Rendering Optimizations and Low-Level Collisions

CPSC 386
Eric May

Collision Detection - Intro

- The standard scenario aided by collision detection is a player with a bounding volume (aka collider) in a room, with walls and obstacles that cannot be traversed through
  - The standard test is to check for intersection between the objects' BVs
  - If the player is, for example, running at an angle into a wall, then they may move at an oblique angle
  - Another example is a truck driving over terrain; care should be made that the vehicle is kept moving over the terrain (falling through is bad!)
  - Or to check if projectiles have hit a unit, etc
- The nature of our collision detection algorithms depends on the type of objects involved + required information

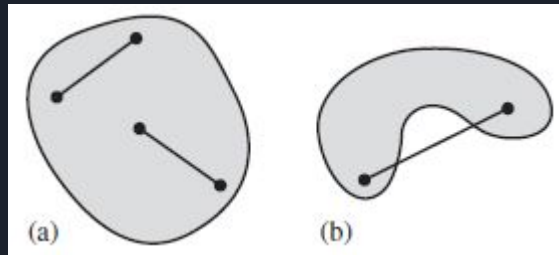
Collision Detection - Intro

- The different broad groups are:
  - Stationary objects, when both objects are not moving
    - There are test-intersection queries, where we only detect **if** they intersect, or we can use..
    - Find-intersection queries, where we detect the intersection set of the objects. The set will be empty if the objects don't intersect.
  - Moving objects, where one or both objects are moving
    - If both are moving, we subtract the velocity of the first from the velocity of the second and treat it as if it were one moving and one stationary object
      - There will be a time interval where the query applies, ie range $[0, t_{max}]$
      - If the objects intersect in the interval, they do so at time t_{first} in $[0, t_{max}]$. Referred to as contact time, the intersecting set referred to as contact set
    - As with stationary objects, we can either determine if they collide, or which portions collide

Collision Detection - Intro

- Due to complexity of distance calculations, one alternative commonly used is *pseudodistance*
  - Positive when objects are separated, zero when they're resting against each other, and negative when they intersect
    - Positive magnitude depends on distance between them
    - Negative magnitude depends on relative volume intersecting
  - A popular method for this is the *method of separating axes*
- One common assumption for ease of development and optimization is that all objects with collision checks are convex
  - Items that are not convex can be represented as a collection of convex items

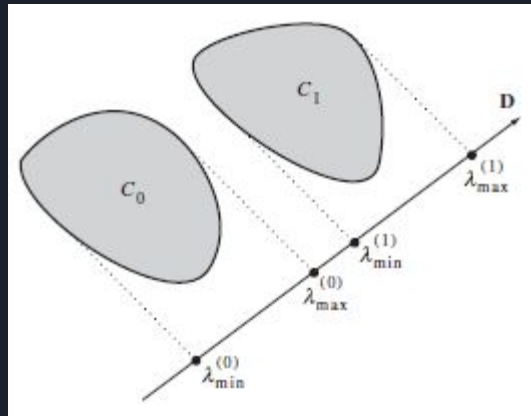
- Picking, or selecting an object(s) based on intersection with a ray, is not typically grouped with collision detection, but it is somewhat related



Collision Detection - Separating Axes

- We can determine that two convex objects do not intersect if there is a line, where their projections do not overlap
  - We call this line a *separating line*, or *separating axis*

- Algebraically, the equation for a verified non-intersection is as follows $\lambda_{\min}^{(0)}(\mathbf{D}) > \lambda_{\max}^{(1)}(\mathbf{D}) \quad \text{or} \quad \lambda_{\max}^{(0)}(\mathbf{D}) < \lambda_{\min}^{(1)}(\mathbf{D})$
- The calculations don't change based on the length of D, or whether we use +/- D
  - The direction vector for our separating line is called our *separating direction*
  - Since objects are stored as a collection of vertices, the process of finding the projected minimums and maximums will involve iterating through our vertices, and projecting them to the separating line
  - There exist algorithms that perform this in O(n) and O(log n) time. For smaller objects, the O(n) algorithm may be faster, this threshold is called the *break-even point*



