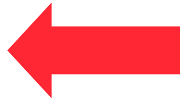
4 Programming with Animations

4.1 Animated Graphics: Principles and History

4.2 Types of Animation

4.3 Programming Animations

4.4 Design of Animations

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

4.5 Game Physics

Literature:

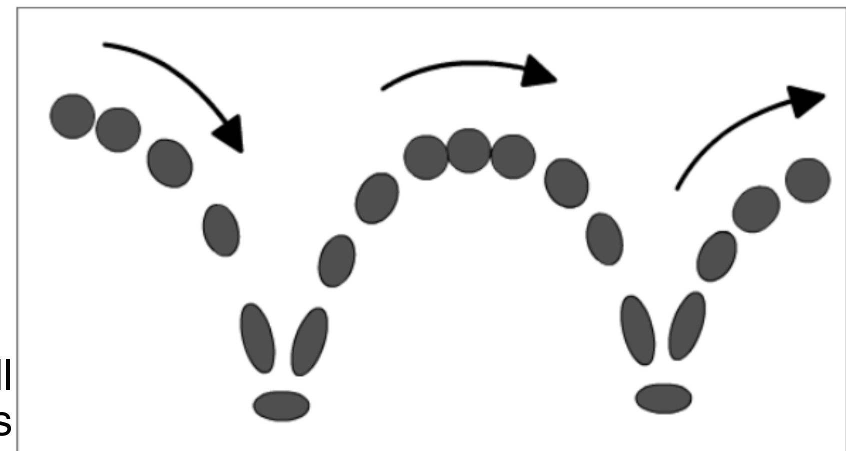
K. Besley et al.: Flash MX 2004 Games Most Wanted, Apress/Friends of ED 2004

– Section 4.4 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

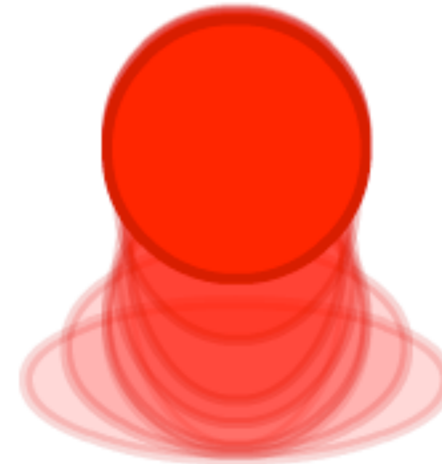
- Squash and Stretch
  - Shape of subject reacts to speed and force of movement
- Timing
  - E.g. ease-in and ease-out: Gives animation a sense of weight and gravity
- Anticipation, Action, Reaction, Overlapping Action
  - Anticipation: Build up energy before a movement
  - Reaction: Don't simply stop, 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

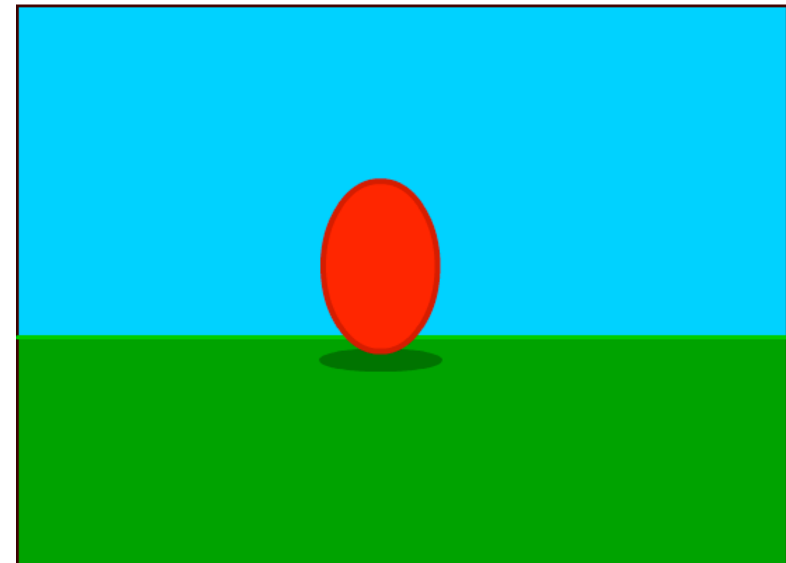


Bouncing ball
Source: T. Jones

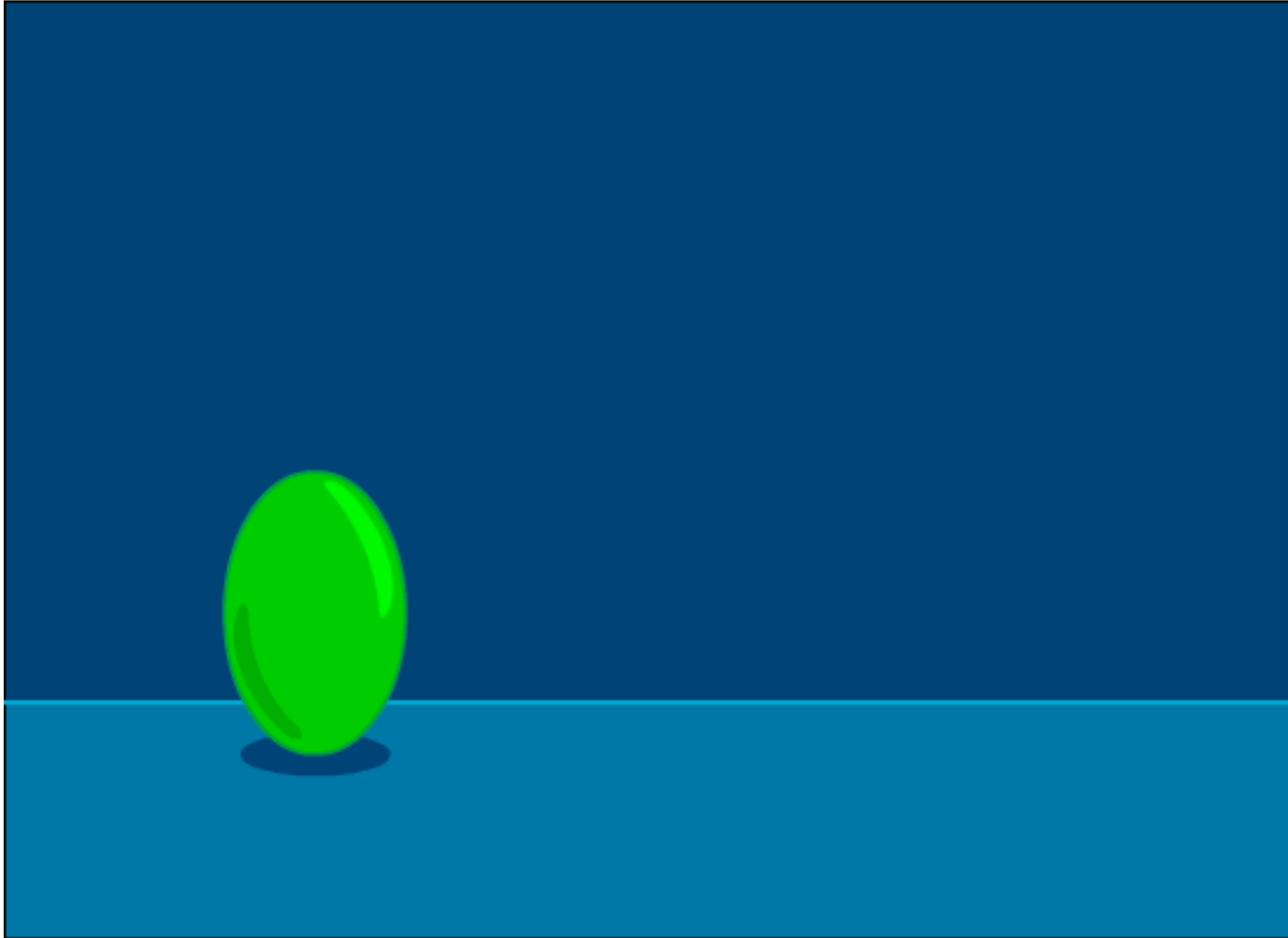
Squash & Stretch: Animating a Bouncing Ball

- When the ball is going up, it is fighting the force of gravity and will therefore be slower than when it falls.
- On rise and fall, the ball is stretched to give the illusion it is travelling quickly. This effect should be more extreme on the fall.
- At the top of movement, the ball has a certain hang time.
- As soon as the ball hits the ground (and not before), it gets squashed horizontally.
- A shadow animation increases the optical illusion.
- Please note: These are exaggerations for the sake of a stronger illusion.

