UCLA

CS174A : Introduction to Computer Graphics

Royce 190
TT 4-6pm

Scott Friedman, Ph.D
UCLA Institute for Digital Research and Education

Picking and Selection

- Using the mouse to select an item.
- There are several methods available.
- The basic ones
  - The old glSelect() mechanism
  - Color buffer picking
- More advanced
  - Occlusion query extension
  - Ray casting

Picking and Selection

- glSelect()
  - Has been in OpenGL from version 1.0
    - Is very slow
      - Implemented in software
      - Stalls the hardware rendering pipeline
    - Is depreciated after version 3.0 (basically gone now)
  - The basic idea is simple
    - Use integers to "name" objects in the scene
    - Mark the stream of geometry with those "names"

Picking and Selection

- glSelect()
  - Works by rendering the scene in a special mode.
  - A buffer is allocated the collects “hits”
  - A “hit” is any “name” intersecting the view volume.
  - The function gluPickMatrix() helps by restricting the projection to an area around the mouse click.
    - Say, an area of 4x4 pixels to introduce some fuzz.
  - Any primitives rendered *after* a “name” object has been defined in placed into a predefined hit buffer.
  - The “name” objects in the hit buffer tell us what was under the mouse.

Picking and Selection

- Color Buffer Picking
  - Much simpler and faster.
  - Here an additional rendering pass is made.
  - Every object we wish to identify is assigned a unique color.
    - Very simple fragment shader – directly assign color.
  - No lighting or texture mapping performed.
  - Otherwise scene rendered normally.
    - Z-buffer on.
    - Double buffering required. (or render into a dedicated color buffer for picking)
    - Only draw objects we wish to consider for picking.

Picking and Selection

- Color Buffer Picking
  - Once render pass is complete.
  - We *do not* swap the buffers!
  - Call gl.readPixels() to recover result.
    - Using mouse/window x, y position.
  - The pixel color value retrieved corresponds to the object underneath the mouse.
  - A variation of this technique uses a separate color buffer rendered in parallel to the normal buffer.

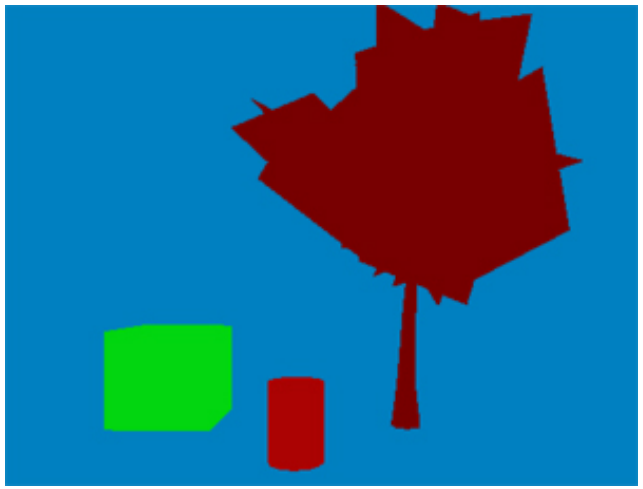
Picking and Selection

- Color Buffer Picking
  - Further enhancement (speed-up)
    - rendering only bounding volumes for scene objects.
    - Must be sure not to try and identify more objects than there are bits of precision in the buffer.
    - Only need to render objects that need to be considered for selection.
  - Problematic for objects with transparent textures.
    - Manageable with some effort

```
// Reading back color value
Glubyte pixel;
glReadPixels( mouseX, mouseY, 1, 1, GL_RGBA, GL_UNSIGNED_BYTE, &pixel );
```

Picking and Selection

- Color Buffer Picking – Example
  - Notice how a completely transparent texture would pose a problem?
  - We haven't talked specifically about textures with alpha values – what needs to be considered?



