Outline

• Student Presentations
• Collision Detection Basics
• Collision Shapes
• Collision Detection Algorithms
• Optimizations
• Project Discussion
Outline

- **Student Presentations**
- Collision Detection Basics
- Collision Shapes
- Collision Detection Algorithms
- Optimizations
- Project Discussion
Student Presentations

• Amar Patel:
  • “Advanced Wall Building for RTS Games”

• Josep Valls:
  • “Procedural Modeling of Cities”

• Stephen Ramusivich:
  • “Overview of the OGRE Game Engine”
Outline

- Student Presentations
- **Collision Detection Basics**
- Collision Shapes
- Collision Detection Algorithms
- Optimizations
- Project Discussion
Collision Detection Problems

Why is collision detection important?

http://www.youtube.com/watch?v=mYq9emwQzZU
http://www.youtube.com/watch?v=ER05CrlzdC0
http://www.youtube.com/watch?v=uP47Jro2oAM
http://www.youtube.com/watch?v=cCtBaFcMZ44
http://www.youtube.com/watch?v=VWypWK91T5U
Game Engine Architecture

- Game Specific
- Game Engine Functionalities
- Resource Management
- Utility Layer
- Platform Independence Layer
- SDKs
- OS
- DRIVERS
- HARDWARE
Game Engine Architecture

- Rendering Engine
- Scripting
- Artificial Intelligence
- Online Multiplayer
- Gameplay Foundations (Game Loop)
- Animation Engine
- Physics
- Audio Subsystem
- Collisions
- Profiling & Debugging

Gameplay Foundations (Game Loop)
Collision Detection Basics

• Why?
  • Detect **contact** between game objects (moving objects, or objects vs background)

• How?
  • Each game object (including background) associated with a **collision shape** (simpler geometry than the one used for rendering)
  • Calculate collision between collision shapes

• Typical functionalities:
  • Boolean collision checks
  • Collision volume / area
Collision Shapes

- Geometry used for rendering:
- Geometry used for collision detection:
Implications for Game State

- You need two separate representations of game objects
  - Visual representation: used for rendering
  - Collision representation: used for determining the behavior of the game object in game world.

- Games with simple shapes (e.g. Super Mario), can merge the two:
  - Visual representation: Sprite (bitmap + hotspot), Transform
  - Collision representation: the bounding box of the sprite + the same transform as the visual representation

- In general, you need the two. Even the collision skeletons and the rendering skeletons might be different, if the rendering ones are too complex. But they need to be kept in sync.
Collision World

- For complex games, it is recommended that the collision detection module maintains its own internal state in something called the “Collision World”
Collision World

- For complex games, it is recommended that the collision detection module maintains its own internal state in something called the “Collision World”
Collision World

- From a programming point of view, the Collision Module API should be something like this:

  - Constructor:
    - `collisionWorld(…);`

  - Adding removing objects:
    - `int addObject(CollisionShape cs, Transform t);`
    - `removeObject(int ID);`

  - Updating objects:
    - `void updateObjectTransform(int ID, Transform t);`

  - Collision Queries:
    - `bool collision(int ID1, int ID2);`
    - `List<CollisionData> checkForAnyCollisions();`
    - etc. (whatever functionalities you want to offer)
Collision World

• From a programming point of view, the Collision Module API should be something like this:

  • Constructor:
    • collisionWorld(…);

  • Adding removing objects:
    • int addObject(CollisionShape cs, Transform t);
    • removeObject(int ID);

  • Updating objects:
    • void updateObjectTransform(int ID, Transform t);

  • Collision Detection:
    • bool collision(int ID1, int ID2);
    • List<CollisionData> checkForAnyCollisions();
    • etc. (whatever functionalities you want to offer)

Some game engines, simply have a method called “cycle()”, that checks for all the collisions, and let you specify “callbacks” to each object, so that if a collision happens, the callback gets executed.