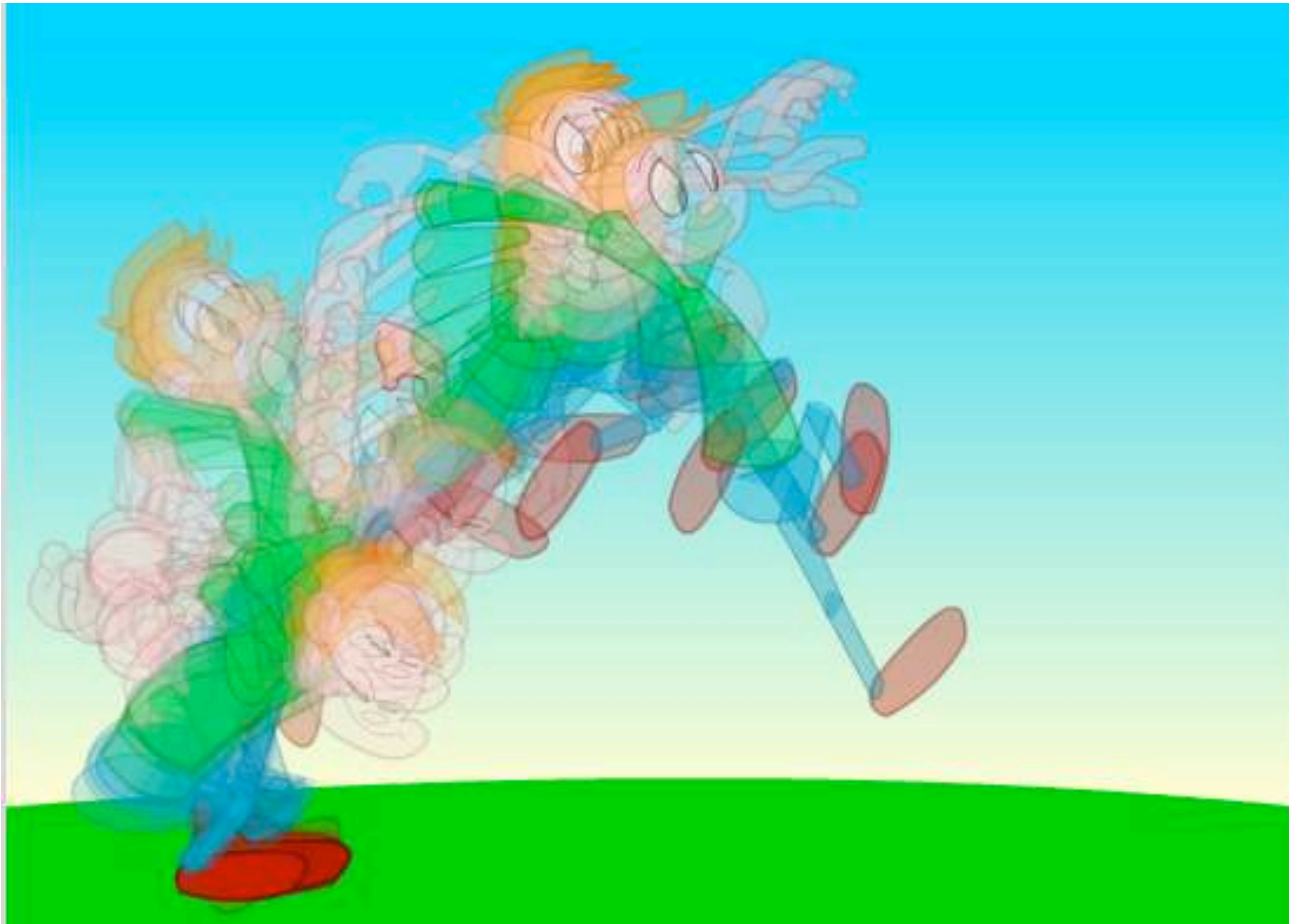




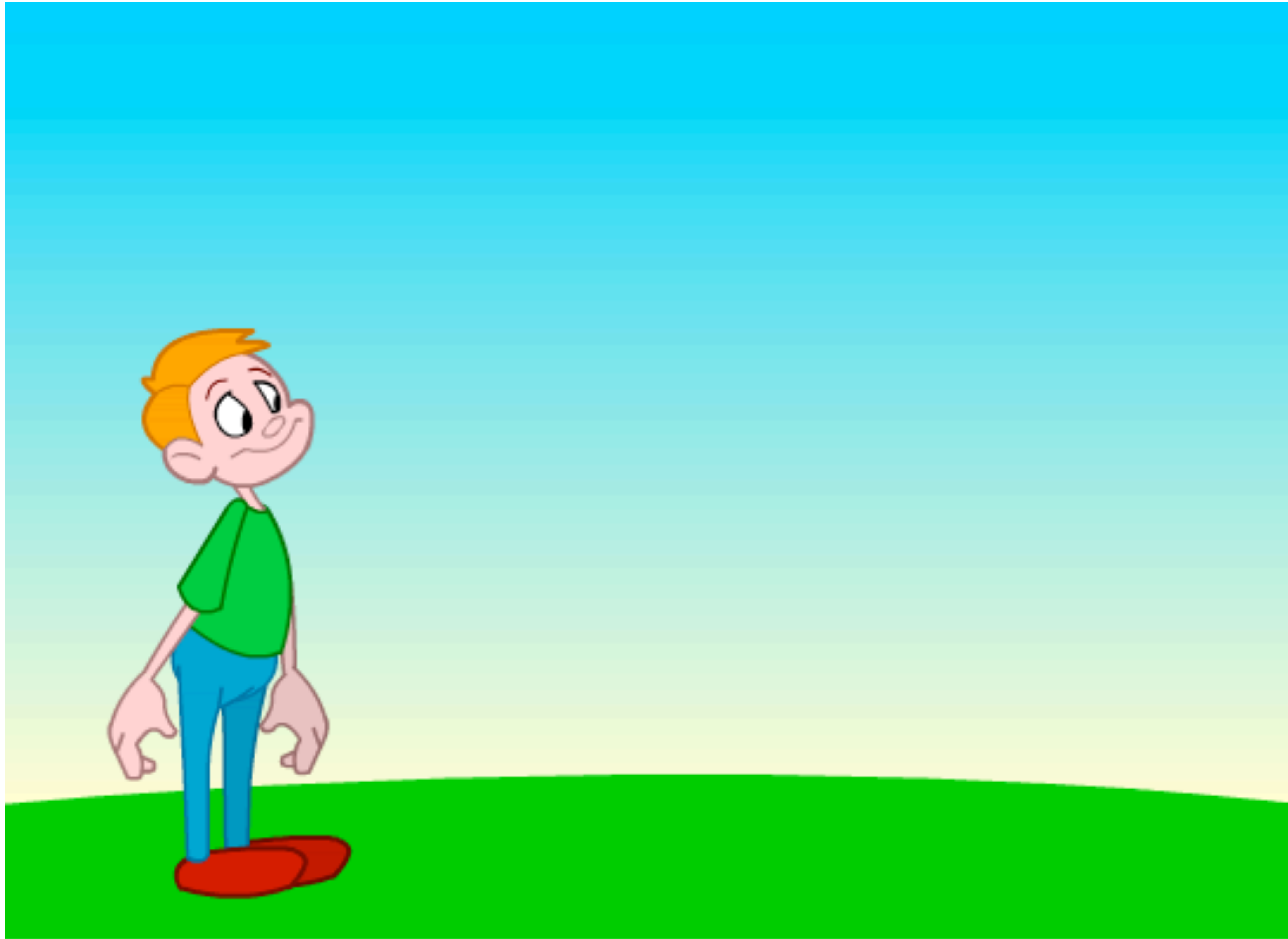
Anticipation/Reaction: Jumping Slime Ball



An Animated Comic Character (1)

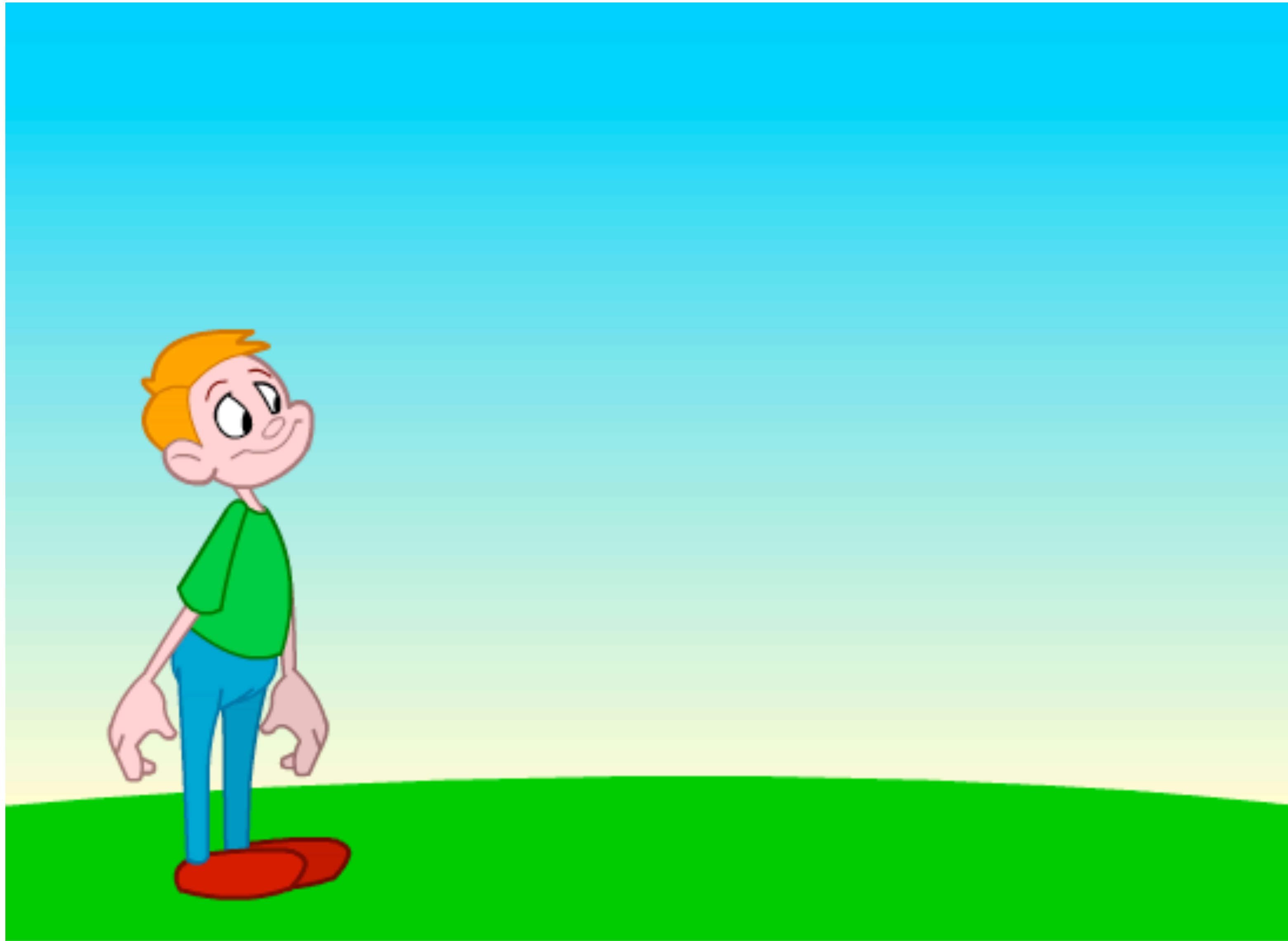


An Animated Comic Character (2)



Overlapping: Complexity of Movements

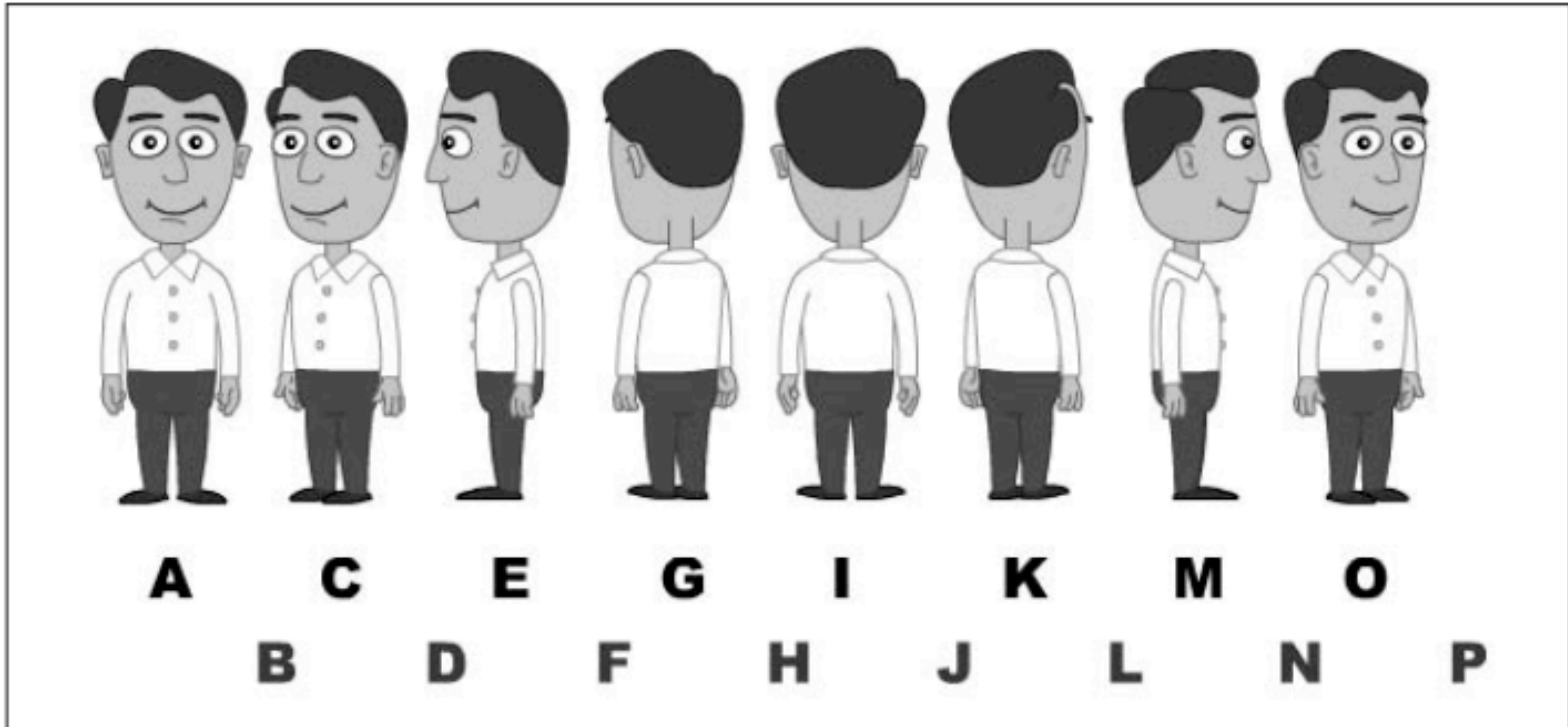
Smooth Arcs



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

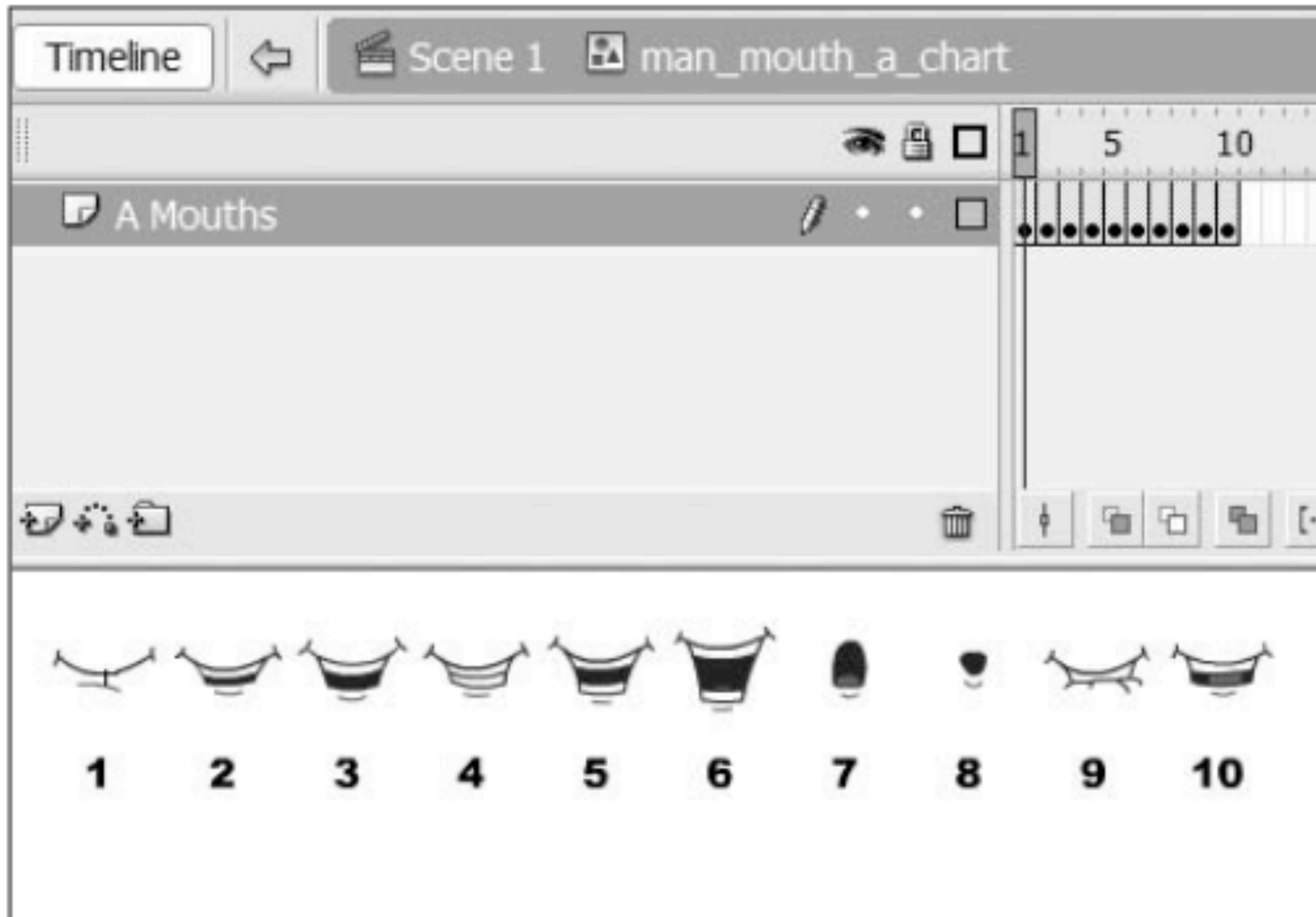


Figure 2-17. man_mouth_a_chart contains ten different mouth shapes.

