3 Introduction to Computer Game Programming

3.1 Logic-Based Games

3.2 Graphical Design of Game Characters

- Optimizing Vector Graphics
- Principles of Animation
- Creating a Story and a Hero

3.3 Simulation-Based Games

3.4 Interaction and Sound

Literature:

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

Section 3.2 based on book chapter 2 by ***Brad Ferguson***

Source code for all examples at www.friendsofed.com

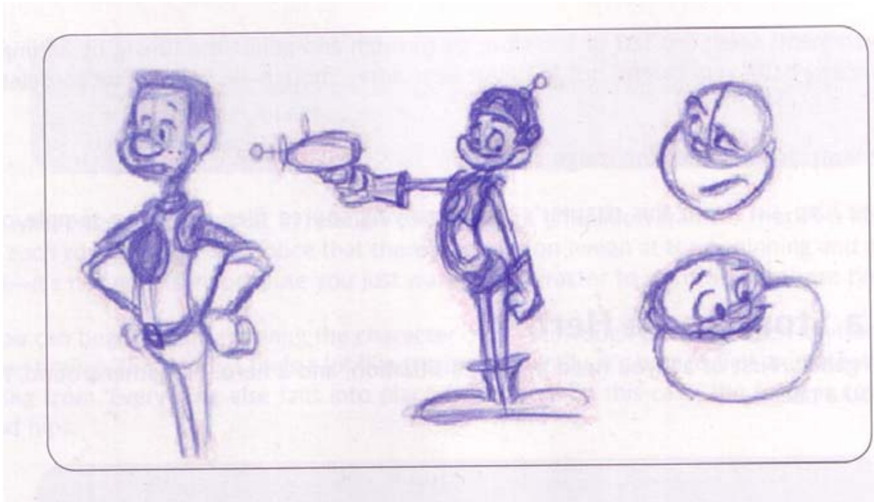
The Story and Situation for a New Game

"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.

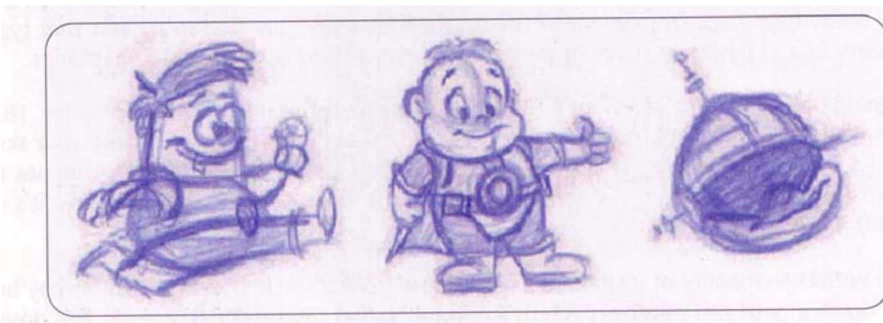
Immediately Space Kid knew there could only be one ruthless and vile enemy capable of committing such an atrocity: his longtime arch nemesis, Lord Notaniceguy.

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."

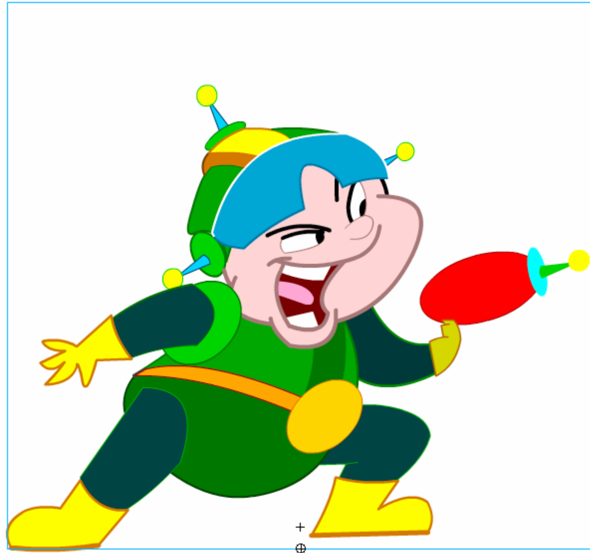
Character Brainstorming (1)



Character Brainstorming (2)



