

# Functionality-based Hierarchical Scenegraph

How to create an easy to handle, debug and maintain  
scenegraph/entity manager for your game project

Publish Date: 22/may/2007  
Latest Revision: 09/feb/2009  
areiacreations.com

by Georgios Tryfonas

*Certificate in Computer Science and Audio Technology  
BSc(Hons) in Computer Science with Games Development*

**References:** *No resource specific information was used*

**Disclaimer:** *The following text is not an approved academic or scientific article and should not be regarded as so. It's based on independent research and the author doesn't hold any responsibility for incorrect information that may be included in this article or possible misinterpretation of the outcome you may produce using this material. You are free to use any of the following information for your own research or your project. In the case of this article's reproduction or representation (on printed, offline or online form), credits should be given to the author or areiacreations.com and this header should be included.*

**Warning:** *The sample code displayed in this article is not complete and is not structured for immediate adaption into your project. It's rather a guide on how you could actually apply this design.*

# CONTENTS

1 Introduction .....	3
2 Data representation .....	4
2.1 Type representation .....	4
2.2 Hierarchical representation.....	5
3 The Hierarchy .....	6
3.1 Tree hierarchy: SceneNode .....	6
3.2 Transformation: GameObject3D.....	7
3.3 Rendering: ModelNode .....	9
3.4 Physics: WorldModel.....	10
3.5 AI: Entity .....	11
4 The Objects .....	12
4.1 The World .....	12
4.2 Buildings .....	14
4.3 Terrain .....	14
4.4 Zones .....	14
4.5 Game-level objects.....	15
5 Conclusion .....	16

# 1 INTRODUCTION

A game world may contain many thousand objects, each with its own rendering, physics and logic settings. These objects need to be organized in a way that the information is accessible in real-time, so all the required functions can be performed on each object. Having the game's objects organized is also vital for making the game more efficient by performing functions such as determining the objects that should actually be sent to the graphics pipeline or the objects that have chances colliding.

In a large game world there might be objects that need to share some information. For example imagine a small building with a table in its interior. Then on top of that table there is a fork. If the table is moved for some reason, the chances are that you want the fork moved as well. The same should happen for the table when the building is moved. Therefore the table's position should be relative to the building's position and the fork's position should be relative to the table's position. Furthermore it's not only about position. It's probably about orientation (rotation) and scaling (size) too. In this case it said that the fork is a child of the table and the table is the parent of the fork. In the same fashion the table is a child of the building and the building is the parent of the table.

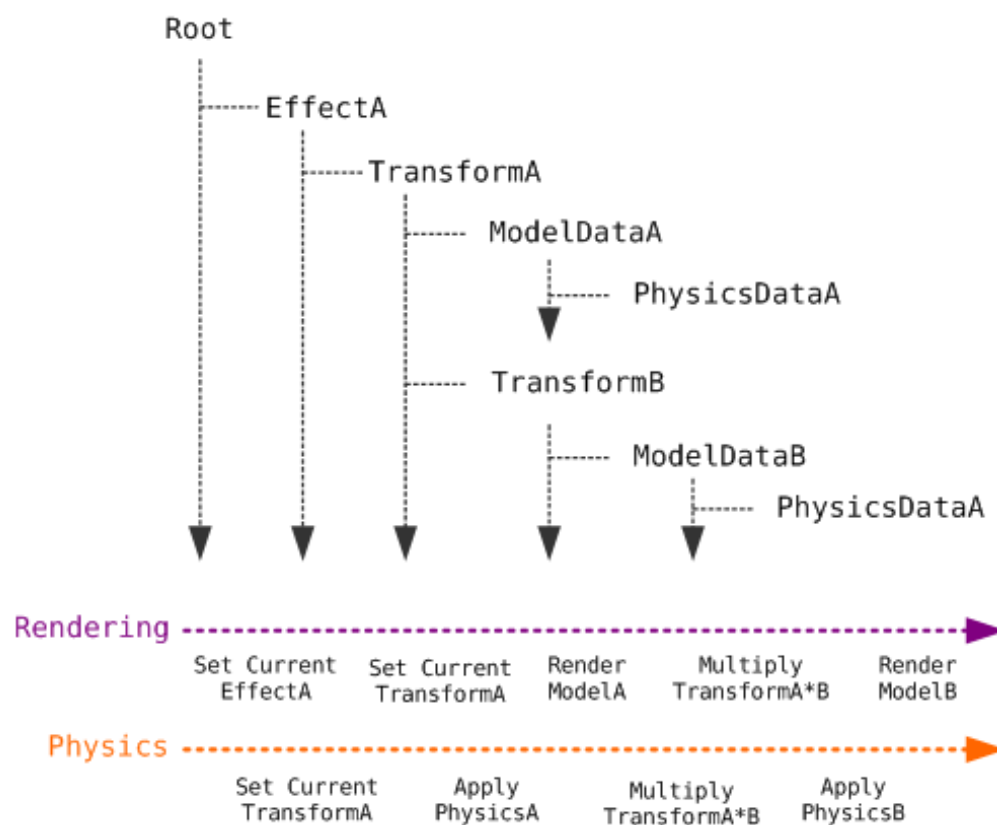
So how can we accurately describe this relationship in programming? In this case, the best solution seems to be a tree. A tree structure (as its name implies) represents the structure of a real tree. It has a root, many branches (that can have many branches as well) and each branch can have many leafs. A tree's branch in a scenegraph is called a node. That might be a good answer on how to represent the required child/parent relationship but there is another issue: how to represent the actual data?

