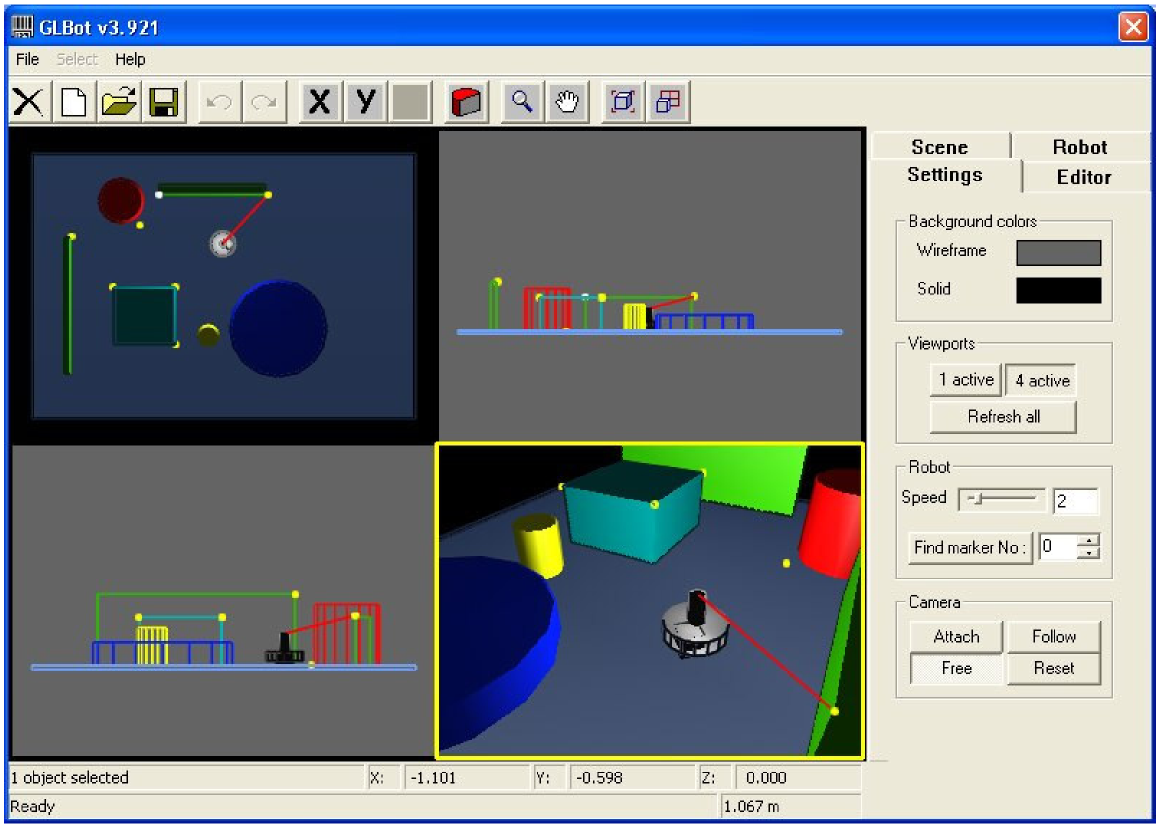
Graphics acceleration techniques for a mobile robot simulator

Tamas Juhasz
jt123@hszk.bme.hu

Department of Control Engineering and Information Technology,
Technical University of Budapest



Abstract

Today there is a wide variety of mobile robot simulators on the market. All have a common feature: some information coming from the virtual environment is placed at the user's disposal. The simulators are claimed to realize a range-finder (a sonar or a laser~) and / or a vision system (for example a CCD camera). Usually an implemented laser range-finder utilizes the forward ray-tracing (ray-shooting) mechanism to measure the current distance. This paper reviews the ray-shooting and collision detection acceleration algorithms to be applied in a mobile robot simulator.

