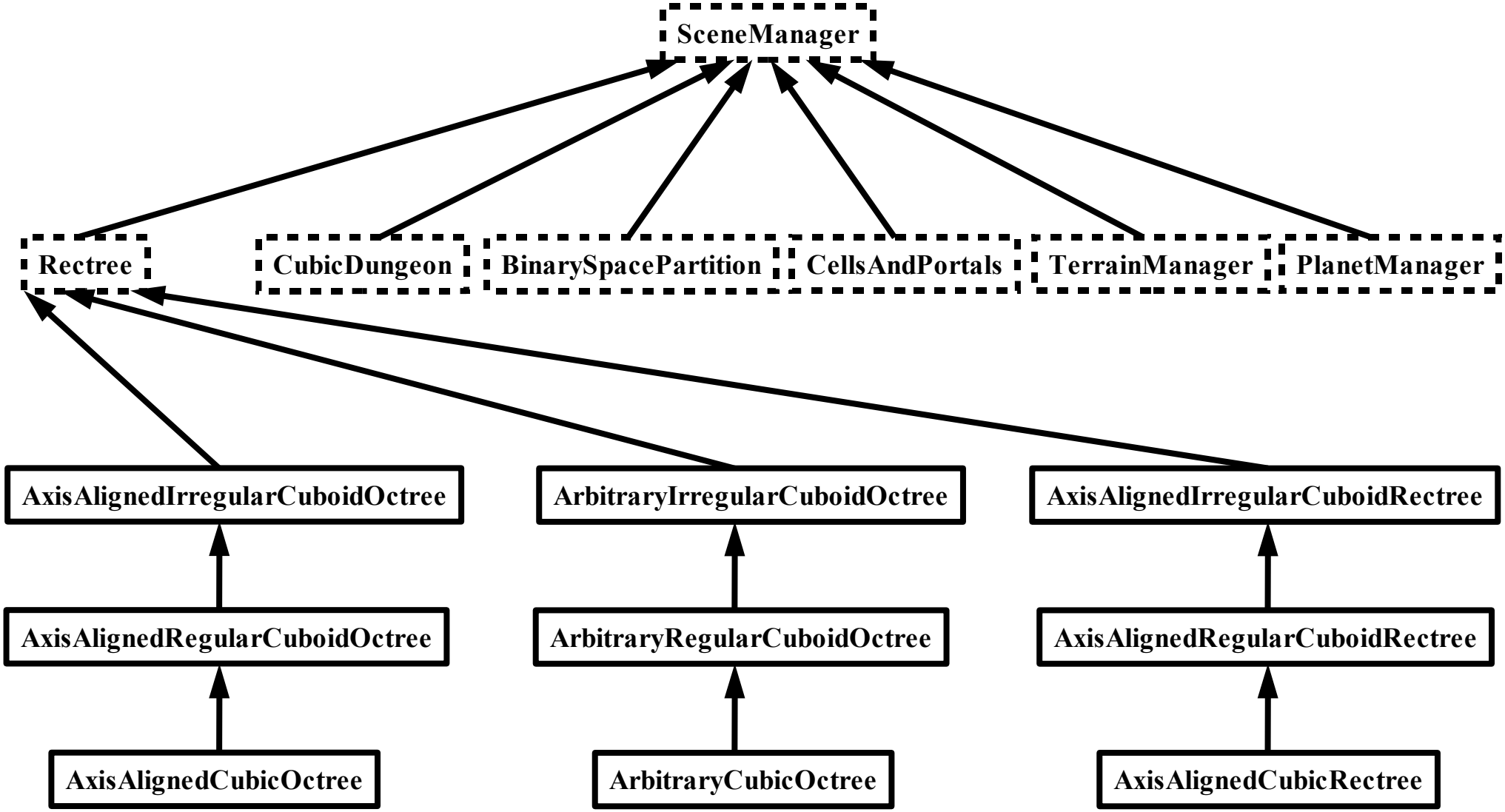
Table of Content

SceneManagers

Concept: SceneManagers are mainly used to perform culling and give hints for a ResourceManager (about necessary resources).



SceneManager

A SceneManager keeps track of a list of managed dynamic objects (e.g. animals, people, airplanes, cars, ...) and a list of managed static objects (e.g. rocks, trees, terrains, planets, ...). These two lists are managed in an associative memory (HashMap), for quick searches.

→ attributes:
DynamicObjects, **StaticObjects**

There are methods to insert and to remove objects – both return integer values indicating if the insertion or removal was successful (code "zero" means – everything was okay...).

