# LumberJack

Dan Cuccia

Will Custode

# Technical Design Document

# LumberJack Master UML File

*(if you are viewing this in MS World, double click to expand and explore)*



# Renderer

**Class**: Renderer

**Name Space**: Managers

**Description**: The Renderer class manages all aspects of rendering in 3D space, using custom HLSL shading techniques, multiple render targets, advanced lighting techniques, and advanced imposter drawing techniques to create the unique cartoon look of the environment. This class is a singleton, therefore only one instance is ever allowed to exist, and is easily accessible anywhere within the project's solution. To achieve silhouette's, the world is drawn in its entirety twice; once for all diffuse colors including lighting, and a second time which draws each pixel's normal into a new texture. This second texture is then used in conjunction with the *Sobel Filter* to create dynamic silhouettes around and within a model.



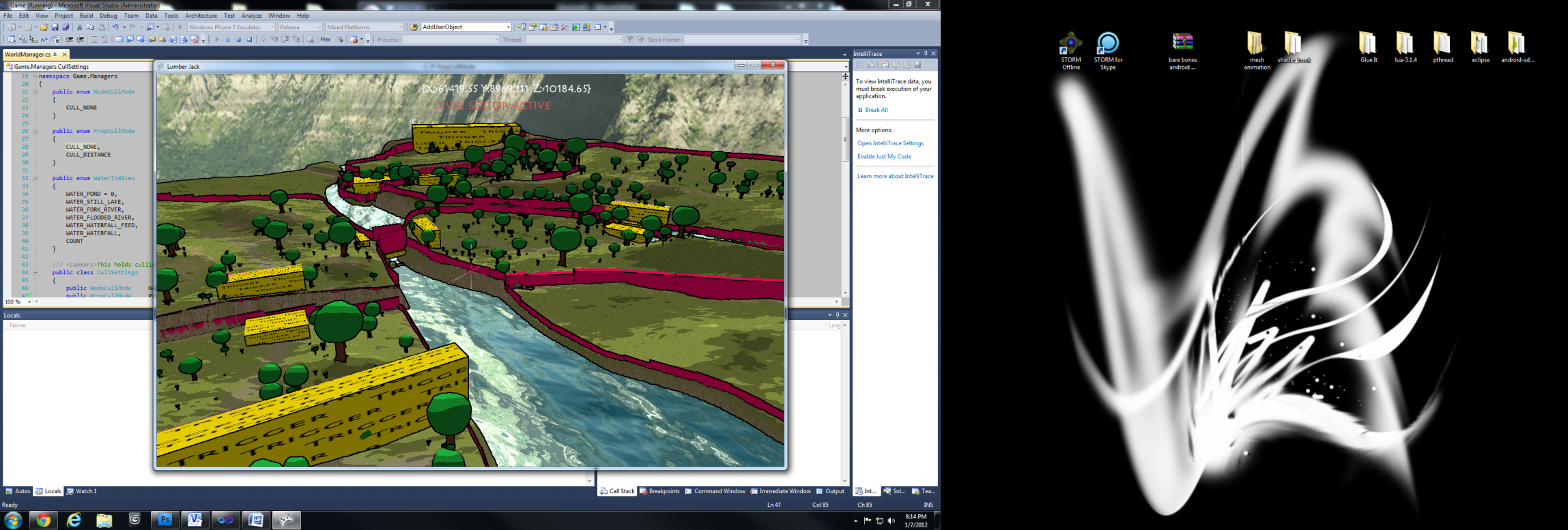
|  |  |
| --- | --- |
| C:\Users\Coosh\Desktop\CONTEST\run-time shots\diffuse.jpg | **Diffuse Pass - Draw list**   * *Skybox* * *Node terrain* * *World water* * *Props (distance culled)* * *Debug*   + *WorldBorders*   + *Triggers*   + *OrientedBoundingBox* |
| C:\Users\Coosh\Desktop\CONTEST\run-time shots\normals.jpg | **Normal Pass - Draw list**   * *Node terrain* * *World water* * *Props (distance culled)* * *Debug*   + *worldBorders*   + *Triggers* |
| C:\Users\Coosh\Desktop\CONTEST\run-time shots\depth.jpg | **Depth Pass**   * *Drawn into the alpha layer of the Normal&Depth Render Target* |
| C:\Users\Coosh\Desktop\CONTEST\run-time shots\silhouettePass.jpg | **Silhouette Pass**   * *Using the Normal Pass, and the Depth Pass, create dynamic silhouettes using Sobel Filtering Techniques* |
| C:\Users\Coosh\Desktop\CONTEST\run-time shots\defaultRender.jpg | **Composition Pass**   * *Combine the Silhouette Pass onto the diffuse pass for this final render, which is presented to the back buffer* |

