# PA199
# Advanced Game Design

Lecture 6

Collision Detection

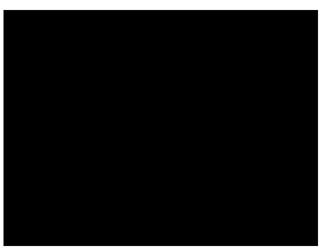
Dr. Fotis Liarokapis

30th March 2016

## Motivation

- Techniques for collision detection depend on the type of game
- For many games rough approximations are fine
  - i.e. Arcade-style games
- For more complex games need to be familiar with a variety of techniques ranging from simple to complex
  - i.e. 3D games

## Rough Approximations Example



## Collision Detection

- Do objects collide/intersect?
  - Static
  - Dynamic
- Picking is simple special case of general collision detection problem
  - Check if ray cast from cursor position collides with any object in scene
  - Simple shooting
    - Projectile arrives instantly, zero travel time

## Collision Detection .

- A better solution
  - Projectile and target move over time
  - See if collides with object during trajectory

## Collision Detection Applications

- Determining if player hit wall/floor/obstacle and stop them walking through it
  - Terrain following (floor)
  - Maze games (walls)
- Determining if projectile has hit target
- Determining if player has hit target
  - Punch/kick (desired)
  - Car crash (not desired)

## Collision Detection Applications .

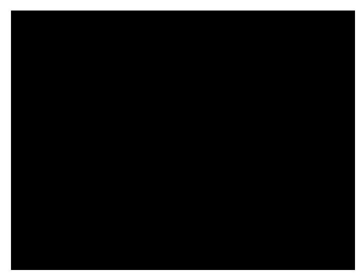
- Detecting points at which behavior should change
  - Car in the air returning to the ground
- Cleaning up animation
  - Making sure a motion-captured character's feet do not pass through the floor
- Simulating motion
  - Physics, or cloth, or something else

## Simulating Motion

## Why it is Hard?

- Complicated for two reasons
  - Geometry is typically very complex
    - Potentially requiring expensive testing
  - Naïve solution is O(n2) time complexity
    - Since every object can potentially collide with every other object

## Why it is Hard - Example



## Basic Concepts

## From Simple to Complex

- Boundary check
  - Perimeter of world vs. viewpoint or objects
    - 2D/3D absolute coordinates for bounds
    - Simple point in space for viewpoint/objects
- Set of fixed barriers
  - Walls in maze game
    - 2D/3D absolute coordinate system

## From Simple to Complex .

- Set of moveable objects
  - One object against set of items
    - Missile vs. several tanks
  - Multiple objects against each other
    - Punching game: arms and legs of players
    - Room of bouncing balls

## Naive General Collision Detection

- For each object i containing polygons p
  - Test for intersection with object j containing polygons q
- For polyhedral objects, test if object i penetrates surface of j
  - Test if vertices of i straddle polygon q of j
    - If straddle, then test intersection of polygon q with polygon p of object i
- Very expensive! O(n2)

## Fundamental Design Principles

- Fast simple tests first, eliminate many potential collisions
  - Test bounding volumes before testing individual triangles
- Exploit locality, eliminate many potential collisions
  - Use cell structures to avoid considering distant objects

## Fundamental Design Principles .

- Use as much information as possible about geometry
  - Spheres have special properties that speed collision testing
- Exploit coherence between successive tests
  - Things don't typically change much between two frames

## Example: Player-Wall Collisions

- 'First person' games must prevent the player from walking through walls and other obstacles
- Most general case
  - Player and walls are polygonal meshes
- Each frame, player moves along path not known in advance
  - Assume piecewise linear
    - Straight steps on each frame
  - Assume player's motion could be fast

## Simple Approach

- On each step, do a general mesh-to-mesh intersection test to find out if the player intersects the wall
- If they do, refuse to allow the player to move
- Problems with this approach? how can we improve:
  - In response?
  - In speed?

## Collision Response

- Frustrating to just stop
  - For player motions, often best thing to do is move player tangentially to obstacle
- Do recursively to ensure all collisions caught
  - Find time and place of collision
  - Adjust velocity of player
  - Repeat with new velocity, start time, start position (reduced time interval)
- Handling multiple contacts at same time
  - Find a direction that is tangential to all contacts

## Typical Approaches

## Collision Detection Approaches

- Two basic techniques:
  - Overlap testing
    - Detects whether a collision has already occurred
  - Intersection testing
    - Predicts whether a collision will occur in the future

## Overlap Testing

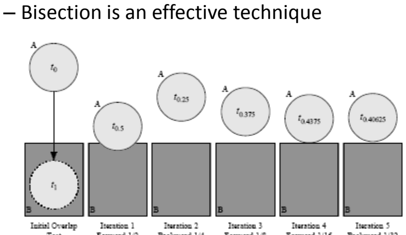
- Facts
  - Most common technique used in games
  - Exhibits more error than intersection testing
- Concept
  - For every simulation step, test every pair of objects to see if they overlap
  - Easy for simple volumes like spheres, harder for polygonal models

## Overlap Testing: Useful Results

- Useful results of detected collision
  - Time collision took place
  - Collision normal vector

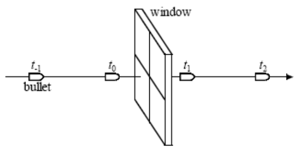
## Overlap Testing: Collision Time

- Collision time calculated by moving object back in time until right before collision
  - Bisection is an effective technique



## Overlap Testing: Limitations

- Fails with objects that move too fast
  - Unlikely to catch time slice during overlap
- Possible solutions
  - Design constraint on speed of objects
  - Reduce simulation step size

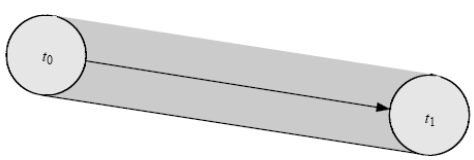



## Intersection Testing

- Predict future collisions
- When predicted:
  - Move simulation to time of collision
  - Resolve collision
  - Simulate remaining time step

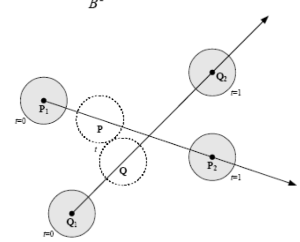
## Intersection Testing: Swept Geometry

- Extrude geometry in direction of movement
- Swept sphere turns into a 'capsule' shape




