University of West-Attica
"Modelling & Animation" Course

PW 2. Collision Detection

Get the corresponding zipfile and uncompress it. You will find two folders: *./lib* which contains some useful libraries and *./collisiondetection* which contains the main files of the application.

- collisiondetection.html: main file to open in a web browser compatible with WebGL (tested with recent versions of Edge, Firefox and Chrome).
- main.js: contains a creation routine called by the html code which creates the animation environment and provides a callback function which is called regularly (depending on the web browser, usually 60Hz).
- graphics.js: contains useful functions to manage WebGL display (initialization of graphics context).
- blinnphongshaders.js: shaders to provide a simple lighted display of graphics objects.
- WireframeBox.js: a class representing a geometric box to display in wireframe (useful to display bounding boxes).
- Polyhedron.js: class to create purely geometric polyhedra such as tetrahedra, cubes, octahedra, etc, as well as a single triangle (for test purpose) and convert them to WebGL *buffer objects* (faces are decomposed into triangles). Its constructor permits to select which polyhedron must be built.
- VirtualObject.js: main class to manage objects for the animation. Objects are geometrically defined by polyhedra (see Polyhedron.js). They can be positioned in space with a complex motion (the composition of several translations and rotations, associated with a velocity). The obtained trajectory roughly corresponds to an inclined ellipse. This class also includes all necessary routines to manage axis-aligned bounding boxes.
- Simulation.js: class for the environment of the animation, it manages the display of all the objects, the animation steps and… collision detection.

For this project, you only need to fill in methods in *VirtualObject* and *Simulation* classes. You can also change the number of objects to create. In function *startSimul()* in *main.js*, the last parameter of the *Simulation* constructor is the number of objects (the other ones are the sizes along *x*,*y* and *z* axes of the environment).

Note that, if you meet difficulties, it is always possible de avoid one of the steps (except the first one) and go on with the following step. The final result will be less efficient, but it can work. Some clues are given to circumvent each step (except the first one which is mandatory).

1) A simple (naive) approach for collision detection [trivial]

Find the method *computeAABB()* in *VirtualObject*. This method aims at computing an axis-aligned bounding box (AABB) for the current object. This box can be computed by finding the min and max coordinates of all the vertices along each axes and store them in *this.boxmin* and *this.boxmax* (this means that the bounding box is only represented by two vertices corresponding to its bottom/left/far and top/right/near corners).

1.1) *computeAABB()* method is partially filled with a loop that computes for each vertex of the current polyhedron, its position after a global positioning of the object in the simulation scene. This final position is stored in a temporary variable called *pos*. Using this variable, compute the AABB by finding the current min and max positions of vertices along each axis.

1.2) In the *Simulation* class, uncomment line *obj.drawAABB(...)* in *this.draw()* to display the objects' AABBs and check if they are correctly computed.

1.3) Complete method *compareAABB()* in *VirtualObject*. This method must return a boolean value telling if the AABB of the current virtual object (*this*) overlaps the AABB of the virtual object given in parameter (*that*). This method returns *true* in case of overlapping, and *false* otherwise.

Class *Simulation* includes a method called *step()* that is only partially filled. Currently, it just calls each virtual object to make it move one step further in time. You now have to complete this method to detect collision in order to display all colliding objects.

1.4) Complete this *step()* method by adding a new loop which considers each couple of objects (you can draw your inspiration from the first loop of the method which considers each single object). Do not forget that a couple of objects must not be considered twice: (A,B) and (B,A) are the same. For each couple, call *compareAABB()* method to check if the objects' AABBs overlap.

To visualize collision results of this step (and the following), you can change the *collision* attribute of colliding objects to *true*. This alters the colour of these objects and their bounding box the next time they are displayed (this allows you to directly visualize colliding object). For this first step, we display a collision when AABBs overlap (even if their corresponding objects do not intersect). More precise criteria for object intersection are used in further steps.

2) Broad phase [rather easy]

The method which considers each couple of objects to detect AABB overlaps needs to check *n*(n-1)/2* couples, where *n* is the number of objects. This requires computation time for a high number of objects. This section proposes to implement the "Sweep&Prune" (S&P) technique to optimize this step.

