COMP4002/G54GAM Games

Physics and Simulation

The Game Loop (attempt 2)

start game
start render thread
while(user doesn't exit)
{
    how much time has elapsed?
    get user input
    get network messages
    simulate game world(**elapsed time)**
    resolve collisions
    move objects
    play sounds
    sleep(desired-elapsed time)
}
exit

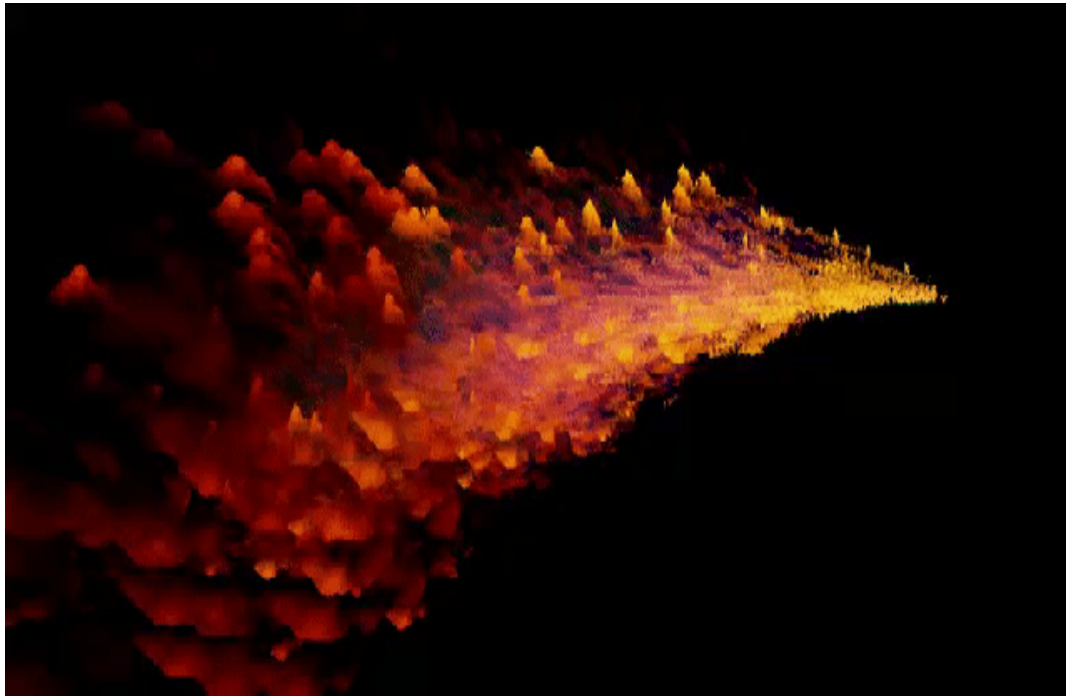
while(user doesn't exit)
{
    draw graphics
    sleep(desired-elapsed time)
}
exit

Physics Engine

- Responsibilities
  - Update and maintain positions and orientations of items
    - In response to user input
  - Determine what movements are allowable
  - Determine what collisions have happened
  - Generate data regarding those collisions
    - Drive game play events
  - Resolve those collisions
- *Physical* representations
  - Particle
  - Rigid Body
  - Soft Body

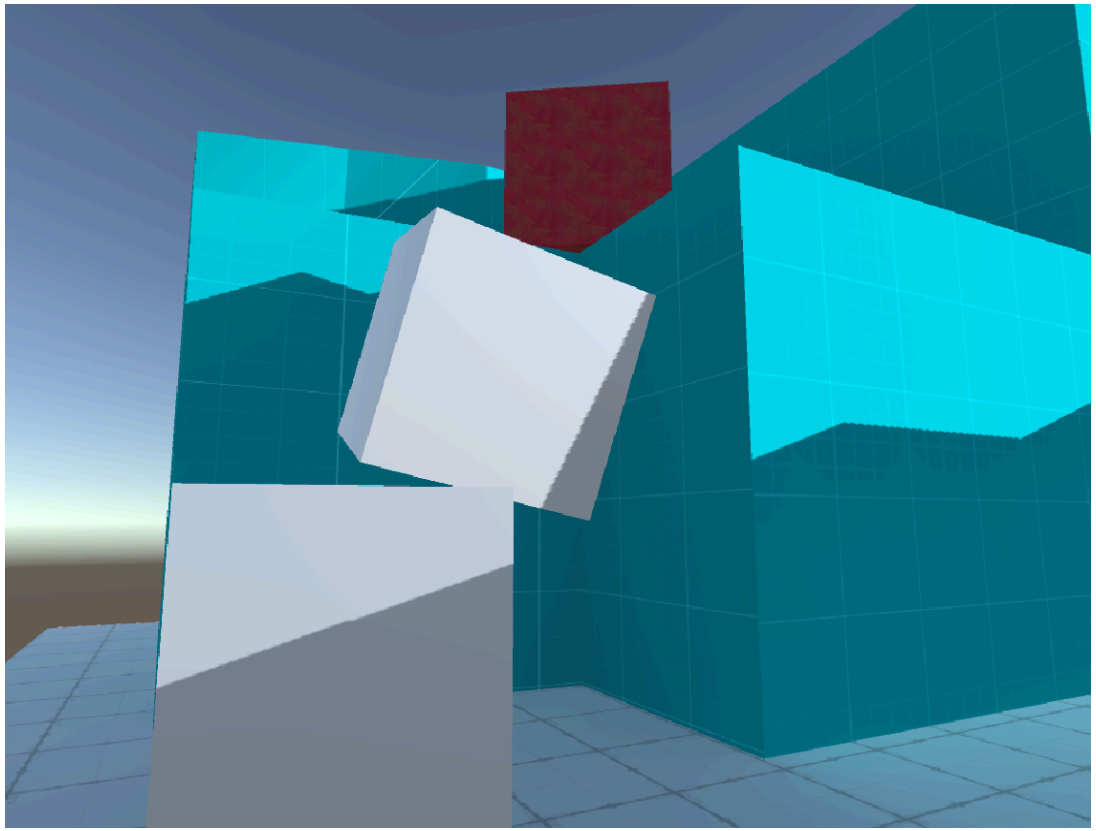
Particle Physics

- A particle based physics system doesn't care about collisions, only motion
  - All objects in the system are particles
    - Dimensionless points in space.
    - They have no radius
  - Because they have no radius, they don't collide
    - Might have nominal mass
      - Attract / repel one another
  - More applicable to graphical effects than physics behaviours



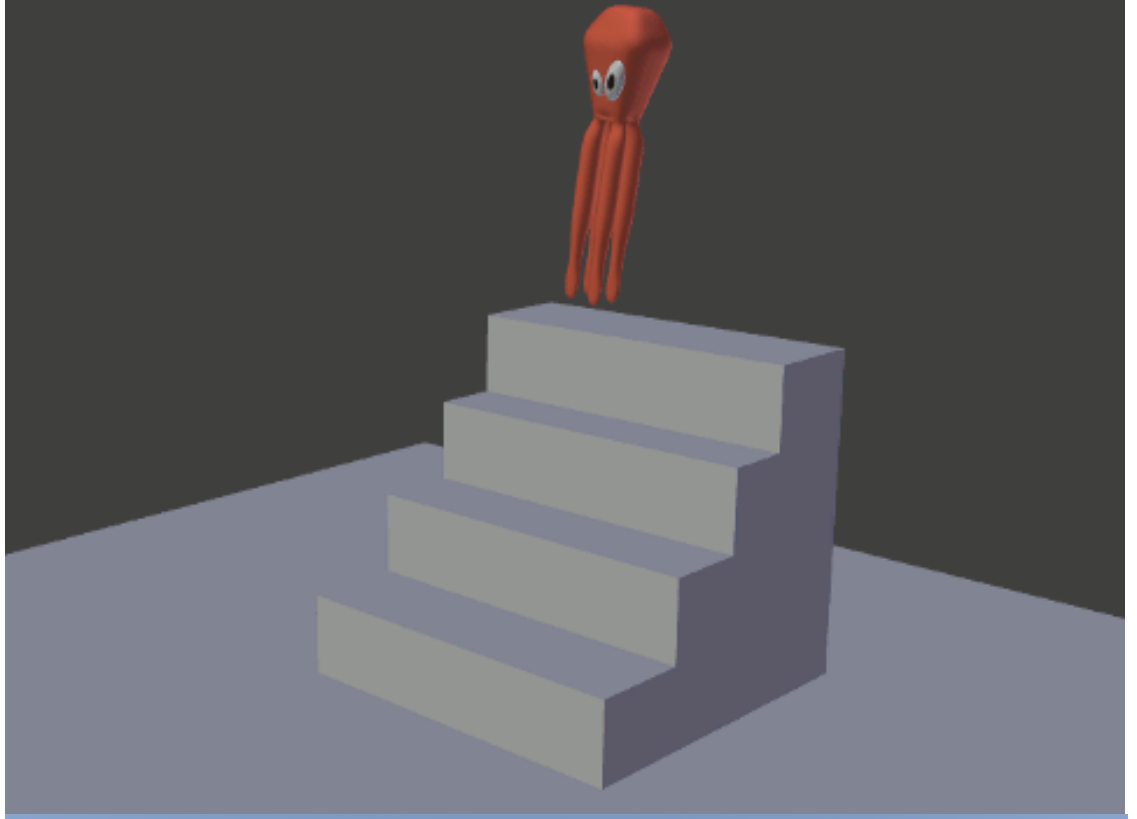
Rigid Body Physics

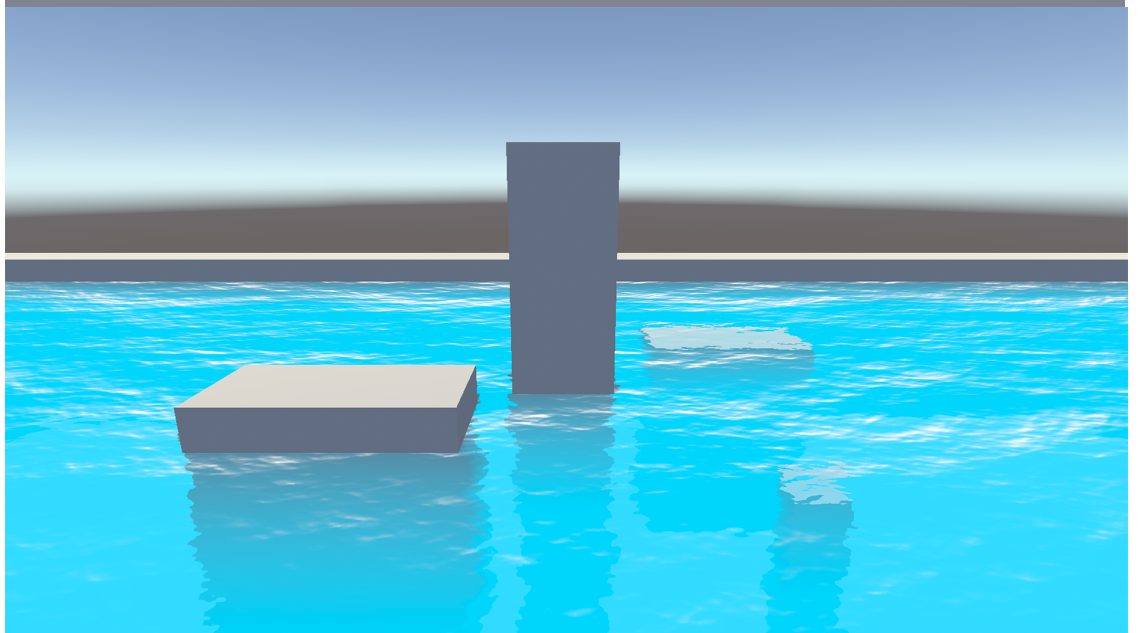
- Rigid bodies have a physical presence
  - A sphere, a cuboid, a capsule, a height-map
- Defining characteristic is that they **don’t deform**
  - A balloon in the real world can be squeezed and stretched
  - A balloon represented in a rigid body physics system cannot
- Because they have physical presence, they have dimensionality
  - Radius, width, height,
  - They can collide
  - They can can experience torque and rotate
- Computationally significantly less expensive
  - Than computing physical properties of soft bodies



