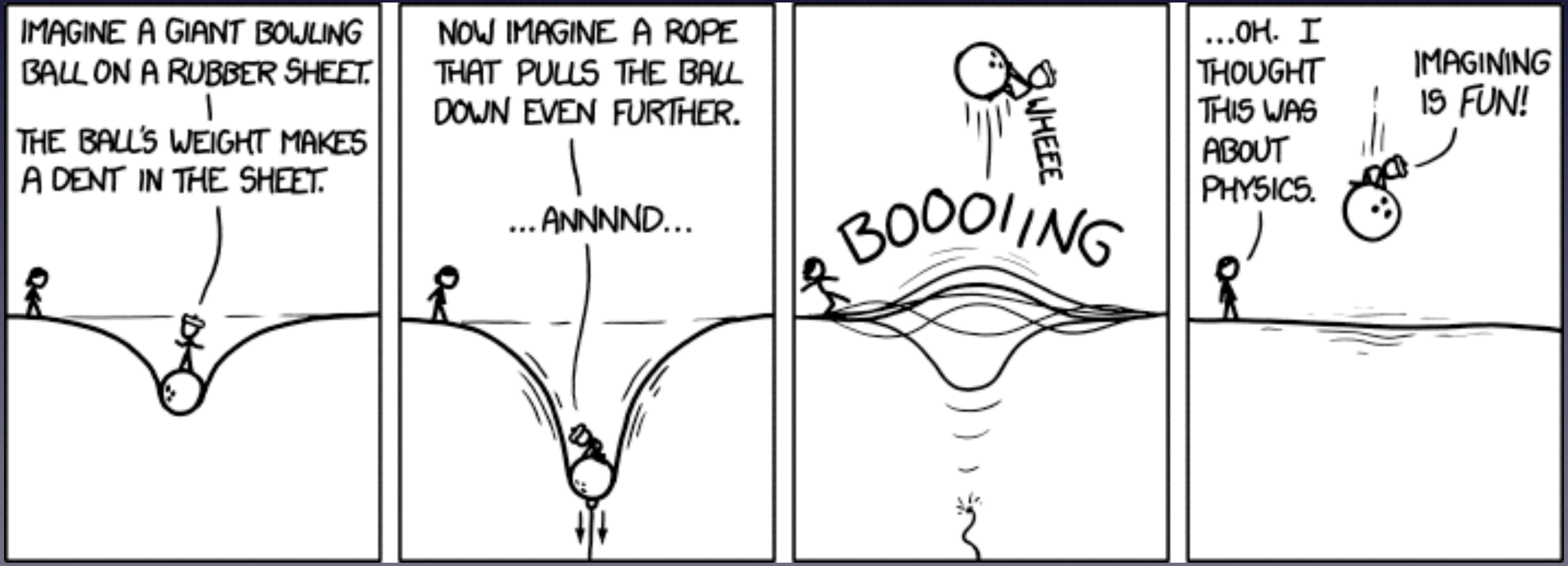
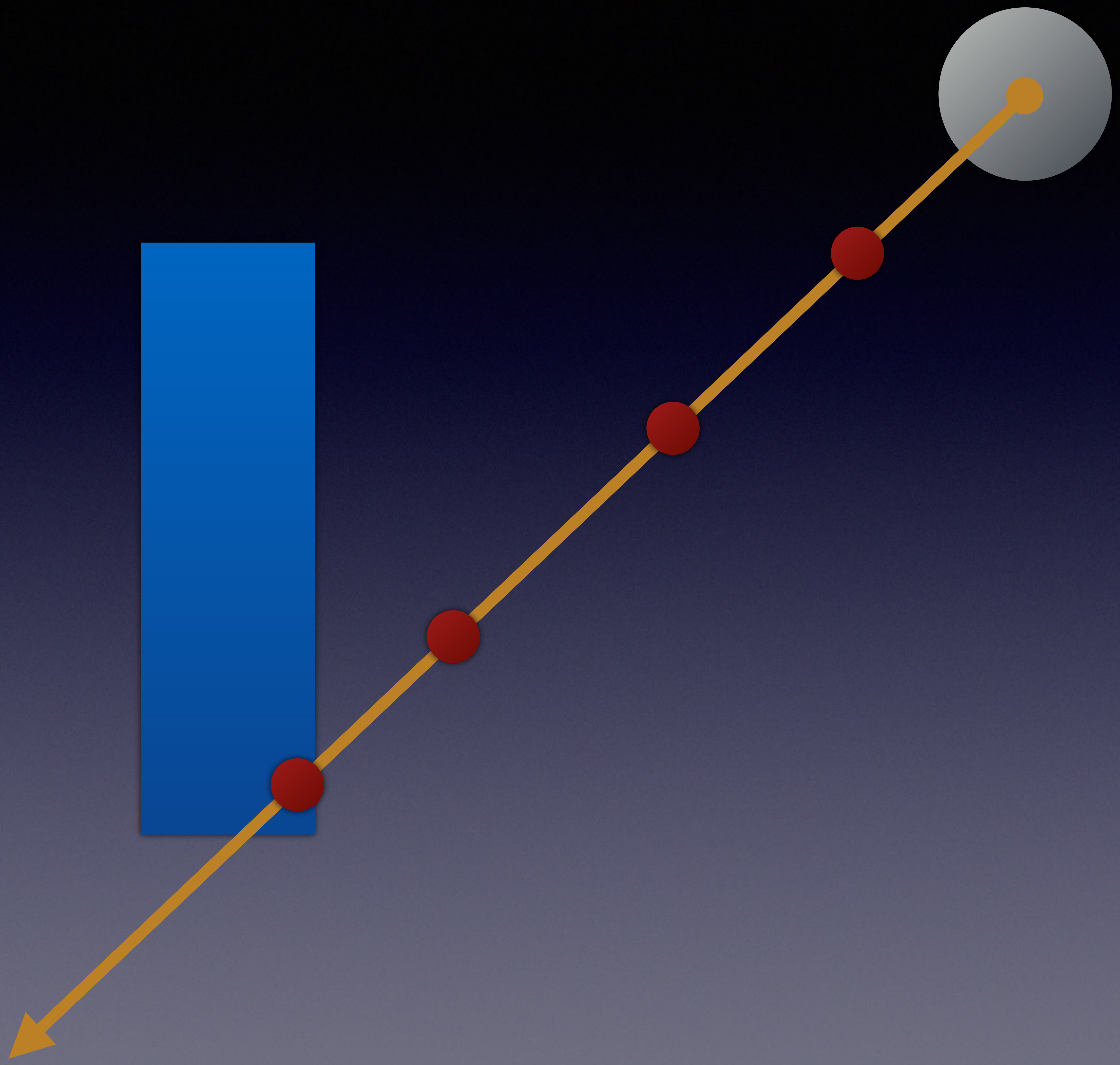
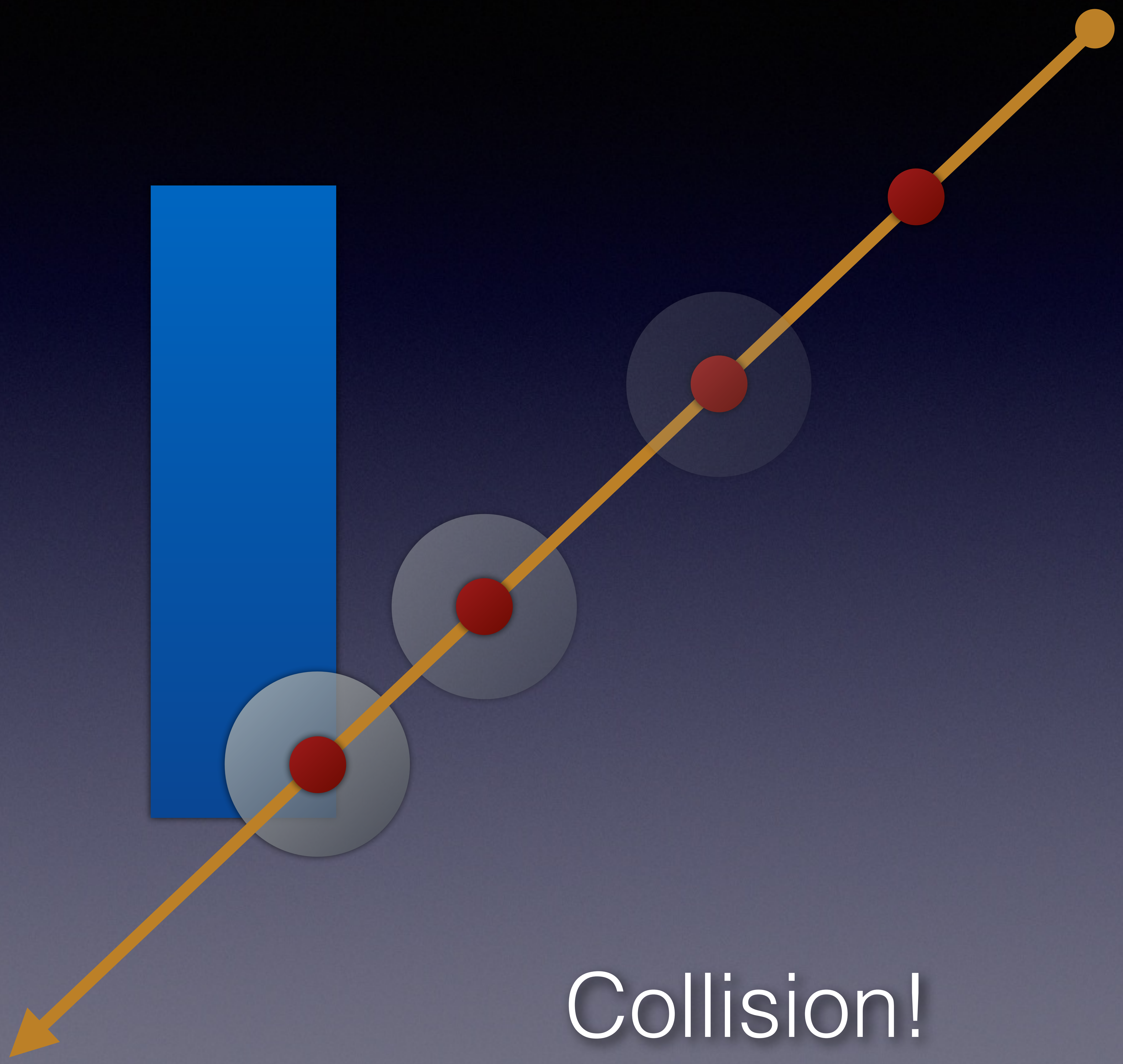
Basic physics
and
collision detection.