## Intersection Testing: Sphere-Sphere Collision

$$t = \frac{-(\mathbf{A}\cdot\mathbf{B}) - \sqrt{(\mathbf{A}\cdot\mathbf{B})^2 - B^2\left(A^2 - (r_P + r_Q)^2\right)}}{B^2}, \qquad \begin{aligned}\mathbf{A} &= \mathbf{P}_1 - \mathbf{Q}_1\\ \mathbf{B} &= (\mathbf{P}_2 - \mathbf{P}_1) - (\mathbf{Q}_2 - \mathbf{Q}_1).\end{aligned}$$




## Intersection Testing: Limitations

- Issue with networked games
  - Future predictions rely on exact state of world at present time
  - Due to packet latency, current state not always coherent
- Assumes constant velocity and zero acceleration over simulation step
  - Has implications for physics model and choice of integrator

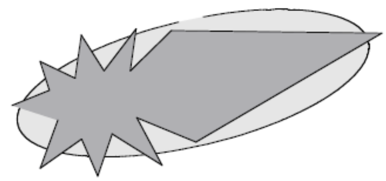
## Complexity Issues

## Dealing with Complexity

- Two common issues when dealing with complexity:
  - Complex geometry must be simplified
    - Not so easy!
  - Reduce number of object pair tests
    - Varies depending on the types of objects

## Simplified Geometry

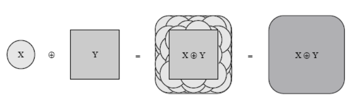
- Approximate complex objects with simpler geometry
  - i.e. Ellipsoid shown below



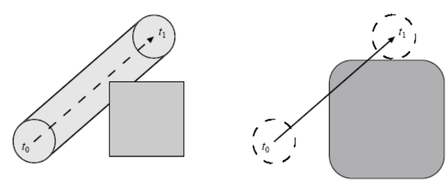
## Minkowski Sum

- By taking the Minkowski Sum of two complex volumes and creating a new volume then overlap can be found
  - By testing if a single point is within the new volume

$$X \oplus Y = \{A + B : A \in X \text{ and } B \in Y\}$$



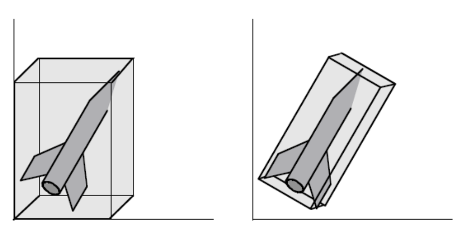

## Minkowski Sum Example




## Bounding Volumes

- Bounding volume is a simple geometric shape
  - Completely encapsulates object
  - If no collision with bounding volume, no more testing is required
- Most common bounding volumes is box
  - More later on…

## Box Bounding Volumes

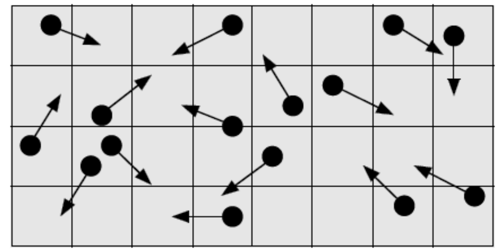


Axis-Aligned Bounding Box    Oriented Bounding Box

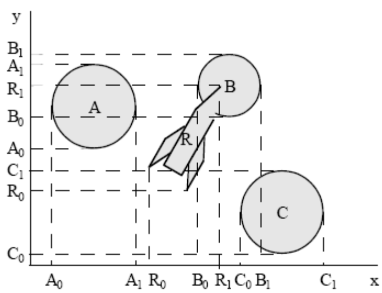
## Achieving O(n) Time Complexity

- Possible solutions for O(n) time complexity
  - Partition space
  - Plane sweep algorithm

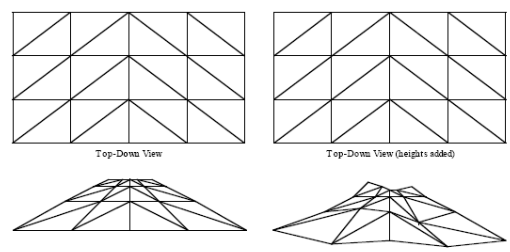
## Partition Space Solution



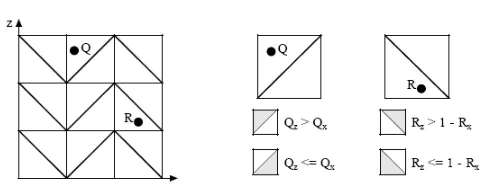
## Plane Sweep Algorithm Solution




## Terrain Collision Detection




## Locate Triangle on Height Field




## Collision Resolution: Examples

- Two billiard balls strike
  - Calculate ball positions at time of impact
  - Impart new velocities on balls
  - Play “clinking” sound effect
- Rocket slams into wall
  - Rocket disappears
  - Explosion spawned and explosion sound effect
  - Wall charred and area damage inflicted on nearby characters
- Character walks through wall
  - Magical sound effect triggered
  - No trajectories or velocities affected

## Collision Resolution Components

- Resolution has three parts:
  - Prologue
  - Collision
  - Epilogue

## Prologue Stage

- Collision known to have occurred
- Check if collision should be ignored
- Other events might be triggered
  - Sound effects
  - Send collision notification messages

## Collision Stage

- Place objects at point of impact
- Assign new velocities using either
  - Physics
  - Some other decision logic

## Epilogue Stage

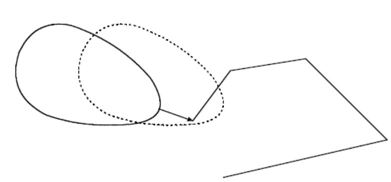
- Propagate post-collision effects
- Possible effects
  - Destroy one or both objects
  - Play sound effect
  - Inflict damage
- Many effects can be done either in the prologue or epilogue

## Resolving Overlap Testing

- Four common stages:
  - Extract collision normal
  - Extract penetration depth
  - Move the two objects apart
  - Compute new velocities