→ methods:
InsertDynamicObject, **InsertDynamicObjects**, **RemoveDynamicObject**, **RemoveDynamicObjects**, **InsertStaticObject**, **InsertStaticObjects**, **RemoveStaticObject**, **RemoveStaticObjects**

Two other methods can be used to perform culling and deliver a list of visible objects that can be seen from the viewpoint of the virtual camera.

→ methods:
GetVisibleStaticObjects, **GetVisibleDynamicObjects**

In order to cascade visibility calculations of different SceneManagers (for instance to connect one outdoor-manager with one indoor-manager), there should also be a method that needs a list of objects as parameter and directly calculate the visibles (and return a new list of all now visible objects after having performed culling with this second SceneManager).

→ methods:
CheckVisibleStaticObjectsList, **CheckVisibleDynamicObjectsList**

Further helper methods are integrated, e.g. to refresh the SceneManager or to perform prefetching for a given zone or camera position. And there could be some methods that inform a ResourcesManager wich then loads / unloads textures, models and so on...

→ methods:
Refresh, **Prefetch**, **GetHintOnUnusedObjects**

Rectree

A Rectree is a tree consisting of many cuboids – each containing another Rectree and (as usual for a SceneManager) two lists for dynamic and static objects that are completely enclosed by the Rectree.

There are methods to insert and to remove objects – both return integer values indicating if the insertion or removal was successful (code "zero" means – everything was okay...).
If the bounding box of the object doesn't fit into a Rectree, the operation cannot be performed and an error code is returned (or an exception is thrown – whatever is better).

The bounds of the Rectree will not always be sufficient. In order to manage the extension of the Rectree there are two variables controling the behavior of the Rectree.

→ attributes:
RectreeExtensionControl, **AllowedRectreeExtensionTypes**

RectreeExtensionControl:

Control over the extensibility can be "manually" or "automatically". If the automatic control is selected, the Rectree will always try to resize itself (meaning: it will set a new root node and sort in itself – but this extension depends on the second attribute:

AllowedRectreeExtensionTypes:

Should be a list of different possible types, containing the following:

- "None": no extension possible – borders remain always unchanged
- "OctuplicateOnce": if necessary extend the Rectree in all 3 directions (x 8)
- "OctuplicateRepeatedly": octuplicate multiple times until it is enough
- "ExtendAndOctuplicate": add seven Rectrees (irregular cuboids!)
- "QuadruplicateOnce": can duplicate in 2 directions (x 4)
- "QuadruplicateRepeatedly": quadruplicate multiple times until it fits
- "ExtendAndQuadruplicate": add three Rectrees (irregular ones!)
- "DuplicateOnce": able to duplicate along one direction / axis (x 2)
- "DuplicateRepeatedly": duplicate multiple times until sufficient
- "ExtendAndDuplicate": add one new Rectree (with irregular cuboid!)

On the other hand it may be necessary to further subdivide the Rectree because too many objects are positioned inside the Rectree and the subtrees should be smaller.
There are three variables controling the subdivision behavior.

→ attributes:
RectreeSubdivisionControl, **AllowedRectreeSubdivisionTypes**, **MinSize**

RectreeSubdivisionControl:

Control over the divisibility can be "manually" or "automatically". If the automatic control is selected, the Rectree will always try to further divide itself (meaning: it will divide as necessary, but take into account the set MinSize.

AllowedRectreeSubdivisionTypes:

Should be a list of different possible types, containing the following:

- "None": no division possible
- "DivideIntoEightIrregularCuboids": IrregularCuboidOctree-like (x 8)
- "DivideIntoEightRegularCuboidsOnce": RegularCuboidOctree-like (x 8)
- "DivideIntoEightRegularCuboidsRepeatedly": RegularCuboidOctree-like
- "DivideIntoEightCubesOnce": CubicOctree-like (x 8)
- "DivideIntoEightCubesRepeatedly": CubicOctree-like

MinSize:

The MinSize has three sub-attributes:

- "MinSizeX": minimum size for sub-trees or super-trees
- "MinSizeY": minimum size for sub-trees or super-trees
- "MinSizeZ": minimum size for sub-trees or super-trees

If a Rectree doesn't contain any valuable information (is empty) it can request a merge with its neighbor Rectrees. If they are also empty, a merge operation can take place.
To control the merge operation behavior there are again some variables definig the actions:

→ attributes:

RectreeMergeControl

→ methods:

TryMerge

RectreeMergeControl:

Control over the merge processes can be "manually" or "automatically".
If the automatic control is selected, the Rectree will always try to merge.
If the manual control is selected, the merge process must be activated with a call to the method "TryMerge".