Keywords:

robot simulator, range-finder, ray-tracing, space partitioning, collision detecting

1 Introduction

The ***GLBot***® mobile robot simulator is under development at our department. The ***GLBot*** name is an acronym of Open**GL** and ro**bot.** It supports research on landmark-based navigation algorithms. The user can control the robot's behavior in the widely-spread C language. The implemented simulator runs under Windows operating systems (Windows 98 or later) with OpenGL extension enabled. The platform of implementation is the standard Windows Application Interface (Windows API) and the Microsoft Visual C++ 6.0 Integrated Development Environment. The Java language could have been another option, but current Java Virtual Machines are not fast enough to run a 3D application on an average IBM PC.

The most of today's common graphics accelerator cards support the required rasterisation feature of the OpenGL standard through the bundled device drivers. The other wide-spread graphics API is Direct3D and DirectDraw; both are members of the DirectX family where 9.0 is the newest version. Although it is much more complicated than the OpenGL API, it is developing faster. The newer graphics cards support hardware transformation and lighting (HWTL, by means of so called Graphics Processing Units) which is enclosed by DirectX 7.0 or greater. Only the OpenGL 2.0 standard includes the usage of HW T&L, but except for a few cards OpenGL 2.0 is not supported, yet. The simulator itself supports currently only the OpenGL 1.1 standard. In the future there will be a Direct3D option, too. The third (unacceptable) option is software rendering (embedded in the operating system) which degrades the performance and sometimes gives images of poorer quality.

The imaginary robot is equipped with a range-finder. Its functionality – the distance measurement – is carried out by adopting the forward ray-tracing method. This quantifies the "length" of the emitted laser beam from the emitter to the first intersection point. Besides distance measuring the ray-tracing method can be used to select objects with the mouse in the virtual scene editor in which the obstacles are constructed. In an orthogonal viewport the ray starts from the distance and traverses through the pixel which is under the mouse cursor (parallel to the view plane's normal). The case of a perspective view is similar to the previously mentioned one, but the current ray is started in the eye position. The first object that is hit by the ray will be selected. This selection method is not time-critical. Contrarily, each time the robot's position or orientation changes, the current measured distance needs to be recalculated by using the ray-shooting algorithm. Implicitly the individual object-ray intersection tests can hardly be made faster. But the number of those objects which are relevant to the current ray can be decreased radically with some spatial data structures. When hundreds or thousands of objects are in the scene, good data structures can result 10x-100x speed-up in ray-scene intersection tests. Widely-spread data structures are hierarchical bounding boxes, grids, octrees, k_d-trees and BSP trees. The efficiency depends on how many objects are affected before the firstly intersected is found, and how many steps in the spatial data structure are needed to achieve this. [AK89]

2 Spatial data structures

2.1 Hierarchical bounding volumes

When the virtual scene contains complex objects (in case of a robot simulator these objects could be the obstacles), the ray intersection tests can be accelerated by using this algorithm [Got96]:

- wrap the complex object with a simpler one that contains it entirely
- if the ray does not hit the simpler object, it is sure-fire that the complex one is left untouched, too
- at high-level, bounding volumes can be organized to new bounding objects, forming a hierarchy
- if the ray avoids a higher-level bounding object, all of the contained bounding volumes and scene-objects can be cast away

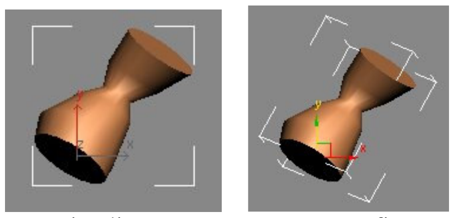


Figure 1 – Types of bounding boxes

Most common bounding objects are spheres or boxes. Axis-aligned bounding boxes can poorly approximate the set they are bounding, leaving large "empty corners". Even if they are the easiest to construct, oriented bounding boxes sometimes pay for the overhead (see Figure 1) by giving a snug fit.

Hierarchical bounding volumes give best results in hierarchical scenes where the bounding objects follow the hierarchy of the scene objects.

