TuEngr Group

International Transaction Journal of Engineering, Management, & Applied Sciences & Technologies

http://TuEngr.com

CYRUS BECK RAY TRACING AND DYNAMIC COLLISION DETECTION

Humera Tariq [a*]

[a] *Department of Computer Science, University of Karachi, PAKISTAN.*

ARTICLEINFO

Article history:
Received 19 February 2018
Received in revised form 14 September 2018
Accepted 23 November 2018
Available online
29 November 2018

Keywords:
Cyrus beck clipping algorithm; OpenGL; Dry Run; 2D Collision detection; Ray tracing.

ABSTRACT

The objective of this paper is to study collision detection and ray tracing in two dimensional (2D) world space. A minimal ray tracing pipeline for dynamic collision detection has been designed, practiced and simulated in C++ and OpenGL. The main contribution of paper is modification of state of the art Cyrus beck clipping algorithm to address collision detection problem. Scene to be displayed has been stored in a simple text file for loading while Animation in 2D world with has been modeled through Markov Chain. Modified pseudocode has been presented along with Dry Run of Cyrus Beck for reader's convenience. Simulation results of 2D ray tracing has been presented by setting up a 2D scene for collision between Polygonal entities. Results shows that Cyrus Beck successfully determine where a ray or polygon with intersect each other and hit time can be successfully estimated with complexity.

1. INTRODUCTION

Collision Detection in 2D is a fascinating alternative term for mathematical formulation of various Intersection Problems including but not limited to Line-Line Intersection, Ray-Line Intersection, Ray-Polygon (Polyhedron) Intersection and Polygon (Polyhedron)-Polygon (Polyhedron) Intersection (More, 2011). Two most important applications in this context are Path (Ray) Tracing and Image Synthesis (Shirley, 2016). The terms Ray tracing and image synthesis are usually employed for raster image creation which is a terminal process of Graphics Pipeline. The performance in this case is measured by dividing the number of primitives (usually triangle) actually intersected by each ray to the total number of triangles exist in scene to be rendered. In contrast to pixel base ray tracing, collision detection is a geometrical term and is meant for vertex based object-object intersection at user's front-end screen in games, movies and simulations. The simplest example to demonstrate both Ray Tracing and Collision Detection is practicing Reflection in Chamber Case study in (Hill, 2007). The case study opens thought process for a variety of concepts like Spatial Partitioning, Spatial Sorting, bounding volumes, overlapping, clipping, mirror reflections and brute force collision detection tests (Lai *et al.*, 2017; Nah *et al.*, 2015; Li and Mukundan, 2013; Kockara, *et al.*, 2007; Wald *et al.*, 2006).

*Corresponding author (HumeraTariq). Tel: +92-3012387949 E-mail: humera@uok.edu.pk

A high level Ray Tracing Pipeline for collision detection comprised of: (1) Scene Loading (2) Object Rendering (3) State Transition (4) Frame Animation (5) Screen Display. The scene loading module takes a text file as input and load the geometrical information of its component objects into appropriate data structure. As a Computer Graphics buff, I always suggest and prefer to employ C++ Standard Template Library to hold description of scene object for e.g. a multiline file containing: { SPIDER_MAN, 200,200} means that an object " SPIDER_MAN" will appear at pixel location $(x, y) = (200, 200)$ on the output screen as shown in Figure 1. The more appropriate and standard way is to use Scene Description language (SDL) or standard OBJ format for reading scene information. (Cahoon *et al.*, 2018; Jo *et al.*, 2018). The scene description in text format shifts programming overhead from Developer to Designer and thus facilitates modular system design approach. Traditional Rendering Modules did two jobs sequentially: (1) Iterate over all scene objects to pixelate and write them into GPU buffer (2) If input scene contains any animated object, it will update its state for e.g position as well. Due to recent and upcoming GPU advancement, these two step has been turn into threads for performance achievement (Bhat and Asberg, 2018). Thus state transition can be separated as a separate module. A Markov chain model to simulate an animated model has been illustrated in Figure 2.

