7 Programming with Animations

7.1 Animated Graphics: Principles and History

7.2 Types of Animation

7.3 Programming Animations: Interpolation

7.4 Design of Animations

- Principles of Animation
- Vector and Bitmap Graphics
- Creating a Game Character

7.5 Game Physics

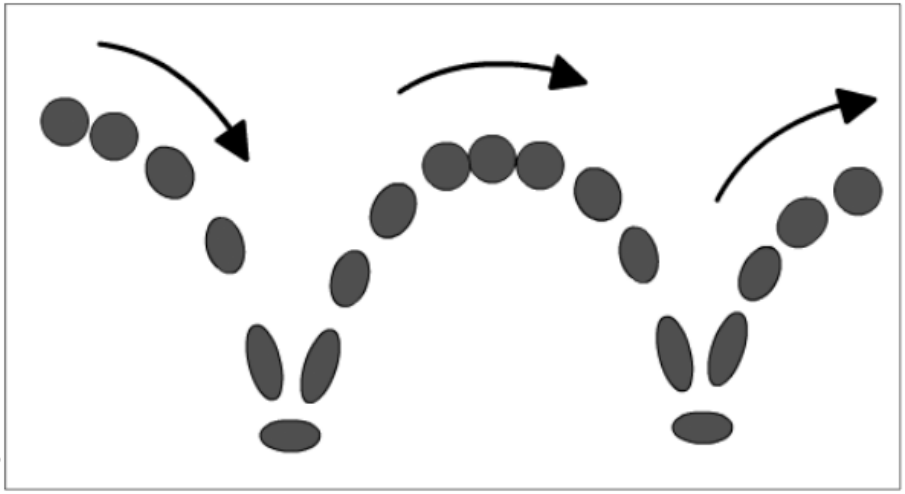
Literature:

K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004
– Section 7.4 based on book chapter 2 by ***Brad Ferguson***

T. Jones et al.: Foundation Flash Cartoon Animation,
Apress/Friends of ED 2007

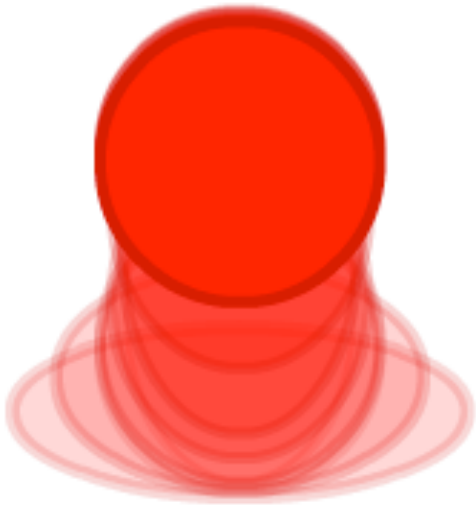
Principles of (2D-)Animation

- Squash and Stretch
  - Shape of subject reacts to speed and force of movement
- Timing
  - E.g. ease-in and ease-out: Gives animation a sense of weight and gravity
- Anticipation, Action, Reaction, Overlapping Action
  - Anticipation: Build up energy before a movement
  - Reaction: Don't simply stop, but show the process of stopping
  - Overlapping: Hierarchy of connected objects moves in a complex way
- Arcs
  - Every object follows a smooth arc of movement

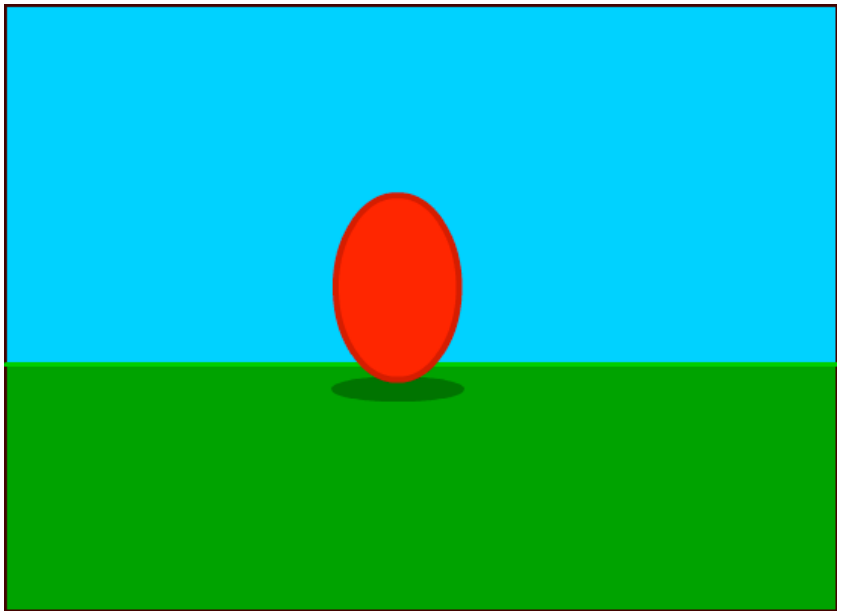


Bouncing ball
Source: T. Jones

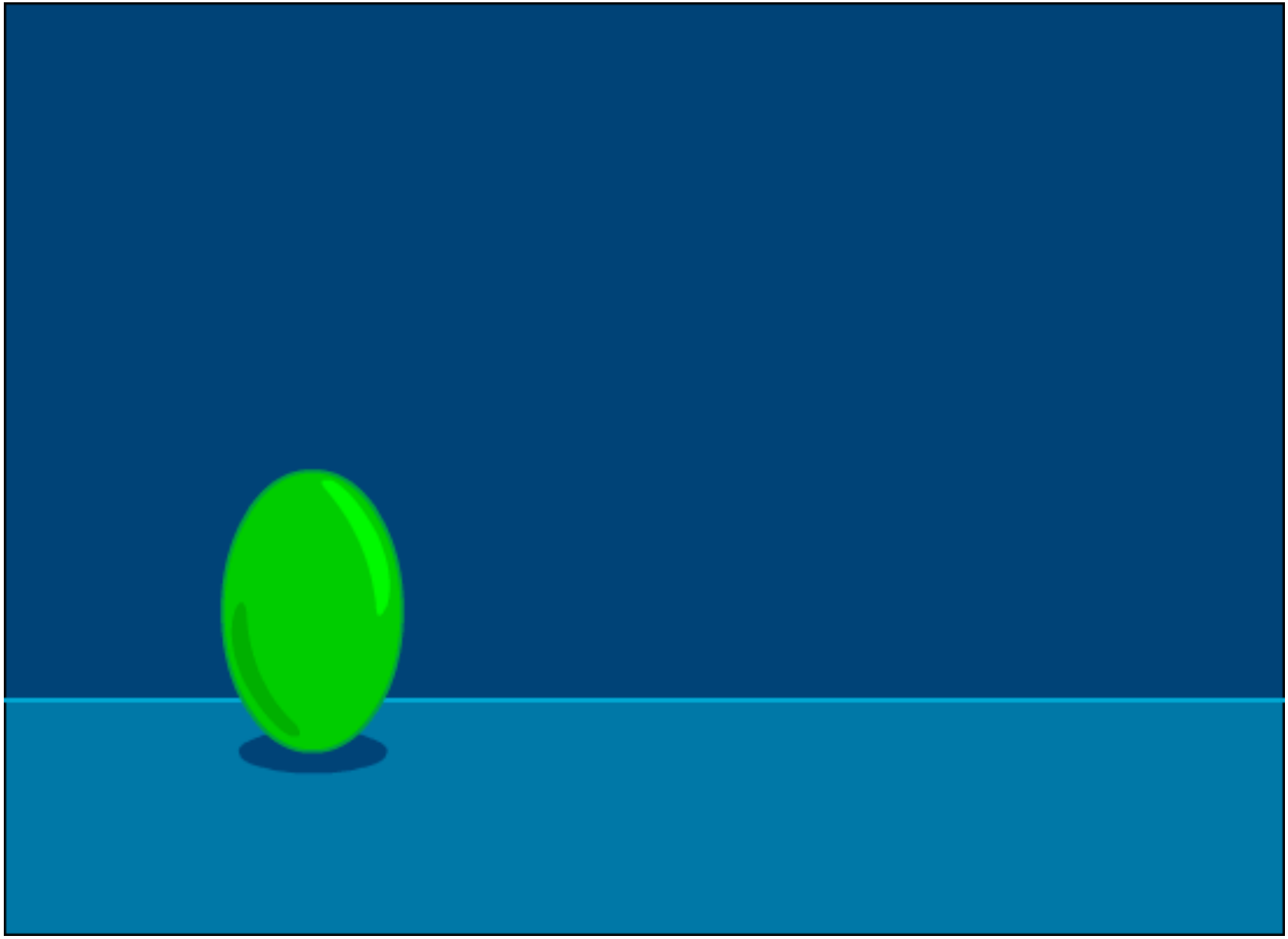
Animating a Bouncing Ball



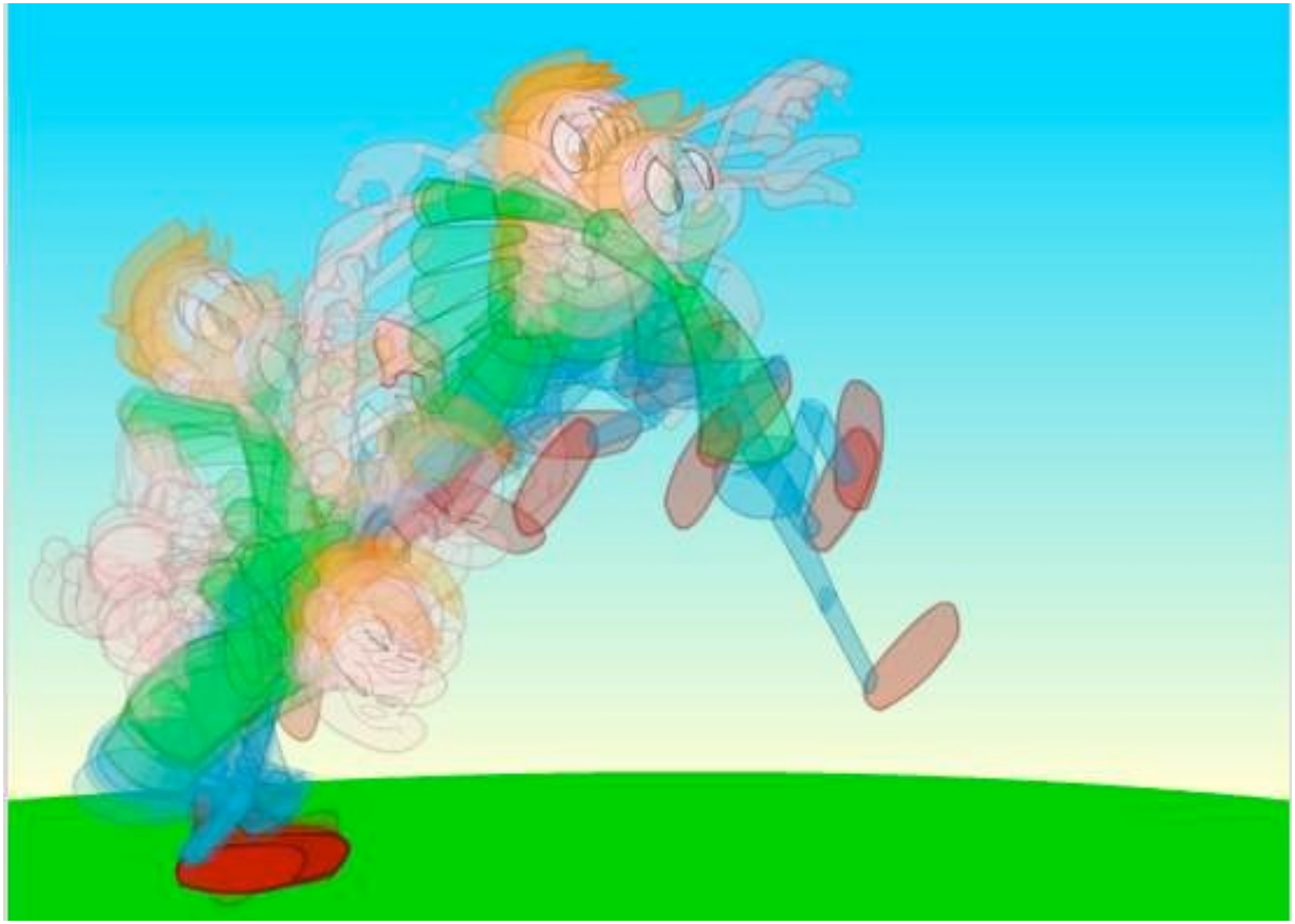
- Going up slower than when falling
- On rise and fall, ball is stretched to give the illusion it is traveling quickly. Effect more extreme on the fall.
- At top of movement: hang time
- When ball hits the ground (and not before), it gets squashed horizontally.
- Shadow animation increases the optical illusion.
- Please note: These are exaggerations for the sake of a stronger illusion.