## 2 DATA REPRESENTATION

The data that each object requires to have its functions performed can vary a lot depending on the game genre. For the purpose of this article we are going to assume that we are creating a scenegraph for a generic 3D game. So what is the data that a game object needs in this case? When a game object is rendered on the screen, the GPU actually reads the model data containing information on what to render. Then the GPU also needs information about how to render this data. This information is called an effect or a shader. What about the position, orientation and the scaling of the object? All this information can be saved in a convenient (and parent/children relationship friendly) way, called a transformation matrix. Additional data is also required for the physics, AI and other functions. So how this data should be represented?

### 2.1 TYPE REPRESENTATION

An obvious way is to create different types of nodes that carry one kind of data each. For example a transformation node, would just carry the transformation matrix and the functions to handle the matrix. An effect node could carry the shader information while a **modelData** node could carry the model's vertex information. Then when an object is constructed, all these different nodes are put together to describe the object. These nodes are put in a tree fashion, so they can also share their information with other nodes. This is the most common scenegraph design.

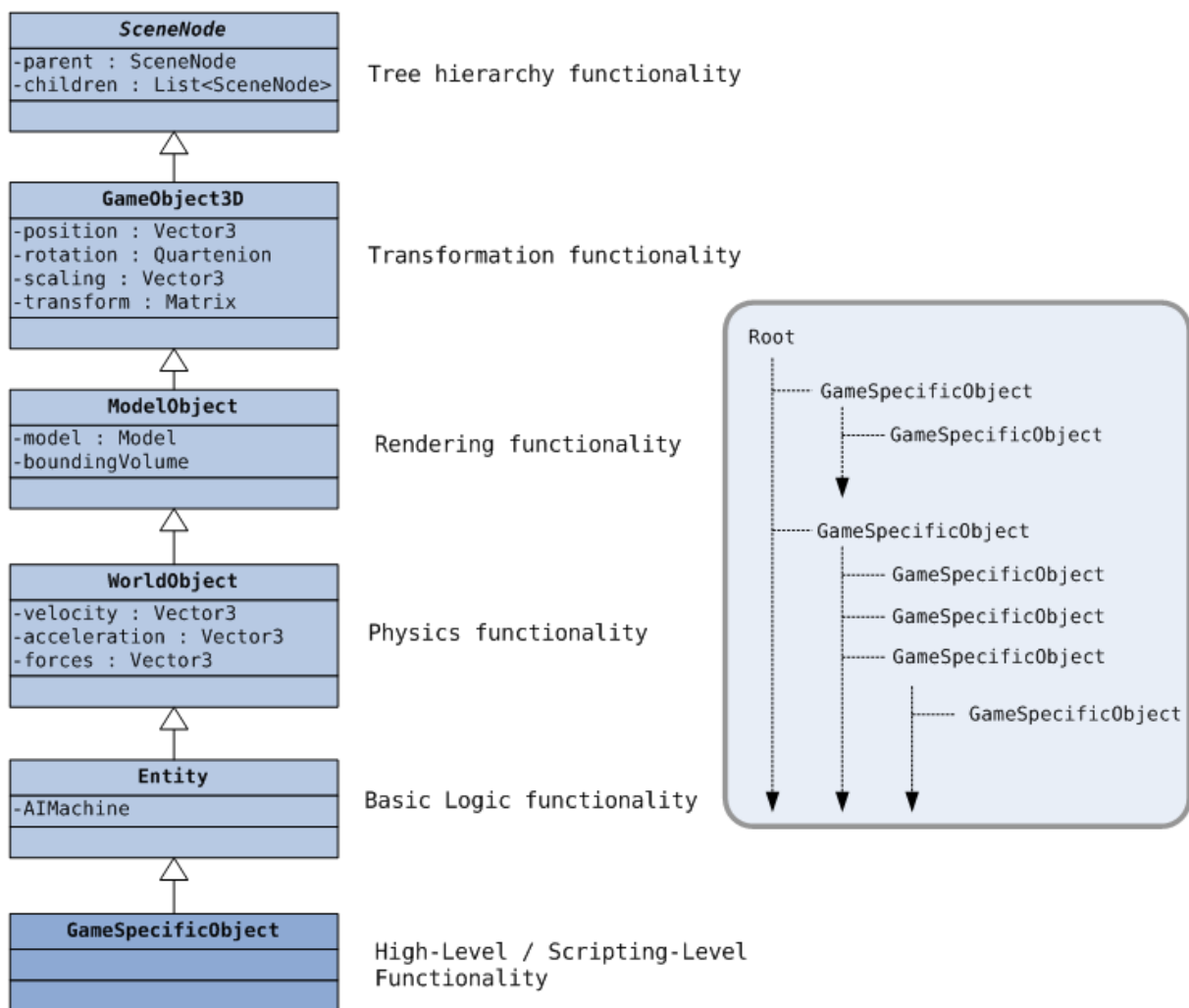


In the above example, rendering is performed by a special function called a traverser. The rendering traverser, starts from the root and keeps going deep into the tree, storing the needed information (like the

transformation). When it finds a **modelData** node, it will use that information to render the model. Then it will go back one step and resume with the rest of the branch's nodes. That goes on till the root has no other children nodes. The physics traverser will do the same thing but this time it will apply its information when it reaches a **physicsData** node.