Soft Body Physics

- Cloth, cushions, water
  - Anything that deforms when subject to a force
  - Most accurately represented as a soft body.
- Soft bodies must be discretised in a physics engine
  - Many interconnected small bodies
    - Connected via *constraints* (e.g. springs)
  - Significant computational expense
- Focused on situations where application enhances user experience/immersion
  - Often replaced by graphical effect rather than appropriate simulation



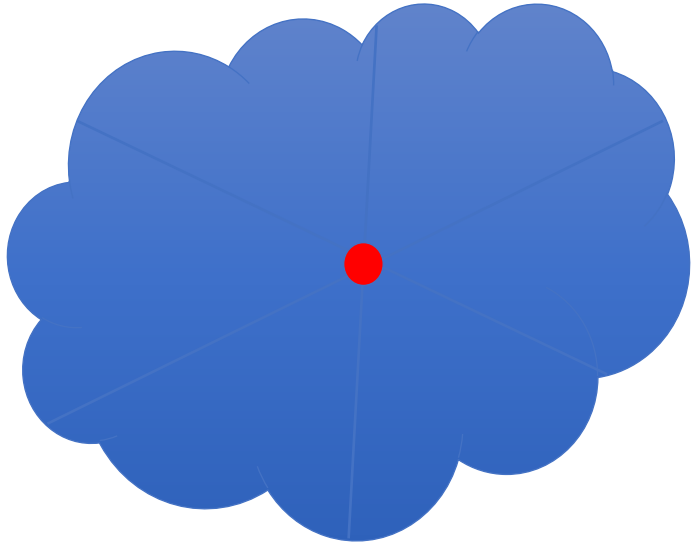


Physics in Games

- **“Simulate the game world”**
  - Iterate the fundamental model
  - Moving objects based on elapsed time
    - Kinematics
      - Motion ignoring external forces
      - Only considering position, velocity
    - Dynamics
      - The effect of forces on the objects
- “Resolve Collisions”
  - Collisions between moving objects
    - Collision detection
      - Did a collision occur?
    - Collision resolution
      - Respond to the collision
      - Are we now in a restricted state due to constraints?

Physical Position vs Representation

- Typically ignore geometry
  - How the object is *physically shaped*
    - Also do not worry about how it *looks*
    - Only needed for collisions
  - Focus on how it *moves*
- Every object as a *point*
  - *Centroid*
    - The average of all points
    - Or the center of mass
    - Generally all objects have a uniform density
  - Often the default **transform** of the entity / actor
  - Consider the motion of the *point*



Kinematics

- Basic Motion
  - “the motion of objects without reference to the forces which cause the motion”
- Determine an object’s displacement *s* at time *t*
  - Typically already know it from a previous time

  s’(t) = ds/dt = v

- Assume constant velocity *v*

  $s(t+\Delta t) = s(t) + v\Delta t$

  $\Delta s = s(t+\Delta t) - s(t) = v\Delta t$

- What is *Δ* (delta) t?

Kinematics

- Velocity
  - Rate of change of displacement over time
  - v=ds/dt
- Acceleration?
  - Rate of change of velocity over time
  - a=dv/dt
- Given Acceleration, how can we obtain Velocity at any given time?
  - by integrating acceleration with respect to time
  - v=$\int a.dt$
- Given Velocity, how can we obtain Displacement?
  - by integrating velocity with respect to time
  - s=$\int v.dt$

Analytical Solutions

- Acceleration
  - rate of change of velocity
    - Rate of change of rate of change of position
- Integrate twice to calculate position

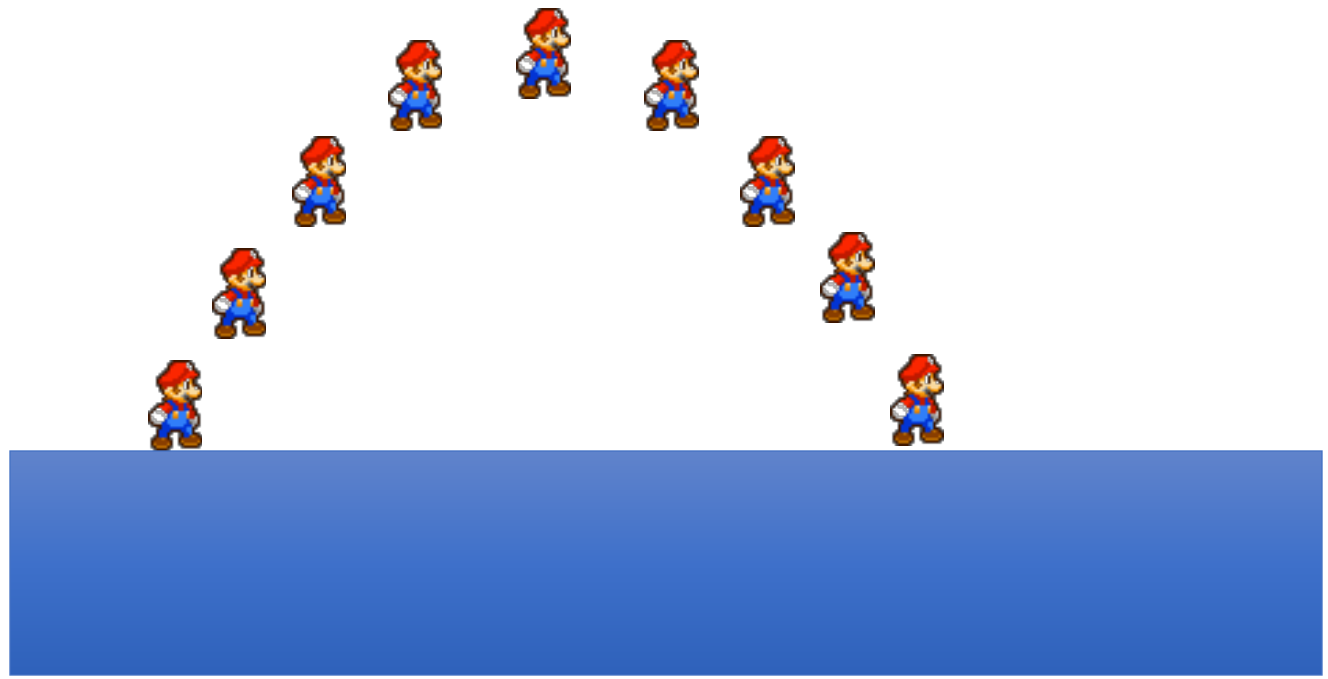
  $s(t)'' = g = -9.81\ m/s^2$

  $s(t)' = gt + v_0$

  $s(t) = 0.5gt^2 + v_0t + s_0$

  $s = ut + 0.5at^2$

- An analytical solution to solving an ordinary differential equation
  - A simple, *closed-form* function
  - Describes position for all possible values of time t
  - “Perfect” simulation of gravity / parabolic movement



Newtonian Linear (Angular) Dynamics

- Newton's laws of physics
  - A body will remain at rest or continue to move in a straight line at a constant speed unless acted upon by a force.
  - The acceleration of a body is proportional to the resultant force acting on the body, and is in the same direction as the resultant force
  - For every action there is an equal and opposite reaction
- Forces affect movement
  - Based on the mass m of an object
    - F=ma
    - Impulses, repulsion, inertia
  - *Constrained* by springs, joints, connections, other objects
  - Calculate changing velocity and acceleration from the forces applied over the frame
- Need a general solution
  - Generally cannot find closed-form solutions for movement under force for all values of time t
    - c.f. *3-body problem*
  - Force, acceleration are rarely constant
    - Function of position, velocity, friction

Numerical Integration

- Given position, velocity and force

  s(t), v(t), F(t, p, v)

  - ...find s(t+Δt), v(t+Δt)
- Can be approximated by a Taylor Series

$$s(t+\Delta t) \approx s(t) + \Delta t\, s'(t) + \frac{\Delta t^2}{2} s''(t) + \ldots + \frac{\Delta tn}{n!}\, s^{(n)(}t)$$

- If Δt is small, approximate this *linearly*

$s_0(t+\Delta t) \approx s_0(t) + \Delta t\, s_0'(t)$

- Euler Integration
  - Approximate forward both velocity and position each step

$$v(t+\Delta t) = v(t) + a(t)\Delta t = v(t) + \frac{F(t)}{m}\Delta t$$
$$s_0(t+\Delta t) = s_0(t) + v(t)\Delta t$$

Euler Integration*

- Starting with a stationary object at the origin weighing one kilogram, we apply a constant force of 10 Newtons and step forward with time steps of one second

