Course 01
Mathematics and Physics for Coding Motion and Interactivity in Web Graphics

Organizer
James Mohler
Department of Computer Graphics Technology
Purdue University

Presenter
Nishant Kothary
Department of Computer Graphics Technology
Purdue University

29th International Conference on Computer Graphics and Interactive Techniques
Conference: **21-26** July 2002
Exhibition: **23-25** July 2002
San Antonio, Texas USA

Table of Contents

Course Summary

This course is intended for the intermediate web developer working with emerging web technologies. The application that will be used in this course is Macromedia Flash MX. The course will provide an overview of the significance of mathematics and physics to multimedia development. Mathematics concepts, such as coordinate systems, linear algebra, trigonometry and vectors, will be covered in brief with interactive examples.

The course will also include an explanation of relevant kinematics, in particular – displacement, speed, velocity, acceleration and the equations of motion in two dimensions – and learn how to apply these concepts effectively in interactive media.

Prerequisites

The course is intended to be of interest to beginner to intermediate level multimedia, web-application and new media developers. Individuals interested in learning how to effectively use mathematics and physics in their Flash MX applications will benefit greatly from this course. Participants are expected to have a fair amount of experience coding with ActionScript (Flash's ECMA 262-based scripting language) and familiarity with high-school level mathematics and physics.

Syllabus

Module 1
Mathematics in Multimedia

8:30-8:45	15 mins	**Introduction – Mohler** Approaching motion and interactivity in multimedia Significance and relevance in industry
8:45-9:00	15 mins	**Multimedia authoring trends – Kothary** Frame-based development Object-oriented Programming Scripting: Not just event handling
9:00-9:30	30 mins	**Introductory Algebra – Mohler** Re-visiting basics Cartesian to Flash Linear Transformations
9:30-9:45	15 mins	Dynamic linear slider construction

9:45-10:15 30 mins **Trigonometry and Vectors – Kothary**
- Degrees and Radians
- Functions, identities, and theorems
- Vector primer

10:15-10:30 **Break**

Module 2
Kinematics in Multimedia

10:30-11:00 30 mins **Elementary Kinematics – Kothary**
- Learning applied physics
- Theoretical concepts and coded synonyms
  - Displacement and Distance
  - Speed and Velocity
  - Acceleration
- Gestalt and Newton
- Pseudo-dynamics in new media: An extension of kinematics

11:00-11:15 15 mins **Two-dimensional frame-based motion – Kothary**
- Vector components and transformations in 2D
- Re-formulating the equations of motion

11:15-11:30 15 mins *Interactive class problem:*
Unrestricted OOP-based motion: Tank

11:30-11:40 10 mins **An overview of 3D in new media – Mohler**
- Flash: Draw functions
- Director: Shockwave 3D
- Essential math skills

11:40-11:50 10 mins **What to expect – Kothary**
- Web-delivered Inverse kinematics
- 3D engines and navigation systems

11:50-12:00 10 mins **Conclusion – Mohler**
- Approaching advanced motion-based programming
- Research and discovery tactics

12:00-12:15 15 mins **Questions and Answers**

Thoughts from the Presenters

The recent popularity of user-friendly software to create rich, interactive and complete multimedia experiences has opened up creative avenues for many in today's design world. Concepts, such as object oriented programming and encapsulation, that were alien to most non-programmers have become frequently-used terms amongst those planning to sustain stability in computer graphics for the years to come. By "everyone" we are referring to graphic designers, game programmers, web developers, multimedia creators, and freelancers.

The growth of the Internet in the mid-90's marked a turning point in the history of information exchange. Similarly, it's evident that new media is starting a revolution as we write this prologue – a revolution of creativity and hi-fidelity information exchange. The advent of technologies such as Flash have made it easier for the intermediate web developer to create a visually appealing and media-rich experience without having to worry about the intricacies of well-formed programming constructs or even necessary syntax.

However, when you start talking about media-rich experiences, you have to start dealing with life-like motion, animated sequences, and event-driven kinematics. All sciences are intertwined and this particular situation best illustrates the dependency of multimedia development on age-old laws of physics and theorems of mathematics. As much as one would like to deny it, mathematics and physics supply the syntax for interactivity and animation. As time progresses, one may expect to see a larger part of the graphics population have access to the development tools necessary to create such interactive experiences. At some point in the future, these tools will assume at least some intermediate knowledge of math and physics foundation on the part of the developer. It is up to the developer to embrace the beauty, logic and simplicity of these concepts and their application in multimedia and hypermedia development.

About this Course

This course is intended for the intermediate web developer working with emerging web technologies such as Flash and Director. We will not be working with Director, but many of the concepts remain constant and are transferable to other software applications.

You will benefit from this course if

- You are a new media developer struggling to find the right direction in achieving intelligent development skills based on necessary mathematics and physics.

- You are coming from mainstream game development. This course should be useful in making a smooth transition into simple game theory in frame-based development environments such as Flash.

- You have no experience with Flash or multimedia development. In general, this course should definitely kindle in you some excitement about the new media revolution and what is possible in applications such as Flash.

In short, if you are interested in new media, basic kinematics and mathematics in multimedia development, or simply want to get better with Flash and other similar new media development products, then this course is for you.

About the Course Notes

This documentation contains three main parts: the introductory materials, course slides and supplementary material. The supplementary material includes an overview of important mathematics and physics concepts that will be used and demonstrated in the hands-on course. You will note that most of this information is theory-oriented. Within the course we will apply this within various examples.

In addition to the body of the supplementary materials, there are three appendices. Appendix A provides a listing of some of the most useful Flash resources available on the web at the time we compiled this document. During the session we will likely append to this list, as the list of Flash oriented development sites grows daily.

Appendix B provides a listing of helpful math and physics resources on the web. If these are topics you're comfortable with or ones that challenge you, you'll find many of these sites helpful in learning more about math and physics. We have to limit the topics we can touch in the course simply because there is just too much. These web resources will be helpful in your quest to learn more about math and physics for multimedia development.