AxisAlignedIrregularCuboidOctree

This Octree is “axis aligned” – meaning that the cuboids are positioned parallel to the main world axes of the global coordinate system.

The contained sub-Octrees can be a fitting AxisAlignedIrregularCuboidOctree or a fitting AxisAlignedRegularCuboidOctree or a fitting AxisAlignedCubicOctree.

AllowedOctreeExtensionTypes:

- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Octree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough
- “ExtendAndOctuplicate”: add seven Octrees (irregular cuboids!)

AllowedOctreeSubdivisionTypes:

- “None”: no division possible
- “DivideIntoEightIrregularCuboids”: IrregularCuboidOctree-like (x 8)
- “DivideIntoEightRegularCuboidsOnce”: RegularCuboidOctree-like (x 8)
- “DivideIntoEightRegularCuboidsRepeatedly”: RegularCuboidOctree-like
- “DivideIntoEightCubesOnce”: only if the Octree is cubic (x 8)
- “DivideIntoEightCubesRepeatedly”: only if the Octree is cubic

MinSize:
The MinSize for each of the three axes can be different.

AxisAlignedRegularCuboidOctree

This Octree is “axis aligned” – meaning that the cuboids are positioned parallel to the main world axes of the global coordinate system.

The contained sub-Octrees can be a fitting AxisAlignedRegularCuboidOctree or a fitting AxisAlignedCubicOctree.

AllowedOctreeExtensionTypes:

- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Octree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough

AllowedOctreeSubdivisionTypes:

- “None”: no division possible
- “DivideIntoEightRegularCuboids”: RegularCuboidOctree-like (x 8)
- “DivideIntoEightRegularCuboidsRepeatedly”: RegularCuboidOctree-like
- “DivideIntoEightCubesOnce”: only if the Octree is cubic (x 8)
- “DivideIntoEightCubesRepeatedly”: only if the Octree is cubic

MinSize:
The MinSize for each of the three axes can be different.

AxisAlignedCubicOctree

This Octree is “axis aligned” – meaning that the cubes are positioned parallel to the main world axes of the global coordinate system.

The contained sub-Octrees can be a fitting AxisAlignedCubicOctree.

AllowedOctreeExtensionTypes:

- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Octree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough

AllowedOctreeSubdivisionTypes:

- “None”: no division possible
- “DivideIntoEightCubesOnce”: divide into eight CubicOctrees (x 8)
- “DivideIntoEightCubesRepeatedly”: divide several times if necessary

MinSize:
The MinSize for each of the three axes must be the same (it’s a cube).

ArbitraryIrregularCuboidOctree

This Octree is “arbitrary” – meaning that the cuboids are not necessarily positioned parallel to the main world axes of the global coordinate system – the Octree keeps track of the translation and rotation.

There are many possibilities to control the kind of transformation and the behavior of the Octree in case of a transformation change:

→ attributes:
PositionChangeAllowed, **RotationChangeAllowed**, **ChangeObjectsAsWell**
→ methods:
ChangePosition, **ChangeRotation**, **ChangeTransformation**

PositionChangeAllowed:
- “true”: a change of the position of the Octree is possible
- “false”: changes of the position are not allowed

RotationChangeAllowed:
- “true”: a change of the rotation of the Octree is possible
- “false”: changes of the rotation are not allowed

ChangeObjectsAsWell:
- “true”: the managed objects will be repositioned and reoriented too
- “falls”: only the octree changes (Refresh afterwards)

ChangePosition, ChangeRotation, ChangeTransformation:
These methods can be used to change the position and orientation of the Octree.

The contained sub-octrees can be a fitting ArbitraryRegularCuboidOctree or a fitting ArbitraryCubicOctree.

AllowedRectreeExtensionTypes:
- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Octree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough
- “ExtendAndOctuplicate”: add seven Octrees (irregular cuboids!)

AllowedRectreeSubdivisionTypes:
- “None”: no division possible
- “DivideIntoEightIrregularCuboids”: IrregularCuboidOctree-like (x 8)
- “DivideIntoEightRegularCuboidsOnce”: RegularCuboidOctree-like (x 8)
- “DivideIntoEightRegularCuboidsRepeatedly”: RegularCuboidOctree-like
- “DivideIntoEightCubesOnce”: only if the Octree is cubic (x 8)
- “DivideIntoEightCubesRepeatedly”: only if the Octree is cubic

MinSize:
The MinSize for each of the three axes can be different.

ArbitraryRegularCuboidOctree

This Octree is “arbitrary” – meaning that the cuboids are not necessarily positioned parallel to the main world axes of the global coordinate system – the Octree keeps track of the translation and rotation.

