Advanced 3D-Data Structures

Martin Haidacher

April 4, 2011

1 Introduction

The basic element in computer graphics is the data which has to be represented. The data can originate from very different sources. It can, e.g., originate from a digitizer, a modeling software package or a simulation. The origin of the data is one reason to choose a certain internal data representation. In this case the data representation should be able to represent the original data best.

Besides the data origin the choice of an internal data representation also depends on features of the application in which the data is rendered. For example it makes a difference if it is important to render data in real-time versus the accurate representation of details.

In this document different data representations are described. All of them have their advantages and disadvantages. Which of them is chosen for a certain data origin and application is dependent on the requirements. In the next section different requirements are listed which influence the choice of a data representation. In Section 3 the most common data representations for spatial data are described in detail.

2 Data Representation Requirements

The following list specifies the main requirements on spatial data representations:

Representation of general objects The data representation is able to represent objects with arbitrary shape. This requirement is important for a modeler who has to generate models of various shapes.

Exact representation of objects It is possible to describe an object exactly with the data representation. For some applications it is not good

enough if an object is represented by approximations. Especially for curved surfaces it makes a difference if the data representation is able to describe the shape exactly or by approximation.

Combinations of objects For some applications it is necessary to combine different objects into a single one. This can be done by logical operations such as AND, OR and XOR.

Linear transformation Linear transformations (e.g., rotation, scaling, translation) are used to change the shape and position of an object in a scene. In some data representations these transformations can be applied very fast in others it is rather complicated to apply them.

Interaction The interaction requirement is of particular interest for a modeler who wants to modify an object. Therefore, it should be possible to easily interact with an object or parts of it.

Fast spatial searches For some applications it is important to do spatial searches to find, e.g., the nearest neighbor to a current point. This is for example necessary for collision detection in computer games.

Memory consumption If objects are very large or if they have to fit in a certain memory space, such as the GPU memory, the memory consumption is important in the selection of a suitable data representation. Especially if it is important to represent complex objects accurately the memory consumption might be excessive with the wrong data representation.

Fast rendering For real-time applications it is important that the rendering process is fast. Some data representations are designed to fulfill this requirement because they are adapted to the rendering pipeline of the graphics hardware.

3 Internal Data Representations

The representation of spatial data is an important issue in game programming, computer graphics, visualization, solid modeling, and related areas including computer vision and geographic information systems (GIS) [3]. Depending on the field and the characteristics of the data different types of representations are useful. In the remainder of this section some of the most common data representations are described. The capability of transformations, combinations and fast rendering of objects are explained in more

detail for each data representation. Furthermore a list of advantages and disadvantages is given.

3.1 Point Cloud

In a point cloud representation data is represented as a cloud of spatial points. This is the simplest representation of spatial data. Each point is an individual object. There exists no information about the relation of different points. In particular, connectivity information is not stored. 3D scanning is an example which generates such a point cloud. In this case a laser range scanner is sampling the surface of an object. For each sample point the distance between the scanner and the surface point is retrieved. By knowing the position of the scanner in respect to the object as well as the accurate direction of the laser, a three dimensional point can be calculated based on this distance. The scans from different view points are finally registered and combined to a single point cloud which represents the whole object. Figure 1 shows an example of such a point cloud.

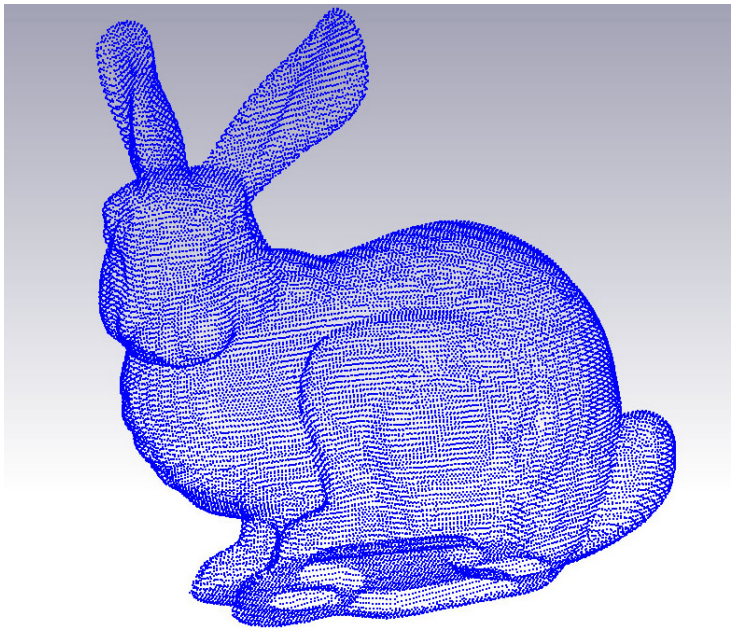

Figure 1: A point cloud generated by a 3D scanner (Source [4]).

Data Structure

- *Object* = List of points

- *Point* = Spatial coordinate (x,y,z)

Transformations

To transform a whole object it is only necessary to transform each individual point of the cloud.

Combinations

In a point cloud representation no information about the inside or outside of an object exists. Therefore it is, e.g., not possible to subtract one object from another. The only combination which is possible is the combination of the points of both objects to a single object.

Memory Consumption

The accurate representation of curved or complex surfaces needs many points and therefore a high memory consumption.

Rendering

The rendering process for points is very simple. Each point is simple projected onto the corresponding pixel on the screen. To represent closed surfaces it is necessary to have enough points (≥ 1 point per pixel). The z-buffer can be used to get the correct occlusion of the points. Due to the increasing power of graphics hardware the complexity of models has rapidly increased in recent years. Single polygons might be much smaller than the area of a pixel on the screen in the common representation. In such a case it is more efficient to represent the model with a point cloud in respect to memory consumption and rendering speed. Therefore it is possible that point clouds are used more often to represent models.

Advantages

+ Fast and easy rendering

+ Exact representation with correct occlusion possible

+ Fast and easy transformations

Disadvantages

- Many points for complex surfaces
- High memory consumption (especially if the surface is complex)
- Many points are needed to represent a closed surface

3.2 Wire-Frame Model

With the wire-frame model each object is represented by a couple of straight edges. A cube, e.g., is represented by its twelve edges. There is no information about the space between edges. Therefore the wire-frame model cannot be used to describe surfaces. It is often used for the construction of buildings or other objects because there it is especially important to see the edges of the model.