IMAGINE A GIANT BOWLING BALL ON A RUBBER SHEET.
THE BALL'S WEIGHT MAKES A DENT IN THE SHEET.
NOW IMAGINE A ROPE THAT PULLS THE BALL DOWN EVEN FURTHER.
...ANNNND...
WHEEE
BOOOIING
...OH. I THOUGHT THIS WAS ABOUT PHYSICS.
IMAGINING IS FUN!

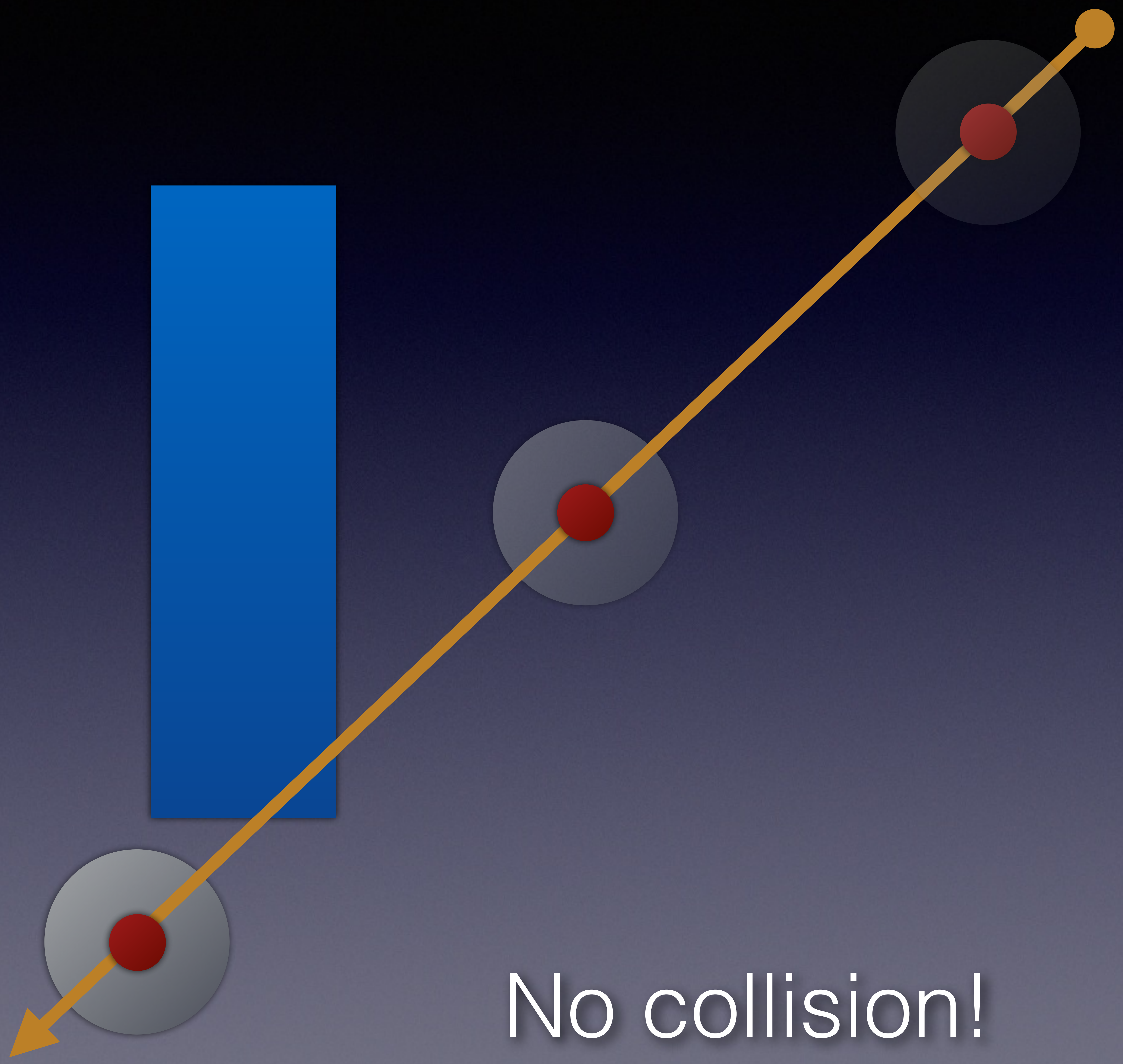
Fixing the timestep.

Problems with variable timestep.



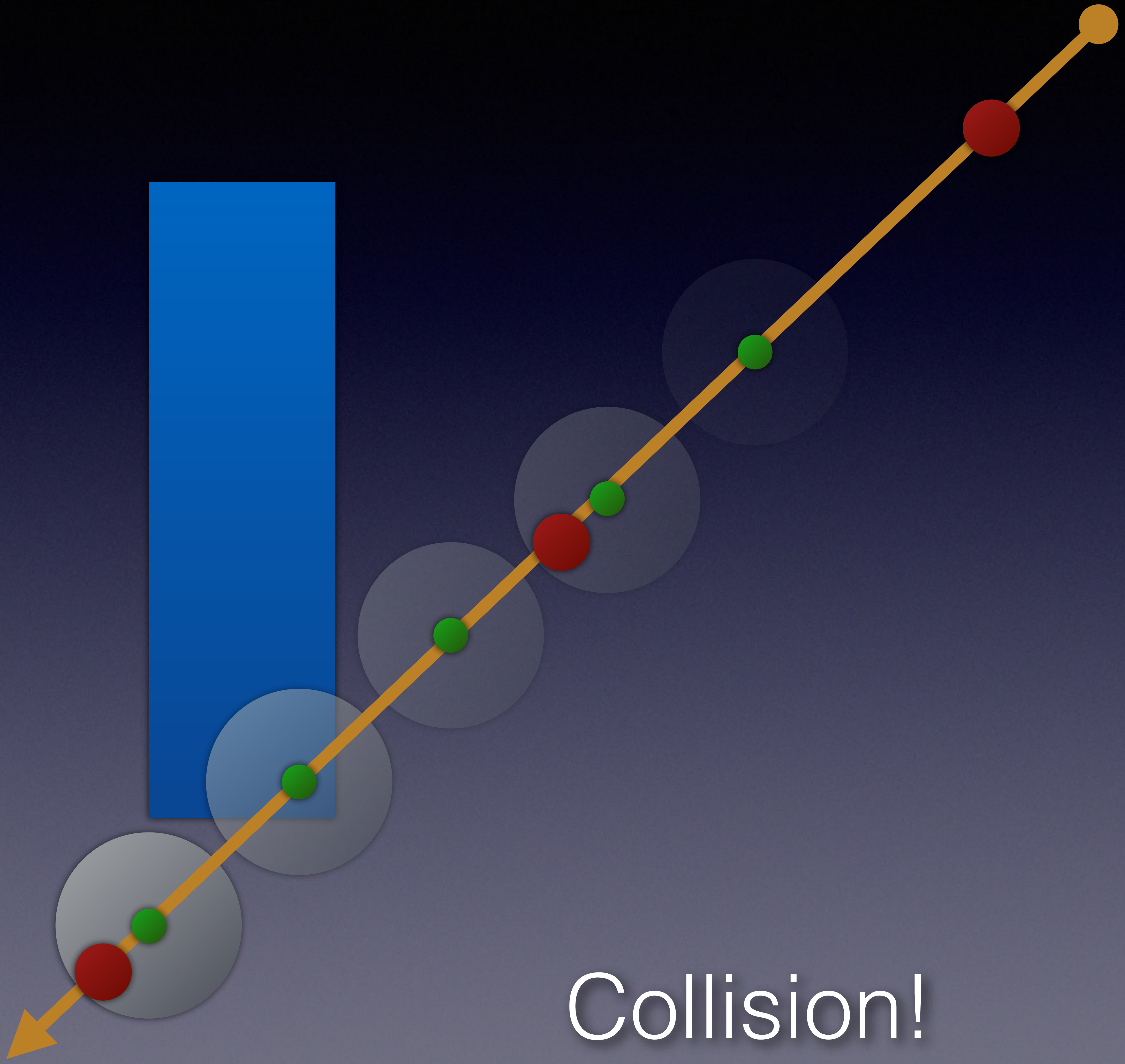


Collision!



