LAMPIRAN A: Kode Program Metode OBB

```
// given two boxes (p1,R1,side1) and (p2,R2,side2), collide them together
//and
// generate contact points. this returns 0 if there is no contact otherwise
// it returns the number of contacts generated.
// `normal' returns the contact normal.
// `depth' returns the maximum penetration depth along that normal.
// `return_code' returns a number indicating the type of contact that was
// detected:
//        1,2,3 = box 2 intersects with a face of box 1
//        4,5,6 = box 1 intersects with a face of box 2
//        7..15 = edge-edge contact
// `maxc' is the maximum number of contacts allowed to be generated, i.e.
// the size of the `contact' array.
// `contact' and `skip' are the contact array information provided to the
// collision functions. this function only fills in the position and depth
// fields.


int dBoxBox (const dVector3 p1, const dMatrix3 R1,
             const dVector3 side1, const dVector3 p2,
             const dMatrix3 R2, const dVector3 side2,
             dVector3 normal, dReal *depth, int *return_code,
             int flags, dContactGeom *contact, int skip)
{
  const dReal fudge_factor = REAL(1.05);
  dVector3 p,pp,normalC={0,0,0};
  const dReal *normalR = 0;
  dReal A[3],B[3],R11,R12,R13,R21,R22,R23,R31,R32,R33,
    Q11,Q12,Q13,Q21,Q22,Q23,Q31,Q32,Q33,s,s2,l,expr1_val;
  int i,j,invert_normal,code;

  // get vector from centers of box 1 to box 2, relative to box 1
  p[0] = p2[0] - p1[0];
  p[1] = p2[1] - p1[1];
  p[2] = p2[2] - p1[2];
  dMultiply1_331 (pp,R1,p);                 // get pp = p relative to body 1

  // get side lengths / 2
  A[0] = side1[0]*REAL(0.5);
  A[1] = side1[1]*REAL(0.5);
  A[2] = side1[2]*REAL(0.5);
  B[0] = side2[0]*REAL(0.5);
  B[1] = side2[1]*REAL(0.5);
  B[2] = side2[2]*REAL(0.5);
```

```
  // Rij is R1'*R2, i.e. the relative rotation between R1 and R2
  R11 = dCalcVectorDot3_44(R1+0,R2+0); R12 =
dCalcVectorDot3_44(R1+0,R2+1); R13 =
dCalcVectorDot3_44(R1+0,R2+2);
  R21 = dCalcVectorDot3_44(R1+1,R2+0); R22 =
dCalcVectorDot3_44(R1+1,R2+1); R23 =
dCalcVectorDot3_44(R1+1,R2+2);
  R31 = dCalcVectorDot3_44(R1+2,R2+0); R32 =
dCalcVectorDot3_44(R1+2,R2+1); R33 =
dCalcVectorDot3_44(R1+2,R2+2);

  Q11 = dFabs(R11); Q12 = dFabs(R12); Q13 = dFabs(R13);
  Q21 = dFabs(R21); Q22 = dFabs(R22); Q23 = dFabs(R23);
  Q31 = dFabs(R31); Q32 = dFabs(R32); Q33 = dFabs(R33);

  // for all 15 possible separating axes:
  //   * see if the axis separates the boxes. if so, return 0.
  //   * find the depth of the penetration along the separating axis (s2)
  //   * if this is the largest depth so far, record it.
  // the normal vector will be set to the separating axis with the smallest
  // depth. note: normalR is set to point to a column of R1 or R2 if that is
  // the smallest depth normal so far. otherwise normalR is 0 and
  // normalC   is
  // set to a vector relative to body 1. invert_normal is 1 if the sign of
  // the normal should be flipped.

  do {
#define TST(expr1,expr2,norm,cc) \
    expr1_val = (expr1); /* Avoid duplicate evaluation of expr1 */ \
    s2 = dFabs(expr1_val) - (expr2); \
    if (s2 > 0) return 0; \
    if (s2 > s) { \
      s = s2; \
      normalR = norm; \
      invert_normal = ((expr1_val) < 0); \
      code = (cc); \
            if (flags & CONTACTS_UNIMPORTANT) break; \
          }

    s = -dInfinity;
    invert_normal = 0;
    code = 0;

    // separating axis = u1,u2,u3
    TST (pp[0],(A[0] + B[0]*Q11 + B[1]*Q12 + B[2]*Q13),R1+0,1);
