# Box Hax

27.10.2015

Patrik Nilsson, Simon Bothén

DA304A

Data structures and Algorithms

Project

## Overview

We have decided to create a top-down tile shooter game for our project. The choice was that we love these type of games but at the same time we knew that a lot of our past projects have lacked good data structures. It became a simple choice for us.

## Goals

1. Have a player controlled in a top down environment.
2. Have different types of enemies that spawn on a random "spawn tile".
3. Have different types of weapons. Some with higher fire rate and some with higher damage.
4. Have enemies track down the player down using path finding.
5. Have walls that bullet can't go through and walls that they can go through. Note that enemies should profit from this as well.
6. Have the player pick up random weapons that spawn on the map.

## Data structures

These are the most used data structures in our game.

### Hashtable

Hashtable is used in multiple places in our project. Some examples are for keybindings, mapping ASCII level data to it's instance in C#, and in *Spatial Hash Grid* (See more later).

We have chosen the *Hashtable* for multiple reasons. First of all it's average is O(1) on search, insert and remove. This makes it very convenient to use because of it's fast execution. The other reason is that we have lot's of places where we have some sort of key and some sort of data that are related. If we take our keybinder as an example we can see that our key might be MOVE_RIGHT and the data we want to store is an *array* of Keys. In our case MOVE_RIGHT gives us {Key.RightArrow, Key.D}.

### Linked List

*Linked list* is the main data structure to handle a variety amount of elements in the game. As an example, we use it to control bullets. We don't know the amount of bullet that exists in the game at a special time. Linked list can vary in size and keep track of the elements.

The linked list is structured using a *singly linked list* which means that the list can only track next element in the list, not previous element.

When we want to find an element in the list, the list is using a *linear-search* which is not the fastest way to find a element in a list. But we barely use the search method. Lists are mainly used to make the game able to keep track of the elements, or in other words, store variety amount of elements.

### Spatial Hash Grid

In our project we use *Spatial Hash Grid* to reduce the number of collision checks (*collision culler*). It's core is that you divide the *world* into a 2D grid of cells. Then each cell is hashed to a index in an 1D *array* and it is a perfect way to use a hashtable. Then each index that we use as a key in our *hashtable* is mapped to a list of objects (called a bucket) that is currently in that specific cell.

So if we want to check who the player is colliding with for instance we can then ask the *Spatial Hash Grid* to return the bucket that the player's position is hashed to. We're not done with the collision yet because the *Spatial Hash Grid* returns near object not necessarily overlapping once. However the the *Spatial Hash Grid's* job is now complete.

## Algorithms

These are our used algorithms.

### A* search algorithm

The *A* star search algorithm* is used in our project when the enemies want's to find the closest path to the player. The A* algorithm is loosely based on dijkstra's algorithm, which calculates the closests way from point *a* to point *b* in a weighted graph. In our case we have made a *tile based* game, which means that from one tile to its neighbor is always the same, so it doesn't matter what the weight is. If we implemented that the neighbor can go diagonally, it would be more important what the weight on the edge are.

The algorithm calculates the shortest path to the player, and it do the calculation every time it enters a new tile. We don't want it to do the algorithm to often, since it's could be very heavily for the game to handle.