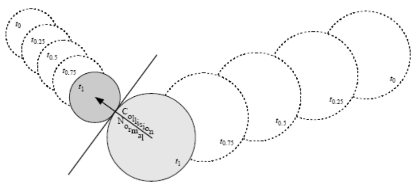
## Extract Collision Normal

- Find position of objects before impact
- Use two closest points to construct the collision normal vector



## Extract Collision Normal .

- Sphere collision normal vector
  - Difference between centers at point of collision



## Resolving Intersection Testing

- Simpler than resolving overlap testing
  - No need to find penetration depth or move objects apart
- Simply just
  - Extract collision normal
  - Compute new velocities

## Acceleration Techniques

## Accelerating Collision Detection

- Two kinds of approaches (many others also)
  - Collision proxies / bounding volumes hierarchies
  - Spatial data structures to localize
- Used for both 2D and 3D
- Accelerates many things, not just collision detection
  - Raytracing
  - Culling geometry before using standard rendering pipeline

## Collision Proxies vs Spatial data Structures

*Collision Proxies:*

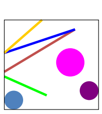
**- Object centric**

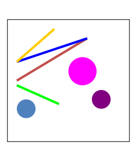
**- Spatial redundancy**

*Spatial data Structures:*

**- Space centric**

**- Object redundancy**





## Collision Proxies vs Spatial data Structures .

*Collision Proxies:*

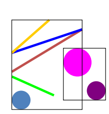
**- Object centric**

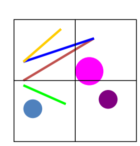
**- Spatial redundancy**

*Spatial data Structures:*

**- Space centric**

**- Object redundancy**

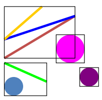


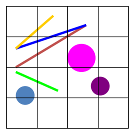


## Collision Proxies vs Spatial data Structures ..

*Collision Proxies:*
- **Object centric**
- **Spatial redundancy**

*Spatial data Structures:*
- **Space centric**
- **Object redundancy**

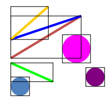


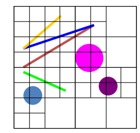


## Collision Proxies vs Spatial data Structures ...

*Collision Proxies:*
- **Object centric**
- **Spatial redundancy**

*Spatial data Structures:*
- **Space centric**
- **Object redundancy**





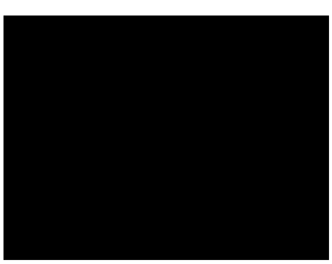
## Collision Proxies

- Proxy
  - Something that takes place of real object
  - Cheaper than general mesh-mesh intersections
- Collision proxy (bounding volume) is piece of geometry used to represent complex object for purposes of finding collision
  - If proxy collides, object is said to collide
  - Collision points mapped back onto original object

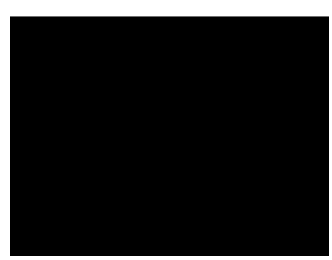
## Collision Proxies .

- Good proxy
  - Cheap to compute collisions for, tight fit to the real geometry
- Common proxies
  - Sphere, cylinder, box, ellipsoid
- Consider
  - Fat player, thin player, rocket, car ...

## Collision Proxies Example 1

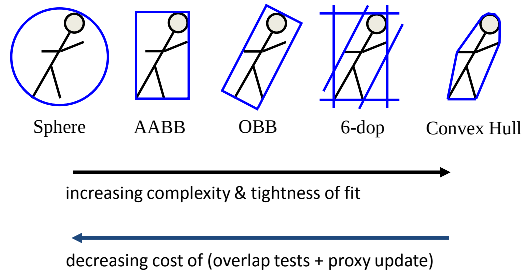


## Collision Proxies Example 2



## Collision Proxies Example 3

## Trade-off in Choosing Proxies




## Trade-off in Choosing Proxies .

- AABB
  - Axis aligned bounding box
- OBB
  - Oriented bounding box, arbitrary alignment
- k-dops
  - Shapes bounded by planes at fixed orientations
    - Discrete orientation

## Pair Reduction

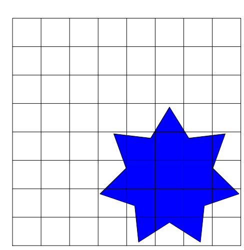
- Want proxy for any moving object requiring collision detection
- Before pair of objects tested in any detail, quickly test if proxies intersect
- When lots of moving objects, even this quick bounding sphere test can take too long:
  - $N^2$ times if there are N objects
- Reducing this $N^2$ problem is called pair reduction
  - Pair testing isn't a big issue until N>50 or so…

## Spatial Data Structures

- Can only hit something that is close
- Spatial data structures tell you what is close to object
  - Uniform grid, octrees, kd-trees, BSP trees
  - Bounding volume hierarchies
    - OBB trees
  - For player-wall problem, typically use same spatial data structure as for rendering
    - BSP trees most common

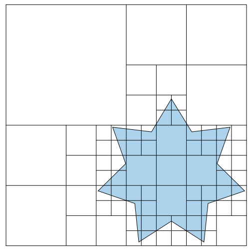
## Uniform Grids

- Axis-aligned
- Divide space uniformly



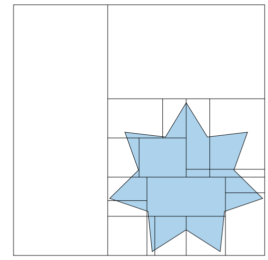
## Quadtrees/Octrees

- Axis-aligned
- Subdivide until no points in cell



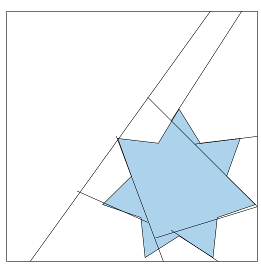
## KD Trees

- Axis-aligned
- Sub-divide in alternating dimensions

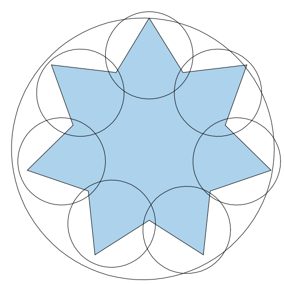


## BSP Trees

- Binary Space Partitioning (BSP)
- Planes at arbitrary orientation

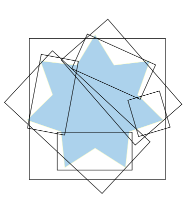


## Bounding Volume Hierarchies



## OBB Trees

- Oriented bounding box (OBB)
- Applicable to a wide range of problems



## BSP Trees Main Idea

- Binary Space Partition (BSP) Tree:
  - Partition space with binary tree of planes
  - Fuchs, Kedem and Naylor `80
- Main idea:
  - Divide space recursively into half-spaces by choosing splitting planes that separate objects in scene