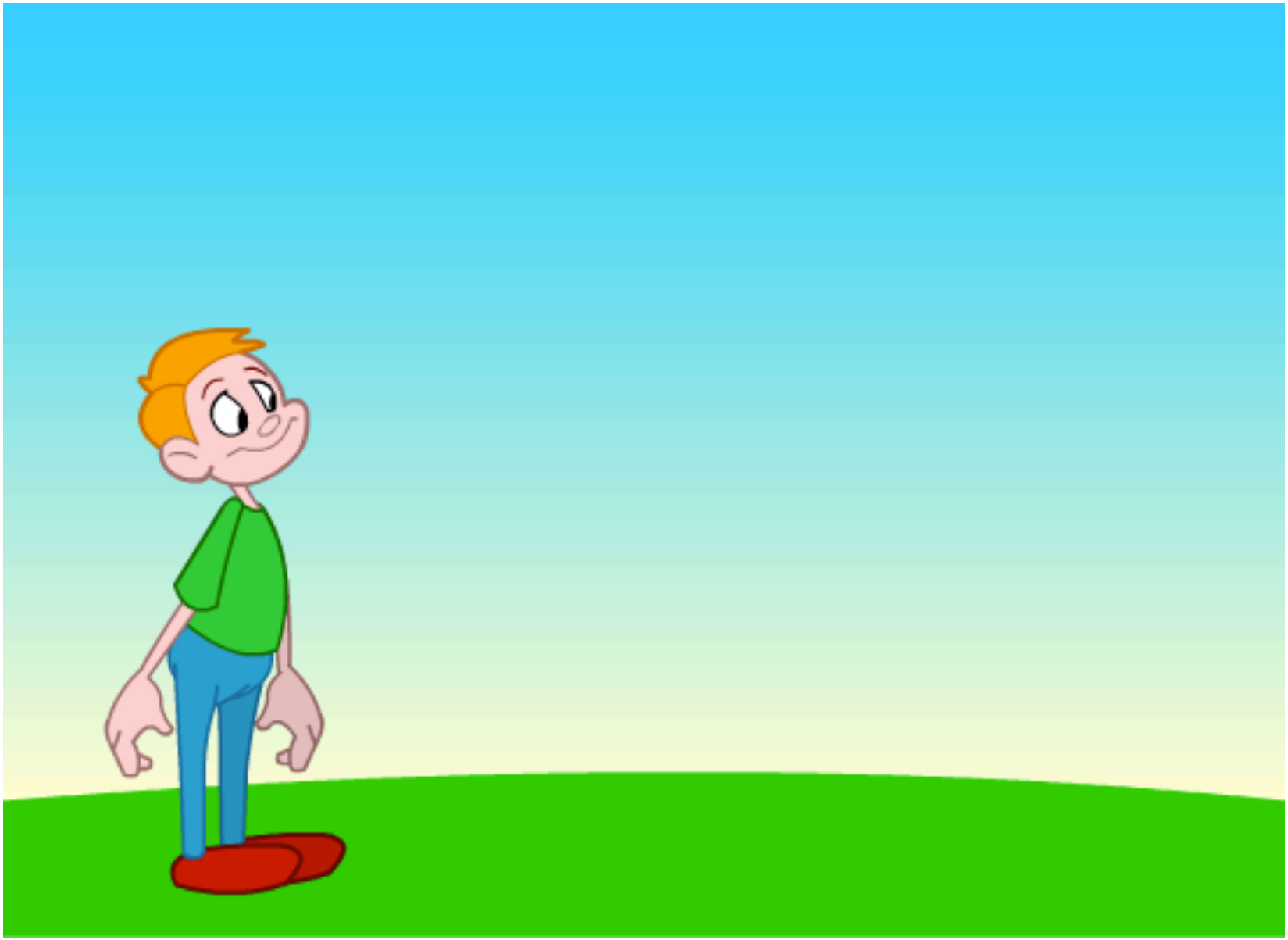
Anticipation/Reaction: Jumping Slime Ball



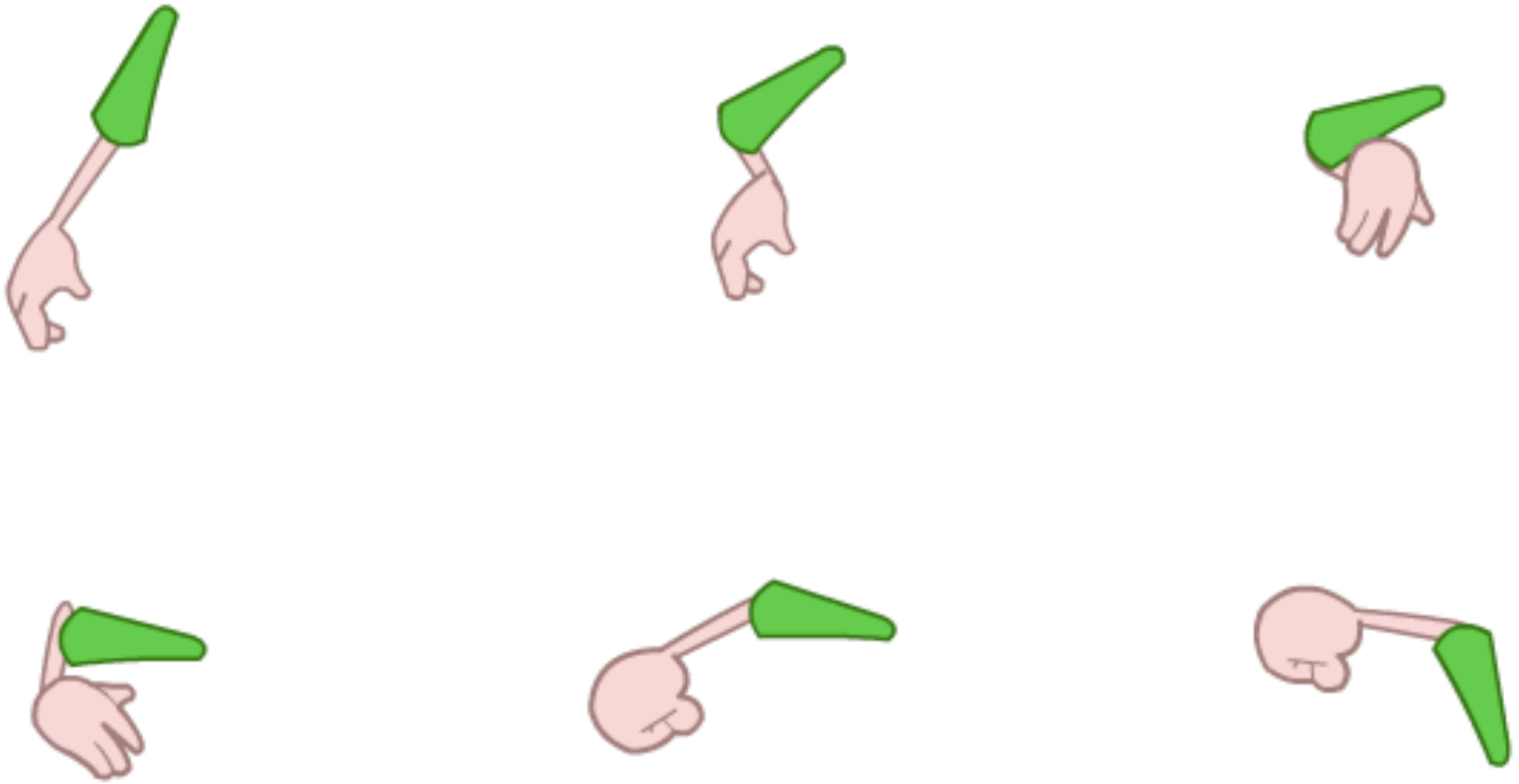
An Animated Comic Character (1)



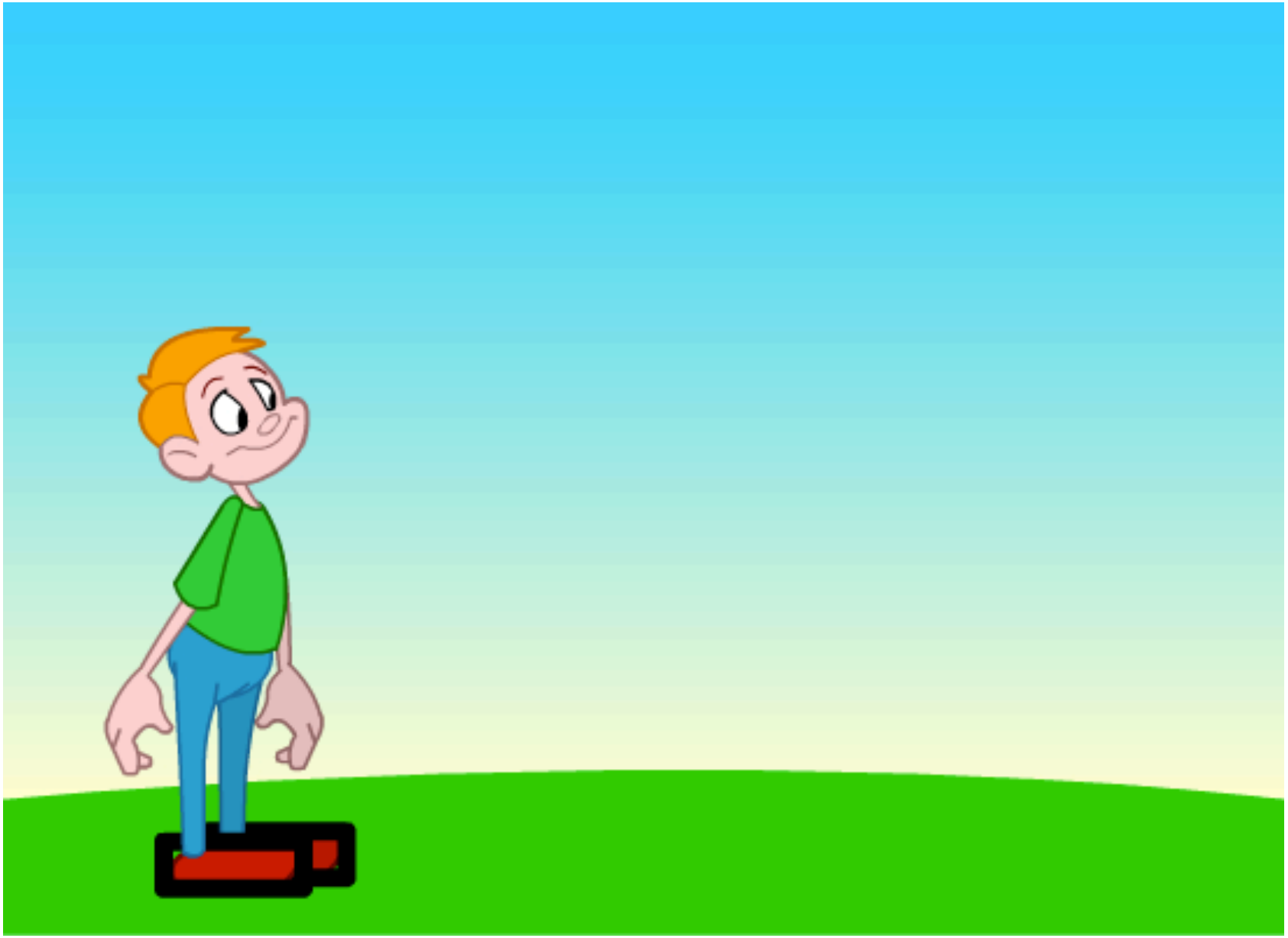
An Animated Comic Character (2)



Overlapping: Complexity of Movements



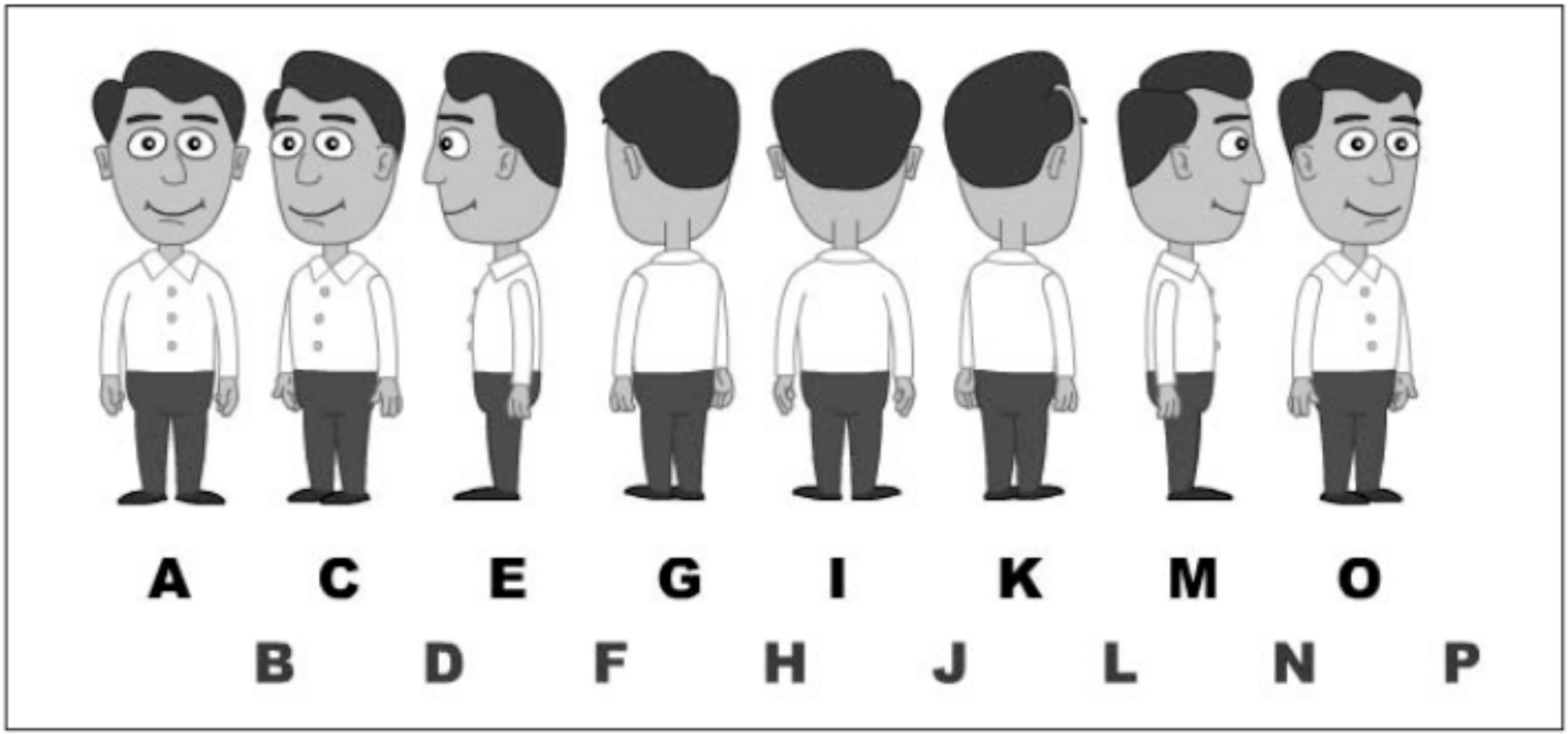
Smooth Arcs



Character Layout vs. Character Library

- Character Layout:
  - Animators draw key character poses for each scene
  - Requires many individual drawings
  - No limits for dynamic adaptation
  - Character may get off model more easily
- Character Library:
  - Uses pre-made library system for parts and poses of character
  - Multiple views of a character (view angles)
  - Eye charts, mouth charts etc.
  - Requires more preproduction time
  - Achieves higher productivity
  - Suitable for re-use of character in various episodes
- Also character layout method can benefit from symbol libraries (as in Flash)