No collision!

Fixed timestep.



Collision!

```
// 60 FPS (1.0f/60.0f)
#define FIXED_TIMESTEP 0.0166666f
#define MAX_TIMESTEPS 6
float timeLeftOver = 0.0f;
```

Fixed timestep loop

```
float fixedElapsed = elapsed + timeLeftOver;
if(fixedElapsed > FIXED_TIMESTEP * MAX_TIMESTEPS) {
    fixedElapsed = FIXED_TIMESTEP * MAX_TIMESTEPS;
}

while (fixedElapsed >= FIXED_TIMESTEP ) {
    fixedElapsed -= FIXED_TIMESTEP;
    FixedUpdate();
}
timeLeftOver = fixedElapsed;

Update(elapsed);
```

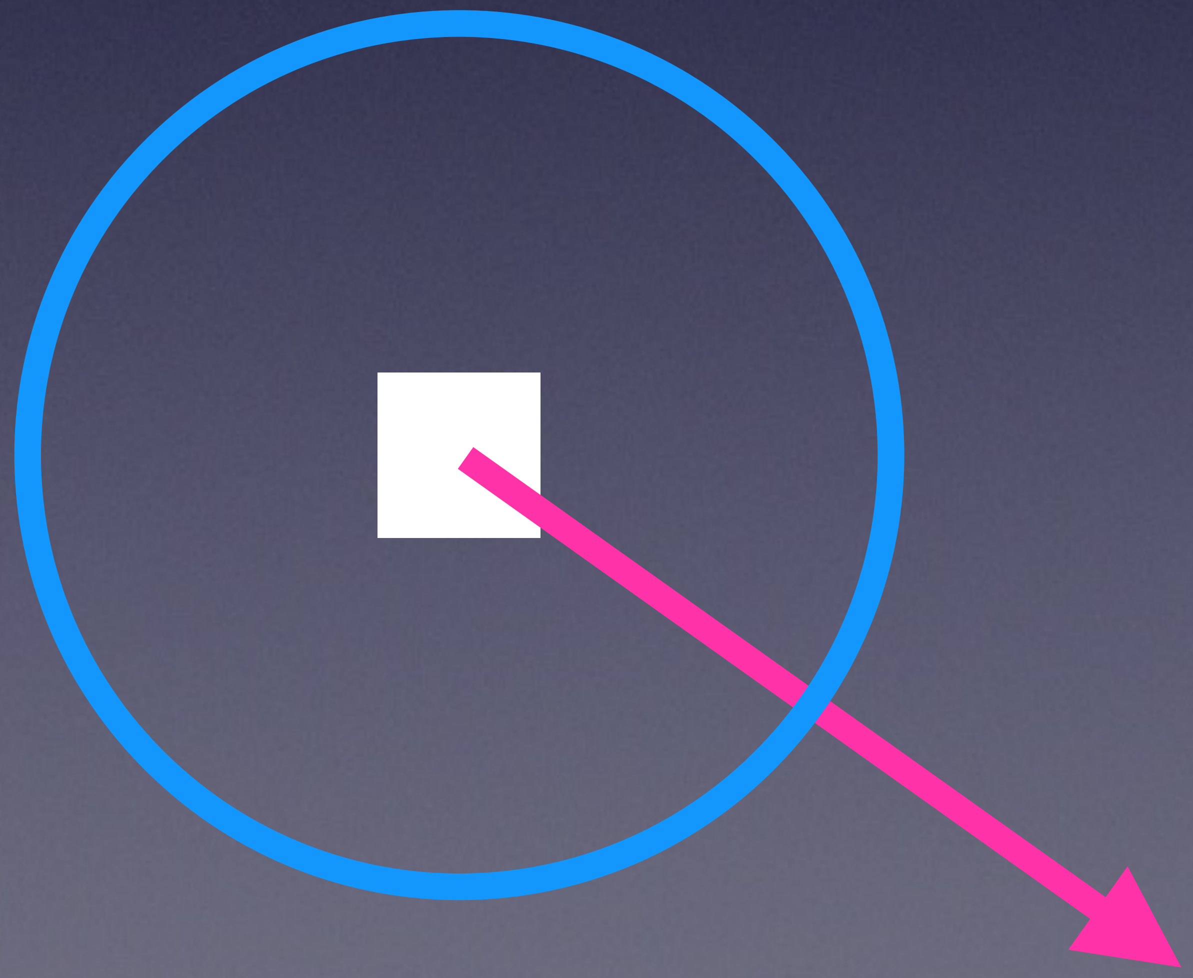
Basic game physics

Velocity and acceleration.

Velocity.

The rate of change of the position of an object.

(Direction * speed)



```
x += dir_x * speed * elapsed;
y += dir_y * speed * elapsed;
```

```
x += velocity_x * elapsed;
y += velocity_y * elapsed;
```

Acceleration.

The rate of change of velocity.

```
velocity_x += acceleration_x * elapsed;
velocity_y += acceleration_y * elapsed;
```

Friction (deceleration).

The rate of decrease of velocity.

```
    velocity_x = lerp(velocity_x, 0.0f, elapsed * friction_x);
    velocity_y = lerp(velocity_y, 0.0f, elapsed * friction_y);
```

lerp???

LERP!