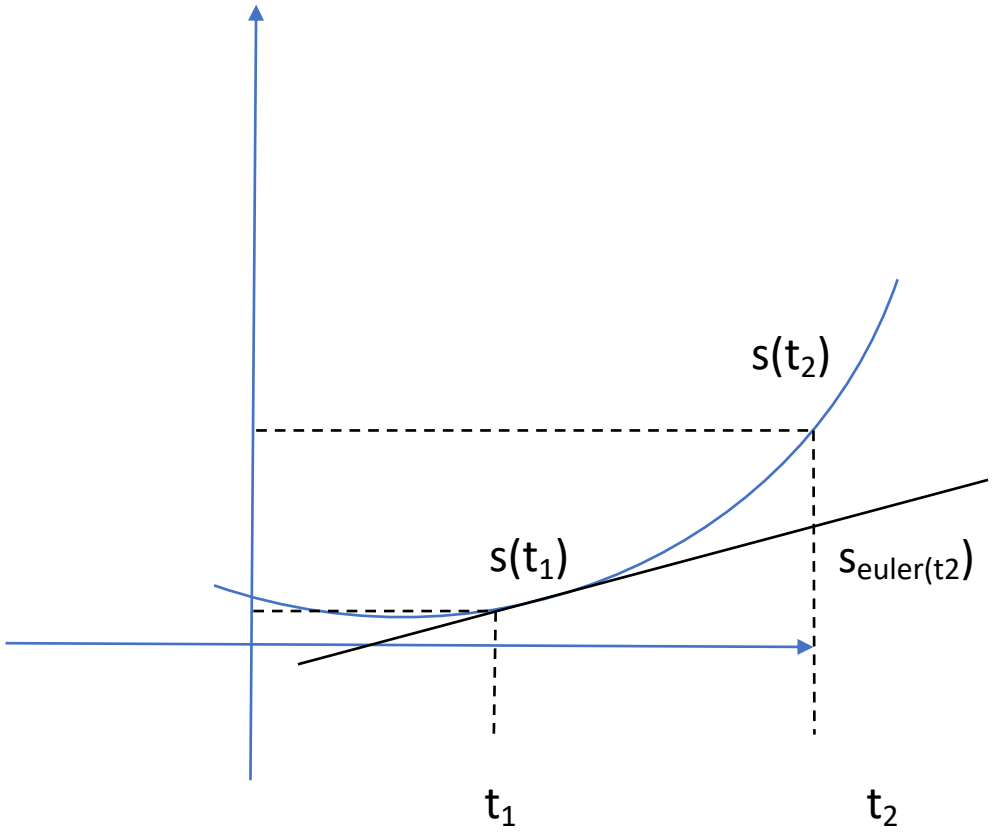
```
float t = 0; float dt = 1;
float velocity = 0; float position = 0;
float force = 10; float mass = 1;

while ( t <= 10 )
{
   position = position + velocity * dt;
   velocity = velocity + ( force / mass ) * dt;
   t = t + dt;
}
```

time	position	velocity
0	0	0
1	0	10
2	10	20
3	30	30
4	60	40
5	100	50
6	150	60
7	210	70
8	280	80
9	360	90
10	**450**	100

Explicit Euler Integration*, Implicit Euler Integration, Semi-implicit Euler Integration, Verlet Integration, RK2, RK4

Euler Integration Errors



$s = ut + 0.5at^2$

$s = (0*10) + 0.5(10)(10)^2$

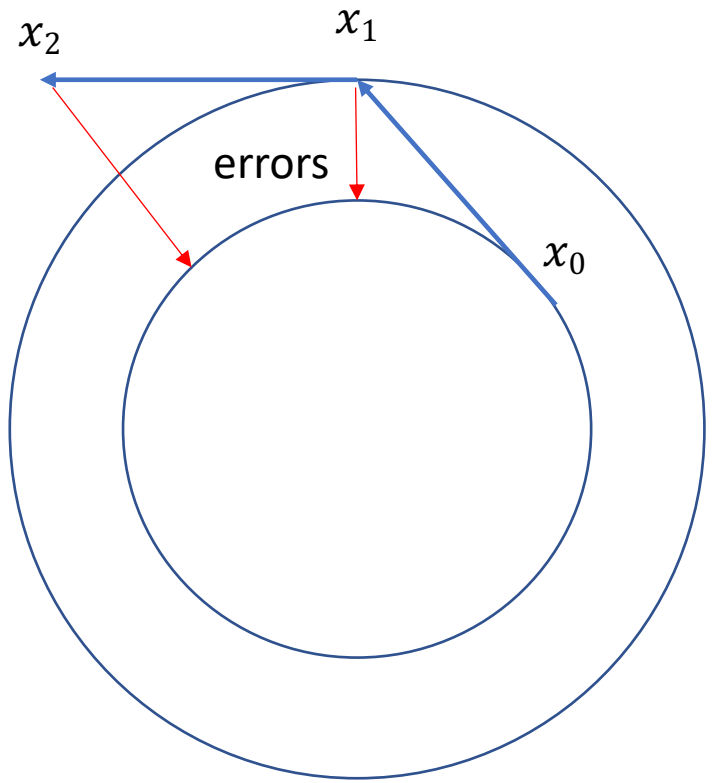
$s = 0.5(10)(100)$

$s = \mathbf{500}$

- Velocity is calculated per frame
  - But *changing* constantly
- As Δt approaches 0
  - Accuracy becomes perfect

Properties of Numerical Integration

- Convergence
  - Does the solution approach the real solution as Δt approaches 0?
- Order
  - How bad is the error?
    - Cannot remove it
  - Error is proportional to some power of Δt
    - Square of the step size
- Stability
  - Does the solution add energy to the system?
    - Errors **accumulate**
    - Energy gain causes extreme glitches
    - Simulations explode
  - Requires ad-hoc solutions
    - Clamping to max-values
    - Manual damping
  - Remove energy from the system
    - Damping

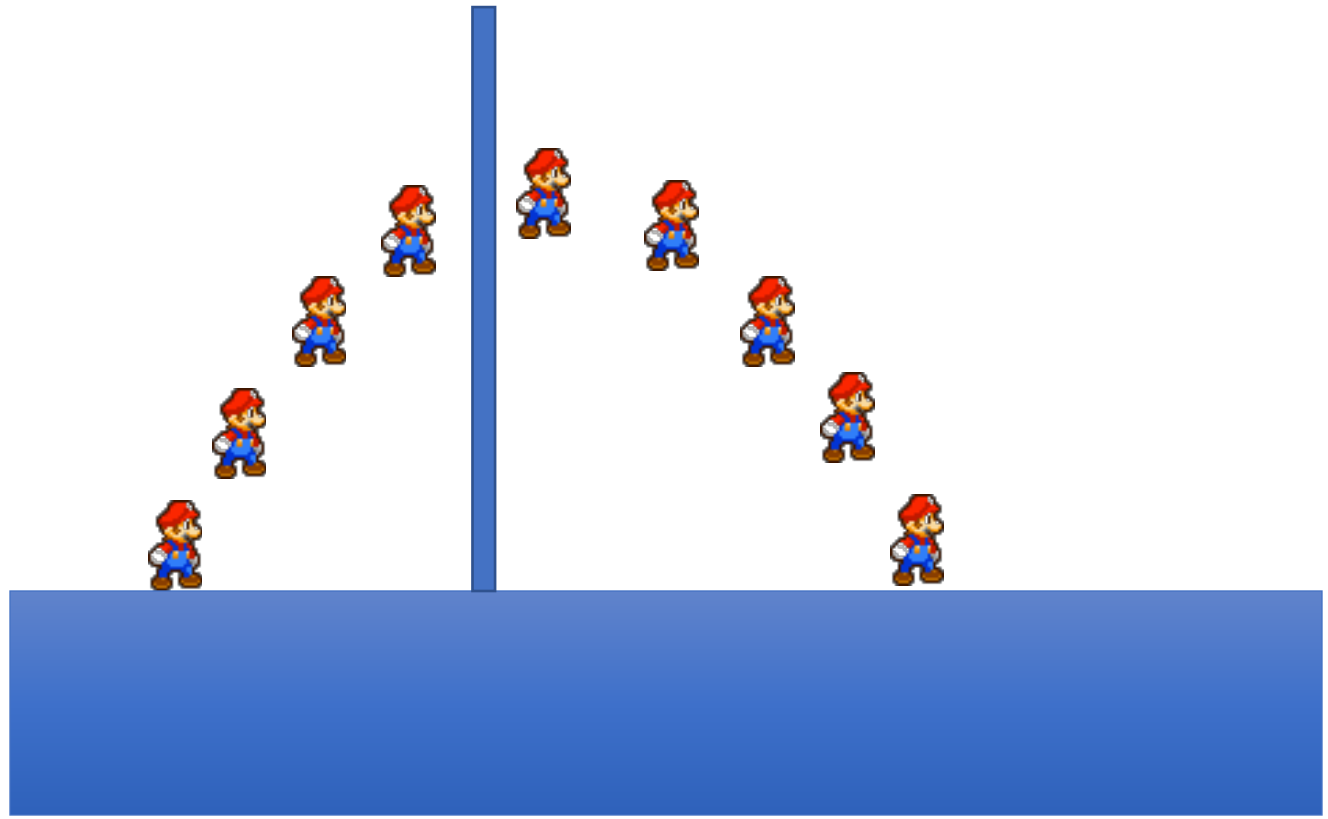


Fixed Time Step

- Determinism
  - Consistent error regardless of CPU
    - Modifies how the simulation behaves
    - Informs damping parameters
- If Δt too large
  - Errors cause unexpected behaviours
- Typically a multiple of expected framerate
  - 30Hz, 60Hz, 120Hz
  - Can't simulate ½ Δt
    - Changes simulation behaviour
    - Simulate(½ Δt) Simulate(½ Δt) != Simulate(Δt)
  - "The renderer produces time and the simulation consumes it in discrete Δt sized chunks"
    - Renderer running at variable rate
- "It doesn't matter"
  - As long as the system is stable, it's doing its job
  - If the physics system performs 120 updates per second, and the level takes 20 minutes to complete, the physics system only needs to be stable for 144,000 updates

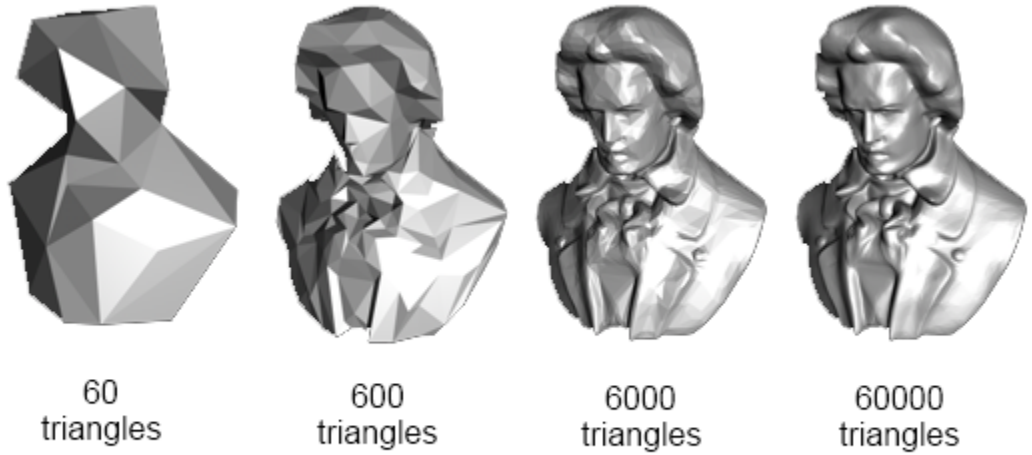
Physics in Games

- “Simulate the game world”
  - Iterate the fundamental model
  - Moving objects based on elapsed time
    - Kinematics
      - Motion ignoring external forces
      - Only considering position, velocity
    - Dynamics
      - The effect of forces on the objects
- “Resolve Collisions”
  - Collisions between moving objects
    - **Collision detection**
      - **Did a collision occur?**
    - Collision resolution
      - Respond to the collision
      - Are we now in a restricted state due to constraints?

