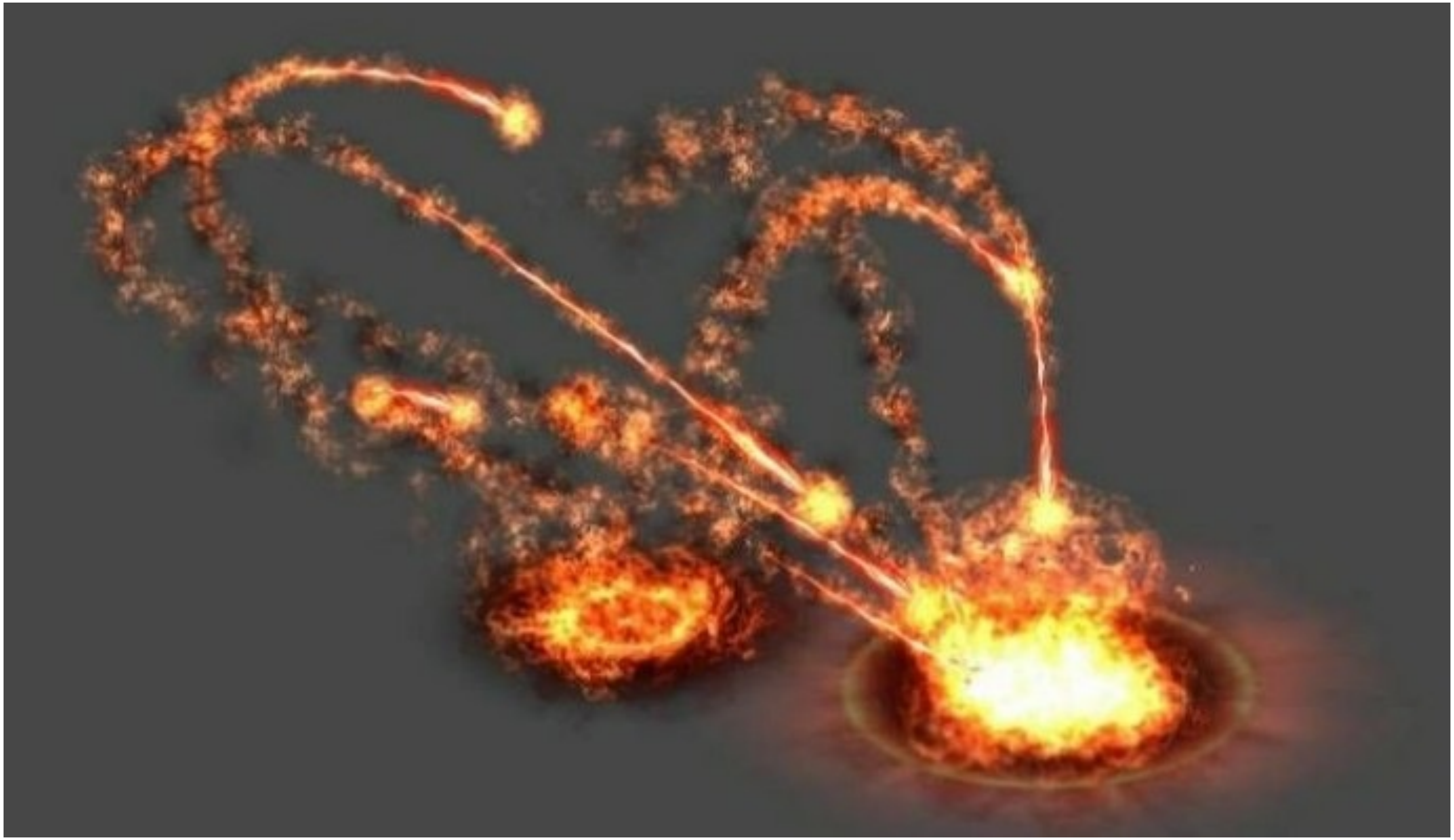
CS354R

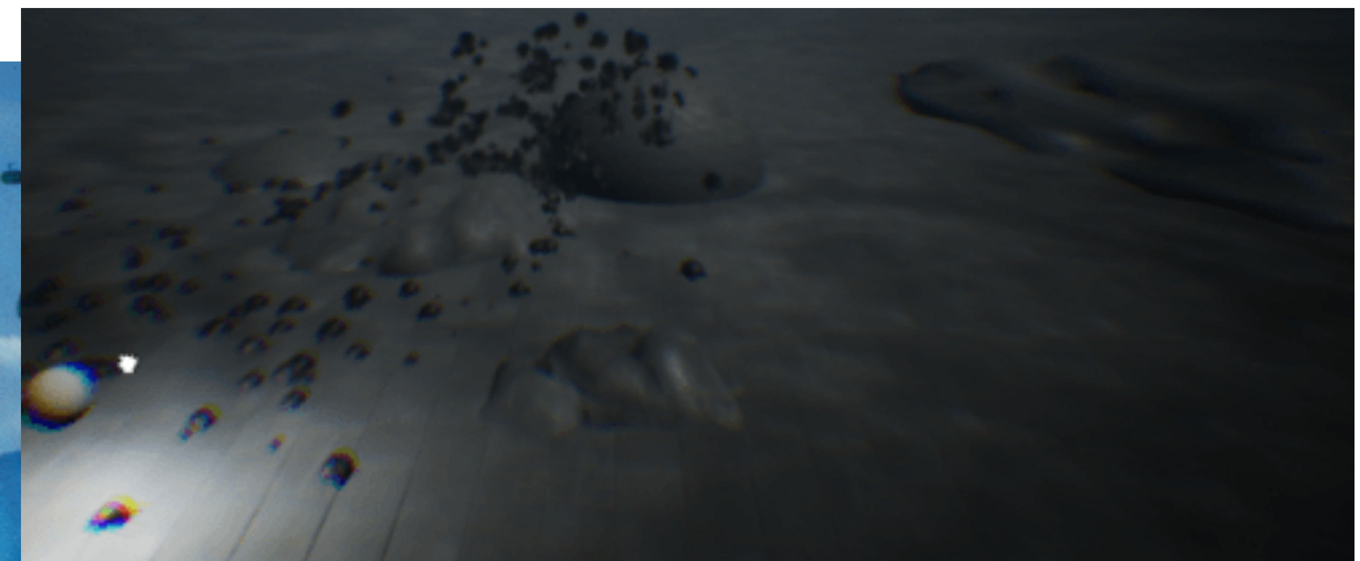
DR SARAH ABRAHAM

PARTICLES AND FLOCKING BEHAVIOR

PARTICLE EFFECTS



PARTICLE EFFECTS IN ACTION



Created by Ashif Ali in Niagara (https://cghow.com/members/asif786ali/)

GENERAL PARTICLE SYSTEMS

- Objects are considered point masses with orientation
- Simple rules control how the particles move
- Particles can be controlled/rendered to simulate different things:
  - Fireworks
  - Waterfalls, spray, foam
  - Explosions (smoke, flame, chunks of debris)
  - Clouds
  - Crowds, herds
- Widely used in movies as well as games

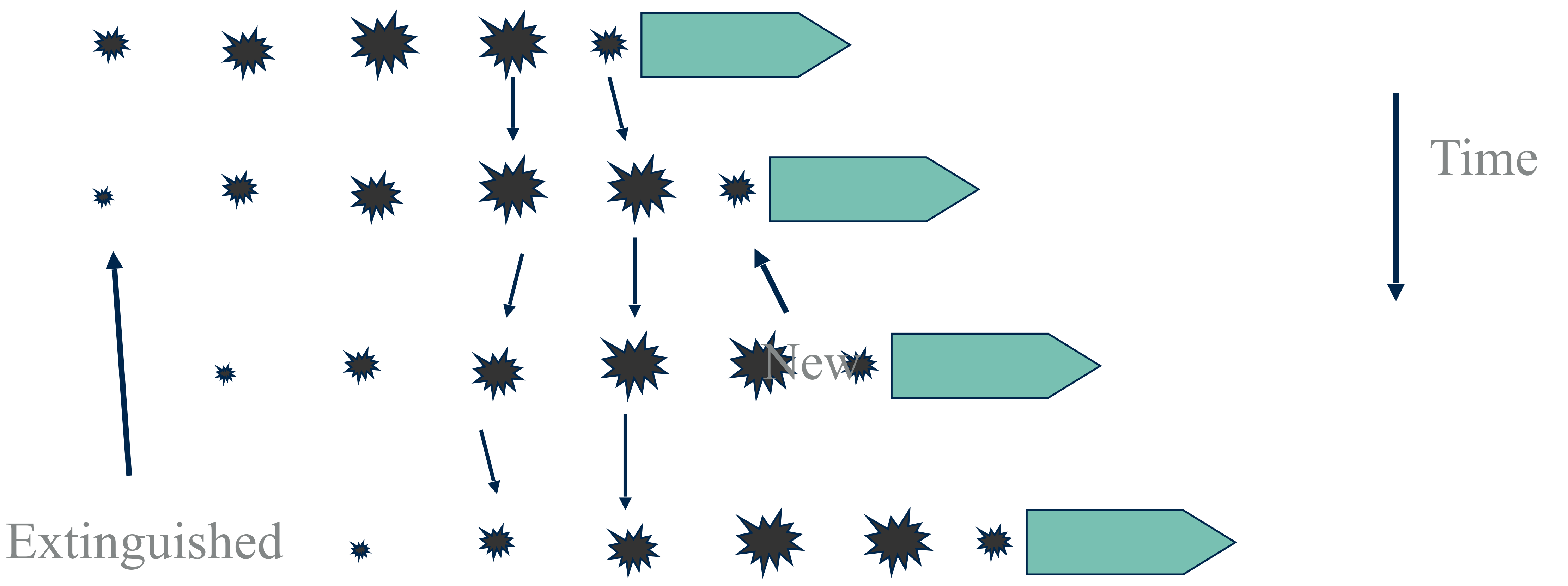
PARTICLE SYSTEM STEP BY STEP

1. Inject new particles into the system and assign individual attributes
   - There may be one or more sources
   - Particles might be generated at random (clouds), in a constant stream (waterfall), or via a script (fireworks)
2. Remove any particles that have exceeded their lifetime
   - May have a fixed lifetime, or die on some condition
3. Move all the current particles according to their script
   - Script typically involves neighboring particles and environment
4. Render all the current particles
   - Many options for rendering (shaders, textures etc)