Data Structure

- *Object* = List of pointers to edges of the model
- *Edge List* = List of all edges of the model
- *Edge* = A pointer to two vertices (start and endpoint of an edge)
- *Vertex List* = List of all vertices of the model
- *Vertex* = Spatial coordinate (x,y,z)

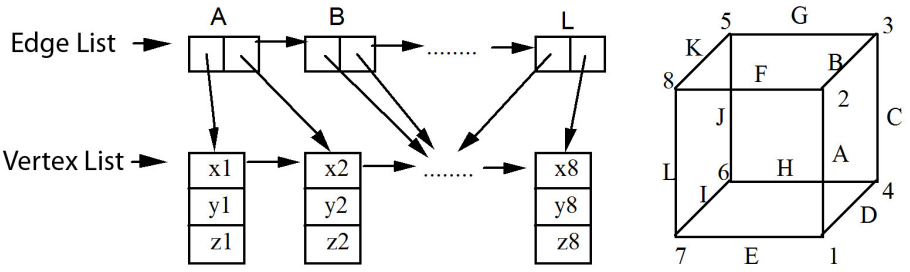

Figure 2: Data structure for the representation of a wire-frame model.

Transformations

Transformations are easy to execute. Only the vertices in the vertex list have to be transformed to transform an object.

Combinations

Similar to the point cloud the wire-frame model does not contain any information about the inside or outside of an object. The only possibility to combine two objects is to combine all vertices and edges of both objects into a single vertex and edge list.

Memory Consumption

The advantage of the wire-frame model is that one vertex can be used as starting point for many edges. This can be used to reduce the memory space. For curved surfaces, such as a sphere, many vertices and edges are necessary to approximate it accurately.

Rendering

This model was invented for output devices which are only able to draw lines (e.g., vector displays or plotters). For these devices this internal representation is good enough since they are not able to render objects more realistically.

For the rendering it is enough to project all points of the vertex list on the image plane and draw the edges inbetween. In this manner the whole model can be represented. With the built-in functions for the projection and line drawing in graphics hardware this is a very fast process. Therefore the wire-frame representation is sometimes used to give a fast overview over a model without rendering it in detail.

One problem with the wire-frame model is the large number of lines which are rendered on the screen. Since it is not possible to eliminate hidden edges, it is hard to recognize which edges are closer and which ones are further away from the viewer. A hidden-line method cannot be applied because the geometric information which would be necessary for such a method does not exist in the wire-frame model.

Advantages

+ Fast rendering

+ Transformations are simple to implement and fast to process

Disadvantages

- High storage consumption for complex objects with curved surfaces

- Loss of information (even hidden edges cannot be eliminated)

- All curved surfaces must be represented by straight edges

3.3 Boundary Representation

With the boundary representation (B-Rep) all objects are represented by their boundaries. Compared to the wire-frame model the B-Rep also contains additional information about the faces. This makes it possible to represent an object as a closed unit.

With the B-Rep it is also common to store additional information for each face. Such information is for example the normal vector of a face. This additional information helps to speed-up the rendering process because certain calculations can be avoided in the rendering process.

Curved surfaces of an object can not exactly be represented by a B-Rep. Similar to the wire-frame model curved surfaces have to be approximated by small polygons.

Data Structure

- *Object* = List of pointers to faces of the model
- *Face List* = List of faces
- *Face* = List of pointers to edges of the model
- *Edge and Vertex List* = Similar to wire-frame model

Usually only planar polygons are accepted as faces. With this restriction the rendering can be done more efficiently. In Figure 3 an example of a B-Rep is shown. In this structure the faces are the central starting point for the list.

An alternative to this model is the winged-edge data-structure. This data structure has also a face, edge and vertex list but in comparison to the B-Rep model the central part of the data structure is the edge. Each edge stores a pointer to two vertices (start point and end point) as well as a pointer to the two faces on each side of the edge. Furthermore a pointer is stored to the successor and predecessor edges of the faces. In Figure 4 an example for a winged edge data structure is given.

The advantage of the winged-edge data-structure is the possibility to find neighboring polygons fast. Since each edge holds all the information about the faces on both sides of the edge, it is possible to find all neighboring faces of one face by stepping through the edge list of this face. In the B-Rep shown

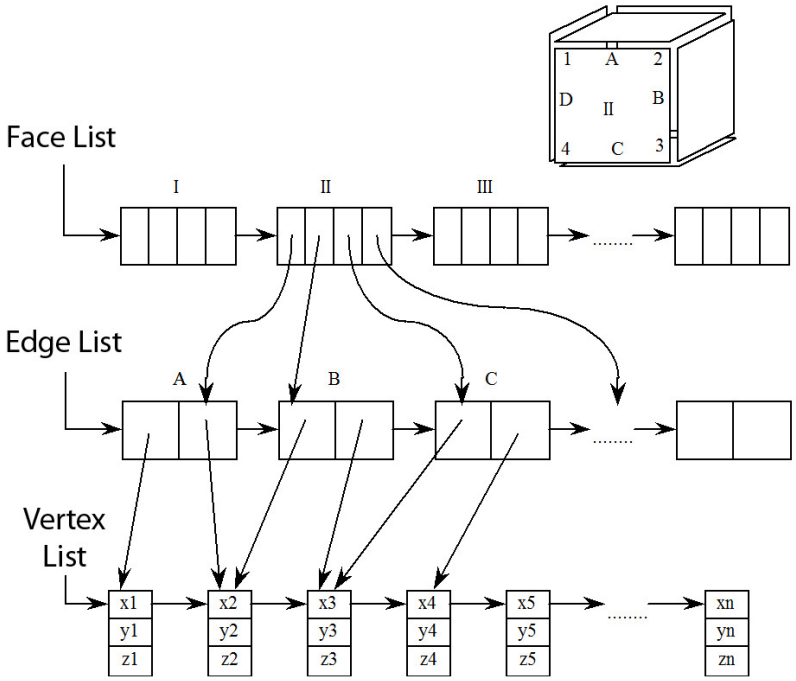

Figure 3: B-Rep: Most common data structure for boundary representation.

in Figure 3 a neighboring face can only be found by querying all other faces in the list. A disadvantage of the winged-edge data-structure is that it has problems with discontinuities in the model and it has to store more pointers in comparison to the most common model in Figure 3.

Transformations

Similar to the wire-frame model it is enough to transform all the vertices in the vertex list to transform an object. If additional information, such as normal vectors, is stored in the face list then the transformation also has to be applied to these vectors. Alternatively the vectors can be re-calculated after the transformation.