Source: T. Jones

4 Programming with Animations

4.1 Animated Graphics: Principles and History

4.2 Types of Animation

4.3 Programming Animations

4.4 Design of Animations

Principles of Animation

Optimizing Vector Graphics

Creating a Game Character

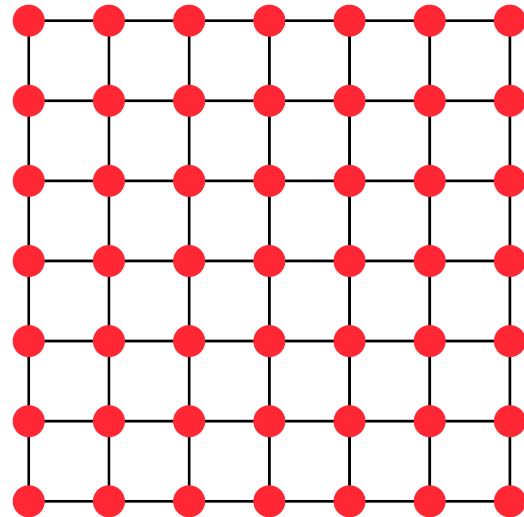
4.5 Game Physics

Literature:

K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004
– Section 4.4 based on book chapter 2 by ***Brad Ferguson***
T. Jones et al.: Foundation Flash Cartoon Animation,
Apress/Friends of ED 2007

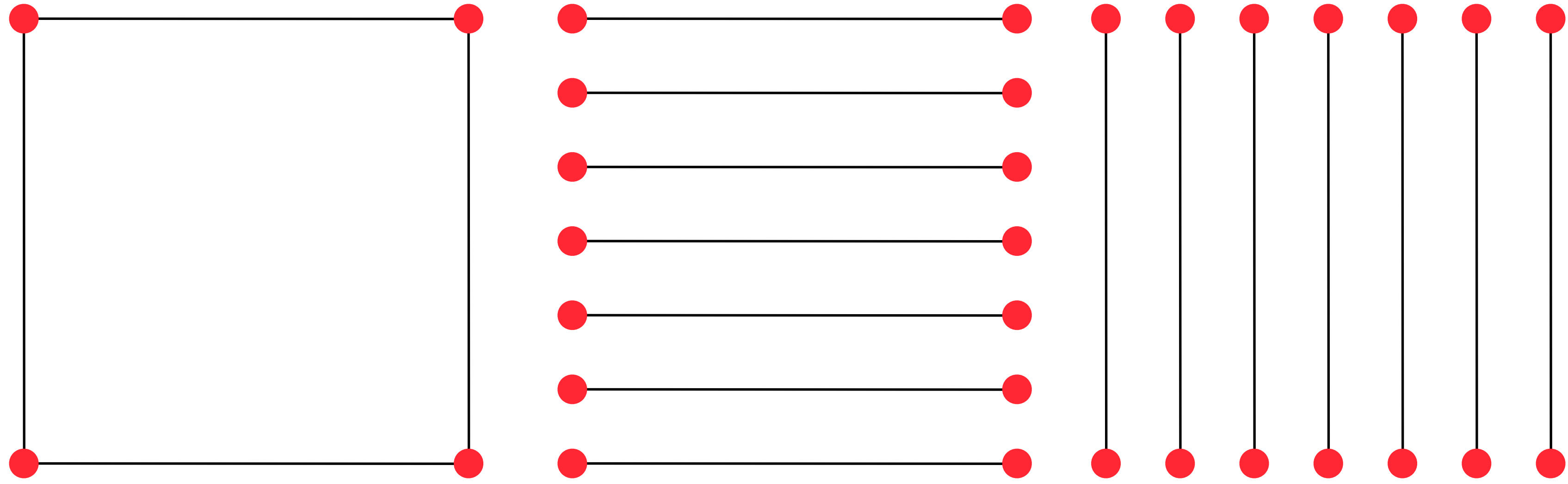
Complexity of Polygon Drawings

- 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

4 Programming with Animations

4.1 Animated Graphics: Principles and History

4.2 Types of Animation

4.3 Programming Animations

4.4 Design of Animations

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

4.5 Game Physics

Literature:

K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004
– Section 4.4 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

Cartoon Animation Process

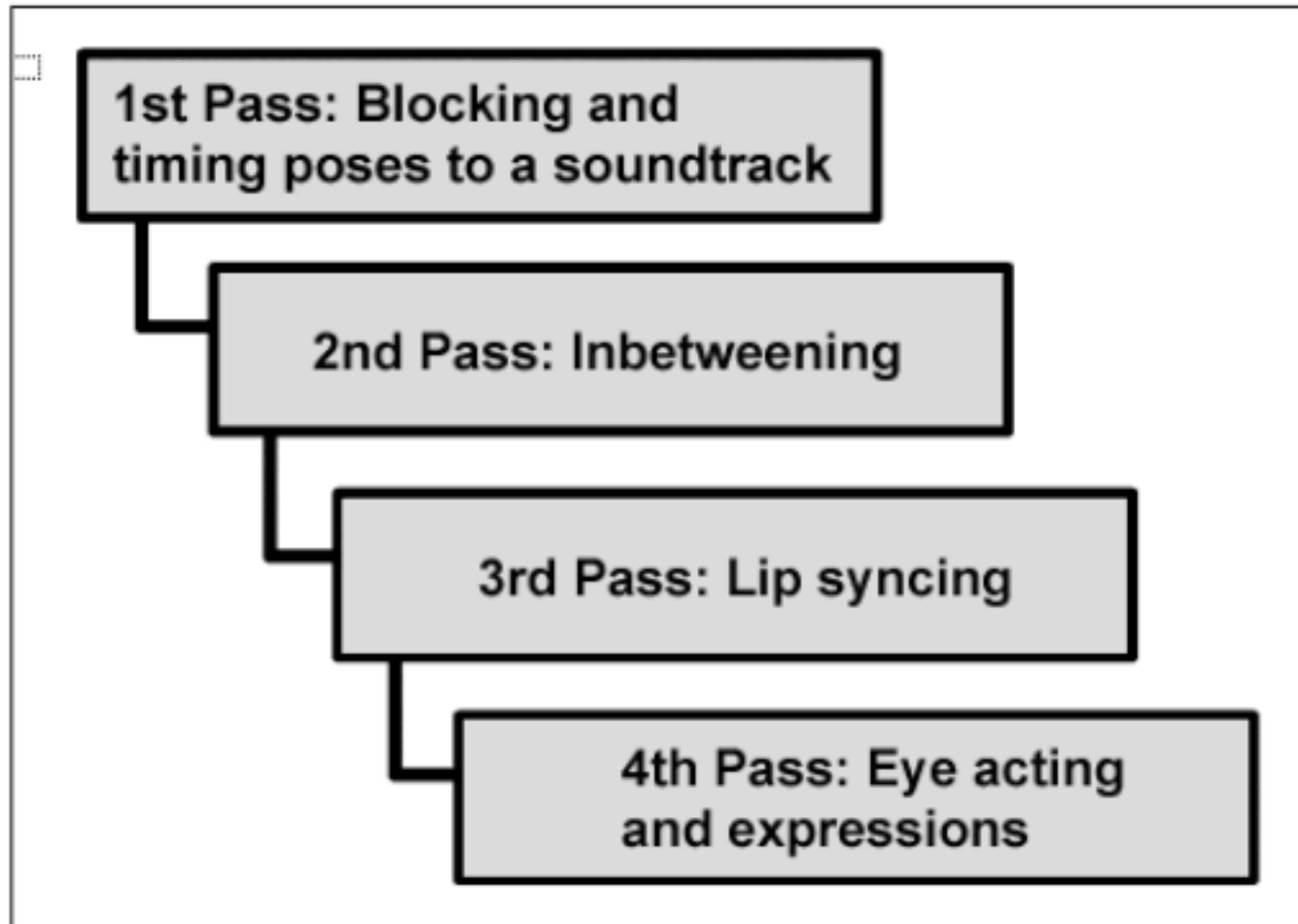
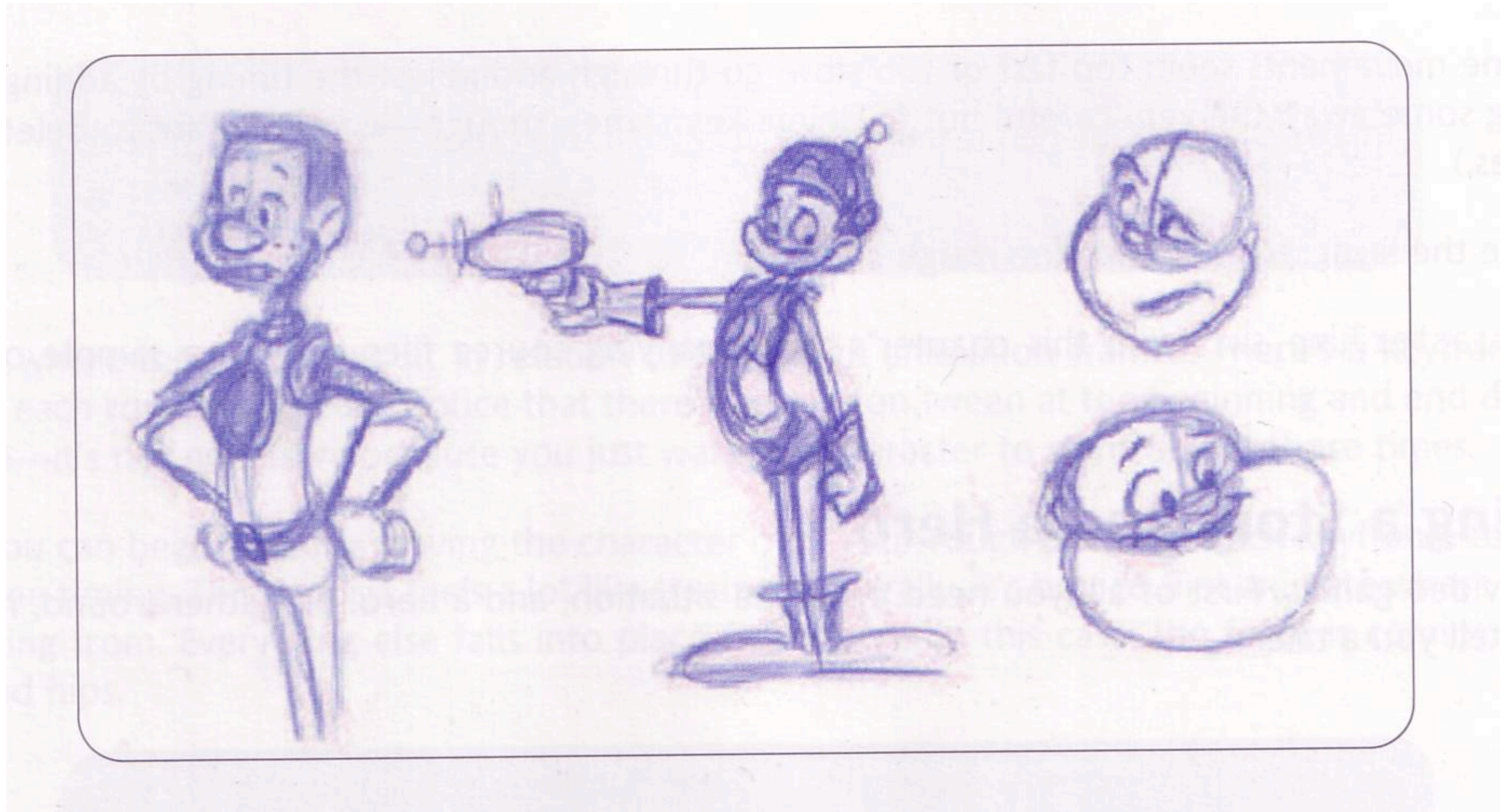