EXAMPLE: ROCKET SMOKE TRAILS

- Particles are spawned at a constant rate
- They have zero initial velocity, or maybe a small velocity going away from the rocket
- Rules:
  - Particles can rise or fall (drift with the wind)
  - Attach a density that grows quickly then falls over time
- Extinguish when density becomes small
- Render with billboard facing the viewer, scaled according to the density of the puff

SMOKE TRAILS



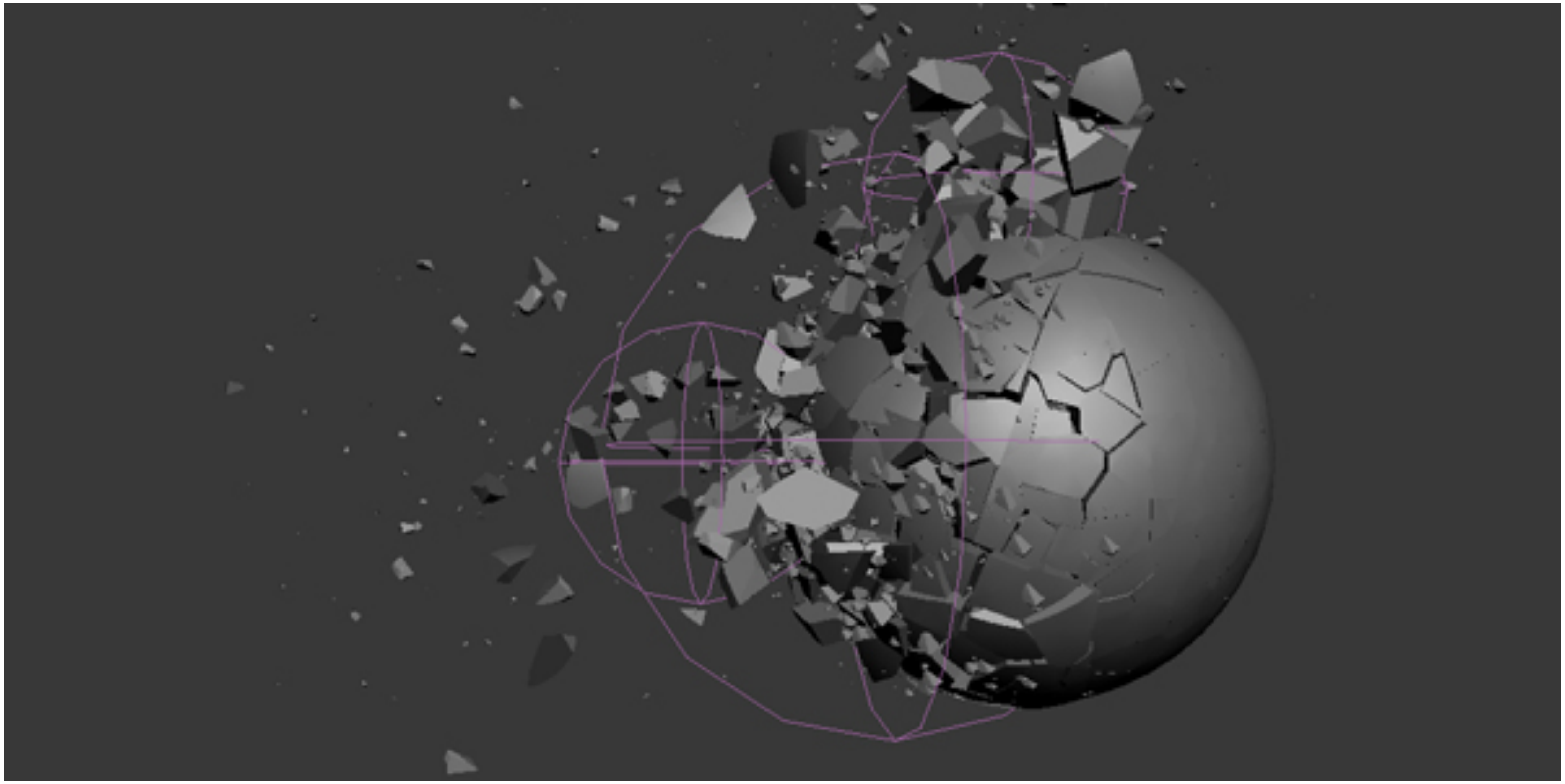
PARTICLE EMITTERS

- Provide sources for spawning particles
- Emitters can specify:
  - Rate of particle emission
  - Shape of particle emission
  - Direction of particle emission
- Can also specify parameterization of individual particle properties

EXAMPLE: OBJECT FRACTURING

- System starts when the target is hit
- Target is broken into pieces and a particle is assigned to each piece
- Each particle gets an initial velocity away from the center of the explosion
- Particle rules:
  - Move ballistically unless there is a collision
  - Computer rigid body rotation or generate random rotation
  - Resolve collisions by reflecting the velocity about the contact normal
- Rendering draws the appropriate piece of target at the particle's location

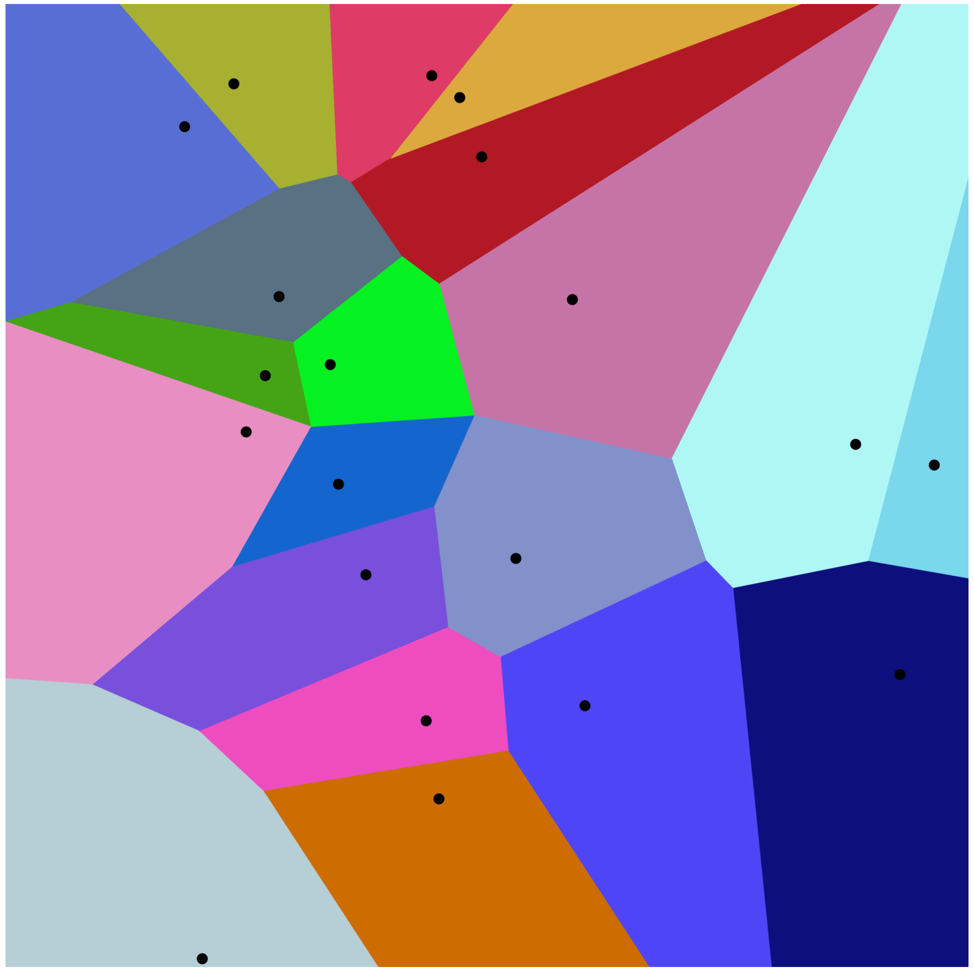
OBJECT FRACTURING



Laurent Renaud (http://cgcookie.com/max/2009/08/18/creating-an-exploding-planet/)

HOW TO FRACTURE?