## 2.2 HIERARCHICAL REPRESENTATION

Another way to represent the same data (and the one I'm going to extensively present in this article) is the hierarchical one. In this case there is only one kind of node, containing all possible information. This is not just a single class though. Instead, a new class is derived every time additional functionality is required. Starting from the basic parent/children relationship functionality, the node is extended into containing all the information to fully represent a game object.



At the above example, all game's objects actually inherit from class Entity. If for example you want to create a class named "Hero" at the highest possible level, this (or a lower-level class) should inherit from class Entity. Then an instance of node "Hero" is inserted into the scenegraph. Any children of "Hero" will share its data. When the Rendering function is run, it will call a virtual rendering function for every child of the Root, then for every child of the children and so on.

## 3 THE HIERARCHY

In the hierarchical representation, there are five basic functionalities that should be adapted. These are:

- **Tree hierarchy:** Each of the nodes must have a parent (with the root being the exception) and may have many children.
- **Transformation:** Each object has a relative position, orientation and scaling to its parent. All this information can be easily represented by a single transformation matrix. In this case two matrices are needed: One that contains only the local transformation and one that contains the total transformation up to the current node.
- **Rendering:** Contains the information needed to render the model like the vertex data and the shader.
- **Physics:** Each object is part of the world physics simulation. All the physics data like the object's velocity and acceleration are in here. Collision might also handled here.
- **AI:** For the game objects to have any use, there must be something controlling them.