Combinations

With the B-Rep it is easily possible to do simple operations like clipping with a plane. For more advanced operations like union, intersection or difference an additional semantic has to be defined for an object: Each object is considered to be closed and has no holes in it. For simplicity, the planes in the face lists are defined in a way that all the normal vectors point away from

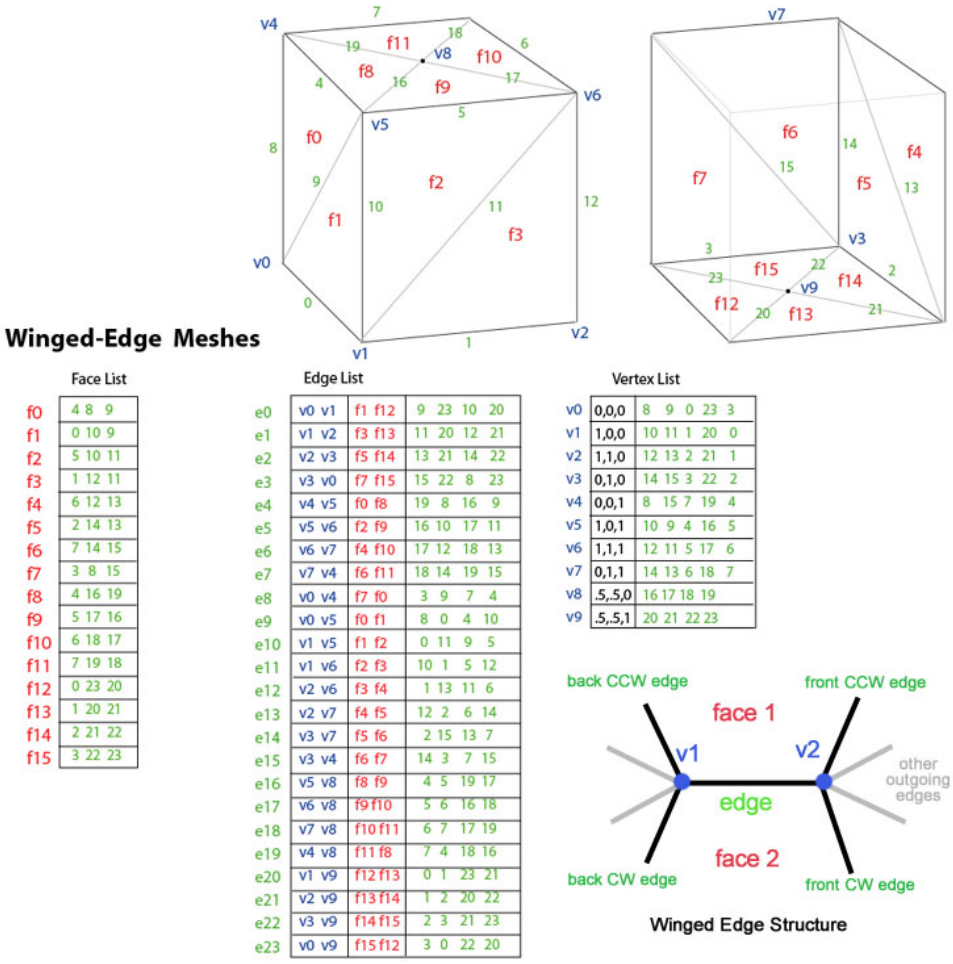

Figure 4: The winged edge data structure (Source [5]). The basis of the structure is the edge.

the object.

If these semantics are fulfilled then a combination in a B-Rep between object A and B can be done with the following algorithm:

1. Cut all polygons of object A with all polygons of object B (if necessary)
2. Cut all polygons of object B with all polygons of object A (if necessary)
3. Classify all polygons of object A as *inside object B*, *outside object B* or *on the surface of object B*
4. Classify all polygons of object B in the same manner with object A

5. Depending on the operation remove the redundant polygons of A and B and merge both B-Reps together

ad 1 and 2: To speed-up the cutting process a bounding box can be defined for each polygon. This additional information allows faster testing if a polygon of one object intersects a polygon of the other object at all. Only when bounding boxes of both polygons intersect, a more complicated intersection test between the two polygons is necessary. To further improve the cutting algorithm all the polygons should be convex. This reduces the number of queries which have to be made during the cutting process.

ad 2 and 3: The test whether a polygon is inside or outside of an object is done by shooting a ray along the normal vector of the polygon. The ray is tested for an intersection with all the polygons of the object. Only the nearest polygon intersected by the ray is then considered for the inside or outside test. If the normal vector of this polygon points in the same direction as the ray then the polygon is inside of the object. Otherwise it is considered to be outside of the object. Figure 5 shows how this test looks like for two polygons.

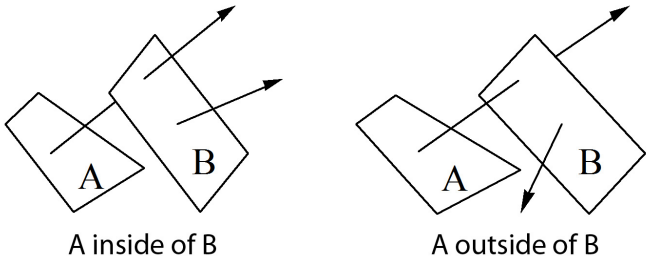

Figure 5: Inside-outside test for two polygons A and B.

Improvement: Points of object A which are on the surface of object B (and vice versa) are classified as border points during the cutting algorithm. Two neighboring polygons always have the same classification (inside or outside) except they are separated by a border point. Therefore it is possible to inherit the classification of one polygon from another one until a border point is reached. In Figure 6 this is illustrated in a simple example.

ad 5: As mentioned before, after the polygons are cut at the intersections between the objects, some polygons are removed depending on the operation. Table 1 gives an overview which polygons of object A have to be removed for the union, intersection and difference operation.

Table 2 shows the removal information for polygons of object B.

In both tables the first two columns are rather straightforward. They describe to remove or not to remove polygons inside of an object or outside

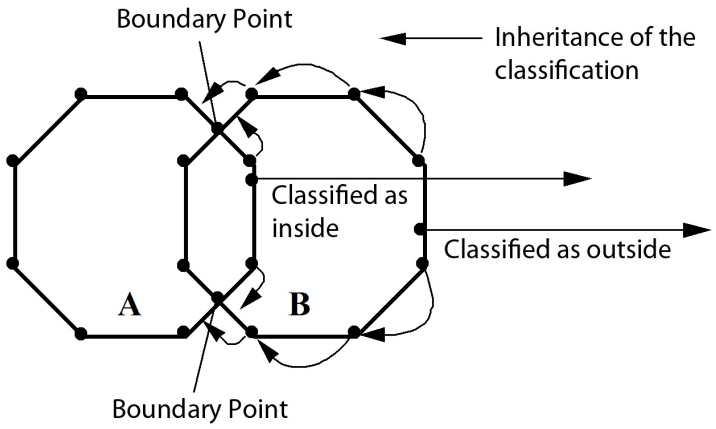

Figure 6: Classifications which are already done can be inherited in the neighborhood until a border point is reached.

	inside B	outside B	on B (coplanar) $N_A = N_B$	$N_A \neq N_B$
A or B	yes	no	no	yes
A and B	no	yes	no	yes
A minus B	yes	no	yes	no

Table 1: Removal of polygons of object A.

of an object. For polygons which are coplanar with polygons in the other object, the normal vector has to be considered for the decision.