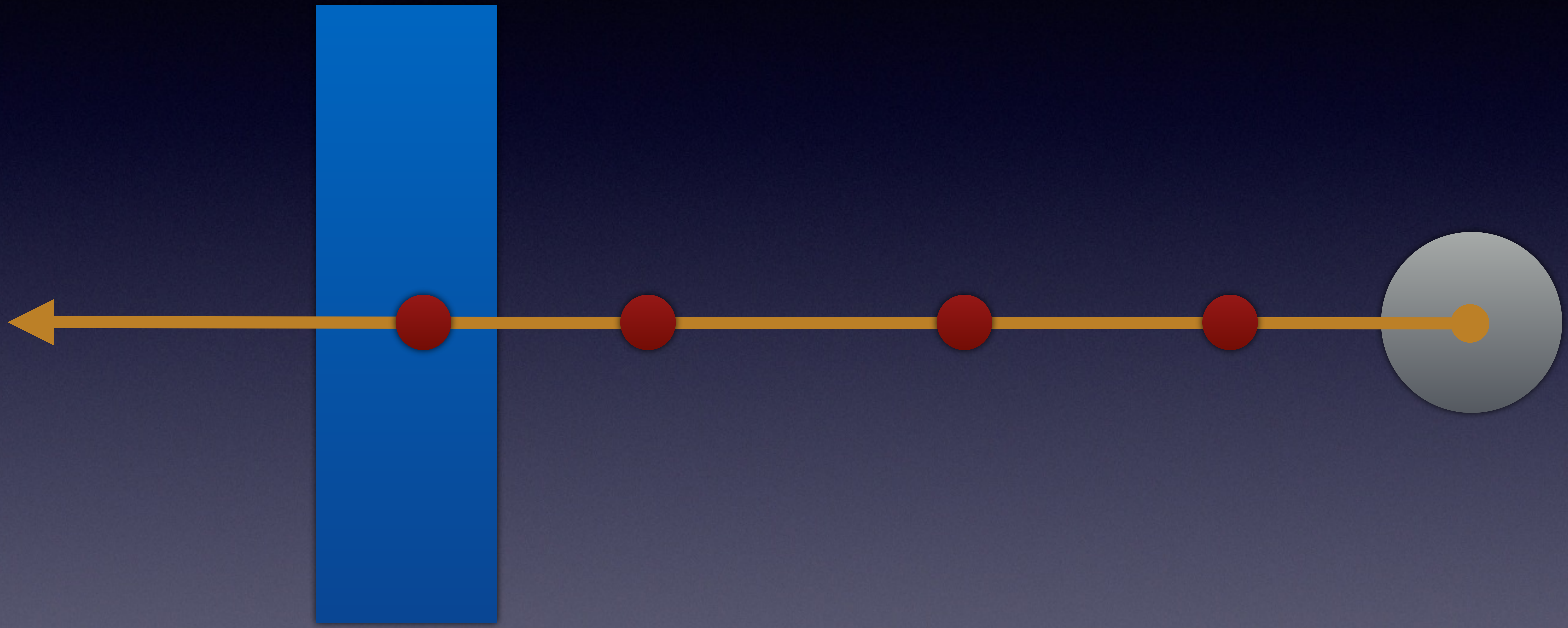
LinEar InteRPolation

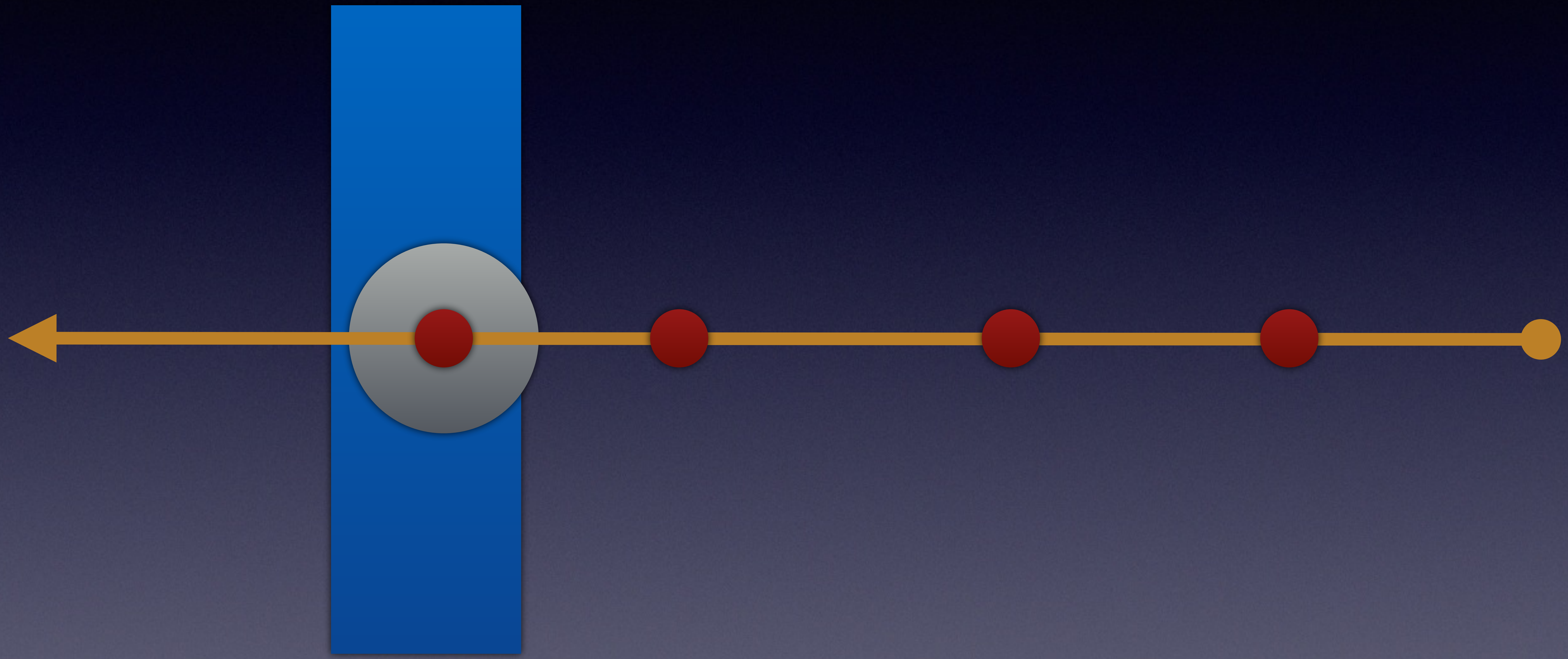
```
float lerp(float v0, float v1, float t) {
    return (1.0-t)*v0 + t*v1;
}
```

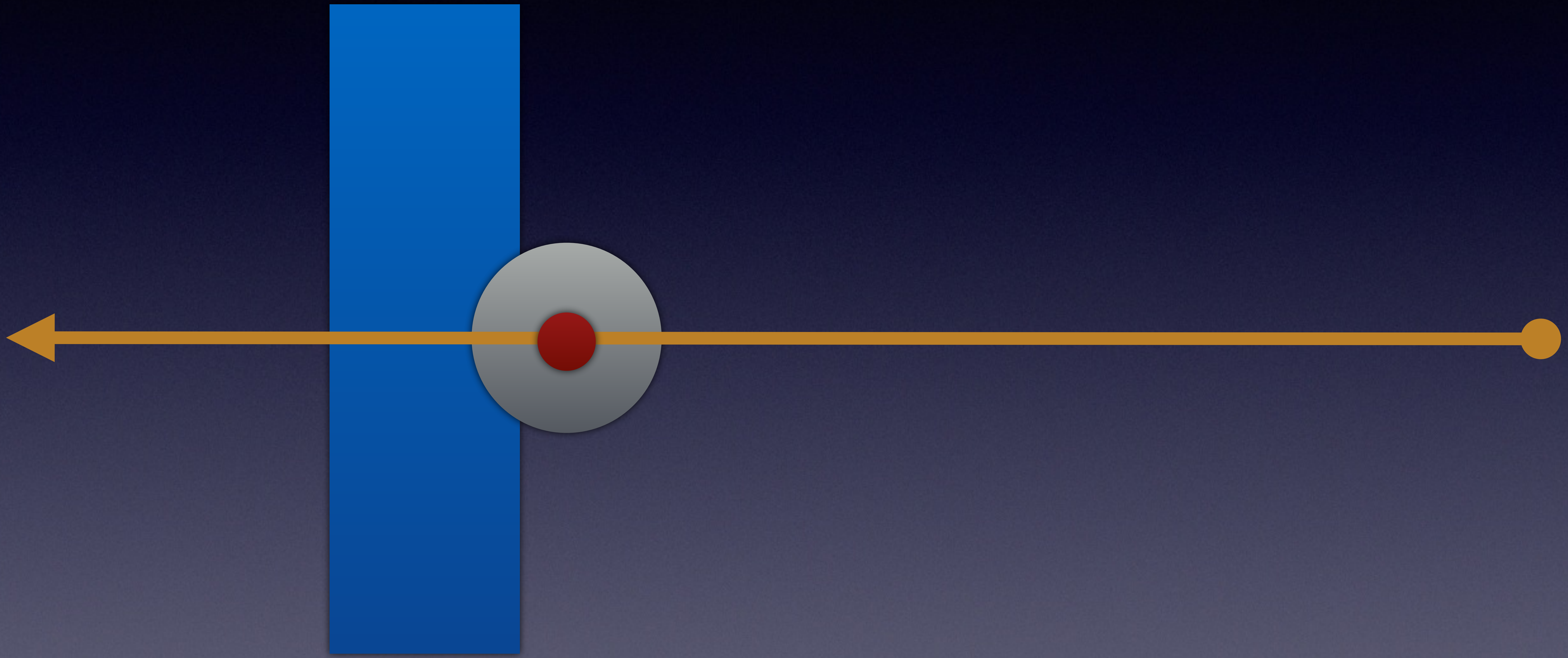
Combined movement.

```
velocity_x = lerp(velocity_x, 0.0f, FIXED_TIMESTEP * friction_x);
velocity_y = lerp(velocity_y, 0.0f, FIXED_TIMESTEP * friction_y);

velocity_x += acceleration_x * FIXED_TIMESTEP;
velocity_y += acceleration_y * FIXED_TIMESTEP;

x += velocity_x * FIXED_TIMESTEP;
y += velocity_y * FIXED_TIMESTEP;
```

Collision response.

