Real-Time Collision Detection and Response

FABIO POLICARPO[1,2] AURA CONCI[2]
[1]Paralelo Computação, http://www.paralelo.com.br, R. Otávio Carneiro, 100/907, Niterói, RJ, Brasil
[2]Instituto de Computação – UFF, R Passo da Pátria 156, 24210-240 Niterói, RJ, Brasil
fabio@paralelo.com.br , aconci@ic.uff.br

Abstract. A new collision detection and response algorithm for axis aligned bound box (AABB) is here presented. It is perfect visually also in very complex scenarios and extremely efficient on real-time application. It was tested for interaction with complex scene geometry made of convex polygons organized in a BSP tree. The algorithm is very useful for real-time 3d simulations and 3d games where the player and most of the dynamic game objects can be represented as AABB for collision detection. The algorithm presented is for the case of a AABB colliding with a convex polygon of any number of edges. The algorithm has been implemented in a game (where actors present complex behavior) and tested over a huge number of level geometry configurations.

1 Introduction

It goes without saying that good collision detection is critically important in a game. What does 'good' mean? Brute force polygon object/polygon object is too expensive and we must make some compromise for the sake of efficiency. Consider now the nature of the bounding box. There is a well known and well researched trade-off between the shape of a bounding volume, its bounding efficiency and its processing cost. It is also the case that in many demanding applications bounding volumes are arranged into hierarchies. In our case we have chosen to use AABBs (axis aligned bounding boxes) to bound dynamic objects and to embed them in a method that very quickly checks for collision between such a box and a BSP processed complex level. The algorithm presented is for the case of a AABB colliding with a convex polygon of any number of edges but it can be easily extended to handle other types of geometry like dynamic LOD Bezier surfaces and triangle soups. The algorithm has been implemented and tested over a huge number of level geometry configurations. The implementation extends the concepts presented here to curved faces and other types of geometry supported by the engine. The collision detection is accomplished by using the following basic intersection checks: ray/polygon intersection check; ray/AABB intersection check; edge/edge intersection check.

2 Collision detection and collision response

We first separate the collision detection from the collision response. The main collision detection function will be called with a local AABB (minimum and maximum points relative to its origin), the current position (p1) - the position the object reached in the previous frame - and the desired destination position (p2) - the position the object wants to move to in the current frame (Fig 1-left). The function will check if the supplied AABB can move from p1 to p2 and, if a collision is found, it will process it applying the collision response code and recourse to compute the path required by the movement. For a simple box/face intersection as in Fig 1-center, only two loops will be required. The first collision moves the box to p1' and computes the new destination position p2' using the response code. Then it loops again doing collision detection for moving from p1' to p2'. As no collision is found between p1' and p2', it will stop the loop and return the p2' as the current position for the AABB. In some cases more loops are needed as in the case of another collision being found between p1' and p2'. The collision detection method has to find if a box defined by its minimum and maximum points, moving from point p1 to point p2, will collide anything. To achieve this we will need to perform several computations,

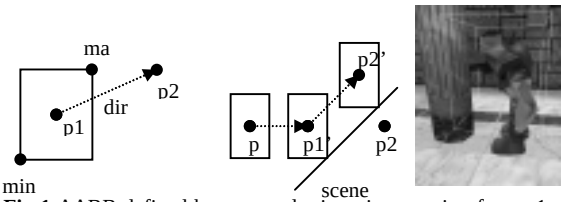

Fig 1 AABB defined by max and min points moving from p1 to p2 (left) Collision detection/response recursion. AABB was moving from p1 to p2. Recursion continues until no collision is found (center). Results on collision detection (right)

but fortunately we can cull several of them with simple dot product tests, and thus facilitating real-time performance. Lets consider the AABB defined by a set of 8 vertices, 12 edges and 6 faces. We then need to compute the closest collision of the following set: collision of each of the 8 vertices of the AABB with the scene geometry (ray/polygon intersection); collision of each of the scene geometry vertices with the 6 faces of the AABB (ray/AABB intersection); collision of each of the 12 edges of the AABB with every other scene edge (edge/edge intersection). It looks like a lot of computation but culling allows us to reduce the computation to a minimum and only compute the vertices and edges that could actually generate a collision. First we start by creating a temporary bound box that will enclose the two (original and destination) bound boxes (Fig 1-right). This super AABB must then enclose the AABB of the moving object when and if it collides with a static object. This box then facilitates the first culling operation. We use this bounding box to recourse the BSP tree and find the BSP tree leaf nodes that intersect the temporary bounding box. For all the faces in these leaf nodes we cull their bounding boxes with the temporary bound box using the very simple and fast code that checks if two AABB intersects.

3 Conclusions

We presented a very efficient and precise way of determining the collision of an AABB with a complex scene made of convex polygons organized in a BSP tree. Collision response is also treated and the algorithm can be extended to other types of geometry commonly used in games like curved surfaces and detail geometry made of triangles. The full version of this paper with detailed implementation and demos can be found at the web site http://www.fly3d.com.br.

4 References

[1] Fly3D Game Engine, Paralelo Computação Ltda, www.fly3d.com.br.
[2] Alan Watt and Fabio Policarpo, *3D Games Technology*, Addison-Wesley (2000).
[3] Alan Watt and Fabio Policarpo, *The Computer Image*, Addison-Wesley, (1997).