P3_Tutorial6 pt.1

Collision Detection

Includes algorithms from checking for intersection between two given solids, to calculating trajectories, impact times and impact points in a physical simulation

MSDN defines collision detect as: Process for determining if any pixels for two images share the same location on the screen.

Basically we detect whether two sprites have collided or not

Almost every 3D game today uses some kind of collision test in their game engine.
For example, you want to decide if a missile has hit a character, or a character is colliding with a wall or trip-wire. The first goal when creating a collision test is to decide what kind of collision test to use. You need to keep a lot of things in mind when making this choice: high frame rate, collision accuracy, terrain, etc.

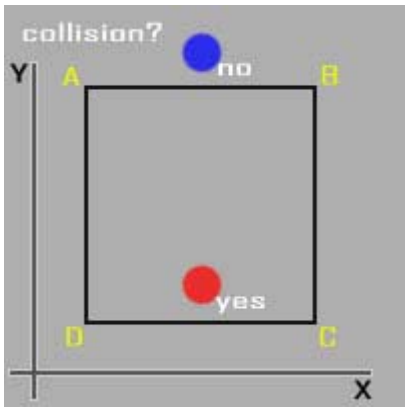
In this tutorial we will cover the basics in one of the easiest collision tests to understand (less math and calculations) “The Bounding Box”.

2D Program Examples

The examples to see for this are Collisions2D, CollisionDIKeyboard and CollisionDIMouse
These use bounding box collision, this tutorial covers the methods they use for bounding box collision in a descriptive way so that you may be able implement such collision techniques in your own code.

Bounding Box

The bounding box collision test is based on a simple inside/outside system:



The above diagram illustrates bounding box collision in a 2D environment. A collision occurs when the box is entered.
Let's say A,B,C and D has the given coordinates: A(2,10), B(10,10), C(10,2) and D(2,2).

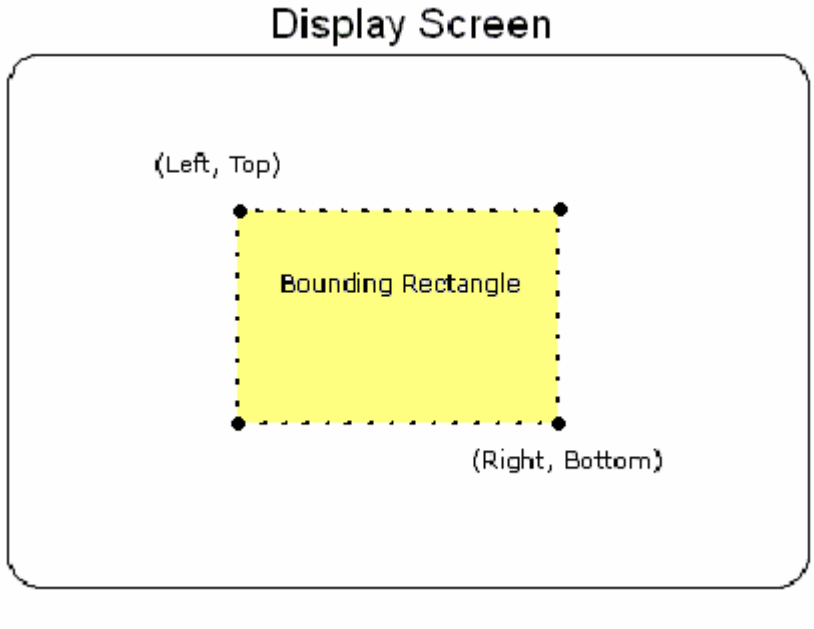
Therefore to be inside the box the sprites position has to be:

```
pos.x < 10    pos.x > 2   //in x-direction.
pos.y < 10    pos.y > 2   //in y-direction.
```

The red dot is inside because it has the position (6, 4).

What we need therefore for bounding box collision detection are structures to hold co-ordinate information for bounding boxing that are going to represent the collision area of our sprites and the ability to call a function/method that will work out if a collision has occurred.
Taking any other appropriate action is the up to you.

A Word from Microsoft
Throughout Microsoft Direct3D and Microsoft Windows programming, objects on the screen are referred to in terms of bounding rectangles. The sides of a bounding rectangle are always parallel to the sides of the screen, so the rectangle can be described by two points, the upper-left corner and lower-right corner. Most C++ applications use the RECT structure to carry information about these two bounding rectangle corners.