### 3.1 TREE HIERARCHY: SCENENODE

First of all each node needs to have a name. This will come very handy for debugging reasons or if you decide to make an editor. Giving a name to every node is a good way to keep things clear in your mind. This could also be used to have access to any node, whenever you want it but what if two nodes have the same name? For internal node identification we will use an unsigned 64-bit integer (ulong). Each node created will have a unique number assigned to it. This way we can also know how many nodes were created at any time.

```
private string m_name;
private ulong m_id;
private static ulong m_sNumObjects = 0;

/// <summary>Gets the unique ID of the node.</summary>
public ulong ID { get { return m_id; } }
/// <summary>Gets or sets the name of the node.</summary>
public string Name { get { return m_name; } set { m_name = value; } }
/// <summary>Gets the total number of created objects.</summary>
public static ulong NumObjects { get { return m_sNumObjects; } }

// constructor
public SceneNode()
{
    // generate unique id
    m_id = m_sNumObjects;
    m_sNumObjects++;

    // generate automatic name
    m_name = String.Format("Obj{0}_{1}", m_id.ToString(), GetType().Name.ToString());
}
```

The main functionality of the **SceneNode** class though is the tree hierarchy. Each **SceneNode** instance must have a parent which is easily done. Then each node can have many children. A list of **SceneNode** references will be maintained for that reason. The function **AddChild()** will be used as an interface to add children nodes to a node. Of course many other functions can be added here for extra access, if required.

```

private SceneNode m_parent = null;
private List<SceneNode> m_children = new List<SceneNode>();

/// <summary>Gets or sets the parent of this node.</summary>
public SceneNode Parent { get { return m_parent; } protected set { m_parent = value; } }

/// <summary>Adds a node to this node if it isn't added already.</summary>
/// <param name="node">The node to add.</param>
/// <returns>Whether the node was added.</returns>
public virtual bool AddChild(SceneNode node)
{
    // Check if the node is already added
    if (!m_children.Contains(node))
    {
        // add the node and set that this node is its parent.
        m_children.Add(node);
        node.m_parent = this;
        return true;
    }
    return false;
}

```

To actually get the tree hierarchy structure really applied you will need some functions that when run, will also call the respective function for every child node. These functions are usually virtual functions that go all up to the hierarchy like **OnRender()** or **OnUpdate()**. If you're using XNA, you will probably make your **SceneNode** class inheriting from **DrawableGameComponent**. In any other case you could make some interfaces with virtual functions like **IRenderable** and **IUpdate** and make **SceneNode** inherit from these.

```

/// <summary>Renders this node and all its children.</summary>
public virtual void Draw()
{
    foreach (SceneNode node in m_children) node.Draw();
}

/// <summary>Updates this node and all its children.</summary>
public virtual void Update()
{
    foreach (SceneNode node in m_children) node.Update();
}

```

## 3.2 TRANSFORMATION: GAMEOBJECT3D

All objects in the world will need to have a position, an orientation and a scaling factor. This information combined is called the object's transformation and will be used by almost every function in the game including the rendering, physics and AI. An easy way to represent it is a 4x4 matrix (for 3D). As already described in the introduction, in many cases a child should also have its parent's transformation applied, along with its own. To describe this I use the term "combined transformation", where in the case of a single object's transformation I will use the term "own transformation". Applying the parent's transformation on its children is easy: you just multiply the matrices.

In a 3D world the position of an object can be described using three numbers, one for each axis. This structure is called a Vector and in the case of 3D we will call it Vector3 (as in XNA). All it contains is three float numbers called X, Y and Z representing the distance from the center of the parent, for each respective axis. In the same manner, scaling will also be described by a Vector3 structure, containing three numbers that represent how much the object should be scaled at each axis, relative to its original size.

