Principles of Computer Game Design and Implementation

Lecture 13
We already knew

• Collision detection – overlap test and intersection test
• Detailed view
• Mid-level view
Outline for today

• High-level view for collision detection
  – Uniform grid
High-Level View

Too many objects in the world problem

- Divide the space into regions
- Check for collisions inside regions
  - An approximation
  - Spatial data structures needed

Stationary objects

Moving objects
Spatial Data Structures

- Uniform grids
  - Implicit grids
- Non-uniform grids
- Arbitrary space partitions

Example: Visibility check as collision detection

Used for collision detection and various other purposes
Uniform Grid

• Split the volume into 3-dimentional cells

• For a *moving object*
  – Identify objects in surrounding cells
  – Test for collision with those objects

From now on all pictures will be in 2D. Same principles apply
Locating Objects

- \( i = \text{(int)} \left( \frac{x}{\text{CellSize}} \right); \quad j = \text{(int)} \left( \frac{y}{\text{CellSize}} \right); \quad k = \text{(int)} \left( \frac{z}{\text{CellSize}} \right) \)
- Array of linked lists
  - Test for collision for every element of the list at \( \text{grid}(i, j, k) \)
Ray Tracing

• Intersection of a *ray* with an object
  – Computer graphics
  – Shooting

Ray = “half-line”

Ray collision detection: which object will it intersect with?
Ray Collision Detection

• One can define mathematically
  – Ray to triangle collision
  – Ray to box collision
  – Ray to sphere collision
  – ...

Grid is ideally suited for tracing rays
Ray Collision Detection in jME

- jMonkeEngine can detect Ray-Geometry collisions
- See Examples coming with the library
Explicit Uniform Grid

Advantages:
• Very fast
• Easy to implement (especially in C, C++)

Disadvantages
• May be difficult in Java (generic /non-generic type mixes)
• Use a lot of memory (proportional to the number of cells)
Spatial Hash

- Represent grids implicitly

\[ i = (\text{int}) \left( \frac{x}{\text{CellSize}} \right); \quad j = (\text{int}) \left( \frac{y}{\text{CellSize}} \right); \quad k = (\text{int}) \left( \frac{z}{\text{CellSize}} \right) \]

```java
class Triple {
    int x, y, z;
    Triple(...) {...};
}

HashMap<Triple, LinkedList<Spatial>> grid;
```
Side Remark: Maps

• How to store *associations* of the form 
  \((key, value)\)?

  – For example,
  
  \((gav, 3)\)
  
  \((mike, 5)\)
  
  \((john, 1)\)

List them

As many elements as pairs
Maps As Lists

• Storing information in a list is memory efficient...
• ... but search is expensive

– Queries like “what age is john” potentially will go through all the stored elements
Hash Function

• Let $h(x)$ maps the key to a number between 0 and N
  – E.g. name -> number of first letter in alphabet
  • gav -> 7
  • john -> 10
  • mike -> 11

Bad idea!
Hash Map

What’s John’s age?
What’s Alice’s age?
Spatial Hash

• Represent grids implicitly

For example,

\[ h(i, j, k) = i + j + k \mod 100 \]

Linked lists of objects

Very bad choice
Good Hash Function

• Ideally, for two keys $k_1, k_2$ there shouldn’t be a clash, that is,

$$h(k_1) \neq h(k_2)$$

• This is impossible to achieve
• Writing a “good” hash function is hard

• Java has in-built support (but you may wish to supply your own implementation of hash function)

(see javadoc on HashMap)
Spatial Hash

Advantages
• Moderate memory use (proportional to the number of objects)
• Fast access
• Easy in Java

Disadvantages
• Slower than array lookup
• Trickier in C/C++
Cell Size

• How fine should the grid be?

Too fine
Too coarse
Too coarse and too fine?
Inadequate

Cell size should roughly be the size of an object.
• Works in some cases
• Does not work in others
Cell Size

• How fine should the grid be?

Too fine
Too coarse
Too coarse and too fine?
Inadequate

Cell size should roughly be the size of an object.
• Works in some cases
• Does not work in others