Figure 6-2. The four stages in the animation process

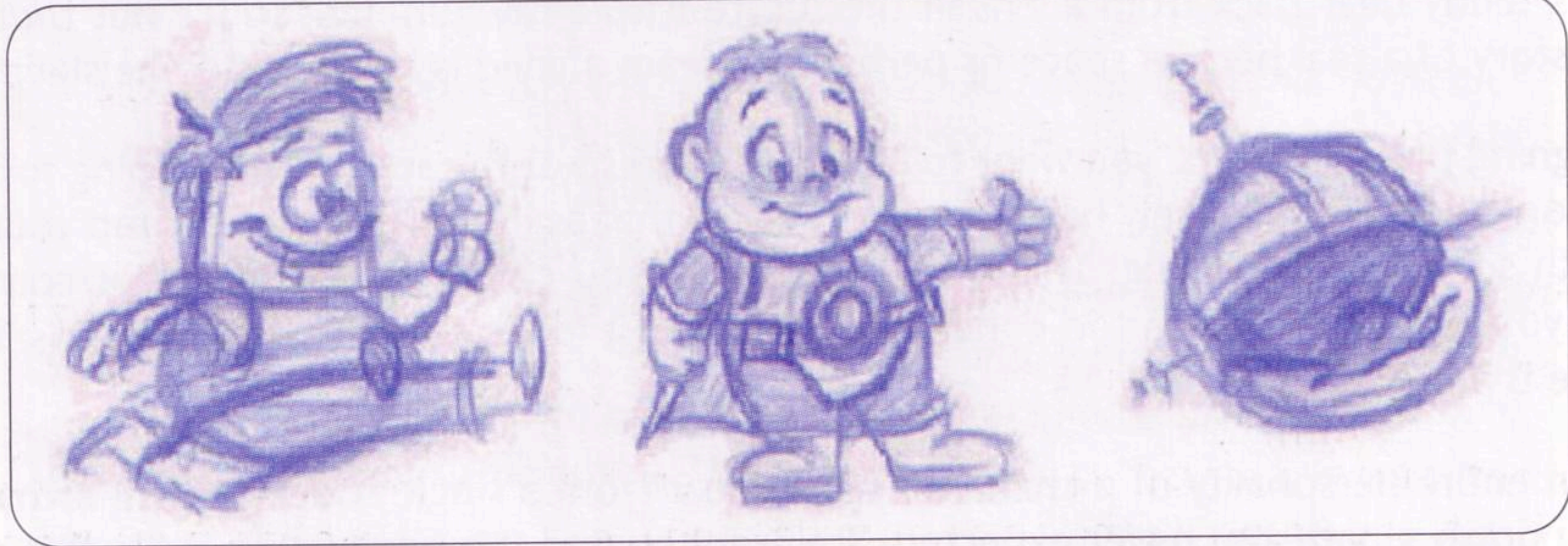
Longer sequences
Soundtrack
Static storyboard

Source: T. Jones

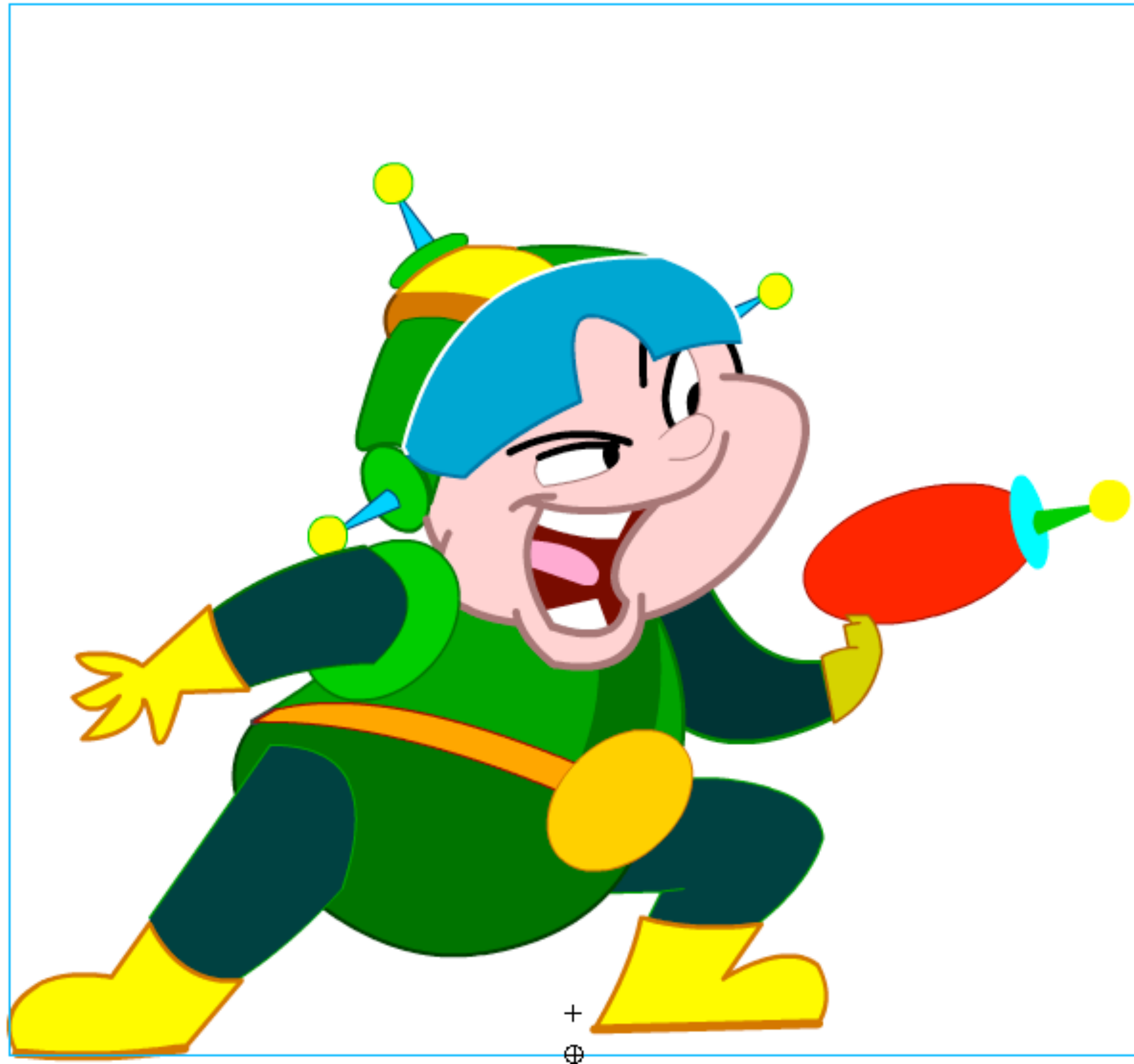
Character Brainstorming (1)



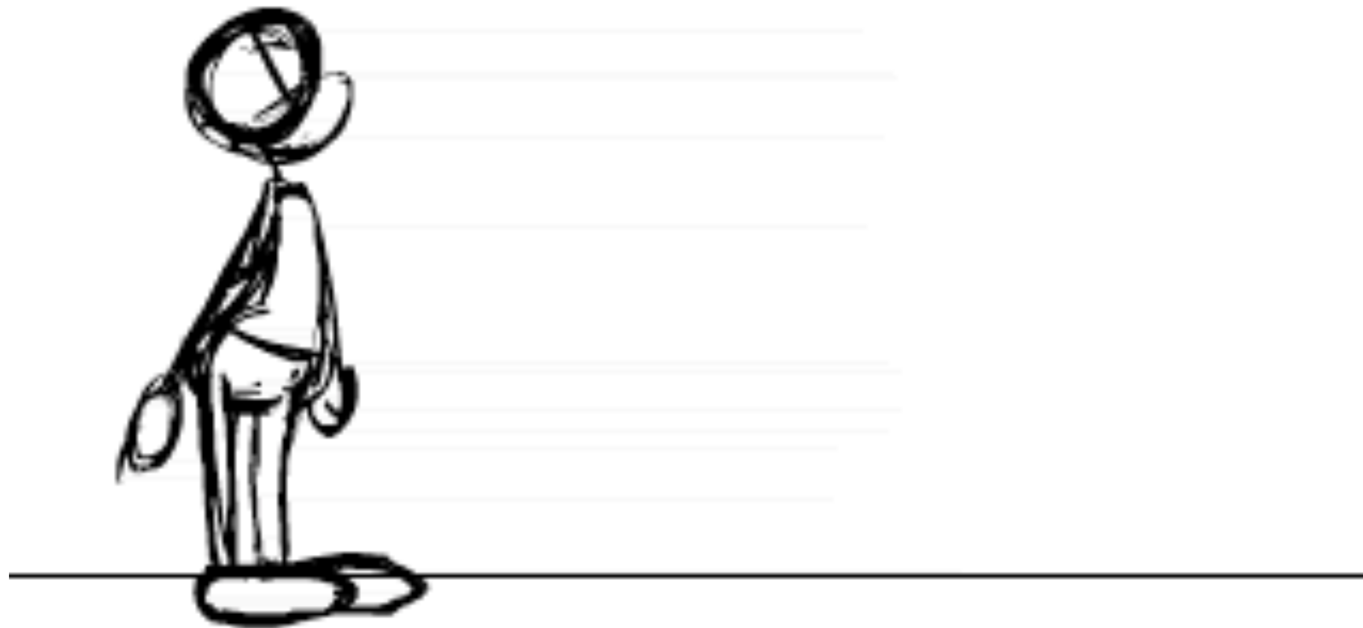
Character Brainstorming (2)



The Final Character

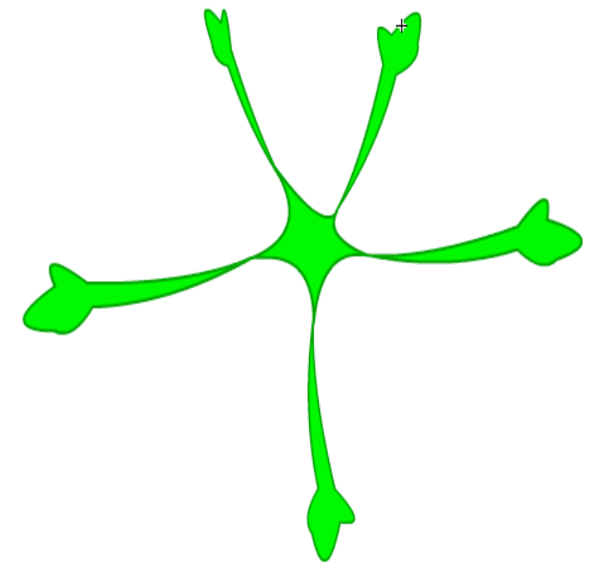


Rough Sketch for “Jump” Animation



The Enemies...

- Lord Notaniceguy’s space slugs...