There are many possibilities to control the kind of transformation and the behavior of the Octree in case of a transformation change:

→ attributes:
PositionChangeAllowed, **RotationChangeAllowed**, **ChangeObjectsAsWell**
→ methods:
ChangePosition, **ChangeRotation**, **ChangeTransformation**

PositionChangeAllowed:
- “true”: a change of the position of the Octree is possible
- “false”: changes of the position are not allowed

RotationChangeAllowed:
- “true”: a change of the rotation of the Octree is possible
- “false”: changes of the rotation are not allowed

ChangeObjectsAsWell:
- “true”: the managed objects will be repositioned and reoriented too
- “falls”: only the octree changes (Refresh afterwards)

ChangePosition, ChangeRotation, ChangeTransformation:
These methods can be used to change the position and orientation of the Octree.

The contained sub-octrees can be a fitting ArbitraryRegularCuboidOctree or a fitting ArbitraryCubicOctree.

AllowedRectreeExtensionTypes:
- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Octree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough

AllowedRectreeSubdivisionTypes:
- “None”: no division possible
- “DivideIntoEightRegularCuboids”: RegularCuboidOctree-like (x 8)
- “DivideIntoEightRegularCuboidsRepeatedly”: RegularCuboidOctree-like
- “DivideIntoEightCubesOnce”: only if the Octree is cubic (x 8)
- “DivideIntoEightCubesRepeatedly”: only if the Octree is cubic

MinSize:
The MinSize for each of the three axes can be different.

ArbitraryCubicOctree

This Octree is “arbitrary” – meaning that the cubes are not necessarily positioned parallel to the main world axes of the global coordinate system – the Octree keeps track of the translation and rotation.

There are many possibilities to control the kind of transformation and the behavior of the Octree in case of a transformation change:

→ attributes:
PositionChangeAllowed, **RotationChangeAllowed**, **ChangeObjectsAsWell**
→ methods:
ChangePosition, **ChangeRotation**, **ChangeTransformation**

PositionChangeAllowed:
- “true”: a change of the position of the Octree is possible
- “false”: changes of the position are not allowed

RotationChangeAllowed:
- “true”: a change of the rotation of the Octree is possible
- “false”: changes of the rotation are not allowed

ChangeObjectsAsWell:
- “true”: the managed objects will be repositioned and reoriented too
- “falls”: only the octree changes (Refresh afterwards)

ChangePosition, ChangeRotation, ChangeTransformation:
These methods can be used to change the position and orientation of the Octree.

The contained sub-octrees can be a fitting ArbitraryRegularCuboidOctree or a fitting ArbitraryCubicOctree.

AllowedRectreeExtensionTypes:
- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Octree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough

AllowedRectreeSubdivisionTypes:
- “None”: no division possible
- “DivideIntoEightCubesOnce”: divide into eight CubicOctrees (x 8)
- “DivideIntoEightCubesRepeatedly”: divide several times if necessary

MinSize:
The MinSize for each of the three axes must be the same (it’s a cube).

AxisAlignedIrregularCuboidRectree

This Rectree is “axis aligned” – meaning that the cuboids are positioned parallel to the main world axes of the global coordinate system.

The contained sub-Rectrees can be a fitting AxisAlignedIrregularCuboidRectree or a fitting AxisAlignedRegularCuboidRectree or a fitting AxisAlignedCubicRectree.

AllowedRectreeExtensionTypes:

- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Rectree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough
- “ExtendAndOctuplicate”: add seven Rectrees (irregular cuboids!)
- “QuadruplicateOnce”: can duplicate in 2 directions (x 4)
- “QuadruplicateRepeatedly”: quadruplicate multiple times until it fits
- “ExtendAndQuadruplicate”: add three Rectrees (irregular ones!)
- “DuplicateOnce”: able to duplicate along one direction / axis (x 2)
- “DuplicateRepeatedly”: duplicate multiple times until sufficient
- “ExtendAndDuplicate”: add one new Rectree (with irregular cuboid!)

AllowedRectreeSubdivisionTypes:

- “None”: no division possible
- “DivideIntoEightIrregularCuboids”: IrregularCuboidRectree-like (x 8)
- “DivideIntoEightRegularCuboidsOnce”: RegularCuboidRectree-like (x 8)
- “DivideIntoEightRegularCuboidsRepeatedly”: RegularCuboidRectree-like
- “DivideIntoEightCubesOnce”: only if the Rectree is cubic (x 8)
- “DivideIntoEightCubesRepeatedly”: only if the Rectree is cubic

MinSize:
The MinSize for each of the three axes can be different.

