CO1301: Games Concepts
Lab 8 - Collision Detection

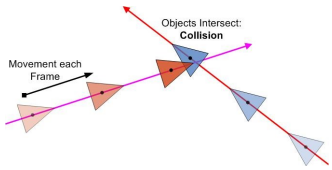
1. Introduction

1. In games, we often need to detect when two models have collided. In this session you will see how to detect collisions using some simple mathematics.

2. Collision in Games

1. In the real world a collision occurs when two moving objects meet. However, in a game world, a collision is often detected only when the models intersect.

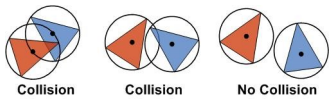
2. This is because of the way we update the position of the models by a small amount on each iteration of the game loop:



You cannot detect a collision by simply testing if the two models have the same position. In the diagram above, the two models above have collided even though their positions are different. However, testing if two complex objects intersect is both mathematically difficult and computationally expensive. We will look at a simple solution to this problem.

3. Bounding Volumes

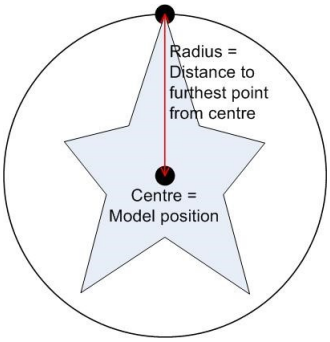
1. The most common solution to this problem is to replace the models with simpler shapes for the purpose of detecting collisions. For each model we choose a mathematically simple shape that is big enough to completely surround it. These simple shapes are tested accurately for intersection against each other. These shapes are called bounding volumes because they define a boundary for the space occupied by the model. Common bounding volumes are spheres and cuboids.



2. This method will correctly find real collisions, but it will also incorrectly find collisions when the models are only close to each other. This can be minimised by making the bounding volume as small as possible (as in the diagram), but this problem cannot be eliminated (unless the object being tested the same shape as the bounding volume).

3. Despite this drawback, this solution can be of great use. Where collision accuracy is not important and computational speed is important, then this is an excellent method. It is also very often used as an initial test in a more complex collision scheme: if a bounding sphere collision check fails then we can guarantee there is no collision and don't need to spend time on a more accurate, complex test.

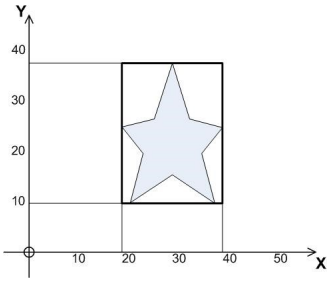
4. To *define a bounding sphere* (or any sphere, for that matter), we must choose the centre point and a radius. The most convenient centre point for a model is simply the position of the model. Then we need to choose a radius so that the sphere will completely surround the model. Selecting the distance from the centre to the furthest point on the model will achieve this. Alternatively this value can simply be estimated by measuring the model.



This choice of sphere will not be ideal if the model origin is not perfectly centred (it isn't in the diagram above). However it will suffice for our current examples.

5. One of the main advantages of using a sphere for a bounding volume is that it remains constant under rotation. In other words, if we rotate the model, the bounding sphere does not change its profile/outline. This simplifies the mathematics immensely and makes spheres a good initial choice for models that can rotate.

6. Another common bounding volume is a box. The simplest form is an axis-aligned bounding box (sometimes called an AABB). The diagram below shows an example using the same model as above:



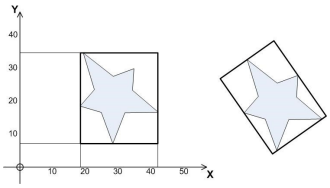
7. The box is called axis-aligned because its edges are parallel to the axes. This is convenient because it means the edges of the box define ranges on each of the axes. E.g. the top and bottom of this box define the range from 9 to 38 on the Y-axis. This simplifies the collision algorithms.

8. Notice how this volume more tightly contains the model in the example. This will lead to less incorrect collision detections. Bounding boxes frequently have this advantage over bounding spheres.