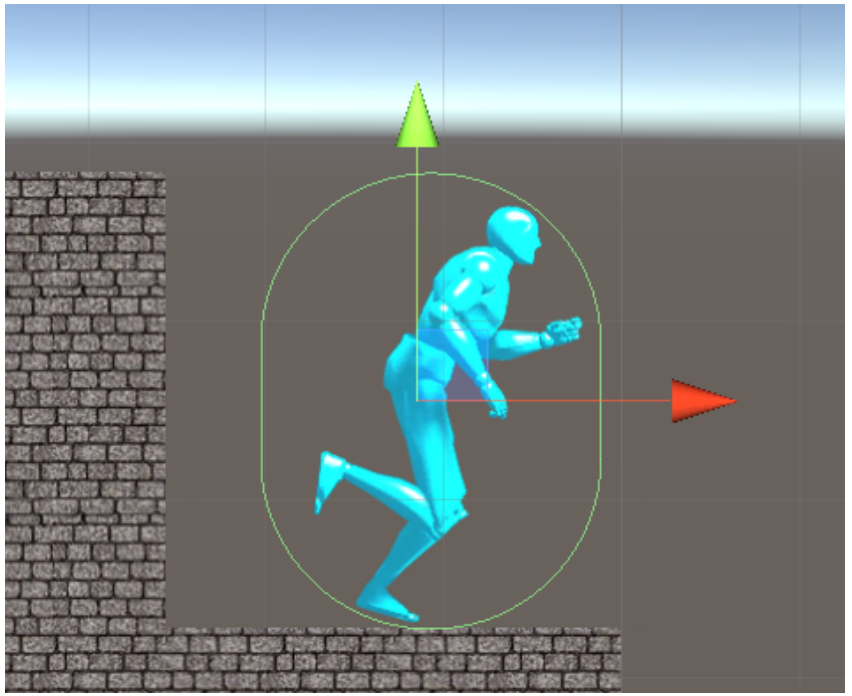


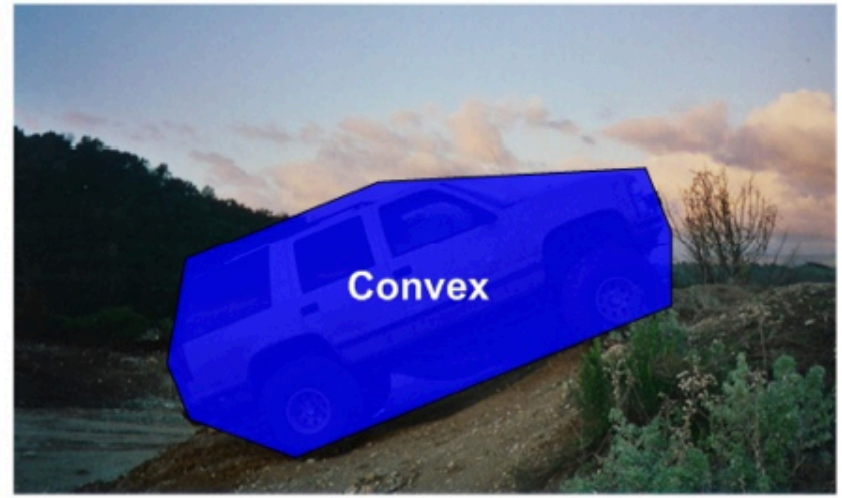
Physical Representation

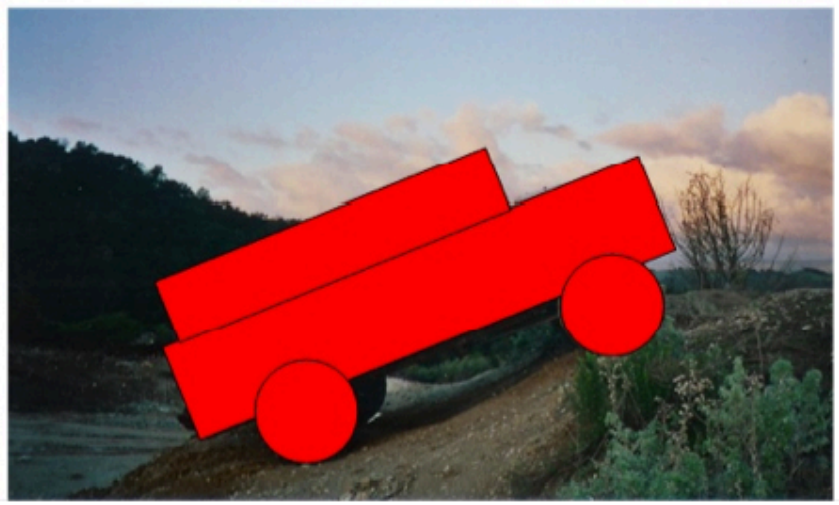
- Collisions require **geometry**
  - Sprites, 3d collision mesh, collection of primitive objects
  - Points are no longer enough to model the object
  - Need to know if, when and how objects meet when they collide
- Physical representation of objects does not need to precisely map to graphical representation
  - Only needs to be detailed enough for its interactions with the environment to appear believable
  - Capsule == character movement
  - Does not need to be as detailed as the graphical model
    - Complexity of an object is a function of its number of components
    - Faces, primitives
    - We reach a point of diminishing returns in physics quickly
- Simple convex mesh
  - Easiest shapes to compute collisions with
  - Break complex concave shapes into multiple convex components
  - Triangles are always convex

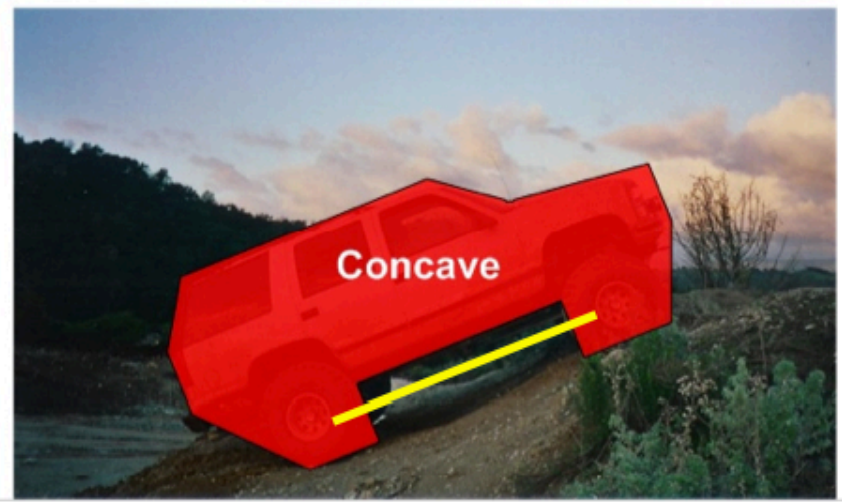


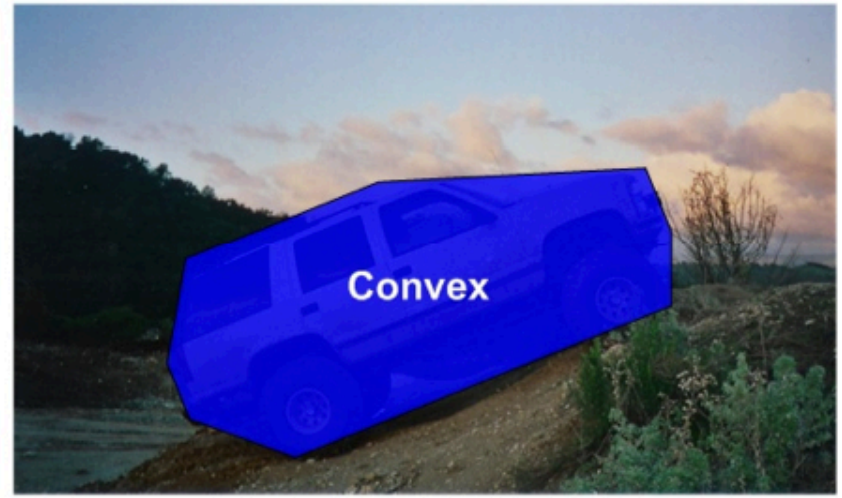
Diminishing returns - 15 years ago even doubling the amount of triangles resulted in a much better mesh. Now, multiplying the amount by 10 hardly does.