2.2 Equidistant grids

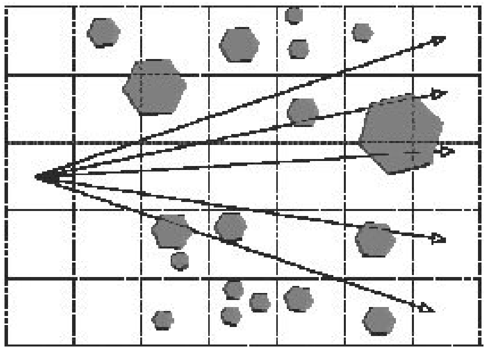


Figure 2 – An equidistant grid

In this case, the 3D space contains cells of equal shape and size in a regular arrangement. Each cell contains a list of pointers referencing those objects that are intersecting it. Calculation method:

1. the starting point of the ray identifies a cell → current cell
2. search for intersection in the current cell
3. if no intersection found, locate the next cell by using some kind of 3D line-drawing algorithm, than go back to 2nd step
4. if there is an intersection, return the closest value
5. if the ray passes a cell and hits an object, it does not always mean, that the first intersection point is located in that cell !

The grid method is effective only in homogenous scenes (when objects are scattered in a balanced way) [Gla84] If case of a non-homogenous environment, where the objects are lying about in the scene, stepping through the empty cells can be time consuming. The accurate number of subdivisions usually needs some a-priori information about the scene. If fewer cells are used, then there are many objects per cell, which degrades the overall performance. Too many (mostly empty) cells lead to the same phenomenon which is discussed previously at non-homogenous scenes.

2.3 Octrees

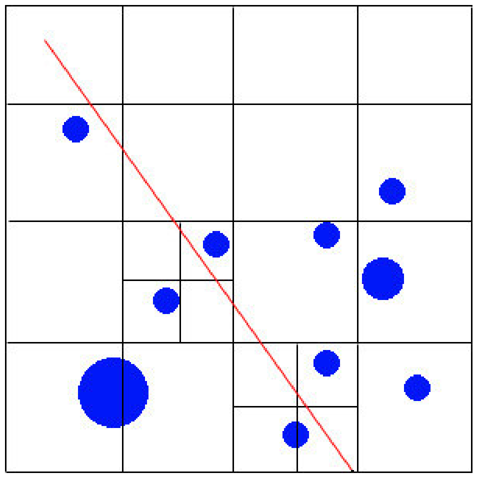


Figure 3 – Top view of an octree (it is called as quadtree) with adaptive segmenting

This method is similar to the previous one: in this case the space is also divided into cells. [Nay93] A cell is divided into 8 equal sub-cells with 3 perpendicular cutting planes, each are bisecting the ancestor cell. The planes are axis-aligned (see top view on Figure 3). Each cell has a list of the contained objects. The cell subdivisions last until a given lowest number of enclosed objects is reached or the depth of the node- tree exceeds a limit. This method adapts very well to non-homogenous scenes. The ray traversal is more complicated, because the resultant cells have different extents:

- if the cell is a non-empty leaf in the node-tree, then return the closest intersection point in that cell
- otherwise, if the ray strikes into a cell that has children, then apply recursion on the 8 sub-cells in a specific order
- if there isn't any intersection in any sub-cells, then find the next cell with the following method. Calculate the position of the exit point. Add an infinitesimal value to the ray parameter which is valid at that point. Traverse the tree structure with the new point starting at the root node to find the cell that contains it. (The whole procedure can be very time consuming if the adjacent cells are at different level.)
- go back to the first step

2.4 K_D trees

The k_d trees (in *k* dimension, where *k* is usually 3) have similar rules as the octrees [AK89]. They offer a relatively efficient way of point location queries. Let's consider ***n*** objects in *k* dimension. The *k* dimensional space is divided into cells with axis-aligned cutting planes which mean not necessary bisecting, but they can split at any point. In case of octrees, always 3 perpendicular planes (parallel to *xy*, *xz* or *yz* planes) are used per cell. In k_d trees there is only one cut per node (with a lower dimensional hyperplane). Each node in the tree is defined by the plane that partitions the set of objects (through one of the dimensions) into left/right sets, halving the object-set of the parent node. These children are partitioned again into equal halves, using planes through a different axis.