Collision Detection - PseudoDistances

- Calculating the distance between objects can be very complex if we only think in terms of the surface
  - By breaking shapes down into their components, we can make mathematically sound, yet simple deductions
- Sphere-Swept volumes (spheres + capsules)
  - The squared distance is cheaper than taking a square root, which the alternative requires
  - When considering two spheres, we simply take the distance between their centers (point to point), and divide it by the sum of their radii. We then subtract by 1.
    - If the result is positive, they are separated, if it's 0, they're just touching, negative means intersection
  - If we substitute one of the spheres with a capsule, we take the distance between the center of the sphere, and the line segment composing the capsule. Then we subtract the radii as before.
  - Chapter 14 discusses the distance functions for various shapes in great detail
    - (3D Game Engine Design, 2nd Ed., David H. Eberly, Morgan Kaufmann ISBN: 0122290631)

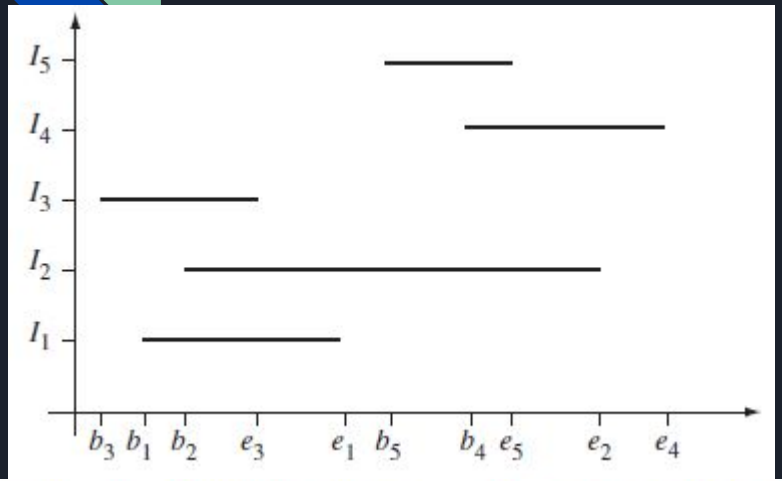
Collision Detection - PseudoDistances

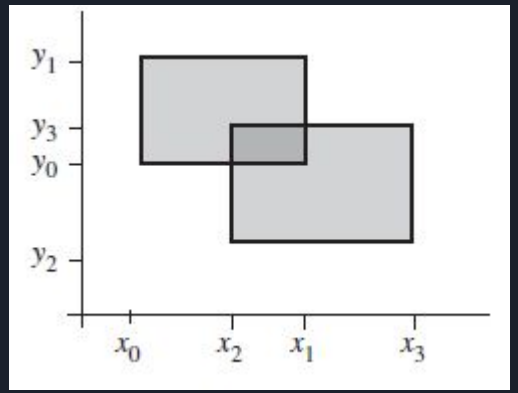
- Shape calculations cont'd
  - For a test between a sphere and a cube, we get the distance from the sphere's center to the cube, divide by the radius and subtract by one
    - Notably different: Now we're just using the radius of the sphere, instead of adding the radii of two sphere-swept shapes
  - For a capsule and a cube, we would repeat above except we would be calculating the distance between the capsule's line segment, instead of just a point
  - For convex polygons/polyhedra, we project the objects onto N different potential separating axes
    - If all axes overlap, use the **largest** one for the pseudodistance
    - If a single axis detects no overlap, a test-intersection query will be happy, a calculation that gives pseudodistance will want to continue checking all axes to find the largest amount of clearance

Collision Detection - Collision Culling

- Detecting contact between objects is sometimes referred to as the *narrow phase* of collision detection
  - For performance, we want an extremely efficient way to eliminate objects that aren't even close to intersecting
  - One way to do this is to give each polyhedron an Axis-Aligned Bounding Box
    - We sweep through the universe, considering one axis at a time, sorting each object's beginning and ending positions
    - When we encounter the beginning of an object, we add it to the list of current objects. Any objects that are already in the queue will report a collision with the new object
      - When we encounter the end of an object, we remove it from the active set
    - The sweep reports all intersecting pairs. This process is then repeated for other axes; if any of the axes report separation, then the more expensive collision tests aren't used
    - Since the objects are sorted by spatial coherence, future updates should use an algorithm that cheaply processes a mostly in-order list