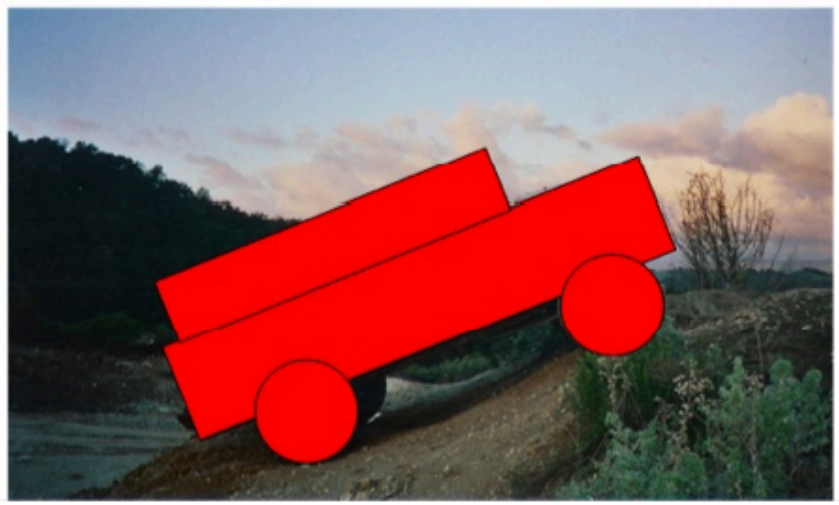


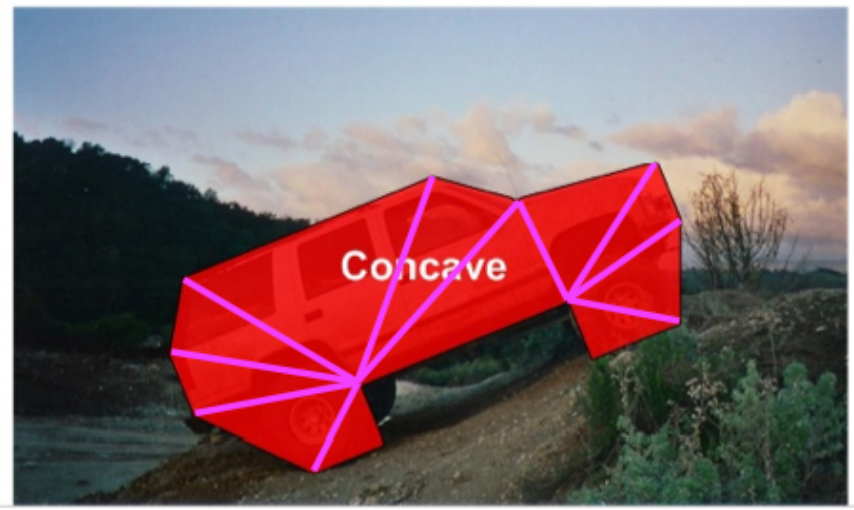

Convex

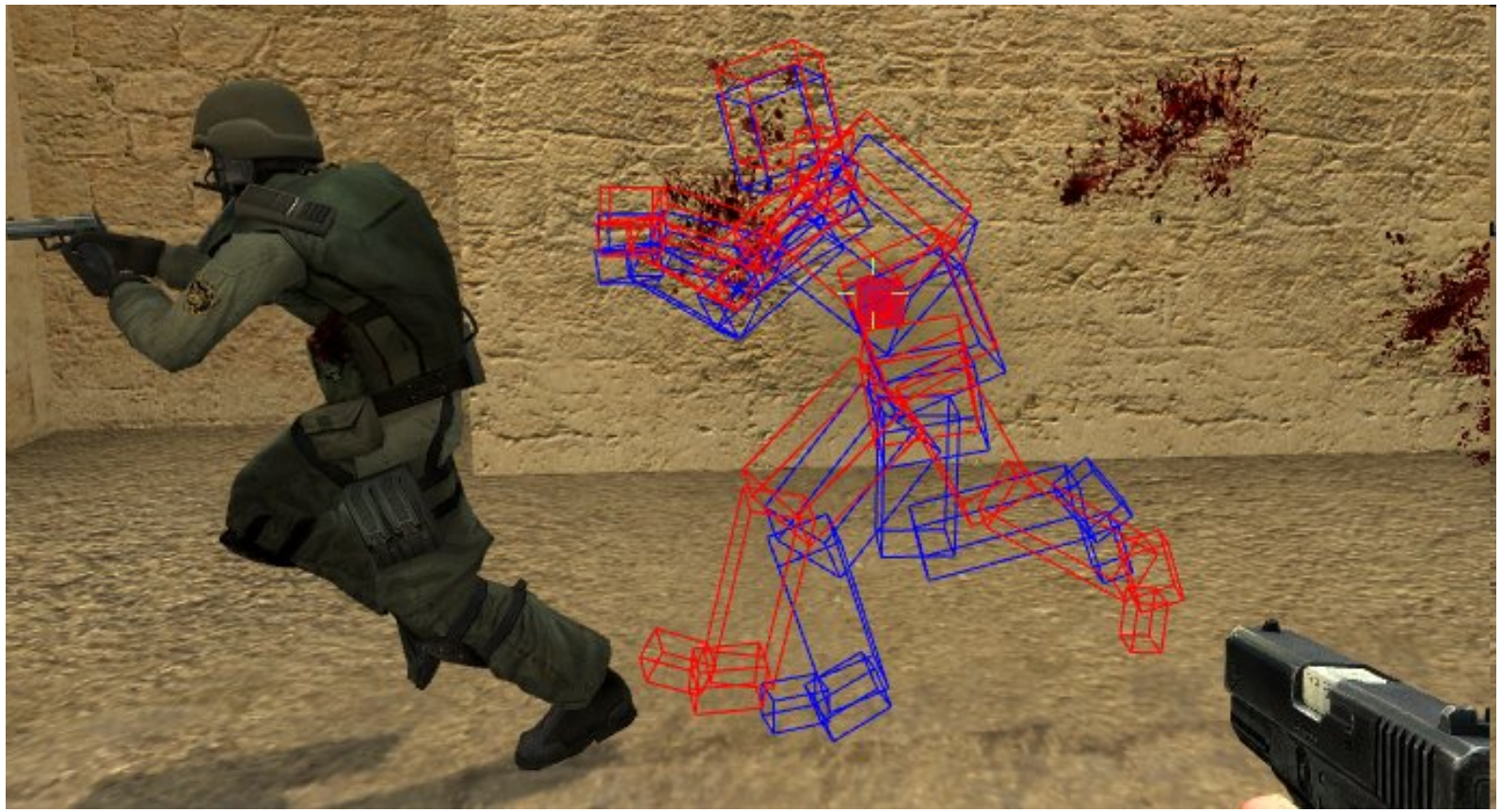



Concave


Convex




Concave



Scalability

- Key constraint
  - 16ms per clock tick (60Hz physics update)
    - Limited time to perform collision detection
- Naïve approach
  - Check two objects pixel-by-pixel for collisions
    - Shapes can be complicated
  - Check every object pair for collisions
    - Many objects can potentially collide
  - $O(n^2)$ checks
    - 200 objects = 19900 checks per tick
- Scaling approaches
  - Reduce number of collision pairs to check
  - Reduce cost of each collision check

Collisions

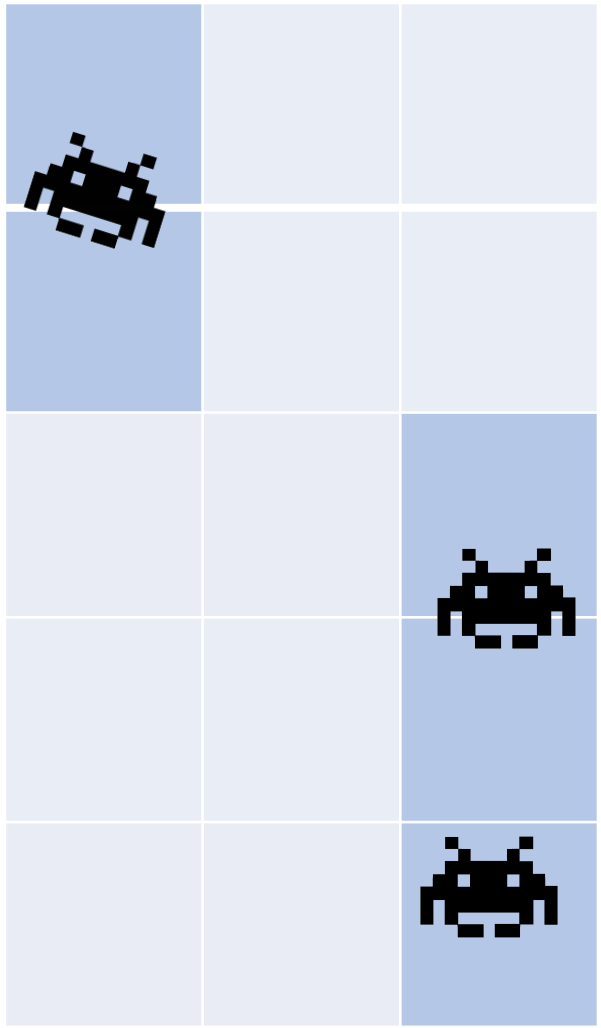
- Collision Detection
  - **If** two objects intersect
  - Did the bullet hit the opponent
- Collision Resolution
  - Determining **when** two objects came into contact
    - At what point during their motion did they intersect
      - With what velocity
  - Determining **where** two objects intersect
    - Which points on the objects touch during a collision
      - Determine how to act next, and in what direction
- Complexity increases
  - If < when < where

Broad Phase vs Narrow Phase

- If there are many small objects in large level
  - Each object only has the potential to collide with nearby objects
  - How can we dramatically reduce the number of checks?
- Broad Phase collision
  - Perform a quick, inaccurate pass over *n* objects to determine pairs with potential to intersect, *p*
- Narrow Phase collision
  - Perform *p x p* check of object pairs to determine actual collision
    - Triangle Intersection calculation
    - Gilbert-Johnson-Keerthi (GJK) algorithm

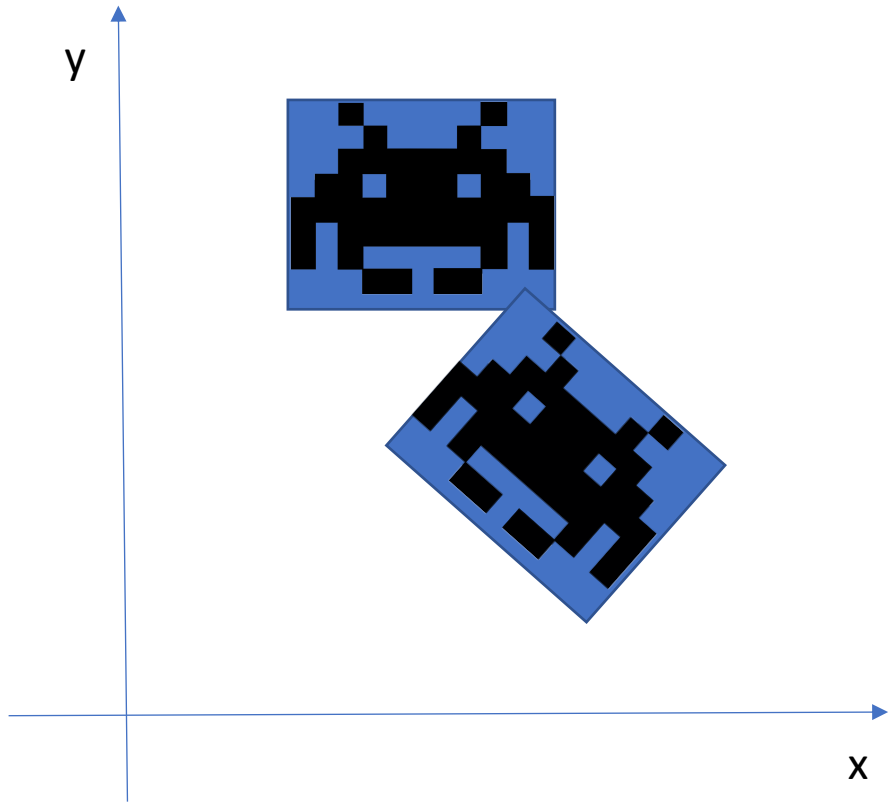
Broad Phase Optimisation Approaches

- Grid
  - Divide level into a grid
  - Place each object in a square
  - Only check the contents of the square against neighbouring squares
- Application-specific
  - Identify classes of object that should collide
    - E.g. Aliens need not collide with one another