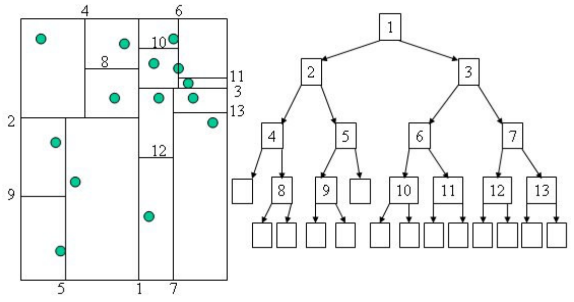


Figure 4 – Subdivided 2D scene with the corresponding *2d* tree

Partitioning stops after log(***n***) levels, with only one object per a leaf cell. Alternate k_d-tree construction algorithms insert objects incrementally and divide the appropriate cell. Such trees can become seriously unbalanced, which degrades traversal performance. A balanced 2d tree can be seen on Figure 4.

The structure of a node is the following:

- children pointers
- (parent pointer) → good for the tree-traversal
- extents of the cell: (x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})
- list of pointers referencing the contained objects
- (neighbour pointers) → typically used in leaf-nodes, eases finding the next cell for the current ray

The K_d trees complexity of $O(\boldsymbol{n}\ log(\boldsymbol{n}))$ in average which is better than that of the BSP trees' $O(\boldsymbol{n}^2)$.

2.5 BSP trees

[SS92] The **B**inary **S**pace **P**artition trees are generalizations of the k_d trees, but the splitting planes are general (*k-1*) dimensional hyperplanes, no axis-alignment is required. They could be defined as a sequence of splits that divide the space into non-overlapping parts. K_d trees can be considered as axis-aligned BSP trees. This model was first used by Naylor and Fuchs for the Painter's algorithm in 1980 (at that time there was not any Z-buffer hardware that could resolve the problem of overlapped polygons at rendering time). The Painter's algorithm utilizes the fact that farther polygons can not hide those that are closer.

A two dimensional plane can be described with the following equation:

$$Ax+By+Cz+D = 0$$

Where [*A,B,C*] is the normal vector of the plane. Let's consider a P point in the 3D space. By substituting its three known coordinates into *x*, *y* and *z*, we can make sure the point is on the plane, if and only if the previous formula gives zero value. If it gives a positive number as a result, then the point is on the front side of the plane, otherwise it is on the back side. The 3D BSP tree is built up from nodes. Each node is representing a 2D plane with its 4 parameters. The algorithm for building a BSP tree is the following [Chin95]:

- if there are more than one objects / polygons in the space, select a plane for partitioning
- cut the current set of objects with that plane into two parts
- recurse with each of the result sets

The way of choosing a cutting plane is dependent on how the tree will be used, and what sort of efficiency criteria is used. It is desirable to have a balanced tree, where each leaf-node is at the same level and they contain approximately equal number of objects. However, it takes much time to achieve this, it is worth in case of static scenes. Namely at this moment, the tree has to be constructed only once, and can be stored in the scene descriptor file.

The efficiency of a BSP tree

Space complexity

[SHBS02] For the problem of ray-polygon intersection testing, consider a set of *n* parallel polygons, and the set of *m* partitioning planes, all of which are perpendicular to the polygons. This has the effect of splitting every polygon with every partition. The number of polygons resulting from this partitioning scheme is $n + (n * m)$. If the partitioning planes are selected from the candidate polygon set (an auto-partition), then $m = n$, and the expression reduces to $n^2 + n$. Thus the worst case spatial complexity of a BSP tree constructed using an auto-partition is $O(n^2)$. It will be extremely difficult to construct a case which satisfies this formula. The "expected" case, repeatedly expressed by Naylor, is $O(n)$.