As a result of this algorithm further requirements are necessary for the boundary representation:

- *Only convex polygons*: Simplifies the algorithm for the cutting of polygons
- *No double vertices*: Otherwise the classification cannot be inherited as explained before
- *No open objects*: Otherwise the inside/outside test cannot be performed
- *Additional pointers in the data structure*:

  In the vertex list, which contains all vertices from the original polygons as well as the vertices from the polygons after cutting them, a pointer is added. This pointer links to other vertices which inherited the classification (inside or outside) from it. If all vertices of a face are classified

	inside A	outside A	on A (coplanar) $N_A = N_B$	$N_A \neq N_B$
A or B	yes	no	yes	yes
A and B	no	yes	yes	yes
A minus B	no	yes	yes	yes

Table 2: Removal of polygons of object B.

as either inside or outside then the entire face can be classified as inside or outside. Figure 7 shows how the additional pointers are added to the vertex list.

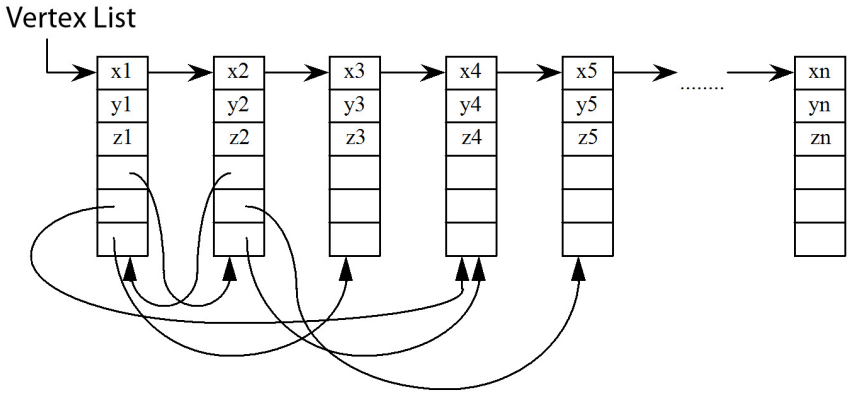

Figure 7: The inheritance of the inside/outside-classification is additionally stored in the vertex list.

Memory Consumption

In addition to the vertex and edge information the B-Rep additionally stores a face list. The face list is in comparison to the edge and vertex list smaller since a face consists of many edges and vertices. Therefore the additional memory consumption is rather small.

The memory consumption can be high if curved surfaces are to be represented with high accuracy. In this case many small polygons are used to approximate highly curved regions. The technique for the thus necessary subdivision depends on the characteristics of the surface:

- *Parametrizable surfaces* (free form surfaces, quadrics, superquadrics): The parameter space is subdivided into small pieces. Each of the corresponding surface pieces is approximated by a planar polygon. The

smaller the subdivision the closer the approximation is to the original function.

- *Non-parametrizable surfaces* (non-planar polygons): The non-planar polygons are subdivided into smaller polygons (mostly triangles). This subdivision is called tesselation. In Figure 8 an example for tesselation is given. The tesselation is a recursive process. It is repeated until the approximation is good enough. As a parameter for the termination of the tesselation the difference in the directions of the normal vectors is used. If the directions of both normal vectors differs by less than a threshold then the tesselation is stopped. The following equation formalizes this:

$$N_1 \cdot N_2 \geq 1 - \epsilon \tag{1}$$

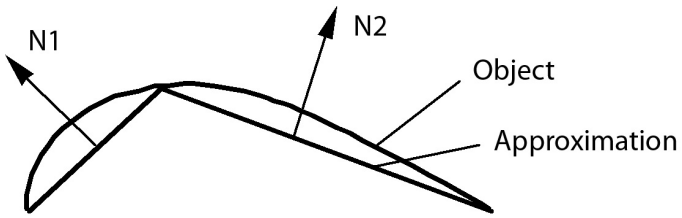

Figure 8: Curved objects are represented by an approximation through flat surfaces.

Rendering

For the rendering it is necessary to use an algorithm which can handle hidden lines and hidden surfaces.

Advantages

+ Transformations are easy to implement

+ General representation of many objects

+ Model generation based on digitized objects is possible

Disadvantages

- High memory consumption (especially for the accurate approximation of curved surfaces)

- Combinations are costly

- Curved surfaces have to be approximated

3.4 Binary Space Partitioning Tree

The binary space partitioning tree (BSP tree) is a special type of a boundary representation. BSP trees are especially practical for static scenes (no animated objects where each frame changes the internal representation). In a BSP tree the polygons of an object are used to partition the space into smaller subspaces. This partition simplifies the rendering process by an easier and faster sorting of the polygons. Depending on the view point the polygons can be drawn from back to front. By doing it in this order, polygons which are farther away from the viewer are occluded by polygons which are closer to the viewer. This kind of representation is mostly used for static scenes of the environment in graphics applications. Examples for such applications are computer games or flight simulators. The advantage is that all polygons can be drawn with the correct occlusion very fast without the need of any hidden surface algorithm. Another common example is scenes with transparent objects.

Data Structure

- *Object* = Pointer to the root node

- *Node* = Polygon, pointers to two child nodes

Each node of the BSP tree represents a planar polygon. The plane in which this polygon lies (separation plane) subdivides the space in two half-spaces. The half-space in the direction of the normal vector is the space in front of the polygon. All polygons which are in this part are added as child nodes on the left subtree of the parent node. Polygons in the half-space behind the polygon are added as child nodes on the right subtree of the parent node. In Figure 9 the basic structure of a BSP tree is shown.

If a polygon intersects a separation plane then the polygon must be cut into two polygons. Each polygon further down in the BSP tree subdivides only the remaining space represented by the parent node. In Figure 10 an example for a simple BSP tree is shown. In this case polygon 1 subdivides the whole space into two subspaces. All other polygons are behind polygon 1. Therefore they are all added on the right side of the node for polygon 1. Polygon 2 subdivides the space again but only the remaining polygons in

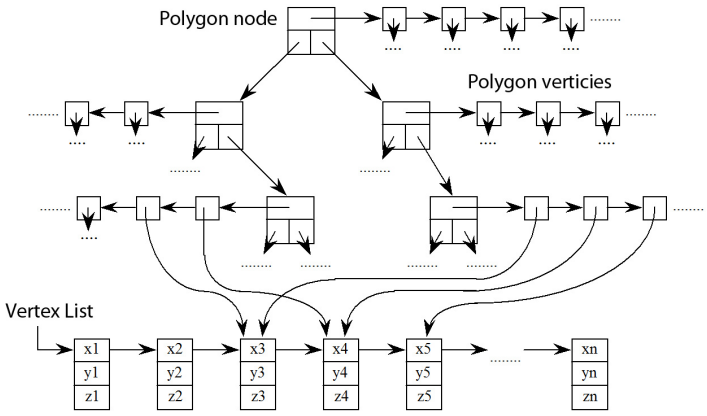

Figure 9: Structure of a BSP tree.

the subtree are considered for the test if they are behind or in front of the separation plane. This procedure is then repeated for polygons 3 and 4.

