Notes based on
Collisions can occur whenever objects are moving. It means two (or more) objects are touching. Handling collisions is an important aspect of computer graphics. There are essentially two aspects: (1) collision detection and collision response. If collisions are not handled objects simply move through each other. Collision detection is known to be an expensive operation, especially as the number of moving objects increases.
A *brute force* approach to collision detection uses intersection tests of all objects against all other objects — or *pairwise testing* — at discrete points in time where objects are assumed to be instantaneously stationary.

```
for i = 1 to n - 1 do
    for j = i to n do
        intersect(object(i), object(j));
```

**Algorithm 1:** Brute Force Collision Detection
Brute Force: Computational Complexity

There are \( \frac{n(n-1)}{2} \) pairs amongst \( n \) objects, and this is the number of object-object intersection tests the brute-force algorithm performs at each point in time.

Using the big O notation, the brute-force algorithm uses \( O(n^2) \) object-object intersection tests. If each intersection test can be performed in \( O(1) \) time, i.e., constant time, then the algorithm is \( O(n^2) \) time.

Usually scenes contain a mixture of moving and stationary (static) objects. Only the moving objects need to be tested against each other and against the static objects. If there is only one moving object the brute force approach becomes \( O(n) \).
Spatial Data Structures

Being $O(n^2)$, the brute force approach to collision detection becomes too slow for an increasing number of objects, i.e. as $n$ increases. This is similar to the way $O(n^2)$ sorting algorithms become too slow as $n$ increases. So better sorting algorithms have been developed, such as merge sort which takes $O(n \cdot \lg(n))$ time.

One way collision detection performance can be improved is by using spatial data structures, e.g.: quad-trees, oct-trees, k-D trees, binary space partition (BSP) trees, and many others. Spatial data structures are also known as geometric data structures.

In spatial data structures objects are organised based on space or geometry. They guide the application of intersection tests to only testing pairs of objects which are “close” — a form of spatial divide-and-conquer.

For a set of 1D keys a (balanced) binary tree can be used to maintain a set of (key, value) pairs supporting a find operation, along with insert and delete operations for adding and removing
Intersection detection is the problem of detecting whether two objects intersect — overlap in space. Intersection detection is carried out by performing intersection tests.

Intersection detection is a problem which occurs in computer graphics in many forms, including: clipping, view-volume culling, ray-tracing, picking and collision detection. Intersection detection lies at the heart of collision detection. Collision detection is intersection detection amongst a set of moving objects.
There are three spatial relationships between two objects:

- no intersection, (disjoint, exclusion),
- partial intersection (overlap) and
- containment (inclusion).

It is important to keep all these possibilities in mind when performing intersection detection.
The classic computer graphics technique of bounding volumes is applied in intersection detection.

Instead of performing expensive intersection tests amongst complex objects (relatively) simpler tests using bounding volumes can be performed.

These tests allow quick ("trivially") determination of exclusion and inclusion cases, where no further work needs be done, and potential overlap cases where further work does need to be done in order to get a correct answer.
Types of Bounding Volumes

Common bounding volumes (BVs) are:

1. Axis-aligned bounding boxes (AABBs).
2. Oriented bounding boxes (OBBs).
3. Discrete oriented polytopes ($k$-DOPs).
4. Spheres.

There is an obvious trade-off: tightness of fit versus complexity and cost of test.

One quantitative measure used in relation to BVs is void volume: the difference between the bounding volume and the object volume.
The tightest fitting bounding volume is the *convex hull*, although it is only strictly defined for polygonal objects or sets of points.
An axis-aligned bounding box (also called an *extent* or a *rectangular box*).

An axis-aligned bounding box is defined by two extreme points. For example an AABB called $A$ is defined by $a^{\text{min}}$ and $a^{\text{max}}$, where $a_i^{\text{min}} \leq a_i^{\text{max}}, \forall i \in x, y, z$. 
Oriented Bounding Boxes (OBBs)

An oriented bounding box is a box which may be arbitrarily oriented (rotated). It is still a box; its faces have normals which are pairwise orthogonal.