Example: Viewing Angles (Turn-Around)



Source: T. Jones

Example: Mouth Chart

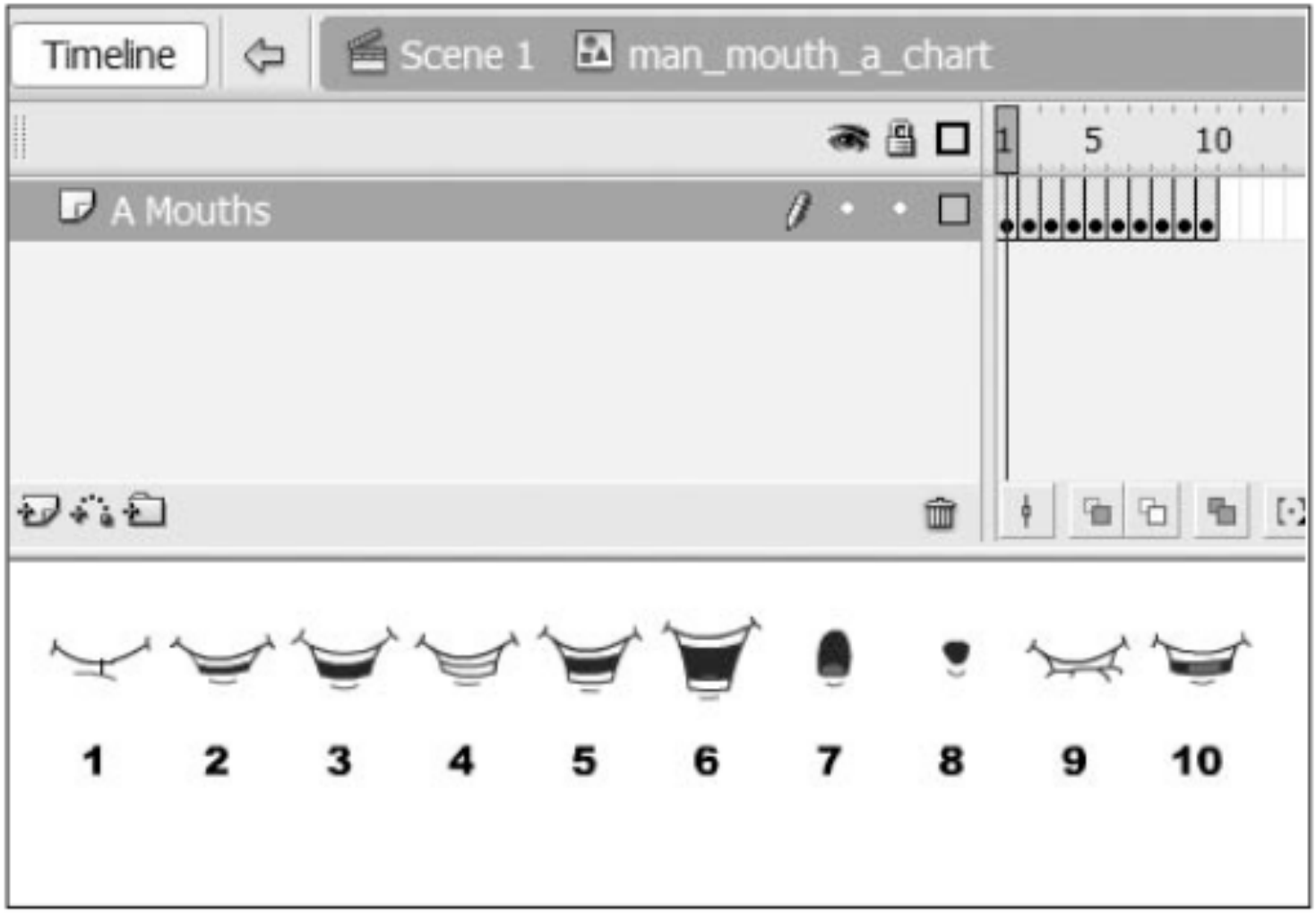


Figure 2-17. man_mouth_a_chart contains ten different mouth shapes.

Source: T. Jones

Sprite Sheets in CreateJS

EaselJS:

Animations created from charts showing individual motion phases

Sprite Sheet = Image file containing motion phases

```
var spriteSheet = new createjs.SpriteSheet({
    images: [imgMonsterARun],
    frames: {width: 64, height: 64, regX: 32, regY: 32},
    animations: {
        walk: [0, 9, "walk"]
    }
});
```

```
bmpAnimation = new
createjs.BitmapAnimation(spriteSheet);
bmpAnimation.gotoAndPlay("walk");
```

Example: David Rousset, http://blogs.msdn.com/b/davrous/

7 Programming with Animations

7.1 Animated Graphics: Principles and History

7.2 Types of Animation

7.3 Programming Animations

7.4 Design of Animations

- Principles of Animation
- Vector and Bitmap Graphics
- Creating a Game Character

7.5 Game Physics

Literature:

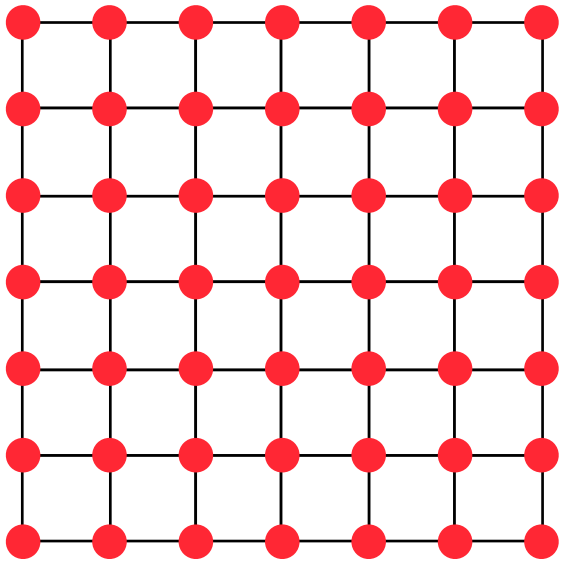
K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004
– Section 7.4 based on book chapter 2 by ***Brad Ferguson***
T. Jones et al.: Foundation Flash Cartoon Animation,
Apress/Friends of ED 2007

Vector vs Bitmap Graphics for Animation

- Vector Graphics
  - Efficient in memory and computation power
  - Requires special runtime engine
  - Adobe Flash is mainly based on vector animation
- Bitmap Graphics
  - Can also be efficient when images are small and directly handled by graphics hardware
  - Efficient bitmap handling built into HTML5-enabled browsers
  - HTML5/JavaScript animation in mainly built on bitmaps

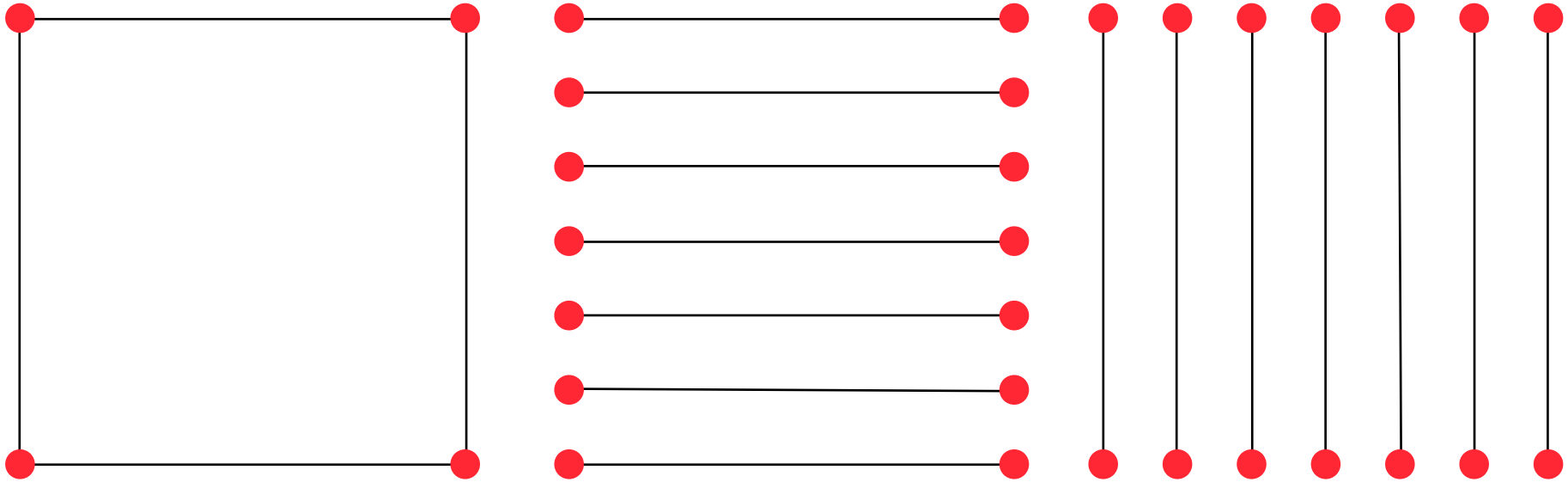
Complexity of Polygon Drawings

- Normal drawings in vector graphics programs (like Flash)
  - Every line has a vector point on each end
  - Every time a line makes a sharp bend, at least one new vector point is needed
  - Every time two lines intersect, yet another vector point is created



49 vector points

Optimized Vector Drawings



4+7*4 = 32 vector points

Crosshair principle: avoid intersections

Flash:

Use separate layers to draw lines which intersect.
Keep each layer free of intersections.

Optimizing a Comic Character

7 Programming with Animations

7.1 Animated Graphics: Principles and History

7.2 Types of Animation

7.3 Programming Animations

7.4 Design of Animations

- Principles of Animation
- Optimizing Vector Graphics
- Creating a Game Character

7.5 Game Physics

Literature:

K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004
– Section 7.4 based on book chapter 2 by ***Brad Ferguson***

T. Jones et al.: Foundation Flash Cartoon Animation,
Apress/Friends of ED 2007

The Story and Situation for a New Game

"The sound of the screaming alarms aboard the space cruiser abruptly awoke Space Kid, our ultimate hero of the futuristic universe, from his cryogenic nap. When our hero investigated, it was obvious his worst fears were now true. His loyal sidekick, teddy, had been bear-napped from the comfort of his own sleep chamber.

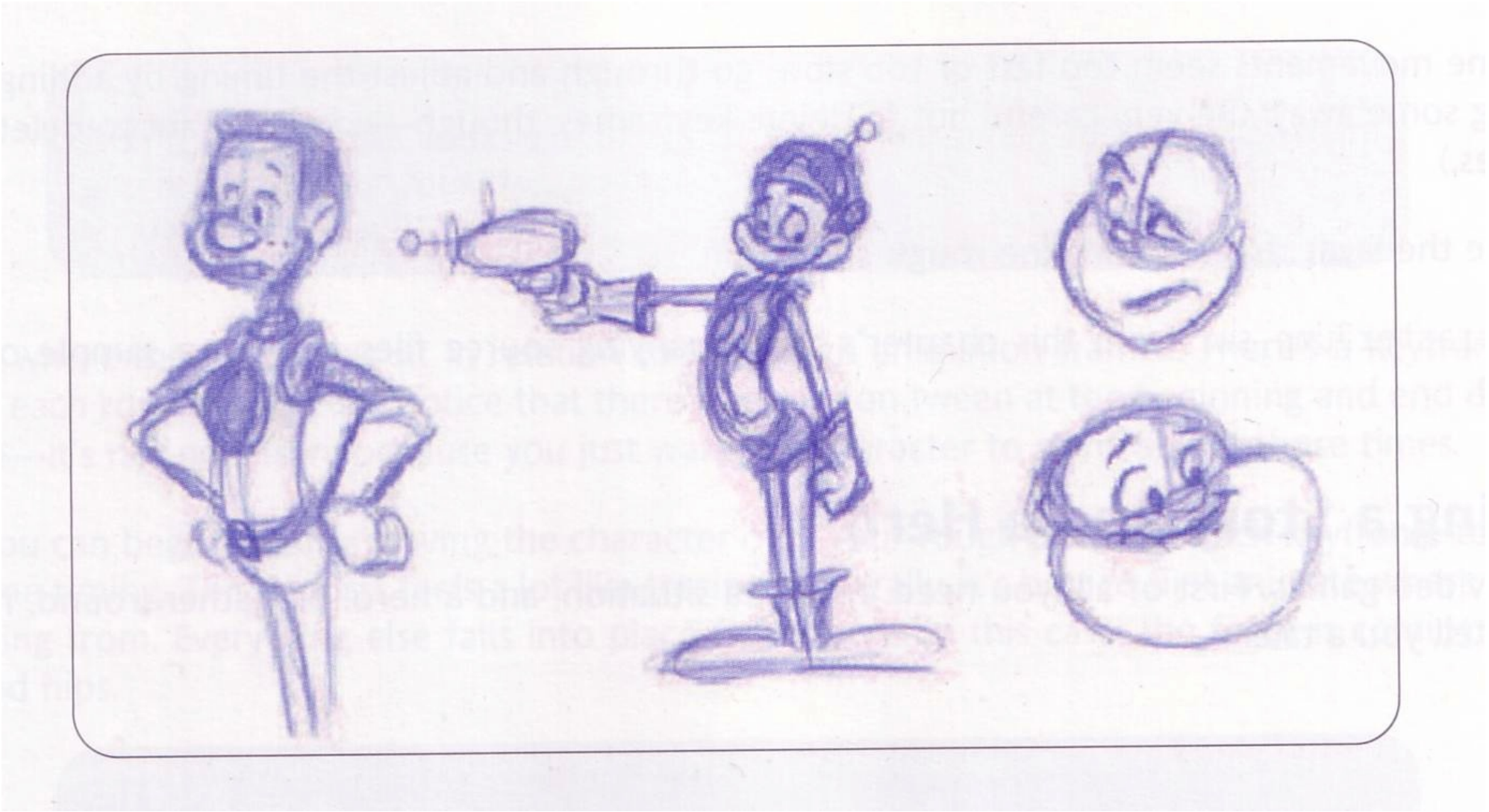
Immediately Space Kid knew there could only be one ruthless and vile enemy capable of committing such an atrocity: his longtime arch nemesis, Lord Notaniceguy.