## BSP Trees Methods

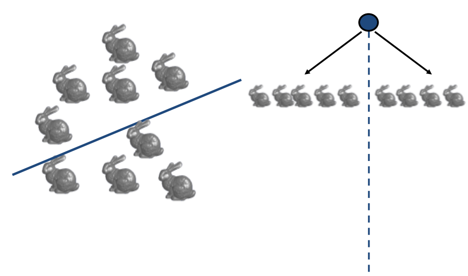
- More general, can deal with inseparable objects
- Automatic, uses as partitions planes defined by the scene polygons
- Method has two steps:
  - Building of the tree independently of viewpoint
  - Traversing the tree from a given viewpoint to get visibility ordering

## BSP Trees Methods .

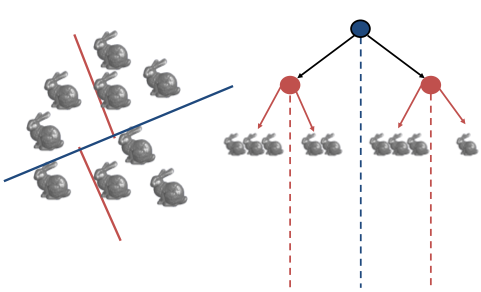
- First step
  - Preprocessing
    - Create binary tree of planes
- Second step
  - Runtime
    - Correctly traversing this tree enumerates objects from back to front

## Creating BSP Trees: Objects

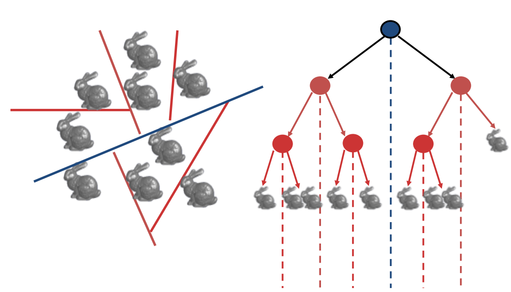
## Creating BSP Trees: Objects .



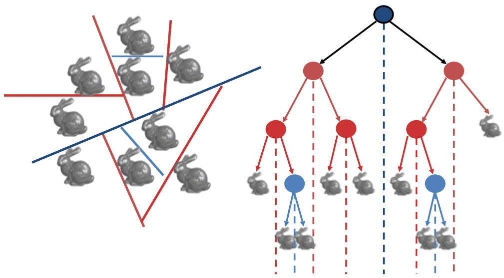
## Creating BSP Trees: Objects ..



## Creating BSP Trees: Objects ...

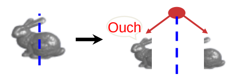


## Creating BSP Trees: Objects ....



## Splitting Objects

- No bunnies were harmed in previous example
- But what if a splitting plane passes through an object?
  - Split the object; give half to each node




## Traversing BSP Trees

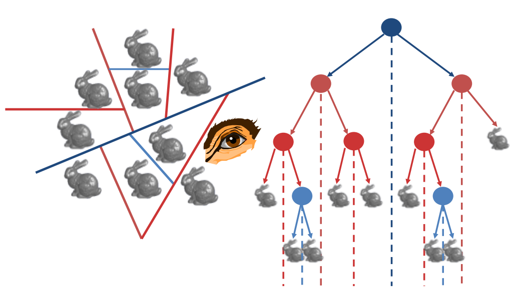
- Tree creation independent of viewpoint
  - Preprocessing step
- Tree traversal uses viewpoint
  - Runtime, happens for many different viewpoints
- Each plane divides world into near and far
  - For given viewpoint, decide which side is near and which is far
    - Check which side of plane viewpoint is on independently for each tree vertex
    - Tree traversal differs depending on viewpoint!
  - Recursive algorithm
    - Recurse on far side
    - Draw object
    - Recurse on near side

## Traversing BSP Trees Pseudo Code

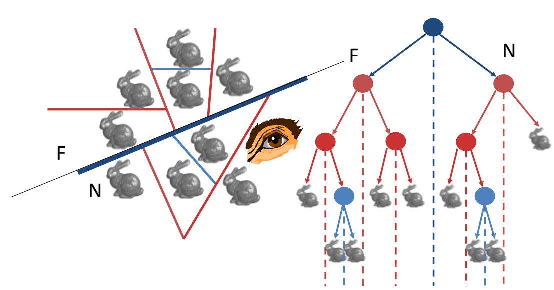
- Query: given a viewpoint, produce an ordered list of (possibly split) objects from back to front

```
renderBSP(BSPtree *T)
  BSPtree *near, *far;
  if (eye on left side of T->plane)
        near = T->left; far = T->right;
  else
        near = T->right; far = T->left;
  renderBSP(far);
  if (T is a leaf node)
        renderObject(T)
  renderBSP(near);
```