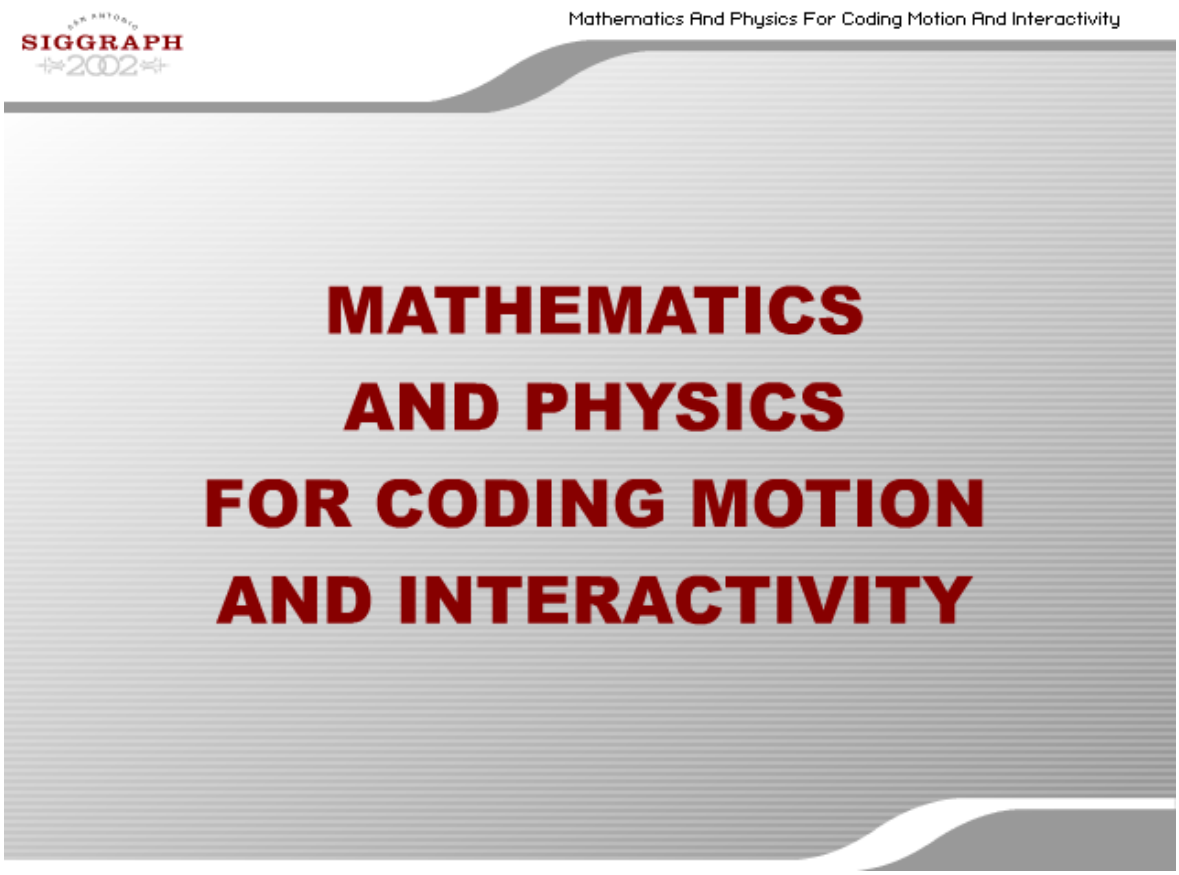
Finally, appendix C provides a brief introduction to object-oriented programming in Flash MX and appendix D provides further readings and references. If you have made the transition from Flash 5 (or are still doing so) you are undoubtedly aware of the major changes in ActionScript coding. We decided to include Appendix C because OOP in Flash is sometimes tricky, particularly if you are not used to the OOP approach.