Figure 11 and 12 show two more examples for the generation of a BSP tree. For the example in Figure 11 two different possibilities are shown how the BSP tree can be generated. With the approach on the right side it is necessary to cut polygon 1 into two pieces (1a and 1b).

Due to the structure of a BSP tree it is easy to answer the question if a point is inside or outside of an object. The position of a point in the tree structure is determined by traversing the tree starting from the root. At each node the tree is further traversed along the right subtree if the point is located behind the considered polygon represented by the considered node. Otherwise the tree is traversed along the left subtree. Only if the last step from the root to the leaf is to the right, the point is inside the object (inbetween there can be left steps).

For the construction of a BSP tree the following algorithm can be used:

- If the object is convex (as in Figure 10) then the generation is trivial (linear list).
- Else the face list of the B-Rep is converted into a BSP tree with the following algorithm:
    1. Search for the polygon which intersects the fewest other polygons with its separation plane (in practice a few polygons are selected randomly and from these the one which intersects the fewest polygons).

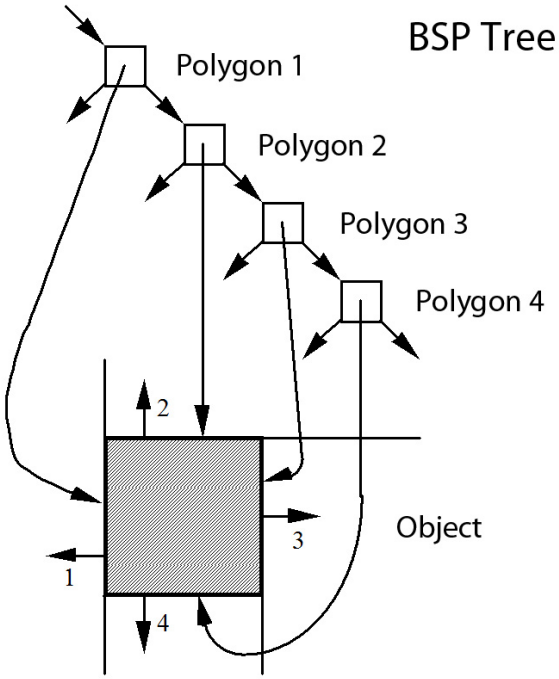

Figure 10: Generation of a BSP tree.

2. Subdivide the face list into two parts: in polygons which are in front of the separation plane and polygons which are behind the separation plane.
3. The selected polygon of step 1 becomes the root node. With the polygons in the two face lists the left and right subtrees are generated. The left subtree contains only polygons which are in front of the separation plane and the right subtree only contains polygons which are behind the separation plane.

Transformation

For the transformation of an object all vertices in the vertex list and the plane equations as well as the normal vectors of the faces have to be transformed.

Combinations

In one possibility the combination of objects can be done on the B-Reps as explained in the previous section. After that the BSP tree of the combined object is generated. A second possibility is to combine two objects directly with their BSP tree representations. This is the faster option.

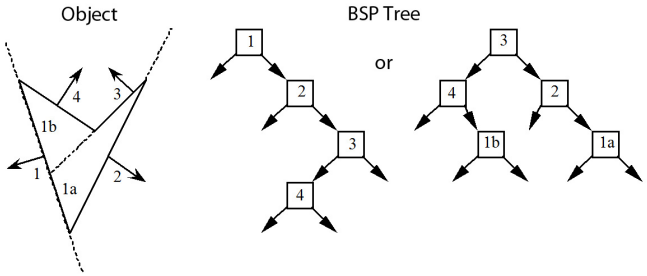

Figure 11: Example for a BSP tree. The tree can be generated in two different ways.

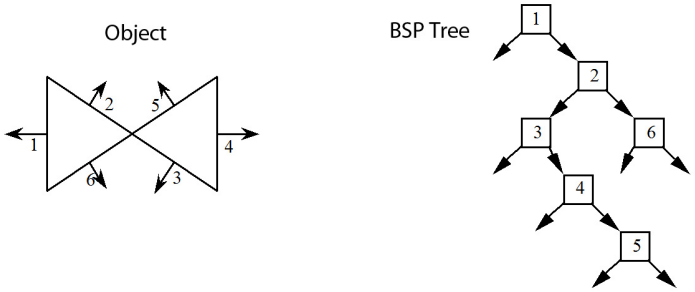

Figure 12: Another example for a BSP tree.

If the combination is directly done in the BSP tree the following problem has to be solved: Object A should be combined by the operation *op* with object B. The operation can be formalized in this way:

$$C = A \text{ op } B \tag{2}$$

Object C is the resulting object of the operation *op*. It should then also be represented in a new BSP tree. The algorithm for the execution of this combination is the following:

- Polygon P_A of the root node of object A is intersected with object B. This results in two new polygons:

$$\begin{aligned} P_A^{in} &= \text{Part of } P_A \text{ inside object } B \\ P_A^{out} &= \text{Part of } P_A \text{ outside object } B \end{aligned} \tag{3}$$

- The separation plane of P_A is used to separate object B into two parts:

$$\begin{aligned} B^- &= \text{Part of } B \text{ behind the separation plane of polygon } P_A \\ B^+ &= \text{Part of } B \text{ in front of the separation plane of polygon } P_A \end{aligned} \tag{4}$$

The root node of the new object C gets the separation plane of P_A as separation plane and depending on the operation P_A^{in} or P_A^{out} as polygon (for union and difference operations P_A^{out}, for the intersection operation P_A^{in}). The subtrees of C are generated recursivly from the subtrees A_l and A_r of object A as well as the separated trees B^- and B^+ of object B:

$$\begin{aligned} C_l &= A_l \text{ op } B^- \\ C_r &= A_r \text{ op } B^+ \end{aligned} \tag{5}$$

The recursion stops if one of the two operands (A or B) represents a homogeneous subtree. This means it is a leaf node and either full or empty. In this case the result of the operation is determined according to Table 3.

op	A	B	A op B
or	inhomogeneous	full	full
	inhomogeneous	empty	A
	full	inhomogeneous	full
	empty	inhomogeneous	B
and	inhomogeneous	full	A
	inhomogeneous	empty	empty
	full	inhomogeneous	B
	empty	inhomogeneous	empty
minus	inhomogeneous	full	empty
	inhomogeneous	empty	A
	full	inhomogeneous	¬B
	empty	inhomogeneous	empty

Table 3: Trivial combination rules

Memory Consumption

The memory consumption is comparable to the B-Rep since the same lists have to be stored.