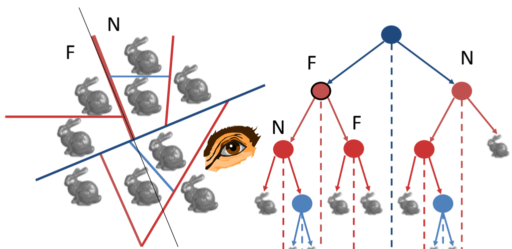
## BSP Trees: Viewpoint A



## BSP Trees: Viewpoint A .

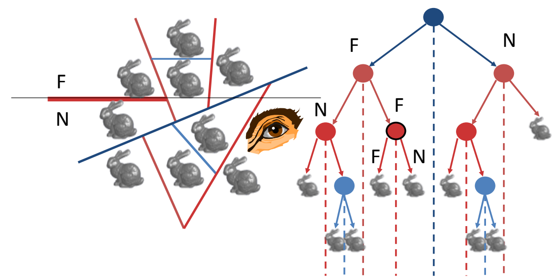


## BSP Trees: Viewpoint A ..

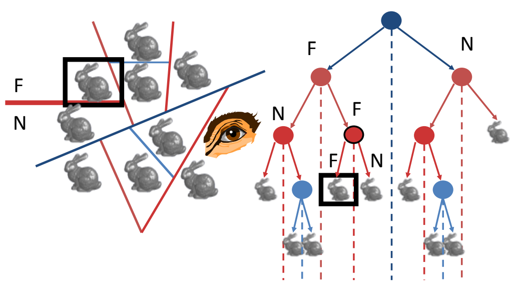



- Decide independently at each tree vertex
- Not just left or right child!

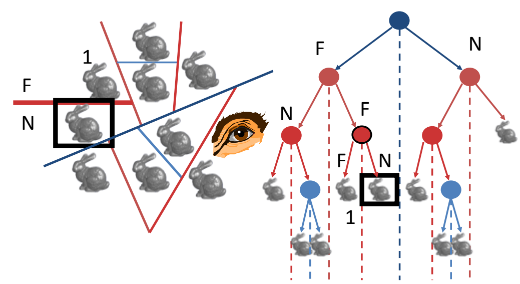
## BSP Trees: Viewpoint A ...



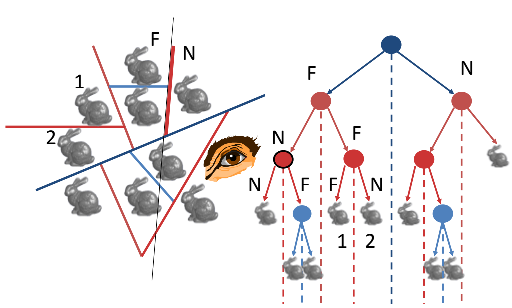

## BSP Trees: Viewpoint A ....



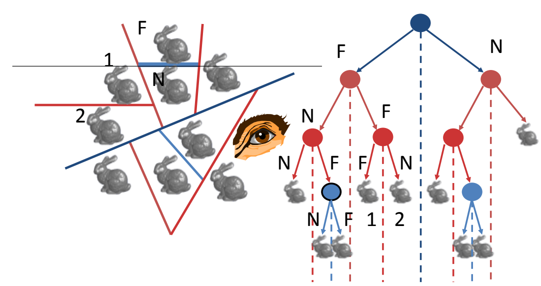

## BSP Trees: Viewpoint A .....



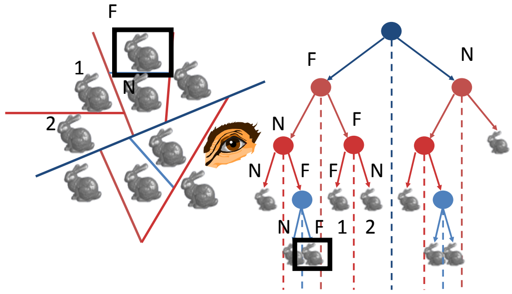

## BSP Trees: Viewpoint A ......



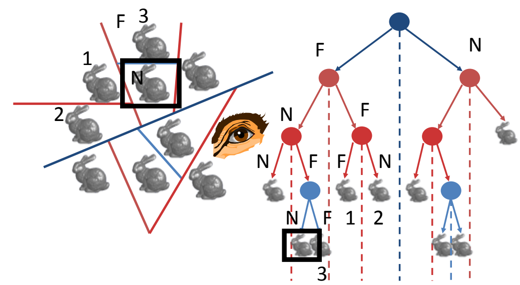

## BSP Trees: Viewpoint A .......



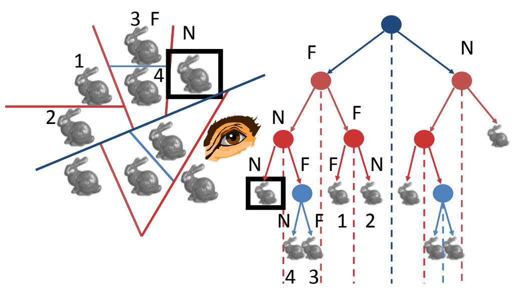
## BSP Trees: Viewpoint A ……..



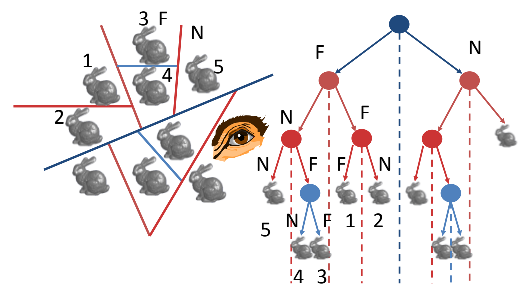
## BSP Trees: Viewpoint A ………



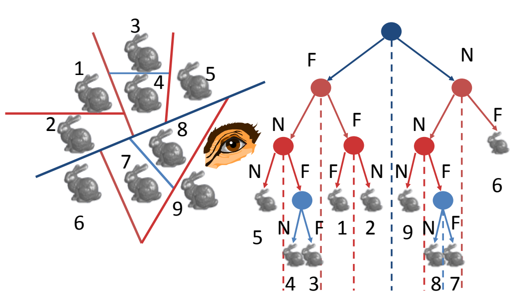
## BSP Trees: Viewpoint A ……….



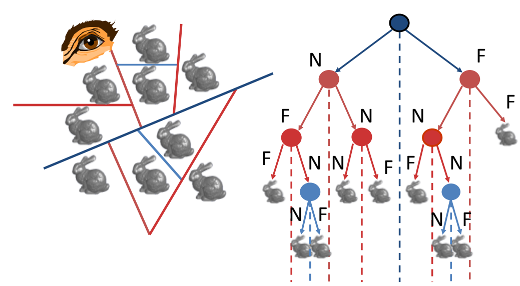
## BSP Trees: Viewpoint A ………..