Armed only with his trusty ray gun, Super Kid changes course for Quexxon Sector-G. Although our brave hero is fully aware he's falling for the bait in a trap, he must save Teddy."

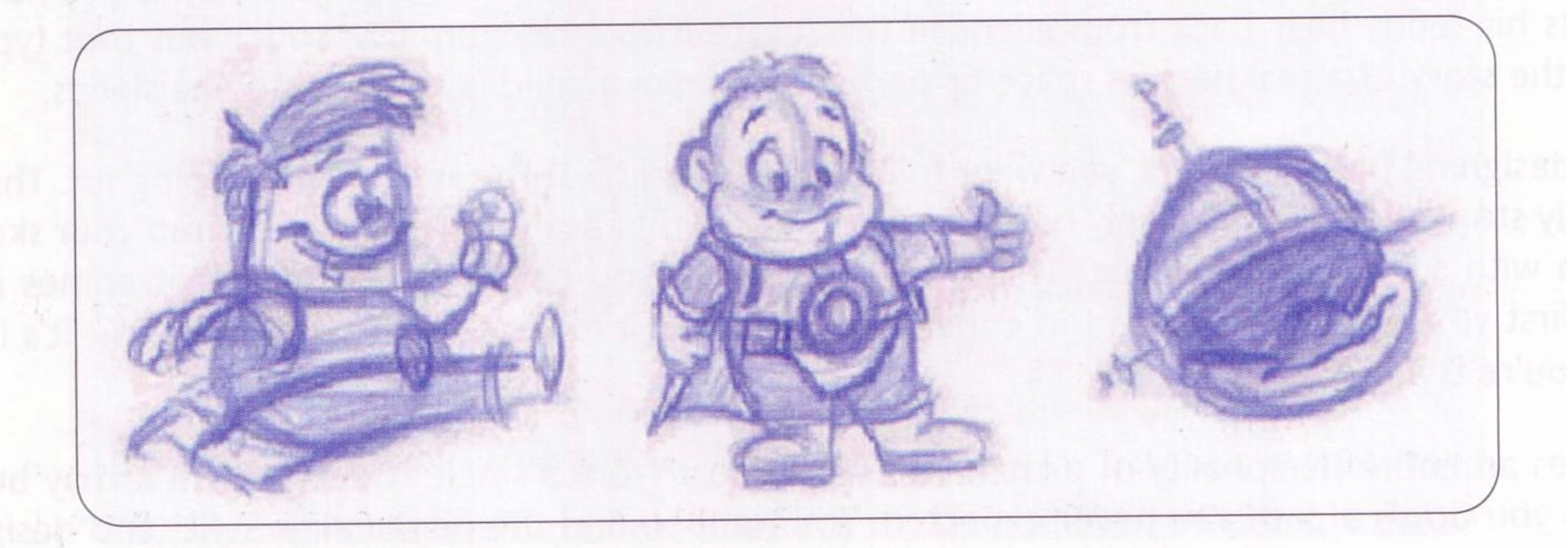
The Design Process

1. Create rough sketches of many different visual interpretations for the character (best with paper and pencil)
   - Brainstorming technique: Do *not* yet select!
2. Select among the gallery of characters according to compatibility with story, credibility, humour/seriousness, ...
3. Create rough sketches ***(paper and pencil)*** for the various animation sequences needed, e.g. run, jump, shoot, ...
   - Here usage of the authoring system can help already
4. Create optimized computer graphics for an “idle” animation.
5. Realize the animation sequences
   - Make sure that all sequences start and end with the idle position

Character Brainstorming (1)

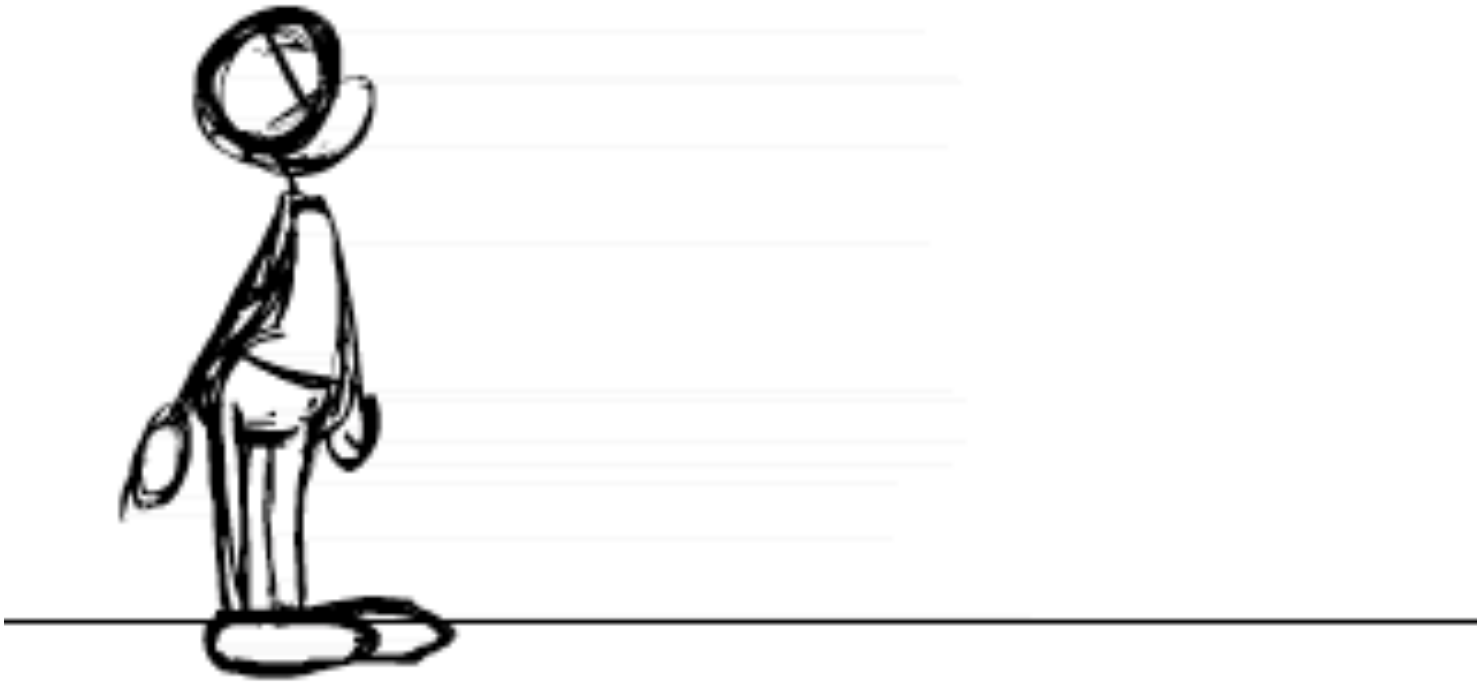


Character Brainstorming (2)



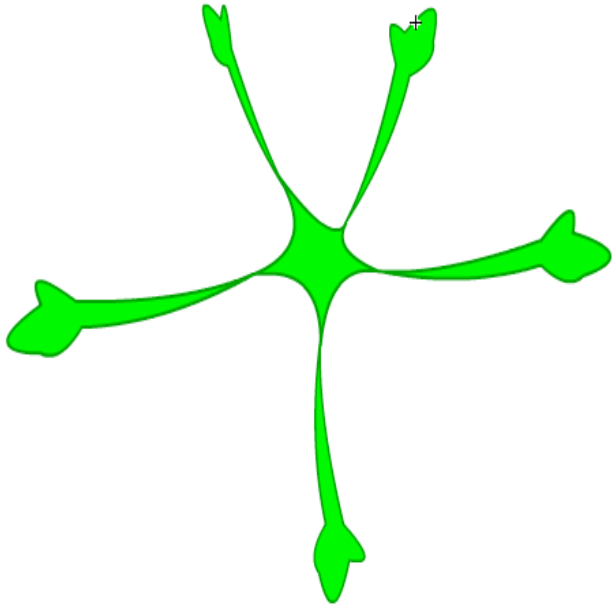
The Final Character

Rough Sketch for “Jump” Animation



The Enemies...

- Lord Notaniceguy’s space slugs...