Collision Detection - Collision Culling





1. b_3 encountered. No intersections reported since A is empty. Update $A = \{I_3\}$.
2. b_1 encountered. Intersection $I_3 \cap I_1$ is reported. Update $A = \{I_3, I_1\}$.
3. b_2 encountered. Intersections $I_3 \cap I_2$ and $I_1 \cap I_2$ reported. Update $A = \{I_3, I_1, I_2\}$.
4. e_3 encountered. Update $A = \{I_1, I_2\}$.
5. e_1 encountered. Update $A = \{I_2\}$.

Applying this to a 3d environment just involves one more axis for consideration

Culling and Clipping

- Pushing information through our geometric pipeline entails a lot of operations, so determining which objects we can exclude from it will optimize our performance
- Only objects (or portions of objects) that are not visible are removed
  - This can be objects outside our view frustum (*object culling*) or surfaces facing away from the camera (*back-face culling*)
- In order to be less expensive enough to impact performance, objects are given *bounding volumes* that approximate the space an object occupies
  - If the bounding volume intersects the view frustum (or camera volume), the associated object is processed
  - Not very precise, some scenarios where bounding volume is in, but object is outside
  - But **fast!**

Culling and Clipping

$$\text{Cost}(\text{inexact\_culling}) < \text{Cost}(\text{exact\_culling})$$

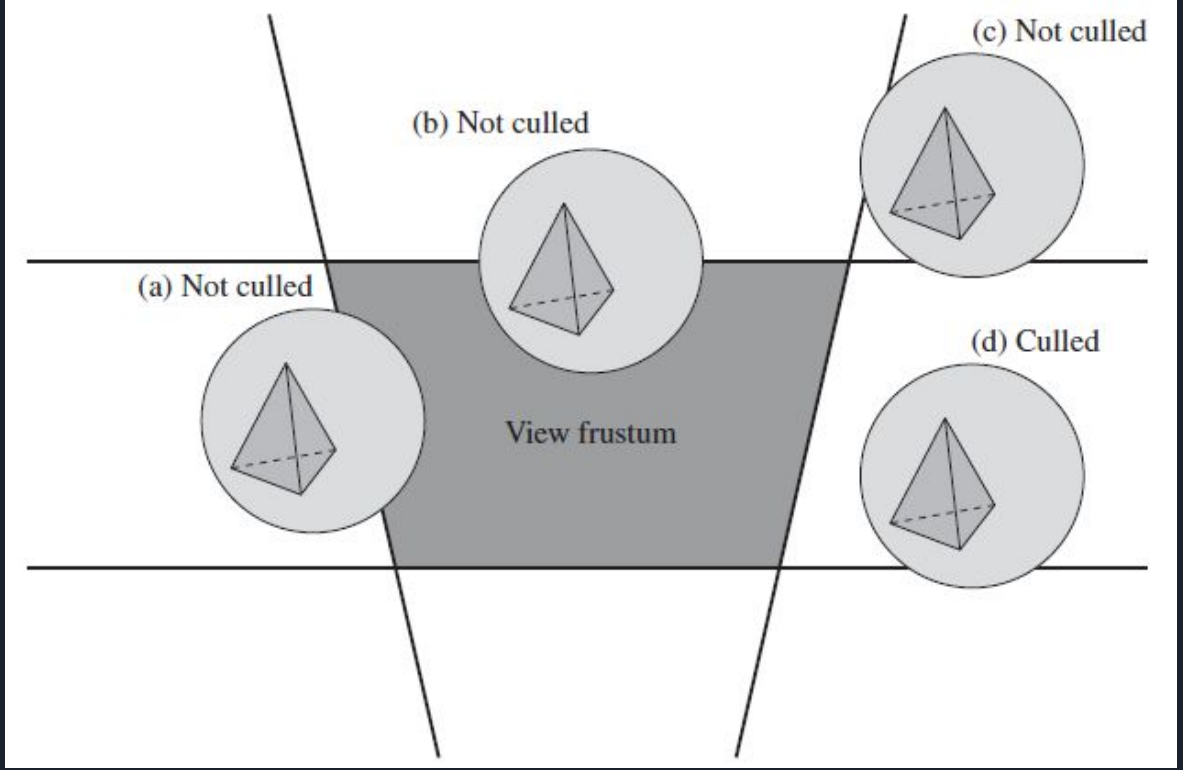
$$\text{Cost}(\text{inexact\_drawing}) > \text{Cost}(\text{exact\_drawing})$$

$$\text{Cost}(\text{inexact\_culling}) + \text{Cost}(\text{inexact\_drawing}) < \text{Cost}(\text{exact\_culling}) + \text{Cost}(\text{exact\_drawing})$$