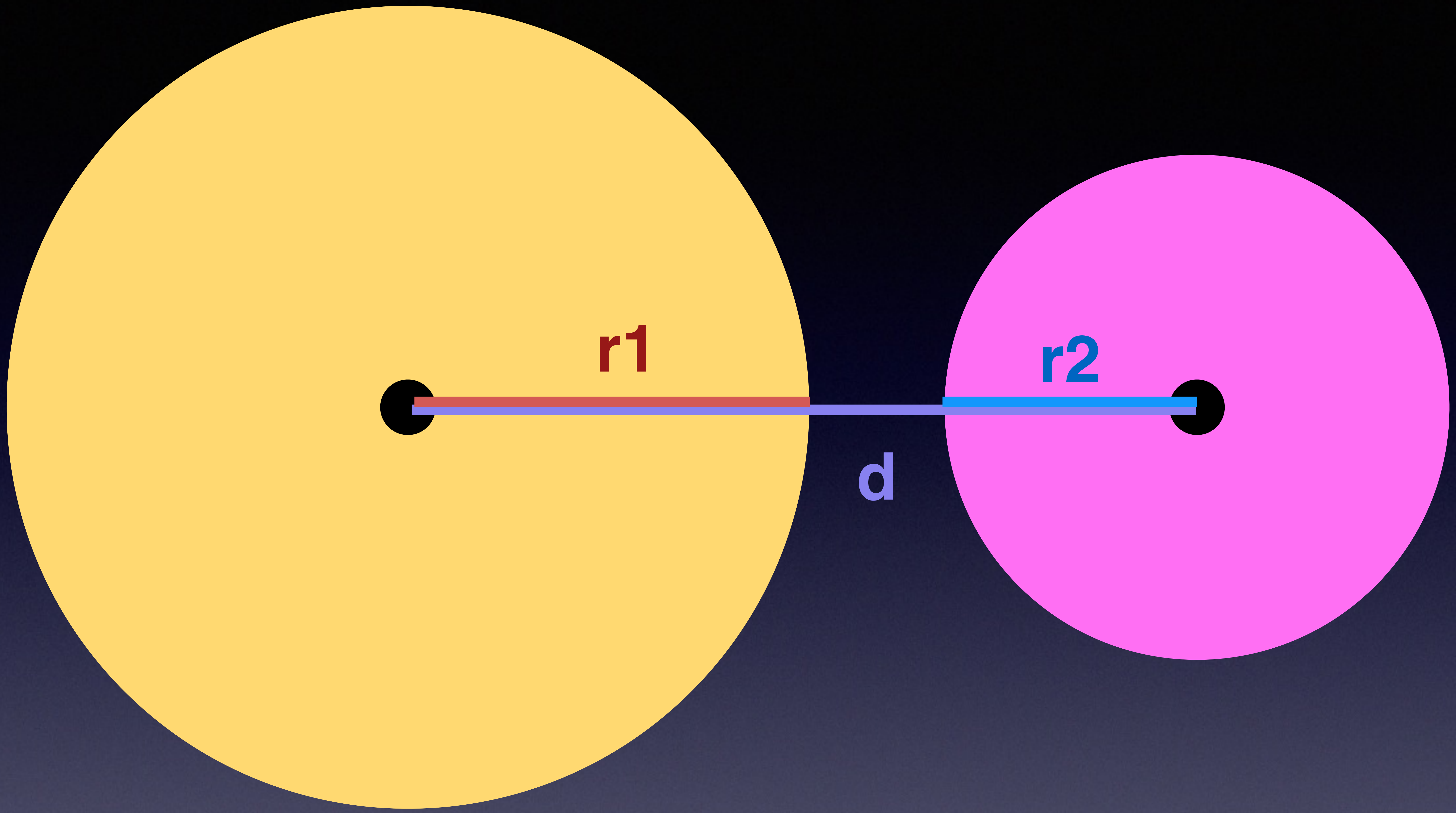




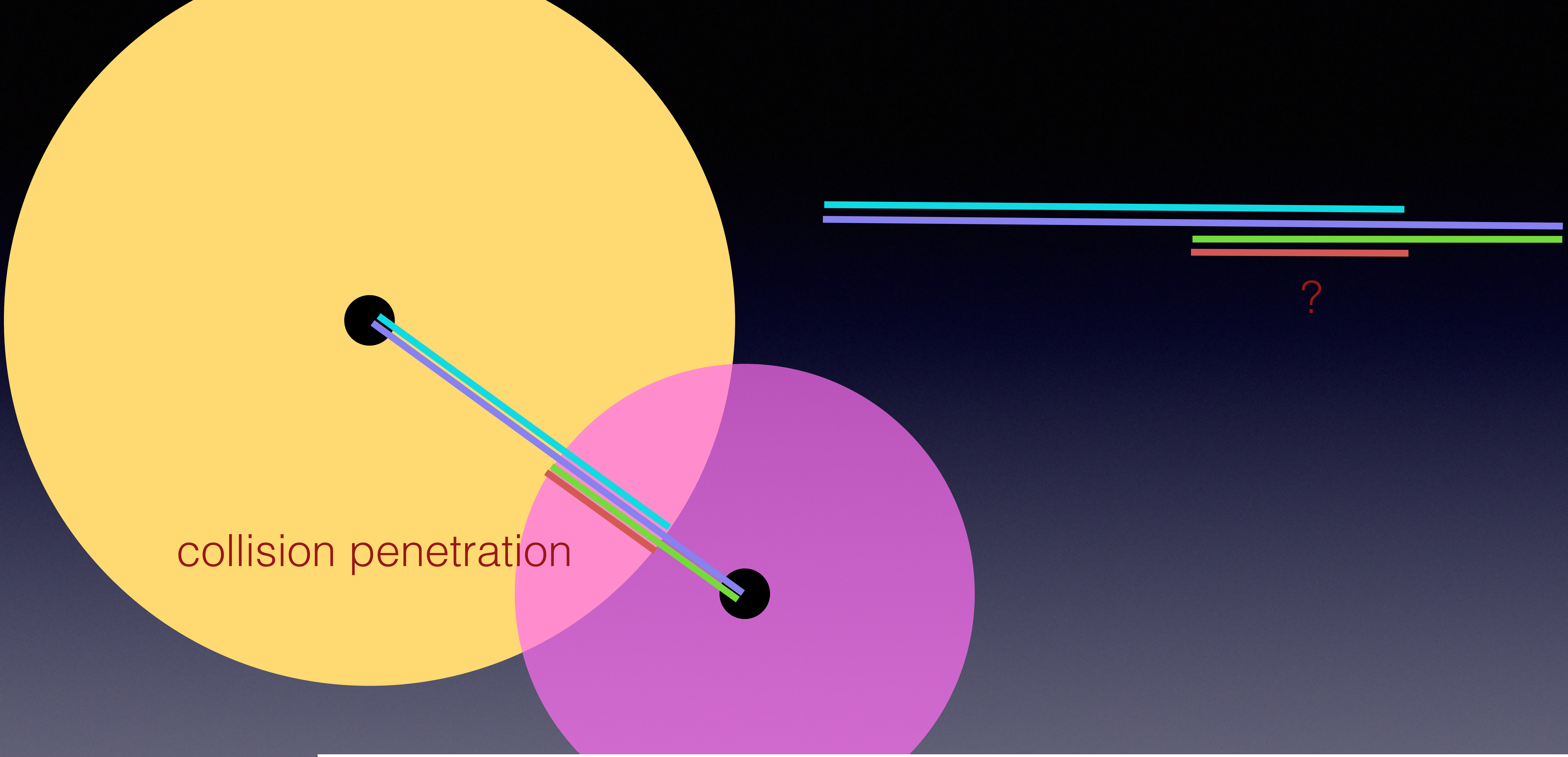


Calculating collision penetration.

Circle/circle collision.



If the distance between two circles is less than or equal to the sum of their radii, the circles are colliding!



```
penetration = fabs(distance - radius1 - radius2)
```