The Final Character



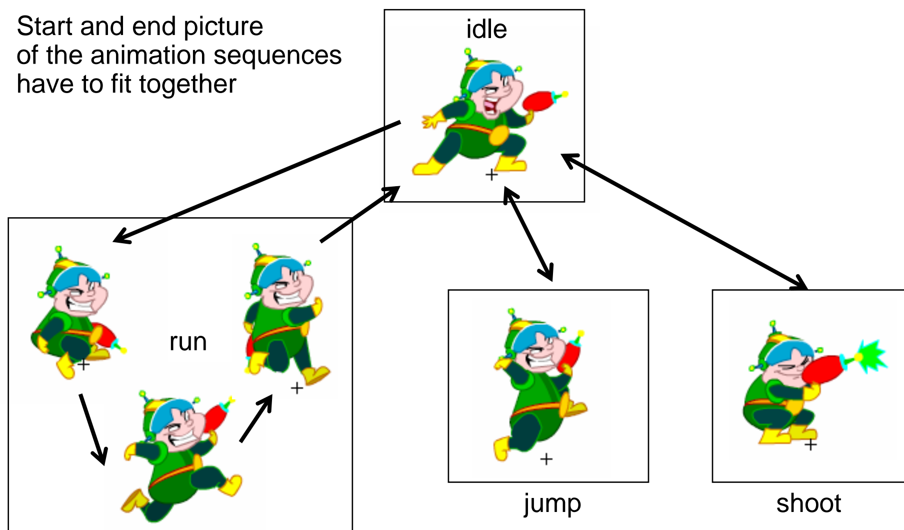
The Design Process

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

Rough Sketch for “Run” Animation

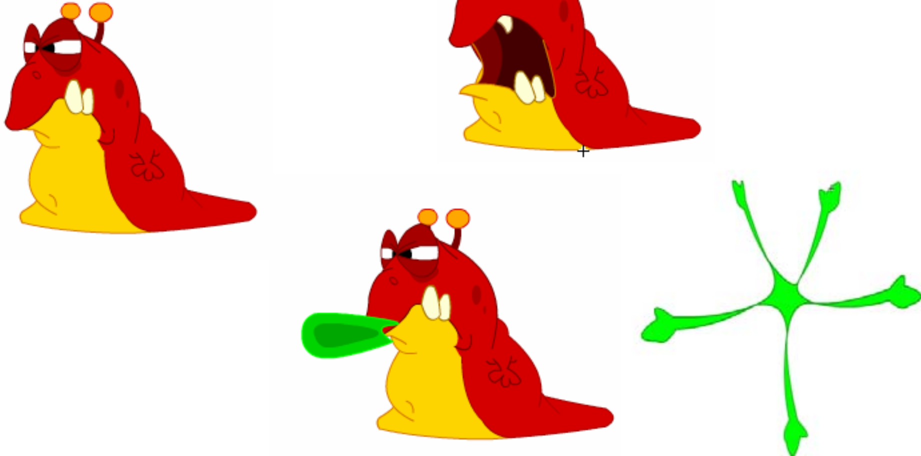
Continuity of Animation Sequences

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



The Enemies...

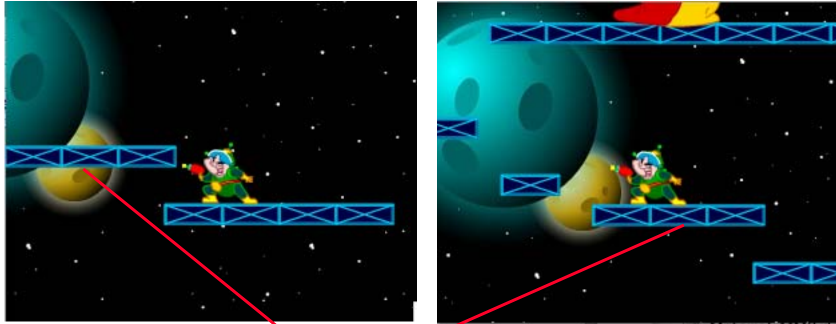
- Lord Notaniceguy's space slugs...



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 (not in the example game) height depending on user interaction
- Physics is controlled with code, not with animation
- Consequence für 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, ...