    TST (pp[1],(A[1] + B[0]*Q21 + B[1]*Q22 + B[2]*Q23),R1+1,2);
    TST (pp[2],(A[2] + B[0]*Q31 + B[1]*Q32 + B[2]*Q33),R1+2,3);
```

```
    // separating axis = v1,v2,v3
    TST (dCalcVectorDot3_41(R2+0,p),(A[0]*Q11 + A[1]*Q21 + A[2]*Q31 +
B[0]),R2+0,4);
    TST (dCalcVectorDot3_41(R2+1,p),(A[0]*Q12 + A[1]*Q22 + A[2]*Q32 +
B[1]),R2+1,5);
    TST (dCalcVectorDot3_41(R2+2,p),(A[0]*Q13 + A[1]*Q23 + A[2]*Q33 +
B[2]),R2+2,6);

    // note: cross product axes need to be scaled when s is computed.
    // normal (n1,n2,n3) is relative to box 1.
#undef TST
#define TST(expr1,expr2,n1,n2,n3,cc) \
    expr1_val = (expr1); /* Avoid duplicate evaluation of expr1 */ \
    s2 = dFabs(expr1_val) - (expr2); \
    if (s2 > 0) return 0; \
    l = dSqrt ((n1)*(n1) + (n2)*(n2) + (n3)*(n3)); \
    if (l > 0) { \
      s2 /= l; \
      if (s2*fudge_factor > s) { \
        s = s2; \
        normalR = 0; \
        normalC[0] = (n1)/l; normalC[1] = (n2)/l; normalC[2] = (n3)/l; \
        invert_normal = ((expr1_val) < 0); \
        code = (cc); \
        if (flags & CONTACTS_UNIMPORTANT) break; \
            } \
          }

    // We only need to check 3 edges per box
    // since parallel edges are equivalent.
    // separating axis = u1 x (v1,v2,v3)
    TST(pp[2]*R21-pp[1]*R31,(A[1]*Q31+A[2]*Q21+B[1]*Q13+B[2]*Q12),0,-
R31,R21,7);
    TST(pp[2]*R22-pp[1]*R32,(A[1]*Q32+A[2]*Q22+B[0]*Q13+B[2]*Q11),0,-
R32,R22,8);
    TST(pp[2]*R23-pp[1]*R33,(A[1]*Q33+A[2]*Q23+B[0]*Q12+B[1]*Q11),0,-
R33,R23,9);

    // separating axis = u2 x (v1,v2,v3)
    TST(pp[0]*R31-
pp[2]*R11,(A[0]*Q31+A[2]*Q11+B[1]*Q23+B[2]*Q22),R31,0,-R11,10);
    TST(pp[0]*R32-
pp[2]*R12,(A[0]*Q32+A[2]*Q12+B[0]*Q23+B[2]*Q21),R32,0,-R12,11);
    TST(pp[0]*R33-
pp[2]*R13,(A[0]*Q33+A[2]*Q13+B[0]*Q22+B[1]*Q21),R33,0,-R13,12);
```

```
    // separating axis = u3 x (v1,v2,v3)
    TST(pp[1]*R11-pp[0]*R21,(A[0]*Q21+A[1]*Q11+B[1]*Q33+B[2]*Q32),-
R21,R11,0,13);
    TST(pp[1]*R12-pp[0]*R22,(A[0]*Q22+A[1]*Q12+B[0]*Q33+B[2]*Q31),-
R22,R12,0,14);
    TST(pp[1]*R13-pp[0]*R23,(A[0]*Q23+A[1]*Q13+B[0]*Q32+B[1]*Q31),-
R23,R13,0,15);
#undef TST
  } while (0);

  if (!code) return 0;

  // if we get to this point, the boxes interpenetrate. compute the normal
  // in global coordinates.
  if (normalR) {
    normal[0] = normalR[0];
    normal[1] = normalR[4];
    normal[2] = normalR[8];
  }
  else {
    dMultiply0_331 (normal,R1,normalC);
  }
  if (invert_normal) {
    normal[0] = -normal[0];
    normal[1] = -normal[1];
    normal[2] = -normal[2];
  }
  *depth = -s;

  // compute contact point(s)

  if (code > 6) {
    // An edge from box 1 touches an edge from box 2.
    // find a point pa on the intersecting edge of box 1
    dVector3 pa;
    dReal sign;
    // Copy p1 into pa
    for (i=0; i<3; i++) pa[i] = p1[i]; // why no memcpy?
    // Get world position of p2 into pa
    for (j=0; j<3; j++) {
      sign = (dCalcVectorDot3_14(normal,R1+j) > 0) ? REAL(1.0) : REAL(-
1.0);
      for (i=0; i<3; i++) pa[i] += sign * A[j] * R1[i*4+j];
    }

    // find a point pb on the intersecting edge of box 2
    dVector3 pb;
    // Copy p2 into pb
```

```
    for (i=0; i<3; i++) pb[i] = p2[i]; // why no memcpy?
    // Get world position of p2 into pb
    for (j=0; j<3; j++) {
      sign = (dCalcVectorDot3_14(normal,R2+j) > 0) ? REAL(-1.0) :
REAL(1.0);
      for (i=0; i<3; i++) pb[i] += sign * B[j] * R2[i*4+j];
    }

    dReal alpha,beta;
    dVector3 ua,ub;
    // Get direction of first edge