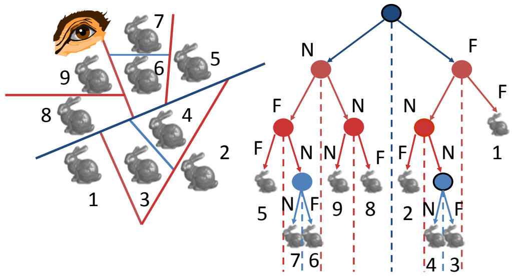
## BSP Trees: Viewpoint A …………



## BSP Trees: Viewpoint B

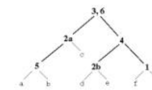


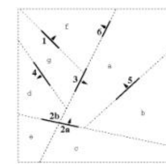
## BSP Trees: Viewpoint B .



## BSP as a Hierarchy of Spaces

- Each node corresponds to a region of space
  - The root is the whole of $R^n$
  - The leaves are homogeneous regions

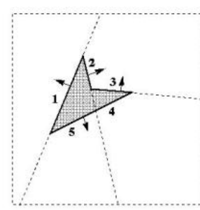


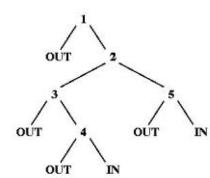


## BSP Tree Traversal: Polygons

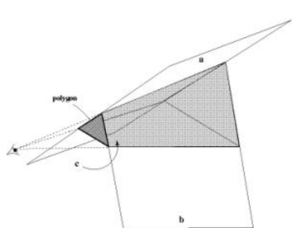
- Split along the plane defined by any polygon from scene
- Classify all polygons into positive or negative half-space of the plane
  - If a polygon intersects plane, split polygon into two and classify them both
- Recurse down the negative half-space
- Recurse down the positive half-space

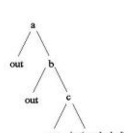
## Representation of Polygons






## Representation of Polyhedra






## BSP Trees for Dynamic Scenes

- When an object moves the planes that represent it must be removed and re-inserted
- Some systems only insert static geometry into the BSP tree
- Otherwise must deal with merging and fixing the BSP cells

## BSP Trees Pos

- Simple, elegant scheme
- Correct version of painter's algorithm back-to-front rendering approach
- Popular for video games

## BSP Trees Cons

- Slow to construct tree
  - O(n log n) to split, sort
- Splitting increases polygon count
  - O(n2) worst-case
- Computationally intense preprocessing stage restricts algorithm to static scenes

## BSP Demo

- http://www.symbolcraft.com/graphics/bsp/

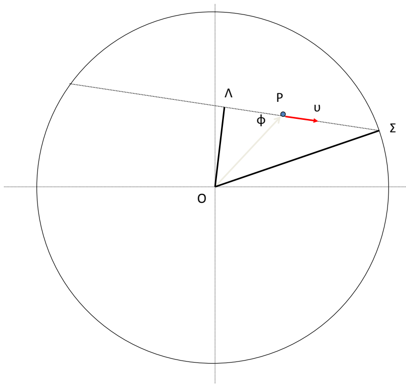
## BSP Videos

- https://www.youtube.com/watch?v=WAd7vzwknF0
- https://www.youtube.com/watch?v=jF2a4imSuvI
- http://www.youtube.com/watch?v=JJjyXRvokE4

## Collision Detection Approach

## Introduction to 3D Breakout

- Most important thing is ball-wall collision detection
- Can be used in:
  - Ball-wall collisions
  - Ball-bat collisions
    - Apart from some cases
  - Ball-Well collisions
    - Apart from some cases (similarly to ball-bat)




## Calculate Collision With Wall

- We are interested in finding the
  - Distance travelled (PΣ)
  - Collision time ($t_{collision}$)
  - Final velocity ($\upsilon_{final}$)

From the previous diagram:

$$P\Sigma = \Lambda\Sigma - \Lambda P \qquad \text{eq. 1}$$

## Pythagoras Theorem

- From Pythagoras:

$$O\Sigma^2 = O\Lambda^2 + \Lambda\Sigma^2 \rightarrow$$
$$\Lambda\Sigma^2 = O\Sigma^2 - O\Lambda^2 \rightarrow$$
$$\Lambda\Sigma = \text{sqrt}(O\Sigma^2 - O\Lambda^2) \qquad \text{eq. 2}$$

## Calculate Distance Travelled

- Also:

$$\Lambda P = OP\cos\phi \qquad \text{eq. 3}$$

- So from eq. 1, eq. 2 and eq. 3:

$$P\Sigma = \text{sqrt}(O\Sigma^2 - O\Lambda^2) - OP\cos\phi \qquad \text{eq. 4}$$

## Calculate Distance Travelled .

- But:

$$\sin\phi = O\Lambda/OP \rightarrow$$
$$O\Lambda = OP\sin\phi$$

- And:

$$O\Lambda^2 = OP^2\sin\phi^2 \qquad \text{eq. 5}$$

## Calculate Distance Travelled ..

- From eq. 4 and eq. 5

$$P\Sigma = \text{sqrt}(O\Sigma^2 - OP^2\sin\phi^2) - OP\cos\phi \qquad \text{eq. 6}$$

- Also from:

$$\sin\phi^2 + \cos\phi^2 = 1 \rightarrow \sin\phi^2 = 1 - \cos\phi^2 \qquad \text{eq. 7}$$

## Calculate Distance Travelled …

- From eq. 6 and eq. 7

$P\Sigma = \text{sqrt}(O\Sigma^2 - OP^2 + OP^2\cos\phi^2) - OP\cos\phi$

- Since $OP \bullet \upsilon = (OP)\upsilon/|\upsilon|\cos\phi$, so the above equation will become:

$P\Sigma = \text{sqrt}(O\Sigma^2 - OP^2 + (OP\upsilon/|\upsilon|\cos\phi)^2) - (OP)\upsilon/|\upsilon|\cos\phi$

## Calculate Distance Travelled ….

- From the dot product the previous equation will become

$P\Sigma = \text{sqrt}(O\Sigma^2 - OP^2 + (OP \bullet \upsilon/|\upsilon|)^2) - OP \bullet \upsilon/|\upsilon|$ eq. 8