Rendering

BSP trees are used for the fast rendering of scenes with correct occlusion. For this purpose an algorithm is used which is called Painter's algorithm and it works the following way:

To render the polygons from back to front (seen from a specific view point) the tree has to be traversed in the correct order. First all polygons are drawn which are not in the half-space where the view point is (recursion on either the left or right subtree). Then the polygon of the root node is drawn. Finally all the polygons in the half-space of the view point are drawn (recursion on either the right or left subtree). With this technique all the nodes in the tree are traversed only once and the occlusion is correct for all polygons.

If the view point is in front of the separation plane of polygon P_A:

1. Render of A^-
2. Render P_A
3. Render of A^+

Else (if the view point is behind the separation plane of polygon P_A):

1. Render of A^+
2. Render P_A
3. Render of A^-

Polygons which are farther away from the view point are overdrawn by polygons which are closer to the view point.

Advantages

+ Transformations are trivial
+ Fast rendering with correct occlusion
+ Can be used to represent general objects
+ Model generation based on digitized objects is possible

Disadvantages

- Curved surfaces must be approximated
- Only convex polygons can be used
- High memory consumption (especially for curved and complex objects)

3.4.1 kD Tree

A kD tree is a special case of a BSP tree. In a kD tree the subdivision of the space is always axis-aligned. Each node only represents a separation plane. In contrast to that a root node in the BSP tree also represents a polygon which defines the separation plane. In the kD tree all the polygons are represented as leaf nodes. If a polygon is intersected by a separation plane then the polygon is cut into two pieces. In Figure 13 an example for a 2D kD tree is given.

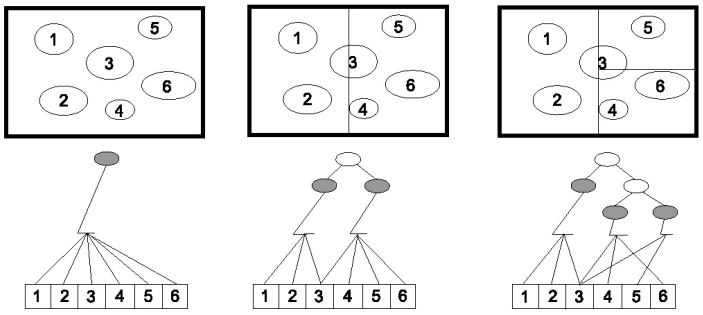

Figure 13: 2D example of a kD tree. In each tree level the subspace is separated by a axis-aligned plane into two subspaces.

The advantage of a kD tree in comparison to a BSP tree is the faster generation since the separation planes are axis-aligned and the intersection test with polygons is therefore faster. The disadvantage is that some of the positive properties, such as the simple inside test in the BSP tree, get lost.

3.5 Octree

Octrees are a volume model. They represent a volume directly in the data structure. The main advantage of this structure is the fast and easy combination of several objects. It is not necessary to do this during the rendering process. Therefore the rendering process becomes faster and easier.

Octrees are based on the principle of space subdivision. Each octree node represents a region in space. If necessary this region is subdivided into 8 subspaces. The subdivision is done through the center of the space along the three main axes.

Each leaf node of the tree represents one block. This block can be either empty (also called white W) or full (also called black B). If the representation by a block in one subspace is not accurate enough, it is further subdivided

(also called gray G). Figure 14 shows an example of a simple octree. On the left side the object which is represented by this octree is shown.

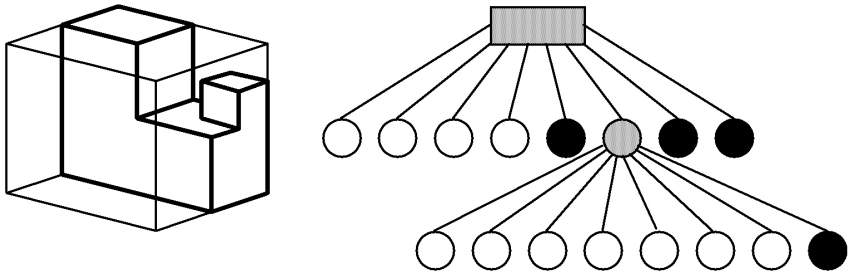

Figure 14: Example of a simple octree.

Data Structure

- *Object* = Pointer to the root node
- *Node* = Either full, empty or a pointer to 8 subnodes

Transformation

Transformations in general are hard to implement. The whole tree has to be constructed again after a transformation. Only specific transformations like a rotation by 90 degrees is smple.

Combinations

Combinatons are easy to process with an octree. The implementation only needs to process Boolean operations. Therefore the processing time of the combination is very fast.

Memory Consumption

Objects with curved surfaces and small details need many subdivisions to be represented accurately by an octree. This requires more memory.

Rendering

For the rendering of the octree an algorithm exists which processes the tree based on the current view direction. It starts to render the subspace which

is farthest away from the viewer. The algorithm ends with rendering of the subspace which is closest to the viewer. This guarantees that the occlusion is correct for all subspaces. In particular the rendering for one node of the tree looks like the following:

- If the node is empty (W), do nothing
- If the node is full (B), draw a cube
- If the node is further subdivided, order the leaf nodes according to the viewing direction and render them in the correct order (recursively)

A problem with the rendering of an octree is the blocky result since only cubes are rendered. This effect can be reduced by further subdividing the octree.

Advantages

+ Combinations can be processed fast and are very easy to implement
+ Rendering is simple
+ Spatial searches are fast
+ Model generation through digitization of real objects is easily possible

Disadvantages

- High storage demands for an accurate approximation of complex and curved objects
- Transformations are hard to implement
- General objects can not be represented exactly

3.5.1 Extended Octree

A possible extension of octrees is the introduction of new leaf nodes:

- Face nodes
- Edge nodes
- Vertex nodes

This extension allows to represent objects with more details without subdividing the octree too often. This saves memory. The generation of the extended octree from a list of polygons works as follows:

- Generate a list of faces and a list of edges (similar to the B-Rep in Section 3.3)
- Separate these lists in 8 different lists for each octant by clipping with the borders between the octants
- For each octant:
  - If both lists are empty: the node is either full or empty
  - If the vertex list contains only one vertex: generate a vertex node
  - If the face list contains only two faces: generate an edge node
  - If the face list contains only one face: generate a face node
  - Else: Further subdivide the octant (recursively)

Figure 15 shows the three additional nodes for the extended octree.