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

3 Introduction to Computer Game Programming

3.1 Logic-Based Games
3.2 Graphical Design of Game Characters
3.3 Simulation-Based Games
3.4 Interaction and Sound

Literature:

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

Section 3.2 based on book chapter 3 by ***Keith Peters***

Source code for all examples at www.friendsofed.com

Games Simulating the Real World

- Games that mirror real- (or more or less possible-)world objects
  - Objects moving according to physical laws (or simplified versions of them)
  - Example:
    - » Spacekid's jumps in last section
- Characteristic properties for simulation-based games:
  - System reaction depends on timing of user action (real-time game)
  - System reaction depends on "analog" aspects of user action like speed of mouse movement (representing force)
- Consequence for internal game design:
  - Mirror (or abstract from) physics
- Practical consequence for (Flash) animations:
  - Boost frame rate (e.g. 60 frames per second)

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)

Ball-to-Wall Bounces, Simple Version

```
BALL_DIAMETER = 20;
BALL_RADIUS = BALL_DIAMETER/2;
TOP = table_mc._y-table_mc._height/2+BALL_RADIUS;
BOTTOM = table_mc._y+table_mc._height/2-BALL_RADIUS;
LEFT = table_mc._x-table_mc._width/2+BALL_RADIUS;
RIGHT = table_mc._x+table_mc._width/2-BALL_RADIUS;
BOUNCE = -1;
whiteBall_mc.vx = Math.random()*5+2; // horizontal velocity
whiteBall_mc.vy = Math.random()*5+2; // vertical velocity
whiteBall_mc.onEnterFrame = ballMove;
function ballMove() {
  this._x += this.vx;
  this._y += this.vy;
  if (this._x>RIGHT) {
     this._x = RIGHT; this.vx *= BOUNCE;
  } else if (this._x<LEFT) {
     this._x = LEFT; this.vx *= BOUNCE;
  }
  if (this._y>BOTTOM) {
     this._y = BOTTOM; this.vy *= BOUNCE;
  } else if (this._y<TOP) {
     this._y = TOP; this.vy *= BOUNCE;
  }
}
```


Bounce and Friction

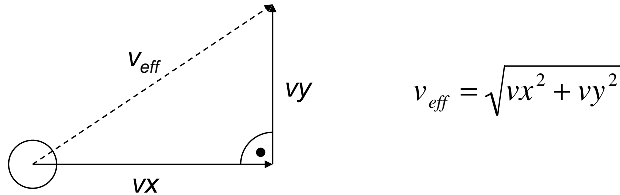
- Bouncing always takes away some part of energy
  - Set BOUNCE to factor smaller than 1
- The surface of the table always absorbs some part of the energy and slows down the ball
  - Reduce velocity by some factor each frame
  - Constant DAMP

```
BALL_DIAMETER = ...
BOUNCE = -0.6;
DAMP = 0.99;
...
function ballMove() {
  this.vx *= DAMP;
  this.vy *= DAMP:
  this._x += this.vx;
  this._y += this.vy;
  if (this._x>RIGHT) {
     ...
  }
  ...
}
```

Minimum Speed

```
MINSPEED = .1;
...
function ballMove() {
      ...
      this.speed = Math.sqrt(this.vx*this.vx+this.vy*this.vy);
      if (this.speed<MINSPEED) {
            this.vx = 0;
            this.vy = 0;
            delete this.onEnterFrame;
      }
}
```

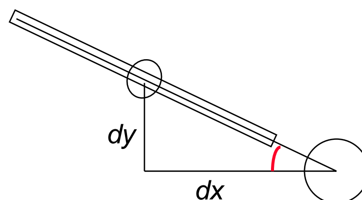
- How to calculate the effective speed out of x and y velocities?



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

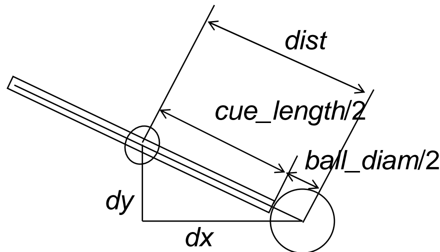
Cue Stick, Aiming

```
stick_mc.onEnterFrame = aim;
function aim() {
  var dx = whiteBall_mc._x-_xmouse;
  var dy = whiteBall_mc._y-_ymouse;
  angle = Math.atan2(dy, dx);
  this._rotation = angle*180/Math.PI;
  this._x = _xmouse;
  this._y = _ymouse;
}
```