As for the rotation, we could use a Vector3 as well to describe how much the object is rotated on each axis but there is a catch here. When rotating an object on many axes at a large amount, there might be some unwanted effects. The quaternion is a rotation friendly structure and comes as a solution to this. Position and scaling are actually set at every frame and new matrices are created to hold that data. The quaternion though doesn't work that way. Instead a quaternion is maintained and for every amount of rotation a new quaternion is produced that it's then multiplied with the one holding the total rotation. For that reason the **Rotate()** function is used to rotate an object. Alternatively a new Quaternion containing the required rotation can be created and multiplied with the original one, via the Rotation property.

```
private Vector3 m_position = Vector3.Zero;
private Vector3 m_scaling = new Vector3(1, 1, 1);
private Quaternion m_rotation = Quaternion.Identity;
private Matrix m_combinedTransform = Matrix.Identity;
private Matrix m_ownTransform = Matrix.Identity;

/// <summary>Gets or sets the position of the object.</summary>
public Vector3 Position { get { return m_position; } set { m_position=value; } }
/// <summary>Gets or sets the rotation of the object.</summary>
public Quaternion Rotation { get { return m_rotation; } set { m_rotation
= value; } }
/// <summary>Gets or sets the scaling of the object.</summary>
public Vector3 Scaling { get { return m_scaling; } set { m_scaling=value; } }

/// <summary>The combined matrix from the root all down to this node.</summary>
public Matrix CombinedTransform
{
    get { return m_combinedTransform; }
    protected set { m_combinedTransform = value; }
}

/// <summary>The transformation matrix of this node.</summary>
public Matrix OwnTransform
{
    get { return m_ownTransform; }
    protected set { m_ownTransform = value; }
}

/// <summary>Rotates the object on the selected axis.</summary>
/// <param name="axis">The axis that the object is rotated at.</param>
/// <param name="angle">The amount to rotate the object in radians.</param>
public void Rotate(Vector3 axis, float angle)
{
    m_rotation *= Quaternion.CreateFromAxisAngle(axis, angle);
}
```

Next we will have to create the transformation matrices. This can be done via the update function that is run at every frame. More complex designs can actually check if the object's transformation has changed since the last update and will only perform the matrices computations when it's required. For the sake of keeping this as simple as possible and avoiding expanding on other topics, I will just update the matrices at every frame.

Updating the matrices may vary from API to API. In the case of XNA, the Matrix structure contains functions that let us create the required matrices. The right order to multiply the matrices is first scaling then rotation and then position. Then the object's transformation is multiplied with its parent's one, to produce the combined transformation.

```

public override void Update()
{
    // Create the current transform matrix
    OwnTransform = Matrix.CreateScale(m_scaling) *
        Matrix.CreateFromQuaternion(m_rotation) *
        Matrix.CreateTranslation(m_position);

    // Create the combined matrix
    if (Parent != null)
    {
        CombinedTransform = OwnTransform *
            (Parent as GameObject3D).CombinedTransform;
    }

    // update base
    base.Update();
}

```

Mind that **GameObject3D** is a very important step in the node hierarchy. Grouped objects that don't need to have mesh data, like buildings and zones should inherit from here. Examples are provided in section 4.

### 3.3 RENDERING: MODELNODE

The next expansion of the node will be towards model data. Each object that needs to be rendered, it has to contain some mesh data, along with information on how to render that data. Collision information can also be saved here since it will get generated from the mesh data. Depending on your collision detection model, there might be a separated structure that holds the collision data or even another step in the node hierarchy.

In this example we are using the XNA's Model structure which is a convenient way of holding all model's data, including mesh transformations, vertex data and textures. If you are using another API, just use the required structures to load your data. The graphics data is always loaded after the graphics device is created. Usually **OnCreateDevice()** function is virtual and part of the **IRenderable** interface. The equivalent in XNA is the **LoadGraphicsContent()** function in the **DrawableGameComponent** class. I'm also using a static reference of XNA's **ContentManager** here to load the model. You may do that as you wish. If for example it was native DirectX, many functions would be used here or you could just make your own content manager.

```

private Model m_model = null;
private string m_assetName = null;

// constructor
public ModelRenderer(string assetName)
{
    m_assetName = assetName;
}

// called when the graphics device is created
public void OnCreateDevice()
{
    // mesh data along with material data is loaded
    m_model = GameCore.ContentManager.Load<Model>(m_assetName);
}

```

Another thing we should take care of is the collision data generation. This data will be generated here to use by the next structure in the tree, which is the **WorldModel**. Collision data can be generated in the **OnCreateDevice()** function, after the data was loaded. In any case this may vary a lot depending on the collision model you have chosen, so I am not going to present anything for collision here.