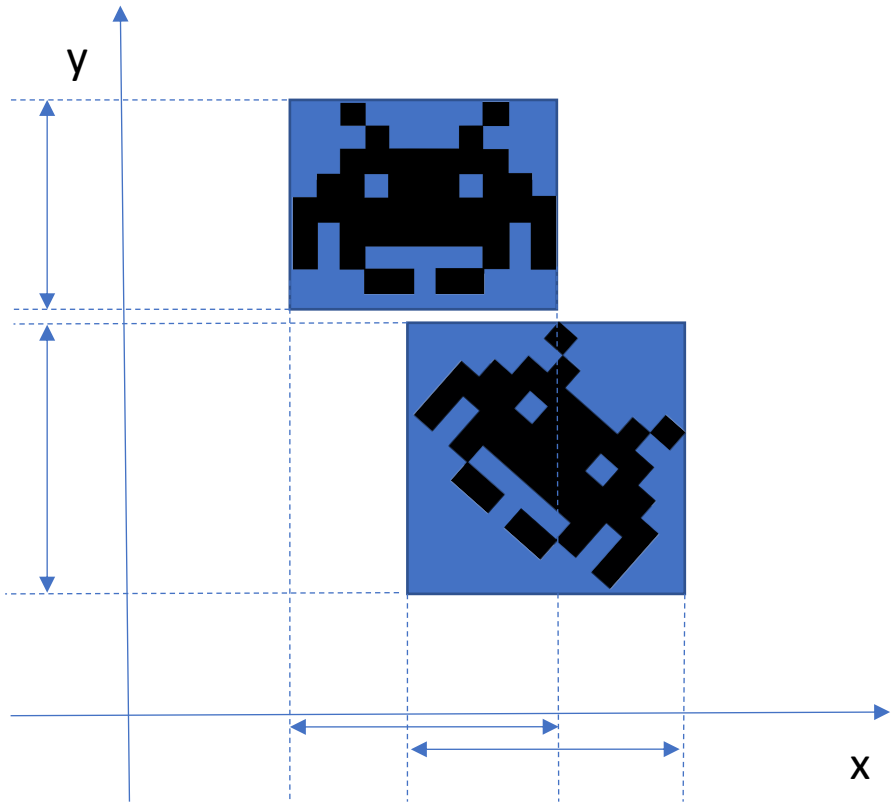
Broad Phase Optimisation Approaches

- Oriented Bounding Box
  - OOB
- Rectangular bounds
  - Minimal rectangle fitting
  - Angled to best fit
- Often less tight a fit
  - Creates false positives
- Just as slow as triangles
  - Boxes may have many different orientations
  - Corner cases



Broad Phase Optimisation Approaches

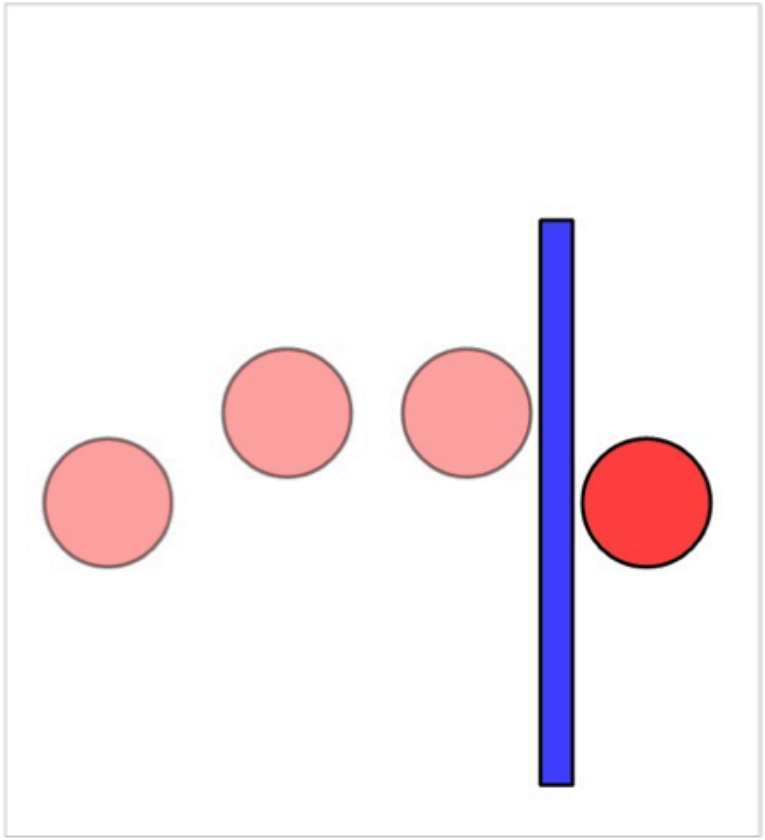
- Axis Aligned Bounding Box
  - AABB
- Similar to OBB
  - Rectangular fit
  - Align with x-y axes
- Often a very poor fit
  - False positives likely
- Checking is very cheap
  - Project box onto axes
  - Check whether intervals overlap

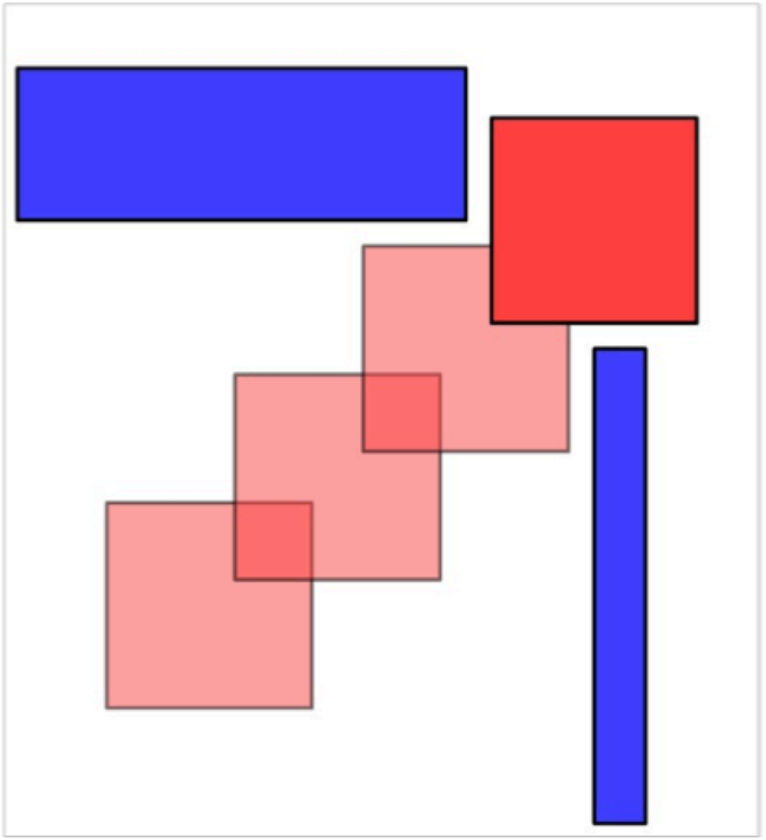


Collision Detection - Tunneling

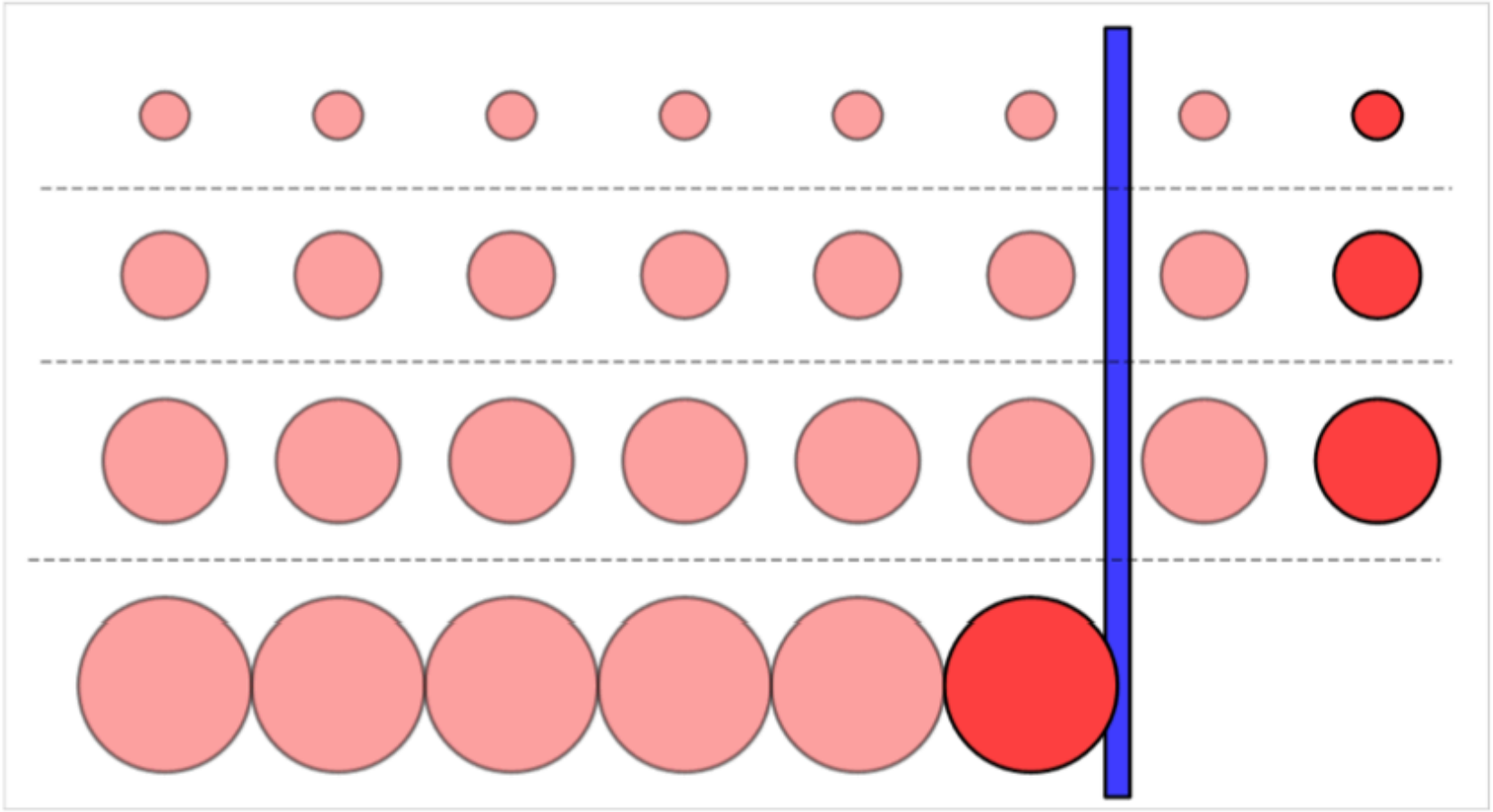
- Collisions in mid-step can lead to tunneling
  - Objects that pass through one another
    - Not colliding at start or end of the simulation
    - Instead colliding somewhere in between
    - A false *negative*
  - Ideally consider *when* the collision occurred
    - I.e. when between frames
    - Computationally expensive
- A serious issue for gameplay
  - Players getting into places that they shouldn’t
  - Players missing an event trigger boundary