Time complexity

The time complexity of ray-shooting using an auto-partition BSP tree is the same as the spatial complexity, that is a worst case of $O(n^2)$, and an "expected" case of $O(n)$.

3 Individual ray-object intersection tests

After reducing the number of those objects that are relevant to the current cell / subspace, the intersection point has to be calculated by using some mathematics. The closest intersection point serves the measured distance for a robot simulator. Without demanding completeness, the following types of objects are discussed in this paper: a box, a sphere, a cylinder and a triangle-mesh, respectively. The former three are the most common types of bounding volumes. If an object is transformed with a homogenous ***A*** transformation matrix, then the inverse matrix is $\boldsymbol{A}^{-1}$. The ray has to be transformed with $\boldsymbol{A}^{-1}$, and continue the calculations in the local coordinate system of the object.

3.1 Boxes

First we have to decide whether the ray hits the bounding object at all. If we are sure of the negative case, there is no need to calculate any intersection points with the contained object. Although the ray is a half-line in 3D space, we can approximate it with an extremely long 3D line-segment.

The 2D Cohen-Sutherland line clipping algorithm [SzKL99] can be extended to the third dimension. The basic concept is to classify each 2D point with a 4 bit length integer bit-code. The 2D cutting region is bounded with 4 (2-2 parallel) lines (Figure 5).

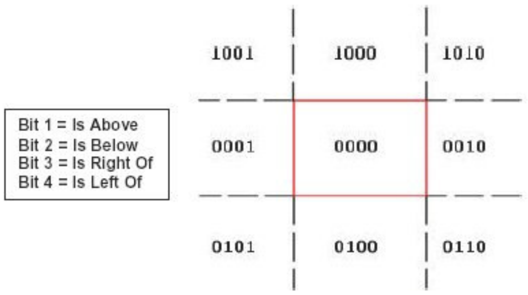


Figure 5 – The 2D Cohen-Sutherland clipping bit-codes

Each bit represents the point's location according to each cutting lines. For example: if the first bit is zero, then the point is below the upper cutting line, the second one stands for the bottom line, etc. Both ends of a line section are classified as C_1 and C_2. There are two basic cases for rejection / acceptance:

- if C_1 & C_2 != 0, then both endpoints are on the same external side of a clipping line, so it must be rejected
- if C_1 | C_2 == 0, then the whole line-segment is internal, so there aren't any intersections (this case can not happen, because our ray's line segment is infinite)

Otherwise the need for calculating intersection cannot be cancelled. The 3D generalization is the following: the bit-code is 6 bits long, representing upper, lower, left, right, front and back cutting planes. The classification of each endpoint is the same as described previously: it requires only 6 comparisons per vertex. The same "C_1 & C_2 != 0 " rejecting condition is valid in this case, too.

If the ray-object hit is bound to happen (neither holds of the trivial acceptance / rejection cases), there's no way to avoid calculating intersections. In local coordinate system, a box is bounded by 3 pairs of axis-aligned planes.

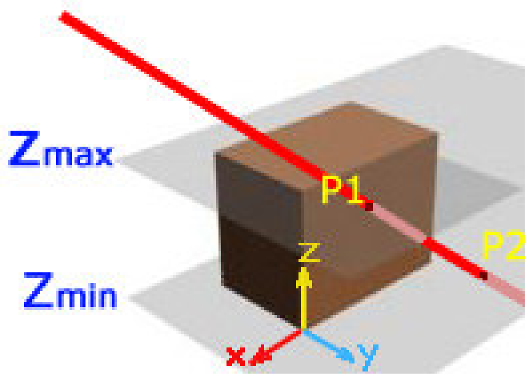


Figure 6 – Intersecting the top and bottom plane of a box

We should calculate t_{in} and t_{out} values for each 3 pair of planes. Then these parameters have to be sorted in an increasing order. If all t_{in} values are ahead of the t_{out} values in the list, then there is an intersection at the least t_{in} value.