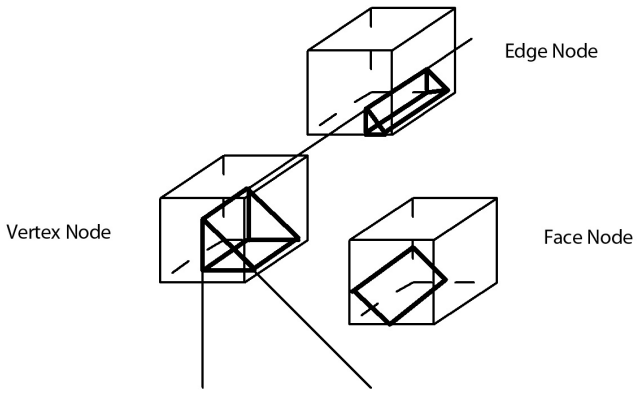

Figure 15: Additional nodes for the extended octree.

Data Structure

- *Object* = Pointer to the root node
- *Node* = Either full, empty, edge node, vertex node, face node or pointer to 8 subnodes

Transformation

To transform an extended octree it is best to convert the octree into a B-Rep and then transform it. After that a new octree can be generated based on the transformed B-Rep.

Combinations

The combination of two extended octrees is done in the following manner:

- Is the node of octree A or B either full or empty then the Boolean operations can be applied
- Is the node of octree A and B further subdivided then all subnodes are combined (recursively)
- Is the node of octree A either an edge, vertex or face node and the node of octree B is further subdivided then the subnodes of B are combined with the node of A (recursively)
- Are both the nodes of octree A and B either an edge, vertex or face node then the resulting combinations are calculated geometrically

Memory Consumption

The memory consumption for the accurate representation of complex objects is lower in comparison to the usual octree.

Rendering

For the rendering the extended octree is converted back to a B-Rep. The vertex nodes are used to reproduce the vertex list. The face and edge nodes are used to reproduce the face list.

Advantages

+ Memory consumption is lower (in comparison to the usual octree)
+ Can represent general objects better

Disadvantages

- Has to be converted into a B-Rep for rendering
- Combinations of two extended octrees are more complicated (in comparison to the usual octree)

3.5.2 Octrees as Spatial Directories

Octrees are often used as a search structure for different objects combined in a scene. This is for example important for computer games when collision detection is needed. In a tree structure the nearest neighbor to the current object can be found very fast.

For the underlying data structure of the object various representations can be used. The depth of the octree depends on the desired number of objects in a subspace. If an object is part of more subnodes then a pointer in each of these nodes is referring to the object. Figure 16 shows an example of such an octree. The nodes of the octree in this case refer to objects of a B-Rep.

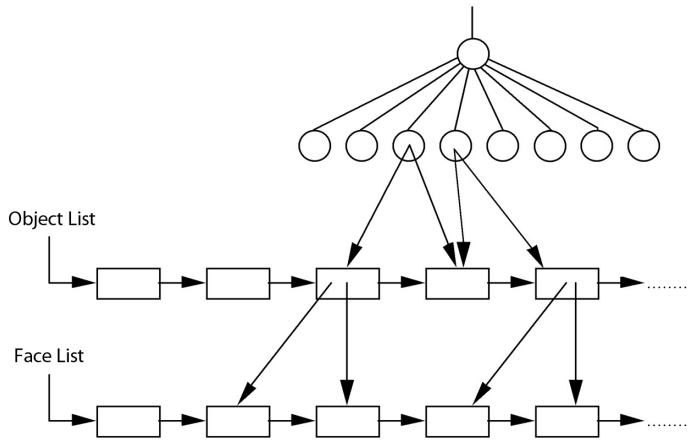

Figure 16: B-Rep represented in an octree.

3.6 Constructive Solid Geometry Tree

A Constructive Solid Geometry Tree (CSG Tree) is a binary tree. The intermediate nodes in the tree represent logical combinations. The leaf nodes represent primitives and their transformations. Figure 17 shows an example of a CSG tree. As primitives only a few elementary objects, such as sphere, block, cylinder or sweeps, are necessary to represent complex objects by the usage of transformations and combinations.

Data Structure

- *Object* = Pointer to an object
- *Node* = either

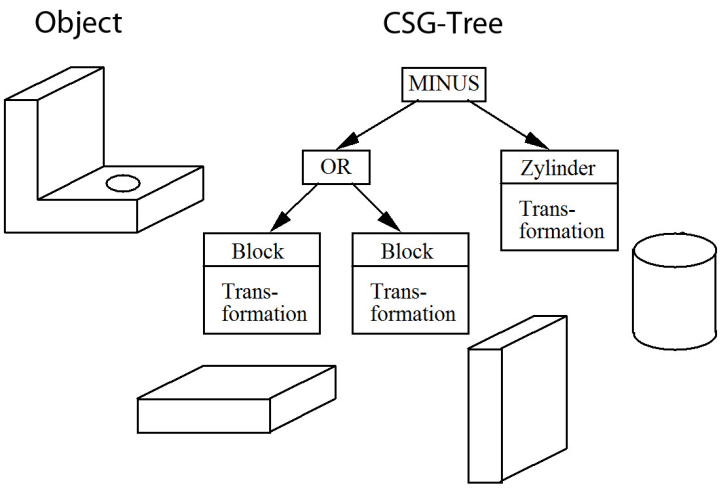

Figure 17: Example of a CSG tree.

- *Intermediate node*: Combination of two objects
- *Leaf node*: Primitive and transformation

Transformation

An object is transformed by adding the transformation to the transformation at each leaf node.

Combinations

Two objects represented by two CSG trees can easily be combined by attaching both trees as subtrees to a new root node with the desired combination operation.

Memory Consumption

For a CSG tree only the structure of the tree with its combinations and the transformations of the primitives have to be stored. The transformations are stored as transformation matrices. The CSG tree is an efficient way to store the exact representation of complex objects.

Rendering

The rendering of a CSG tree is directly not possible. One possibility to render a CSG tree is by converting it into a B-Rep. Through this step the exact

representation gets lost as well as the memory consumption increases. But it can be rendered fast after this conversion.

Another possibility is to use a raytracing algorithm for the rendering. The raytracing directly uses the CSG tree structure. Therefore it is not necessary to convert the CSG tree into any other representation. The disadvantage of the raytracing is the slower rendering speed in comparison to a fast rendering based on B-Reps.