Shooting

```
function shoot() {
  this._x = _xmouse;
  this._y = _ymouse;
  this.vx = this._x-this.oldx;
  this.vy = this._y-this.oldy;
  this.oldx = this._x;
  this.oldy = this._y;
  var dx = whiteBall_mc._x-this._x;
  var dy = whiteBall_mc._y-this._y;
  var dist = Math.sqrt(dx*dx+dy*dy);
  if (dist<110) {
    whiteBall_mc.vx = this.vx;
    whiteBall_mc.vy = this.vy;
    whiteBall_mc.onEnterFrame
      = ballMove;
    this.onEnterFrame = aim;
  }
}
```



$$dist = \sqrt{dx^2 + dy^2}$$

cue_length = 200
ball_diam = 20

Switching Between Modes of Objects

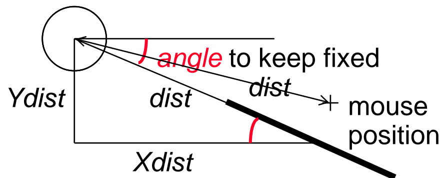
- The ball switches between two states:
  - "moving": **onEnterFrame** handler assigned to **ballMove()**
  - "standing still": **onEnterFrame** handler deleted
- The cue stick switches between two states:
  - "shooting": **onEnterFrame** handler assigned to **shoot()**
  - "aiming": **onEnterFrame** handler assigned to **aim()** (default)

```
stick_mc.onEnterFrame = aim;
onMouseDown = function () {
      stick_mc.onEnterFrame = shoot;
};
onMouseUp = function () {
      stick_mc.onEnterFrame = aim;
};
```

Little Tricks for Shooting...

- It is not sufficient to move the stick with the mouse during shooting at constant angle. Instead, the cue stick shall move back and forth on the line which was defined when aiming.

```
this._x = whiteBall_mc._x-Math.cos(angle)*dist;
this._y = whiteBall_mc._y-Math.sin(angle)*dist;
```



$$\sin angle = \frac{Ydist}{dist}$$

$$\cos angle = \frac{Xdist}{dist}$$

- At mouse down, add a test whether the stick is already in collision with the ball (which does not make sense):

```
onMouseDown = function () { ... Compute distance dist
  if (dist>110)
   stick_mc.onEnterFrame = shoot;
};
```

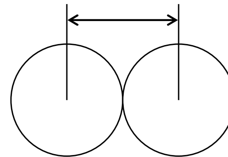

Shooting, Final Version

```
function shoot() {
  var dx = whiteBall_mc._x-_xmouse;
  var dy = whiteBall_mc._y-_ymouse;
  var dist = Math.sqrt(dx*dx+dy*dy);
  this._x = whiteBall_mc._x-Math.cos(angle)*dist;
  this._y = whiteBall_mc._y-Math.sin(angle)*dist;
  this.vx = this._x-this.oldx;
  this.vy = this._y-this.oldy;
  this.oldx = this._x;
  this.oldy = this._y;
  var dx = whiteBall_mc._x-this._x;
  var dy = whiteBall_mc._y-this._y;
  var dist = Math.sqrt(dx*dx+dy*dy);
  if (dist<110) {
     whiteBall_mc.vx = this.vx;
     whiteBall_mc.vy = this.vy;
     whiteBall_mc.onEnterFrame = ballMove;
     this.onEnterFrame = aim;
  }
}
```

Collision Detection Between Balls

- Two balls are able to collide.
  - Collision detection as a separate entity from the two balls
  - E.g. `onEnterFrame` handler on timeline

```
onEnterFrame = checkCollision;
function checkCollision() {
  var dx = redBall_mc._x-whiteBall_mc._x;
  var dy = redBall_mc._y-whiteBall_mc._y;
  var dist = Math.sqrt(dx*dx+dy*dy);
  if(dist<BALL_DIAMETER){
   ... collision detected ...
  }
}
```