    for (i=0; i<3; i++) ua[i] = R1[((code)-7)/3 + i*4];
    // Get direction of second edge
    for (i=0; i<3; i++) ub[i] = R2[((code)-7)%3 + i*4];
    // Get closest points between edges (one at each)
    dLineClosestApproach (pa,ua,pb,ub,&alpha,&beta);
    for (i=0; i<3; i++) pa[i] += ua[i]*alpha;
    for (i=0; i<3; i++) pb[i] += ub[i]*beta;
    // Set the contact point as halfway between the 2 closest points
    for (i=0; i<3; i++) contact[0].pos[i] = REAL(0.5)*(pa[i]+pb[i]);
    contact[0].depth = *depth;
    *return_code = code;
    return 1;
  }

  // okay, we have a face-something intersection (because the separating
  // axis is perpendicular to a face). define face 'a' to be the reference
  // face (i.e. the normal vector is perpendicular to this) and face 'b' to be
  // the incident face (the closest face of the other box).
  // Note: Unmodified parameter values are being used here
  const dReal *Ra,*Rb,*pa,*pb,*Sa,*Sb;
  if (code <= 3) { // One of the faces of box 1 is the reference face
    Ra = R1; // Rotation of 'a'
    Rb = R2; // Rotation of 'b'
    pa = p1; // Center (location) of 'a'
    pb = p2; // Center (location) of 'b'
    Sa = A;  // Side Lenght of 'a'
    Sb = B;  // Side Lenght of 'b'
  }
  else { // One of the faces of box 2 is the reference face
    Ra = R2; // Rotation of 'a'
    Rb = R1; // Rotation of 'b'
    pa = p2; // Center (location) of 'a'
    pb = p1; // Center (location) of 'b'
    Sa = B;  // Side Lenght of 'a'
    Sb = A;  // Side Lenght of 'b'
  }
```

```
  // nr = normal vector of reference face dotted with axes of incident box.
  // anr = absolute values of nr.
  /*
        The normal is flipped if necessary so it always points outward from
box 'a',
        box 'b' is thus always the incident box
  */
  dVector3 normal2,nr,anr;
  if (code <= 3) {
    normal2[0] = normal[0];
    normal2[1] = normal[1];
    normal2[2] = normal[2];
  }
  else {
    normal2[0] = -normal[0];
    normal2[1] = -normal[1];
    normal2[2] = -normal[2];
  }
  // Rotate normal2 in incident box opposite direction
  dMultiply1_331 (nr,Rb,normal2);
  anr[0] = dFabs (nr[0]);
  anr[1] = dFabs (nr[1]);
  anr[2] = dFabs (nr[2]);

  // find the largest compontent of anr: this corresponds to the normal
  // for the incident face. the other axis numbers of the incident face
  // are stored in a1,a2.
  int lanr,a1,a2;
  if (anr[1] > anr[0]) {
    if (anr[1] > anr[2]) {
      a1 = 0;
      lanr = 1;
      a2 = 2;
    }
    else {
      a1 = 0;
      a2 = 1;
      lanr = 2;
    }
  }
  else {
    if (anr[0] > anr[2]) {
      lanr = 0;
      a1 = 1;
      a2 = 2;
    }
    else {
      a1 = 0;
```

```
    a2 = 1;
    lanr = 2;
  }
}

// compute center point of incident face, in reference-face coordinates
dVector3 center;
if (nr[lanr] < 0) {
  for (i=0; i<3; i++) center[i] = pb[i] - pa[i] + Sb[lanr] * Rb[i*4+lanr];
}
else {
  for (i=0; i<3; i++) center[i] = pb[i] - pa[i] - Sb[lanr] * Rb[i*4+lanr];
}

// find the normal and non-normal axis numbers of the reference box
int codeN,code1,code2;
if (code <= 3) codeN = code-1; else codeN = code-4;
if (codeN==0) {
  code1 = 1;
  code2 = 2;
}
else if (codeN==1) {
  code1 = 0;
  code2 = 2;
}
else {
  code1 = 0;
  code2 = 1;
}

// find the four corners of the incident face, in reference-face coordinates
dReal quad[8];       // 2D coordinate of incident face (x,y pairs)
dReal c1,c2,m11,m12,m21,m22;
c1 = dCalcVectorDot3_14 (center,Ra+code1);
c2 = dCalcVectorDot3_14 (center,Ra+code2);