Picking and Selection

- Other techniques
  - Occlusion query – special case
    - We want to know if something will be visible **before** we render it.
      - » Why would we want to do this?
    - Reading z-values instead of color.
    - Allows determining order of objects under mouse
    - Harder to do region queries (e.g. rubber band box)
  - Ray casting
    - Requires objects to be in memory – in some form
    - Usually, bounding solids used
    - Does not work at pixel level
    - Can have trouble with transparency (e.g. picking a hole)
  - Many, many approaches – solutions are outcome driven.

Collision Detection

- A large a complex topic
  - Object Representation and Bounding Volumes.
  - Simple Intersection Tests
  - Bounding Volume Hierarchies.
  - Interested in…
    - Whether object A has collided with object B.

Collision Detection

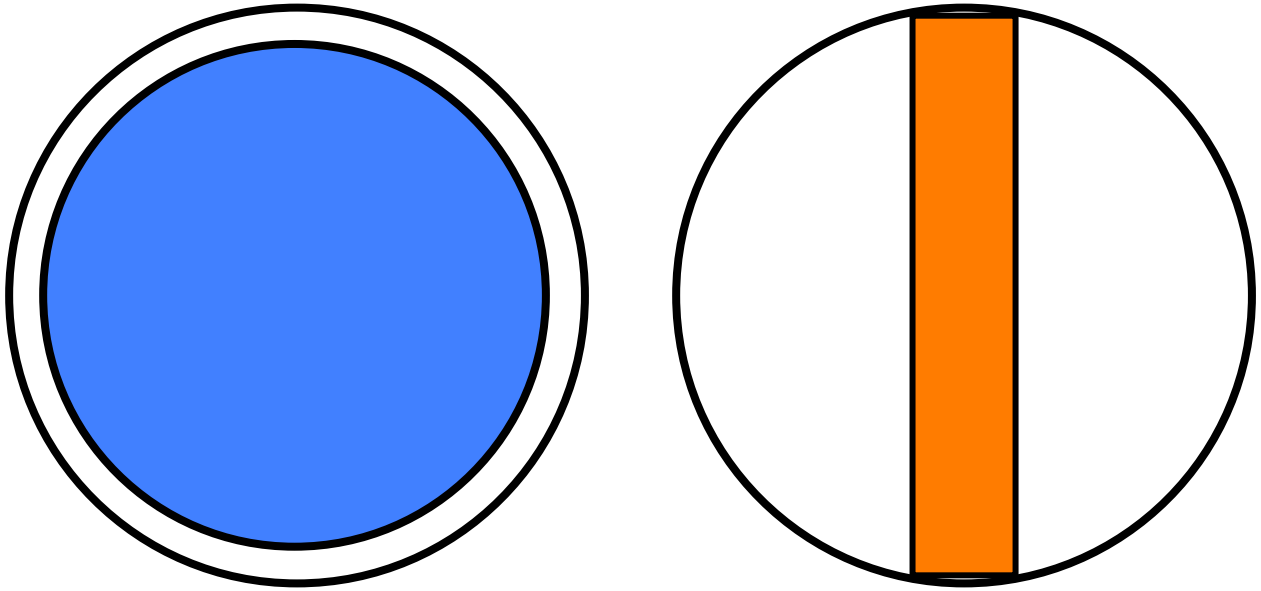
- Object Representation
  - We perform collision detection by computing whether one primitive intersects another.
  - We could do this by comparing all triangles in a scene with each other.
    - This would be slow and produce a lot of useless results.
  - Stick to objects in scene we care about.
    - Still checking objects not colliding with anything.
    - Say we have ten objects that potentially hit each other made up of 100 triangles each
    - Several tens of thousands of comparisons required!
  - Need to be smarter about this…

Collision Detection

- Object Representation
  - Has to be a better way
  - There is, it is called a bounding volume.
    - A sphere is the simplest.
      - Not an exact representation, but…
      - Now we only need around 50 comparisons!
      - If two spheres intersect a more thorough check can then be performed.
        - » We spend time only when we need to
      - Or, test itself can be sufficient if objects are far enough away and close inspection is not necessary.