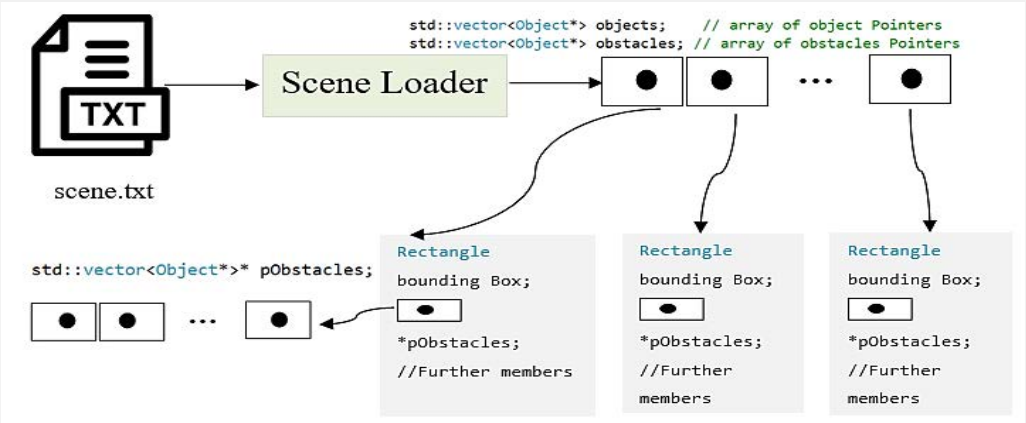

Figure 1: Reading of Simple Scene from Text file into Standard C++ Container.

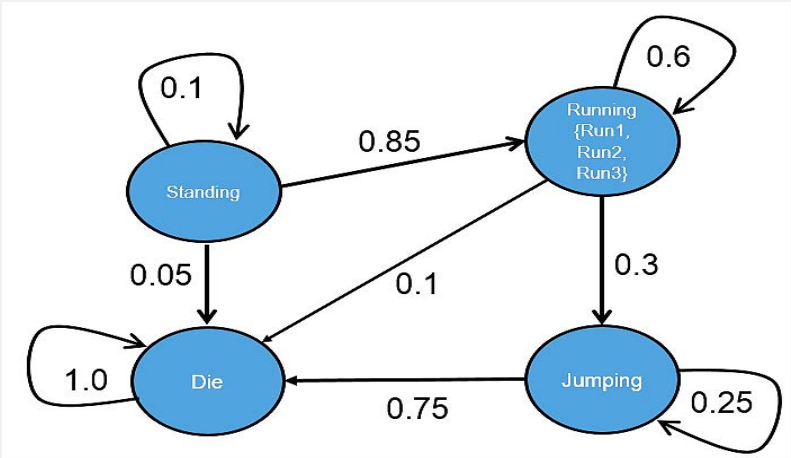

Figure 2: State Transition and Update Mechanism for Moving Entities.

Figure 2 shows that the set of four possible states for an animated character can be expressed as Set S where $S = \{STANDING, RUNNING, JUMPING, DIE\}$. A state can be further sub-divided for

flexibility for e.g. $RUNNING = \{ RUN1, RUN2, RUN3 \}$. At each time step, a character in STANDING state have a 10% chance of staying in that state , have an 85% chance of moving to RUNNING state , and a 5% chance of dying. The same character, when reaches to RUNNING State have a 60% chance of continuing running, 30% chance to move to JUMP state and 1% chance of dying while he is running. The JUMP state continues with 25% chance and it is most dangerous state as the chance of dying in this state will increase to 75%. While Designing animations through Markov model, it is important to note that probabilities out of any state must sum to 1.0. Once the state of the models has been checked and updated, it's time to render new frame so that moving objects will be displayed at new pixels' locations on the old screen. A smart trick to do that is to move the world space (frame) back instead of moving all objects forward. This saves time as only selected and specific animated scene objects needs state transition is called frame animation. This concept of changing world/frame origin $(frame\ X,\ frame\ Y)$ to$(frame\ X \pm object\ velocity,\ frame\ Y + object\ velocity)$ instead of changing character position is illustrated in Figure 3. Computer Graphics Readers knows that addition of point and vector yield new position in 2D space i.e. $Q = P + \vec{v}$. Finally, all the polygonal primitives have been sent to display device screen through low level graphics API for e.g. OpenGL or Vulkan. (Shreiner, 2009; Viggers *et al.*, 2017).

