Vector API (this one is based on Graphics API anyway);
* Physics API;
* Sound API;
* Scripting API;

Scene must also provide:

* Hooks for editor – this enables easy and natural communication from editor to scene;
* Extandability – new libraries can use scene or intercooporate with scene naturally. For example, the GI (global illumination) tool can use Scene library for queries
* Isolated modules – it is not required the whole scene to be initialized with graphics device and everything else what brings overhead and sometimes unwanted tasks. You could use only modules of scene, such as only material library (for example from Material Editor) or Scene graph (for the global illumination);
* Automated optimisations such as:
  + Frustum culling;
  + Occlusion culling (dynamic and static);
  + Automatic batching;
  + Automatic instancing;
  + Limiting state switches;
  + Light range optimisations, light culling;
  + Deferred rendering switch.

# Scene Graph

Scene is organized into hierachial structure. All data is contained within this structure as node properties or node components. Processing on such scene graph is done (mainly) by scene managers.

## SceneNode

A **SceneNode** is the only allowed building block of this hierarchy. It is sealed. Each **SceneNode** has the following properties/methods:

* Name – this property can be used for searches and identification. No special global uniques is required;
* Position and orientation – it can be accessed as vector and quaternion pair or as matrix exposing the pose. The internal storage is as vector and quaternion because it is more efficient this way;
* Static – is the node static or not. Static node's position cannot be changed, as well as orientation;
* Children – a list to other scene nodes. You can add or remove children;
* Locking mechanism – a scene node can be made immutable/mutable using locking mechanism. Components must obey those rules;
* Scene node components[[1]](#footnote-2) - actual representation/function of node is hidden in there. Components may be searched by type and name. They can also be added/remove at runtime;
* »Local« variable collection – useful for scripts and also other components to share data;
* Inherited list – a prefab node that this node extends.

Scene node's children must follow the »contained in parent« manner. More information about how scene node can be used to represent anything is discussed in further sections.

## Scene Component

A **scene component** is simply an object that is attached to scene node. While you can attach any object as component, the component usually holds some sort of data about the object that can be used by managers in some way[[2]](#footnote-3). A scene component may be **Mesh** that holds geometric data about the node; rendering manager may use it to render the node. A component may be shared between more scene nodes.

Because scene components sometimes need more information about the scene node or some sort of synchronization, we provide some standard interfaces. Those interfaces are recognized by **SceneNode** and some managers. Note that interfaces are optional for components:

* **ISCNotify –** the interface has Attached and Detached methods that are called when the scene component is attached or detached from scene node;
* **ISCNamed** – named searches can be done on those components.

Scene component can be tagged with **[SceneComponent]** that allows setting restrictions on number of instances of the same type per node, the query type and user visibility. If the attribute is not present, the default configuration is taken (a singleton with node, query type set to component and visibility set to User). More about specific components is explained in the following sections.

## Scene Manager

A **SceneManager** is the root scene controller. It is given the root **SceneNode** and **IManager**s. A manager is worker – it is invoked by scene manager as described by update rules. Update rules include calls before/after some other manager is invoked, on time basis, whenever enough CPU or memory is available etc.

# Rendering

In this section, rendering concepts are described and how they are applied to scene node as scene components. The most important rendering concepts are the availability of camera, lights and objects to render. The query system that is extensively used by rendering system to configure objects is introduced later – here it is assumed to be working.

**SceneRenderingManager** is the root of rendering system. **SceneRenderingManager** should be attached to **SceneManager** if you want rendering effects. The manager renders from all active cameras and uses rendering information to achieve this task.

## Camera

A camera is described by **Camera** scene component. It holds information about the rendering target, viewport, projection matrix, near/far clipping distance, desired update frequency and attached post effects. The camera must be attached to scene object to gain position and orientation. There may be many cameras at scene, while only some of them are active. To activate a camera, you simply attach it to the list of active cameras in **SceneRenderingManager**.

**SceneRenderingManager** always adds a runtime scene component **CameraContext** to the active camera behind the scenes. Here, additional render targets needed for camera’s post effects are stored. When camera becomes inactive, the context is either destroyed or attached to some other camera.

## LOD

Level of detail can be controlled through scene node component that have query type **ILevelOfDetail**. A level of detail must provide level of detail index given a scene node and camera. The most common level of detail components are **NoLevelOfDetail** and **DistanceLevelofDetail.** If level of detail component is not applied, scene node acts as if **NoLevelOfDetail** was applied.

## RenderingProgram

A rendering program is a structure that returns the **ShaderCode**s for vertex, geometry and pixel shaders as well as the rendering states given a level of detail required for the object.