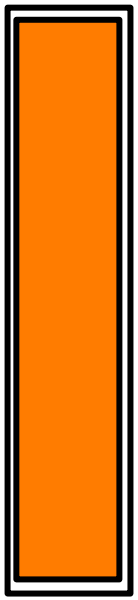
Collision Detection

- Object Representation
  - A good fit is desirable.
  - Spheres, unfortunately, are not always a good choice.



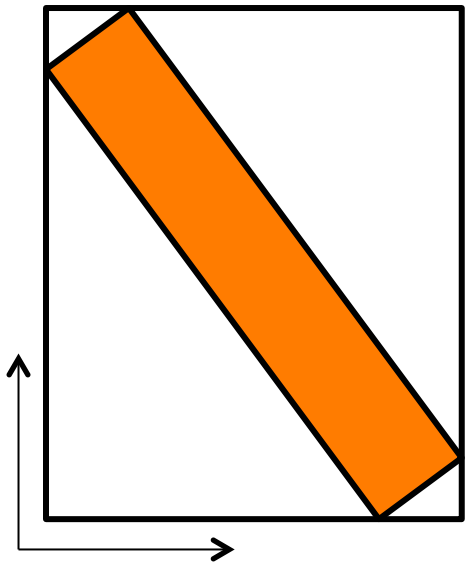
Collision Detection

- Object Representation
  - Sometimes a bounding rectangle would be better.
  - More complex to perform intersection tests, however.



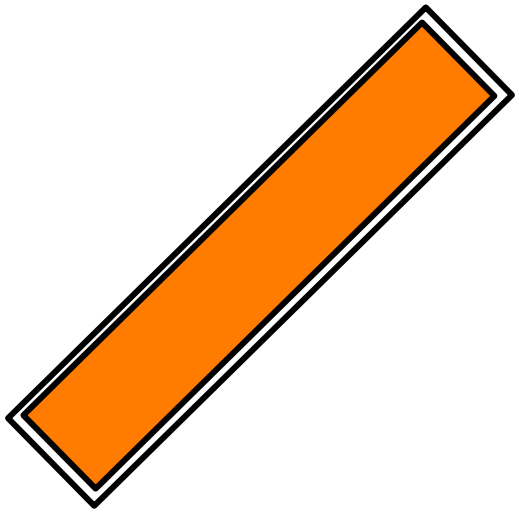
Collision Detection

- Object Representation
  - Sometimes even a bounding rectangle doesn't work.
  - Simplest box is called an axis-aligned bounding box AABB



Collision Detection

- Object Representation
  - An oriented bounding box may be best.
  - *Even more* complex to perform intersection tests, however.
  - As you can see – there are tradeoffs to be considered.



Collision Detection

- Object Representation
  - Which simplified bounding shape to use depends on how accurate you need to be and speed.
  - Whichever one you use there is always a need to determine distance. Which in 3D is computed…

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

  - Hacker 101: When comparing distances we can avoid the square root operation for speed!

Collision Detection

- Object Representation
  - Desirable Properties
    - Inexpensive Intersection Tests
    - Tight Fitting
    - Inexpensive to Compute Bounding Volume
    - Easy to Rotate and Transform
    - Uses Little Memory
  - We want to use inexpensive tests before resorting to more expensive tests.

Collision Detection

- Axis Aligned Bounding Box
  - Simple to compute
  - Several ways to represent them
    - Min / Max extreme points (six floats)
    - Min point and extents (six floats or three floats + 3 half)
    - Center point and half-widths (same as above)
      - » Last two are most efficient memory wise.
  - Computationally,
    - Min-extent is slowest, requiring the most operations.
  - Building an AABB is straightforward regardless.