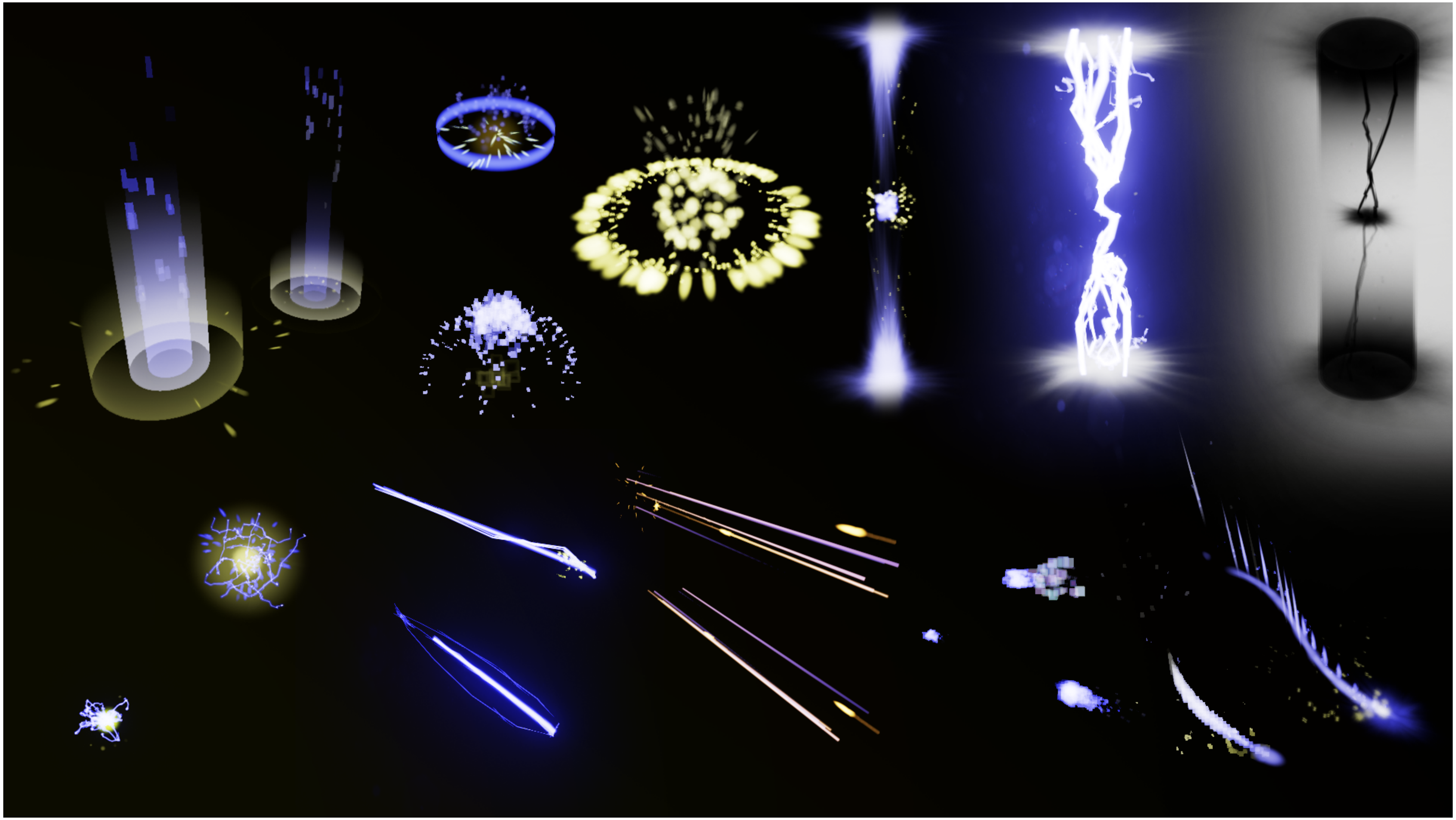
- Voronoi commonly used
  - Can be done in realtime or using pre-processing
- Voronoi partitions created by seeding the surface of a plane (or a 3D space) with points
  - Every point along the surface (or in the 3D space) associated with the closest seeded point
- Voronoi useful in procedural techniques more generally as well!



VISUALIZING PARTICLES

- Particles can be visualized in a number of ways
  - Billboarding (applying a texture to an individual particle)
  - Point shaders (applying a shader to an individual particle)
  - Mesh shaders (applying a shader to a mesh based on the particle positions)
  - Post processing (applying a post processing effect in screen space to represent particles)

VISUALIZING PARTICLES



DC Assets (https://www.unrealengine.com/marketplace/en-US/product/scifi-particle-pack)

PARTICLES IN GAMES

- They're everywhere!
  - https://www.youtube.com/watch?v=6_NsaYtooQA

PARTICLE MANAGEMENT

- Particle systems should include some sort of pool for resource management
  - Particle lifespans relatively short
  - Particles should be reused as much as possible
  - Good caching helps with particle system efficiency
- Particle systems well-suited to parallelization
  - Can be implemented in conjunction with multi-threading/multi-core/GPUs

FLOCKING BEHAVIOR

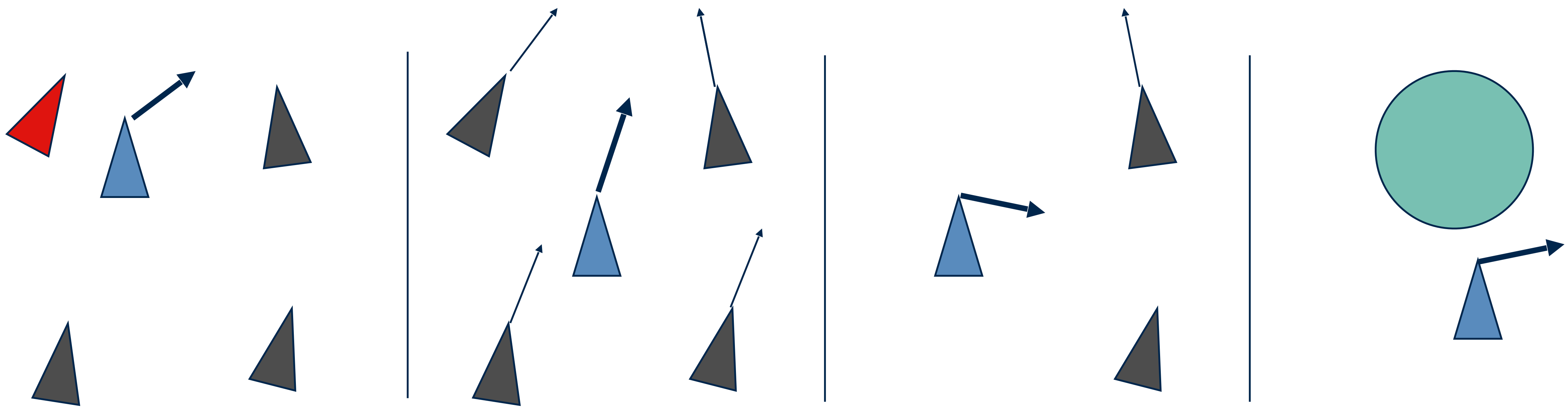
- Particles can also model flocks, swarms, crowds etc

(https://portraitsofwildflowers.wordpress.com/2011/12/10/grackles-revisited/)

FLOCKING MODELS (REYNOLDS '87)

- Potential fields are most often used in avoiding collisions between the members of a group
  - Member pushes on its neighbors to keep from colliding
- Additional rules for groups can be defined (result is flocking, herding, schooling, etc)
- Each rule contributes a desired direction, which are combined in some way to come up with the acceleration
- The aim is to obtain emergent behavior:
  - Define simple rules on individuals that interact to give interesting global behavior
  - e.g individual birds form a flock, but we never explicitly specify a leader, or shape, or speed

FLOCKING RULES ILLUSTRATED



Separation: fly away away from neighbors that are "too close"