An OBB $B$ can be described by the center point of the box $b^c$, and three normalised positively oriented vectors which describe the side directions of the box. The vectors are $b^u$, $b^v$ and $b^w$ with half lengths $h^B_u$, $h^B_v$ and $h^B_w$. 
Discrete Oriented Polytopes ($k$-DOPs)

A $k$-DOP is the intersection of a set of pairs of parallel planes, where each pair of parallel planes is called a slab.
A $k$-DOP is defined by $k/2$ ($k$ even) normalised normals, $n_i$, $1 \leq i \leq k/2$ each with two associated scalar values $d_i^{\text{min}}$ and $d_i^{\text{max}}$ where $d_i^{\text{min}} < d_i^{\text{max}}$. Each triple $(n_i, d_i^{\text{min}}, d_i^{\text{max}})$ defines a slab $S_i$ which is the volume between two planes:

$\pi_i^{\text{min}} : n_i.(x) + d_i^{\text{min}} = 0$ and $\pi_i^{\text{max}} : n_i.(x) + d_i^{\text{max}} = 0$.

The $k$-DOP volume is then the intersection of all slabs $\bigcap_{1 \leq i \leq k/2} S_i$. 

Collisions
Bounding volumes can be placed inside other bounding volumes, recursively, thereby building *bounding volume hierarchies*.

Bounding volume hierarchies are one approach to improving collision detection performance amongst many objects or between complex objects. Bounding volume hierarchies may reduce collision detection from an $O(n^2)$ algorithm to $O(n lg(n))$ or even $O(n)$. 
Intersection Detection Principles or “Rules of Thumb”

- Perform calculations which allow trivial acceptance or trivial rejection.
- If possible, use results from above tests, even if they fail.
- Try re-ordering rejection and acceptance tests for better performance.
- Postpone expensive calculations.
- Consider reducing dimensionality of problem.
- If many objects are being tested against one object, pre-calculate values if possible.
- Perform timing tests and profiling to investigate performance.
- Make code robust (80% of work!).
Two points $p_1(x_1, y_1, z_1)$ and $p_2(x_2, y_2, z_2)$ intersect if they are coincident:

$$x_1 = x_2, y_1 = y_2, z_1 = z_2$$
Two intervals $A = [a^{\text{min}}, a^{\text{max}}]$ and $B = [b^{\text{min}}, b^{\text{max}}]$ are disjoint (do not intersect) if either $a^{\text{min}} > b^{\text{max}}$ or $b^{\text{min}} > a^{\text{max}}$.

```python
interval_intersect(A, B) returns (\{OVERLAP,DISJOINT\})
if (a^{\text{min}} > b^{\text{max}} \text{ or } b^{\text{min}} > a^{\text{max}}) \text{ then}
    \text{return (DISJOINT)}
else
    \text{return (OVERLAP)}
```
Two spheres intersect if the distance between their centres $c_1$ and $c_2$ is less than the sum of their radii $r_1 + r_2$.

```cpp
sphere_intersect(A, B) returns(\{OVERLAP,DISJOINT\});
l = c_2 - c_1;
d^2 = l.l;
if (d^2 < (r_1 + r_2)^2) then
    return (OVERLAP);
else
    return (DISJOINT);
```

**Algorithm 2: Sphere-Sphere Intersection Testing**

Using distance squared rather than distance avoids a square root calculation. Square roots *used* to be *very* expensive c.f. other operations, and are still *somewhat* expensive c.f. other operations.
The sphere-AABB test also uses a distance test. The distance (squared) from the sphere centre to the box is accumulated in each dimension, then a distance test is applied.

```plaintext
sphere_AABB_intersect(c, r, A) returns ({OVERLAP, DISJOINT});

    \[ d^2 = 0; \]
    for \((i \ \text{in} \ x, y, z)\) do
        if \((c_i < a_i^{\text{min}})\) then
            \[ d^2 = d^2 + (c_i - a_i^{\text{min}})^2; \]
        else if \((c_i > a_i^{\text{max}})\) then
            \[ d^2 = d^2 + (c_i - a_i^{\text{max}})^2; \]
        \if (d^2 < r^2) then
            return (OVERLAP);
        \else
            return (DISJOINT);
        \end{if}
```