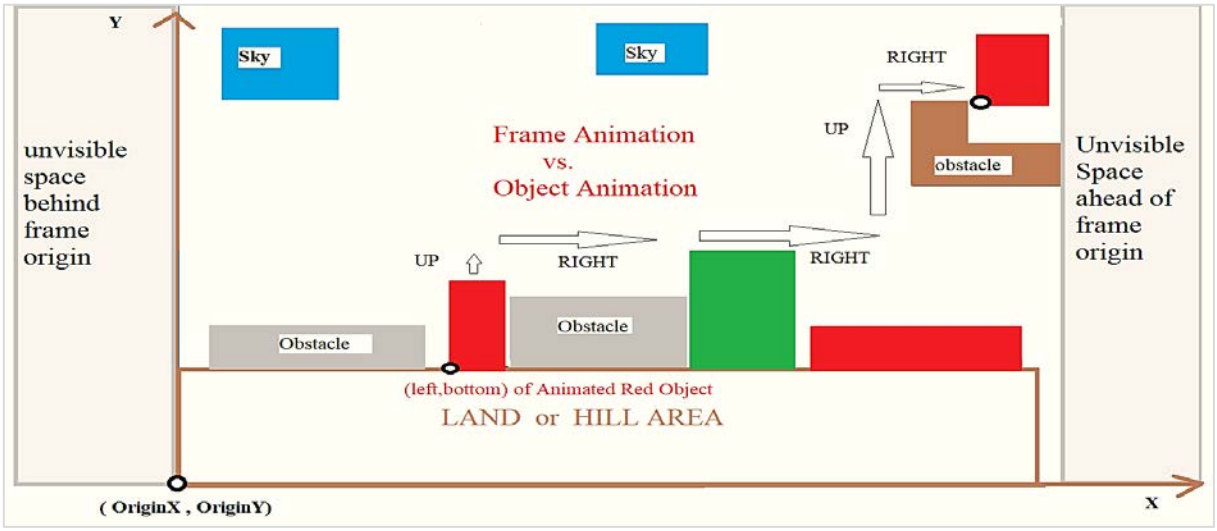

Figure 3: Smart and Popular Frame Animation Trick

2. RAY TRACING SET UP

Cyrus Beck provides solution to the fundamental clipping problem (Skala, 2012). As intersection determination is implicit to clipping, the clipping problem handle collision detection problem as well. Every object in scene exist with a rectangular bounding box around it whose four sides will be treated as a list of rays. Table 1, consider a simple scenario with two moving objects namely **Obj1 (Blue Box)** and **Obj2 (Green Box)** as shown in Figure 4. The set of rays for both are **r_Obj1** and **r_Obj2** respectively and they can be calculated using end points of edges form Equation (1) while normal vectors along edges can be computed by simple swap with correct orientation from Equation (2). Consider one moving object as a set of rays for e.g. **Obj1** with its left bottom (left, bottom) as control point in our case while treat **Obj2** as a complete polygon or Line List. Once rays

along edges has been computed, its intersection with Green Polygon can be estimated as shown in the form of Red Triangle in Figure 4.

$$r[i] = \left(r_x^i, r_y^i\right) = EdgeStartPoint - EdgeEndPoint \; 0 \leq i < 3 \tag{1}$$

$$n[i] = \left(r_y^i, -r_x^i\right) \; 0 \leq i < 3 \tag{2}$$

Figure 4: Rays and Polygon Setup for Cyrus beck Algorithm Collision Detection.