Alignment: steer toward average velocity

Cohesion: steer toward average position

Avoidance: steer away from obstacles

FLOCKING RULES EXPLANATION

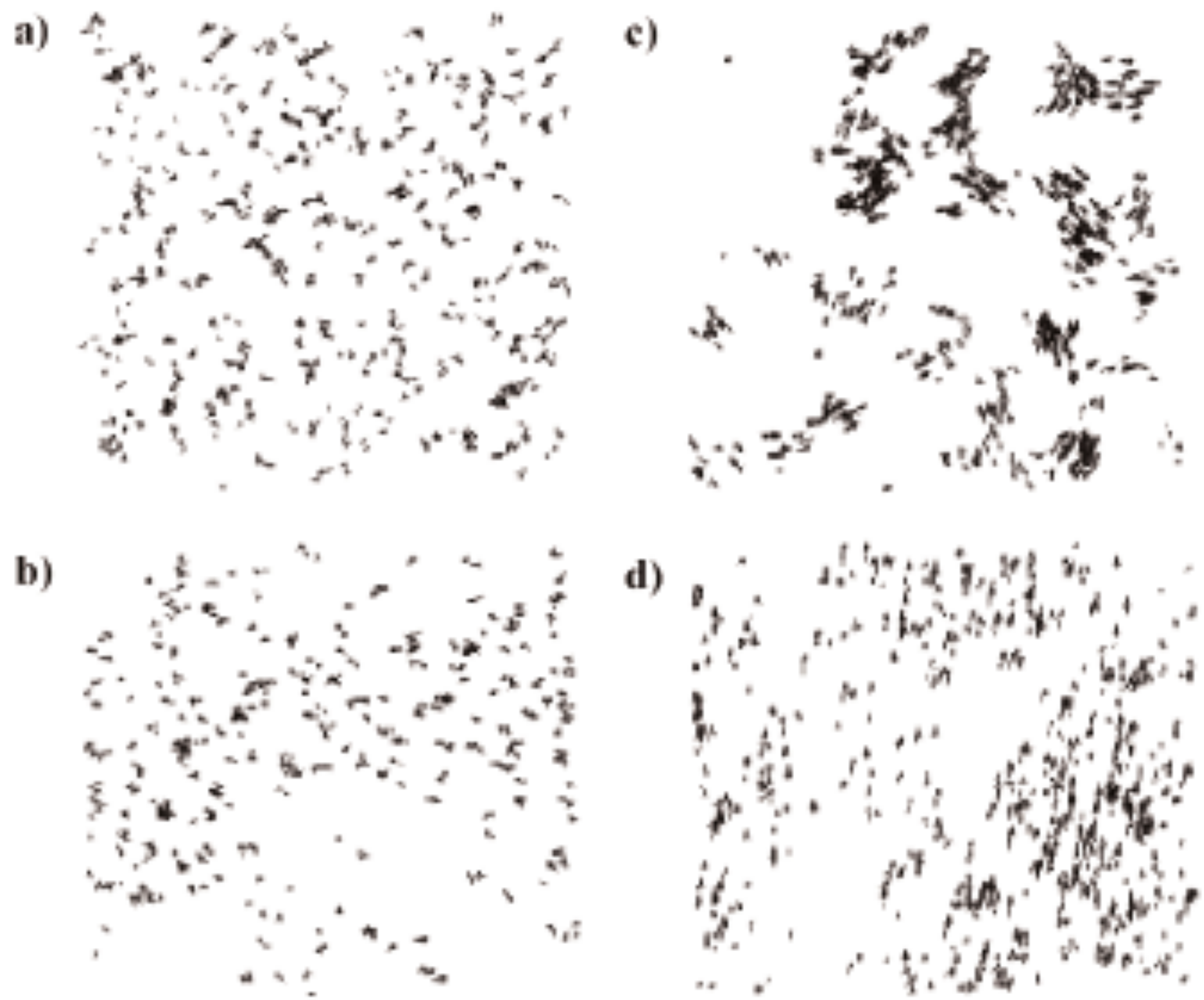
- Separation: Try to avoid running into local flock-mates
  - Works just like potential fields
  - Normally, use a perception volume to limit visible flock-mates
- Alignment: Try to fly in same direction as local flock-mates
  - Gets everyone flying in the same direction
- Cohesion: Try to move toward the average position of local flock-mates
  - Spaces everyone out evenly, and keep boundary members toward the group
- Avoidance: Try to avoid obstacles
  - Just like potential fields!

BALANCING FLOCKING

- Consider commands as accelerations
- Give a weight to each desire
  - e.g. high for avoidance, low for cohesion
- Option 1: Apply in order of highest weight, until a max acceleration is reached
  - Ensures that high priority things happen
- Option 2: Take weighted sum and truncate acceleration
  - Makes sure some part of everything happens

FLOCKING DEMO

- https://www.youtube.com/watch?v=rN8DzlgMt3M



Craig Reynolds (Boids '87)

FLOCKING EVALUATION

- Advantages:
  - Complex behavior from simple rules
  - Many types of behavior can be expressed with different rules and parameters
- Disadvantages:
  - Can be difficult to set parameters to achieve desired result
  - All the problems of potential fields regarding strength of forces

BEYOND BOIDS

- Flocking behaviors vary based on the agents being simulated
- Adjusting the rules (or evaluation of rules) allows for greater variety in simulation

Cyberpunk 2077

ANOTHER EXAMPLE…

Mythic Ocean (https://www.youtube.com/watch?v=dHriVqfqDMI)

MAKING IT FAST

- Comparing a large number of agents/particles gets expensive
- How can we reduce the cost of these interactions?

SPATIAL DATA STRUCTURES

- Data indexed by spatial location (e.g. location or polygons)
- Multitude of uses in video games!
  - Visibility - What can I see?
  - Ray intersections - What did the player just shoot?
  - Collision detection - Did the player just hit a wall?
  - Proximity queries - Where is the nearest power-up?

USING DECOMPOSITIONS

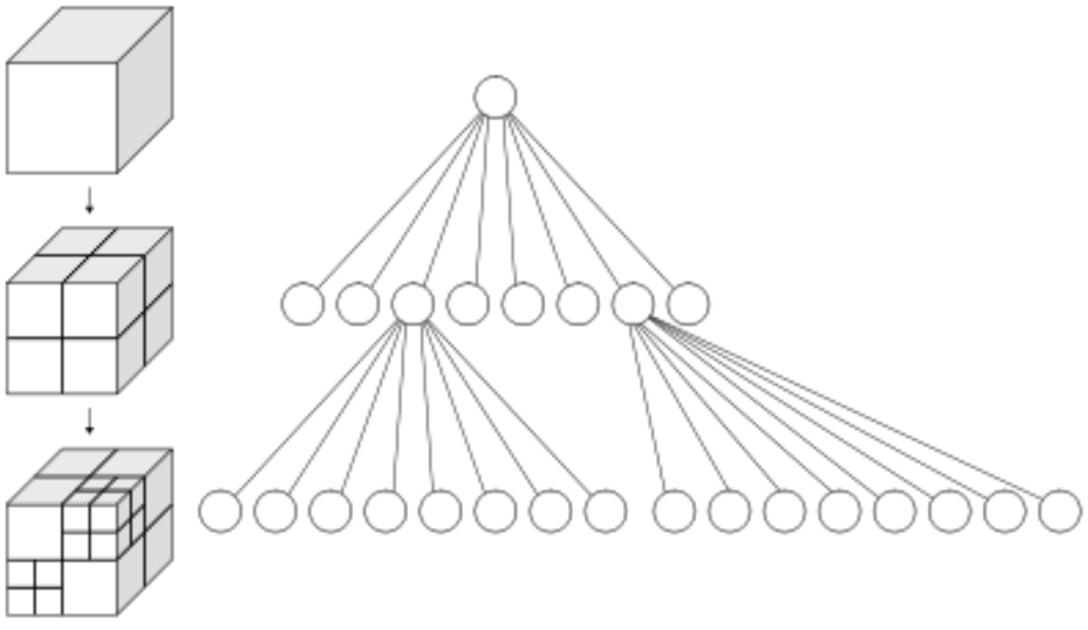
- Geometric queries are expensive
- Reduce the cost with fast, approximate queries that eliminate distant (or hidden) objects
- Trees with a containment property allow us to do this
  - The cell of a parent completely contains all the cells of its children
  - If a query fails for the cell, we know it will fail for all its children
  - If the query succeeds, we try for the children
  - If we get to a leaf, we do the expensive query