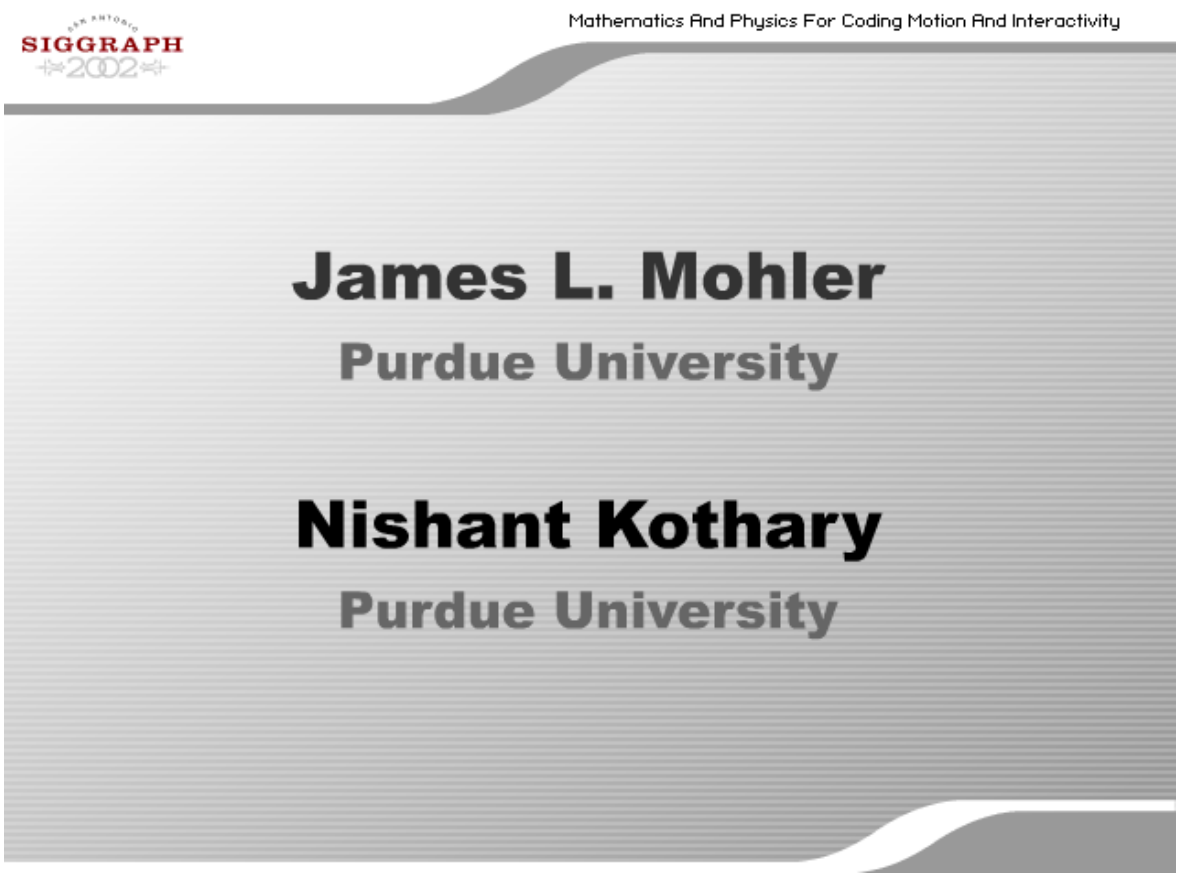
About the Authors

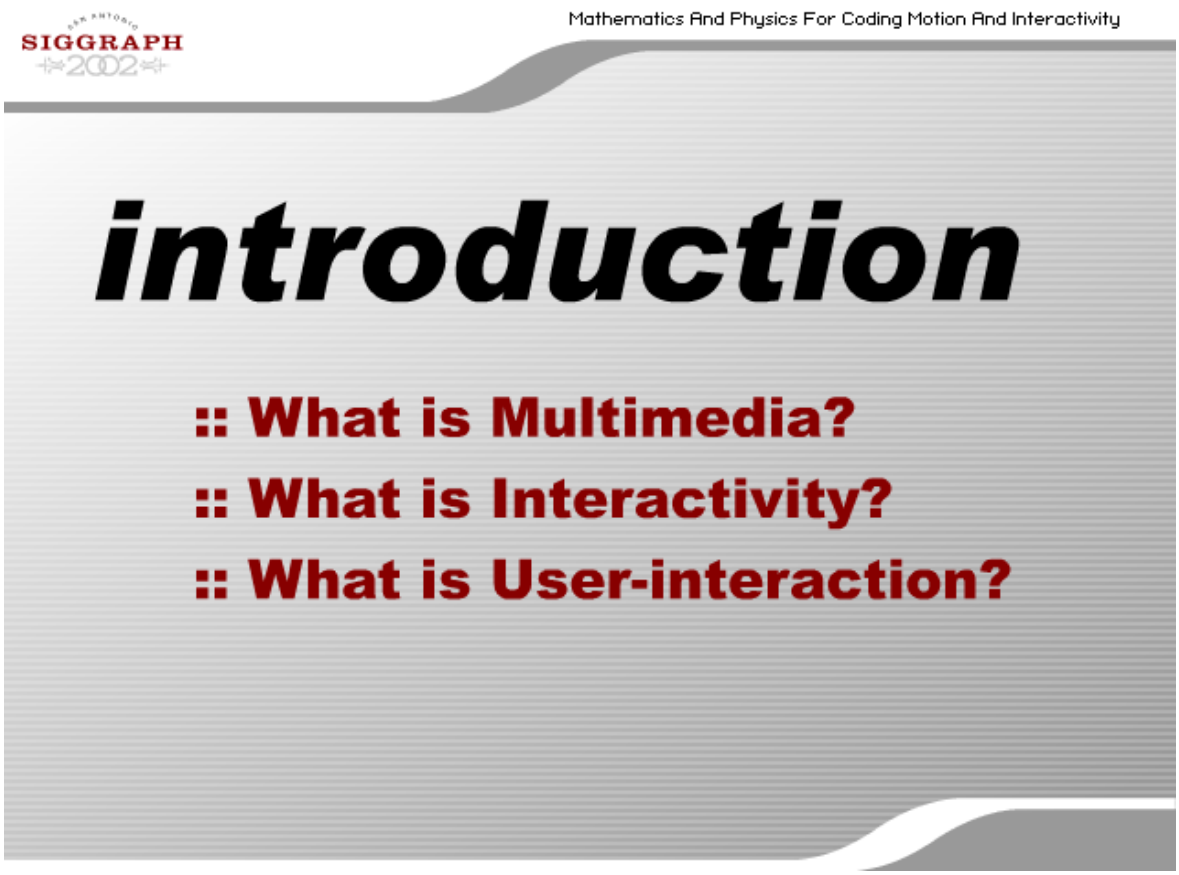
James L. Mohler is an Associate Professor in the Department of Computer Graphics Technology at Purdue University. He has authored or coauthored 16 texts related to multimedia and hypermedia development, presented over 40 papers and workshops at national and international conferences and written 20 articles for academic and trade publications. He has been awarded several teaching awards and was recently chosen as the Fulbright Distinguished Chair in Multimedia for 2002. James can be contacted at **jlmohler@tech.purdue.edu**.

Nishant Kothary is a freelance multimedia developer specializing in web design. He teaches courses at Purdue University dealing with the fundamentals of scripting and tagging, and currently serves as web developer for the Office of Technology Commercialization at Purdue University. Due to his background in computer science, mathematics and computer graphics, Nishant has been able to develop a progressive approach to the implementation of kinematics and mathematics in ActionScript. Nishant can be contacted at **kothary@purdue.edu**.