AxisAlignedRegularCuboidRectree

This Rectree is “axis aligned” – meaning that the cuboids are positioned parallel to the main world axes of the global coordinate system.

The contained sub-Rectrees can be a fitting AxisAlignedRegularCuboidRectree or a fitting AxisAlignedCubicRectree.

AllowedRectreeExtensionTypes:

- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Rectree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough
- “ExtendAndOctuplicate”: add seven Rectrees (irregular cuboids!)
- “QuadruplicateOnce”: can duplicate in 2 directions (x 4)
- “QuadruplicateRepeatedly”: quadruplicate multiple times until it fits
- “ExtendAndQuadruplicate”: add three Rectrees (irregular ones!)
- “DuplicateOnce”: able to duplicate along one direction / axis (x 2)
- “DuplicateRepeatedly”: duplicate multiple times until sufficient
- “ExtendAndDuplicate”: add one new Rectree (with irregular cuboid!)

AllowedRectreeSubdivisionTypes:

- “None”: no division possible
- “DivideIntoEightRegularCuboids”: RegularCuboidRectree-like (x 8)
- “DivideIntoEightRegularCuboidsRepeatedly”: RegularCuboidRectree-like
- “DivideIntoEightCubesOnce”: only if the Rectree is cubic (x 8)
- “DivideIntoEightCubesRepeatedly”: only if the Rectree is cubic

MinSize:
The MinSize for each of the three axes can be different.

AxisAlignedCubicRectree

This Rectree is “axis aligned” – meaning that the cubes are positioned parallel to the main world axes of the global coordinate system.

The contained sub-Rectrees can be a fitting AxisAlignedCubicRectree.

AllowedRectreeExtensionTypes:

- “None”: no extensibility possible – borders remain always unchanged
- “OctuplicateOnce”: if necessary extend the Rectree in all 3 directions (x 8)
- “OctuplicateRepeatedly”: octuplicate multiple times until it is enough
- “ExtendAndOctuplicate”: add seven Rectrees (irregular cuboids!)
- “QuadruplicateOnce”: can duplicate in 2 directions (x 4)
- “QuadruplicateRepeatedly”: quadruplicate multiple times until it fits
- “ExtendAndQuadruplicate”: add three Rectrees (irregular ones!)
- “DuplicateOnce”: able to duplicate along one direction / axis (x 2)
- “DuplicateRepeatedly”: duplicate multiple times until sufficient
- “ExtendAndDuplicate”: add one new Rectree (with irregular cuboid!)

AllowedRectreeSubdivisionTypes:

- “None”: no division possible
- “DivideIntoEightCubesOnce”: divide into eight CubicRectrees (x 8)
- “DivideIntoEightCubesRepeatedly”: divide several times if necessary

MinSize:
The MinSize for each of the three axes must be the same (it’s a cube).

CubicDungeon

A CubicDungeon has a special tree of cubic cells (an AxisAlignedCubicRectree).
All the other space outside the Rectree is called "Walls" – it means that there is a non-transparent boundary around the cubic cells – the user camera can only watch along the cubic cells – if there is no "Closed Door" along the way – that means two neighbored cells are seperated by a non-transparent obstacle that fully closes the whole passage.
Advantage:
Because of the special structure of the "Dungeon", visibility calculations are easy and fast.
Disadvantage:
The structure allows only axis aligned cubic cells and bounds – for alternatives take a look at the "BinarySpacePartition" and the "CellsAndPortals" SceneManagers.

Every single cubic cell contains a list of all static and dynamic objects that have bounding volumes that are (at least partially) inside the cubic cell.
Special methods are used to change the position / orientation of dynamic objects:

→ methods:
TransferDynamicObject

There are methods to insert and to remove objects – both return integer values indicating if the insertion or removal was successful (code "zero" means – everything was okay...).
Even if the bounding box of the object doesn't fit into cubic cell, the operation can still be performed – objects can simultaneously be in many different cubic cells...

There should be some convenience methods to easily load and save a dungeon map.

→ methods:
LoadCubicDungeon, **SaveCubicDungeon**

The loading methods can read a png-file with RGBA information and a translation config file that lists the meaning of the values inside a text script.
For example:
Red = kind of wall texture around this cube (up to 256 different texture combinations)
Green and Blue = items / static objects / doors (up to 65.536 different things)
Alpha = type and number of enemies in this cell (up to 128 different combinations – the first bit tells the program if the enemies are mobile and autonomous or holding the position)

MaxSize:

The MinSize has three sub-attributes:

- "MaxSizeX": maximum size (number of cubes) along the global X-axis
- "MaxSizeY": maximum size (number of cubes) along the global Y-axis
- "MaxSizeZ": maximum size (number of cubes) along the global Z-axis

CenterOfDungeon:

The middle of the dungeon (in world coordinates)

BinarySpacePartition

A BSP (Binary Space Partition) utilizes many clipping planes (positioned at the walls of the level) to generate a graph that can be used for culling that is mainly used for “indoor scenes”.