- Must take absolute value in case $\phi > 90$

$P\Sigma = |(\text{sqrt}(O\Sigma^2 - OP^2 + (OP \bullet \upsilon/|\upsilon|)^2) - OP \bullet \upsilon/|\upsilon||$ eq. 9

## Calculate Collision Time

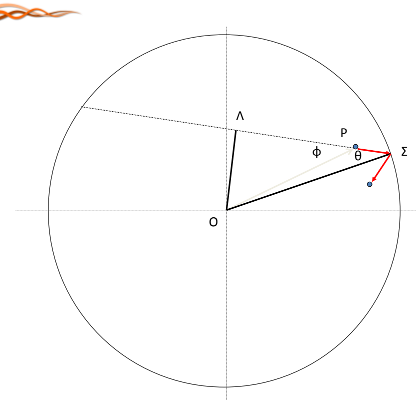
- From motion equation:

$S = \upsilon_{collision} t_{collision}$

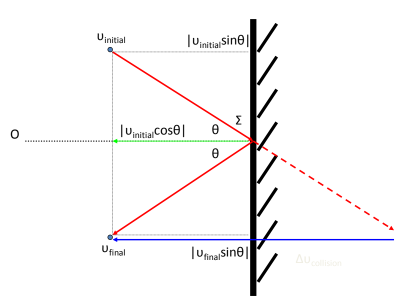
- But $S = P\Sigma$, so:

$P\Sigma = \upsilon_{collision} t_{collision} \rightarrow$

$t_{collision} = P\Sigma/\upsilon_{collision}$ eq. 10




## Calculate Final Velocity




## Calculate Final Velocity .

- The change in ball velocity from the collision:

$|\Delta\upsilon_{collision}| = |\upsilon_{final} - \upsilon_{initial}|$ eq. 11

- From the above figure:

$|\Delta\upsilon_{collision}| = 2|\upsilon_{initial}\cos\theta|$ or eq. 12

$|\Delta\upsilon_{collision}| = 2\upsilon_{initial} \bullet (O\Sigma/|O\Sigma|)$ eq. 13

# Calculate Final Velocity ..

- But Δυ is anti-parallel to ΟΣ and we want to make $Δυ_{collision}$ a vector
- From eq. 12 we do:

$$Δυ_{collision} = -2|υ_{initial}\cos θ|(ΟΣ)/|ΟΣ|\rightarrow$$

- From eq. 13 we do:

$$Δυ_{collision} = -2(ΟΣ)\ (υ_{initial}\bullet ΟΣ)/|ΟΣ|^2$$

# Assignment Tips

# Some Tips

- Important 3D objects for collision detection in 3D Breakout Assignment
  - Invisible ground (optional)
  - Ball
  - Bat
  - Well

# Class TBall.h

```
class TBall
{
    public:
        double _radius;          // defines the radius of the ball
        TVector _position;       // defines the position of the ball
        TVector _velocity;       // defines the velocity v0 of the ball

        // Constructors
        TBall();
        TBall(const double& Radius, const TVector& Position, const
                TVector& Velocity)
        {_radius=Radius; _position=Position; _velocity=Velocity;};
```

# Class TBall.h .

```
        // Selectors
        double GetBallRadius()    const {return _radius;};
        TVector GetBallPosition() const {return _position;};
        TVector GetBallVelocity() const {return _velocity;};

        void DrawBall();          // Draws the ball

        void CalculateVelocity(const TVector& velocity, const double&
                seconds);         // Assigns the ball a velocity

        TVector CalcDistanceTravelled(const double& seconds) const;
                // Calculates the distance traveled

        void MoveBall(const double& seconds); // Moves the ball
};
```

# Default Constructor for the Ball

```
TBall::TBall()
{
        // Assign default values for the attributes
        // of the ball
        _radius = 4.0;
        _position = TVector(0.0, 0.0, 0.0);
        _velocity = TVector(1.0, 0.0, 0.0);
}
```

## Function to Draw the Ball

```
void TBall::DrawBall()
{
	glPushMatrix();
		glTranslatef(_position.X(),
		_position.Y(), _position.Z());
		glutSolidSphere(_radius, 20, 20);
	glPopMatrix();
}
```

## More Functions

- Functions for the TBall Class:
  - CalculateVelocity
  - CalcDistanceTravelled
  - MoveBall
- Function for TDisplayImp
  - Idle
- TBat Class

## CalculateVelocity Function

```
void TBall::CalculateVelocity(const TVector&
  velocity, const double& seconds)
{
	_velocity = velocity;
}
```

## CalcDistanceTravelled Function

```
TVector TBall::CalcDistanceTravelled(const double&
  seconds) const
{
  TVector new_velocity, new_position;

  new_velocity = _velocity;
  new_position = _position +
  new_velocity*seconds;

  return new_position;
}
```

## MoveBall Function

```
void TBall::MoveBall(const double& seconds)
{
  _position = CalcDistanceTravelled(seconds);
}
```

## Idle Function

```
void TDisplayImp::idle(void)
{
  // Set the time for the simulation
  _scene->CalculateSimulations();

  glutPostRedisplay();
}
```

## Class TBat

```
class TBat
{
   public:
      TVector _points[16];            // points for the first bat
      TVector _normal[15];      // normal of the ground

   public:
      // Default constructor
      TBat(){};
      TBat(double rotation_angle);
```

## Class TBat .

```
void DrawBat();          // Draws the bats
void MoveBatRight();     // Moves bat on the right
void MoveBatLeft();      // Moves the bat on the right
int BatCollisions(const TBall &ball, const double& seconds);
int BatCollisionsSides(const TBall &ball, const double& seconds);
int BatCollisionsEdges(const TBall &ball, const double& seconds);
```

## Class TBat ..

```
TVector Bat_Faces_Reflection(TBall &ball, const double& seconds, const double& distance);

TVector Bat_Left_Side_Reflections(TBall &ball, const double& seconds, const double& parameter);

TVector Bat_Right_Side_Reflections(TBall &ball, const double& seconds, const double& parameter);
```

## Class TBat ...

```
TVector Bat_Edge12_Reflections(TBall &ball, const double& seconds);

TVector Bat_Edge15_Reflections(TBall &ball, const double& seconds);

TVector Bat_Edge13_Reflections(TBall &ball, const double& seconds);

TVector Bat_Edge11_Reflections(TBall &ball, const double& seconds);
};
```