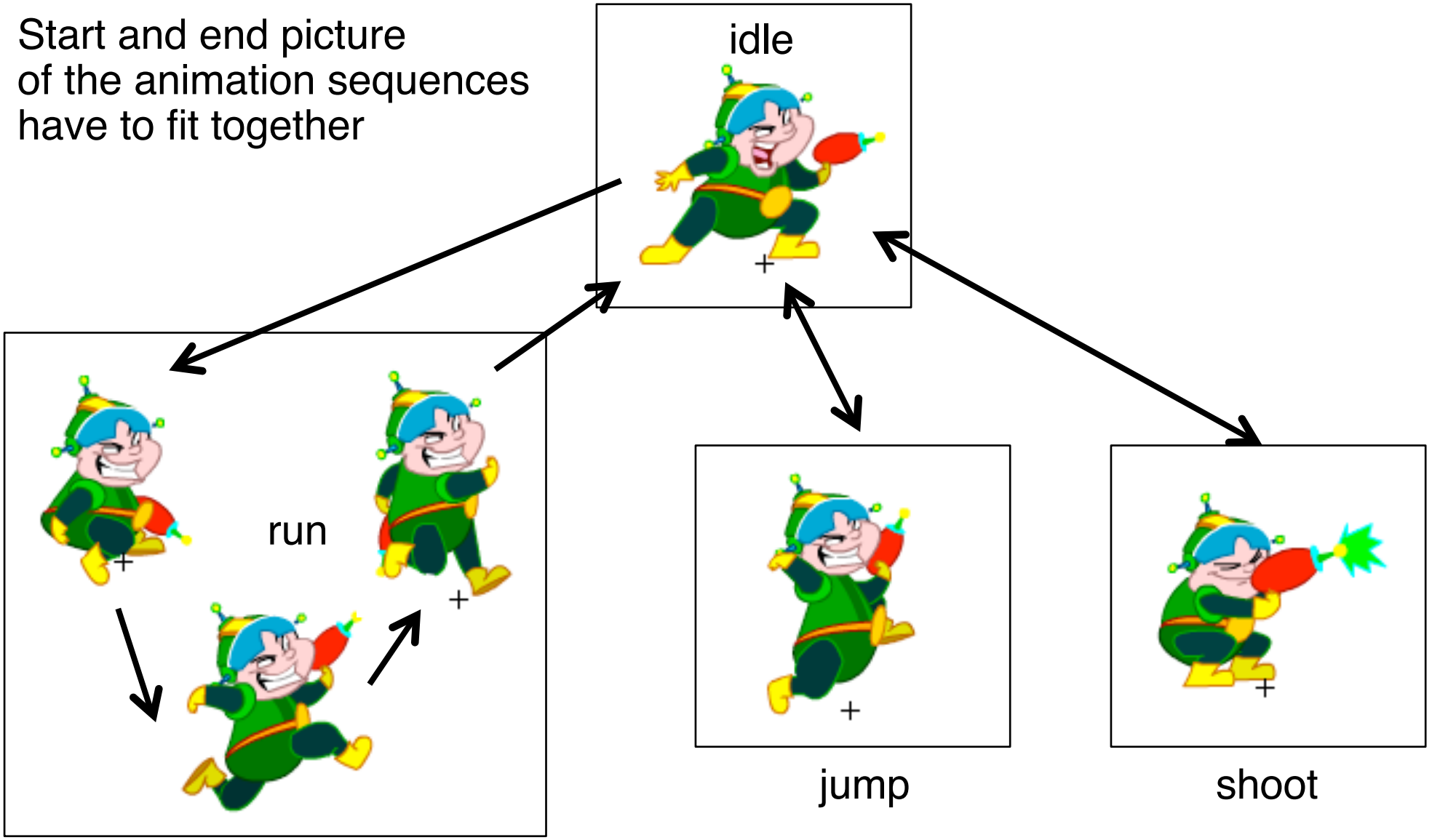


Physics and Animation

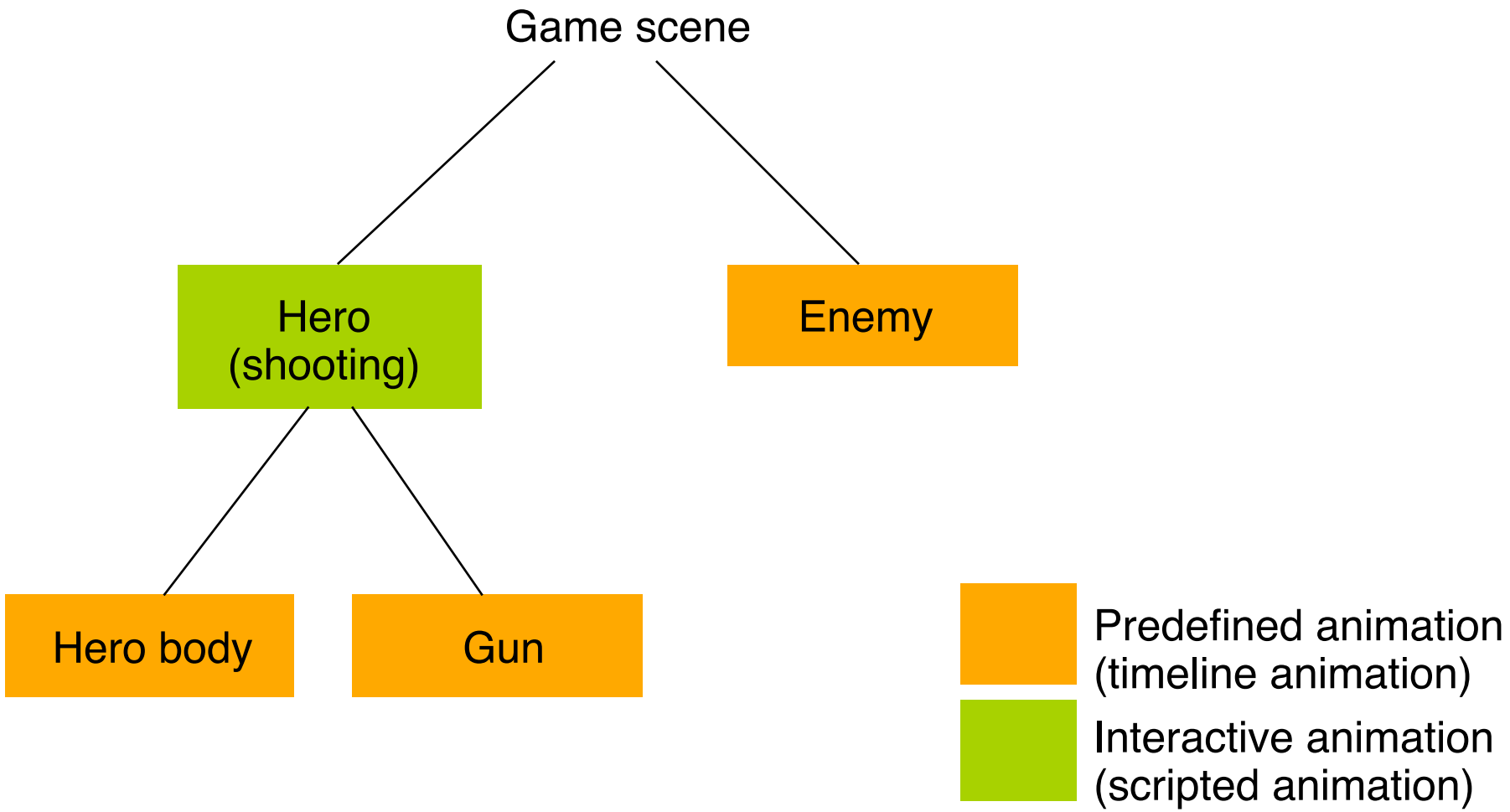
- Sprite animation:
  - Local changes in appearance
  - Often in a loop or a simple repetitive logic
  - May be parameterized
- Sprite movement:
  - Placement of sprite on stage, including speed and other parameters
  - Realized in program logic based on global event loop
- Consequence for movement animations (like jump):
  - Movements “as if staying on the ground”
  - Character design provides *central transition point* for the character
    - » In the middle of the bottom
    - » Must be the same point across all animation phases
    - » Used to determine whether ground has been hit, whether we are falling off an edge, ...

Continuity of Animation Sequences

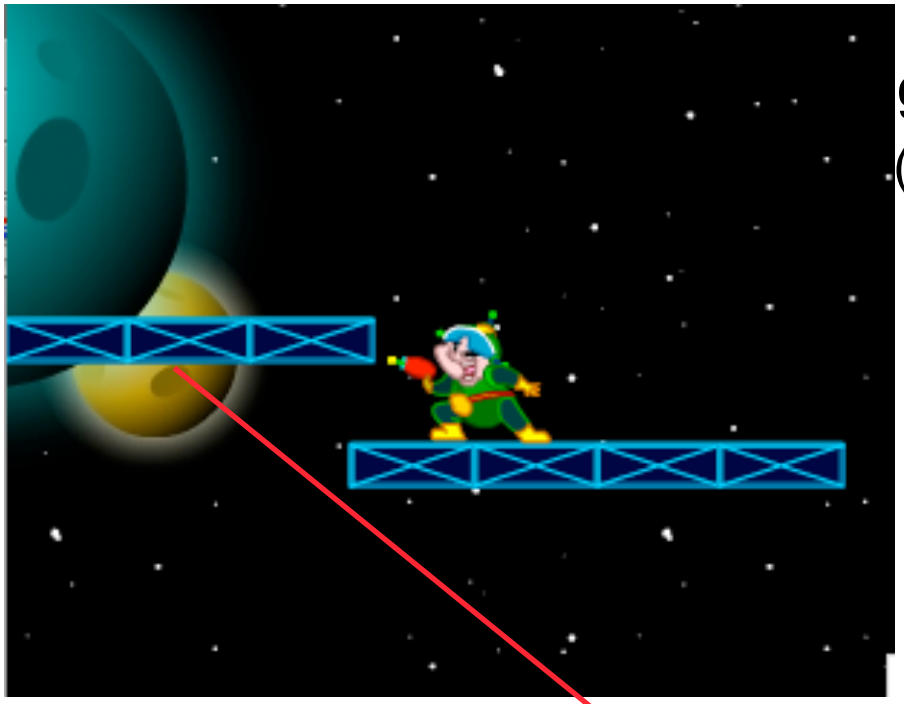
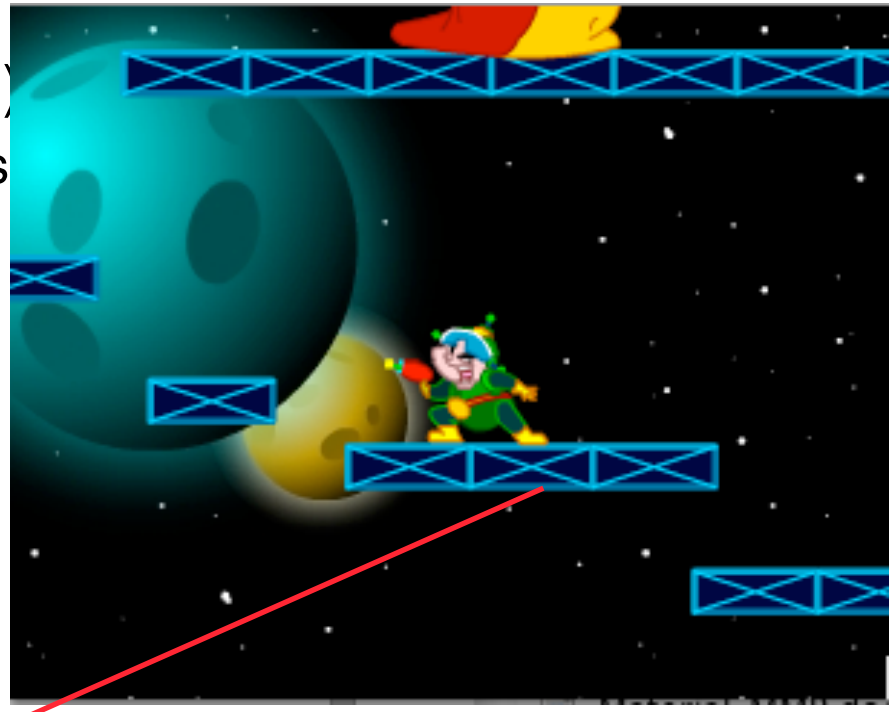
Start and end picture
of the animation sequences
have to fit together



Hierarchical Animations



Parallax/Multiplane Effect

- 
g)
(s


The same platform

7 Programming with Animations

7.1 Animated Graphics: Principles and History

7.2 Types of Animation

7.3 Programming Animations

7.4 Design of Animations

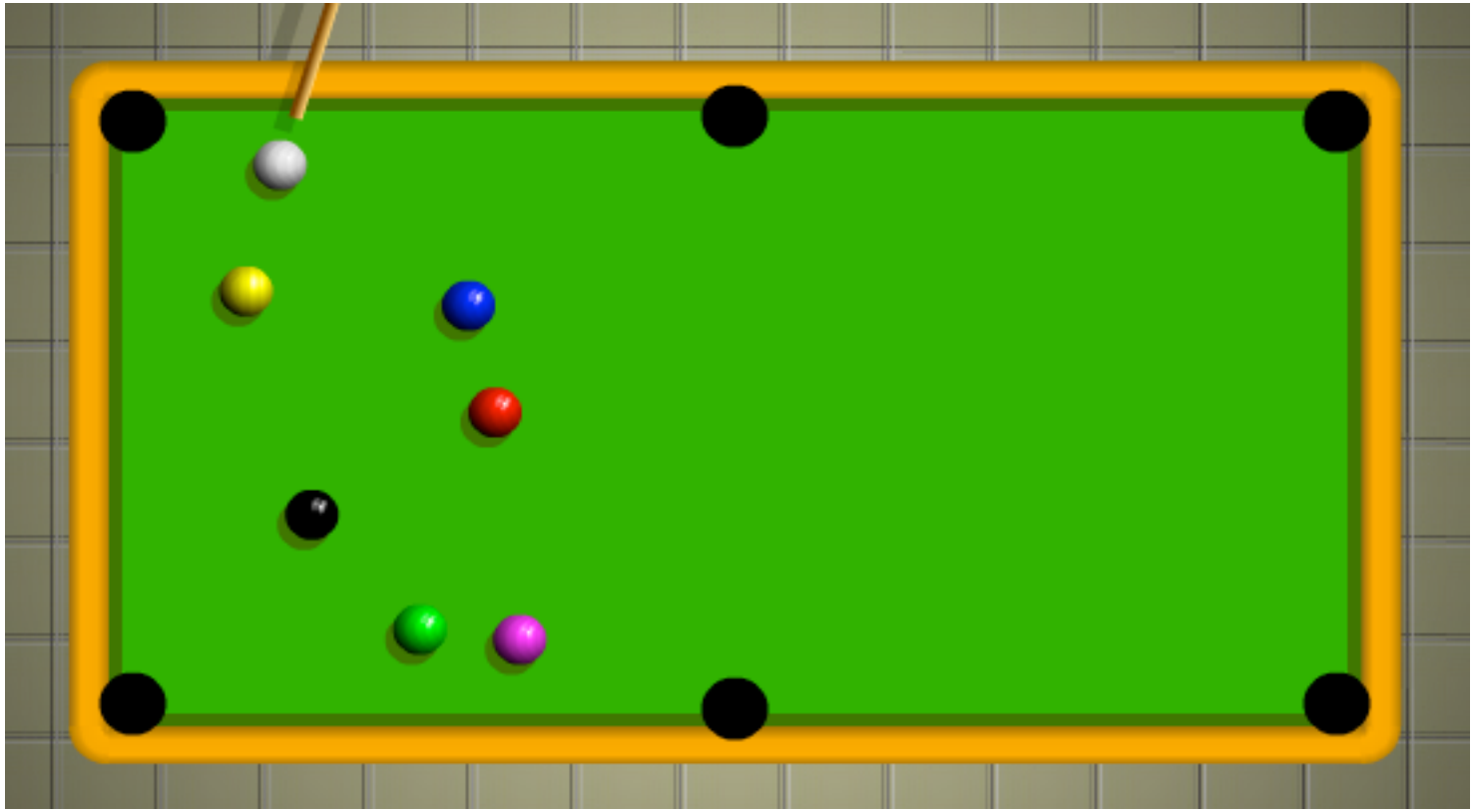
7.5 Game Physics

Literature:

K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004 (chapter 3 by Keith Peters)

Billiard-Game Physics

- Typical problem:
  - Two round objects moving at different speeds and angles hitting each other
  - How to determine resulting speeds and directions?
- Classical example: Billiard game

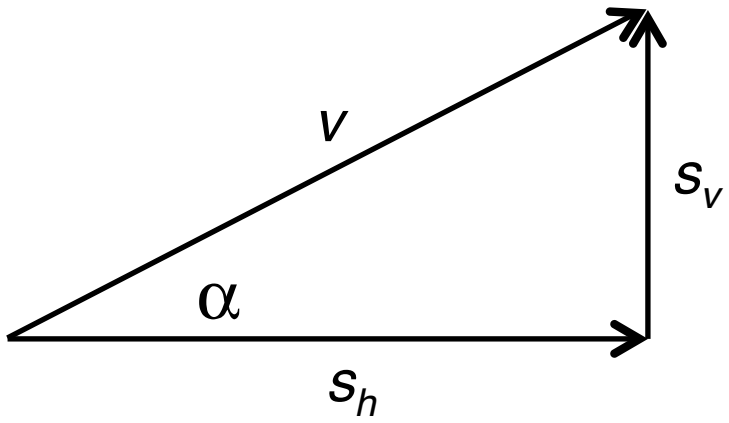


Speed and Velocity

- Speed:
  - Magnitude (single number), measured in px/s
  - Suitable for movement along one axis (e.g. x axis)
- Velocity:
  - Speed plus direction