# World Manager

**Class**: WorldManager

**Name Space**: Managers

**Description**: The world manager is a robust class that manages all aspects and functionality of the entire environment. It contains all of the master lists, and the 2D array of *WorldNodes.* All master culling logic is held within this manager's main draw3D call, which iterates and calls all of the composed *WorldNodes* within its 2D array. For easy accessibility throughout the entire solution, this class is a singleton, which also means there will only ever be one instance of this class created.



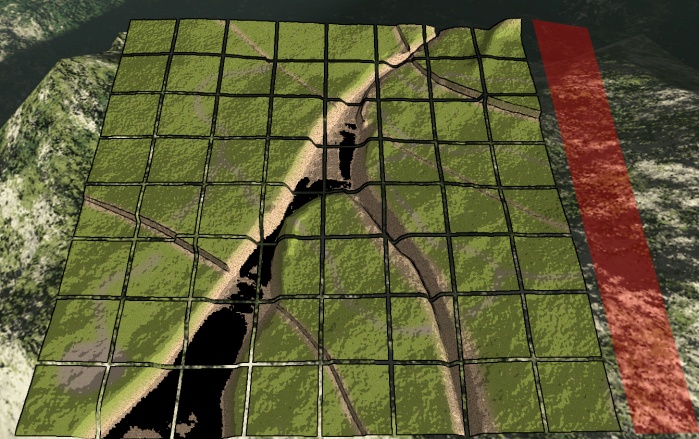


# WorldNode

**Class**: WorldNode

**Name Space**: Game Objects

**Description**: Each world node is a piece of a grid. This grid of world nodes is the entire environment (*managed by WorldManager)*. Each node contains its own plot of land, and list of props that is comprised of trees, rocks, and world boundaries.





# Camera

**Class**: Camera / BuildCamera

**Name Space**: Managers

**Description**: Camera class encapsulates all functionality for viewing the world. There are four main *CameraTypes* that the developer may use. There is also a slim camera version that may be used called the*BuildCamera*, this version of a camera provides the basic functionality needed to view the world, but holds much less member variables. The *BuildCamera* was outside of the main world's environment, specifically inside of tablet menu states.

**CAM\_CHASE**: Update's the camera to follow a position in space. Holding the middle mouse button will rotate the camera around the chased position, scrolling the mouse wheel will zoom the camera in and out.

**CAM\_FREE**: First Person flying mode. WASD control movement, QE controls height separately, spacebar gives a x2 speed boost for faster travel, and the mouse will rotate the camera accordingly.

**CAM\_STATIONARY**: No movement or input takes affect when the camera is in this mode.

**CAM\_AUTOMATED**: Calling *SmoothStepTo(position, threshold)* will automate the camera from its current location to the input location, using quadratic interpolation for smooth movement. Due to the fact that this interpolation will technically never reach the exact destination, a world-space distance threshold takes effect, once the camera's position is less than this threshold, the *AutomationCompleteCallback, (*which is a delegate function pointer)takes effect.



# Generated Terrain

**Class**: GeneratedTerrain

**Name Space**: Drawing Objects

**Description**: This class handles drawing and the creation of terrain using a bitmap image. Each RGBA pixel in the bitmap is used to determine 4 things of each vertex in the terrain. *RED*: path texture weight, *GREEN*: grass texture weight, *BLUE*: rock texture weight, *ALPHA*: vertex height. Using the Content Extension Pipeline provided with the XNA framework, all images are converted to models during the final compilation of the game, and not during runtime. This drastically cuts down loading times for the player.