Tunneling

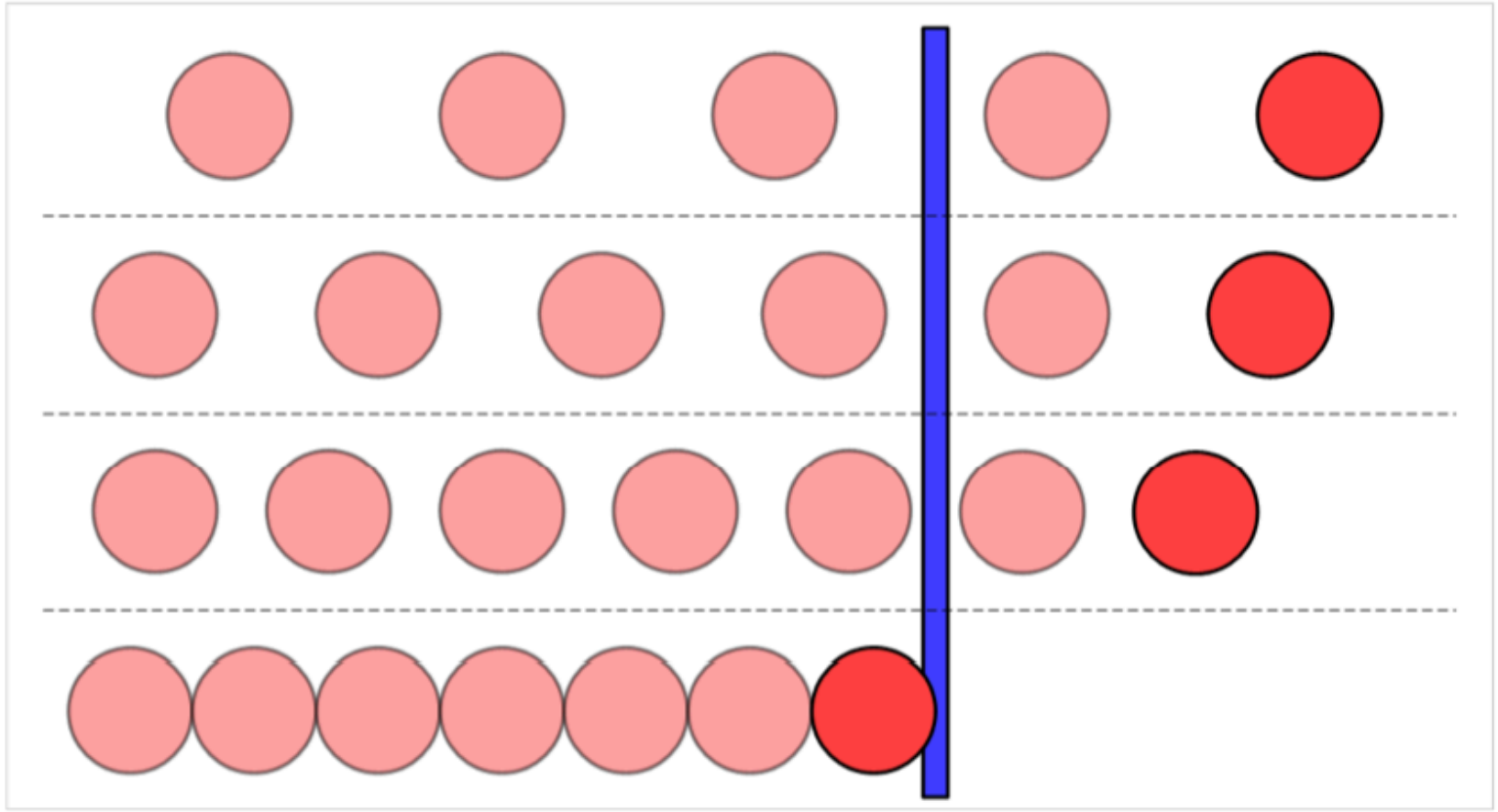




Tunneling – Small Objects



Tunneling – Fast Objects

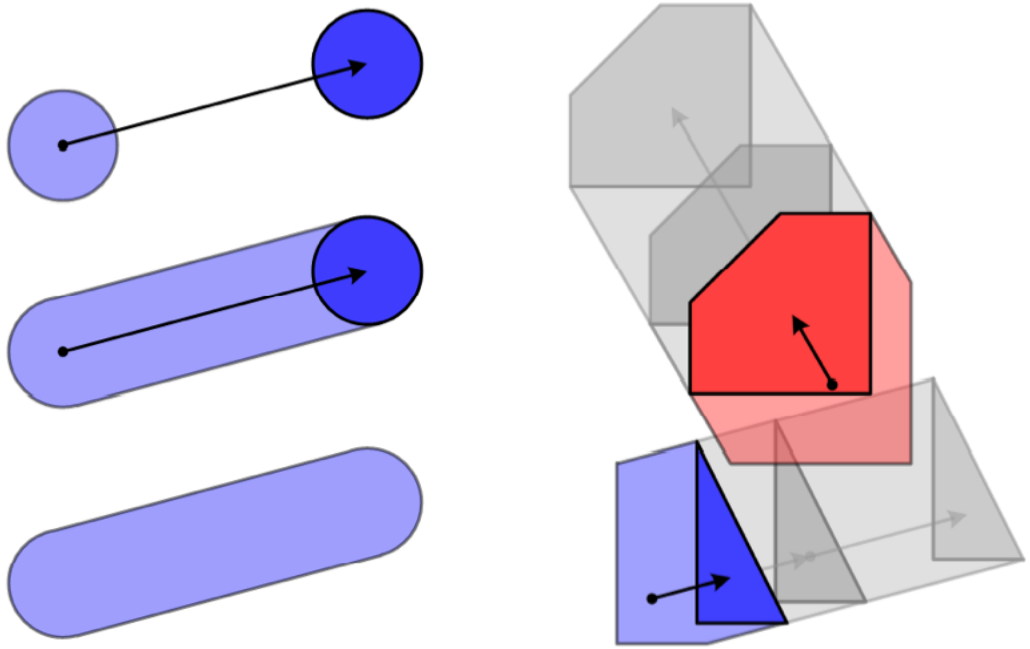


Solutions to Tunneling

- Minimum size requirement
  - Fast objects still tunnel
- Maximum speed limit
  - Speed limit is a function of object size
  - Small and fast objects not allowed
- Smaller time step
  - Essentially the same as a speed limit
- Solutions generally inadequate
  - However physics engine should be expected to operate best within a certain *range*

Swept Collision Shapes

- Movement volume defined by start and end position of the object
  - Swept disk = capsule
  - Swept triangle convex poly
  - Swept convex = convex poly
  - Swept AABB = convex poly
- Still has false positives
  - Removes many problems of false negatives



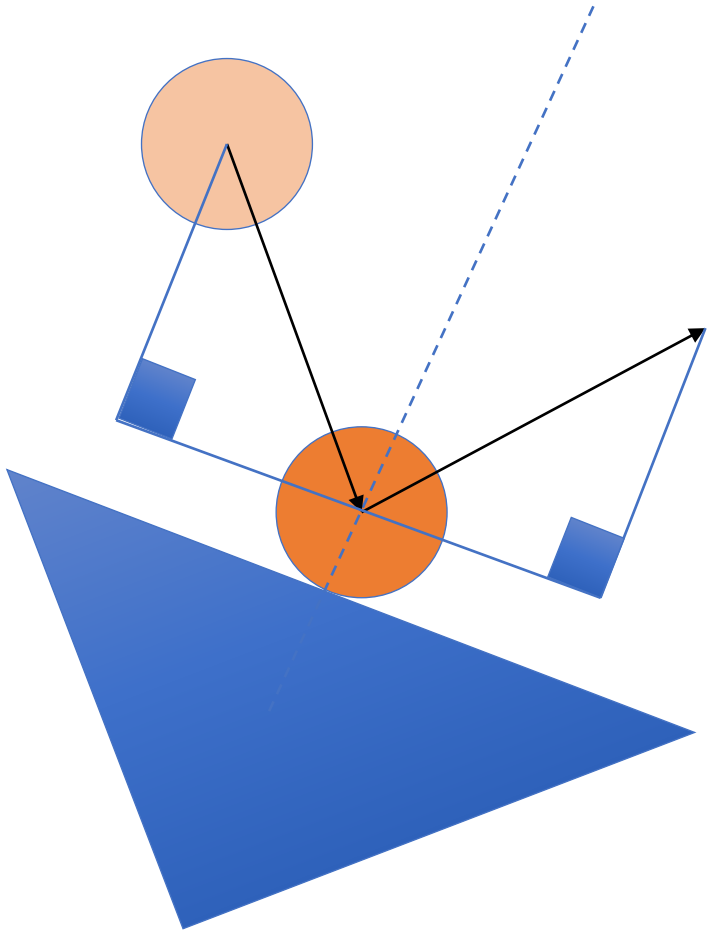
Physics in Games

- “Simulate the game world”
  - Iterate the fundamental model
  - Moving objects based on elapsed time
    - Kinematics
      - Motion ignoring external forces
      - Only considering position, velocity
    - Dynamics
      - The effect of forces on the objects
- “Resolve Collisions”
  - Collisions between moving objects
    - Collision detection
      - Did a collision occur?
    - **Collision resolution**
      - **Respond to the collision**
      - **Are we now in a restricted state due to constraints?**


22

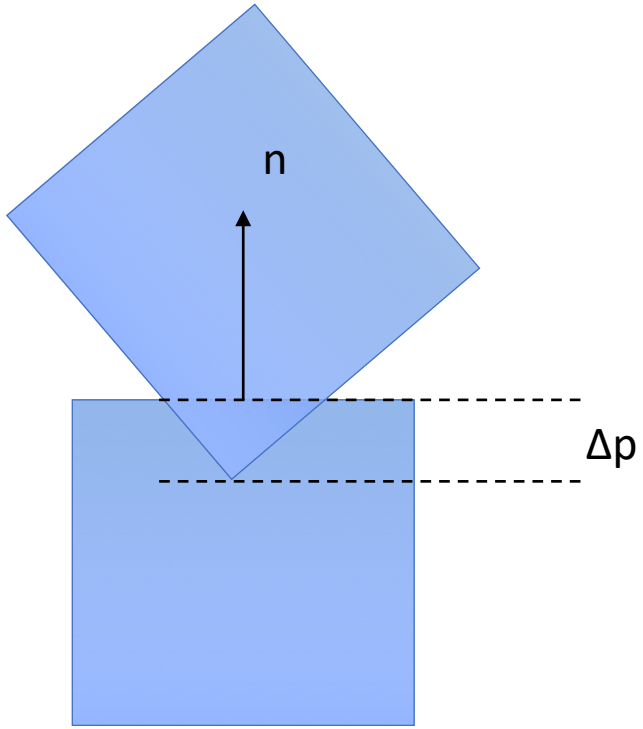
Collision Response

- Calculate the *collision normal*
  - Using the positions of the objects from the previous frame
  - Perpendicular component
    - Direction to reverse movement
    - Conservation of momentum
- Inelastic collisions
  - No energy preserved
    - Objects stop in place (v=0)
    - Easy to implement
- Elastic collisions
  - 100% of energy preserved
  - Snooker balls
- Partially elastic
  - x% of energy preserved
  - What is the object made of?
  - Rubber, steel
    - Coefficient of restitution – a parameter for the engine