// optimize this? - we have already computed this data above, but it is not
// stored in an easy-to-index format. for now it's quicker just to recompute
// the four dot products.
m11 = dCalcVectorDot3_44 (Ra+code1,Rb+a1);
m12 = dCalcVectorDot3_44 (Ra+code1,Rb+a2);
m21 = dCalcVectorDot3_44 (Ra+code2,Rb+a1);
m22 = dCalcVectorDot3_44 (Ra+code2,Rb+a2);
{
  dReal k1 = m11*Sb[a1];
  dReal k2 = m21*Sb[a1];
  dReal k3 = m12*Sb[a2];
  dReal k4 = m22*Sb[a2];
  quad[0] = c1 - k1 - k3;
```

```
  quad[1] = c2 - k2 - k4;
  quad[2] = c1 - k1 + k3;
  quad[3] = c2 - k2 + k4;
  quad[4] = c1 + k1 + k3;
  quad[5] = c2 + k2 + k4;
  quad[6] = c1 + k1 - k3;
  quad[7] = c2 + k2 - k4;
}

// find the size of the reference face
dReal rect[2];
rect[0] = Sa[code1];
rect[1] = Sa[code2];

// intersect the incident and reference faces
dReal ret[16];
int n = intersectRectQuad (rect,quad,ret);
if (n < 1) return 0;            // this should never happen

// convert the intersection points into reference-face coordinates,
// and compute the contact position and depth for each point. only keep
// those points that have a positive (penetrating) depth. delete points in
// the 'ret' array as necessary so that 'point' and 'ret' correspond.
dReal point[3*8];           // penetrating contact points
dReal dep[8];                       // depths for those points
dReal det1 = dRecip(m11*m22 - m12*m21);
m11 *= det1;
m12 *= det1;
m21 *= det1;
m22 *= det1;
int cnum = 0;                       // number of penetrating contact points
found
for (j=0; j < n; j++) {
  dReal k1 =  m22*(ret[j*2]-c1) - m12*(ret[j*2+1]-c2);
  dReal k2 = -m21*(ret[j*2]-c1) + m11*(ret[j*2+1]-c2);
  for (i=0; i<3; i++) point[cnum*3+i] =
                      center[i] + k1*Rb[i*4+a1] + k2*Rb[i*4+a2];
  dep[cnum] = Sa[codeN] - dCalcVectorDot3(normal2,point+cnum*3);
  if (dep[cnum] >= 0) {
    ret[cnum*2] = ret[j*2];
    ret[cnum*2+1] = ret[j*2+1];
    cnum++;
        if ((cnum | CONTACTS_UNIMPORTANT) == (flags &
(NUMC_MASK | CONTACTS_UNIMPORTANT))) {
                break;
        }
  }
}
```

```
  if (cnum < 1) {
            return 0;      // this should not happen, yet does at times
(demo_plane2d single precision).
  }

  // we can't generate more contacts than we actually have
  int maxc = flags & NUMC_MASK;
  if (maxc > cnum) maxc = cnum;
  if (maxc < 1) maxc = 1;     // Even though max count must not be zero this
check is kept for backward compatibility as this is a public function

  if (cnum <= maxc) {
    // we have less contacts than we need, so we use them all
    for (j=0; j < cnum; j++) {
      dContactGeom *con = CONTACT(contact,skip*j);
      for (i=0; i<3; i++) con->pos[i] = point[j*3+i] + pa[i];
      con->depth = dep[j];
    }
  }
  else {
    dIASSERT(!(flags & CONTACTS_UNIMPORTANT));
// cnum should be   // generated not greater than maxc so that "then"
//clause is executed
// we have more contacts than are wanted, some of them must be culled.
 // find the deepest point, it is always the first contact.
    int i1 = 0;
    dReal maxdepth = dep[0];
    for (i=1; i<cnum; i++) {
      if (dep[i] > maxdepth) {
          maxdepth = dep[i];
          i1 = i;
      }
    }

    int iret[8];
    cullPoints (cnum,ret,maxc,i1,iret);

    for (j=0; j < maxc; j++) {
      dContactGeom *con = CONTACT(contact,skip*j);
      for (i=0; i<3; i++) con->pos[i] = point[iret[j]*3+i] + pa[i];
      con->depth = dep[iret[j]];
    }
    cnum = maxc;
  }

  *return_code = code;
  return cnum;
}
```