There are methods to insert and to remove objects – both return integer values indicating if the insertion or removal was successful (code “zero” means – everything was okay...).
If the bounding box of the object doesn’t fit into a Rectree, the operation cannot be performed and an error code is returned (or an exception is thrown – whatever is better).

There should be some convenience methods to easily load and save a BSP map.

→ methods:
LoadBSPScene, **SaveBSPScene**

The loading methods can read and import several standard formats for BSP scenes and the saving methods should be able to save an “enriched BSP scene” that holds information about any dynamic and static objects other than walls and textures.

MaxSize:

There should be bounding volume showing the bounds of the BSP scene

CellsAndPortals

“Cells & Portals” is another effective algorithm for indoor scenes.
It is similar to the CubicDungeon, but allows for arbitrary sizes of the cells and portals.

There are methods to insert and to remove objects – both return integer values indicating if the insertion or removal was successful (code “zero” means – everything was okay...).
It is important that the dynamic objects don’t “fly through walls” and are always in the “inside” not in the “outside” – a sophisticated collision detection becomes essential.

There should be some convenience methods to easily load and save a C&P map.

→ methods:
LoadCAPScene, **SaveCAPScene**

The loading methods can read and import different formats.
The saving methods should be able to save an “enriched CAP scene” that holds information about dynamic objects, static objects and textures.

MaxSize:

There should be bounding volume showing the bounds of the C&P scene

TerrainManager

A terrain manager can dynamically load terrain tiles that may have different properties – for example they could be saved in a special format (a height map) or they could be generated and decorated procedurally on the fly (for endless terrains).
To easily calculate visibilities the terrain manager should be combined with a special axis aligned cuboid rectree (that is able to delete sub-trees / neighbored nodes).
To further enhance the the culling efficiency occlusion culling should be integrated (it becomes essential when the scene shows a canyon or a city with skyscrapers.

There are methods to insert and to remove objects – both return integer values indicating if the insertion or removal was successful (code "zero" means – everything was okay...).
Dynamic objects can be autonomous or they freeze their position when the get out of sight: When the user camera leaves an area, the other users or the computer-controled "people" will move on, even if they cannot be seen and even if the terrain tile they are walking on isn't loaded into the memory.
Thus there should be a sophisticated resource management integrated into the terrain manager or at least an association to a good resource manager should be available...

There should be some convenience methods to easily load and save a height map.

→ methods:
LoadHeightMap, **SaveHeightMap**

The loading methods can read and import different formats.
The saving methods should be able to save an "enriched height map" that holds information about dynamic, static objects and textures (e.g. trees or buildings positioned on the map).

If the terrain is generated on-the-fly, scripts for procedural calculations and their variable values should be loadable and savable.

→ methods:
LoadProceduralConfigScript, **SaveProceduralConfigScript**, **SaveHeightMap**

The saving methods should be able to save an "enriched height map" that holds information about dynamic and static objects (e.g. trees or buildings positioned on the map) and textures.

MapSizeX:

This variable could hold different meanings:

- "UnlimitedX": never ending map – you can endlessly walk to east or west
- "WrapX": if you walk long enough to the west, you will enter again from east

MapSizeY:

This variable could hold different meanings:

- "UnlimitedY": never ending map – you can endlessly walk to north or south
- "WrapY": if you walk long enough to the north, you will enter again from south

PlanetManager

A PlanetManager uses a special spherical shell containing a Tri-Quad-Tree for the surface and a clipping plane for those parts of the surface that "lie behind the horizon".
The planet can be loaded either from a high resolution height map (e.g. those images that can be retrieved from the NASA-repository or other sources) or the planet can be generated on-the-fly by a procedural algorithm.

There are methods to insert and to remove objects – both return integer values indicating if the insertion or removal was successful (code "zero" means – everything was okay...).
If the bounding box of the object doesn't fit into the spherical shell or is below the surface, the insertion will be rejected and an error code is returned (or an exception is thrown – whatever is better).

There should be some convenience methods to easily load and save a height maps.