collision penetration
?

```
circle2.position -= penetration + offset
```

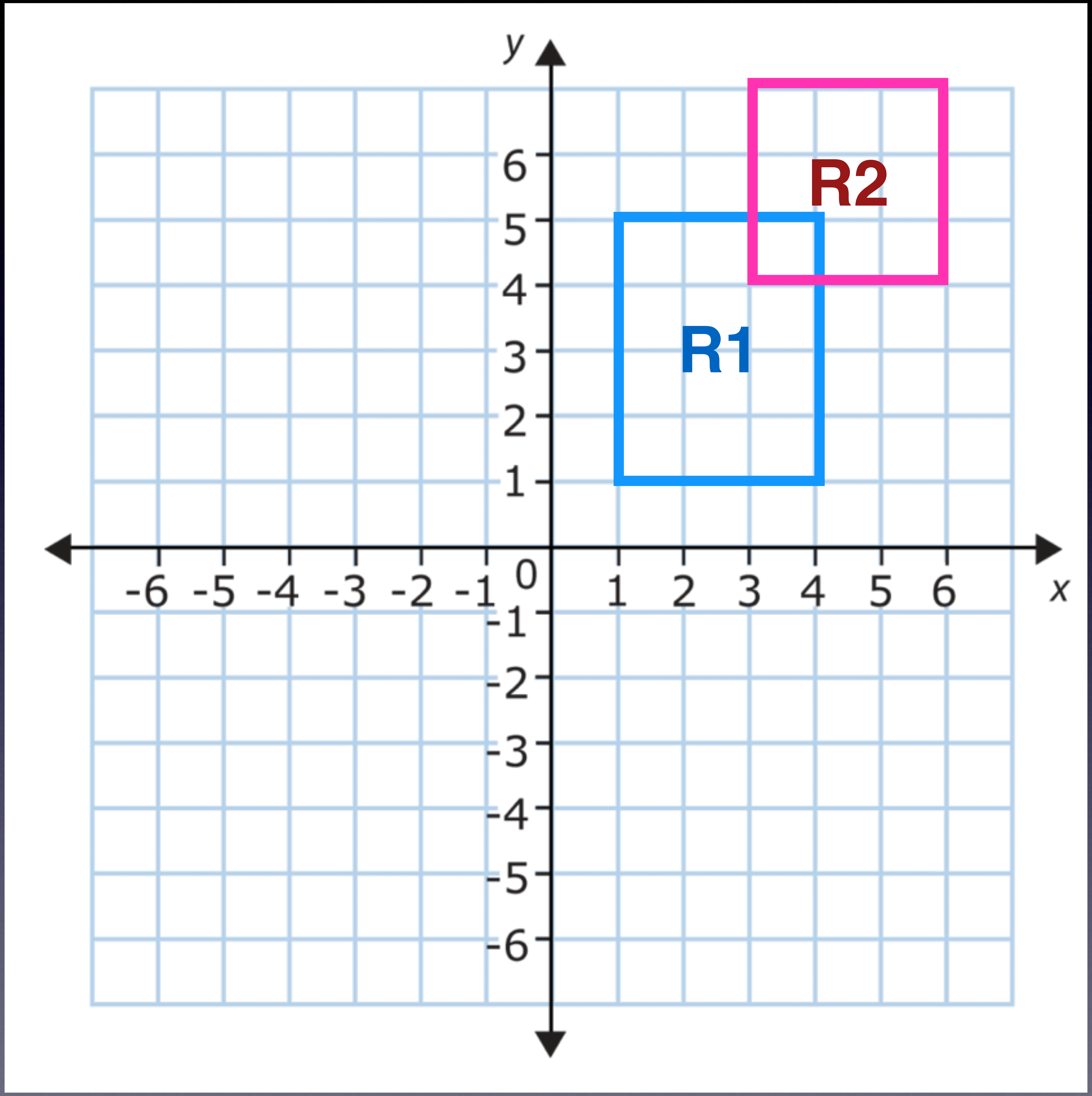
Box/box collision.

a) is R1's bottom higher than R2's top?
b) is R1's top lower than R2's bottom?
c) is R1's left larger than R2's right?
d) is R1's right smaller than R2's left

If ANY of the above are true, then the two rectangles are not intersecting!

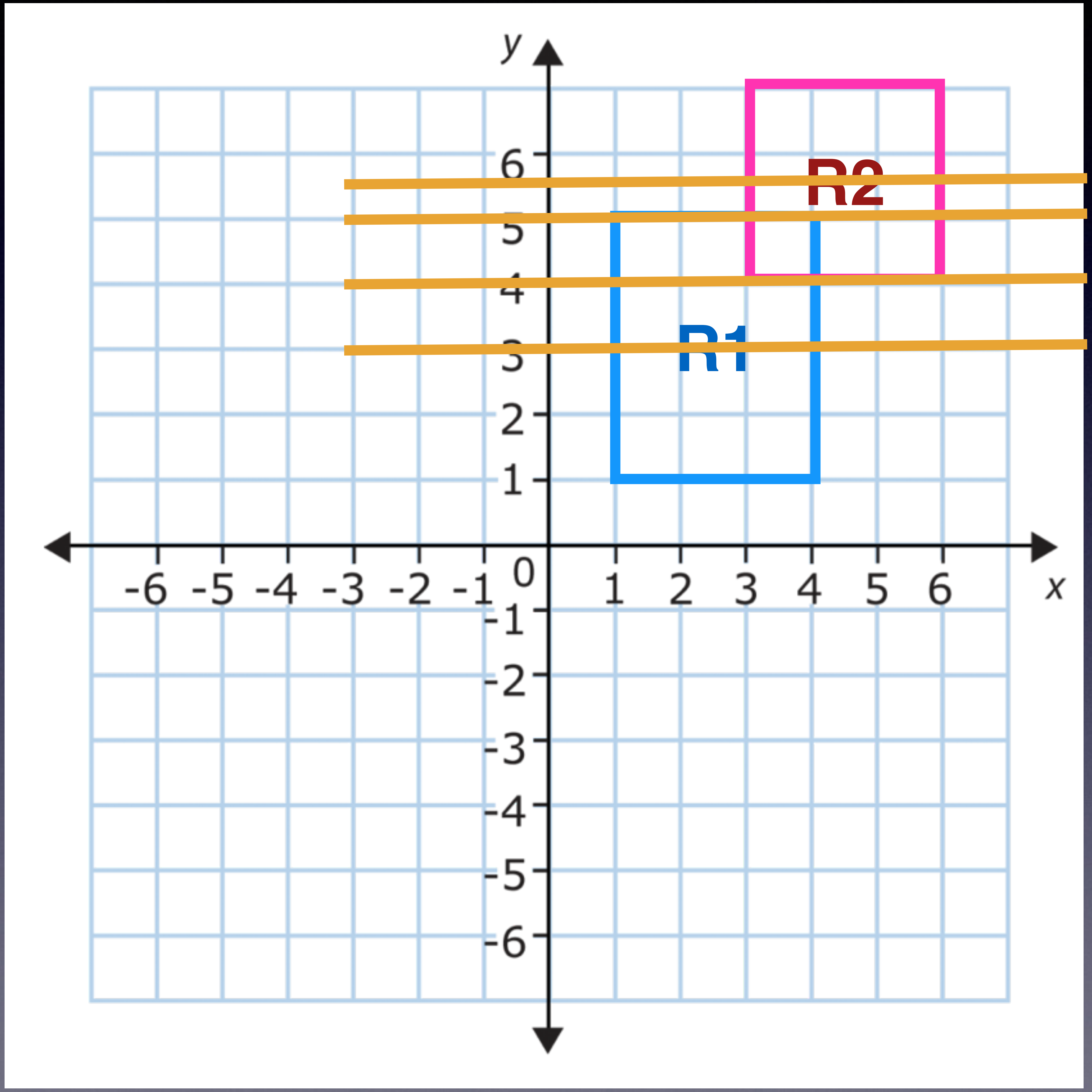
OR

The rectangles are intersecting if NONE of the above are true.



Separate collision on each axis.

Step1.



Move entity on the Y axis only.

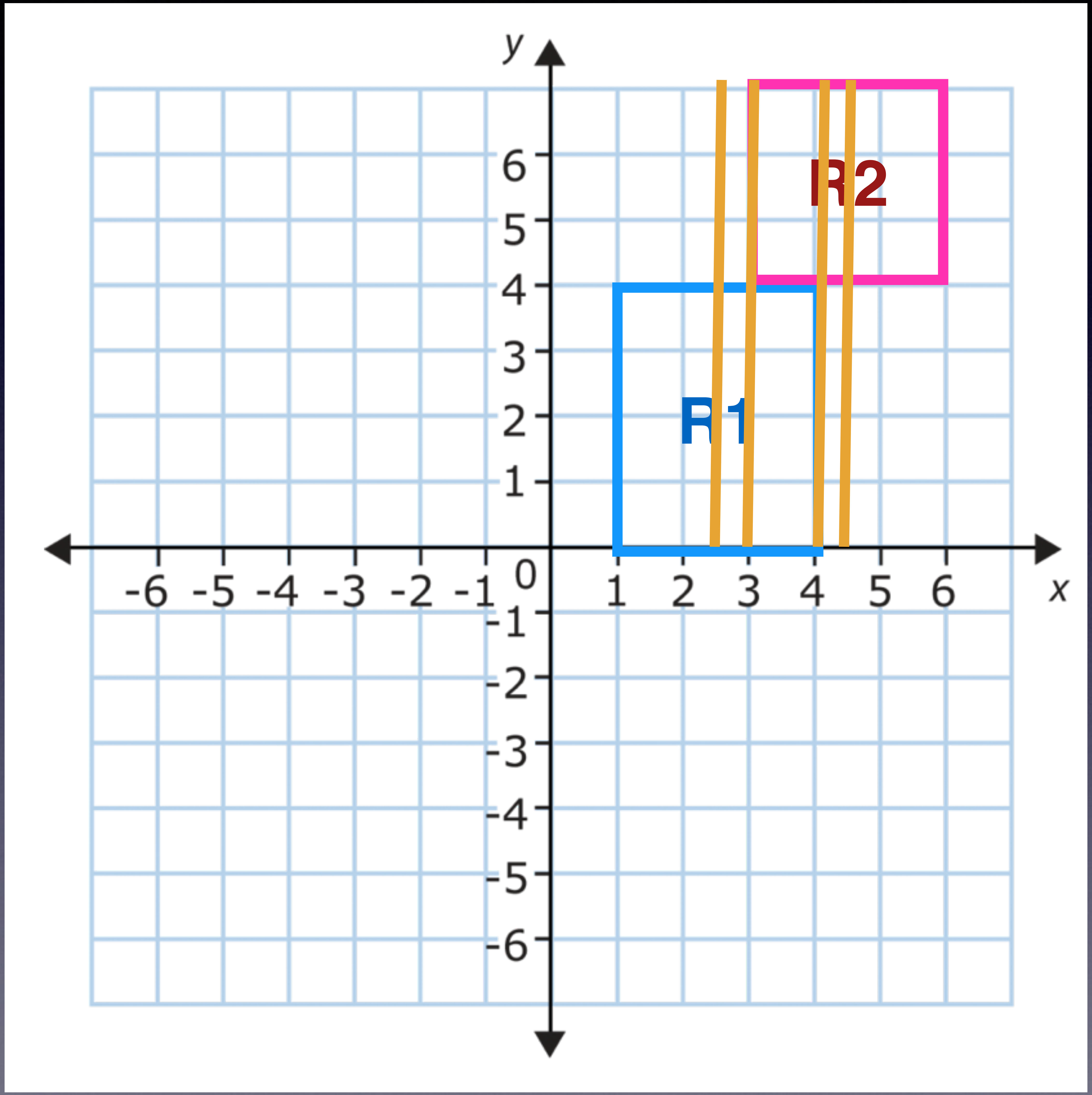
Check against all collidable entities.