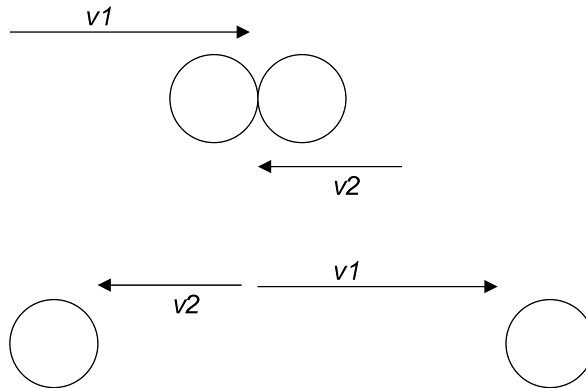


Physics: Speed, Velocity, Mass, Momentum

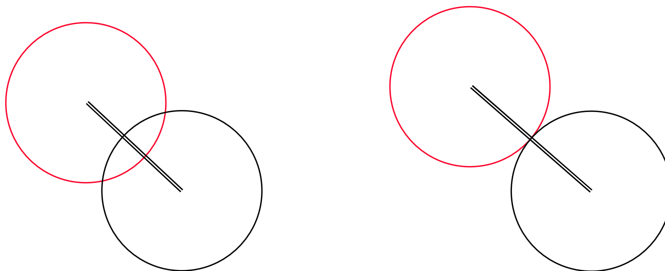
- *Speed:*
  - How fast is something moving (length/time)
- *Velocity:*
  - Vector describing movement: *speed* + *direction*
- *Mass:*
  - Basic property of object, depending on its material, leads under gravity to its *weight*
- *Momentum (dt. Impuls):*
  - Mass x Velocity
- *Principle of Conservation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

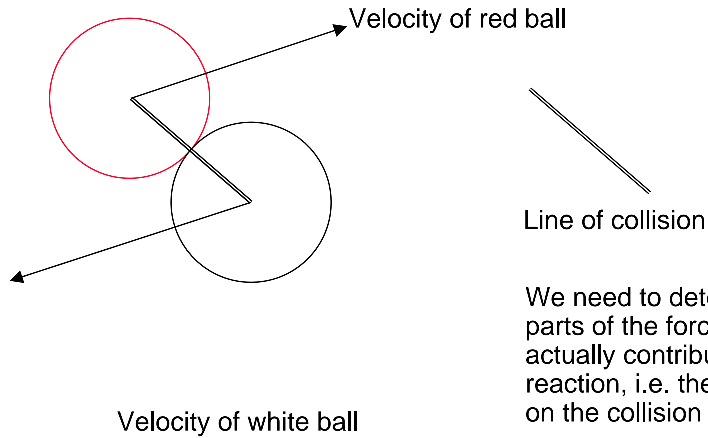


Placing Balls At Collision Time



This is a simplification compared to the actual physical laws.