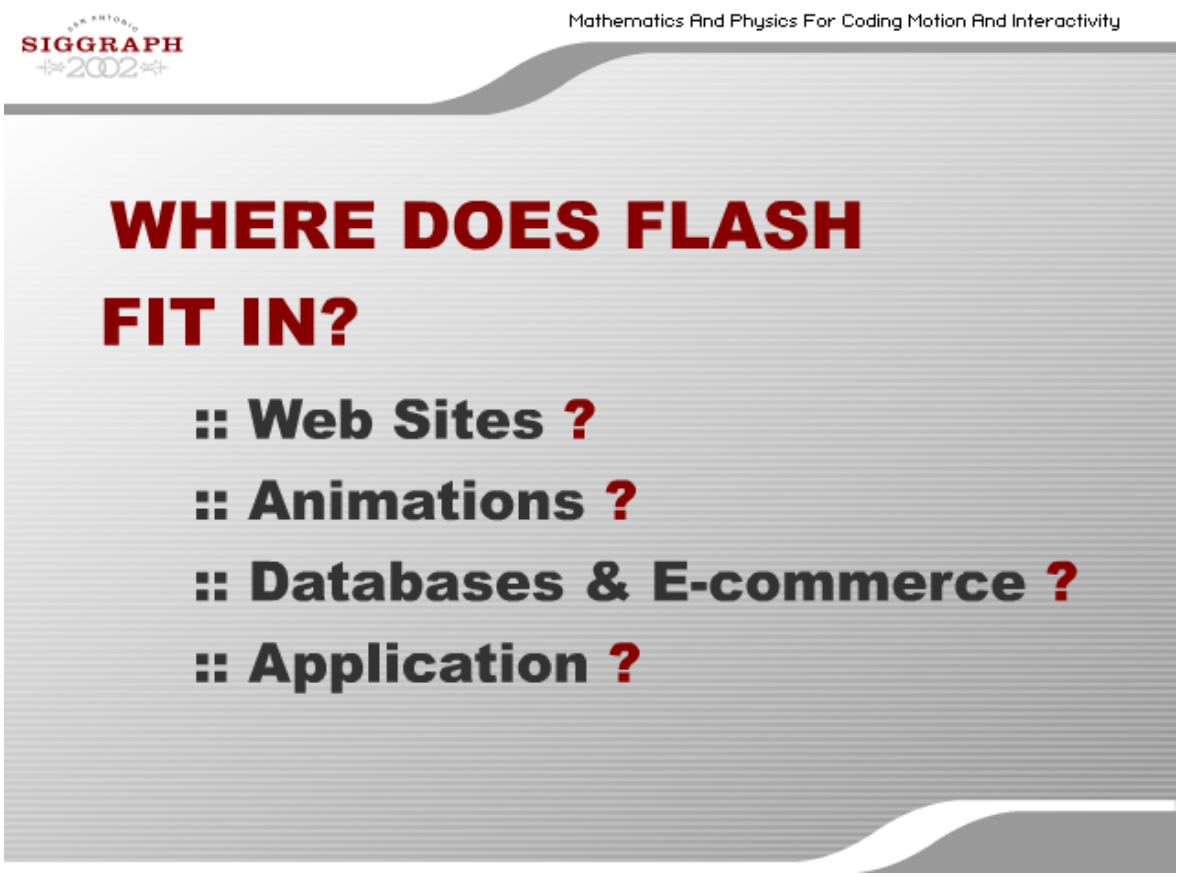
Course Slides


SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
MATHEMATICS
AND PHYSICS
FOR CODING MOTION
AND INTERACTIVITY

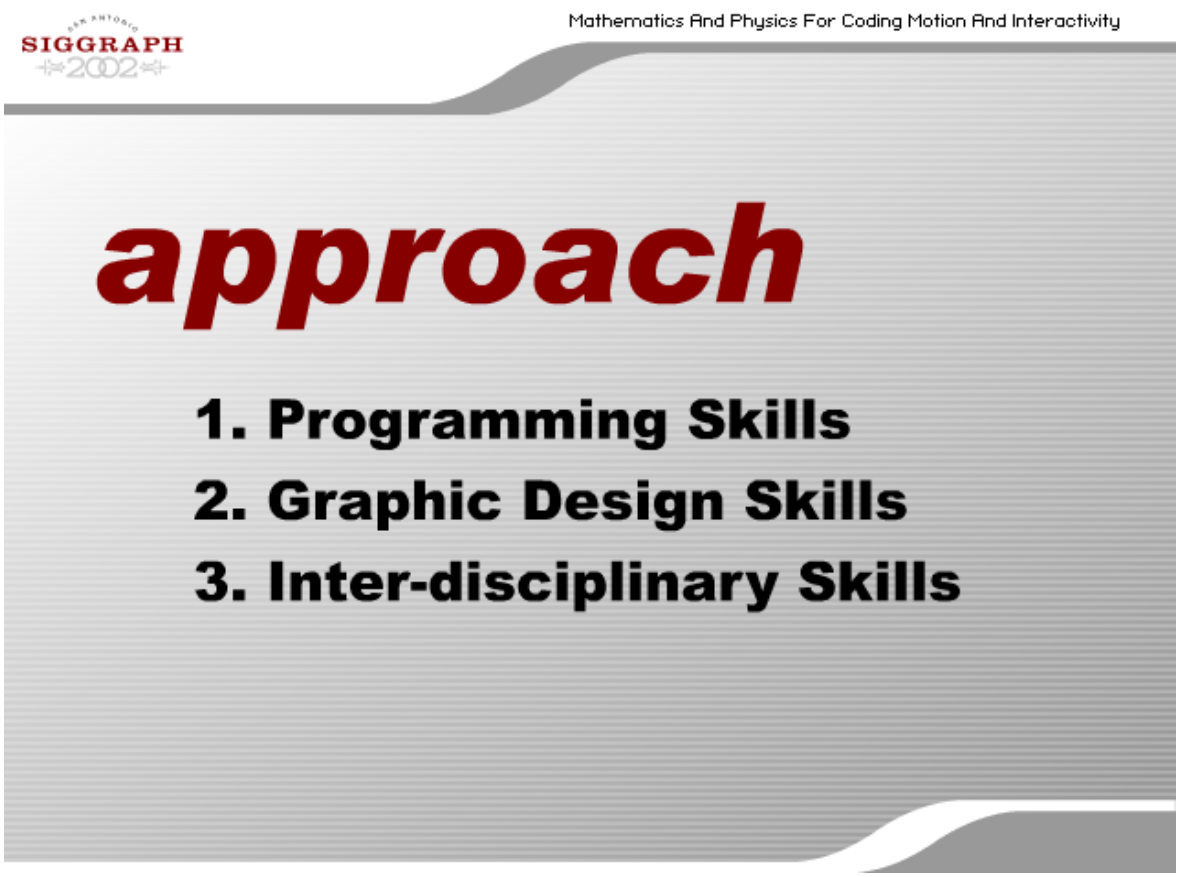

SAN ANTONIO
SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
James L. Mohler
Purdue University
Nishant Kothary
Purdue University

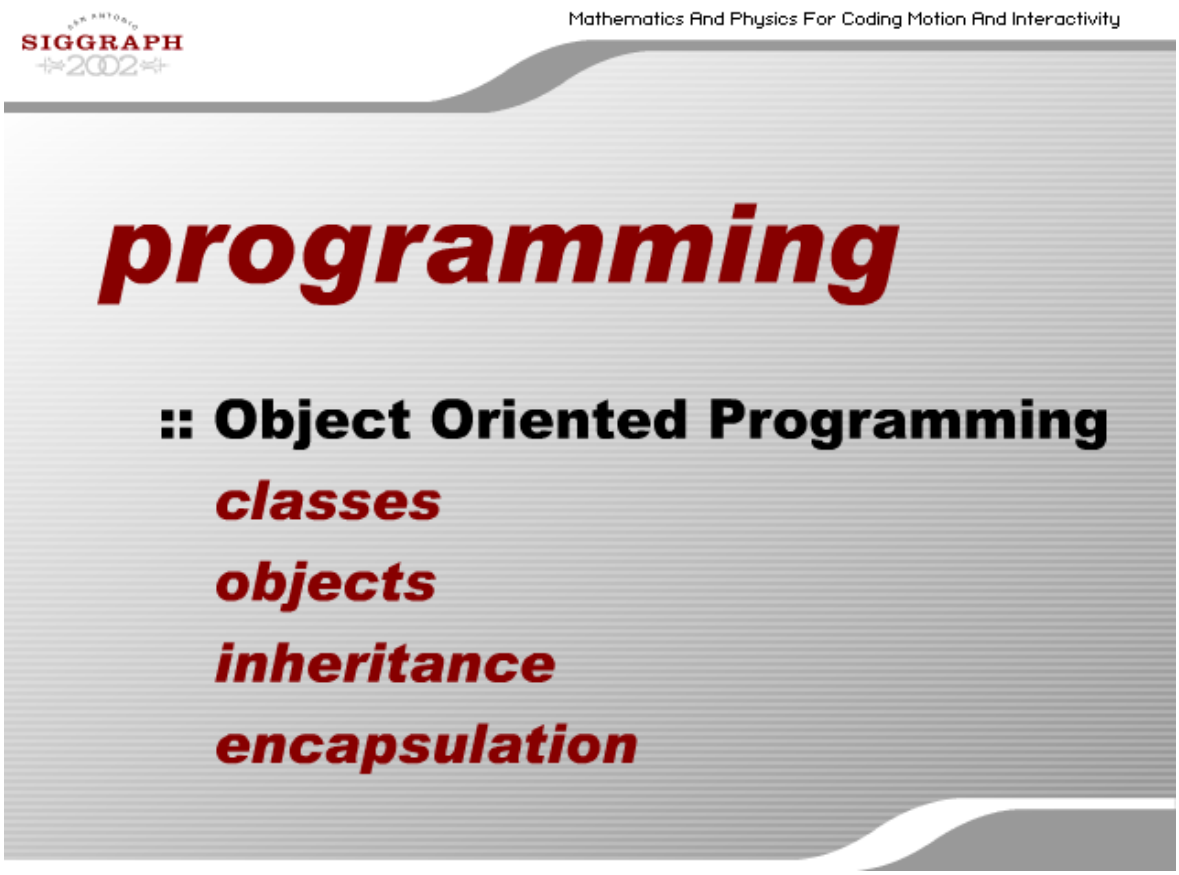

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
introduction
:: What is Multimedia?
:: What is Interactivity?
:: What is User-interaction?