LAMPIRAN B: Personalia Tenaga Peneliti

Ketua Peneliti:

Nama	: Dr. Elfizar, S.Si, M.Kom
NIDN	: 0027037402
Jabatan Fungsional	: Lektor Kepala
Program Studi	: Manajemen Informatika
Alamat email	: elfizarmd@gmail.com
Kualifikasi	: Kecerdasan Buatan, Jaringan Komputer, Distributed System

Anggota Peneliti:

Nama	: Drs. Sukamto, M.Kom
NIDN	: 0004036401
Jabatan Fungsional	: Lektor Kepala
Program Studi	: Manajemen Informatika
Alamat email	: amto_s@yahoo.com
Kualifikasi	: Application Programming, Matematika Diskrit.

LAMPIRAN C: Publikasi

Elfizar & Sukamto. 2014. Analysis of Axis Aligned Bounding Box in Distributed Virtual Environment. *International Journal of Computer Applications*, Vol. 105, No. 2, pp. 29-33.

Elfizar, Sukamto & Gita Sastria. 2014. Enhanced Oriented Bounding Box Implementation in Distributed Virtual Environment. *Proc. International Conference on Computer System*, December 5, 2014, Makassar.

International Journal of Computer Applications (0975 – 8887)
Volume 105 – No. 2, November 2014

Analysis of Axis Aligned Bounding Box in Distributed Virtual Environment

Elfizar
Department of Information System
University of Riau
Pekanbaru 28293, Indonesia

Sukamto
Department of Information System
University of Riau
Pekanbaru 28293, Indonesia

ABSTRACT
Axis Aligned Bounding Box (AABB) is the simple method for object collision detection, but it has limitation in detection process. In decades, some better methods have been generated such as Oriented Bounding Box (OBB) and HPCCD. Unfortunately, these methods are not used in DVE. This paper aims to analyze why most DVEs still use AABB in detecting objects collision in the environment. This research begins with developing the suitable DVE. The DVE should make many users collaborate with each other, and it has physics activities such as gravity pole, movement, etc. Each user is able to create objects and they should be visible to other users. To detect the object collision, AABB is implemented in the DVE. Further, to analyze the collision detection process and the performance of DVE, there are two parameters used, i.e. runtime and frame rate of simulation application. The experiment results show that adding the computation workload into AABB on DVE increases the runtime significantly compared with regular application. The lack of performance is also shown by the application frame rates in which strictly decrease so that the DVE performance degrades.