If collision occurs, check the Y-axis penetration.

y_penetration = fabs(y_distance - R1_height/2 - R2_height/2)

Adjust the entity's Y position based on penetration

Step2.



Move entity on the X axis only.

Check against all collidable entities.

If collision occurs, check the X-axis penetration.

x_penetration = fabs(x_distance - R1_width/2 - R2_width/2)

Adjust the entity's X position based on penetration

Gravity.

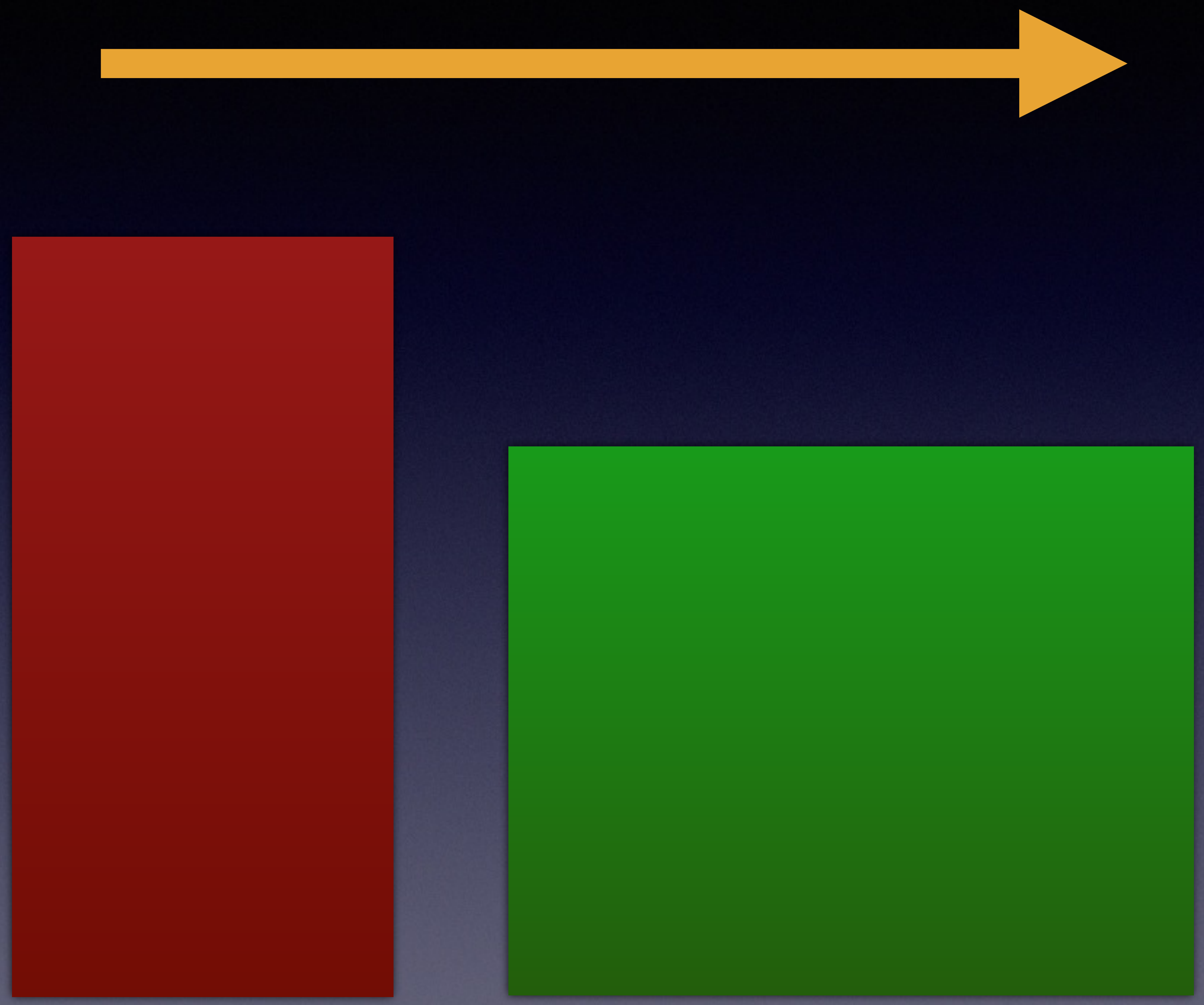
Gravity.

A constant acceleration.

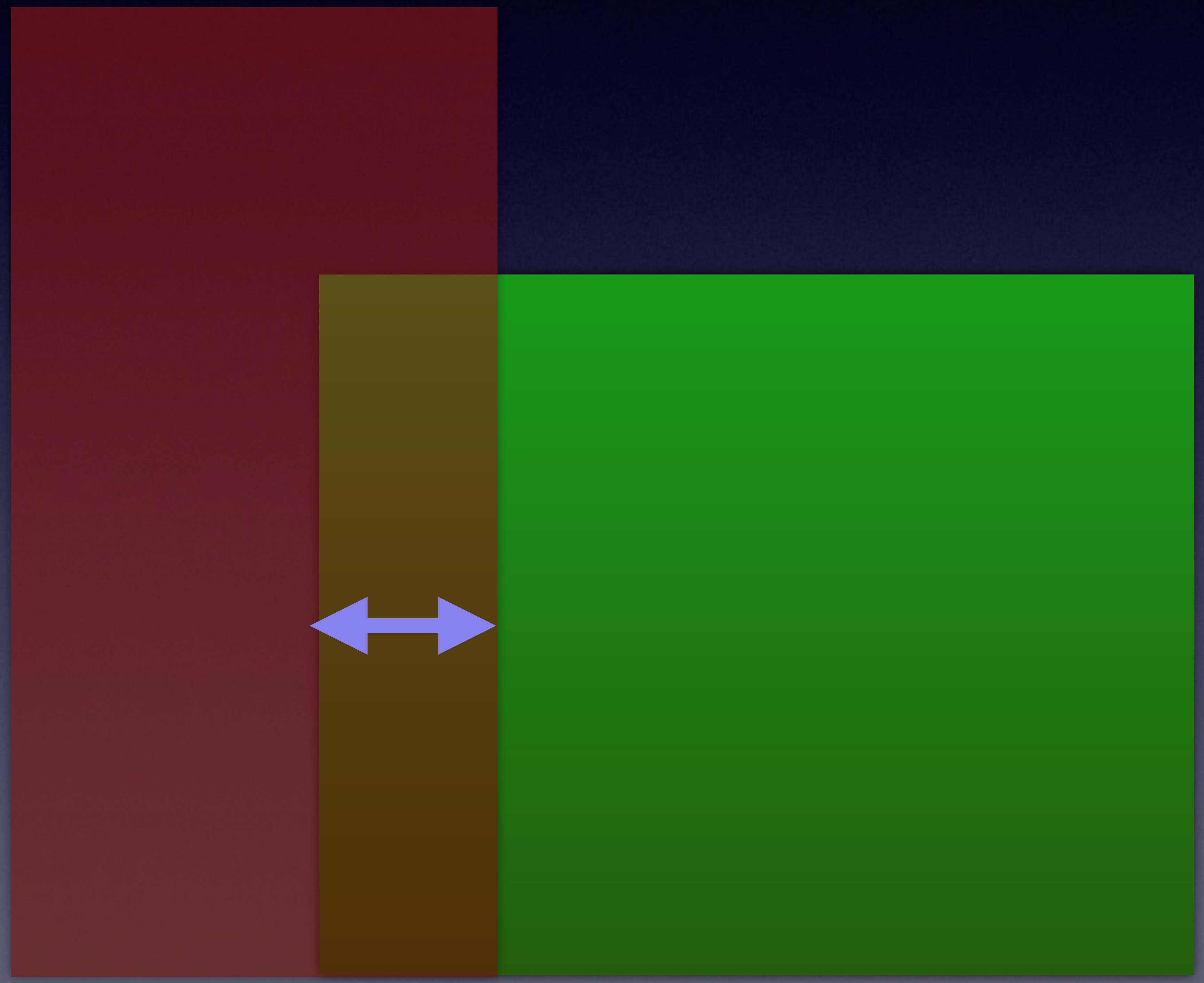
```
velocity_x += gravity_x * elapsed;
velocity_y += gravity_y * elapsed;
```

Apply friction based on collisions.