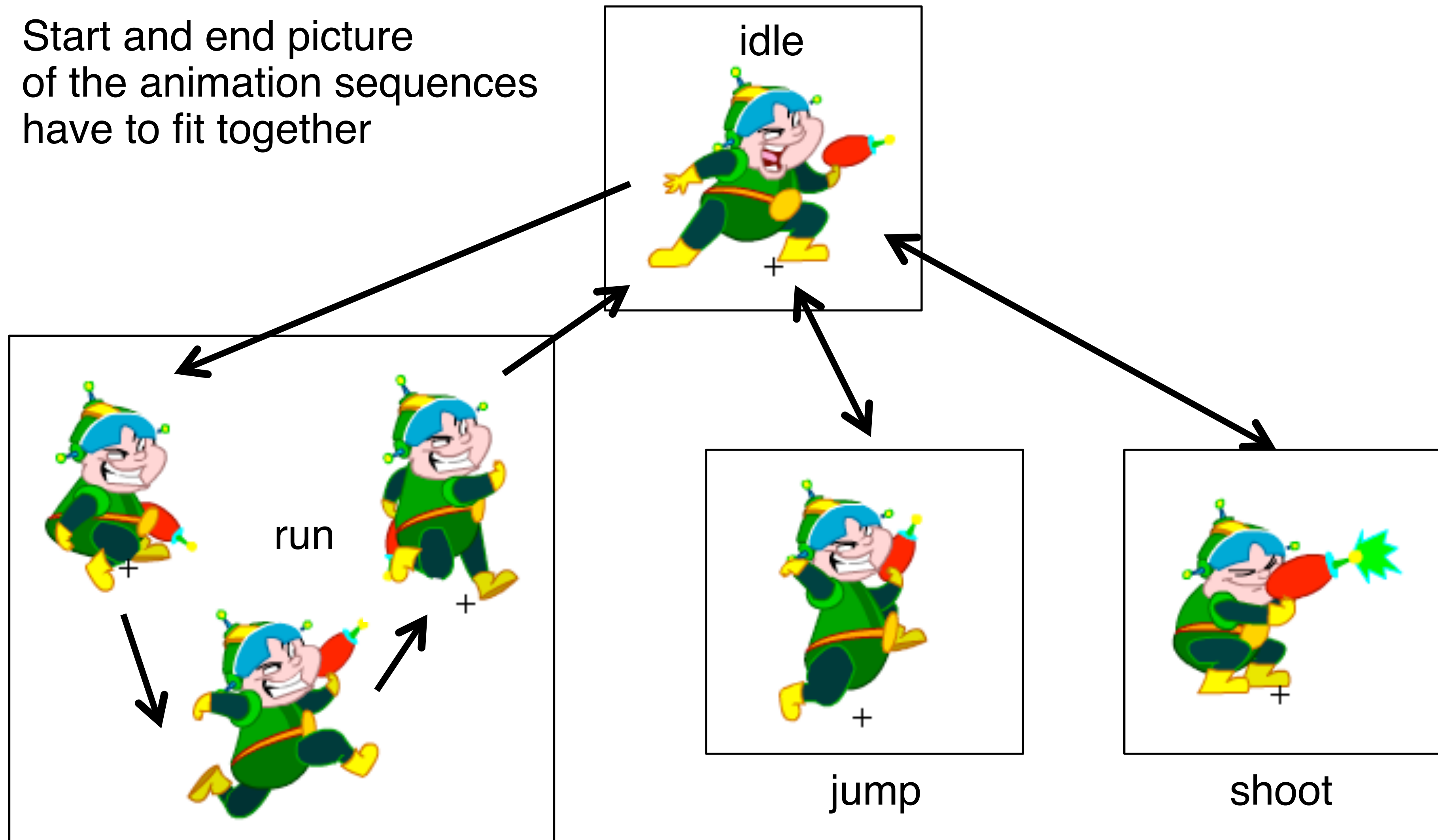


Physics and Animation

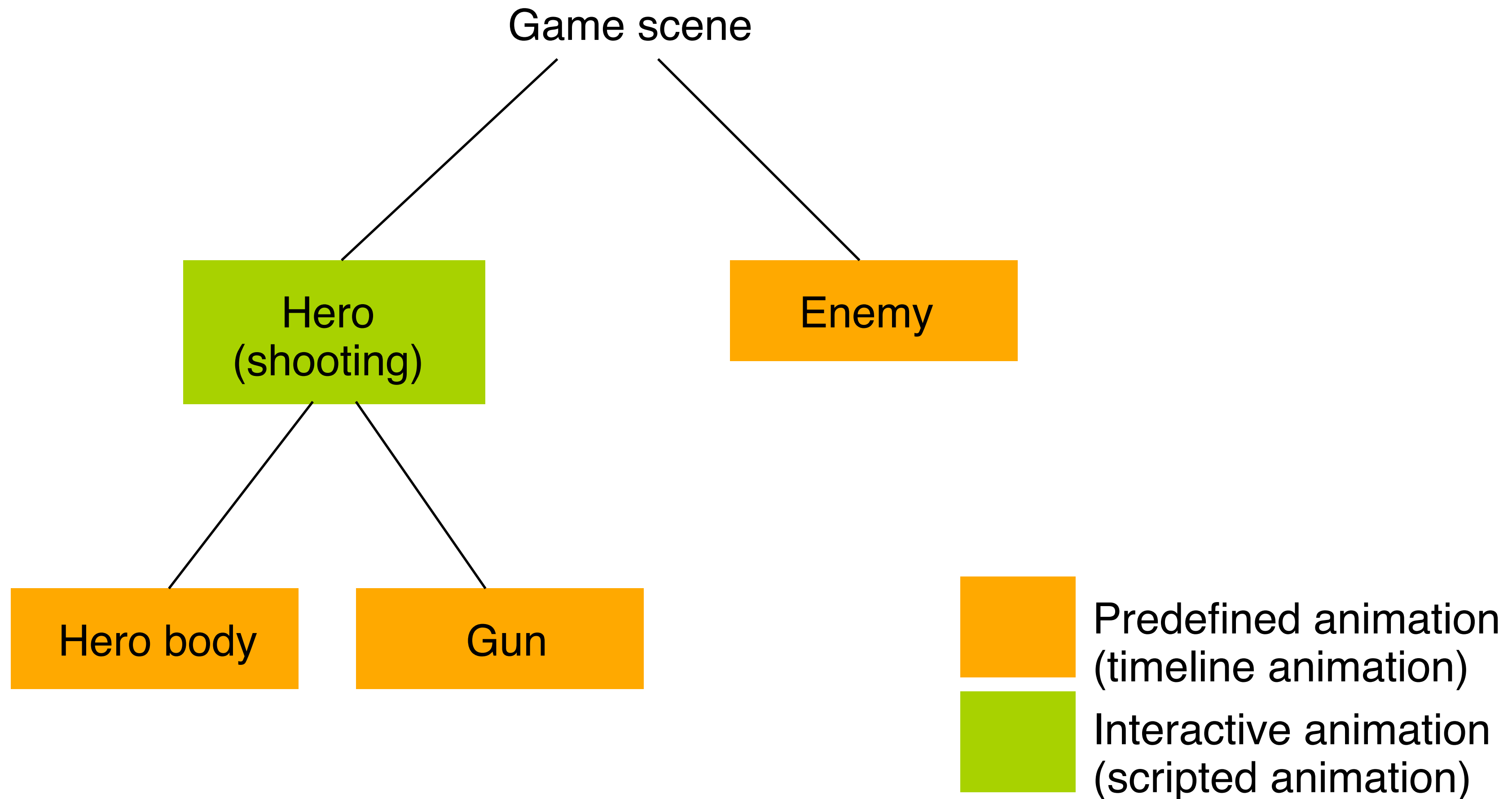
- Jumping character:
  - Trajectory is computed by interactive program
  - *dx* and *dy* values for updating the character’s position
  - Jump may have different width and height depending on user interaction
- Physics is controlled with code, not with timeline animation
  - Reaction to events (including time “ticks”)
- Consequence for movement animations (like jump):
  - Movements “as if staying on the ground”
  - Character design provides one central 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

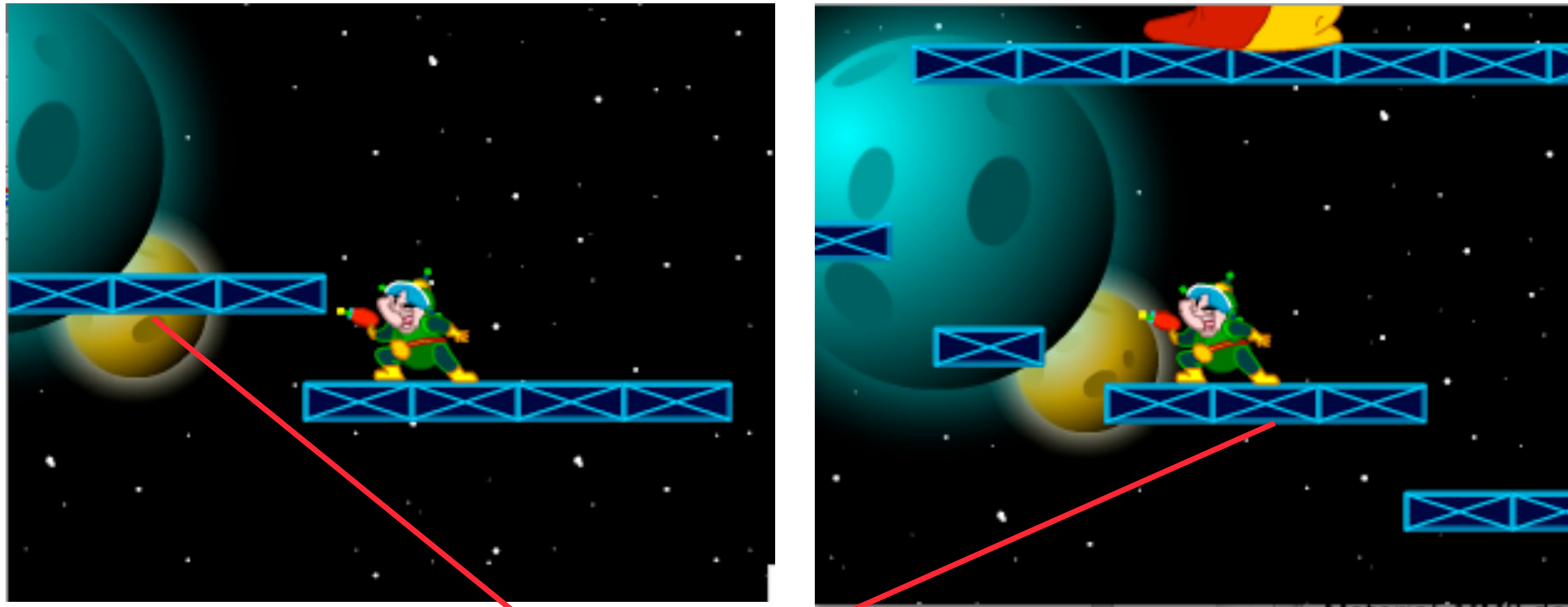
Start and end picture
of the animation sequences
have to fit together



Hierarchical Animations



Parallax/Multiplane Effect



The same platform

- Parallax effect (game programming) / multiplane (animation):
  - Move the background at different (slower) rate than the foreground
  - Creates a sensation of depth

4 Programming with Animations

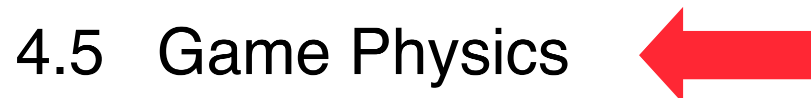
4.1 Animated Graphics: Principles and History

4.2 Types of Animation

4.3 Programming Animations

4.4 Design of Animations

4.5 Game Physics



Literature:

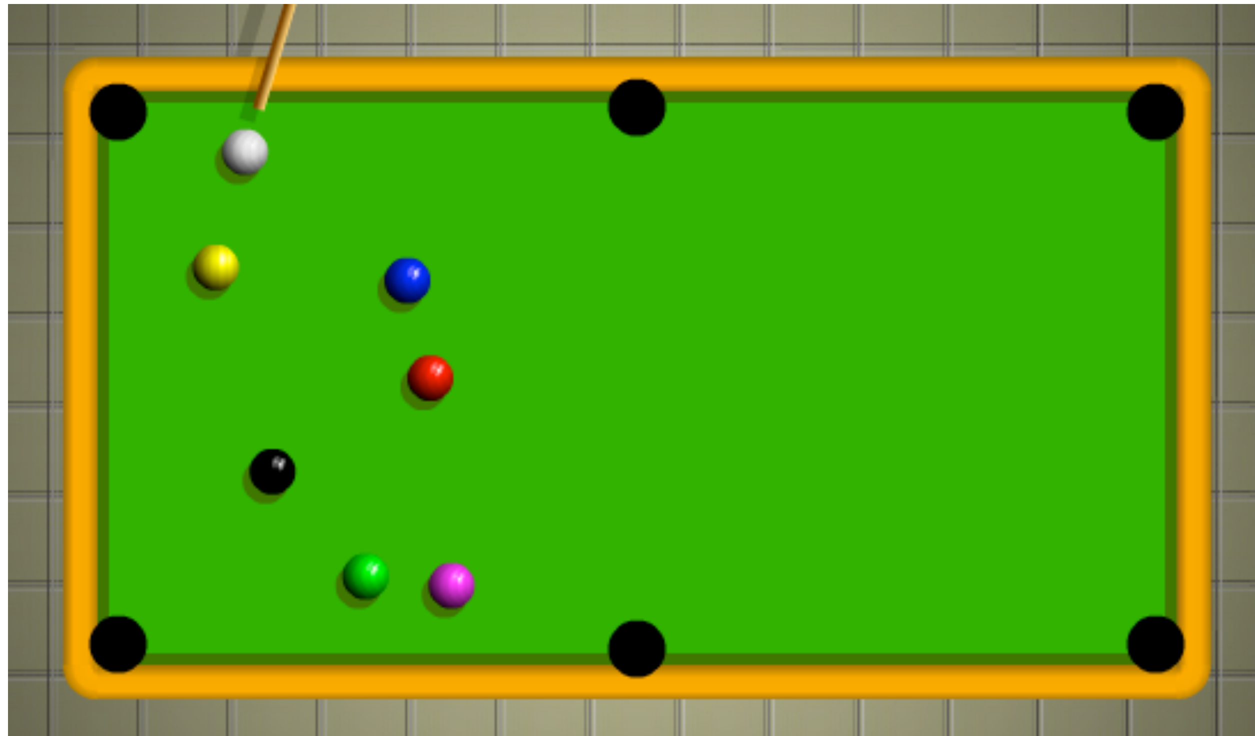
K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004 (chapter 3 by Keith Peters)

K. Peters: Foundation ActionScript 3.0 Animation – Making Things Move!
Apress/Friends of ED 2007

W. Sanders, C. Cumaranatunge: ActionScript 3.0 Design Patterns,
O'Reilly 2007

Billiard-Game Physics

- Typical problem:
  - Two round objects moving at different speeds and angles hitting each other
  - How to determine resulting speeds and directions?
- Example used here:
  - Billiard game (of course)



Flash/ActionScript Code for Moving Ball

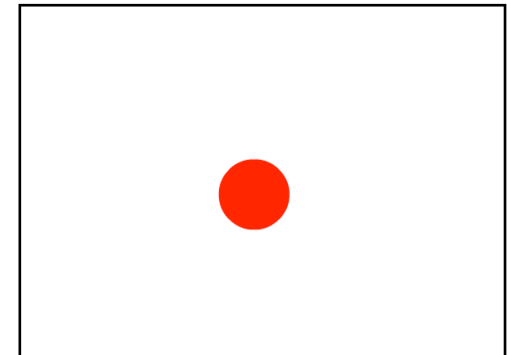
```
public class Bouncing extends Sprite
{
    private var ball:Ball;
    private var vx:Number;  //Horizontal velocity
    private var vy:Number;  //Vertical velocity

    public function Bouncing()
    {
          init();
    }

    private function init():void
    {
          stage.scaleMode = StageScaleMode.NO_SCALE;
          stage.align=StageAlign.TOP_LEFT;

          ball = new Ball();
          ball.x = stage.stageWidth / 2;
          ball.y = stage.stageHeight / 2;
          vx = Math.random() * 50 - 25;
          vy = Math.random() * 50 - 25;
          addChild(ball);
          addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
...
```