## TBat Constructor

```
TBat::TBat(double rotation_angle)
{
   TVector initial_vector, upper_vector, construction_vector;

   // Define a vector for the construction of the ground points of the bats
   initial_vector = TVector(1.0, 0.0, 0.0);

   // Define a vector for the construction of the upper points of the bats
   upper_vector = TVector(0.0, 10.0, 0.0);
```

## TBat Constructor .

```
// Define the rotation axis
TVector rotation_axis(0.0,1.0,0.0);

// Define the three rotation matrices for the bats
TMatrix33 bat_construction = TMatrix33(rotation_axis, rotation_angle);

// Define the vector used for the construction of the bats
construction_vector = bat_construction*initial_vector;

// Define the rotation matrix for the constuction of the bats
TMatrix33 bat_rotation = TMatrix33(rotation_axis, angle);
```

## TBat Constructor ..

```
// Construct the 16 points of the bats
_points[0] = construction_vector*bat_radius1;
_points[1] = bat_rotation*_points[0];
_points[2] = bat_rotation*_points[1];
_points[3] = bat_rotation*_points[2];
_points[7] = construction_vector*bat_radius2;
_points[6] = bat_rotation*_points[7];
_points[5] = bat_rotation*_points[6];
_points[4] = bat_rotation*_points[5];
_points[8]  = _points[0] + upper_vector;
_points[9]  = _points[1] + upper_vector;
_points[10] = _points[2] + upper_vector;
_points[11] = _points[3] + upper_vector;
_points[15] = _points[7] + upper_vector;
_points[14] = _points[6] + upper_vector;
_points[13] = _points[5] + upper_vector;
_points[12] = _points[4] + upper_vector;
}
```

## Drawing Front Side of Bats

```
glBegin(GL_QUAD_STRIP);

// Front face, normal of first surface
_normal[0] = ((_points[8] - _points[0])*(_points[1] - _points[0])).unit();
glNormal3f(_normal[0].X(), _normal[0].Y(), _normal[0].Z());

// Construct first quad
glVertex3f(_points[0].X(), _points[0].Y(), _points[0].Z());
glVertex3f(_points[8].X(), _points[8].Y(), _points[8].Z());

// Front face, second surface
_normal[1] = ((_points[9] - _points[1])*(_points[2] - _points[1])).unit();
glNormal3f(_normal[1].X(), _normal[1].Y(), _normal[1].Z());
```

## Drawing Front Side of Bats .

```
// Construct second quad
glVertex3f(_points[1].X(), _points[1].Y(), _points[1].Z());
glVertex3f(_points[9].X(), _points[9].Y(), _points[9].Z());

// Front face, third surface
_normal[2] = ((_points[10] - _points[2])*(_points[3] - _points[2])).unit();

glNormal3f(_normal[2].X(), _normal[2].Y(), _normal[2].Z());

// Construct third quad
glVertex3f(_points[2].X(), _points[2].Y(), _points[2].Z());
glVertex3f(_points[10].X(), _points[10].Y(), _points[10].Z());
glNormal3f(_normal[2].X(), _normal[2].Y(), _normal[2].Z());

// Construct fourth quad
glVertex3f(_points[3].X(), _points[3].Y(), _points[3].Z());
glVertex3f(_points[11].X(), _points[11].Y(), _points[11].Z());

glEnd();
```

## Drawing the Rest of the Bats

- In the same way you will have to draw the:
  - Left side of the bat
  - Back side of the bat
  - Right side of the bat
  - Up side of the bat

## Bat Collisions

- At least three checks:
  - Check for collisions between the ball and the three bats like ball-wall
  - Check for collisions between the ball and the side of the bats
  - Check for collisions between the ball and the edges of the bats
- Repeat the same procedure for reflections of the ball after collisions

## Calculate the reflection of the ball after collision

```
double TBounds::Ball_Reflection(TBall &ball, const
  double& seconds)
{
  TVector ball_velocity_after_collision,
  previous_ball_position, collision_vector, final_velocity;

  // Perform calculations for the previous time step
  previous_ball_position = ball.GetBallPosition() -
  ball.GetBallVelocity()*seconds;

  double absBallVelocity =
  sqrt(ball.GetBallVelocity().dot(ball.GetBallVelocity()));
```

## Calculate the reflection of the ball after collision .

```
// Calculate the Ri*V to calculate the collision time
double RV = previous_ball_position.dot(ball.GetBallVelocity())/absBallVelocity;

// Absolute RV
double abs_RV = abs(RV);

// Define the initial distance = 100 - 4 = 96
double initial_distance = 100.0 - ball.GetBallRadius();
```

## Calculate the reflection of the ball after collision ..

```
// Calculate the determinant
double Determinant = ((RV*RV) - previous_ball_position.dot(previous_ball_position) + initial_distance*initial_distance);

// Calculate the collision time
double collision_time = abs(-abs_RV + sqrt(Determinant))/absBallVelocity;

// Calculate the collision vector (normal vector) from: R = r + v*t
collision_vector = previous_ball_position + ball.GetBallVelocity()*collision_time;

// Make the collision vector (normal vector) unit vector
TVector unit_collision_vector = TVector::unit(collision_vector);
```

## Calculate the reflection of the ball after collision ...

```
// Define velocity by: Vreflected = (Vinitial*Normal.unit)*Normal.unit
ball_velocity_after_collision = unit_collision_vector*(ball.GetBallVelocity().dot(unit_collision_vector));

// Calculate the velocity of the ball after collision with the invisible wall
final_velocity = ball.GetBallVelocity() - ball_velocity_after_collision*2.0;

ball.CalculateVelocity(final_velocity, collision_time);

return collision_time;
}
```

## References


- http://www.cs.wisc.edu/~schenney/courses/cs679-f2003/lectures/cs679-22.ppt
- http://graphics.ucsd.edu/courses/cse169_w05/CSE169_17.ppt


## Links

- http://en.wikipedia.org/wiki/Bounding_volume
- http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=30
- http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html
- http://www.devmaster.net/articles/bsp-trees/
- http://maven.smith.edu/~mcharley/bsp/createbsptree.html
- http://www.cs.unc.edu/~geom/
- http://www.cs.ox.ac.uk/stephen.cameron/distances/

## Questions