# Particle Emission

**Class** : BillboardParticleEmitter

**Name Space**: Drawing Objects

Description: Billboard particle emission was built with optimization and speed in mind. Each particle does not contain its own drawing object for display, instead only the data needed to display. This emitter may use both 3D models or animated sprite billboards in 3D space to display. Vertices are dynamically created depending on each particle in the emitter's list, or a single model iterates through the particles and draw's their data.

Many display and behavior parameters are found in the composed *EmitterSettings* class.

**MaxParticles**: The maximum amount of particles this emitter may have on any cycle.

**MaxDistance:** The maximum distance a particle may travel away from the emitter's position.

**MaxLife:** The maximum milliseconds each particle may exist for.

**Spread:** Random modulator of initial velocity, more spread will scatter each particle's initial velocity around the emitter's vector.

**SpawnRadius**: The box or sphere in which all particles will emit from.

**Frequency**: Time between particle spawns in milliseconds

**GrowForMillies / ShrinkForMillies**: adding to these variables will scale the object in and out of existance. If *GrowForMillies = 500* the first half second of the particles existence will be uniformly scaling from 0 to 1.

**ShowForMillies / FadeForMillies**: adding to these variables will fade the object in and out of existence, using the same algorithm as *Grow/Shrink* parameters.

**Speed**: The initial velocity in which all newly created particles receive, modulated by *Spread*.

**SpeedVariance**: This variable will randomize the initial speed in which newly created particles receive.





# Water Volume

**Class**: WaterVolume

**Name Space**: Drawing Objects

**Description**: This drawing object is used to display reflective water. Using advanced lighting techniques in Effect: *ToonMaster.fx* The plane that is displayed uses normal mapping, fresnal, and HDR elements to create this dynamic and cartoon-looking water. There are many parameters the developer may set.

**BumpHeight**: The 0 to 1 modulator in which the normal mapping will take effect.

**TextureScale**: How many times the normal map will tile.

**BumpSpeed**: The X/Y velocity in which the normal map will scroll.

**FresnalBias**: Changes the angle in which the normal map will be viewed at.

**FresnalPower**: How deep the normal map will be viewed.

**HDRMultiplier**: Scales the overall brightness of the reflecting light source.

**DeepColor**: The Color to be displayed in low parts of the normal map.

**ShallowColor**: The color to be displayed in high parts of the normal map.

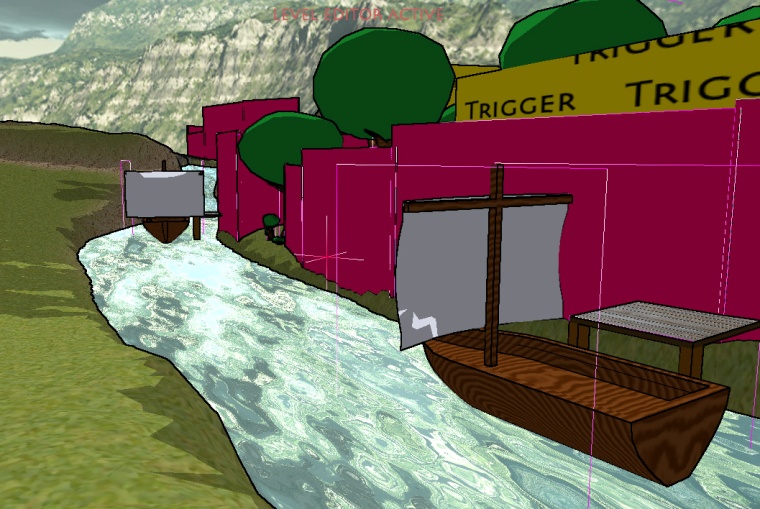
**ReflectiveColor**: multiplied against the reflecting skybox.

**ReflectionAmount**: The 0 to 1 scale in which the reflected skybox will be displayed.

**WaterAmount**: The higher this is, the less *DeepColor* will be displayed.

**WaveAmplitude**: The height of waves, 0 for no waves.

**WaveFrequency**: Velocity of each wave.





# Player

**Class**: Player

**Name Space**: Game Objects

**Description**: This is the main class that encapsulates all logic for the player. It contains an *AnimatedMesh* to display Andy, the main character, a *BoneControlledModel* to display Andy's axe when he chops down a tree, and a *StaticMesh* to display his tablet when the player opens up the game menu. The player class also contains all inventory related information, along with all XML serialization needed for saving. In order to optimize for speed and efficiency, an invisible *OrientedBoundingBox* is placed at Andy's feet, this box is used for all collision physics, and trigger activation. All user-input controls this box per cycle and at the end of the class's update, all drawing information updates according to the box's new parameters.