Collision Detection

- Axis Aligned Bounding Box
  - Example, intersection of two AABB

```
struct AABB {
  Point min;
  Point max;
};

int testAABB( AABB a, AABB b )
{
  // Exit if separated along an axis
  if ( a.max[0] < b.min[0] || a.min[0] > b.max[0] ) return 0;
  if ( a.max[1] < b.min[1] || a.min[1] > b.max[1] ) return 0;
  if ( a.max[2] < b.min[2] || a.min[2] > b.max[2] ) return 0;
  // Overlapping on all axes means there is an intersection
  return 1;
}
```

Collision Detection

- Bounding Sphere
  - Simple intersection test…

```
struct Sphere {
  Point c;
  float r;
};

int testSphere( Sphere a, Sphere b )
{
  // Calculate squared distance between centers
  Vector d = a.c – b.c;
  float dist2 = Dot( d, d );
  // Spheres intersect if squared distance is less than
  //   squared sum of radii
  float radiusSum = a.r + b.r;
  return dist2 <= radiusSum * radiusSum;
}
```

Collision Detection

- Bounding Sphere – be careful
  - Computing the bounding sphere itself, not so simple…
  - A naïve brute force algorithm runs in **$O(n^5)$!**
  - Ritter, describes an iterative algorithm.
    - Start with a sphere based on two points.
    - Iteratively add points to grow.
    - Does reasonably well except
      - » Quality is sensitive to the order points are added.
  - Welzl, uses computational geometry to achieve an expected $O(n^1)$ time. ☺
    - Implementation is complex, however. ☹

Collision Detection

- Bounding Sphere - Ritter

```
// given sphere s and point p, update s to just encompass p
void SphereOfSphereAndPt( Sphere &s, Point &p )
{
  // compute squared distance between point and sphere center
  Vector d = p – s.c;
  float dist2 = Dot( d, d );
  // only update s if point p is outside it
  if ( dist2 > s.r * s.r ) {
    float dist – sqrt( dist2 );
    float newRadius = ( s.r + dist ) * 0.5f;
    float k = (newRadius – s.r) / dist;
    s.r = newRadius;
    s.c += d * k;
  }
}

Void RitterSphere( Sphere &s, Point pt[], int numPts )
{
  // get sphere encompassing two approximately most distant pts
  SphereFromDistantPoints( s, pt, numPts );
  // grow sphere to include all points
  for ( int i=0; i<numPts; i++ )
    SphereOfSphereAndPr( s, pt[i] );
}
```

Collision Detection

- Processing
  - Need a data structure to hold BVs
  - Organization depends on application
    - Maybe all you care about is your spaceship hitting an asteroid?
    - Then all you want to check is the spaceship against the asteroids.
    - You would not bother with asteroid/asteroid checks.
  - Advanced – what about hierarchies of bounding volumes?
    - If represented objects move these bounding volumes of bounding values have to be updated.

Collision Detection

- Processing
  - There are many types of bounding objects
  - Spheres and AABB are suitable for the project
    - Others get more complex - not worth with the time you have available.
  - You may also want to sphere against a plane
    - Useful to know if eye/camera has hit something
    - Again, very simple

Collision Detection

- Processing

```
struct Sphere {
  Point c;
  float r;
};

struct Plane {
  Vector n;     // plane normal, points on plane satisfy
  float d;      // d=Dot(n,p) for a given point p on the plane.
};

int testSpherePlane( Sphere s, Plane p )
{
  // for a normalized plane (|p.n|=1), evaluating the plane equation
  //  for a point gives the signed distance of the point to the plane
  float dist = Dot( s.c, p.n ) – p.d;
  // if sphere center within +/- radius from plane, plane intersects sphere
  return Abs( dist ) < s.r;
}
```

Collision Detection

- Processing
  - Weird way of representing a plane?
  - Not really, just compact.

```
struct Plane {
  Vector n;     // plane normal, points on place satisfy
  float d;      // d=Dot(n,p) for a given point p on the plane.
};

Plane computePlane( Point a, Point b, Point c )
{
  // given three noncollinear points (CCW), compute plane equation
  Plane p;
  p.n = Normalize( Cross( b – a, c – a ) );
  p.d = Dot( p.n, a );
  return p;
}
```

Collision Detection

- Bounding Sphere
  - Ritter is suitable for this class.
    - It is easy to implement and understand.
  - We can help you with other types – keep simple.

- Picking and Collision detection are each considered advanced topics for your projects.