SPATIAL DECOMPOSITIONS

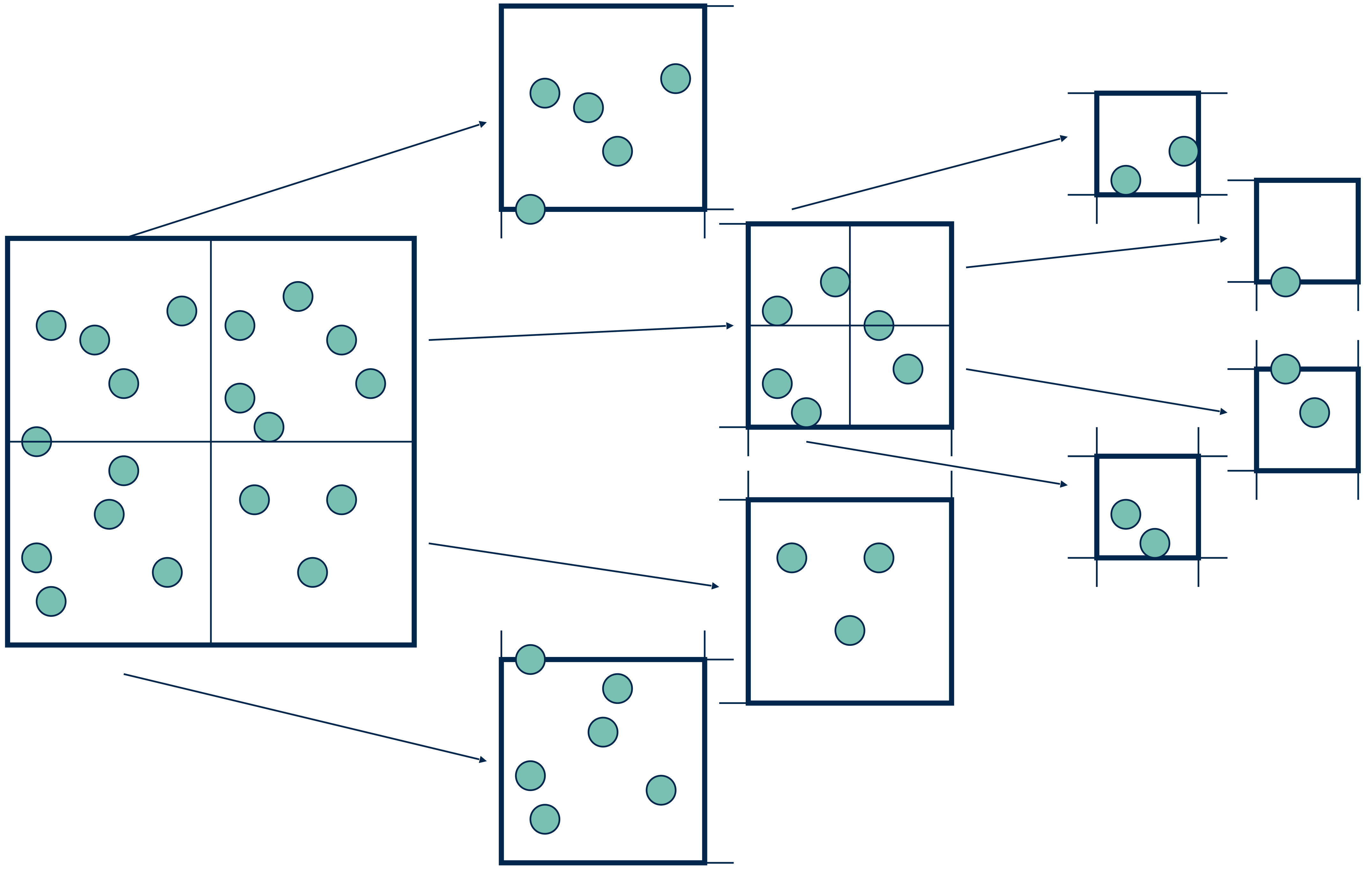
- Partition space into regions, or cells, of some type
- Octrees (Quadtrees): Axis aligned, regularly spaced planes cut space into cubes (squares)
- Kd-trees: Axis aligned planes cut space into rectilinear regions
- BSP trees: Arbitrarily aligned planes cut space into convex regions
- BVHs: Geometry hierarchically arranged within the tree

OCTREE

- Root node represents a cube containing entire world
- Each node has eight children nodes
    - Quadtree is for 2D decompositions - root is square and four children are sub-squares
- Objects assigned to nodes in one of two common ways:
    - All objects are in leaf nodes
    - Each object is in the leaf that partially contains it

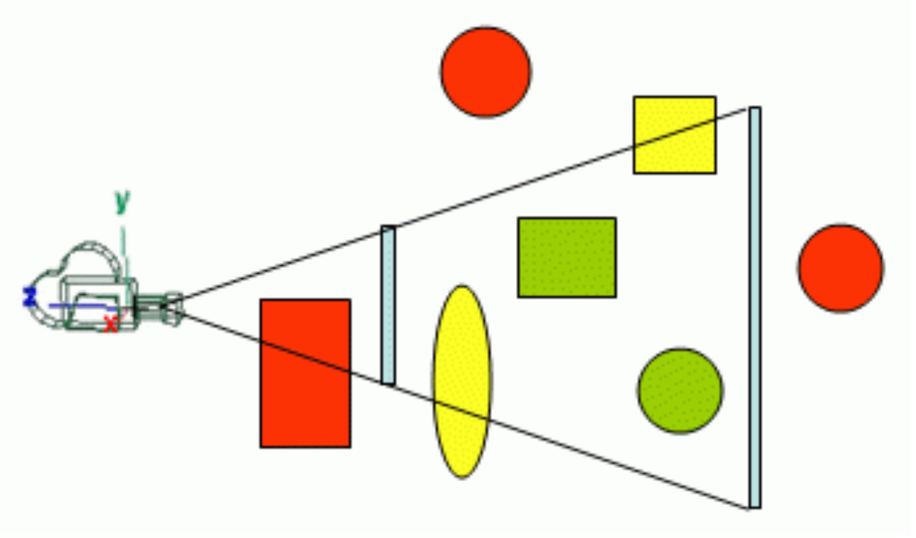


QUADTREE EXAMPLE CONSTRUCTION



FRUSTUM CULLING WITH OCTREES

- Eliminate objects that do not intersect the view frustum
- Have a test that succeeds if a cell may be visible
  - Test corners of cell against each clip plane
- Starting with the root node cell, perform the test
  - If it fails, nothing inside the cell is visible
  - If it succeeds, something inside the cell might be visible
  - Recurse for each child of a visible cell



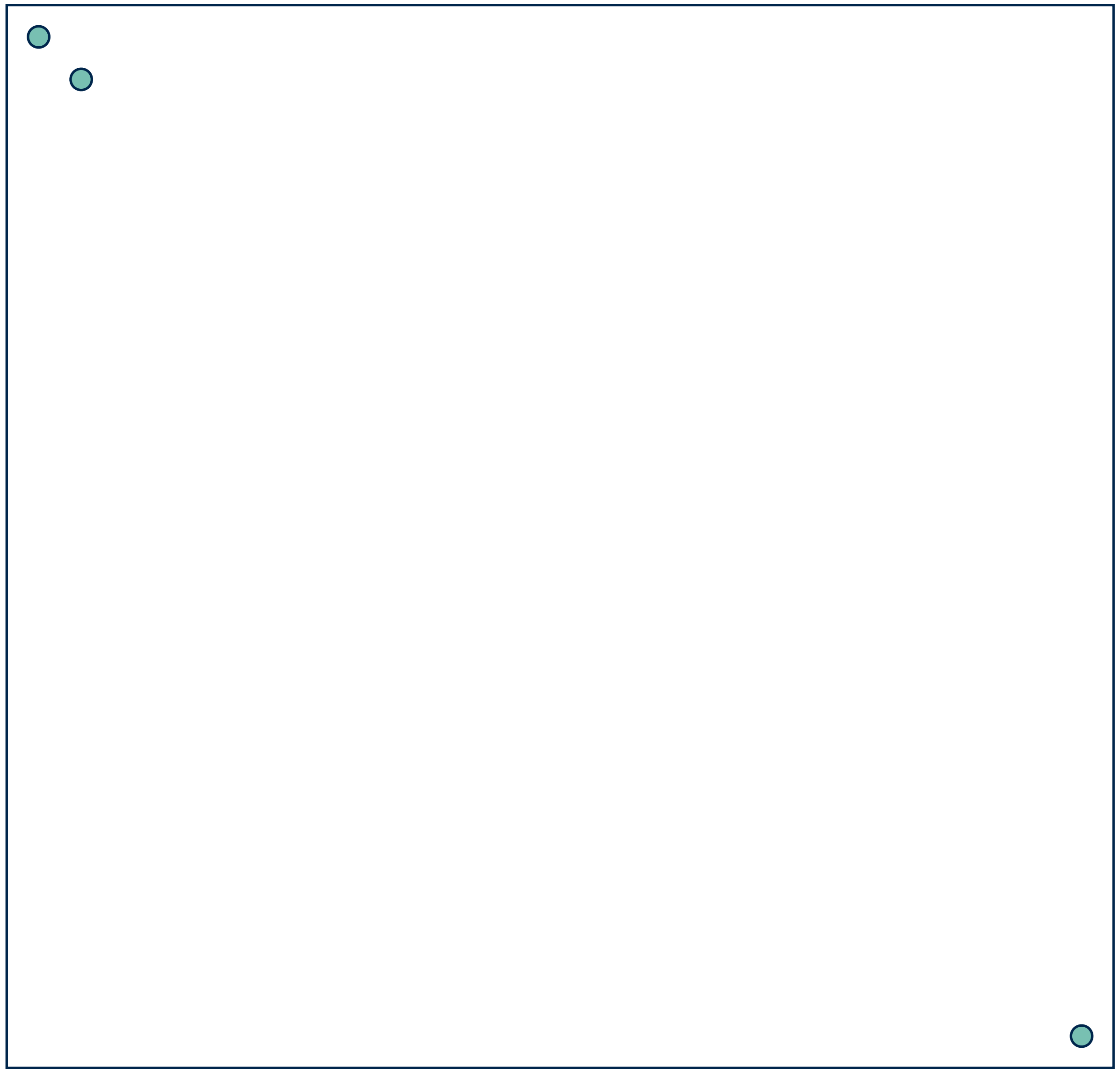
(Lighthouse3

OTHER COMMON TESTS

- Interference Testing (which cells an object collides with)
- Ray Intersection Testing (which cells a ray intersects)