3.2 Spheres

A sphere has the following equation in local coordinate system:

$$x^2 + y^2 + z^2 = R^2 \tag{1}$$

A ray has the following formula:

$$r(t) = s + t \cdot d \tag{2}$$

where "s" is the (inverse transformed) start-point, "d" is the unit direction vector and t is the parameter. The (2) vector equation can be transformed to three scalar equations as follows:

$$\begin{aligned} r_x &= s_x + t \cdot d_x \\ r_y &= s_y + t \cdot d_y \\ r_z &= s_z + t \cdot d_z \end{aligned} \tag{3}$$

Substitute r_x, r_y and r_z from (3) with x, y and z respectively into (1), and reorder it to zero. This results a quadratic vector equation in *t* parameter:

$$(d \bullet d)t^2 + 2(d \bullet s)t + (s \bullet s) - R^2 = 0 \tag{4}$$

If the determinant of the equation is negative, then the ray does not hit the sphere. If it is zero, then the ray touches it tangentially. Otherwise there are two crossing points with two positive *t* values. The smaller one is the distance from the ray's emitter point.

3.3 Cylinders

An infinite cylinder-side has the following equation:

$$x^2 + y^2 = R^2 \tag{2}$$

("R" is the radius of the cylinder)

By substituting the x,y,z variables with the **r** vector's coefficients from (1) formula, we can rearrange (2) to get a quadratic equation with parameter *t*.

If the quadratic equation has no solutions (negative determinant), that means the ray does not hit the side of the cylinder. If there is one solution (the determinant is zero) that means the ray hits the surface tangentially. Otherwise there are two intersection points at t_1 and t_2 parameter values.

We have to find intersection points with the upper and lower base planes of the object. These give us the t_3 and t_4 parameter values. Next we have to make sure these points are really internal points of the base disks.

Finally we have got $t_{1\ldots4}$ values (some of them may be missing if there were not any solution of that proper equation). We have to choose the least positive value, substitute it into (1), so we get the first intersection point, too.

Other parametric implicit surfaces can be handled the same way as cylinders or spheres. The ray's equation must be substituted into their given formula, and reordered to form an equation /similar to (4)/ with only 1 unknown quantity: the *t* parameter. This equation can be solved with numerical root-finder algorithms.

3.4 Meshes

Mesh objects are made from connected triangles. A given edge between two adjacent vertices is usually shared by two neighbouring triangle faces. Every obstacle can be approximated with this type of objects. Those triangles that are facing their back side to the ray will be dropped off during ray tracing. This hidden surface removal [DKW85] can be accelerated with a BSP tree. A ray hits a triangle's plane if its direction vector is not perpendicular to the plane's normal. If an intersection point is found, then it has to be compared with each sides of the triangle to check if it is an internal point. If there are more triangles in a cell, they have to be tested against the ray one-by-one.

4 Collision testing in the robot simulator

Mobile robot simulators utilize another important area of 3D game engines: the collision testing. If a robot-obstacle collision occurs, it must be detected, and the user must be informed about that. The main goal is to avoid interpenetrations and responding to collision too early. When fast moving objects exist (in this case only the robot can move fast, because the environment is static) collision detection must be evaluated a lot of times per second. [Lin93] Let's assume that every obstacle and the robot itself are represented by closed convex polyhedrons. If they are colliding, at least one face of one object is penetrating into a face of the other one. An extremely simple and slow method is the following: cycle through each object in the scene and test if they collide with the robot, i.e. if there is an obstacle-polygon that intersects any of the robot's polygons. Let's consider a simple scene with 5 objects, each having 100 polygons, and let our imaginary robot contain 50 polygons. The simplest algorithm leads to 5*100*50=25,000 polygon-polygon intersection tests per cycle, which has to be evaluated at least ten times per second to avoid large interpenetrations. The final result is 250,000 polygon-polygon tests per second, which cannot be carried out. We have to appeal to some computer graphics algorithms to accelerate these procedures.