This is more efficient, but might be more complex to use.
Collision Detection Output

• The most basic test: Boolean intersection test
• We might want more:
  • Exact area of volume of intersection might not be needed in a game, but...
  • **Point(s) of contact**: approximate center of the collision volume
    • Uses:
      • Rigid body dynamics
      • bullet marks in walls (need point of contact)

• **Separation vector**: vector along the approximate shortest dimension of the volume of contact, along which we can slide the objects to make them not collide
  • Uses:
    • Separate objects before rendering, to avoid overlap (used in sport games)
Outline

- Student Presentations
- Collision Detection Basics
- Collision Shapes
- Collision Detection Algorithms
- Optimizations
- Project Discussion
Collision Shapes

- Spheres
- Capsules
- Bounding Boxes
- Discrete Oriented Polytopes (DOP)
- Arbitrary convex volumes
- “Poly mesh”
- Compound shapes
Collision Shapes: Spheres

- Spheres (or circles in 2D) are the most simple collision shapes
- Defined by just 4 numbers (3 in 2D):
  - Center \((x,y,z)\)
  - Radius \((r)\)
- Very quick simple collision check
Collision Shapes: Capsules

- Capsules are the next easiest shape to perform collisions, and also very easy to represent.
- 2 points and a radius:
  - \(x_1, y_1, z_1\)
  - \(X_2, y_2, z_2\)
  - \(r\)
- Easier to represent and to collide than:
  - Cylinders
  - Boxes
- Commonly used for body parts
Collision Shapes: Bounding Boxes

• Smallest box (cuboid) that encloses a given figure
• 2 types commonly used:
  • AABB (Axis-Aligned Bounding Boxes):
    • Fastest to perform collision detection
    • Might misfit some shapes
  • OBB (Oriented Bounding Boxes)
    • Relatively fast collision detection
    • Better fit for some shapes
Collision Shapes: Bounding Boxes

- **Axis-Aligned Bounding Boxes (AABB)**
  - Represented by just 2 points
  - Recomputed if object rotates

- **Oriented Bounding Boxes (OBB)**
  - Represented by 3 half-radiiuses and a transform
  - Very commonly used
Collision Shapes: k-DOPs

- Discrete Oriented Polytopes
- Given k vectors (each determining an orientation)
- Move a plane perpendicular to each vector as close as possible to the object at hand
- Ex, $k = 6$: 
Collision Shapes: k-DOPs

- Discrete Oriented Polytopes
- Given $k$ vectors (each determining an orientation)
- Move a plane perpendicular to each vector as close as possible to the object at hand

Ex, $k = 6$: 

![Diagram of k-DOPs](image)
Collision Shapes: k-DOPs

- Discrete Oriented Polytopes
- Given $k$ vectors (each determining an orientation)
- Move a plane perpendicular to each vector as close as possible to the object at hand
- Ex, $k = 6$: 
Collision Shapes: k-DOPs

- Discrete Oriented Polytopes
- Given k vectors (each determining an orientation)
- Move a plane perpendicular to each vector as close as possible to the object at hand
- Ex, k = 6:
Collision Shapes: k-DOPs

- Discrete Oriented Polytopes
- Given k vectors (each determining an orientation)
- Move a plane perpendicular to each vector as close as possible to the object at hand
- Ex, k = 6:
Collision Shapes: k-DOPs

- Notice that OBB are a particular case with just 4 vectors (in 2D) and 6 vectors (in 3D), which are aligned with the 3 axis.
- Ex, 4-DOP:
Collision Shapes: Arbitrary Convex Shapes

- A 3D shape defined by an artist by directly creating triangles (e.g. in Maya)
- If it is convex, it is equivalent to a k-DOP
- It can be converted automatically: the normal of each triangle define the k vectors
Collision Shapes: Meshes

- Arbitrary triangle meshes
- This are the most complex, slowest collision shapes
- To check collision:
  - Check each individual triangle for collision!
  - Only collision with the surface
    - Since it might not even be a closed shape!
Collision Shapes: Compound Shapes

- Sometimes it’s easier to represent a complex shape as a composition of simpler shapes, than going for the full triangle mesh.