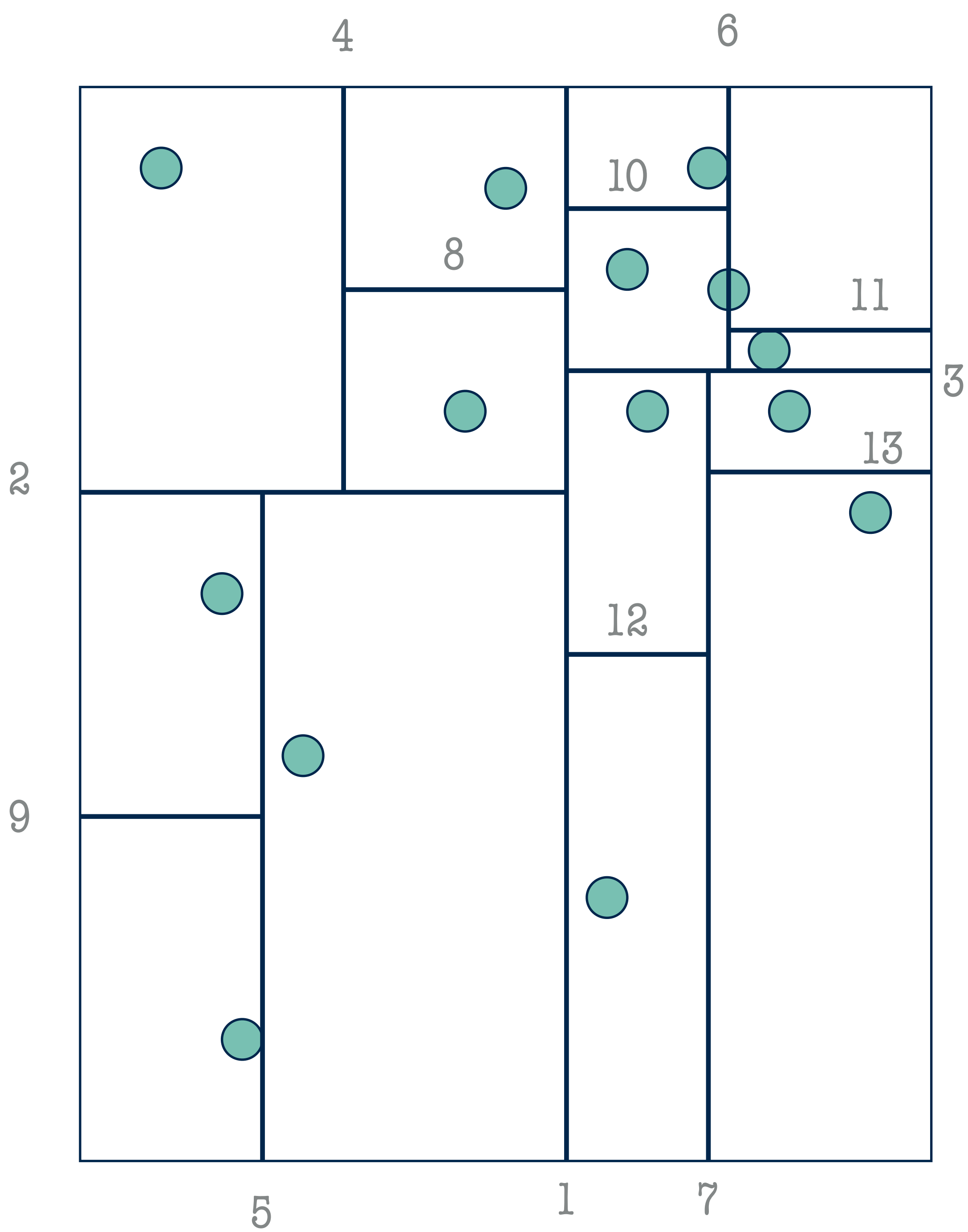
OCTREE PROBLEMS

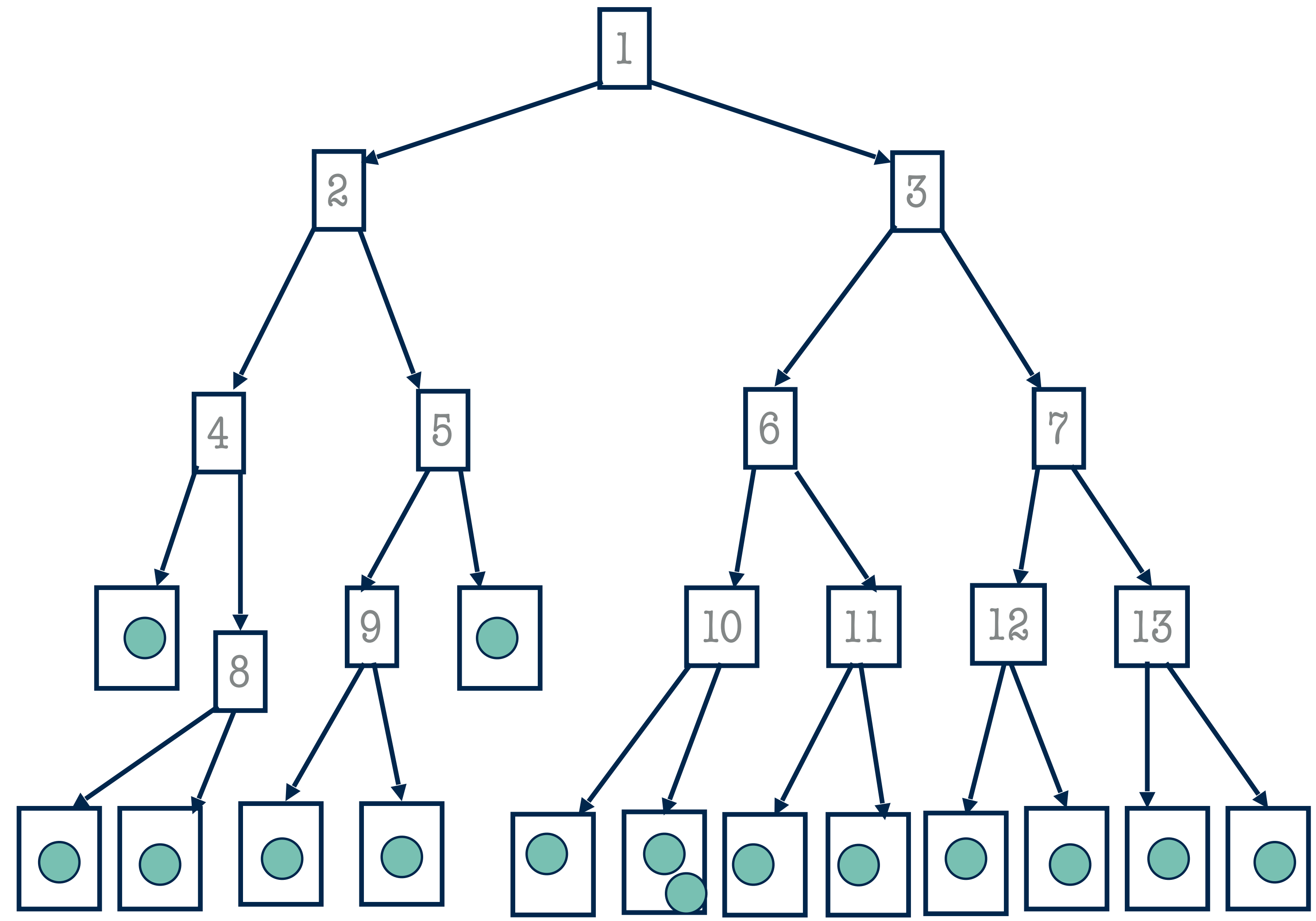
- Octrees become very unbalanced if the objects are far from a uniform distribution
- Problem is the requirement that cube always be equally split amongst children