Flash/ActionScript Code for Ball Class

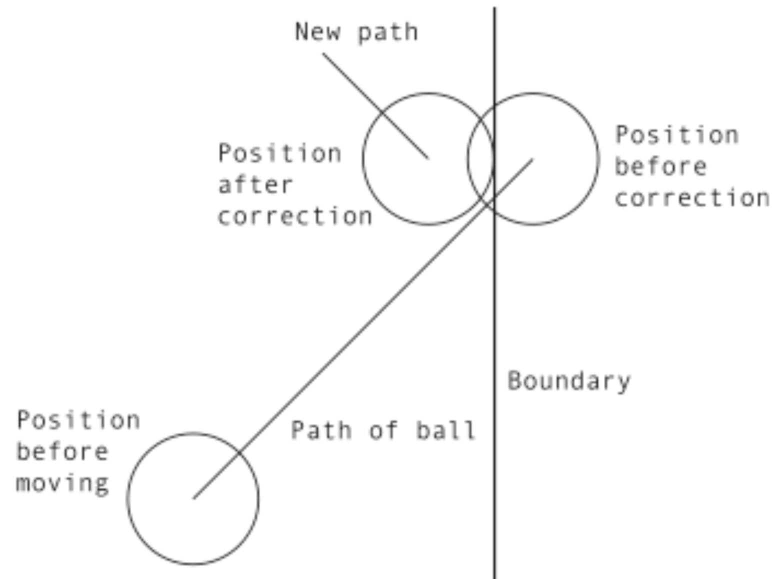
```
package {
  import flash.display.Sprite;

  public class Ball extends Sprite {
     public var radius:Number;
     private var color:uint;
     public var vx:Number = 0;
     public var vy:Number = 0;

     public function Ball(radius:Number=40,
      color:uint=0xff0000) {
          this.radius = radius;
          this.color = color;
          init();
     }
     public function init():void {
          graphics.beginFill(color);
          graphics.drawCircle(0, 0, radius);
          graphics.endFill();
     }
  }
}
```

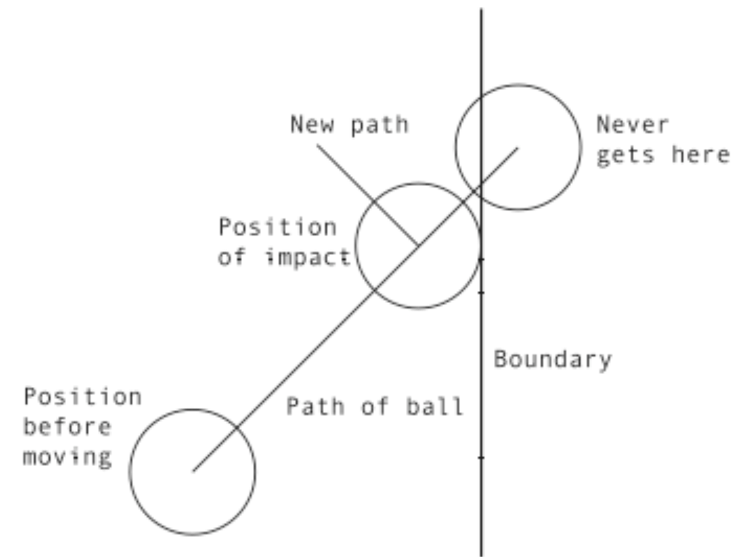
Simple Wall-to-Wall Bouncing

```
…       private function onEnterFrame(event:Event):void
        {
                ball.x += vx;
                ball.y += vy;
                var left:Number = 0;
                var right:Number = stage.stageWidth;
                var top:Number = 0;
                var bottom:Number = stage.stageHeight;

                if(ball.x + ball.radius > right)
                {
                        ball.x = right - ball.radius;
                        vx *= -1;
                }
                else if(ball.x - ball.radius < left)
                {
                        ball.x = left + ball.radius;
                        vx *= -1;
                }
                if(ball.y + ball.radius > bottom)
                {
                        ball.y = bottom - ball.radius;
                        vy *= -1;
                }...
        }
}
```



Model and Reality





Energy Loss

- Bouncing always takes away some part of energy
  - Set “bouncing factor” to value smaller than 1

```
private var bounce:Number = 0.8;
...
private function onEnterFrame(event:Event):void
{
	...

	if(ball.x + ball.radius > right)
	{
		ball.x = right - ball.radius;
		vx *= -bounce;
	}
	else if(ball.x - ball.radius < left)
	{
		ball.x = left + ball.radius;
		vx *= -bounce;
	} ...
}
```

Bounce and Friction

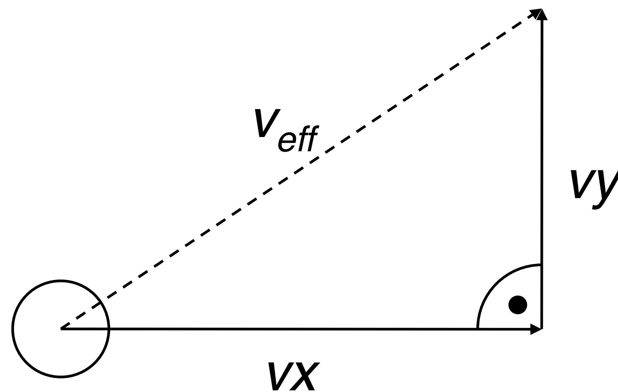
- The surface of the table always absorbs some part of the energy and slows down the ball
  - Reduce velocity by some factor each frame

```
private var friction:Number = 0.99;
...
private function onEnterFrame(event:Event):void
{
        ...

        if(ball.x + ball.radius > right)
        {
                ball.x = right - ball.radius;
                vx *= -bounce;
        }
        else if(ball.x - ball.radius < left)
        {
                ball.x = left + ball.radius;
                vx *= -bounce;
        }...
        vx *= friction;
        vy *= friction;
}
```

Minimum Speed

```
...
        vx *= friction;
        vy *= friction;
        var effspeed:Number = Math.sqrt(vx*vx+vy*vy);
        if (effspeed<MINSPEED) {
                vx = 0;
                vy = 0;
        }
}
```

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



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

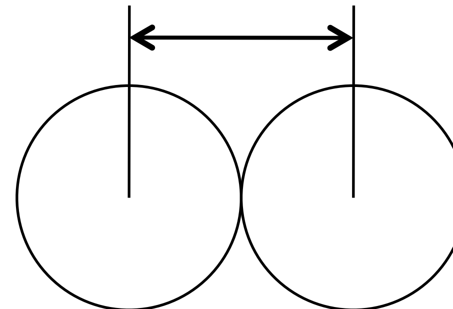
Collision Detection Between Balls (1)

- Two balls are able to collide.
  - Collision detection as a separate entity from the two balls
  - E.g. as part of global **`onEnterFrame`** handler

```
private function onEnterFrame(event:Event):void
{
        ball0.x += ball0.vx;
        ball0.y += ball0.vy;
        ball1.x += ball1.vx;
        ball1.y += ball1.vy;
        checkCollision(ball0, ball1);
        checkWalls(ball0);
        checkWalls(ball1);
}
```

Collision Detection Between Balls (2)

```
private function checkCollision(ball0:Ball,
      ball1:Ball):void
{
     var dx:Number = ball1.x - ball0.x;
     var dy:Number = ball1.y - ball0.y;
     var dist:Number = Math.sqrt(dx*dx + dy*dy);
     if(dist < ball0.radius + ball1.radius)
     {
          Collision detected ...
     }
}
```

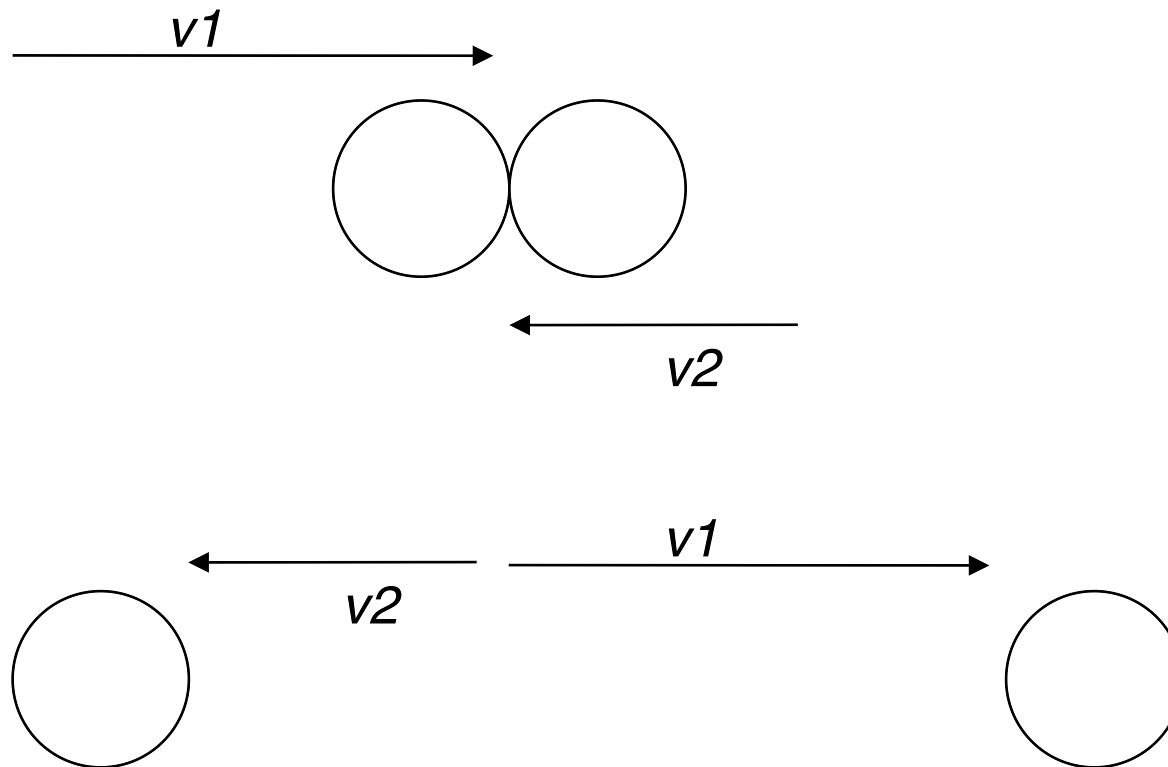


Physics: Speed, Velocity, Mass, Momentum

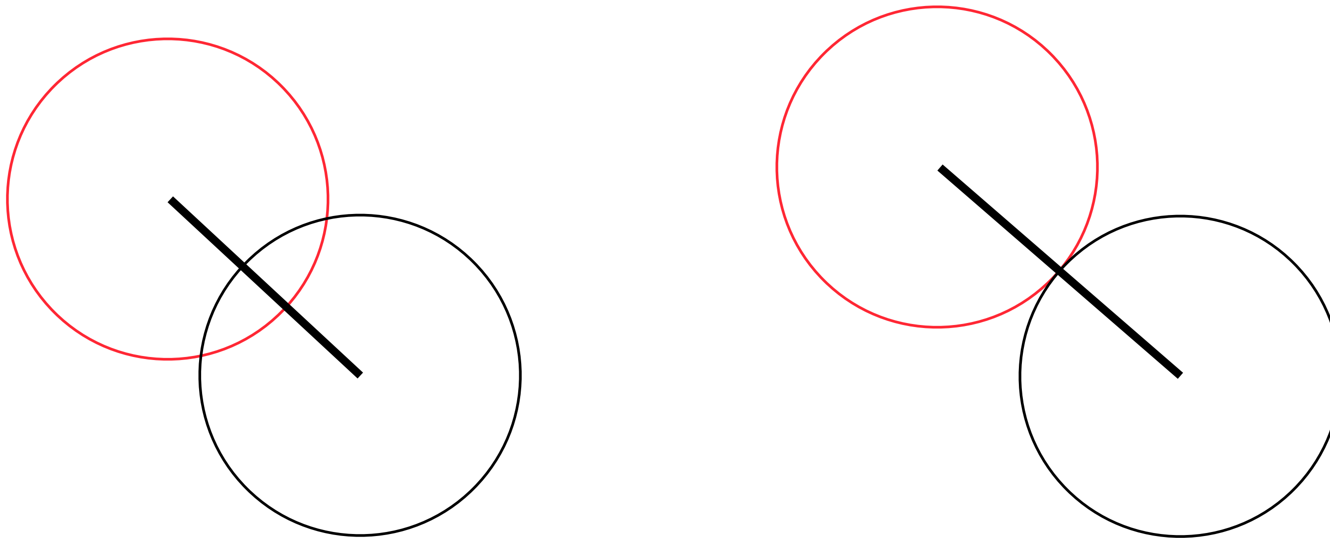
- *Speed:*
  - How fast is something moving (length/time)
- *Velocity:*
  - Vector describing movement: *speed* + *direction*
- *Mass:*
  - Basic property of object, depending on its material, leads under gravity to its *weight*
- *Momentum (dt. Impuls):*
  - Mass x Velocity
- *Principle of Conservation 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