4.1 Naïve collision testing

Let's approximate the robot with a vertical cylinder. In this simple case, the obstacles are represented with their bounding cylinders or boxes. Thus cylinder-cylinder and box-cylinder collision detection methods have to be developed. In case of overlapping bounding volumes, the contained objects can be matched to find out whether they are really colliding or not.

The cylinder-cylinder case

We assume both cylinders are vertical. They are overlapping, if their centre-to-centre distance is smaller than the sum of their radiuses (Figure 7).

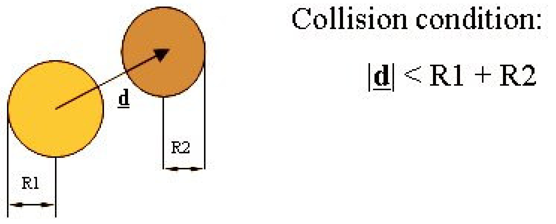


Figure 7 – Cylinder-cylinder collision

To avoid using square root function, the same fact is true for the squared distance and sum of squared radiuses.

The cylinder-box case

We assume the cylinder object is upright, and the box bounding volume is aligned to the Z axis, too. First attach the cylinder to the box as a child object. Rotate the box around the Z axis to be fully axis-aligned. Translate the box with the centre of its bottom face to the origin (only needed to ease the conditional tests). Draw an outline around the box with a radius of the cylinder, the result can be seen on Figure 8.

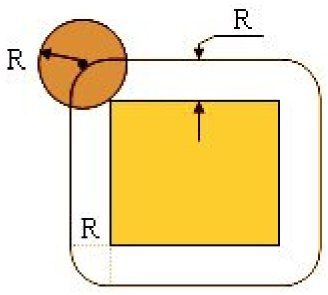


Figure 8 – Box-cylinder collision

The cylinder is overlapping the box, when its centre point enters the outline shape. This polygon is constructed from 4 circles with radius R at each corner and 4 rectangles with a width of R.

4.2 Grid method

A regular 2D grid is attached to the robot, with spacing equal to its diameter. If the robot is moving, the grid translates with it, too. The cylindrical robot can hit only those objects, whose base plane-projection intersects any of the 9 surrounding grid cells.

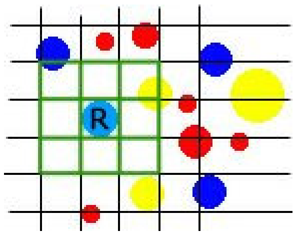


Figure 9 – The active cells around the R robot

The "active" cells are highlighted on Figure 9: instead of 12, only 2 objects are relevant to the current robot position (R). At next step the squared relative distance of the candidate objects' bounding cylinders is calculated using the Pythagorean formula. If this value is greater, than the radius of the robot's cylinder squared plus the radius of the object's bounding cylinder squared, then the robot does not hit that candidate object. Spheres can also be used as bounding objects, but they offer a rough estimation only: it is frequently encountered that the bounding spheres are overlapping while the objects aren't.

4.3 BSP trees

[Wade97] Let's consider two non-overlapping convex polyhedrons. It is a true fact, that one can always find a dividing plane between them. *N* number of tree nodes allow search operations to complete in log(*N*) time, and collision detection with a BSP tree is basically a search. If we are detecting a collision between two objects ***A*** and ***B***, and if we know that ***A*** lies entirely on one side of some plane P that cuts through ***B***, then we need only test ***A*** against the parts of ***B*** that are on the same side of the plane as ***A***. So if we recurse down the BSP tree of ***B***, finding whether ***A***'s bounding sphere intersects each separating plane (a very cheap operation in itself), we can ignore many polygons of ***B*** that have no chance of colliding with ***A***. For each polygon of ***B*** that passes this filter, we will call a procedure that compares it with every polygon in ***A***. After selecting potentially colliding polygons from ***B***, rather than testing them against all of ***A***, we can drop them down ***A***'s BSP tree, eliminating collision tests with much of ***A***. By the time we're done with all of that, we should end up performing relatively few polygon-polygon intersection tests.