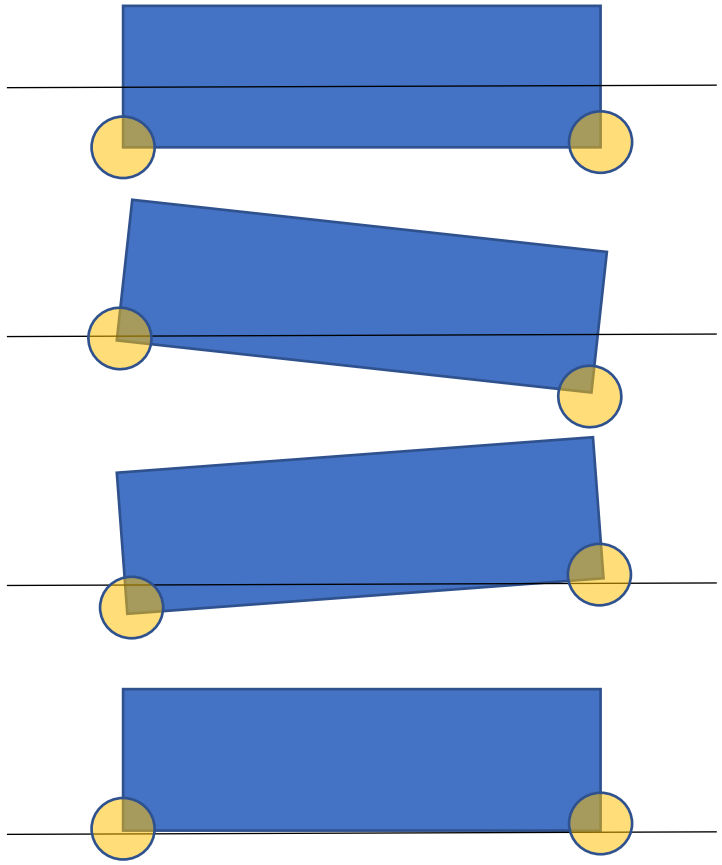
Non-Penetration Constraint

- Collision happens between t -> t + Δt
  - Objects are now *interpenetrating*
- Collision detection algorithms provide:
  - Closest point on one of the objects
  - Collision normal
  - Interpenetration depth.
- What velocity is required to resolve the constraint?
  - Baumgarte Stabilization
    - Apply impulse force proportional to Δp

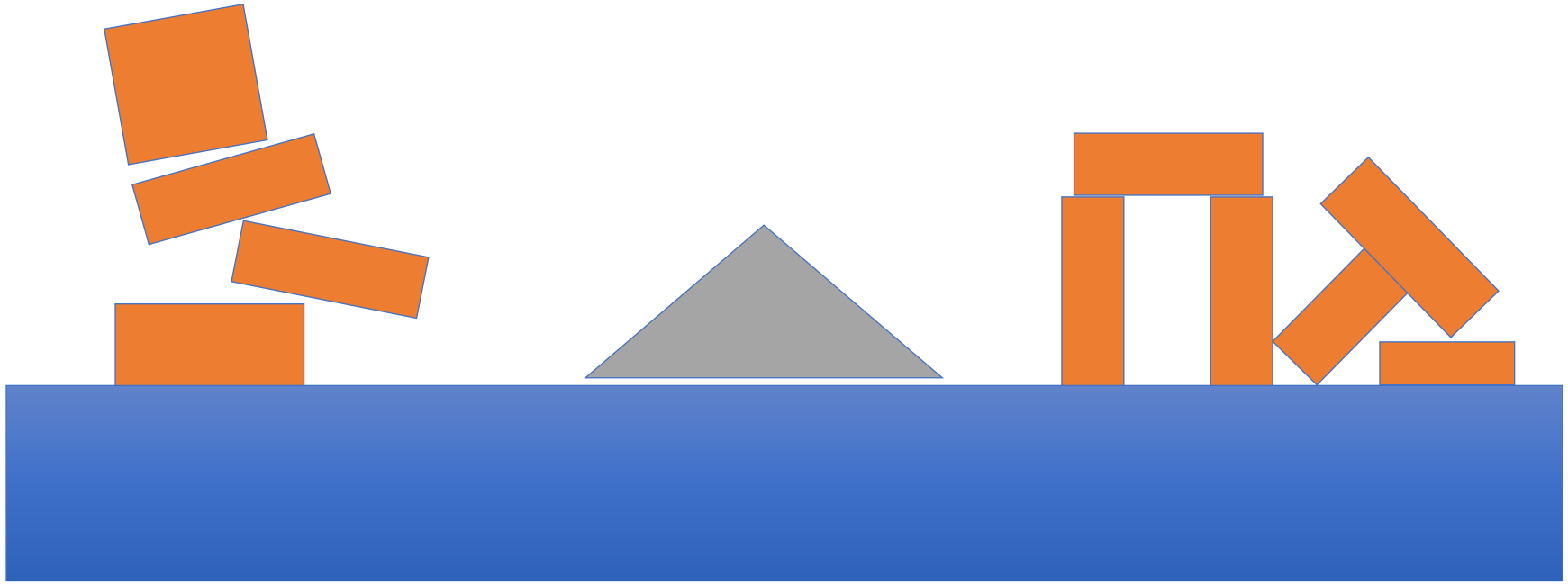


Non-Penetration Constraint

- Apply angular and linear moments
  - Apply an impulse force
  - Or a velocity?
- Iterate across multiple frames until resolved
  - Always resolve the worst first
  - Update which interpenetration is the worst by applying the velocities from the previous iteration
- Leads to *jitter*
  - Never achieve a perfect solution
  - When force < threshold
    - Sleep

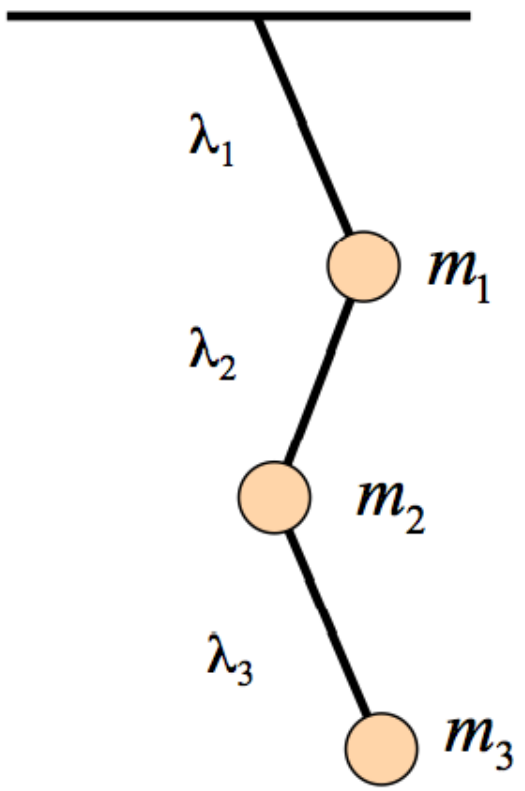


Sleeping



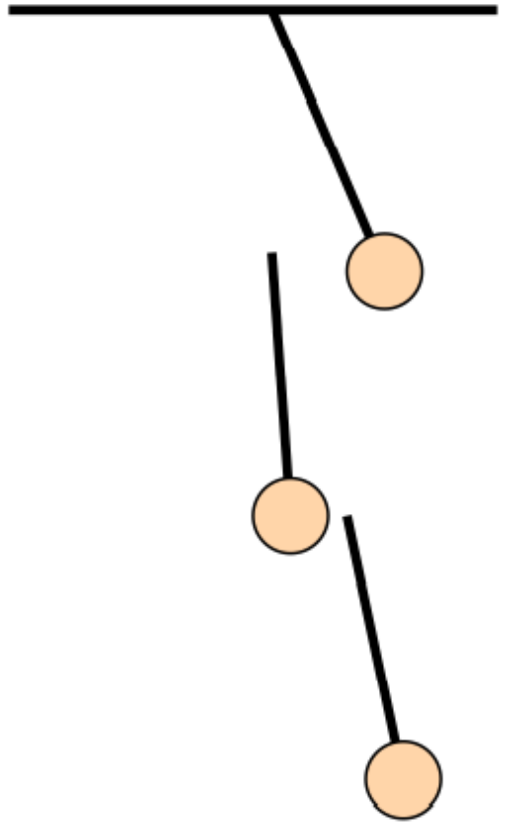
Constraint Solvers

- Solve a collection of constraints
  - Position, velocity
- Global constraint solver
  - Solve for λ1, λ2, λ3 simultaneously
- Iterative constraint solvers
  - Fast
  - Solve for λ1, λ2, λ3 in sequence
  - Apply impulse forces at each constraint
  - Converge to a global solution
- Applications
  - Ropes, chains, cloth
  - Box stacking



Constraint Solvers

- Difficult to implement
  - Errors
    - Causes joints to fall apart
  - Position drift
- Use the physics engine as a black box
  - Engine implements constraint solvers for various *kinds* of joint / constraint



Physics in Practice

- A collection of constraint models
  - Joints
    - Rotational, linear
  - Springs and masses
  - Cloth
    - Connected systems of rigid bodies
  - Ragdolls
    - Avatars constructed from jointed collections of capsules
  - Soft-body physics
- Reduce problem to a collection of simple rigid-body primitives
  - Capsules, cubes, spheres
  - Abstraction, limited range of reasonable values, shapes
- How to integrate with animation?
  - Kinematics meets linear dynamics
    - Inverse kinematics
  - Simple capsule constructs as player controllers
    - Change to ragdoll on “death”


fps: 66 ping: 56 ms
in : 61 1.53 k/s 17.3/s
out: 66 1.44 k/s 18.2/s
30/s
30/s
77
0
$ 0
30 | 53
5.56

References

- Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game. Ian Millington (2010)

- Game Physics. David Eberly (2010)