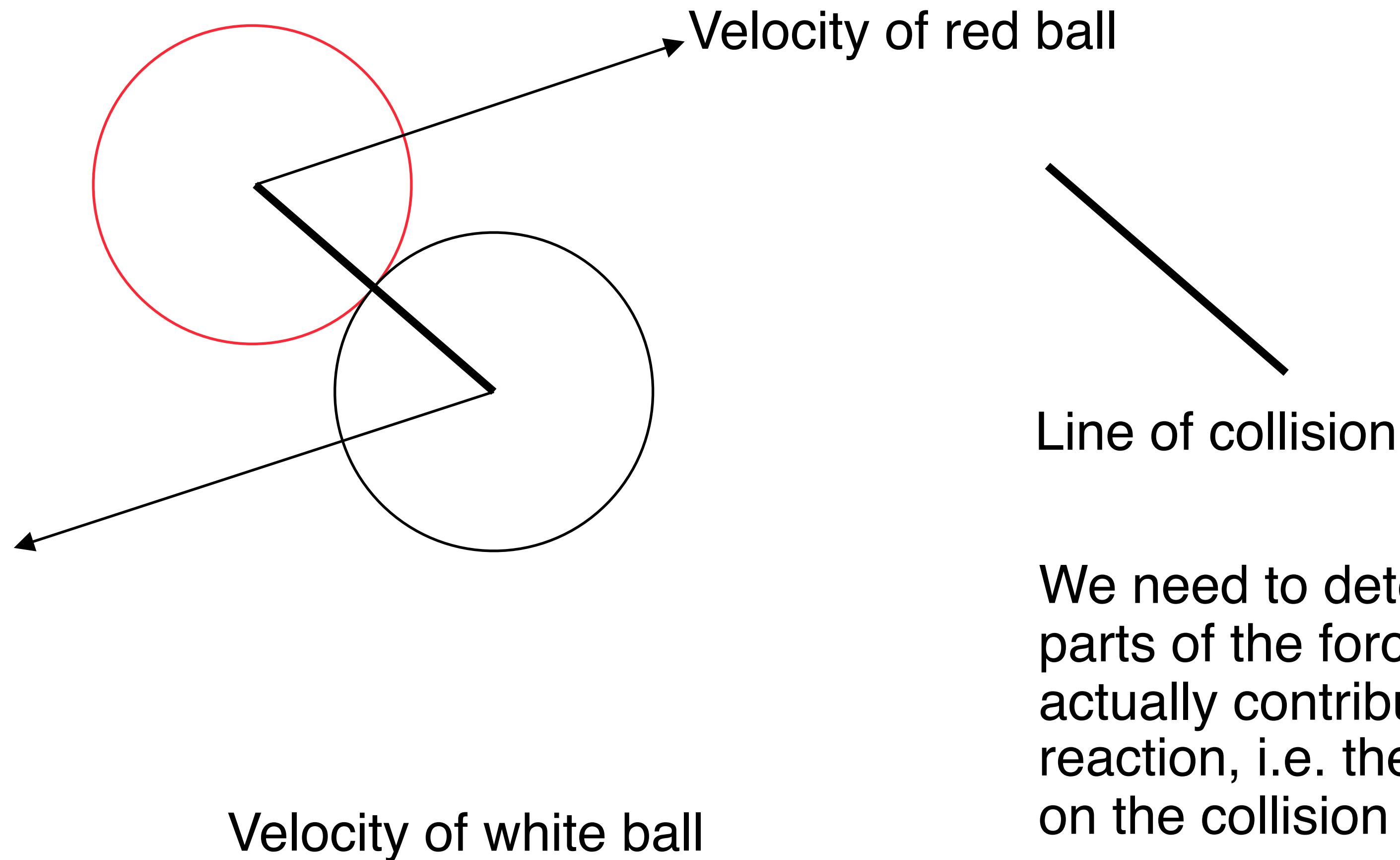


Placing Balls At Collision Time



This is a simplification compared to the actual physical laws.