A bad octree case

SOLUTION: KD-TREE





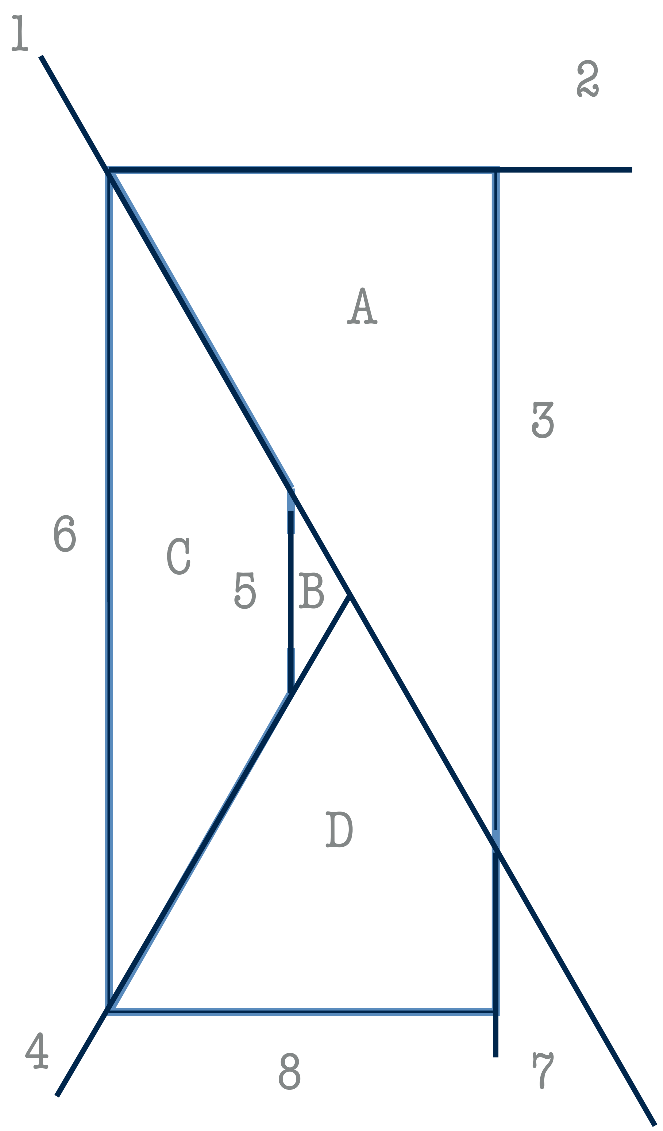
KD-TREE

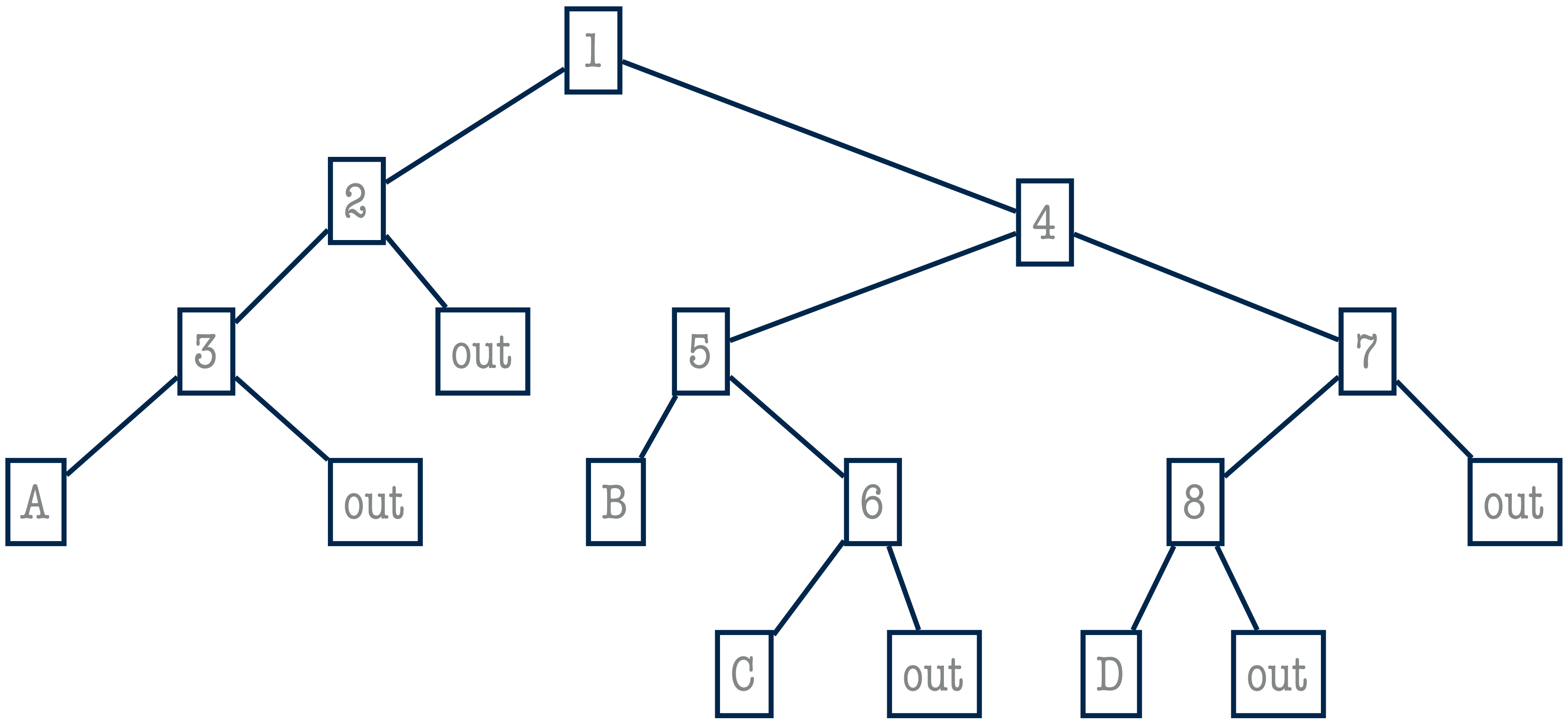
- Properties
  - Node represents a rectilinear region (faces are **axis-aligned**)
  - Node associated with an axis-aligned plane that cuts its region into two
  - Cut planes can be different in different sub-trees at the same level
- Applications
  - Ideal when axis-aligned planes cut space into meaningful cells
  - View frustum culling extends trivially to kd-tress
  - Often used as data structures for other algorithms (e.g. visibility/rendering)

BSP TREES

- Binary Space Partition trees
  - Sequence of cuts that divide a region of space into two
- Cutting planes can be of any orientation
  - Generalization of kd-trees (kd-tree is an axis-aligned BSP tree)
- Divides space into convex cells
- Industry standard for spatial subdivision in many game environments
  - General enough to handle most common environments
  - Easy enough to manage and understand
  - Big performance gains

BSP EXAMPLE





- Notes:
    - Splitting planes end when they intersect their parent node's planes
    - Internal node labeled with planes, leaf nodes with regions

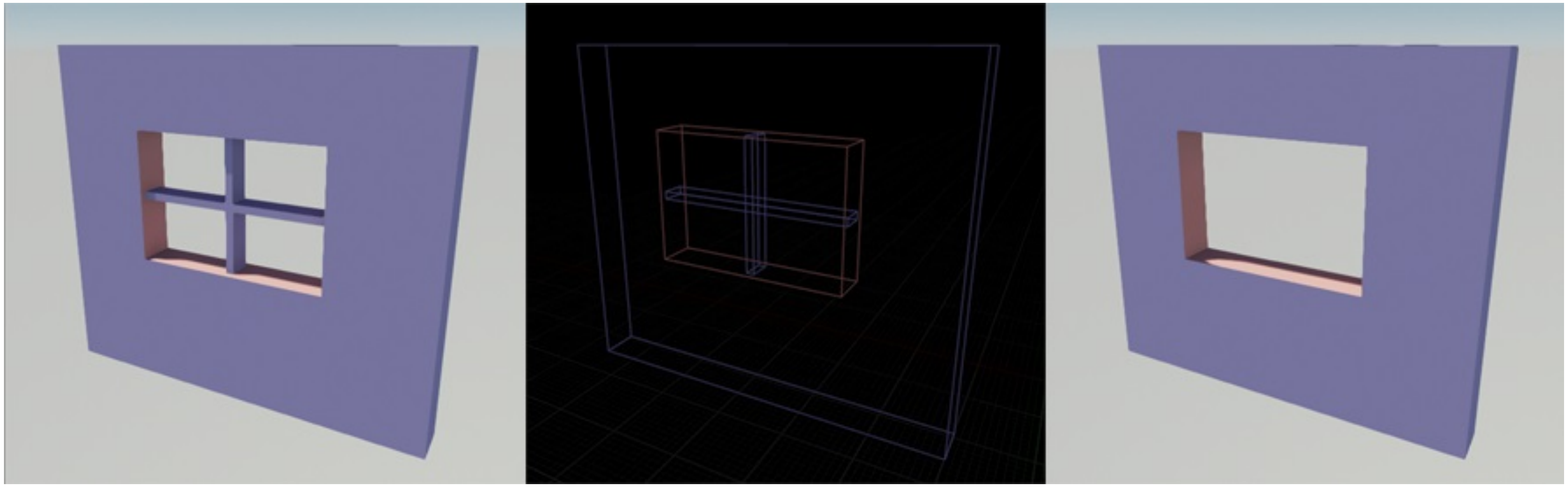
CHOOSING SPLITTING PLANES

- Goals:
    - Trees with few cells
    - Planes that are mostly opaque (best for visibility calculations)
    - Objects not split across cells
- Some heuristics:
    - Choose planes that are also polygon planes
    - Choose large polygons first
    - Choose planes that don't split many polygons
    - Choose planes that evenly divide the data
    - User selects or otherwise guides the splitting process
    - Random choice of splitting planes doesn't do too badly!

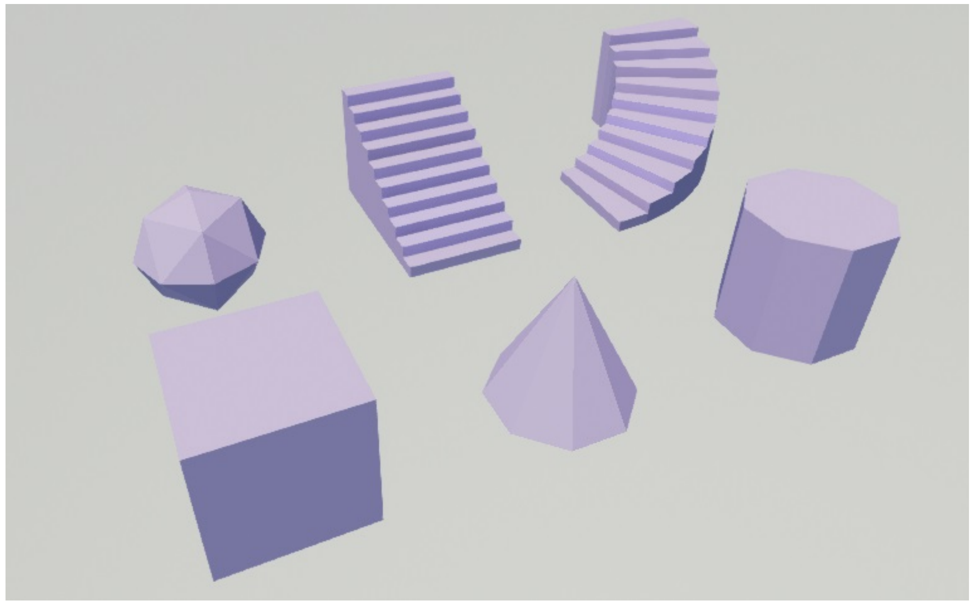
BSPS IN GAMES

- BSP trees can partition space as you would with an octree or kd-tree
  - Leaf nodes are cells with lists of objects
  - Cells typically correspond to “rooms” but don’t have to
  - Fast visibility and ray-trace queries
- Polygons used in the partitioning are defined by the level designer
  - A **brush** is a region of space that contributes planes to the BSP
  - Artists lay out brushes, then populate them with objects
  - Additional planes may be specified
    - Sky planes for outdoor scenes to block off visibility
    - Planes defined to block sight-lines, but not visible themselves