Most C++ applications use the RECT structure to carry information about these two bounding rectangle corners. The RECT structure holds the following information

```
typedef struct tagRECT {
   LONG left;
   LONG top;
   LONG right;
   LONG bottom;
} RECT;
```

Members
left
Specifies the x-coordinate of the upper-left corner of a rectangle
top
Specifies the y-coordinate of the upper-left corner of a rectangle
right
Specifies the x-coordinate of the lower-right corner of a rectangle
bottom
Specifies the y-coordinate of the lower-right corner of a rectangle

Defining the Bounding Box
We need a structure to hold our Boundary Box information:

```
struct TBB
{
     TRect bb_sRect, bb_rect;
     TPoint bb_pt1, bb_pt2, bb_pt3, bb_pt4;
};
```

We can either use already defined structures such as RECTS and POINTS or we can define our own - TRect and TPoint are our own defined types that look like this:

```
struct TPoint
{
     float pX, pY;
};

struct TRect
{
     float rL, rR, rT, rB;
};
```

These structures hold the co-ordinates of the points in 2D space – hence X and Y and the Left, Right, Top, Bottom co-ordinates.
It is possible rather than using the user defined structure for the rectangle to use the RECT structure already supplied.

It is possible to make your bounding box smaller than your sprite size; this is recommended if your bitmap has any transparent edges.

The bounding box position either needs to calculated every frame, based on your sprite, or be updated with your sprite so that the bounding box has the correct co-ordinates to correspond with the relevant sprite.

A method that determines if there is a collision is generally called after a position has been updated but before the scene is rendered so that any necessary action may be taken.

The provided 2D collision code through mathematics works out a bounding box that rotates with the sprite, and in the case of CollisionDIMouse and CollisionDIKeyboard draw an additional box further out when the sprite is rotating – see the DrawBB function is sprite.h.

The games using the 2D presented call the create function, passing many of the initial arguments but leaving some as defaults.
The RECT structure defined holds the initial the size bitmap, these values are supplied by the user.
This RECT is passed to the sprite class via a calling of the create function.

The RECT that is passed as an argument is then assigned to the member data RECT imageRect.

After the initialisation of some more member data with values passed as arguments or left as defaults the method ComputeDimensions() is called, this sets member data such as sp_width and sp_height, as well as the translation. Any scaling also takes place here.
When ComputeBB() is called it then uses these scaled member data and translation to work out where the Bounding Box is located.

The call to create has the following parameters:

```
char*, // where to load the file from
RECT, // bitmap, image rect
float = 0.0f,      // translation – where on the screen to start (x)
float = 0.0f,      // translation - where on the screen to start (y)
bool = false,      // translation - only use our values if set to true
float = 0.0f,      // rotation – rotational centre value
float = 0.0f,      // rotation – rotational centre value
float = 0.0f,      // rotation – angle of rotation
bool = false,      // rotation - only rotate if set to true
float = 1.0f,      // scaling (x) (default 1.0 = actual size)
float = 1.0f,      // scaling (y) (default 1.0 = actual size)
bool = false,      // scaling - only scale if set to true
DWORD = 0xFF000000,
DWORD = 0xFFFFFFFF);// transparent/draw colours
```

Those set to equals are default parameters; this makes sure that important member data is initialised with a value if the user does not supply one.

The DrawBB() method uses vertices to draw the boundary box.

The CollisionRR(), CollisionRNR() and similar functions check for collision on multiple sides.

Examine the Code
Look at the collision code already mentioned and also at this week's collision code (Basic Collision), available online from my website.
The Basic Collision example strips away a lot and reduces it to the fundamentals necessary for the code to function, and demonstrate through drawing lines, the principle of Boundary Box Collision.

Unlike the other code examples, this code is more self-contained and so it should be easier to track what is occurring, it also abstains from the use of classes.

Exercises

1) Work through the code, especially for CollisionDIMouse and Basic Collision. Fathom how the sprite class is initialised, following the member data as it is used to calculate the Bounding Box
2) For CollisionDIMouse in the file DIMouseControlLooper, go to the function InitLooperSprites and alter the initial RECT values passed before the call to create. What happens and why?
3) In BasicCollision alter the values passed to the CreateBB() function, so that the Boundary Box is tighter against the image
4) Try altering the scaling, rotation and translation to determine the effects on box the Bounding Box and the actual sprite image
5) Alter your sprite class to include collision detection