9. To define a bounding box we simply find the minimum and maximum X, Y and Z values for all the points in the model.

10. If we rotate the model when using a bounding box we must either:

    - Rotate the bounding box, but then it will not be axis-aligned and will lose some of its mathematical simplicity.
    - Resize the box and keep it axis-aligned, but this may be computationally expensive.



So axis-aligned bounding boxes are better suited to models that do not rotate.

11. Different algorithms are used to detect collisions between bounding volumes depending on the types of volumes involved. In this lab, We will examine four cases:

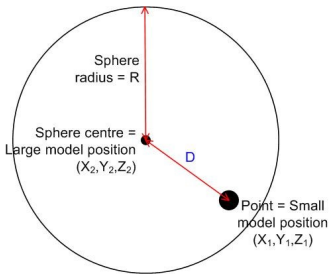
    - Point – Sphere collisions

- Sphere – Sphere collisions
- Point – Box collisions
- Sphere – Box collisions

4. Point - Sphere Collision

1. When testing for a collision between two models, a small model and a large model, sometimes it is possible to use a bounding volume for one model (the large model) but only the position for the other (the small model). Effectively we are using a point as the bounding volume for the small model - this can be appropriate when the model is so small we simply need to test if its position, the point, is within the sphere bounding volume of the large model.

2. The collision will occur only if the distance between the point and the sphere centre is less than the sphere radius.



3. We can find this distance using the vector methods from our previous work. The vector V between the point and the sphere centre is:

   $V\ (X_2 - X_1,\ Y_2 - Y_1,\ Z_2 - Z_1)$

   And the length D of this vector is:

   $D = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$

   If D is less than the sphere radius R then we have a collision. The code for this algorithm would look something like this:

```
float x, y, z;
x = SmallModel->GetX() - LargeModel->GetX();    // Using the large model's
y = SmallModel->GetY() - LargeModel->GetY();    // position as its
z = SmallModel->GetZ() - LargeModel->GetZ();    // bounding sphere centre

float collisionDist = sqrt( x*x + y*y + z*z );  // need #include

if (collisionDist < sphereRadius)
{
    // Collision occurred…
}
```

5. Point - Sphere Collision Exercise

1. Create a new TL-Engine project called "Lab8_SphereCollisions_Project".

2. Get the sample TL-Engine code from collisions.cpp on Blackboard and copy the code into your project's .cpp file (to avoid errors check that the path to the default Tl-Engine media folder is correct). It is a very simple program allowing you to move a sphere around.

3. Using the "Bullet.x" mesh, add a bullet model at (40, 10, 40) – it is small, so we will assume it is a point.

4. In the game-loop, add code to check for collision between the bullet (point) and the sphere at every frame.

Use the code snippet from Section 4 to implement the collision detection. The sphere is your "LargeModel" and the bullet is your "SmallModel". The sphere radius here is 10 – in this case the bounding volume is exactly the right shape. Declare a float variable to store the radius before using it.

5. If a collision occurs, set the bullet's Y position to 25.
6. Add some code to set the bullet's Y position back to 10 if you press a key.
7. The code you have written to complete the above 3 steps should look like the code below:

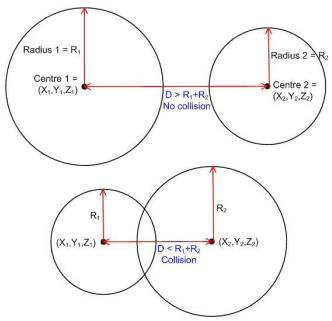
```
float x, y, z;
float sphereRadius = 10.0f; //also serves as the boundary volume's radius

//the components of the vector between the bullet and the center of the sphere
x = bullet->GetX() - sphere->GetX();
y = bullet->GetY() - sphere->GetY();
z = bullet->GetZ() - sphere->GetZ();

float collisionDist = sqrt(x * x + y * y + z * z);      //the length of the vector i.e. the di

if (collisionDist < sphereRadius) //check if the distance is less than sphereRaidus, then move
{
    bullet->SetY(25.0f);
}

//reset the position of the bullet after collision resolution has moved it
if (myEngine->KeyHit(Key_R)) {
    bullet->SetY(10.0f);
}
```

8. Now test if your collision works – see how accurate it is.

   Extra: try using smaller/larger values for sphereRaidus to see how that affects the precision of collision detection.