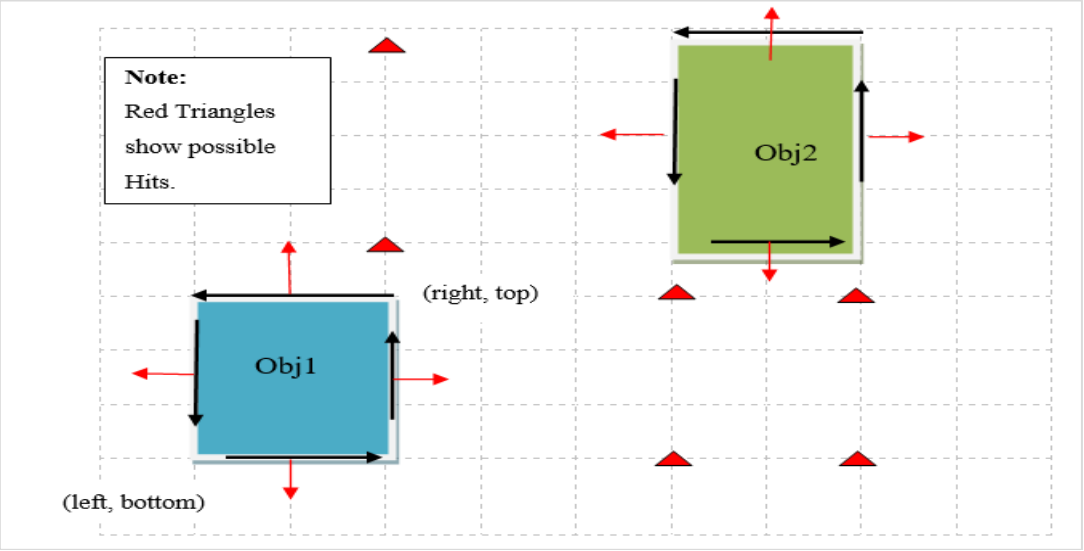

Table 1: Cyrus Beck algorithm for Collision detection and Ray tracing.

```
Create Bounding Box for every Scene Object
hit := false; tIn =0.0; tOut = 1.0;
Obj1 _ r: = ray_list; {Pointer to rays to map bounding box on r[0], r[1],r[2],r[3] }
Obj2 _ r: = ray_list; {Pointer to rays to map bounding box on r[0], r[1],r[2],r[3] }
Vector c :=  r_Obj1
for (i:=0; i< r.size( ); i++)
    Vector n := r_Obj2[i].norm();
    Vector temp = r_Obj2[i].base()- r_Obj1[i].base();
    Numerator = n.dot(temp);              Denominator = n.dot(obj2 Edge Vector)
if( denom < 0 ) // ray is entering
   { float tHit = numer / denom;  if(tHit > tOut )    return false;
     else if( tHit > tIn ) tIn = tHit;
   }
else if( denom > 0 )  // ray is exiting
          {
          float tHit = numer / denom; if( tHit < tIn)return false;
          else if( tHit < tOut ) tOut = tHit;
          }
else if( numer <= 0 ) return false; // Denominator is zero and ray is parallel
          }
if( tOut <= 1.0f || tIn >= 0.0f)
   {   return true;}
```

3. RAY TRACING ALGORITHM

Once rays and polygon has been created and set as described in Section 2. Cyrus Beck can determine all the possible hits for a particular ray say $r[0]$ of **Obj 1** against the line list of **Obj2**. It calculates hit time using Equation 3.

$$t_{hit} = \frac{numerator}{Denominator} = \frac{\vec{n}.\overrightarrow{tmp}}{\vec{n}.\vec{c}} = \frac{\vec{n}.(Obj2_{1st\ Point} - Obj1_{1st\ Point})}{\vec{n}.(Ray_{end} - Ray_{start})} \quad (3)$$

4. SIMULATION RESULTS

Simulation Result can be divided into two cases: (1) Intersection of Ray against Polygon (2) Polygon vs. Polygon Intersection.