If the **RenderingProgram** is not applied to node, it functions as if the default rendering program was applied to scene node.

## Material

**Material** is a structure that returns **IBDRF** given a level of detail required for the object. **IBDRF** describes the material properties in very generic way. Not that BDRF functions usually include materials. The most common BDRFs are **DiffuseBDRF, SpecularBDRF, PhongBDRF, TextureBDRF, ComposedBDRF** …

## Lights and Lighting

A light component is anything derived from **ILight**. Built-in light types are **PointLight, DirectionalLight, SpecularLight, AmbientLight** and **CubeLight**. Whenever there is an **ILight** attached to scene node, the node will also act as a light.

Lighting is automatically enabled for all object that have the **LightingOperation** in one of its shaders. You can control lighting more precisely using the following scene components applied to scene node:

* **Lighting** – can define maximum number of lights, enable/disable lighting, override the search distance etc.
* **IgnoreLights** – can ignore some lights, useful if lights very prebaked by some tool and need not (must not) affect the object.
* **ILightingCombine** – defines how lights are combined in **LightingOperation**.
* **ICustomLighting** – defines a custom lighting, this will be automatically called every time lighting occurs.

## ShaderSuppliedVariable, ShaderSuppliedConstant, ShaderSuppliedTexture

Those scene components are recognized by rendering manager and bound to shaders as appropriate. The components are named and name is used to link it to shader (shader constants are named). The variable allows setting node’s local variables to shader, supplied constant allows setting a constant (that can be changed “on hand”) and supplied texture allows setting a texture object.

## Shadowing

Shadowing can also be controlled by scene node. There are three options for shadow casting and receiving: yes, no or don’t care. The default is “don’t care” for both, but this can be modified using shadowing information classes:

* **ShadowReceiver –** can set the shadow receiving of scene node to **Receive, DontReceive or DontCare;**
* **ShadowCaster –** can set shadow casting to **CastShadows, DontCast or DontCare;**

Special shadowing algorithms may also allow/require object to contain specific information about that shadowing technique. Furthermore, many shadowing techniques may store cached shadowing data in the object itself (for example, stencil shadows may store shadowing volumes for static lights in the scene node directly, this may actually be a pre-processed result).

Shadowing technique can be chosen in render manager. There may also be no shadowing.

## Transformation

Transformations on rendered data are more or less automatically applied to shaders through **TransformOperation**. If you scene node requires special transform treatment (if data is for example already stored in world coordinates), than apply one of the following scene components to it:

* **MeshInWorldCoordinates –** mesh is already stored in world coordinates;
* **ICustomTransform** – overrides transform based on position/quaternion and camera.

## Rendering Data

Information how to render data is held in several different manners. The most common is **Mesh** that holds data in **vertex buffer** plus optional **index buffer** manner. Mesh can be used (with proper configuration) for almost everything; from procedural algorithms (adding vertices in geometry buffer stage) to bone based animated characters.

The other rendering data representations useful are **IArea2f** and **IArea3f** objects. If a scene component targets it s query type to one of those interfaces, it will also be dynamically tessellated and rendered correctly.

The third data worth mentioning is the world geometry. This data is usually not represented as **Mesh** but as a hierarchical structure that is *converted* to mesh on runtime, based on the camera. Example of world geometry would be **BSPMesh**.

FIMXE: need more information how world geometry would be stored, how interaction with other scene nodes would be resolved and how the scene graph would be kept as hierarchical as possible.

There is no limit on the number of rendering data per node. All share the same shader program, state and so on.

Other representations are discussed in other sections. One example of such is **ParticleSystem**.

## Custom rendering

Custom rendering can be implemented if **ICustomRender** interface is present. Such renderer must provide:

* A **Render** method, invoked with **GraphicsDevice**, **Camera** and **SceneNode**. Graphics device is configured to render to camera’s render target. The rendering is expected to keep all states the same as they were before the call;
* **BoundingBox** property that is used to cull the object.

Frustum and occlusion culling are working for such objects (occlusion culling only as receiver), as well as Z-ordering flags. However, custom rendering affects performance because batching is not allowed.

## Occlusion Culling

Occlusion culling is controlled through the following scene components:

* **Occluder –** the object is potencial occluder (sureness parameters may be added);

By default, no object is occluder and all objects receive occlusion. Occlusion culling can be enabled/disabled in rendering manager.

## Rendering order

Rendering order of objects may need to be specified. This is useful for blending effects or when omitting the use of Z-buffers. To apply a special ordering rule, you can use the **RenderingGroupIndex** scene component. This assigns an object to rendering group index.