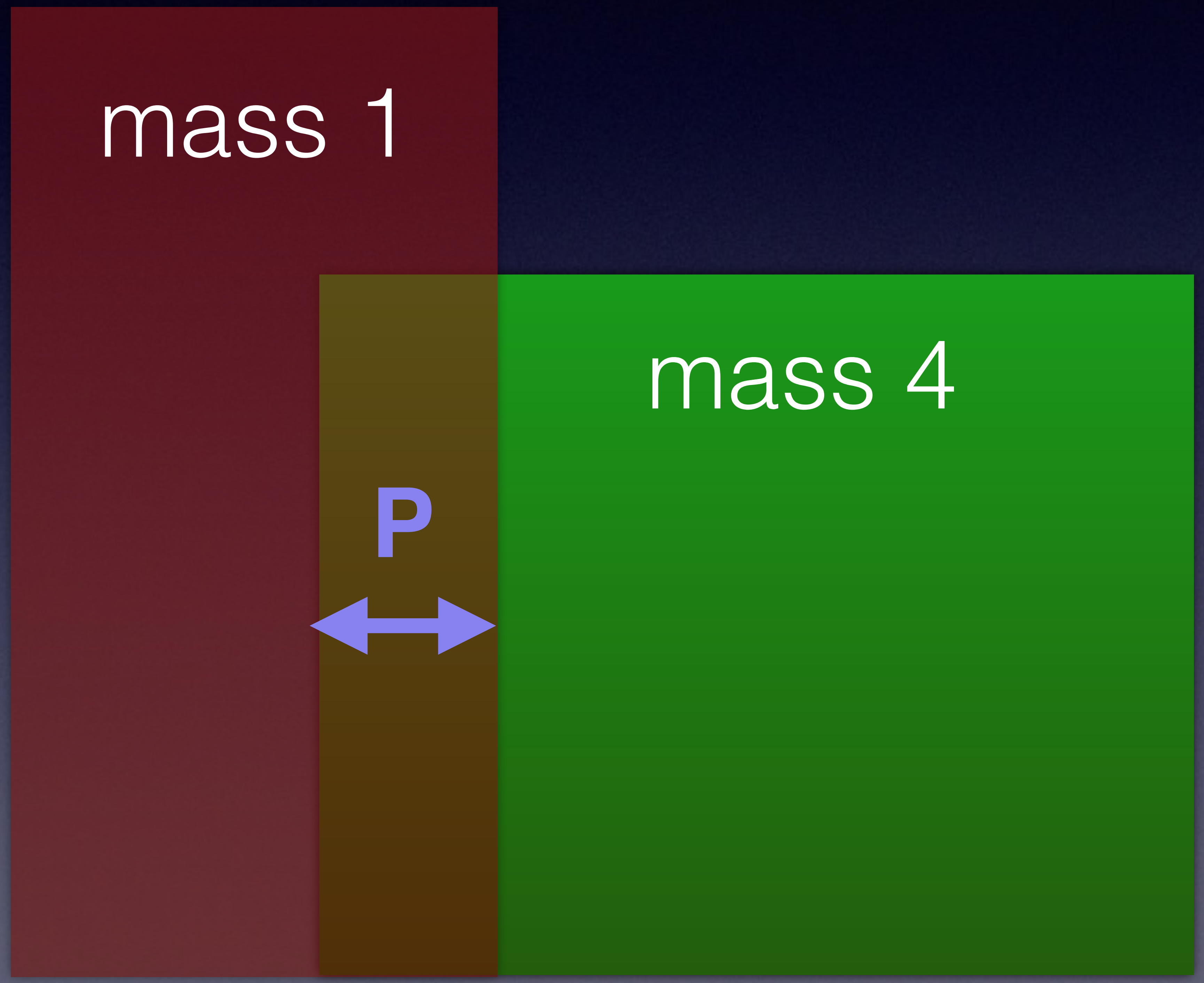
Mass and basic forces.



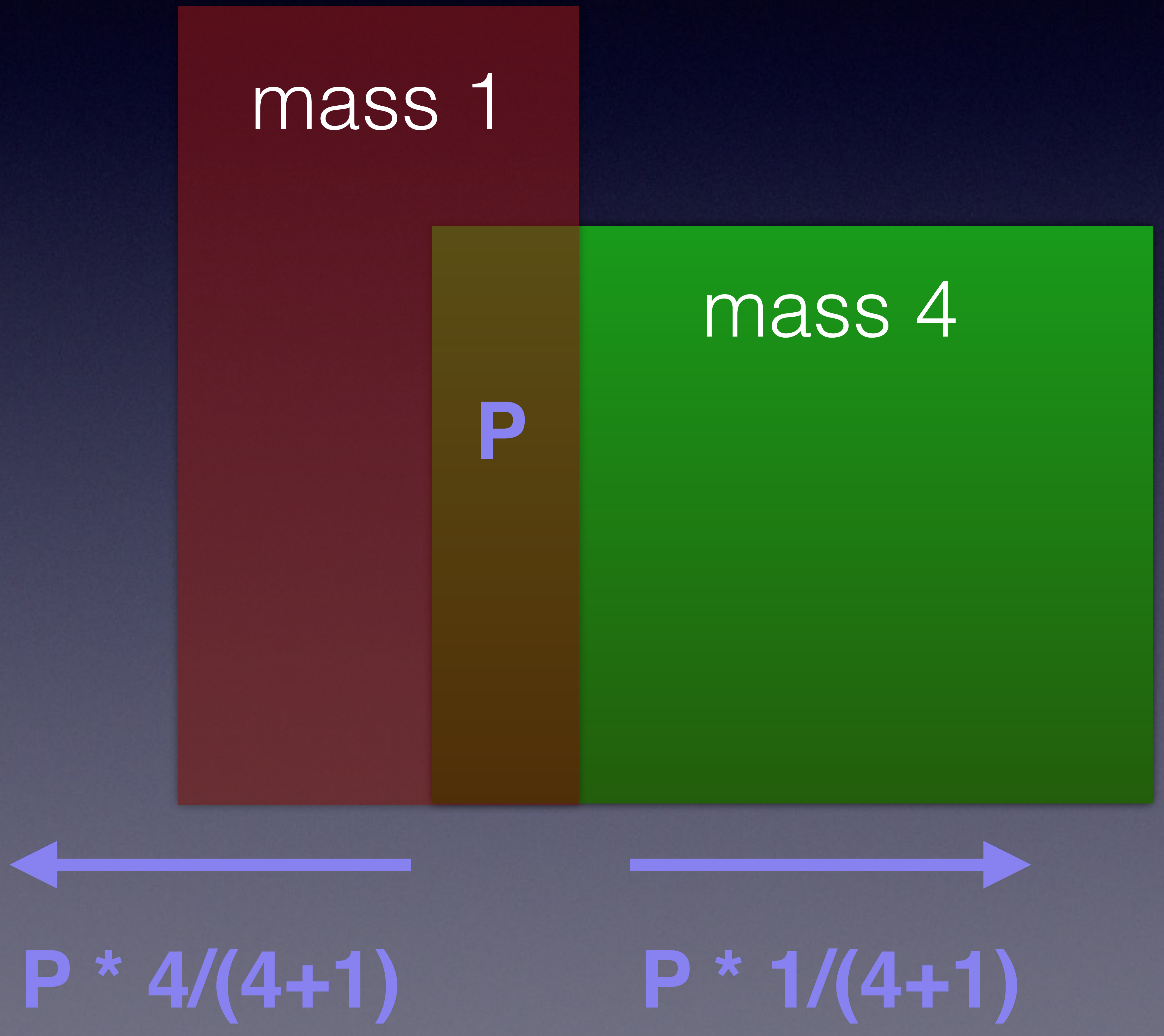
Which block moves?



Move each object away based on the ratio of their mass to the sum of their masses.



Move each object away based on the ratio of their mass to the sum of their masses.



Heavier mass also means more friction!