4.1 INTERSECTION OF RAY AGAINST POLYGON

Result of Case I, Intersection of Ray against Polygon, is shown in Tables 2 and 3.

Table 2: Polygon and Ray Setup in Case I

i	Pi	A	B	bi or temp (Pi-A)	ei or ray (P(i+1)-Pi)	e or norm
0	(20 2)	(1 2)	(23 12)	(19 0)	(-4 8)	(8 4)
1	(16 10)	(1 2)	(23 12)	(15 8)	(-6 0)	(0 6)
2	(10 10)	(1 2)	(23 12)	(9 8)	(-5 -5)	(-5 5)
3	(5 5)	(1 2)	(23 12)	(4 3)	(15 -3)	(-3 -15)

i	Pi	A	B	bi or temp (Pi-A)	ei or ray (P(i+1)-Pi)	e or norm
0	(885 378)	(275 200)	(314 200)	(610 178)	(64 0)	(0 -64)
1	(949 378)	(275 200)	(314 200)	(674 178)	(0 128)	(128 0)
2	(949 506)	(275 200)	(314 200)	(674 306)	(-64 0)	(0 64)
3	(885 506)	(275 200)	(314 200)	(610 306)	(0 -128)	(-128 0)

Table 3: Candidate Interval for Intersection in Case I

vector c (B-A)	e.bi(numer)	e.c (denom)	t (e.bi)/(e.c)	Hit Status	Candidate Interval
(22 10)	152	216	0.703703704	exiting	(1 0.703704)
(22 10)	48	60	0.8	exiting	(0.7037037 0.8)
(22 10)	-5	-60	0.083333333	entering	(0 0.083333)
(22 10)	-57	-216	0.263888889	entering	(0.0833333 0.263889)

vector c (B-A)	e.bi(numer)	e.c (denom)	t (e.bi)/(e.c)	Hit Status	Candidate Interval
(39 0)	-11392	0	0	Parallel	(0 0)
(39 0)	86272	4992	17.28205128	exiting	(0 17.28205)
(39 0)	19584	0	1	Parallel	(17.282051 1)
(39 0)	-78080	-4992	15.64102564	entering	(1 15.64103)

*Corresponding author (HumeraTariq). Tel: +92-3012387949 E-mail: humera@uok.edu.pk Volume 9 No.6 ISSN 2228-9860 eISSN 1906-9642 http://TUENGR.COM/V09/589.pdf https://doi.org/10.14456/ITJEMAST.2018.54

4.2 POLYGON VS. POLYGON INTERSECTION

Result of Case II, Polygon vs. Polygon Intersection, has been shown in Tables 4 and 5.

Table 4: Ray and Polygon Setup for Case II.

Polygon 1 Rays and Normals			
r[0].Start	r[1].Start	r[2].Start	r[3].Start
275	314	314	250
200	225	328	303
r[0].End	r[1].End	r[2].End	r[3].End
314	314	250	250
200	303	328	225
r[0].dir	r[1].dir	r[2].dir	r[3].dir
39	0	-64	0
0	78	0	-78
n[0].dir	n[1].dir	n[2].dir	n[3].dir
0	78	0	-78
-39	0	64	0

Polygon 2 Rays and Normals			
base/ r[0].Start	base/r[1].Start	base/r[2].Star	base/r[3].Start
910	949	949	885
378	403	506	481
r[0].End	r[1].End	r[2].End	r[3].End
949	949	885	885
378	481	506	403
r[0].dir	r[1].dir	r[2].dir	r[3].dir
39	0	-64	0
0	78	0	-78
n[0].dir	n[1].dir	n[2].dir	n[3].dir
0	78	0	-78
-39	0	64	0

Table 5: Hit results based on Cyrus beck intersection Algorithm for case II.