    Magnitude (px/s) and angle (degrees)
  - Expressed as a 2D vector:

    velocity = (*horizontal_speed*, *vertical_speed*)



$$s_h = \cos(\alpha) \cdot v$$

$$s_v = \sin(\alpha) \cdot v$$

$$v = \sqrt{s_h^2 + s_v^2}$$

$$\alpha = \operatorname{atan}(s_v / s_h)$$

Velocity and Acceleration

- Velocity
  - is added to the position values in each frame cycle
- Acceleration
  - is a force *changing velocity*
  - *Acceleration* is added to velocity in each frame cycle
  - *Deceleration* is negative acceleration
- Angular acceleration
  - Acceleration is a 2D vector (or a magnitude plus angle)

```
vx += ax
vy += ay
x += vx
y += vy
```

(*ax*, *ay*) acceleration
(*vx*, *vy*) velocity

Object Orientation for Animated Graphics

- Each moving part of the scene is an *object* (belonging to a class)
- Class definitions comprise:
  - Reference to graphical representation
  - Properties mirroring physical properties (e.g. position, speed)
  - Properties and methods required for core program logic
- Two ways of referring to graphics framework:
  - *Subclassing:* Moving object belongs to a subclass of a framework class
  - *Delegation:* Moving object contains a reference to a framework object
  - Example JavaFX: Framework class **`Node`**

JavaFX Moving Ball as Direct Subclass

```
class Ball extends Circle {

    private double radius;
    protected double vx; // velocity x
    protected double vy; // velocity y

    public Ball(double posx, double posy,
                double vx, double vy, double r, Color c) {
        // reference for posx, posy is center of circle
        super(new Circle(r, c)); // centerX and centerY are 0
        radius = r;
        n.setTranslateX(posx);
        n.setTranslateY(posy);
        this.vx = vx;
        this.vy = vy;
    }
    …
}
```

JavaFX Sprite Class with Delegation

```
public class Sprite {

    protected Node n;
    protected double vx;
    protected double vy;
    private double maxX;
    private double maxY;

    public Sprite(Node n) {
        this.n = n;
    }
…
    public void update() {
        n.setTranslateX(n.getTranslateX()+vx);
        n.setTranslateY(n.getTranslateY()+vy);
    }
}
```

JavaFX Moving Ball as Sprite Subclass

```
class BallSprite extends Sprite {

    private double radius;

    public BallSprite(double posx, double posy,
              double vx, double vy, double r, Color c) {

        super(new Circle(r, c));
        radius = r;
        n.setTranslateX(posx);
        n.setTranslateY(posy);
        this.vx = vx;
        this.vy = vy;
    }
...
}
```

Main program for Moving Balls

```
BallSprite ball1 = new BallSprite
      (40, 40, 20, 10, r, Color.RED);
sprites.addSprite(ball1);
root.getChildren().add(ball1.getNode());

BallSprite ball2 = new BallSprite
      (300, 200, -15, -15, r, Color.BLUE);
sprites.addSprite(ball2);
root.getChildren().add(ball2.getNode());

Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
KeyFrame updateNodes = new KeyFrame(Duration.millis(20),
      new EventHandler<ActionEvent>() {
          public void handle(ActionEvent event) {
              //somehow call ball1.update(), ball2.update()
          }
      });
timeline.getKeyFrames().add(updateNodes);
```

Direct or through manager object

Collision Detection

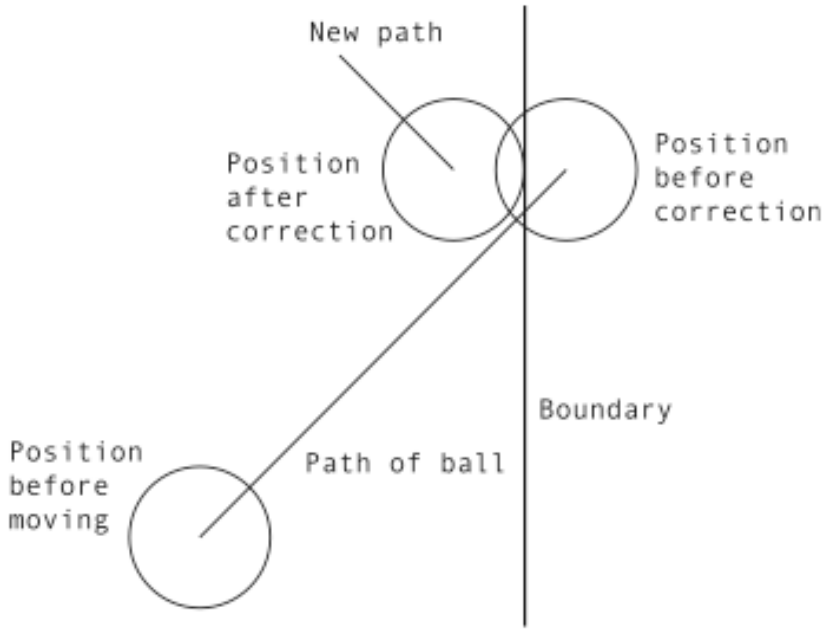
- Moving objects may meet other objects and boundaries
  - *Collision detection* algorithm detecting such situations
- Simple collision detection:
  - Width and/or height goes beyond some limit
  - Computation e.g. in JavaFX:
    `n.getBoundsInParent().getMaxX()`
- Potential problems:
  - Rounding errors may conceal collision event
  - Due to update rate, deep collisions may happen:
    - » Even after reaction to collision, collision condition is still true
    - » May lead to strange quick bouncing behavior

Repositioning of Object on Collision

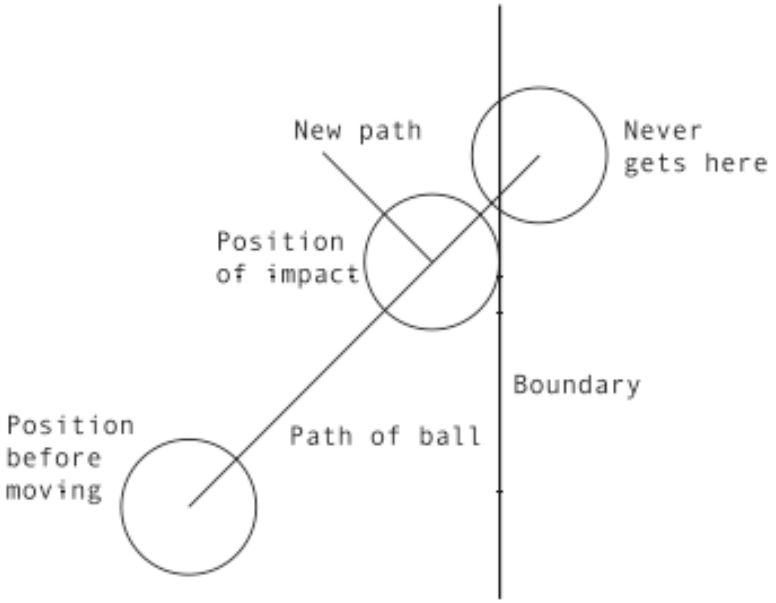
```
double overshootHR = n.getBoundsInParent().getMaxX()-maxX;
double overshootHL = n.getBoundsInParent().getMinX();
double overshootVD = n.getBoundsInParent().getMaxY()-maxY;
double overshootVU = n.getBoundsInParent().getMinY();
if (overshootHR > 0) {
    n.setTranslateX(n.getTranslateX()-overshootHR);
}
if (overshootHL < 0) {
    n.setTranslateX(n.getTranslateX()-overshootHL);
}
if (overshootVD > 0) {
    n.setTranslateY(n.getTranslateY()-overshootVD);
}
if (overshootVU < 0) {
    n.setTranslateY(n.getTranslateY()-overshootVU);
}
```

Model and Reality

What we are doing:



Real-world situation:

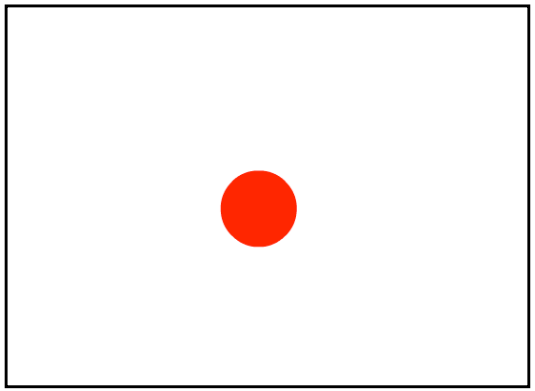


Simple Wall-to-Wall Bouncing

Bouncing always takes away some part of energy

- Use “bouncing factor”
- 1.0: No loss (unrealistic)
- In most cases use value smaller than 1, e.g. 0.9

```
if (overshootHR > 0 || overshootHL < 0) {
    vx = -vx*BOUNCELOSS;
}
if (overshootVD > 0 || overshootVU < 0) {
    vy = -vy*BOUNCELOSS;
}
```



Bounce and Friction

Surface absorbs some part of the energy and slows down the ball

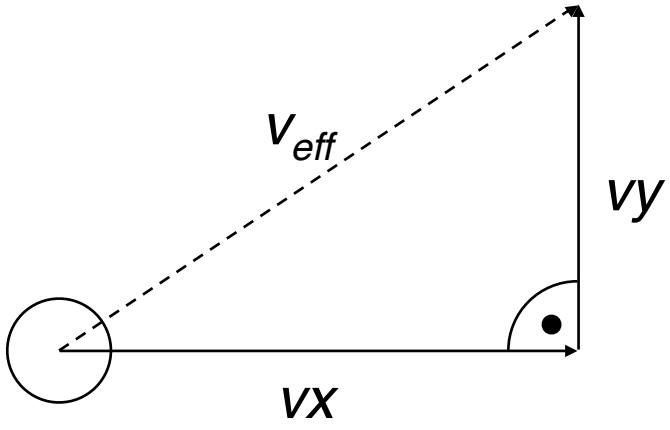
- Reduce velocity by some factor each frame
- Use “friction factor”
- 1.0: No friction (unrealistic)
- In most cases use value smaller than 1, e.g. 0.999

```
vx = FRICTLOSS*vx;
vy = FRICTLOSS*vy;
```

Minimum Speed

```
double effSpeed = Math.sqrt(vx*vx+vy*vy);
if (effSpeed < MINSPEED) {
    vx = 0;
    vy = 0;
}
```

Needed for this: Effective speed out of x and y velocities



$$v_{eff} = \sqrt{vx^2 + vy^2}$$

Collision Detection Between Balls (1)

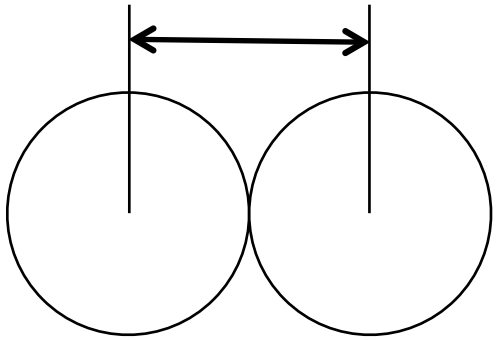
- Two balls are able to collide.
  - Collision detection needs access to data of both balls
  - E.g. Sprite manager class regularly calls collision check for sprite pairs
  - Bounding box is often not adequate for detection (e.g. for balls)
- Possible solution:
  - Abstract method in Sprite class
  - implemented in adequate way in all subclasses (e.g. BallSprite)

```
public void updateSprites() {
    for (Sprite s: sprite_list) {
        s.update();
    }
    for (Sprite s1: sprite_list) {
        for (Sprite s2: sprite_list) {
            if (s1 != s2) {
                s1.handleCollision(s2);
            }
        }
    }
```

```
abstract public void handleCollision(Sprite other);
```

Collision Detection Between Balls (2)

```
public void handleCollision(Sprite other) {
  if (other instanceof BallSprite) {
    Circle c1 = (Circle)this.getNode();
    Circle c2 = (Circle)other.getNode();
    double r1 = c1.getRadius();
    double r2 = c2.getRadius();
    double vx1 = this.getVelocityX();
    double vx2 = other.getVelocityX();
    double vy1 = this.getVelocityY();
    double vy2 = other.getVelocityY();
    double dx = c1.getTranslateX()-c2.getTranslateX();
    double dy = c1.getTranslateY()-c2.getTranslateY();
    double dist = Math.sqrt(dx*dx + dy*dy);
    if (dist < r1+r2) {
       collision detected
    }
...
```