Rendering groups must be registered to **SceneRenderingManager**. Each rendering group has an index and information about the ordering (irrelevant, front-to-back, back-to-front). Groups are rendered according to their indices (first group 0, then group 1 …).

## Batching

Batching is automatically done by **SceneRenderingManager** if feature is enabled. Because batching usually needs at least some runtime feedback of scene, you can apply **BatchingInfo** that contains information about how often the object will be moved. This information can (in future) also be adjusted at runtime by the batching system itself.

Batching system adds runtime scene components to nodes that contained “batched meshes” of multi geometry.

## Examples

We list some example nodes:

* Node (Name=”Alien”)
  + Mesh
    - Lod 0: an alien mesh with texture coordinates
  + Material
    - Lod 0: DiffuseBDRF { Colour: Red }
  + RenderingProgram
    - Lod 0: all states normal, rendering program that simulates textures mapping and normal mapping (“Texture Map”, “Normal Map” texture)
  + SuppliedShaderTexture (LinkName=”Normal Map”)
    - Lod 0: normal map for the object
  + SuppliedShaderTexture (LinkName=”Texture Map”)
    - Lod 0: texture map for the object
  + … (other non rendering data)

# Query

In this section, mechanisms for fast scene query and organisation are described. The query system is controlled by **QueryManager**. The manager helps itself by injecting cache components into scene nodes.

The query system depends heavily on the spatial organisation of scene. The following queries are present:

* Find scene node(s) intersecting the ray, find first point of intersection;
* Find scene nodes in box/capsule/sphere/frustum;
* Find scene nodes matching a name pattern;
* Find scene nodes that contain specific component type and where components match some user provided pattern;
* ...

The query system is tighly integrated with rendering system because it is based on the same information.

TODO: depends heavily how world geometry is handled. Should also resolve the caching mechanisms.

# Animation

There are several kinds of animations available on scene node. Animation system also supports immediate results, such as deformations of object based on impact.

## Scene Node Animations

Scene node animations change the position of scene node. To apply this kind of animation to scene node, add a **NodeAnimator** scene component. The node animator contains information about the current *time* and *animation track*. **NodeAnimator** uses the **NodeAnimation** objects to change position of scene node. Animations are stored as **NodeAnimationTrack**s, each holding several keyframes about the position offset and orientation at current track. Interpolation is used to generate smooth animation.

## Morph Animation

In order to create morph animation, a special mesh type **MorphAnimatedMesh** must be used. The mesh has two parts; a shared vertex buffer and index buffer part and a per keyframe based vertex data. Keyframes are further grouped into animation tracks. Animation can either be done in software or hardware.

Morph animation is not recommended because a lot information is usually required. Try using skeletal animation instead if applicable.

The system sometimes uses morph animation for generating »interpolated picture« of deformations on impacts.

## Skeletal Animation

Skeletal animation can be applied to regular **Mesh** object. The mesh must have additional information about the bone indices and weights.

Skeletal animation is applied to scene node using the **SkeletalAnimator** component. The animator uses **SkeletalAnimation** object to receive bone matrices in given animation tracks at given time. The animator also support animation track mixing.

## Particle System

If you want you scene node to be a particle system, you apply a **ParticleSystem** scene component.

Particle system is composed of configure emitters, sinks and colliders. The animation system simulate the particle system for you given the properties. This kind of system will run on GPU. A particle system may also be updated by physics (fluids). In this mode, particle system will still animate some variables (colour, size) while leave the emission of new particles, their destruction and movement to physics.

# Physics

Physics scene components are just the basic physics types. The **ScenePhysicsManager** will hold (possibly multiple) **PhysicsScenes** and update the on regular basis.

## StaticObject

A **StaticObject** scene component (actually implemented by Physics library) can be added to scene node to add collision detection that will be takeninto account by physics. Note that collision primitives are usually used and the actual rendering representation is usually only approximated to speed up the collision detection.

## RigidBody

A **RigidBody** scene component (actually implemented by Physics library) can be added to scene node. This will make object physically responsive. The physics manager will update the scene node's position and orientation based on body's position everytime that position changes.

## Fluid

A **Fluid** scene component (actually implemented by Physics library) is not useful without data representation. In order to present data, they must be somehow transfered to rendering format. The scene node components are searched for **IFluidSynhronizable** (implemented by **ParticleSystem**) to find a synhronization point.

## Cloth

## DeformableBody

1. Do not confuse Scene node components with configuration components. Scene node components are (usually) not configured. [↑](#footnote-ref-2)
2. A data may be that that scene component actually exists at the node itself. [↑](#footnote-ref-3)