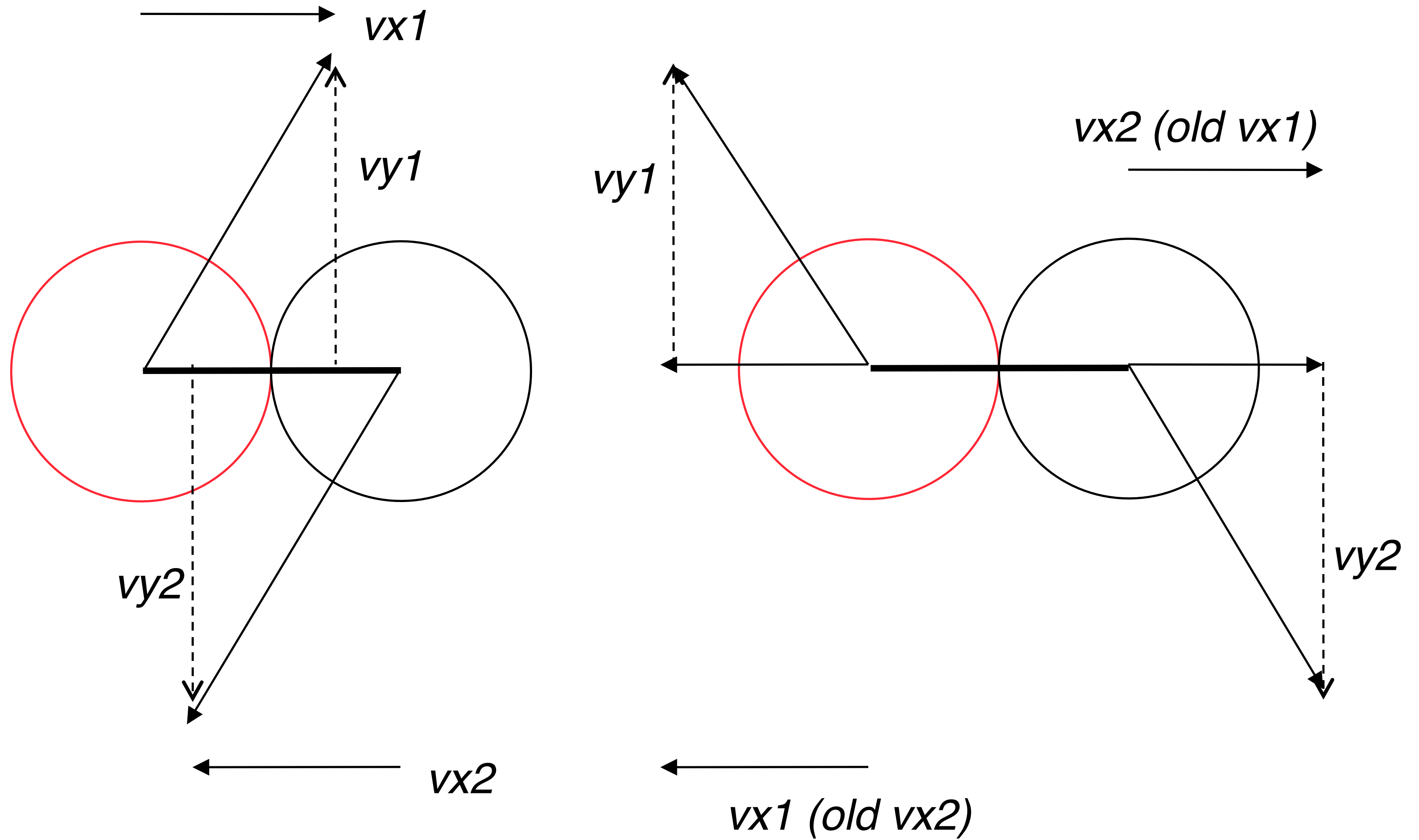
Physics of Collision, Step 1



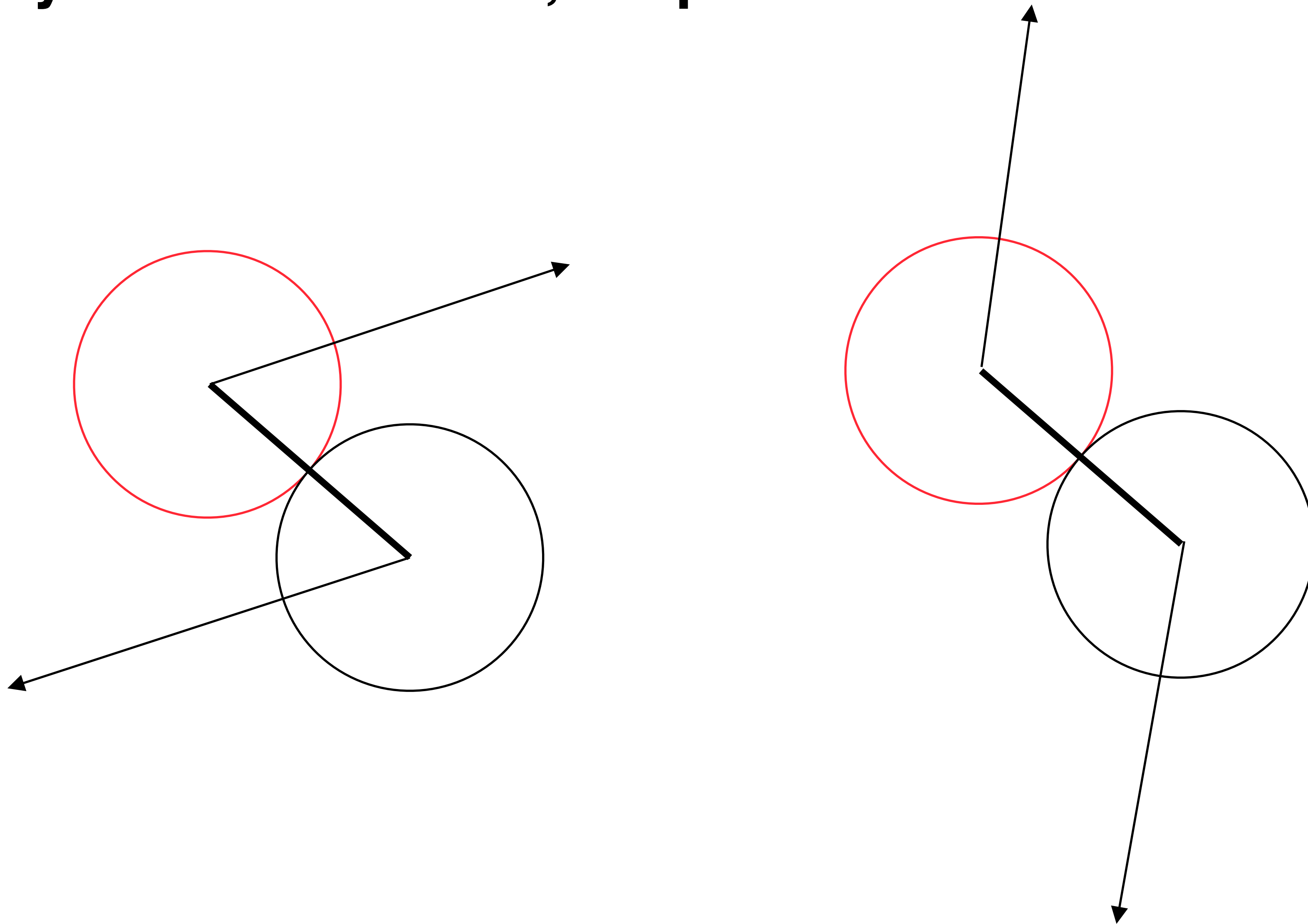
Line of collision

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

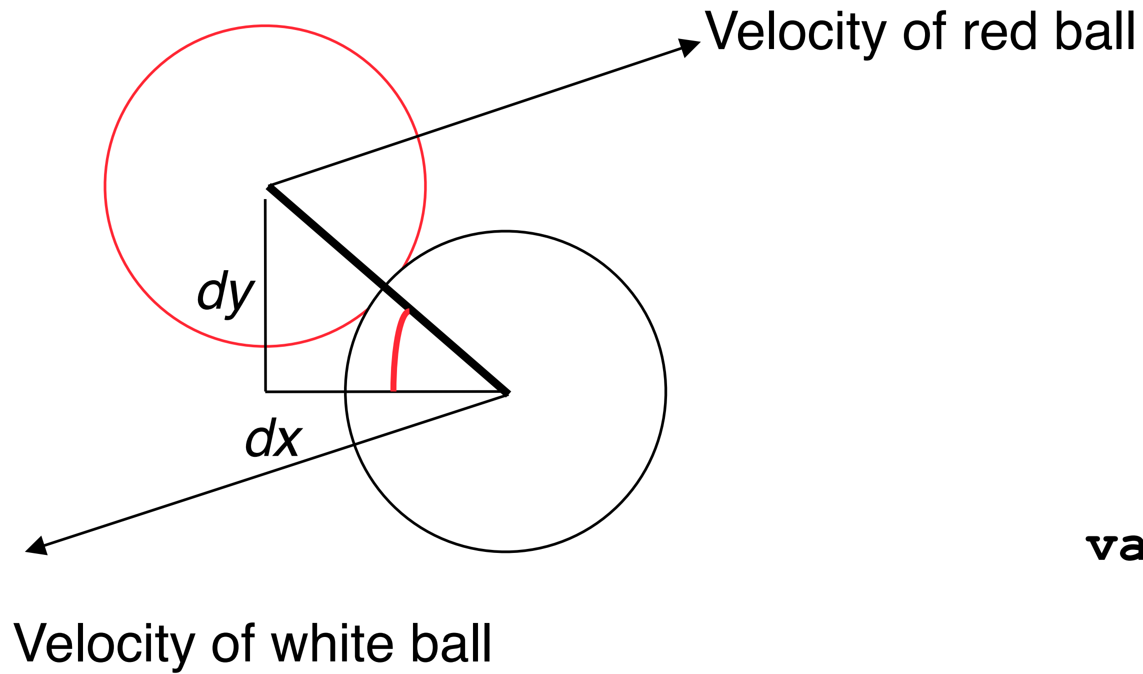
Physics of Collision, Step 2



Physics of Collision, Step 3

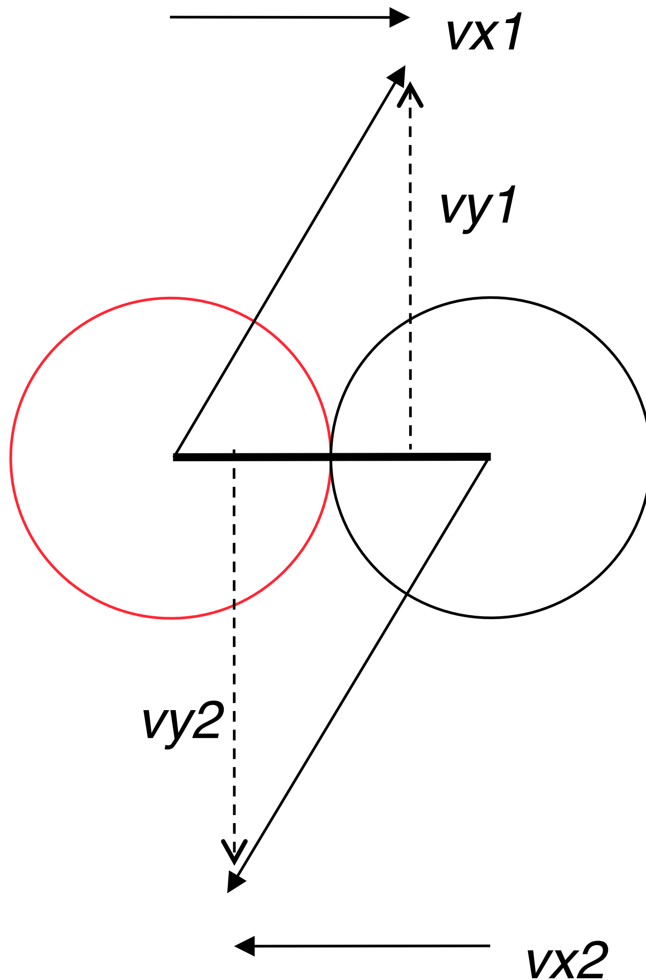


Computing Step 1




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

Computing Step 2, Part 1



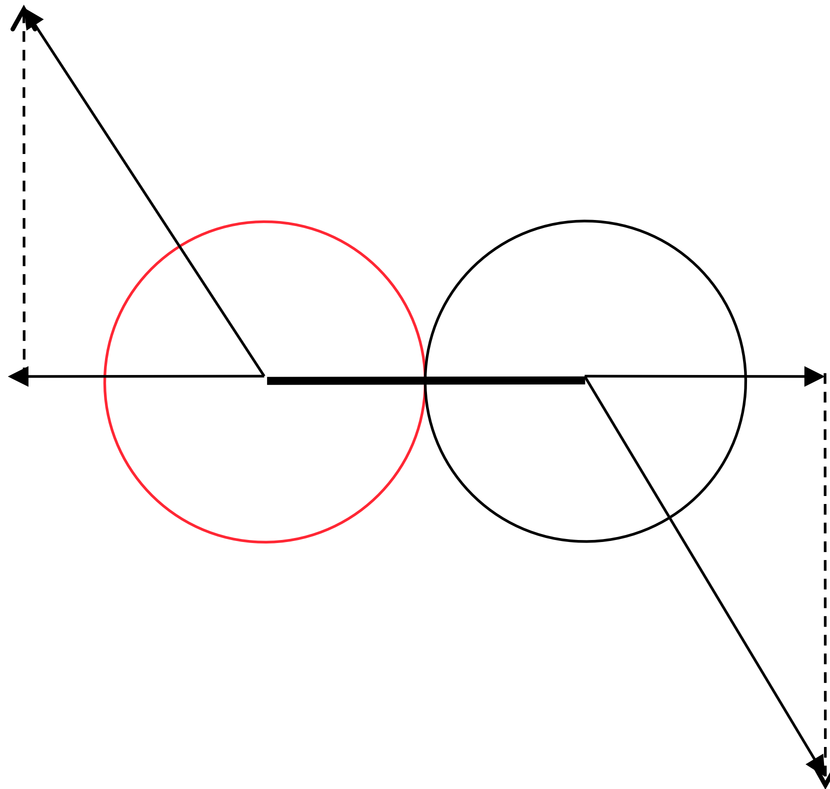
Counterclockwise rotation of vector (x, y):

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

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

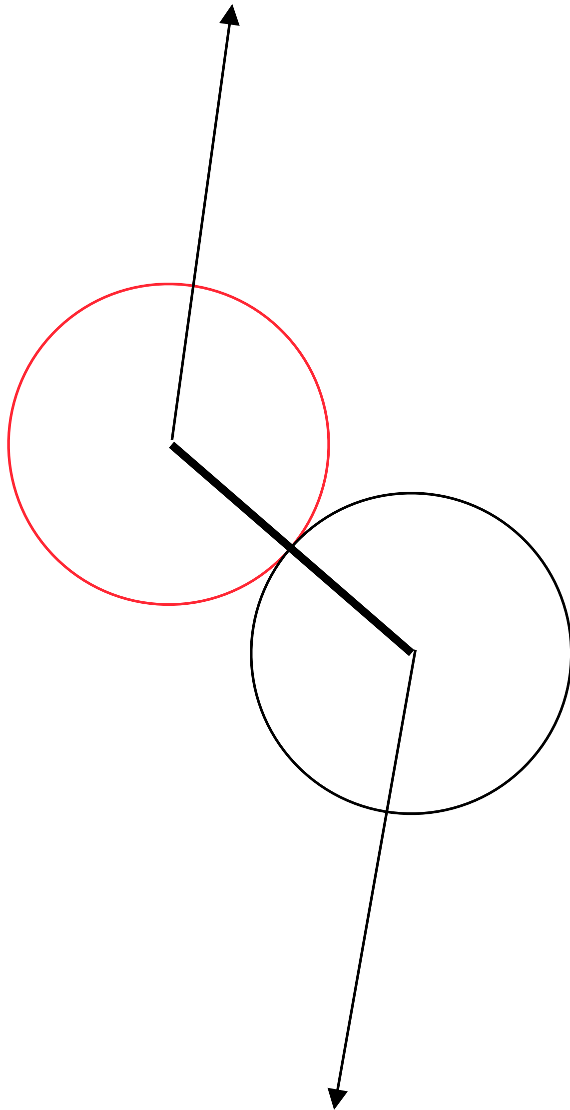
```
var angle =
  Math.atan2(dy,dx);
var cosa = Math.cos(angle);
var sina = Math.sin(angle);
var vx1 =
  cosa*redBall.vx +
  sina*redBall_.vy;
var vy1 =
  cosa*redBall.vx -
  sina*redBall.vy;
... vx2, vy2
```

Computing Step 2, Part 2



```
...
var tempvx = vx1;
vx2 = vx1;
vx1 = tempvx;
```

Computing Step 3



Clockwise rotation of vector (x, y):

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

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

```
redBall.vx =
  cosa*vx1 - sina*vy1;
redBall.vy =
  cosa*vy1 + sina*vx1;
whiteBall.vx =
  cosa*vx2 - sina*vy2;
whiteBall.vy =
  cosa*vy2 + sina*vx2;
```

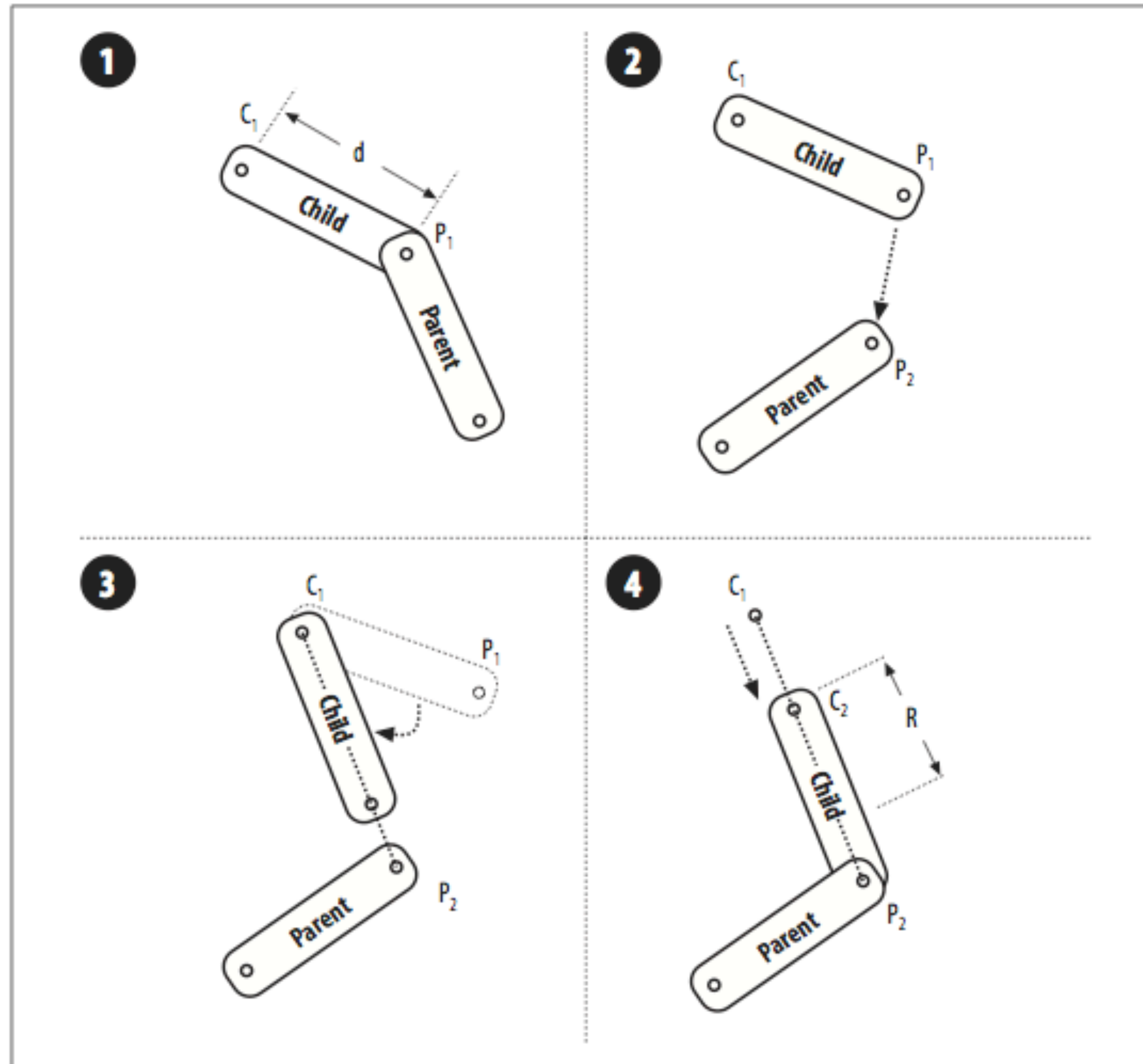
Forward and Inverse Kinematics

- ***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.

Kinematic Pairs

- A *kinematic pair* consists of two rigid objects connected by a joint
  - *Parent* segment and *child* segment
  - Position of the parent segment, connection position, angle of child segment
- Parent segment pulls child segment
  - Child segment has to follow
- Parent segment takes new position (and orientation)
  - leads to new position of connection point for child segment
  - child segment orients towards new position
  - if necessary, child segment moves to joint parent at new connection position

Four-Step Pulling Process for Kinematic Pairs



Sanders/Cumaratunge

Example: Snake Animation

Code may use the *Composite* design pattern

```
override public function update():void {
    var myParent:Composite = this.getParent();
    var parentLoc:Point = new Point(myParent.x, myParent.y);
    var myLoc:Point = new Point(this.x, this.y);
    // rotate to orient to parents new location
    var tempPoint:Point = parentLoc.subtract(myLoc);
    var angle:Number = Math.atan2(tempPoint.y, tempPoint.x);
    this.rotation = angle * 180 / Math.PI;
    // move to maintain distance
    var currentDistance:Number = Point.distance(parentLoc, myLoc);
    var myNewLoc:Point = Point.interpolate(myLoc, parentLoc, segLen /
         currentDistance);
    this.x = myNewLoc.x;
    this.y = myNewLoc.y;
}
```