- A precise test of the intersection between the view frustum and the bounding volume is called an *exact culling test*
- An *inexact culling test* is designed to be faster, at the expense that it sometimes reports intersections where there are none
- The hope is that the time saved by using the inexact test compared to the exact test
- Relationship between (in)exact culling and drawing will be displayed above

- First two lines are always true, third line is ideally true.
  - We can't know for sure without comparing the performance of both implementations

- Progressively more precise tests are applied:
  - Axis-Aligned Bounding Volumes <- Least Accurate, Fastest
  - Bounding Volumes
  - Actual geometry <- Most Accurate, Slowest
- Any information discarded by less accurate tests is still assured to be valid, it just keeps more unnecessary data



- The typical process for inexact object culling is to compare the bounding volume to the frustum planes one at a time
- (a) would not be culled under either inexact or exact object culling, because the bounding volume intersects the view frustum. (c) would be culled using exact object culling, one of our false positives

Level of Detail

- Since games became 3d, there have been more and more spectacular levels of detail with 3d modeling, with breathtaking results
  - When we're dealing with a face, having thousands of polygons is worth the extra rendering time it costs
  - On the other hand, if we're talking about a distant object, that burden can't be justified
    - If an object takes up a few pixels, the number of triangles processed should be proportional
  - The solution is geometric level of detail (LOD), where models are made as precise as they have to be
  - Rendering time is the primary concern, but there are other applications, such as creating a mesh for collision detection
    - A more coarse representation of a detailed model would perform much better

Level of Detail - Sprites and Billboards

- The simplest representation of a 3d object is to use a pre-rendered image of that object, aka a *sprite*
  - Removes nearly all vertices, causing it to draw much more quickly!
  - Downside is that movement of the eye point will quickly display the flimsy nature of the representation as the perspective doesn't change
    - Which means it's still pretty good for distant objects
  - Can be dealt with by holding renderings of the object at different angles, or always orient the image based on the eye point location and orientation
    - In the latter case, we call the sprite a *billboard*
    - Billboards are extremely useful for particle systems, where large numbers of simple geometric primitives are rendered for a singular visual effect, such as smoke or water droplets

Level of Detail - Sprites and Billboards

- A billboard consists of its origin, two axes representing edge directions, and a normal vector, as well with a textured image to be displayed
  - It can be oriented such that the normal vector can be screen aligned or axis aligned.
    - Screen alignment works for particle systems like smoke, model-space up vector is aligned with the screen up vector
    - Axis alignment is for items like trees, where the billboard's rotation shouldn't change if the viewer tilts their head. Billboard rotates around its up vector so its normal vector aligns with the vector from the eye point to its projection onto the billboard's up axis
  - Alignment of a billboard relies on identifying a coordinate frame, and changing it with respect to the eye point's coordinate frame. Kept in mind, this means we can create a billboard class derived as a special type of Node, where the children can be arbitrary objects, not just flat polygons.