S&P supposes that all the bounding boxes are sorted along each axis (which means that we manipulate three lists). The sorting key is the min value of each box along the considered axis.

2.1) At the end the constructor of *Simulation*, create three lists (as attributes of *Simulation*) and sort objects along the tree axes (using *object.boxmin[current_axis]* as sorting key). It is advised to use a simple bubble sort algorithm. Although it is not efficient at this stage of the algorithm (which is only initialization, so we don't mind), it will be useful (and efficient) in animation steps.

2.2) In the *step()* method of *Simulation* class, after the first loop that computes the next step of each object, update the lists to take into account the new positions of object. Indeed, lists must be sorted again to take into account objects' motion. By relying on temporal coherence, we can consider that the lists are already nearly sorted (from previous step) so a bubble sort algorithm can be rather efficient at this stage.

We now want to take benefit of the sorted list to quickly find the colliding boxes. By iterating in a list, you consider each box and find the following ones in the list that overlap it. The overlap is here simple to find: it appears if the min value of one of the following boxes is less than the max value of the currently considered box (note that, since the lists are sorted, we know that the min value of boxes following a box are greater that the min value of this box). Once a box in the following boxes in the list does not overlap the current box, there's no need to continue walking through the list and check for other overlap for the current box. You can go on with the next box in the list (and check its own following boxes for overlap).

When walking through a list, you can find the couple of boxes that overlap on one of the axes. However, bounding boxes overlap if, and only if, they overlap on the three axes at the same time. One solution to deal with the three axes is to use a boolean triangular matrix (lower one for instance), which tells in *mat[i,j] (with j<i)*, for a given axis, if boxes *i* and *j* overlap. One matrix for each axis is filled for the two first checked axes, and only read when checking for the last axis to conclude if two boxes overlap (on the three axes).

2.3) At the end of the *step()* method in *Simulation*, change the loop you wrote in question 1.4) in order to exploit the sorted lists of bounding boxes and quickly find (and display) overlapping boxes.

Note: it is possible to avoid this step since it is only an optimization phase. You can still rely on the loop proposed in question 1.4), that checks for every couple of boxes to find overlapping ones.

3) Narrow phase [Difficult]

Previous steps permit to generate a list of couples of possibly intersecting objects (with overlapping AABBs). This does not necessarily mean that objets actually overlap. For this purpose, we need to find if triangles of these objects intersect. This step aims at finding couples of triangles that possibly intersect, that is, when their AABBs overlap. The exact intersection of triangles is the goal of the following steps. Note that, in this step and the following, we completely ignore the fact that an object may lie completely inside another one. No collision will be detected in this case (and we accept that).

Checking for all couples of triangles may be costly. A solution is to use a narrow-phase to quickly find overlapping triangle's AABBs. For this purpose, a binary tree of axis-aligned bounding boxes must be created for each object. The root node of this tree contains the AABB of the whole object. Each node of the tree corresponds to a set of triangles that is divided in two disjoint subsets (of similar size or number of triangles) supplied to the two children. Each node computes the AABB of the list of triangles to which it corresponds. Leaf nodes correspond to a single triangle and compute its AABB.

3.1) Build a tree for each object that associates a set of triangles to each node (all triangles for the root node). We propose a top-down building of the tree (from root to leaves): On a node, the set of associated triangles is divided in two subsets that will be given to child nodes and so, recursively, until lists with only one triangle give leaf nodes. To implement this routine, you can consider a general routine for *VirtualObject* (objects defined as a set of triangles) or dedicated construction for each kind of polyhedra in Polyhedron class, where it can be easier to subdivide, in a clever way, triangle sets into two distinct subsets.

Several general strategies are possible if you don't want to build a dedicated tree for each kind of polyhedron. At a given node, consider a plan that cuts the current bounding box along its greatest dimension and dispatch the triangles depending on which side of the plane their centre lies. Another strategy consists in computing the median of the centres of triangles along the greatest dimension of their bounding box and divide the set of triangles in two depending on the fact that they are less or greater than the median value. This last strategy permits to divide the current node's list of triangles in two lists with an equivalent number of triangles (so that the complete tree is balanced).