**Algorithm 3:** Sphere-AABB Intersection Test
Rays are directed lines. They may be finite, semi-infinite or infinite.

Rays are an important geometric object ("shape") used in intersection detection, (and, of course, ray tracing!). Rays are useful in intersection detection because:

1. They are good models of paths taken by fast moving objects, e.g., lasers or high speed bullets.

2. They can be used as a computationally efficient ("cheap") way to perform approximate intersection detection.

However, we will not consider rays in this course.
When a collision is detected the next step is collision response — working out what happens to the colliding objects. In general proper collision response requires using physics concepts, such as conservation of momentum, conservation of energy, impulses, forces, masses, accelerations etc. However, collision response can be dramatically simplified if objects are hard, and thus not deform, and simply “bounce” off each other. In physics these types of (ideal) collisions are known as specular.
One simple case is a sphere, e.g. a ball, bouncing off a fixed wall. The sphere simply has the appropriate component of its velocity reversed. For example, as below, the ball hits the right wall and thus the $x$ component of its velocity reversed, the $y$ component is unchanged.
Using a more mathematical analysis, the ball has bounced off the wall such that the reflection direction makes an equal angle to normal as the incident direction.

Also note that the velocity has changed only in the normal direction, and is unchanged in the tangent direction.
This case often occurs in computer graphics when spheres are used as bounding volumes for objects. For example, a skier on a ski slope being tested against collisions with trees or boulders. It can be solved in a quite similar way to the ball against wall. Again the incident angle equals the reflection angle. However, the normal is now along the direction between the two spheres centres:
Effectively, the moving sphere bounces off the fixed sphere as though there was a wall in the tangent direction. Solving it requires resolving velocity vectors into components along the normal $N$ direction and its perpendicular $T$. 
Two spheres of equal mass travelling in 1D colliding is quite simple: they just bounce off each other, but with the velocities interchanged. In 2D the problem is again solved by seeing that it like the 1D problem the only direction in which the velocities change is and solving the problem is seeing that at the point of contact along the sphere centres the problem is basically the same as the 1D problem.
The reflection vector $\mathbf{R}$ is calculated according to the *Law of Reflection*: the angle of incidence is equal to the angle of reflection and the reflection vector lies in the plane of incidence.

$$\theta_i = \theta_r$$

The reflection vector may be calculated as follows:
Assume the incident vector $\mathbf{I}$ is shown as a vector pointing towards the incoming direction of the object. This is actually the reverse or negative of the velocity vector $\mathbf{v}$ (be careful about this). Doing so simplifies (and matches the case in lighting where the vector $\mathbf{I}$ points towards the light source).

Any vector may be resolved into a projected component and a perpendicular component.
Reflection Vector III

The projected component of \( \mathbf{I} \) in the direction of the normal vector is

\[
proj_{\mathbf{N}} \mathbf{i} = (\mathbf{N} \cdot \mathbf{I})\mathbf{N}
\]

The perpendicular component of \( \mathbf{I} \) is then found by difference, i.e. vector subtraction

\[
perp_{\mathbf{N}} \mathbf{I} = \mathbf{I} - proj_{\mathbf{N}} \mathbf{I} = \mathbf{I} - (\mathbf{N} \cdot \mathbf{I})\mathbf{N}
\]

The reflection vector is then

\[
\mathbf{R} = \mathbf{I} - 2 perp_{\mathbf{N}} \mathbf{I} = \mathbf{I} - 2[\mathbf{I} - (\mathbf{N} \cdot \mathbf{I})\mathbf{N}] = 2(\mathbf{N} \cdot \mathbf{I})\mathbf{N} - \mathbf{I}
\]