BSP BRUSHES IN UNREAL



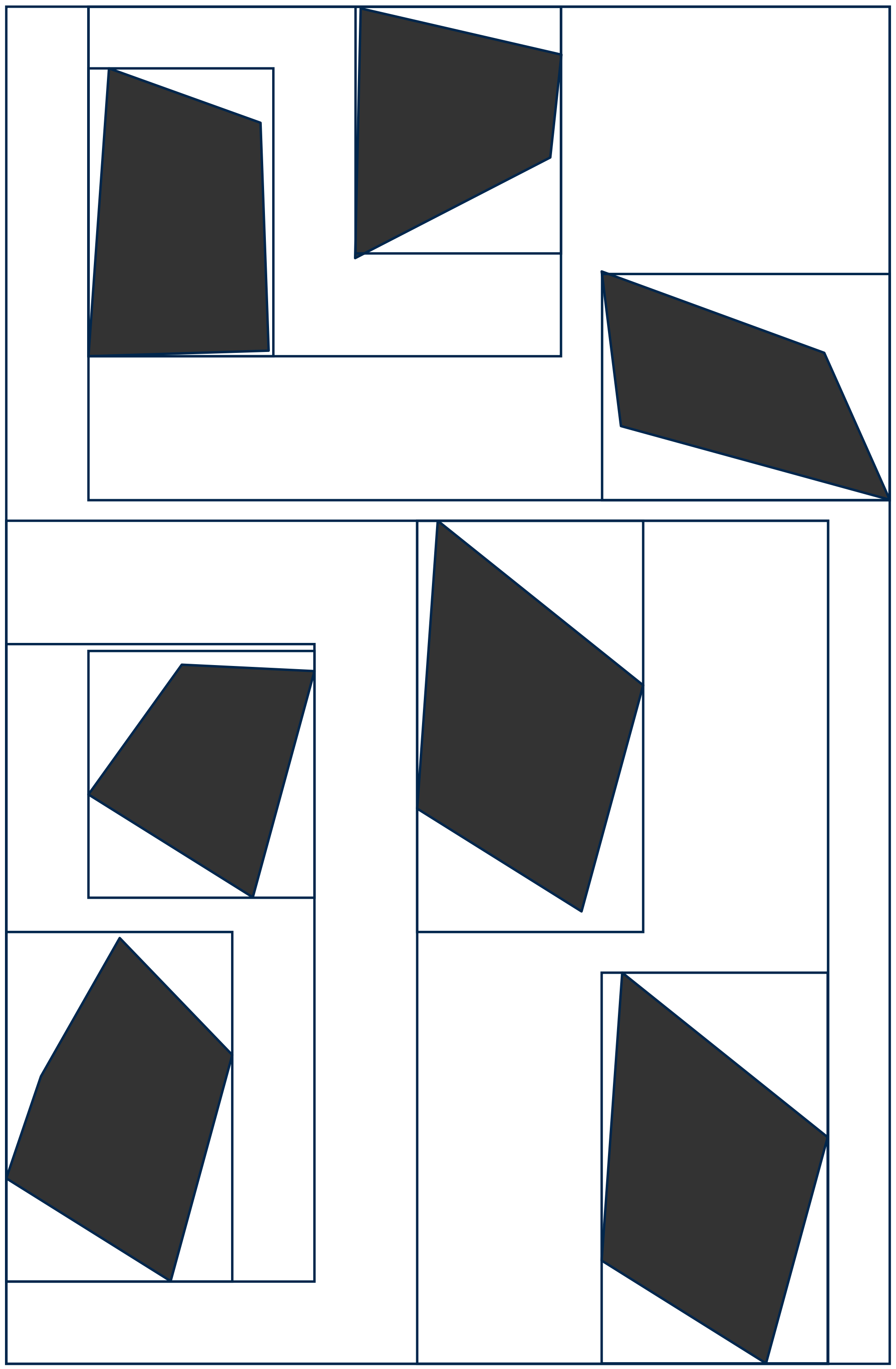
- Used for level block out
  - CSGs (constructive solid geometries) generated to form planes
  - Stored and efficiently rendered using a BSP

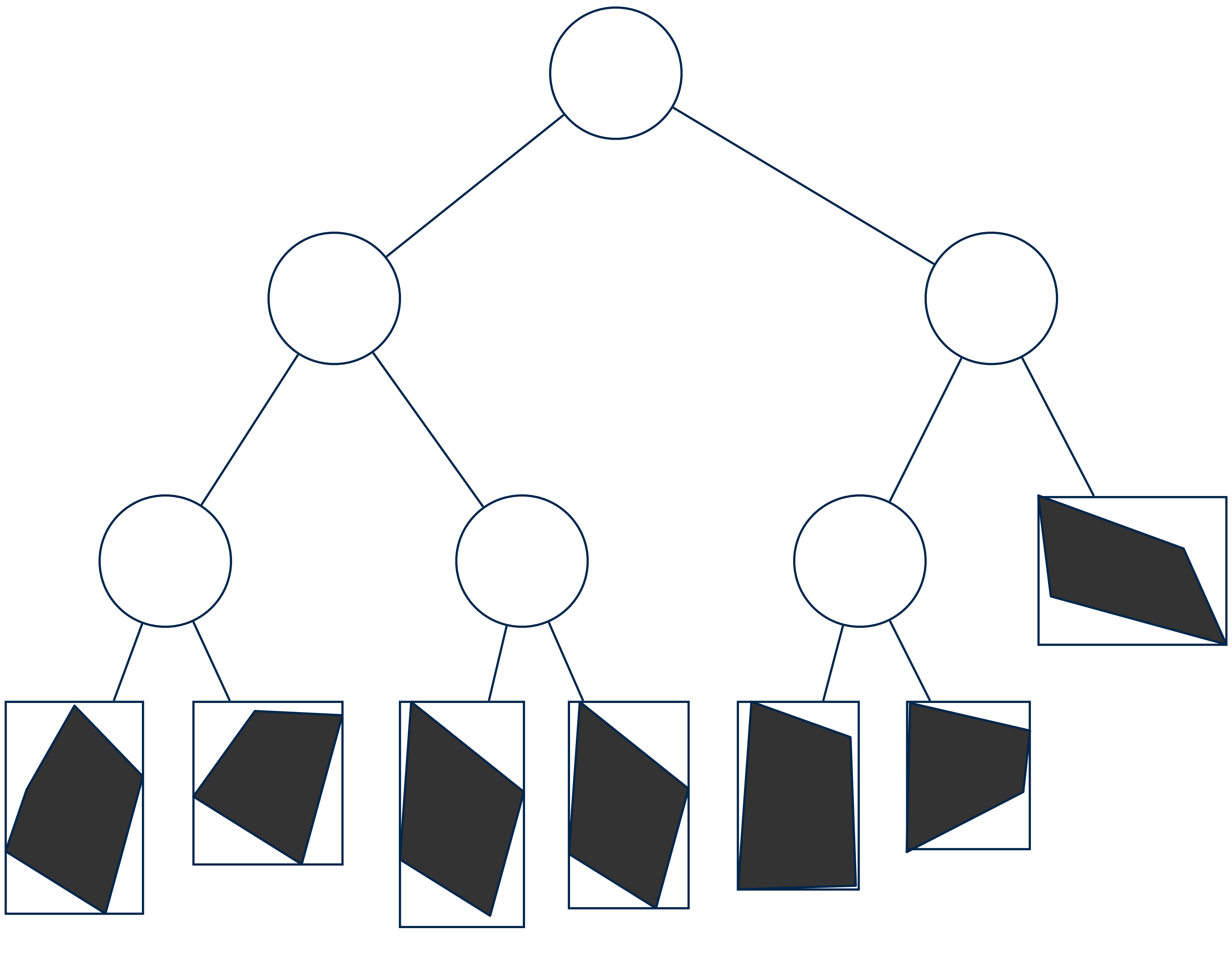


BOUNDING VOLUME HIERARCHIES

- BVHs have a bounding volume for each object
  - Spheres, AABBs etc
- Parent bounds bound their children's bounds
  - Children bounds the same type as their parent's
  - Fixed or variable number of children per node
- No notion of cells

BVH EXAMPLE





FURTHER READING

- Flocks, Herds, and Schools: a Distributed Behavioral Model (http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/)
- Particle Systems (https://natureofcode.com/book/chapter-4-particle-systems/)