- Think of compound shapes as skeletons: basic collision shapes connected together.

- Allow many optimizations (later)
Outline

• Student Presentations
• Collision Detection Basics
• Collision Shapes
• Collision Detection Algorithms
• Optimizations
• Project Discussion
Collision Testing

- The collision module will support a subset of the previous shapes.
- For each pair of shapes, a different collision detection algorithm is used.
- Typically a matrix like this is specified:

<table>
<thead>
<tr>
<th></th>
<th>Sphere</th>
<th>OBB</th>
<th>Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>supported</td>
<td>supported</td>
<td>Boolean check only</td>
</tr>
<tr>
<td>OBB</td>
<td>supported</td>
<td>supported</td>
<td>no</td>
</tr>
<tr>
<td>Mesh</td>
<td>Boolean check only</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Collision Testing: Sphere-Sphere

• Easiest possible check:
  • Sphere: \((x,y,z,r)\)

```cpp
bool collision(s1, s2) {
    dx = s1.x - s2.x
    dy = s1.y - s2.y
    dz = s1.z - s2.z
    d = sqrt(dx*dx + dy*dy + dz*dz)
    return (d < s1.r + s2.r)
}
```
Collision Testing: Sphere-Sphere

- Easiest possible check:
  - Sphere: \((x, y, z, r)\)

```cpp
bool collision(s1, s2) {
    dx = s1.x – s2.x
    dy = s1.y – s2.y
    dz = s1.z – s2.z
    d = sqrt(dx*dx + dy*dy + dz*dz)
    return (d < s1.r + s2.r)
}
```

If we need to provide a separation vector, the vector would just be the one connecting the two centers, and the amount we need to move a sphere to avoid the contact is: 
\[ s1.r + s2.r - d \]
Collision Testing: Bounding Boxes

- Separating Axis Theorem
- Already covered last week in a student presentation, so, I’ll just remind the main idea:
Collision Testing: Bounding Boxes

- Separating Axis Theorem
- Already covered last week in a student presentation, so, I’ll just remind the main idea:

If we can find an axis, where the projections of the shapes are disjoint, then the objects are not colliding.
Collision Testing: Bounding Boxes

- Separating Axis Theorem
- Already covered last week in a student presentation, so, I’ll just remind the main idea:

  In general, this is not helpful, since finding that axis might be very hard.

  But for the case of boxes, we can proof that you only need to check axis aligned with each of the faces and edges of the boxes!!

  If we can find an axis, where the projections of the shapes are disjoint, then the objects are not colliding.
Collision Testing: The GJK Algorithm

- If many different shapes are supported, we need special algorithms for each combination of shapes
  - This increases the complexity of the collision detection module

- The GJK (Gilbert, Johnson, Keerthi) algorithm solves this problem for any convex shape
Collision Testing: The GJK Algorithm

• Main idea:
  • “Minkowski difference”
  • Two convex shapes collide only if their Minkowski difference contains the origin.
Collision Testing: The GJK Algorithm

• Main idea:
  • “Minkowski difference”

  \[ A - B \]

  Minkowski difference is the shape that results from taking each pair of points a in A and b in B, and computing: \( a - b \)

  Result looks like sweeping B around the perimeter of A (or the other way around)

• Two convex shapes collide only if their Minkowski difference contains the origin.
Collision Testing: The GJK Algorithm

• Main idea:
  • “Minkowski difference”

- Two convex shapes collide only if their Minkowski difference contains the origin.