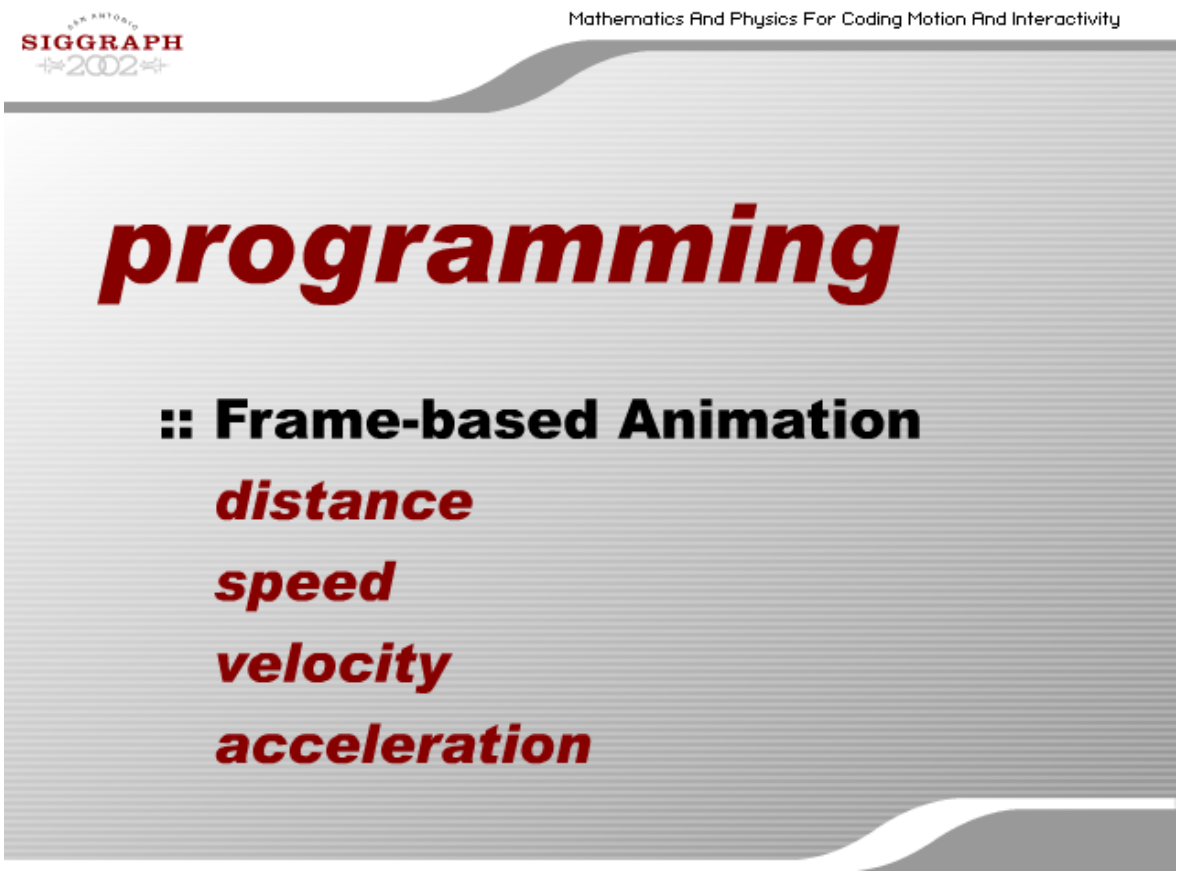

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
NEW MEDIA

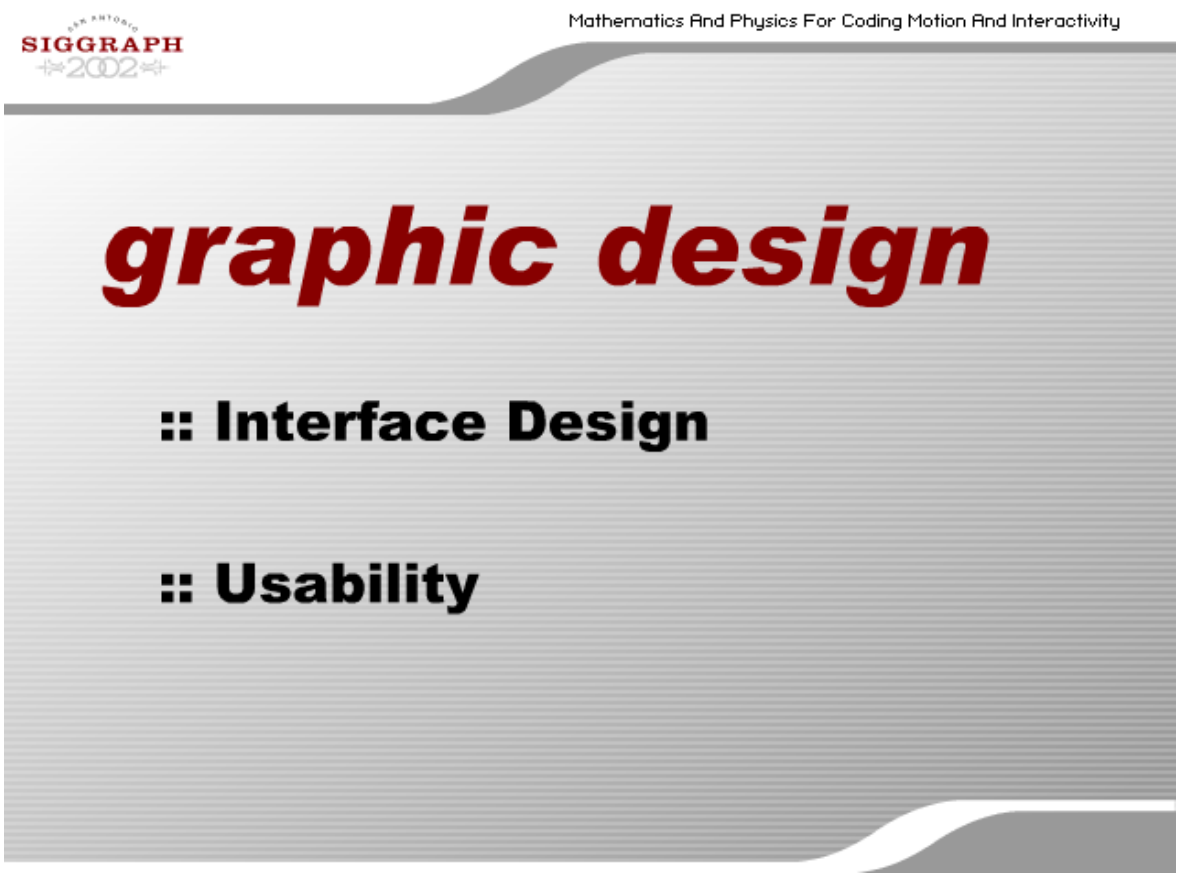

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
WHERE DOES FLASH FIT IN?
:: Web Sites ?
:: Animations ?
:: Databases & E-commerce ?
:: Application ?


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
HOW ABOUT
EVERYWHERE!