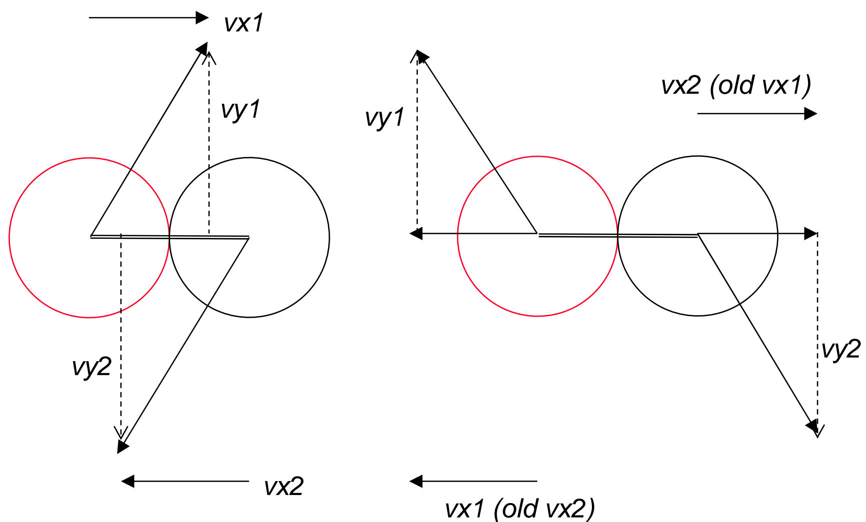
Physics of Collision, Step 1



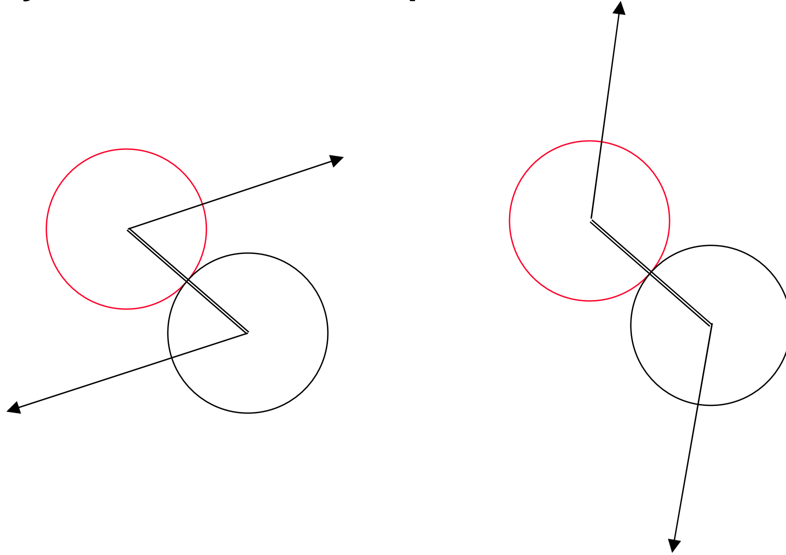
Line of collision

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

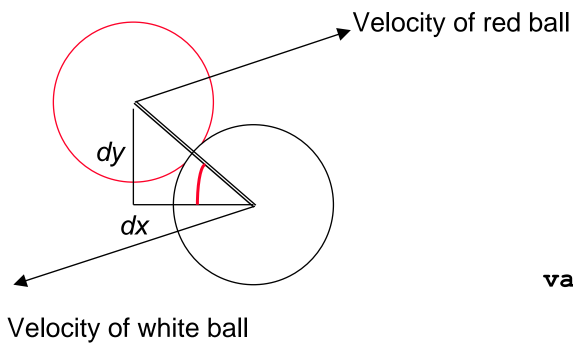
Physics of Collision, Step 2



Physics of Collision, Step 3

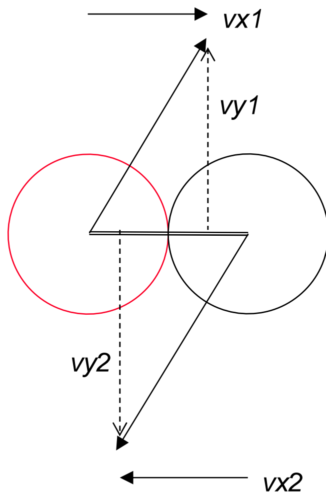


Computing Step 1




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

Computing Step 2, Part 1



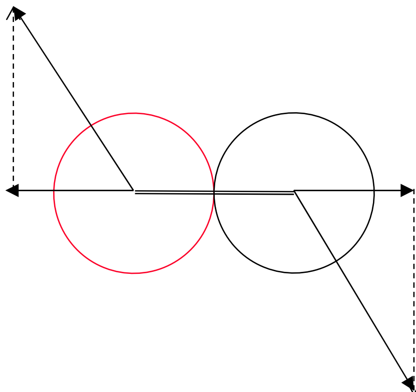
Counterclockwise rotation of vector (x, y):

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

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

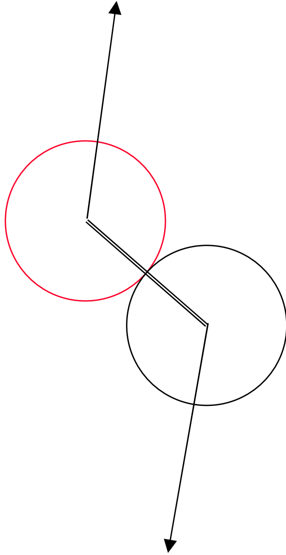
```
var angle =
  Math.atan2(dy,dx);
var cosa = Math.cos(angle);
var sina = Math.sin(angle);
var vx1 =
  cosa*redBall_mc.vx +
  sina*redBall_mc.vy;
var vy1 =
  cosa*redBall_mc.vx -
  sina*redBall_mc.vy;
... vx2, vy2
```


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_mc.vx =
  cosa*vx1 - sina*vy1;
redBall_mc.vy =
  cosa*vy1 + sina*vx1;
whiteBall_mc.vx =
  cosa*vx2 - sina*vy2;
whiteBall_mc.vy =
  cosa*vy2 + sina*vx2;

redBall_mc.onEnterFrame =
  ballMove;
whiteBall_mc.onEnterFrame =
  ballMove;
```