→ methods:
LoadPlanetHeightMap, **SavePlanetHeightMap**

The loading methods can read and import different formats (interpolation and extrapolation is often inevitable – the resolution of the projected pictures are seldomly in the right format).
The saving methods should be able to save an "enriched height map" that holds information about dynamic objects, static objects (e.g. trees or buildings positioned on the map) and textures and other stuff.

If the planet surface terrain is generated on-the-fly, scripts for procedural calculations and their variable values should be loadable and savable.

→ methods:
LoadProceduralConfigScript, **SaveProceduralConfigScript**, **SaveHeightMap**

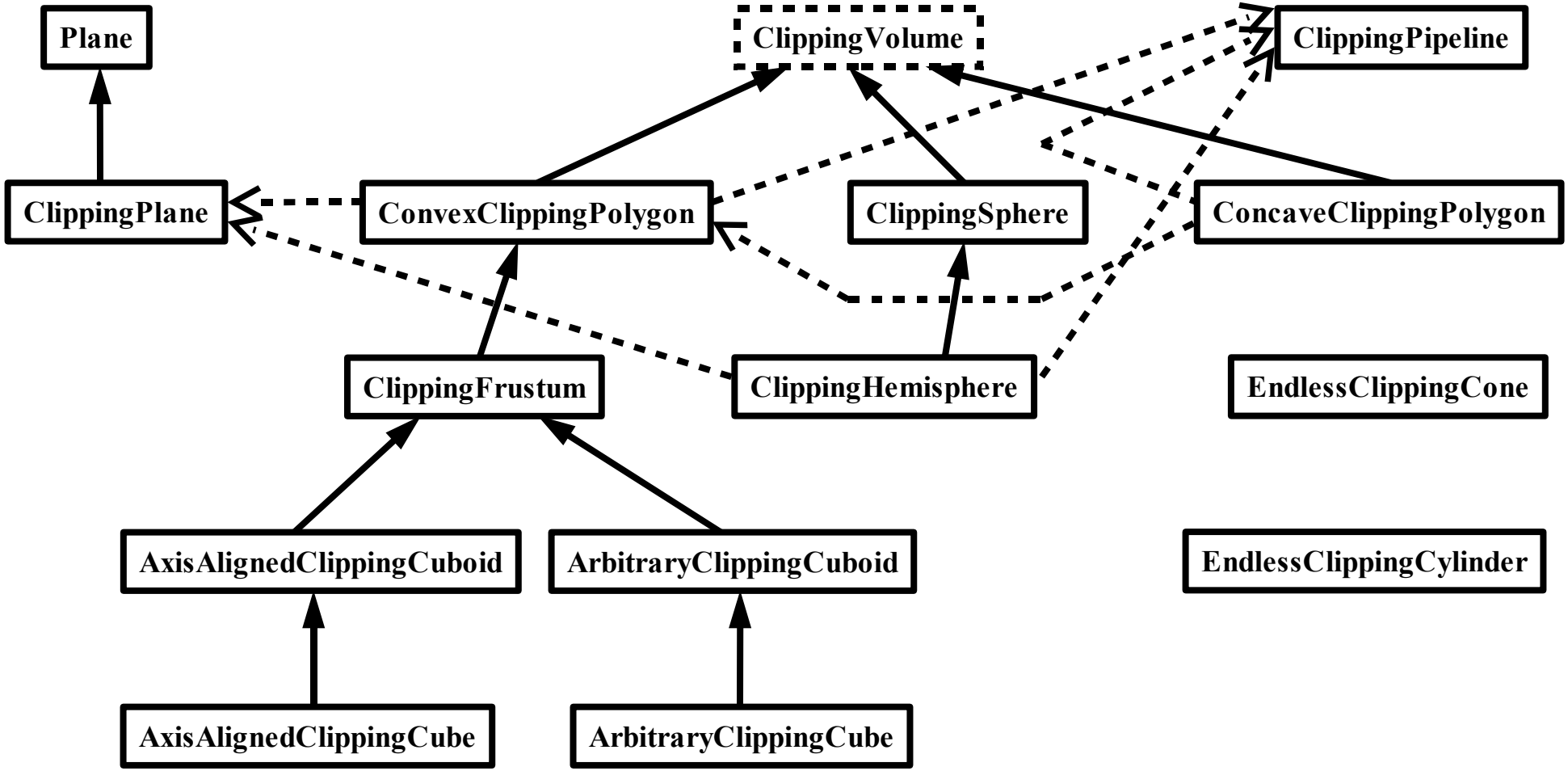
The saving methods should be able to save an "enriched height map" that holds information about dynamic and static objects (e.g. trees or buildings positioned on the map) and textures.