6. Sphere - Sphere Collision

1. This case can be easily understood with the diagram below:



2. The process is similar to the point-sphere case:
   - Calculate the vector between the two sphere centres (model positions)
   - Get the length of this vector (the distance between the models)
   - If this distance is less than the sum of the sphere radii then there is a collision

7. Sphere - Sphere Collision Exercise

1. Using the "Torus.x" mesh, update your Lab8_SphereCollisions_Project by adding a torus model at (-40, 10, 40).
2. Add code to check for collision between the sphere and torus. Use a bounding sphere with radius of 15 for the torus.

HINT: You will need to declare a new float to store torusRadius, calculate the vector between the torus and the sphere, calculate the length of the vector and check if the distance is less than(torusRadius + sphereRadius)

3. Again make the torus jump up if there is a collision and make it come down on a keypress.

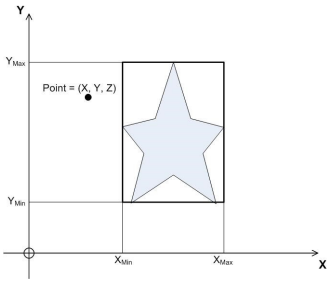
8. Point – Box Collision

1. The process here is also fairly simple: there is a collision if the X, Y and Z coordinates of the point all lie within the three axis ranges defined by the bounding box i.e. there is a collision if:

   X > XMin && X < XMax &&

   Y > YMin && Y < YMax &&

   Z > ZMin && Z < ZMax



   There is no collision in the diagram above because X < XMin. Note that the diagram is 2-dimensional and does not show the Z axis.

9. Point – Box Collision Exercise

1. Create a new TL-Engine project called "Lab8_BoxCollisions_Project".
2. Get the sample TL-Engine code from collisions.cpp on Blackboard again but this time change the sphere model to a bullet model.
3. Add a box model using the "Cube.x" mesh at (40, 10, 40).
4. Define a bounding box by declaring six float constants: cubeMinX, cubeMaxX, cubeMinY etc.
5. The box's dimensions are 10x10x10, its centre point (model origin) is in the centre of the bottom face and you placed it at (40, 10, 40). Use this information to set the initial values for the bounding box variables you have just created (the bounding box should be perfectly accurate for this case).
6. Add code to check for collision between the bullet and box.
7. Again make the box jump if there is a collision. Then reposition it on a key press.

10. Sphere – Box Collision

1. So far the algorithms have been fairly simple. However, true collisions between spheres and boxes are rather more complex. We will use a trick to simplify things:
   - Expand the bounding box outwards by the sphere's radius, i.e. XMin becomes XMin – Radius, XMax becomes XMax + Radius etc.
   - Shrink the sphere's radius down to 0, i.e. assume it is a point.
   - Now we simply have the point-box collision situation again, and we can use the formula from Section 8.

- This method won't be accurate around the corners of the box, but using approximations like this is perfectly appropriate in some situations.

11. Sphere – Box Collision Exercises

1. In your Lab8_BoxCollisions_Project, change the bullet model back to a Sphere.
2. Update your collision code to manage a sphere – box collision, (sphere radius = 10).
3. Make the cube jump on a collision. Can you spot the inaccuracies with this trick.
4. Can you spot the collision detection inaccuracies produced by this method?

12. Advanced Exercises - Collision Detection

1. Now create float variables to store X and Z velocity of the sphere, initialised to 0.
2. Change the keyboard controls so that they don't move the sphere, but increase or decrease the velocity variables instead.
3. You will need to reduce the movement speeds used in your program. You may want to set maximum speeds also.
4. Make the sphere move by the stored velocity every frame (regardless of key presses) and remove the cube jump.
5. Now when you detect a collision against the box try to make the sphere bounce. You will need to work out which side you hit (front/back or left/right) and then negate the appropriate velocity variable.

   Hint: try to find which box edge (MaxX, MinX…) is nearest to the sphere position