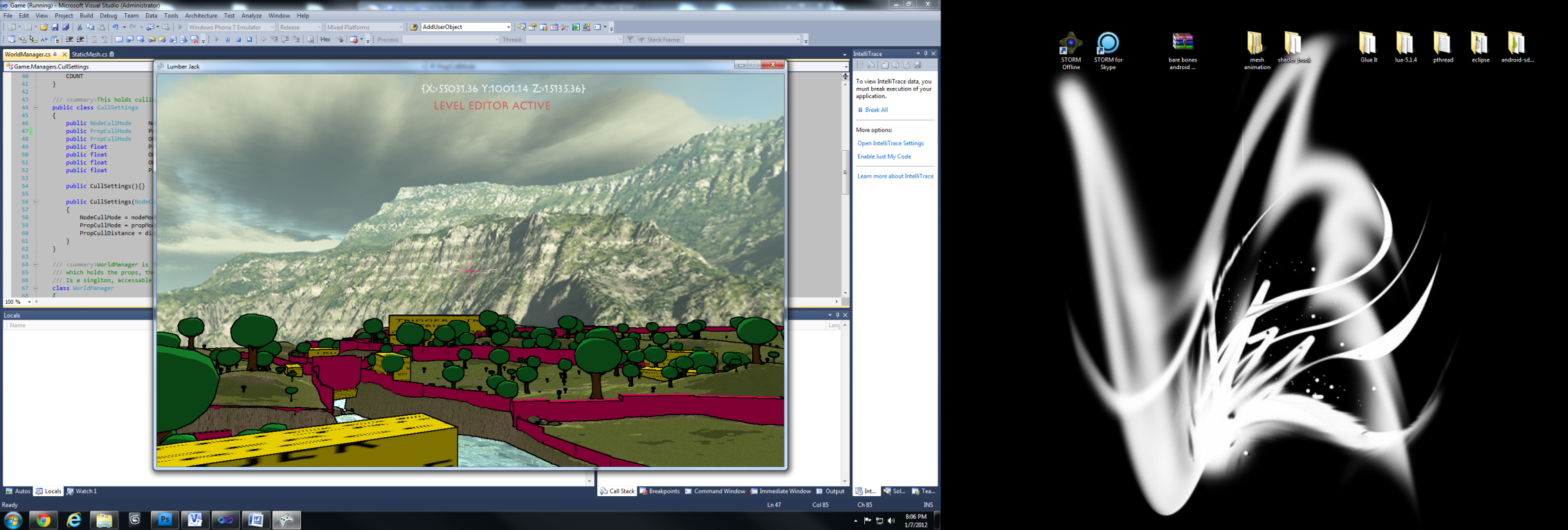


# SkyBox

**Class**: SkyBox

**Name Space**: Drawing Objects

**Description**: This drawing object takes care of all initialization and functionality to draw a skybox using a cubic texture. Using a technique in *ToonMaster.fx* this is drawn first to the diffuse backbuffer. The cubic texture is drawn onto a cube model, which no matter what is locked to the camera's position.





# Static Mesh

**Class**: StaticMesh

**Name Space**: Drawing Objects

**Description**: This class is the base class to draw 3D models that have no animation functionality, but instead all needed functionality to draw many different shading techniques in 3D space. Using 5 different Initialize overloads, the developer can easily create a mesh that draws differently with the *Renderer.*

**RT\_TOON**: Use a diffuse texture on the model, under cartoon lighting shader.

**RT\_WOOD**: Use dynamic wood shading, lit with cartoon light shading.

**RT\_DARKWOOD**: Use dynamic wood shading with dark colors, lit with cartoon light shading, meant for trees.

**RT\_COLOR**: draws the model with a solid color, lit with cartoon light shading