The object's drawing is actually happening in this class. In the virtual **Draw()** function you should do all the drawing work. In the XNA's case, all data is hold on the Model structure including the shader data. Again I'm not going to present anything about drawing here, since it might vary a lot from API to API but the idea is same: Just draw the model on the screen using the API's functions.

Note that the Model data can be any kind of mesh data container, including animated and skinned meshes. In this case, the hierarchy system doesn't have to change. Again, if you find that the collision system along with the mesh data system it's too much for one class, you might just separate these functionalities and add another class after this and before the world model.

### 3.4 PHYSICS: WORLDMODEL

As the mesh data model above, the physics model can vary a lot depending on what system you're using. The example I'm going to present here is just for the purposes of this article and doesn't represent a full physics model. Instead a very simple physics model is used, with forces applied and having the acceleration and velocity applied on the object. This is done via the **ApplyForce()** function that can externally used by the game's logic.

During the update, the acceleration is calculated and it's then added to the total velocity. Next we check for collisions. I just use a hypothetical **CollisionData** structure that is supposed to return all the required data after the collision check. The collision test logic is supposed to be handled inside the **CheckCollision()** function. When the **CollisionData** structure is returned, we can check for various aspects and perform the required alterations on the object's velocity and acceleration. Finally the new object's position is calculated.

```
private Vector3 m sumForces = Vector3.Zero;
private Vector3 m linearAcceleration = Vector3.Zero;
private Vector3 m linearVelocity = Vector3.Zero;
private float m_mass;

/// <summary>Apply a force to this object.</summary>
/// <param name="force">The force to apply.</param>
public void ApplyForce(Vector3 force)
{
    m_sumForces += force;
}

public override void Update()
{
    // get the elapsed time since the last update
    float elapsedTime = Timers.MainTimer.Elapsed;
    // acceleration = force / mass
    m linearAcceleration = m sumForces / m mass;
    // velocity = acceleration * time
    m linearVelocity += m linearAcceleration * elapsedTime;

    // perform collision test
    CollisionData data = null;
    CheckCollision(out data);

    // check here the returned CollisionData structure
    // and perform the required changes...

    // distance = velocity * time
    Position += m linearVelocity * elapsedTime;

    base.Update(gameTime);
}

/// <summary>Check if this object collides with another one.</summary>
/// <param name="data">The collision data.</param>
private void CheckCollision(out CollisionData data)
{
    // collision check here
}
```

You may expand the physics model as much as you want. In this hierarchy example though, as we will see in section 4, we are also going to expand the Root into being the World. World then applies gravity to all objects. You may decide to complete move all physics to the World object instead and just keep the variables in the **WorldModel** class. In that case you should add the required properties and functions so the World object can get and set the physics data on a **WorldModel**.

### 3.5 AI: ENTITY

The final step to complete the basic node hierarchy is the logic. Every object in the game world will be static and dull except some forces are applied on it. This class will serve as an interface to the inheriting classes so the game specific logic will be universal to all game entities. Both player and computer controlled entities will use this interface to perform their actions inside the world. No matter what kind of AI you're going to use, there has to be a logic control that will have a generic interface. This class will simply hold that control. This generic AI interface will become specific later on the game specific classes. When a game engine is made though, it's a good idea to offer a generic finite state machine, inside the basic level to start with.

```
private ILogicControl m_control = null;
/// <summary>The entity's logic controller</summary>
public ILogicControl Control { get { return m_control; } set { m_control = value; } }

public override void Update()
{
    // Update the logic
    if (m_control != null) m_control.Update();
    base.Update();
}
```

This is the clearest way possible to add AI capabilities to the game object. In the case of a finite state machine, the high-level programmer will just have to write the entity's specific states or a state hierarchy.