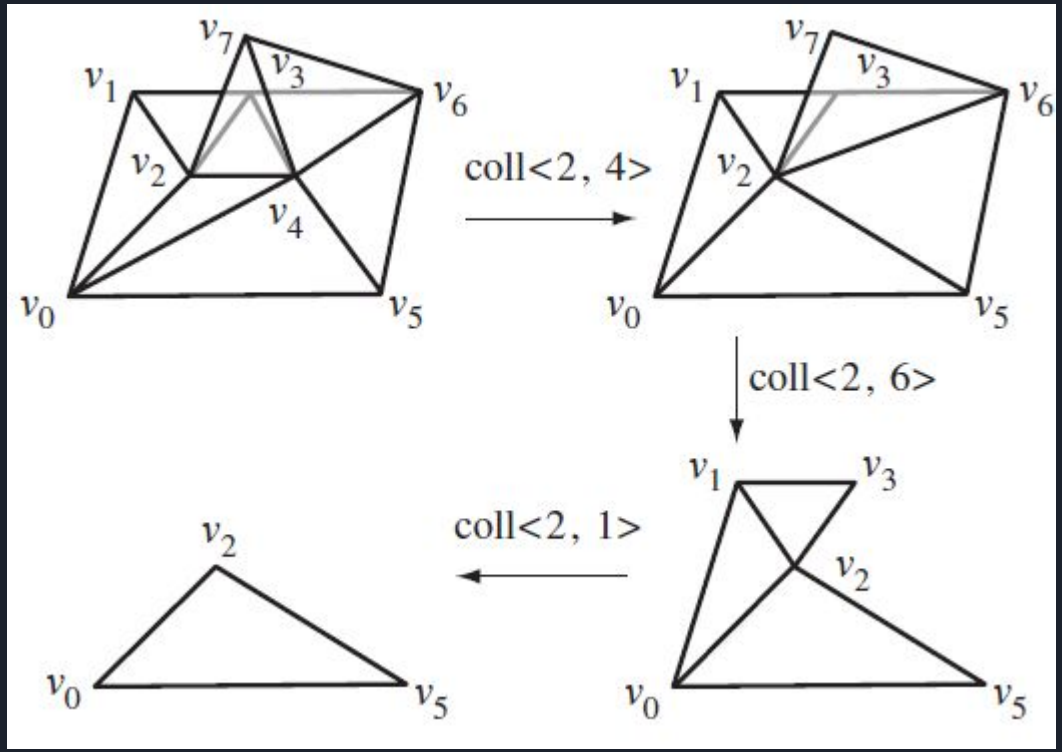
Level of Detail - Discrete LoD

- While 2d representations work for distant objects, there's a big gap between that distance and a close-up suitable for a detailed face
- We can represent an object as a set of triangle meshes with varying precision
  - Stored as a switch node in our scene graph; we choose one out of a list of options
  - We determine which model we use based on the distance between the eye point and our Node's center point
    - The model with most triangles will be rendered when they're at their closest, diminishing until we get our least detailed model rendered at the furthest distance
  - A few downsides:
    - At the distance where we swap models, there will be a noticeable popping between models. We can avoid this by smoothly interpolating between models, but very computationally expensive; results not always in line with expectations
    - Requires artists to create multiple models independent of each other, with same center

Level of Detail - Continuous LoD

- An alternative to providing multiple models is continuous level of detail
  - We're going to reference the Garland-Heckbert algorithm, which builds a large sequence of models obtained by simplifying the most detailed mesh progressively
    - Two large benefits: Less noticeable popping, and reduced need for assets
    - We're not just concerned about geometry, but also other vertex attributes
  - The sequence of changes we follow to change our level of detail consist of contracting vertices in a way that tries to preserve geometric information about the model, with a few choices:
    - Vertex decimation involves removing a vertex and adjacent triangles, then retriangulating the hole left by the removal
    - Vertex clustering involves grouping very close vertices into a single vertex, then removing and adjusting the triangles as necessary
    - Edge contraction involves replacing two vertices and the edge between them with a vertex, with subsequent adjustments to the mesh

Level of Detail - Continuous LoD



Level of Detail - Continuous LoD

- Terrain presents a unique problem, as it is (typically) always present at multiple levels of detail
  - We have two components, computation wise: 1) Calculate the triangles to be drawn, and 2) Draw the triangles
    - If we focus too much on 1, we starve our renderer
    - If we don't discard enough information, then our renderer will take a ton of time per frame
  - As technology has grown, it's become more common to separate terrain into chunks or pages, so the active pages of terrain are loaded for quick access
    - Loading and discarding of pages simply calculated using camera position, orientation, and speed
    - Programmable GPUs have allowed for algorithms that use the GPU to handle clod
- By using fractal-based geometry, there is a lot of flexibility as it pertains to scaling, giving us infinite level of detail. Practically bounded by number of on-screen triangles