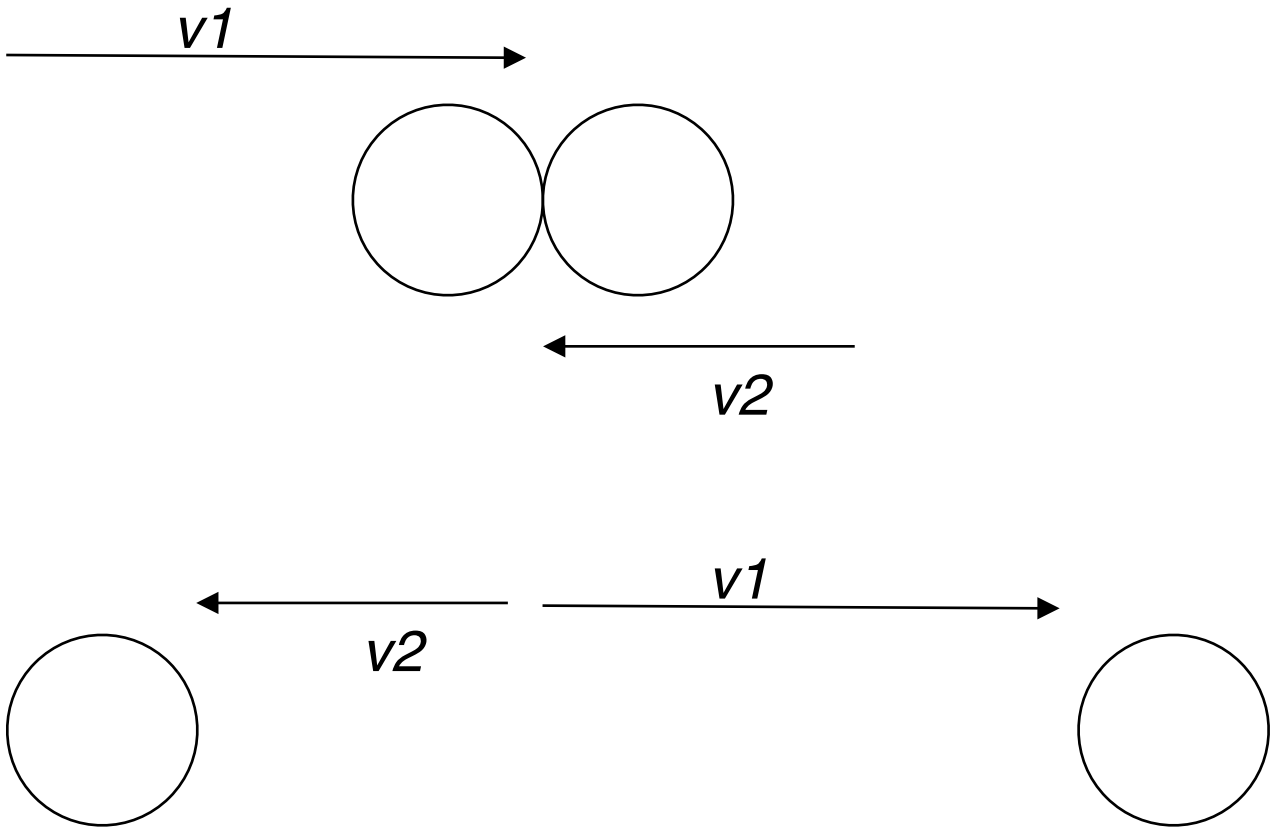


Physics: Speed, Velocity, Mass, Momentum

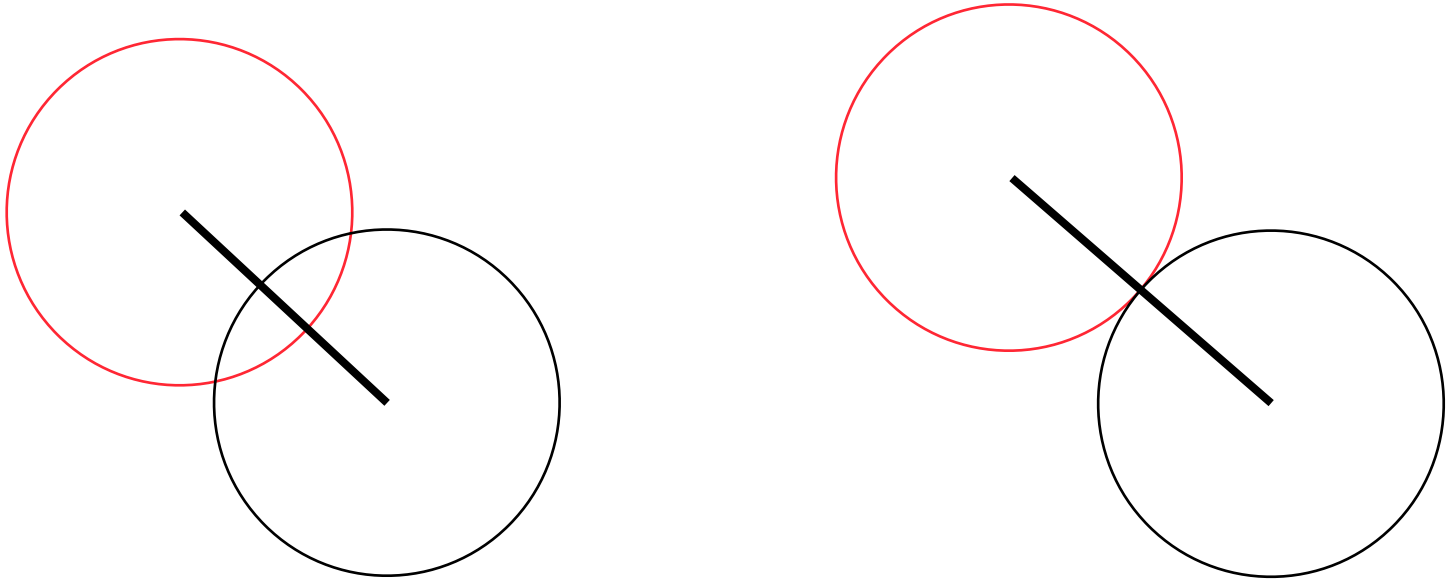
- *Speed:*
  - How fast is something moving (length/time)
- *Velocity:*
  - Vector describing movement: *speed* + *direction*
- *Mass:*
  - Basic property of object, depending on its material, leads under gravity to its *weight*
- *Momentum (dt. Impuls):*
  - Mass x Velocity
- *Principle of Conservation of Momentum (dt. Impulserhaltung):*
  - Total momentum of the two objects before the collision is equal to the total momentum after the collision.

A Simple Case of Collision

- Two balls collide “head on”
- Balls have same size and same mass

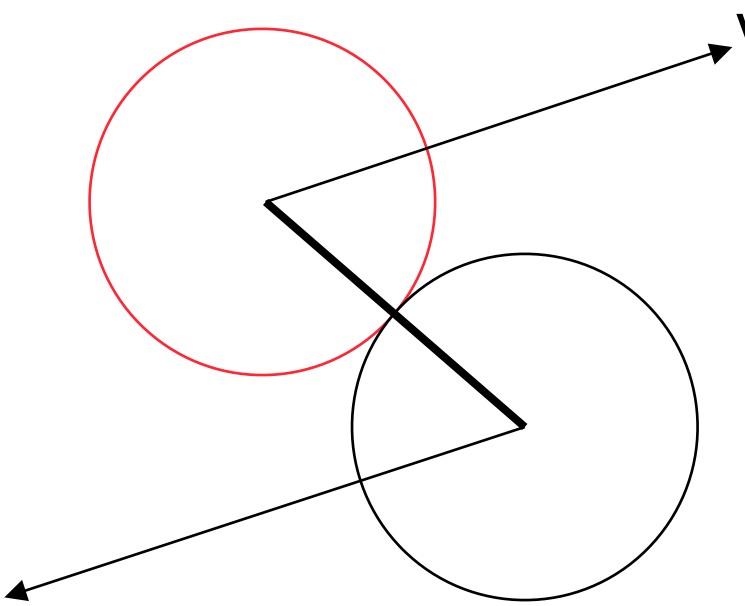


Repositioning Balls At Collision Time



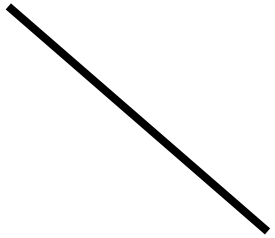
This is a simplification compared to the actual physical laws.

Physics of Collision, Step 1



Velocity of red ball

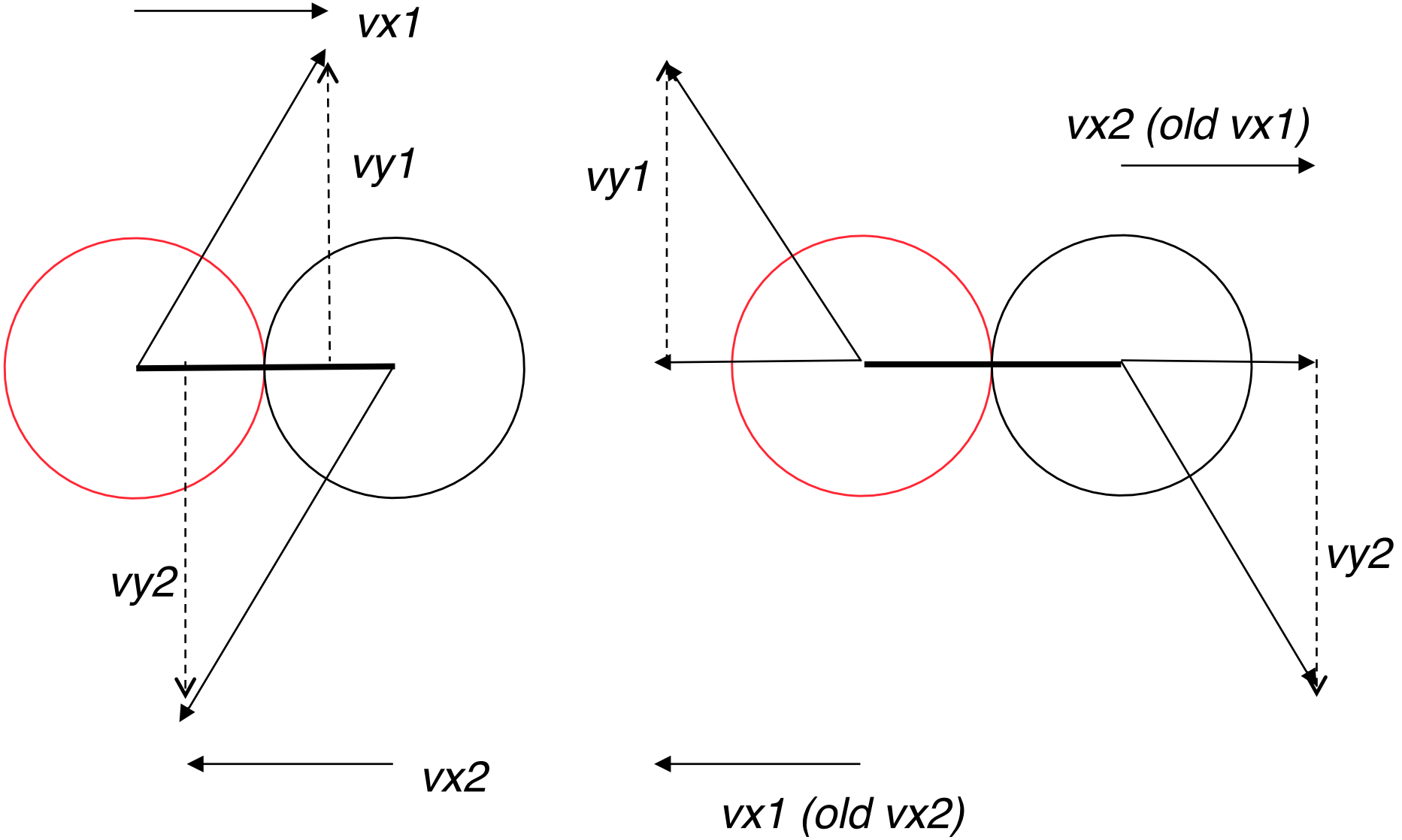
Velocity of white ball



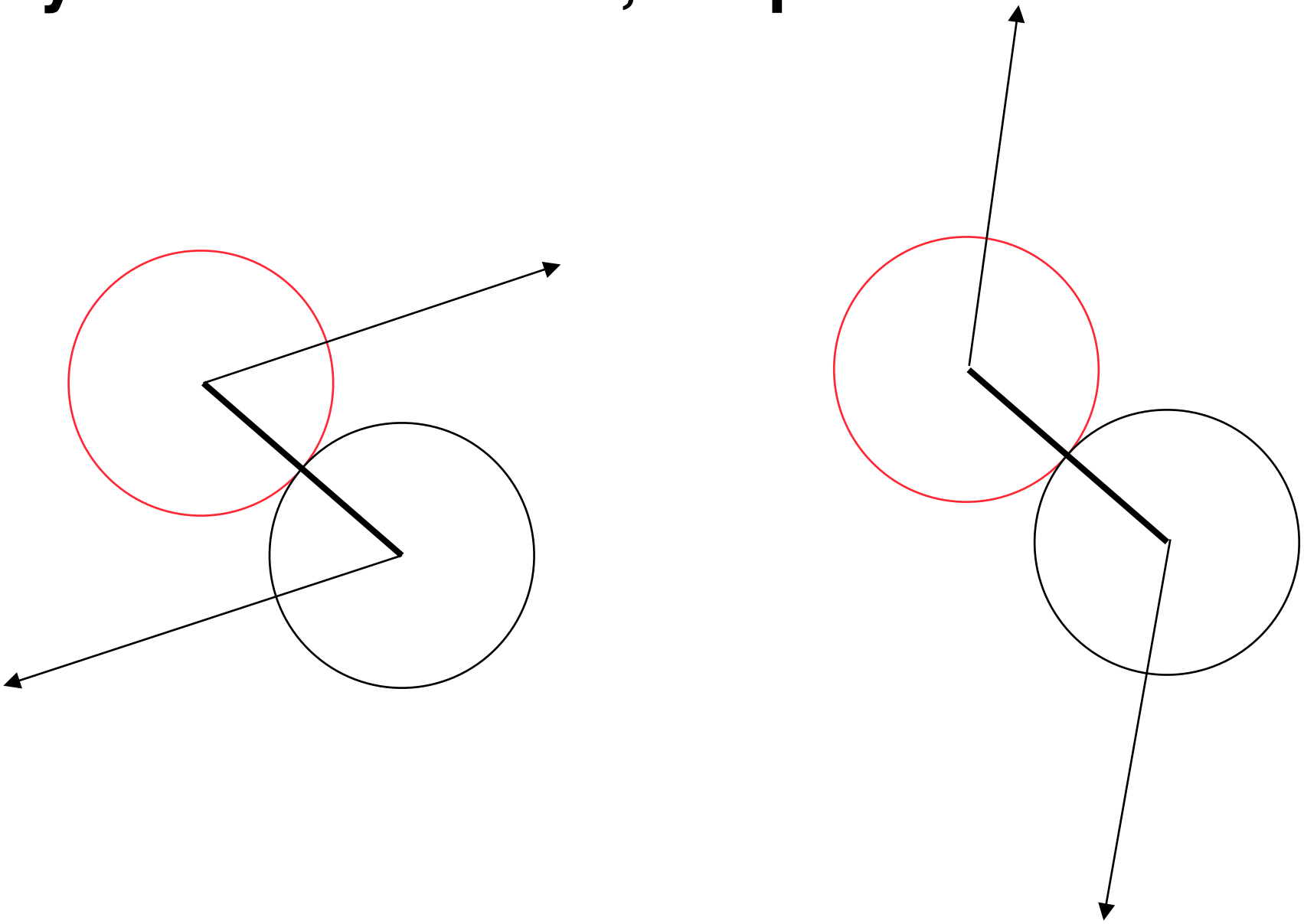
Line of collision

We need to determine those parts of the forces which actually contribute to the reaction, i.e. the projections on the collision line