## 4 THE OBJECTS

Although the basic structure is created, we still need some generic components to make things work and offer the high-level programmer ways to easily access and handle the game objects. The root of the scene graph will be a specialized node that carries much more information than a typical object. This will also get expanded to accommodate generic physics and other functions. Then some specialized nodes like buildings, terrains and games zones are introduced. These objects don't follow the classic node system introduced above and they need to inherit from somewhere between. Finally we will get to the game level and present some examples on how the hierarchy can be expanded.

### 4.1 THE WORLD

The scene graph root is different to a simple root in that contains additional specific data. Such critical data is the **Camera**. Rendering starts from the root, so the generic rendering information such as the area to be rendered should be specified from here. In the example presented below, the **SceneRoot** class inherits from **GameObject3D** and contains a list with cameras. Then the programmers can select the desired camera and all the objects will be rendered (and culled) based on that information.

```
public abstract class SceneRoot : GameObject3D
{
    private List<Camera> m_cameras = new List<Camera>();
    private Camera m_activeCamera = null;

    /// <summary>Adds a new camera object to the scene.</summary>
    /// <param name="camera">The camera to add.</param>
    /// <returns>Whether the camera was added.</returns>
    public bool AddCamera(Camera camera)
    {
        // add the camera if it's not already there
        if (!m_cameras.Contains(camera))
        {
            m_cameras.Add(camera);
            // if this is the first camera, set it the active camera
            if (m_cameras.Count == 1)
                ActiveCamera = camera;
            return true;
        }
        return false;
    }

    /// <summary>Gets a camera from the camera list.</summary>
    /// <param name="camera">The camera to get.</param>
    public Camera GetCamera(string name)
    {
        foreach (Camera cam in m_cameras)
        {
            if (cam.Name == name)
                return cam;
        }

        // if the specified camera is not found, return the active one
        return m_activeCamera;
    }

    /// <summary>Sets the camera that the rendered will use to render this scene.</summary>
    public Camera ActiveCamera
    {
        get { return m_activeCamera; }
        set
        {
            if (m_cameras.Contains(value))
                m_activeCamera = value;
        }
    }
}
```

```

    }
}

public override void Update()
{
    // first update the camera and then continue with the scene graph
    m_activeCamera.Update();
    base.Update();
}
}

```

For this system to work the camera class should calculate its view and projection matrices inside the **Update()** function like the example below.

```

public class Camera
{
    // Camera data here

    public void Update()
    {
        // calculate camera's view & projection matrices
        m_view = Matrix.CreateLookAt(m_position, m_target, m_upVector);
        m_projection = Matrix.CreatePerspectiveFieldOfView(
            (float)Math.PI / 4.0f, m_aspectRatio, m_nearPlane, m_farPlane);
    }
}

```

The next step is to further expand the **SceneRoot** class to include physics functionality. Depending on your physics system, this class could contain from nothing to a whole physics simulator, processing each of the scene graph's objects. Since this is a very implementation specific class I won't provide any examples here.

Mind that the whole idea is to make the **Root** accessible to every object so each object can then have access to the current camera and other similar global information. This can be done through your game core's class. You just have to make sure that the world object will be created and set as the current world before you reference it from anywhere else in your project.

```

public class GameCore
{
    // the scene root
    private static World m_rootWorld;
    public static World CurrentWorld { get { return m_rootWorld; } }

    // the static reference to the current running game core
    public static GameCore Core = null;

    // constructor
    public GameCore()
    {
        Core = this;
    }
}

// from anywhere in the project
Vector3 camPosition = GameCore.CurrentWorld.ActiveCamera.Position;

```

## 4.2 BUILDINGS

Buildings in a game world are created from combining many objects together. These objects can be wall blocks, furniture and any other kind of objects you would find in a building. Therefore, there is no need for the Building node to carry any model data information. Still a building carries some transformation information and that's why it should derive from **GameObject3D**.