5 Application

We considered many acceleration techniques in our mobile robot simulator, and came to the following conclusion: the BSP tree's diversified features are suiting our project most of all.

The BSP tree is constructed in advance as it does not change during the simulation because the scene is static. It accelerates both ray-shooting (with its backface-culling, hidden surface removal, and ray-order object-sort features) and collision detection at the same time without extra construction and memory usage overhead.

6 Summary

Some important computer graphics algorithms have been surveyed in this paper. They are useful to achieve performance improvements for the GLBot® mobile robot simulator application which is currently under development by the author. Thanks to the utilized BSP tree's versatility, the program does not need extra memory to separate data structures handling the collision detection and ray-shooting acceleration. If only ray-shooting was the topic of acceleration, the axis-aligned k_d-trees would be better choice instead of BSP trees because they offer an easier and faster way of ray traversal.

7 References

[AK89] J. Arvo, D. Kirk. "*A survey of ray-tracing acceleration techniques*" In Andrew S. Glassner editor, "*An Introduction to Ray Tracing*", Academic Press, London, 1989.

[Chin95] N. Chin, "*A Walk through BSP Trees*", *Graphics Gems V,* AP Professional, pp 121-138, 1995.

[Dev89] O. Devillers. "*The macro-regions: an efficient space subdivision structure for ray tracing*". In *Eurographics '89*, pp 27-38, 1989

[FKN80] H. Fuchs, Z.M. Kedem, B. Naylor, "*On visible surface generation by a priori tree structures*", Proceedings of the 7th annual conference on Computer graphics and interactive techniques, pp 124-130, 1980

[Gla84] A. S. Glassner, "*Space subdivision or fast ray tracing*". IEEE Computer Graphics and Applications, AP, 1984.

[Gla89] A. S. Glassner, "*An Introduction to Ray Tracing*". Academic Press, London, 1989.

[Got96] S. Gottschalk, M. Lin and D. Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection", *Siggraph '96.*

[Lin93] M.C. Lin, "*Efficient collision detection for animation and robotics*", PhD thesis, Department of Electrical Engineering and Computer Sciences, Berkley, CA, 1993

[LM] M.C. Lin, D. Manocha, "*Efficient Contact Determination Between Geometric Models*" http://www.cs.unc.edu/~manocha/collision.html.

[Nay93] B. Naylor, "*Constructing good partitioning trees*", In proceedings of Graphics Interface '93, 1993

[O94] J. O'Rourke, "*Computational Geometry in C*", Cambridge University Press, 1994

[PML] K. Madhav, Ponamgi, Dinesh Manocha, and Ming C. Lin, "*Incremental algorithms for collision detection between solid models*", available at http://www.cs.unc.edu/~manocha/collision.html

[PY90] M. Peterson, F. Yao, "*Efficient Binary Space Partitions for hidden surface removal and solid modelling*", Discrete and Computational Geometry, 1990

[SS92] K. Sung, P. Shirley, "*Ray Tracing with the BSP Tree*", Graphics Gems III, 1992.

[SHBS02] L. Szirmay-Kalos, V. Havran, B. Benedek, L. Szecsi, "*On the efficiency of ray-shooting acceleration schemes*", Spring Conference of Computer Graphics, 2002

[SzKL95] L. Szirmay-Kalos, "*Theory of Three Dimensional Computer Graphics*", (editor), Publishing House of Academy of Sciences, 1995.

[SzKL99] L. Szirmay-Kalos, "*Számítógépes grafika (Computer Graphics)* ", *ComputerBooks*, 1999.

[Wade97] B. Wade, "*BSP Tree Frequently Asked Questions*" http://rtfm.mit.edu/pub/usenet/news.answers/graphics/bsptree-faq