General Terms
Distributed Virtual Environment, Distributed Simulation

Keywords
AABB, Collision detection, Distributed Virtual Environment

1. INTRODUCTION
Virtual Environment (VE) is environment that imitates the real environment and makes user feel as residing in the real world. Some activities and situation in this environment should meet the real environment requirements. As the VE involves some users locating in different places geometrically, it is known as Distributed Virtual Environment (DVE). Currently, DVE has been used widely in many applications such as training, education, games, social communities, etc. Even DVE has been used as a powerful tool for autism children training [1].

An aspect in a real world influencing VE is a constraint that two objects are not able to occupy the same point in a space at the same time. Generally, object representation in VE does not allow penetration between objects. Therefore, to develop a simulation environment that represent a real world this constraint should be satisfied. One of important tasks is to detect collision among objects. Collision detection is a mechanism that is able to detect when and where the objects will collide [2].

Collision detection can be classified into two categories, i.e. discrete and continue. Discrete collision detection is a method that just detects the collision at a certain time, for instance at time *t*. Its consequence is that this method misses many collision detections between two consecutive configurations. It is called tunneling problem. Discrete collision detection does not require many computations so that the process is faster.

Continue collision detection can address the tunneling problem because it uses interpolation algorithm to examine the collision in a continue movement. It yields accurate solution for collision detection. Unfortunately, this method is slower than discrete method [3]. One of approaches used in continue collision detection is bounding volume that can be done by using box, sphere, etc. Axis Aligned Bounding Box (AABB) [4] is a method included in this category.

Because of its reliability in detection process, continue collision detection methods have been used by many researches to invent the new faster method. [5] have used linear interpolation between model vertices and computed the first time of collision occurred based on hierarchy selection as well as done basic testing between triangle pairs [6].

Another approach that has been used to accelerate the continue collision detection is using parallel computation [7]. It is inspired by the capability improvement of current processor/CPU that uses multi cores. [8] have yielded parallel collision detection algorithm which run parallel on computers with eight cores CPU and on 16 cores. Each computer gives collision detection speed of 7x and 13x faster, respectively.

Besides using multi processors, acceleration of collision detection speed has been done by using some Graphics Processing Unit (GPU). In contrast to the CPU, GPU processors are very suitable for parallel computation. It is caused by the number of GPU cores is greater than CPU cores, and GPU has more bandwidth than CPU [9-12].

Because bandwidth of both CPU and GPU is restricted, it is required to integrate CPU and GPU in order to compute the collision detection among objects. [13] have used four cores CPU and two GPU. The results show that acceleration can be achieved from 50% to 80% compared with using just CPU for the same test model.

Unfortunately, the improvement in these researches is not followed by the implementation of the resulted methods into DVE. DVE is still using simple method to do the collision detection among objects such as measuring distance between objects [14], and AABB. It affects the DVEs generated by developers in which they give inaccurate collision detection in their environment.

MUSYAWARAH NASIONAL IV
ASOSIASI PERGURUAN TINGGI INFORMATIKA DAN ILMU KOMPUTER

Nomor : 012/ICCS/APTIKOM/XI/2014 Makassar, 10 November 2014
Lampiran : 1 Lembar
Perihal : **Pengumuman Paper Diterima**

Kepada Yth.
Dr. Elfizar, S.Si, M.Kom
Di
Tempat

Dengan hormat,

Berdasarkan hasil telaah dari reviewer International Conference on Computer Systems (ICCS) 2014 dengan ini kami sampaikan bahwa draft paper anda **DITERIMA** dan berhak untuk ikut dalam event ICCS 2014 di Makassar.