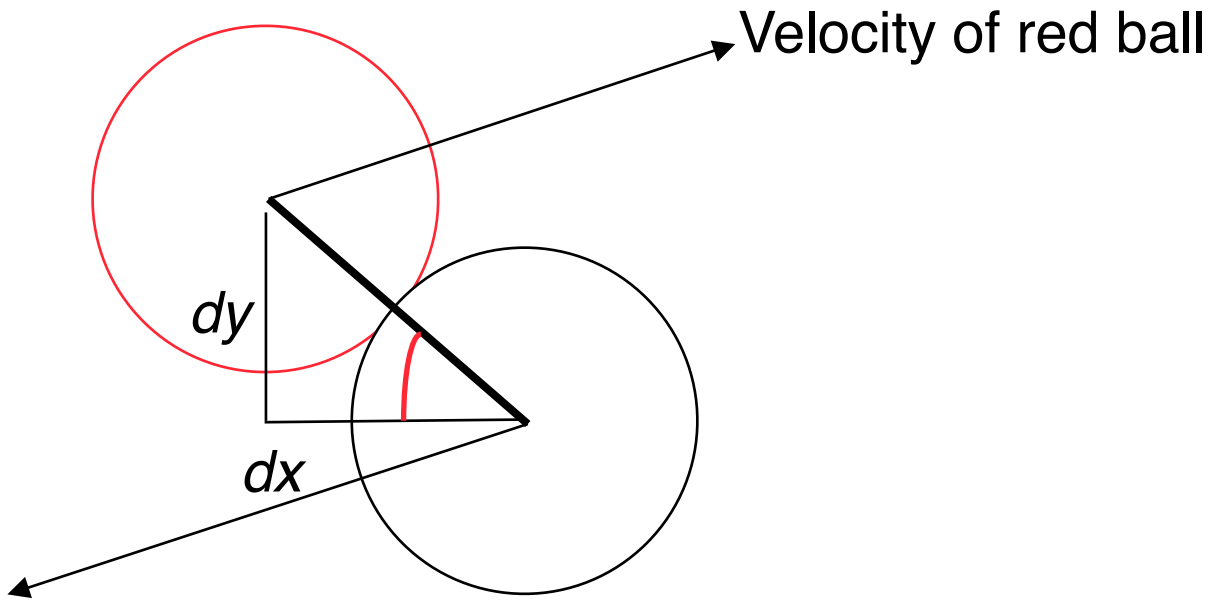
Physics of Collision, Step 2



Physics of Collision, Step 3

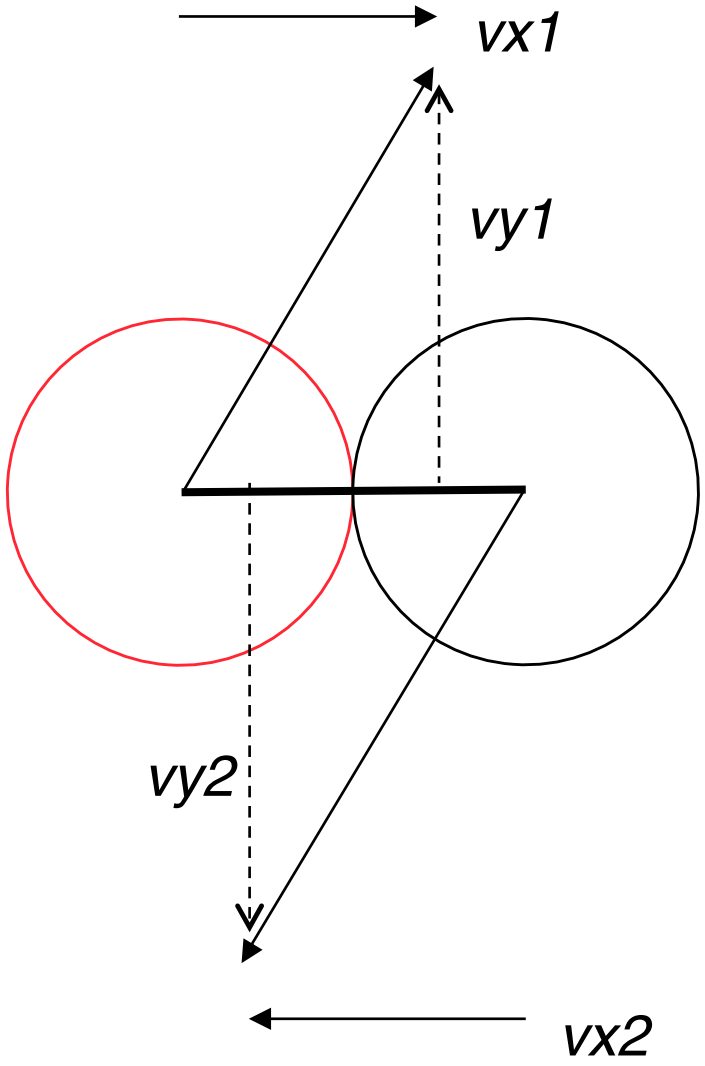


Computing Step 1




```
double angle =
  Math.atan2(dy,dx);
```

Computing Step 2, Part 1



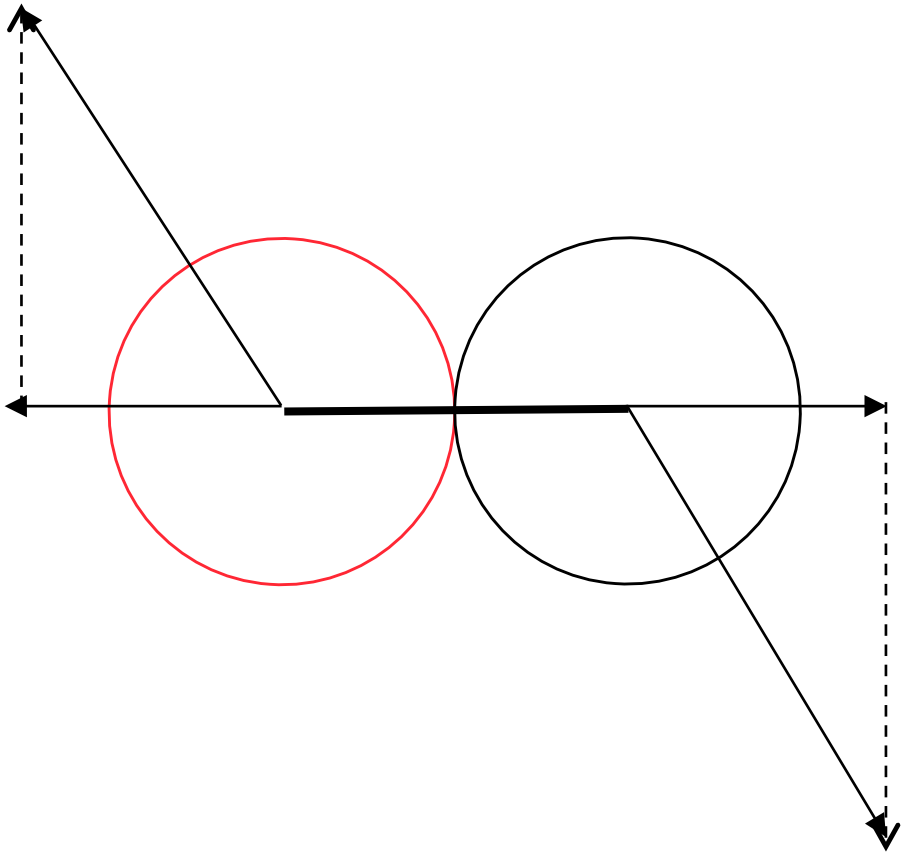
Counterclockwise rotation of vector (x, y):

$$x1 = \cos(\alpha) \cdot x + \sin(\alpha) \cdot y$$

$$y1 = \cos(\alpha) \cdot y - \sin(\alpha) \cdot x$$

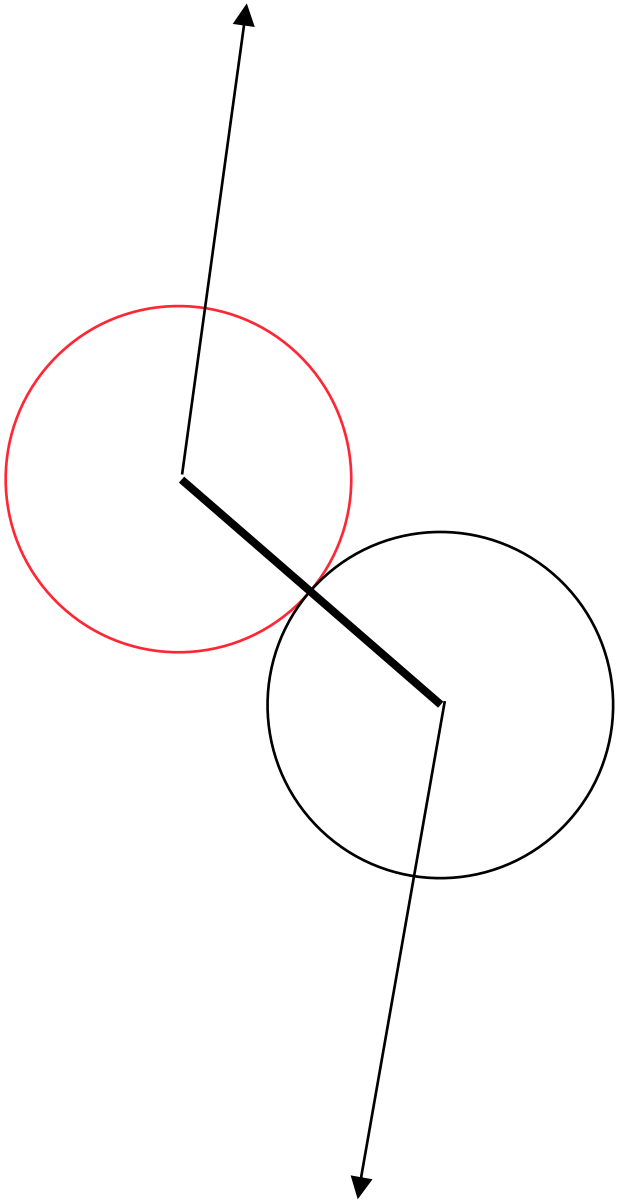
```
double angle = Math.atan2(dy,dx);
double cosa = Math.cos(angle);
double sina = Math.sin(angle);
double px1 = cosa*vx1+sina*vy1;
double py1 = cosa*vx1-sina*vy1;
double px2 = cosa*vx2+sina*vy2;
double py2 = cosa*vx2-sina*vy2;
```

Computing Step 2, Part 2



```
double pNewx1 = px2;
double pNewy1 = py2;
double pNewx2 = px1;
double pNewy2 = py1;
```

Computing Step 3



Clockwise rotation
of vector (x, y):

$$x1 = \cos(\alpha) \cdot x - \sin(\alpha) \cdot y$$

$$y1 = \cos(\alpha) \cdot y + \sin(\alpha) \cdot x$$

```
double vxNew1 = cosa*pNewx1-sina*pNewy1;
double vyNew1 = cosa*pNewx1+sina*pNewy1;
double vxNew2 = cosa*pNewx2-sina*pNewy2;
double vyNew2 = cosa*pNewx2+sina*pNewy2;
```

Forward and Inverse Kinematics

- ***Kinematics*** = "the branch of classical mechanics that describes the motion of objects without consideration of the causes leading to the motion" (Wikipedia)
  - Important in *robotics* and *animation*
  - Considering complex/composite objects (not only atomic objects like balls)
- ***Forward*** kinematics:
  - Given a complex sequence of joints, and all the angles of the joints, what is the position of the end?
- ***Inverse*** kinematics:
  - Given a complex sequence of joints with some constraints, and all the desired position of the end, what are the angles of the joints to achieve the given end position?
- Animals and humans intuitively use inverse kinematics.
- In robotics and animation, we need adequate algorithms to model inverse kinematics.