Advantages

+ Low memory consumption
+ Can represent objects exactly
+ Combinations and transformations are simple
+ Fast spatial searches due to the tree structure

Disadvantages

- Rendering techniques are complicated and slow
- It is not easily possible to generate a CSG tree based on digitized data (e.g., from a 3D scanner)

3.7 Bintree

The bintree is used as structure for spatial searches. The name bintree originates from binary tree. The root node of the tree subdivides the space into two subspaces. The two child nodes of the root node subdivide the two half spaces again. Figure 18 shows an example for the construction of a bintree. All the subdivision planes are perpendicular to one axis (x, y, or z). The axes are alternated for each tree level (e.g., xyzxyz...).

In comparison to the octree, which also can be used as spatial directory (see Section 3.5.2), the subdivision does not necessarily separate the space into two equally large subspaces. This property helps to balance the tree better in the case the objects, which should be represented by the bintree, are not distributed equally in space. A better balanced tree also needs less nodes and therefore a lower memory consumption.

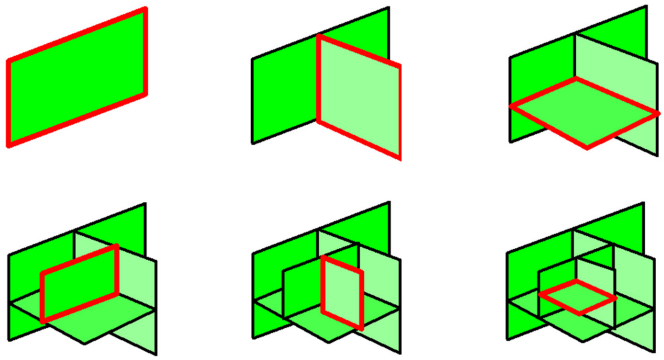

Figure 18: Construction of a bintree.

Data Structure

- *Object* = Pointer to a node
- *Node* = either
  - *Intermediate node*: Subdivision position
  - *Leaf node*: Geometry (e.g., vertex in a B-Rep)

The subdivision position is a value between 0 and 1. If it is 0.5 then it separates the space exactly into two equally large subspaces.

3.8 Grid

A regular subdivision of the space in the form of a 3D grid can also be used as a structure for spatial searches. In contrast to trees this structure has the advantage that there is only one level. In this level all the cells of the grid can be addressed directly. Therefore a grid is especially good in finding neighboring cells. Since there is only one level in the structure the processing time for this search is constant.

The grid structure is very flexible in storing different kind of data at each grid point. It is possible to use the grid for spatial indexing of some geometry. This is similar to Section 3.5.2 where octrees are used as spatial directories. Furthermore each grid point can store one or more values of some volumetric phenomena at the corresponding position. In this case the grid structure is also referred to as volumetric data set. An example for such a data set is Computed Tomography (CT) where each point in the grid describes the physical density at a certain position in space. Another example is a flow

simulation. In this case it is possible to store a vector at each grid point which represents the flow direction and strength at this position.

No matter for which purpose the grid is used, a problem with the grid structure is the critical choice of an appropriate cell size. If the size of the cells is too large the grid is not able to represent an object accurate enough. If the size of the cells is too small the whole grid needs high memory consumption.

One solution to prevent this problem is a hierarchical grid. In this case the base structure of the grid consists of larger cells. For subspaces where a higher resolution is necessary smaller cells are used to represent these areas. In Figure 19 a 2D example for a hierarchical grid is given.

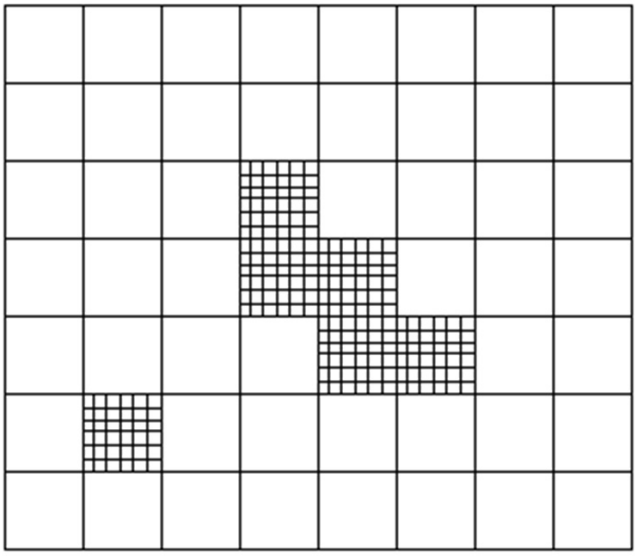

Figure 19: Hierarchical grid.

Data Structure

- *Object* = Array of grid points
- *Grid Point* = Value(s) or reference to a geometry

Through the dimension of the grid (x, y, and z) and the array index of a grid point, it is possible to restore the 3D position of a grid point. Therefore it is not necessary to store the spatial position of each grid point. This reduces the memory consumption and makes it easier to traverse the data. In a hierarchical grid the resolution for each block must be stored to restore the correct position of each grid point.

Transformation

Transformations of a grid are difficult in general. Only simple transformatons like rotations by 90 degrees are easy to compute. For all other transformations the whole grid has to be re-computed.

Combinations

The grid structure does not contain any geometrical information of the object. Hence it is not possible to apply logical operations based on the shape and position of objects. But it is possible to apply operations like MAX, MIN or DIFF on a cell level. To process these operations both grids have to have the same grid resolution. If this is not the case one grid has to be converted to fit the resolution of the other grid.

Memory Consumption

For the grid structure the memory consumption is only dependent on the resolution of the grid. The smaller the cells, the larger the grid and the higher the memory consumption.

Rendering

The grid structure is especially suitable for algorithms which have to frequently sample the neighboring cells during the rendering process. In the grid structure this sample process can always be computed in a constant time, no matter at which location inside the grid. An example for such an algorithm is raycasting.

Advantages

+ Constant processing time of all cells
+ Memory consumption is directly dependent on the resolution
+ Can be used to represent an object coming from a digitizer

Disadvantages

- Contains no geometrical information of the object
- Memory consumption can be high for an accurate representation of an object

- Transformations are not trivial to apply

References

[1] Java Applet for BSP Tree Construction. http://symbolcraft.com/graphics/bsp/index.php, January 2011.

[2] Java Applet for kD Tree Construction. http://homes.ieu.edu.tr/~hakcan/projects/kdtree/kdTree.html, January 2011.

[3] Hanan Samet. Spatital data structures. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, 2007.

[4] Engineering Specifier. http://www.engineeringspecifier.com/Design-Software/CCEs-EnSuite-Delivers-Support-for-Point-Cloud-Files.asp, January 2011.

[5] Engineering Specifier. http://cse.csusb.edu/tongyu/courses/cs520/notes/mesh.php, January 2011.