The building data should be loaded from an external source. Here we just use a XML file but that can be really anything, even hard coded data. Normally this data will come from an editor.

```
public class Building : GameObject3D
{
    // constructor
    public Building(string buildingResource)
    {
        LoadBuildingFromXML(buildingResource);
    }

    private void LoadBuildingFromXML(string buildingResource)
    {
        // load the data from the XML file
        // and create the required objects
    }
}
```

The building node can be further expanded to perform more functions such as special culling when the player is in the building.

## 4.3 TERRAIN

The Terrain node can vary a lot, depending on the terrain system you are using. If your terrain is rather simple and you just import it as a model, it could even inherit from **ModelNode**. This is not suggested though. A better approach is to inherit from **GameObject3D** and use the Terrain node to load terrain data from a more specialized format that will allow you to load and calculate all the required information that will come handy later when manipulating the terrain.

## 4.4 ZONES

Zones are invisible objects that do not carry any model data. These objects are useful to the high-level programmers for specifying where an action should start or end. Creating the **Zone** node is simply achieved by inheriting from **GameObject3D**. Then code specific to zone types can be added. An example would be an audio zone, that when the player enters it, a specific music track will be played. Same can be done with sound effects.

Zones can also come handy when writing game logic. You can specify limits that entities can move within or special collision zones in the case you decide your player should enter some specific region. An additional expansion of the Zone node can accommodate the door/key functionality, where a region is locked until the player performs some action, like hitting a button or killing some monsters. Zones can be of any shape, like bounding spheres, boxes or more specialized real-time defined shapes.

## 4.5 GAME-LEVEL OBJECTS

The whole hierarchy was created so when we reach this level (the actual game level) we will be able to have a clear view of how to create the actual gameplay objects. If you need to add some extra information common to all your game entities and specific to your game this is probably the best place to do so. You can make a class inheriting from class Entity to store that information. Then at the highest level, each game object will finally inherit from that class. Here is just a simple example on how that can be done.

```
public abstract class GameEntity : Entity
{
    protected bool m isFriend;
    protected uint m hitPoints;
    protected uint m maxhitPoints;
}

public class Hero : GameEntity
{
    // handle player logic here, like input
}

public class GameMonster : GameEntity
{
    // handle generic monster data here
}

public class FlyingMonster : GameMonster
{
    // handle specific logic here
}
```

Mind that this is an example in C# which is good even as a scripting language, as it's friendly enough for high level programmers. In the case you're adapting this system in C++ I would suggest the highest level presented here to be inputted from a scripting language or C#.

Usually the hierarchy up to the Entity class would be inside the game engine, were the above presented objects would reside to a different project that would adapt the functionality of the engine. That allows you to easily reuse the basic engine's functionality at your future projects too, without having to clear any code from your previous games. In another scenario you could create a middle level project that will accommodate high level information common to all your games.

## 5 CONCLUSION

The above described hierarchy might not be the most efficient or best looking structure ever presented. Throughout my research though I found it to be easy to handle, maintain and debug. Furthermore any new functionality can be easily added over the existing one. If you are careful on the initial design so that no changes are required later in the first few steps of the hierarchy, everything will just bind nicely up to the highest level. In any case, I suggest you experiment a bit at the beginning with different class hierarchies till they fit your needs. That can be another class between the hierarchy that only handles the collision or anything else you might find more compatible with your project.

The above design can have build-in multi-threading support, just make sure that you add the required lock (this) {} statements (if in C#) and that you follow the general multi-threading rules. From my own experience I found out that keeping both **Draw()** and **Update()** on the same thread and making a totally another thread just for physics works great. In that case the **Update()** function will contain much less code. Creating a totally another thread or combining the logic on the physics thread is not a bad idea too. Just mind that multi-threading can lead into debug nightmares and sometimes is very hard to accurately predict the behaviour of your code.

In case you're using C# for your project, the most important fact is that the high level and logic programmers will find the whole structure very friendly since they will be able to see only what they need to and they won't have to get involved with low level details. This allows them to be more creative. This is also a level editor friendly structure in terms that the level designers can think of actual game objects as simple programming objects and not as some complicated "mystical" structures. Everything is crystal clear and therefore changes can be performed without having to mess with the internals.