If they intersect, there will be at least one point a in A and one b in B which are the same, and thus a – b is the origin of coordinates.
GJK algorithm is an algorithm to check whether a point is inside of a convex hull:

1. Initialize a set \( Q \) with up to \( d+1 \) points from \( C \) (in \( d \) dimensions)
2. Compute point \( P \) of closest to the origin in \( \text{CH}(Q) \)
3. If \( P \) is the origin; return 0
4. Reduce \( Q \) to the smallest subset \( Q' \) of \( Q \), such that \( P \) in \( \text{CH}(Q') \)
5. Let \( V=S_C(-P) \) be a **supporting point** in direction \( -P \)
6. If \( V \) no more extreme in direction \( -P \) than \( P \) itself; return \( ||P|| \)
7. Add \( V \) to \( Q \). Go to step 2

Full algorithm at: [http://realltimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_GJK_notes.pdf](http://realltimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_GJK_notes.pdf)

Following slides from Christer Ericson. Originals at: [http://realltimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt](http://realltimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt)
Collision Testing: The GJK Algorithm

- GJK algorithm is an algorithm to check whether a point is inside a convex hull:
  1. Initialize a set $Q$ with up to $d+1$ points from $C$ (in $d$ dimensions).
  2. Compute point $P$ of closest to the origin in $\text{CH}(Q)$.
  3. If $P$ is the origin; return 0.
  4. Reduce $Q$ to the smallest subset $Q'$ of $Q$, such that $P$ in $\text{CH}(Q')$.
  5. Let $V= S_C(-P)$ be a supporting point in direction $-P$.
  6. If $V$ no more extreme in direction $-P$ than $P$ itself; return $||P||$.
  7. Add $V$ to $Q$. Go to step 2.

- Full algorithm at:

- Following slides from Christer Ericson. Originals at:
  [http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt](http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt)

The most extreme point of a shape in a given direction:

\(d\)
INPUT: Convex polyhedron $C$ given as the convex hull of a set of points
GJK example 2(10)

1. Initialize the simplex set $Q$ with up to $d+1$ points from $C$ (in $d$ dimensions)

$$Q = \{Q_0, Q_1, Q_2\}$$
GJK example 3(10)

2. Compute point $P$ of minimum norm in $\text{CH}(Q)$

$$Q = \{Q_0, Q_1, Q_2\}$$
GJK example 4(10)

3. If $P$ is the origin, exit; return 0
4. Reduce $Q$ to the smallest subset $Q'$ of $Q$, such that $P$ in $\text{CH}(Q')$

$$Q = \{Q_1, Q_2\}$$
5. Let $V = S_C(-P)$ be a supporting point in direction $-P$

$$Q = \{Q_1, Q_2\}$$
GJK example 6(10)

6. If V no more extreme in direction \(-P\) than \(P\) itself, exit; return \(||P||\)
7. Add V to Q. Go to step 2

\[ Q = \{Q_1, Q_2, V\} \]
2. Compute point $P$ of minimum norm in $\text{CH}(Q)$

$$Q = \{Q_1, Q_2, V\}$$
GJK example 8(10)

3. If $P$ is the origin, exit; return 0
4. Reduce $Q$ to the smallest subset $Q'$ of $Q$, such that $P$ in $\text{CH}(Q')$

$$Q = \{Q_2, V\}$$
GJK example 9(10)

5. Let \( V = S_c(-P) \) be a supporting point in direction \(-P\)

\[ Q = \{Q_2, V\} \]
GJK example 10(10)

6. If $V$ no more extreme in direction $-P$ than $P$ itself, exit; return $||P||$

$$Q = \{Q_2, V\}$$
Moving Bodies

- Or “bullet through paper” effect:

If objects move too fast for the size of the objects in the world, they might go “through” other objects, without any collision being detected.
Moving Bodies: Swept Shapes

• Swept shape: a new shape formed by the motion of a primitive shape

• Some basic shapes generate easy swept shapes:
  • Spheres generate capsules

• In general, a convex shape generates another convex shape
Moving Bodies: Swept Shapes