Dalam surat ini juga kami sampaikan beberapa hal yang terkait dengan Konferensi pada Munas Aptikom 2014 :

1. Bagi paper yang diterima, untuk dapat mengirimkan Full Paper ke panitia melalui email munasaptikom2014@gmail.com dan di cc ke dedy.triawans@gmail.com dengan subyek email **ICCS_nama peserta** dan **format DOC/DOCX** serta sudah perbaikan jika ada revisi.
2. Full Paper diterima panitia paling lambat tanggal 19 November 2014.
3. Bagi paper yang telah diterima diharapkan untuk segera melakukan pembayaran sesuai dengan kategori peserta paling lambat tanggal 19 November 2014 melalui:
   Rekening BNI 46
   No. Rekening : 4444666227
   an MUNAS APTIKOM (Yayasan Pendidikan Handayani)
   Dan setelah melakukan pembayaran diharapkan untuk konfirmasi ke panitia dengan melampirkan bukti pembayaran.

Demikian surat ini kami sampaikan, atas perhatian dan dukungannya, kami ucapkan terima kasih.

Hormat Kami

Dr. Eng. Armin Lawi, M.Eng
Ketua APTIKOM Wil.IX

Dr. Moh. Alifuddin, MM
Ketua Panitia

Contact Person : Dedy Triawan, S.Kom, MMSI (0811 411 5300)

Enhanced Oriented Bounding Box Method Implementation in Distributed Virtual Environment

Elfizar[1], Sukamto[2], and Gita Sastria[3]

Abstract—This paper aims to implement the Oriented Bounding Box (OBB) in Distributed Virtual Environment. This method is used to do the collision detection between objects. Collision detection is one of important aspects in three-dimensional (3D) applications. Currently, most DVEs only use simple methods in objects collision detection. It influences the accuracy of DVE itself.

This research begins with developing the suitable DVE. It should make many users collaborate with each other. The DVE should also meet the real world requirements, such as gravity pole and object movements. To detect the collision between objects, this research uses OBB. There are two scenarios used in the experiment: using OBB in common DVE, and using it in object-based simulators architecture as an improvement of this research. Further, runtime and frame rates of application are two parameters used to evaluate the performance of DVE.

The experiment results show that the second scenario has higher performance than the first one, either in runtime or frame rates of application.

Keywords—Collision detection, OBB, Distributed Virtual environment.

I. INTRODUCTION

IRTUAL environment (VE) imitates the real environment so that it should meet the real world requirements such gravity pole, kinetics, etc. As VE is used by many users who are geographically distributed, it is known as Distributed Virtual Environment (DVE).

DVE has been used in many applications such as education, training, games, etc. Even, DVE is used as a powerful training tool for autism children [1].

An aspect in a real world influencing VE is a constraint that two objects are not able to occupy the same point in a space at the same time. Generally, object representation in VE does not allow penetration between objects. Therefore, to develop a simulation environment that represent a real world this constraint should be satisfied. One of important tasks is to detect collision among objects. Collision detection is a mechanism that is able to detect when and where the objects will collide [2].

Collision detection can be classified into two categories, i.e. discrete and continue. Discrete collision detection is a method that just detects the collision at a certain time, for instance at time *t*. Its consequence is this method misses many collision detections between two consecutive configurations. It is called tunneling problem. Discrete collision detection does not require many computations so that the process is faster.

Continue collision detection can address the tunneling problem because it uses interpolation algorithm to examine the collision in a continue movement. It yields accurate solution for collision detection. Unfortunately, this method is slower than discrete method [3]. One of approaches used in continue collision detection is bounding volume that can be done by using box, sphere, etc. Axis Aligned Bounding Box [4] and Oriented Bounding Box (OBB) [5] are two methods included in this category.

Actually, many researchers have yielded better methods in collision detection. These methods contain enhanced algorithm to provide more accurately methods [6],[7]. Besides that, several methods have been generated that can be implemented by using multi-processor (CPU) [8], Graphics Processing Unit (GPU) [9], or hybrid of them [10].

Unfortunately, those enhanced methods are not implemented in DVE. Most DVEs only use simple method for collision detection such as AABB.

Elfizar[1] is with the Department of Information System, University of Riau, Pekanbaru 28293, Indonesia (e-mail: elfizarmd@gmail.com).

Sukamto[2] is with the Department of Information System, University of Riau, Pekanbaru 28293, Indonesia.

Gita Sastria[3] is with the Department of Information System, University of Riau, Pekanbaru 28293, Indonesia.