MaxSize:

There should be bounding volume showing the bounds of the whole planet

Helper Classes

Concept: Some clipping utilities (you always have the choice: “clip the inside” or “clip the outside”).



Plane

The plane should be able to calculate the Hesse normal form from many different other equation types or given spanning vectors / points in the plane.
With the Hesse normal form it will be easy to calculate distances to the plane in three-dimensional space. By definition a distance greater than zero means “outside” and a distance smaller than zero means “inside”.

ClippingPlane

A clipping plane should be able to calculate distances to many different bounding volumes and can then itself be part of complex bounding volumes.
One prominent use of clipping planes is the use as a side of the view frustum.
BinarySpacePartition also uses the ClippingPlane class.
However any ConvexClippingPolygon can be build using the ClippingPlane class.

ClippingVolume

A clipping volume can be used to clip anything outside or inside the volume.
The decision whether to clip the inside or the outside should be made by the programmer.
A clipping volume must be completely closed to guarantee a clear definition of “inside” and “outside” – even though a “half space” / “unlimited volume” could also be possible. In order to implement such a half space or unlimited volume one could easily use the class “ClippingPipeline” that offers an intelligent queue or combination of different clipping utilities like clipping planes, clipping spheres and so on...

ClippingPipeline

Clipping pipelines are just sequences or combinations of tests against several possibly different clipping objects like clipping planes and clipping volumes and all the others.
The ClippingPipeline allows further configuration of the way these tests are performed and if sophisticatedly implemented it can even find shortcuts that shorten the time needed to perform the calculation. With such a pipeline very complex clipping volumes can be constructed or complex half spaces.

ConvexClippingPolygon

A convex polygon can be defined by a number of clipping planes – but the polygon should be closed. If the polygon isn't convex and closed, the result of the clipping calculations becomes erroneous. A concave polygon can be described by a combination of many convex polygons (see the description of the "ConcaveClippingPolygon").

ClippingFrustum

A clipping frustum has the form of a truncated pyramid – that means it consists of six clipping planes. The front plane is also called the "near clipping plane" and the clipping plane on the other side is called the "far clipping plane". The left, right, top and bottom planes are constructed in such a way that the near and far clipping planes become rectangles. A clipping frustum can also be constructed from a virtual user camera – the eye point, opening angle and the distance to the near and far clipping planes can be used to calculate a frustum.

AxisAlignedClippingCuboid

A cuboid is similar to a frustum, but it has parallel sides with right angles.
"Axis aligned" means that the cuboid is also parallel to the three world axes.

AxisAlignedClippingCube

A cube is similar to a cuboid, but all edges have the same length.
"Axis aligned" means that the cube is also parallel to the three world axes.

ArbitraryClippingCuboid

A cuboid is similar to a frustum, but it has parallel sides with right angles.
"Arbitrary" means that the orientation of the cuboid can be of any kind.

ArbitraryClippingCube

A cube is similar to a cuboid, but all edges have the same length.
"Arbitrary" means that the orientation of the cube can be of any kind.

ClippingSphere

A sphere is a very simple clipping volume – it has a center and a maximum allowed distance (the “radius”). The outside is everything that lies farther away than the radius.

ClippingHemisphere

A hemisphere is a combination of a sphere and a clipping plane that runs through the center point of the sphere. In order to construct a ClippingHemisphere you could also use a “ClippingPipeline” with a sphere and a plane – thus this class is just for your convenience.

ConcaveClippingPolygon

A concave polygon can be described as a combination of many convex sub-polygons – then the calculation becomes very simple:
If a tested object is inside any of the sub-polygons, the object is inside the concave polygon.
If a tested object subtends any of the sub-polygons, the object subtends the concave polygon.
If a tested object is outside all of the sub-polygons, the object is outside the convex polygon.

EndlessClippingCone

Such an “endless cone” can be defined by a starting point, a direction vector and an opening angle, smaller than ninety degrees. For other angles (even or greater than ninety degrees):
If the angle should be even to 90°: You can use a plane perpendicular to the direction vector.
If the angle is greater than 90°: Just invert (switch the definitions of “outside” and “inside”).
A “ClippingCone” (special type of “ClippingVolume”) can be constructed from an EndlessClipingCone and a perpendiclar clipping plane.

EndlessClippingCylinder

An “endless cone” can be described by a straight line and a maximum distance to this line. The equation of the line should be in the Hesse normal form, to keep things easy, but as you know, such an equation can simply be derived from a given point and a vector.
A “ClippingCone” (a special type of “ClippingVolume” can be build by an endless cone and two perpendicular and parallel clipping planes with different orientations.