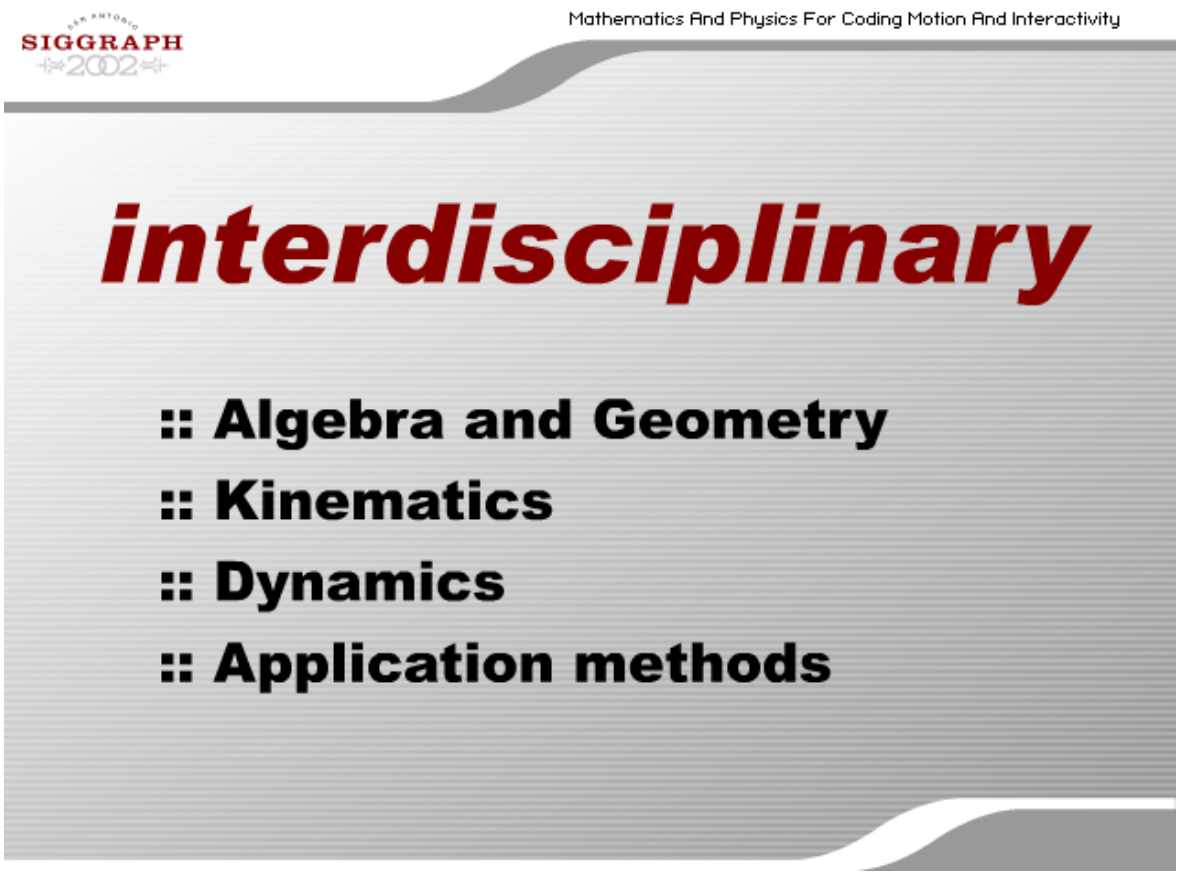

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
approach
1. Programming Skills
2. Graphic Design Skills
3. Inter-disciplinary Skills

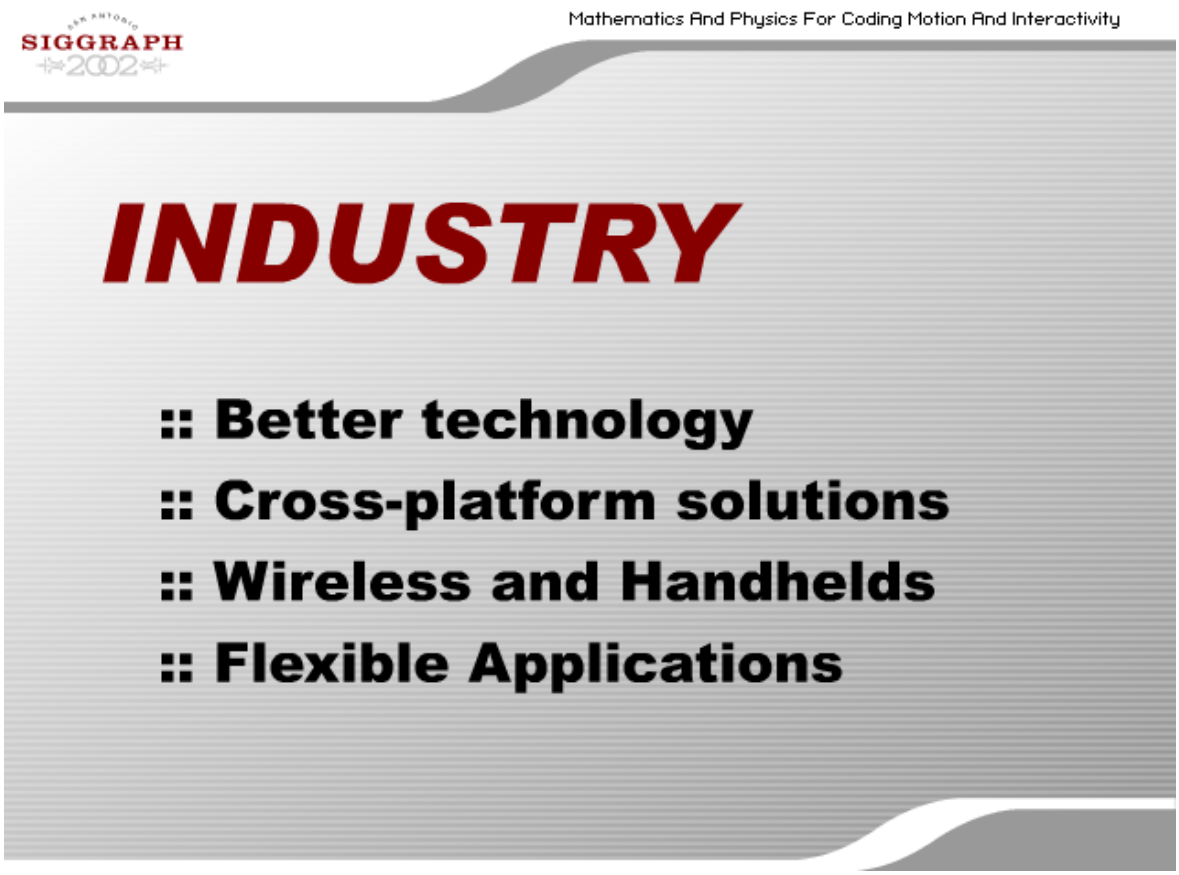

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
programming
:: Object Oriented Programming
classes
objects
inheritance
encapsulation


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
programming
:: Frame-based Animation
distance
speed
velocity
acceleration