• Problems:
  • Non rectilinear movement

• Rotation

Non convex swept shapes!
Moving Bodies: Swept Shapes

- Problems:
  - Non rectilinear movement
  - Rotation

Possible solution: use a smaller simulation time interval, and approximate with linear motion.
Non convex swept shapes!
Moving Bodies

• Swept shapes offer a quick solution that would work for 95% of the games

• If more precision is required, there are specialized algorithms:
  • Based on the Minkowski difference, we can calculate what is the minimum distance between objects in the world, and calculate what is the maximum time step we can simulate to ensure no “bullet-through-paper” effect.
  • Specialized algorithms: Continuous Collision Detection (CCD)
    • See:  
      http://www.continuousphysics.com/BulletContinuousCollisionDetection.pdf
Outline

• Student Presentations
• Collision Detection Basics
• Collision Shapes
• Collision Detection Algorithms
• Optimizations
• Project Discussion
Optimizations

• **Temporal coherency:**
  - When using GJK algorithm we know the distance between two objects:
    - If they have not moved more than that amount in a series of frames, no point to testing collision again.

• **Spatial Partitioning:**
  - Divide the world using some hierarchical scheme (for example: octtrees)
  - Each object belongs to some nodes of that partition.
  - Unless two objects belong to the same partition, no need to check for collision
  - Instead of check for N*N tests (N is number of objects), just go over the nodes in the octtree, and only check collision between the nodes present at each node.
Spatial Partitioning Example

9 objects: 72 potential collisions (ignoring background)
## Spatial Partitioning Example

9 objects: 36 potential collisions (ignoring background)
After partitioning in 4 areas, only 13 potential collisions

<table>
<thead>
<tr>
<th>0 objects: 0 potential collisions</th>
<th>1 object: 0 potential collisions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1.png" alt="Image" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 objects: 3 potential collisions</th>
<th>5 objects: 10 potential collisions</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Spatial Partitioning Example

9 objects: 36 potential collisions (ignoring background)
After partitioning in 4 areas, only 13 potential collisions

Can be done recursively a few levels, to obtain drastic performance gains. In 2D: quadtrees, in 3D: octtrees (student presentation)
Optimizations

• Broad Phase, Midphase and Narrow Phase:
  • Broad phase:
    • Use space partition to determine potential collisions
  • Midphase:
    • Use bounding shapes (spheres, capsules, AABB) of compound shapes to narrow down potential collisions (e.g. one big AABB for a whole character skeleton)
  • Narrow phase:
    • Check all the individual shapes that are still candidates
Outline

- Student Presentations
- Collision Detection Basics
- Collision Shapes
- Collision Detection Algorithms
- Optimizations
- **Project Discussion**
Libraries For Collision Detection

• **ODE (Open Dynamics Engine):**
  • General physics simulation engine
  • Can be used also just for collision detection

• **SWIFT**
  • Collision between convex shapes

• **RAPID**
  • Collision between convex and non-convex shapes
Project Topics

• Compound shape collision detection (using AABB and midphase narrow-phase detection)

• Moving shapes collision detection (swept shapes, continuous collision detection)

• Collision detection amongst arbitrary convex shapes (GJK algorithm)

• Efficient collision detection allowing triangle meshes

• Etc. (only basic collision between spheres (circles) and boxes would be way too simple)
Links to Game Videos

• This week’s interesting games:
  • Crush: http://www.youtube.com/watch?v=FdGwPQs64Rg
Remember that next week:

• First Project Deliverable:
  • Groups
  • Topics
  • Specific algorithms
  • Specific libraries you will use
  • Specific structure of your game engine (not necessarily a class diagram, see the diagrams shown in class)
  • Demo you will create

• Submission procedure:
  • Email to (copy both):
    • Santiago Ontañón santi@cs.drexel.edu
    • Stephen Lombardi sal64@drexel.edu
  • Subject: CS480-680 Project Deliverable 1 Group #