Origin and Input Polygons Corners		
Origin	Rect1 (l,b)	Rect1 (r,t)
0	250	314
0	200	328
Width, Height & Polygon 2 Corners		
Widht & Height	Rect2 (l,b)	Rect2 (r,t)
64	885	949
128	378	506

Difference between First Point of both Polygons			
vector tmp1	vector tmp2	vector tmp3	vector tmp4
635	635	635	635
178	178	178	178
Numerator = n.dot(temp) where tmp = mfirst-tfirst			
-6942	49530	11392	-49530
Denominator = n.dot(v) where v = turtle first ray			
0	0	0	0
Hit Condition based on Denominator			
Parallel	Parallel	Parallel	Parallel

5. CONCLUSION

The Paper review and discuss Cyrus Beck intersection algorithm along with its usage scenario. Two cases have been designed and simulated for results. Results shows that Cyrus Beck successfully determine where a ray or polygon with intersect each other and hit time can be successfully estimated with complexity $O(n^2)$ and hence it is important to further improve this work for both efficiency and rendering perspective.

6. REFERENCES

Hill., F. S., (2007). *Computer Graphics Using Open Gl*. Upper Saddle River NJ: Prentice Hall.

Kockara, S. et. al. (2007). Collision Detection: A Survey. A survey, in: *IEEE International Conference on Systems, Man and Cybernetics*, Montreal, QC, Canada, 2007, pp. 4046-4051.

Lai, W.H, Tang, C.Y. and Chang, C.F. (2017). Ray Tracing API Integration for OpenGL Applications. WSCG 2017: poster papers proceedings: *25th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* in co-operation with EUROGRAPHICS Association, pp. 1-5.

Mora, B. (2011). Naive Ray-Tracing: A Divide-And-Conquer Approach. *ACM Transactions on Graphics*, 30(5), 117:1-117:12.

Nah, J. et. al (2015) HART: A Hybrid Architecture for Ray Tracing Animated Scenes. *IEEE Transactions on Visualization and Computer Graphics*, 21(3), pp 1-15.

Shreiner, D. (2009). OPENGL PROGRAMMING GUIDE, 7th ed., Addison-Wesley Professional.

Viggers, S., Malnar,T., Ramkissoon, S.R. (2017). Systems and methods for using an opengl api with a vulkan graphics driver. US010102605B1.

Skala , V. (2012). S-Clip E2 : A New Concept of Clipping Algorithms. SIGGRAPH Singapore, ASIA.

Cahoon, R.M, Linscott, G., Treuille, G. (2018) US Patent App.15/405,649, Patents.

Jo, S., Jeong, Y. & Lee, S. J (2018). GPU-Driven Scalable Parser for OBJ Models. *Comput. Sci. Technol*., 33: 417. https://doi.org/10.1007/s11390-018-1827-2

Shirley, P. (2016). *Ray Tracing in One Weekend*.

Wald, I., Boulos, S. and Shirley, P. (2006). Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*.

Tony, L.A, Gose, D. Chakravarthy, A. (2017). Avoidance maps: A new concept in UAV collision avoidance. *International Conference on Unmanned Aircraft Systems (ICUAS)*

Bhat, N., Asberg, F. (2018). Performance of Priority-Based Game Object Scheduling. *Degree Project In Technology*, First Cycle, 15 Credits Stockholm, Sweden.

Dr. Humera Tariq is an Assistant Professor at Department of Computer Science, University of Karachi (UoK), Pakistan. She holds MS/PhD in Computer Science and B.E in Electrical Engineering from UoK and NED University of Engineering and Technology, respectively. She possesses extensive experience in Public Sector Teaching and interested in reforms development in higher education sector. Her research interest includes image processing, biomedical imaging, computer graphics, deep learning and simulation.