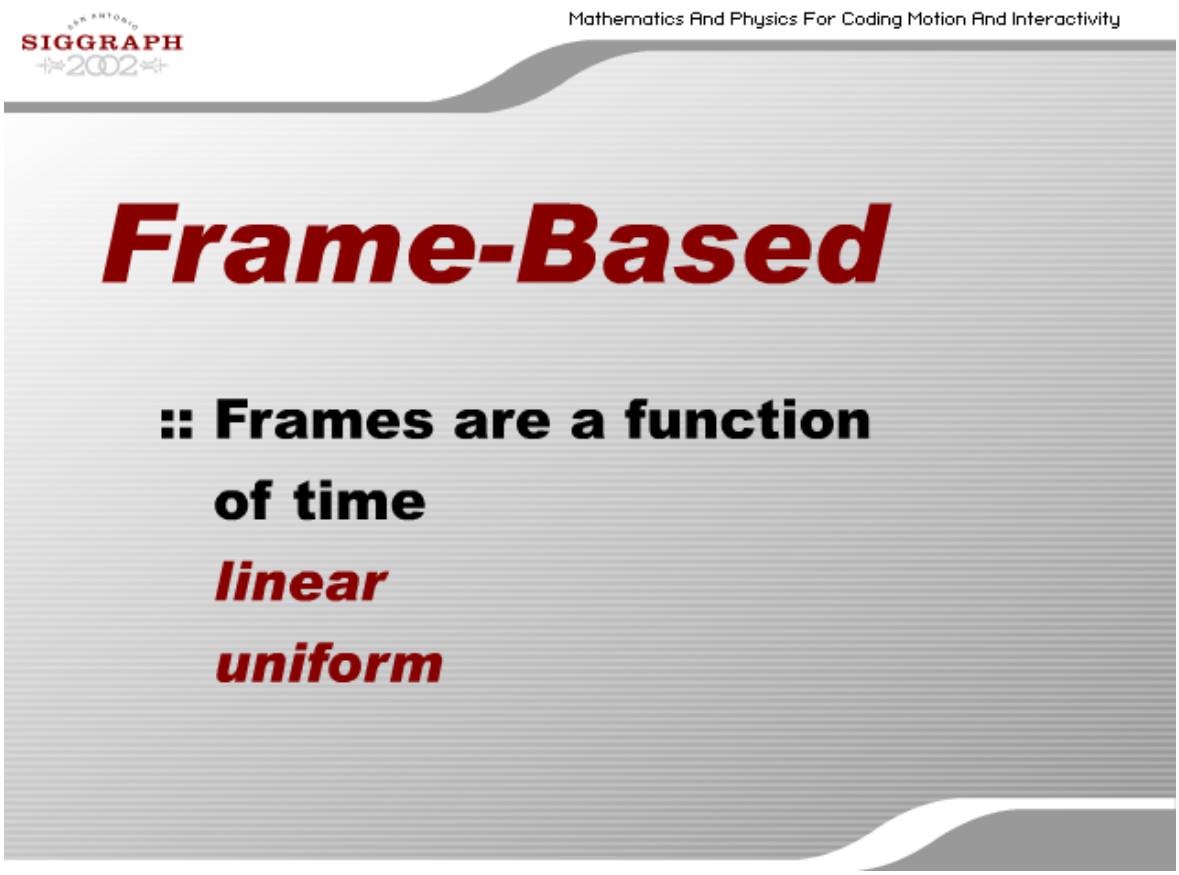

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
graphic design
:: Interface Design
:: Usability