A hierarchy of bounding boxes must be updated after a motion of its object, that is, computing the bounding box associated with the set of triangles corresponding to each node of the tree (building a new tree depending on the new triangle positions is inefficient). Note the bounding box associated with a node is the min of the *boxmin* and the max of the *boxmax* values of its child nodes. A tree can therefore be updated by a bottom-up walk through the tree. Note that there is no longer need to compute the bounding box of a whole object explicitly (question 1.1)), since this box corresponds to the root box of the tree.

3.2) At the end of the *step()* method of *VirtualObject*, update an object's tree by computing the bounding box associated with each node of the tree in a recursive bottom-up walk (from leaves to root).

All the collision detection process now needs some updates. First, if step 2) is done, you use the root bounding boxes of the trees. Thereafter, when two objects AABBs overlap, you can use their trees to find the possibly intersecting triangles of these objects. This consists in walking through the trees of the two objects in the following fashion. When two nodes' AABB collide, compare their volume, keep the smaller one and only consider its overlap with the children of the bigger one (descend only in one tree and not both at the same time). Naturally if the bigger box is a leaf node, consider the children of the smaller one to test for overlap. You can go on, recursively, the same way. When both nodes are leaf nodes, then you have found potentially-colliding triangles.

3.3) At the end of the *step()* method in *Simulation*, use the bounding box hierarchies to quickly find possible couples of colliding triangles (which bounding boxes overlap) of couple of possibly-colliding objects. It is possible to modify *compareAABB* written in question 1.3), by considering tree nodes as parameters and not objects. Change the visualization so that objects including possibly colliding triangles are marked as colliding.

Note: it is possible to avoid this step. Add a method that computes the AABB of any triangle of an object and, in* step() *method in* Simulation*, add a double loop which considers each couple of triangles (of possibly colliding objects) and check the overlap of their AABB. This approach is not very efficient, but it works!

4) Exact Collision Detection [Difficult]

At this stage, a list of possibly colliding triangles has been computed. Now we have to check whether the triangles of these couples really intersect with an exact intersection detection.

Efficient methods, not focusing on edges and vertices and considering triangles as a whole, have been proposed, including Möller's one that has been described during the lecture. The original paper[1] can be found on the Internet, but it includes several optimizations that make the implementation more complex to understand, so <u>it is not advised to rely on it</u>.

4.1) Create a routine that implements the complete Möller's method. The first step consists in checking if the vertices of one triangles lie on both sides of the supporting plane of the other triangle. If ABC is one of the triangles, with normal **n**=(AB*AC), check the sign of AM.**n**, for each vertex M of the second triangle. If all the vertices have the same sign, no intersection occurs. Do the same after swapping the two triangles.

4.2) Complete this routine by considering edges that intersect the supporting plane (one vertex is positive and the other is negative). Compute their intersection with the supporting plane of the other triangle (see note below). This step should provide two points for each triangle (say P,Q for one triangle and S,T for the other). These points are the extremities of two segments, intersections of each triangle and the plane of the other triangle.

4.3) Finalize the routine by first computing the direction of the segments, defined as the cross product of the normal of the triangles **u**=(**n1*n2**). Then, consider the position parameter of each segment extremity, defined as PM.**u**, for M=P,Q, S and T. Using this position, find min and max parameters of each segment and check for collision (segments can be seen as bounding boxes with only one dimension, so AABB intersection test on axis **u** can be used).

1 T. Möller, "A Fast Triangle-Triangle Intersection Test", *Journal of Graphics Tools*, 1997.

4.4) At the end of the *step*() method of *Simulation*, when couples of triangles with overlapping AABBs are found, call Möller's algorithm then mark colliding objects (*this.collision=true*) only if there is really (at least) one of their triangles that is intersected.

Note: A naive but simpler approach to address this step consists in checking edges of one triangle against the other triangle considered as a face (and vice versa). For such purpose, the intersection algorithm is similar to the one used in ray-tracing (we consider an edge instead of a ray). If ABC is a face with normal n=AB*AC and DE an edge, then the intersection is found by solving n.(AM)=0 with M=D+l*DE, a point on DE which position is defined by a real parameter l. This gives a linear equation in l. After computing a solution (when it exists), it must be verified that 0<=l<=1, that is, M is inside the segment DE, and M belongs to ABC (by considering that the three quantities n.(MA*MB), n.(MB*MC), n.(MC*MA) are positive or null).