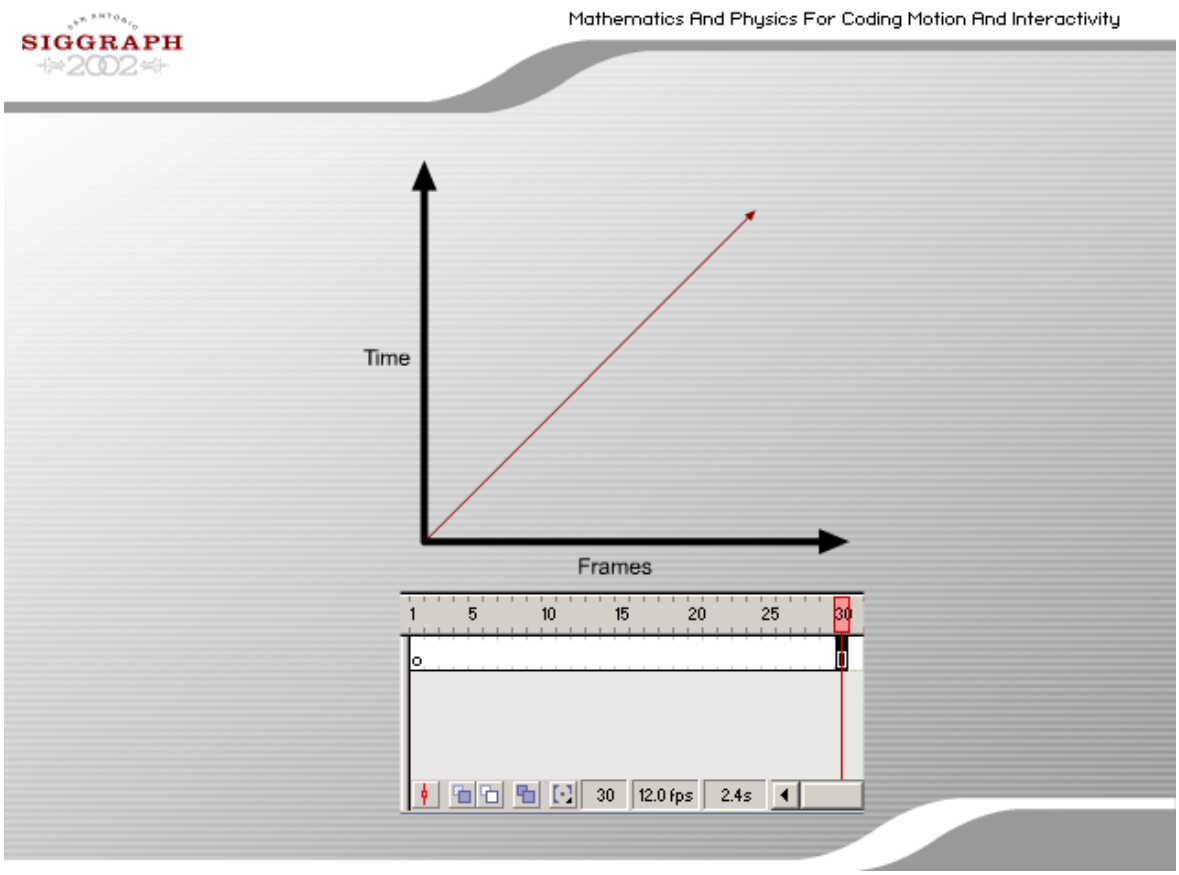

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
LASTBUTNOT
THE
LEAST

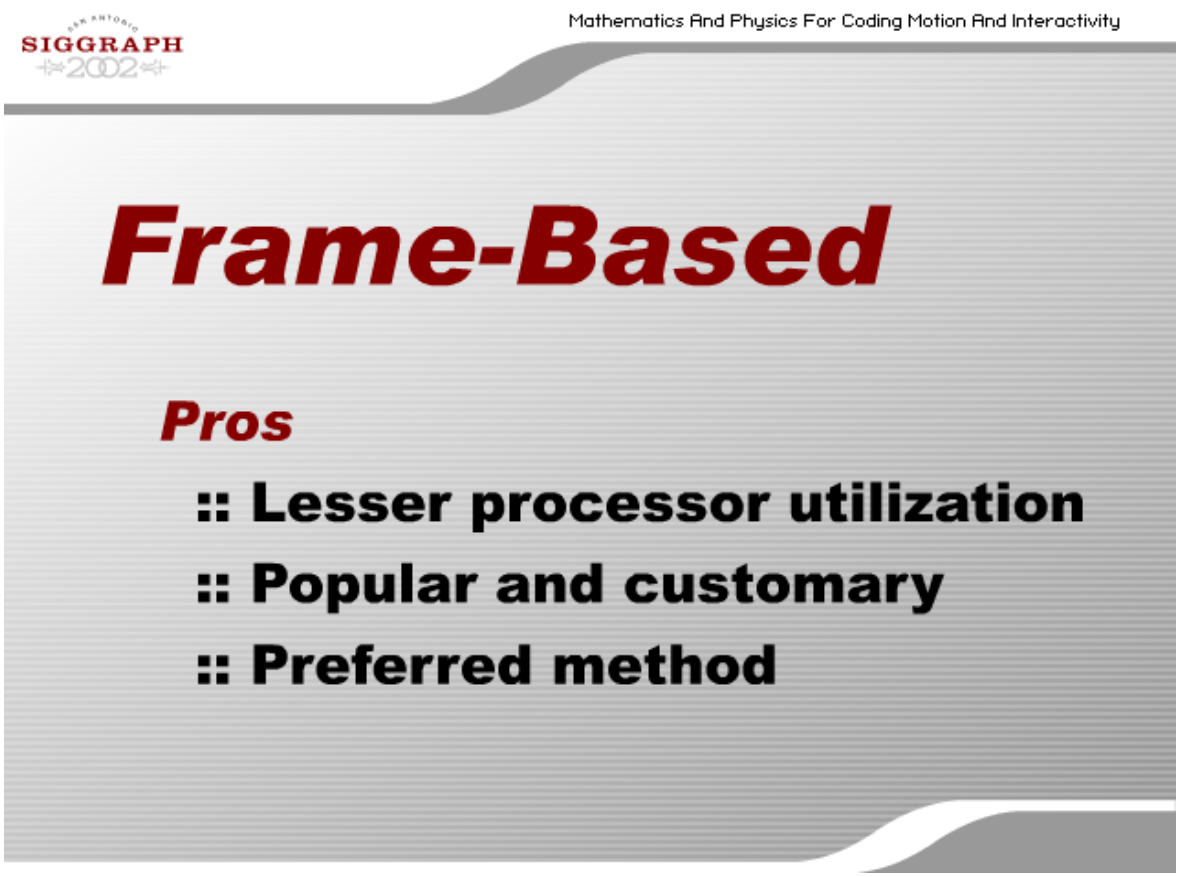

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
interdisciplinary
:: Algebra and Geometry
:: Kinematics
:: Dynamics
:: Application methods

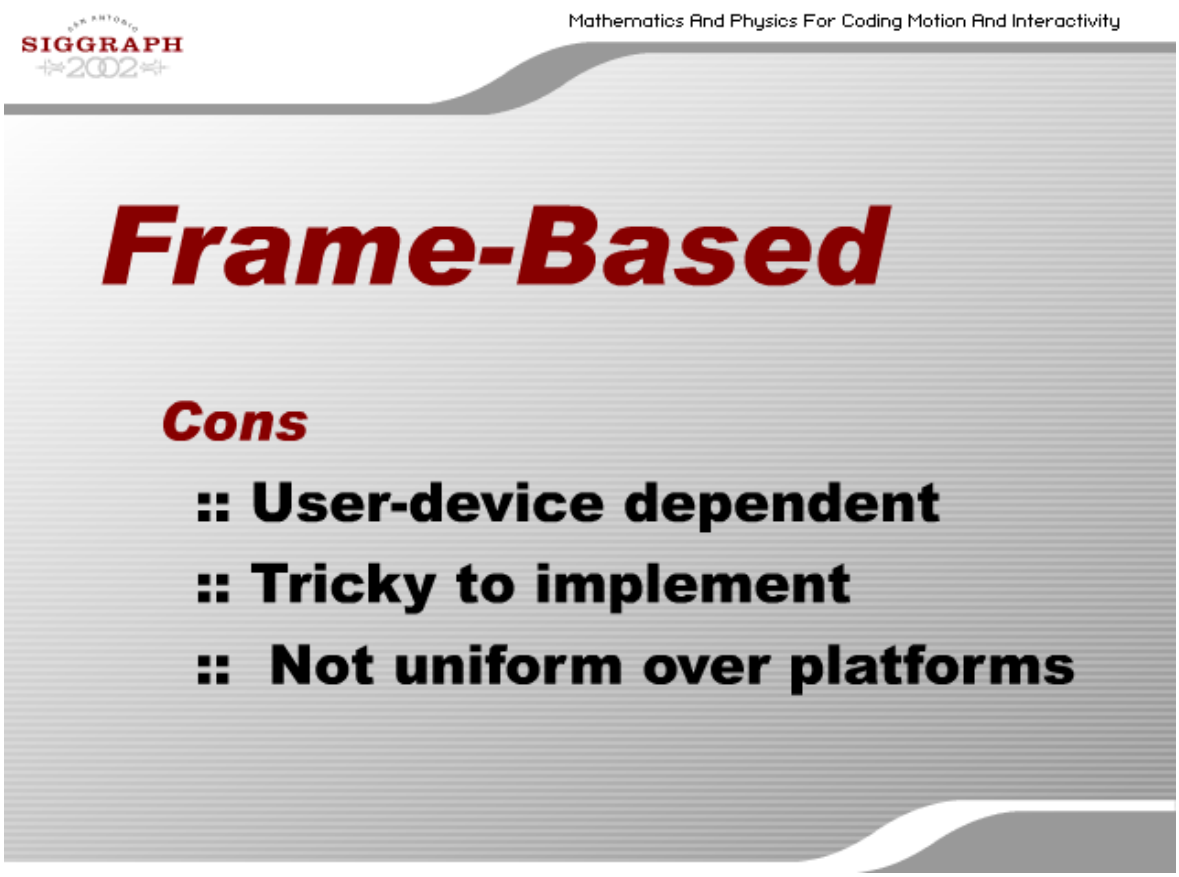

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
INDUSTRY
:: Better technology
:: Cross-platform solutions
:: Wireless and Handhelds
:: Flexible Applications

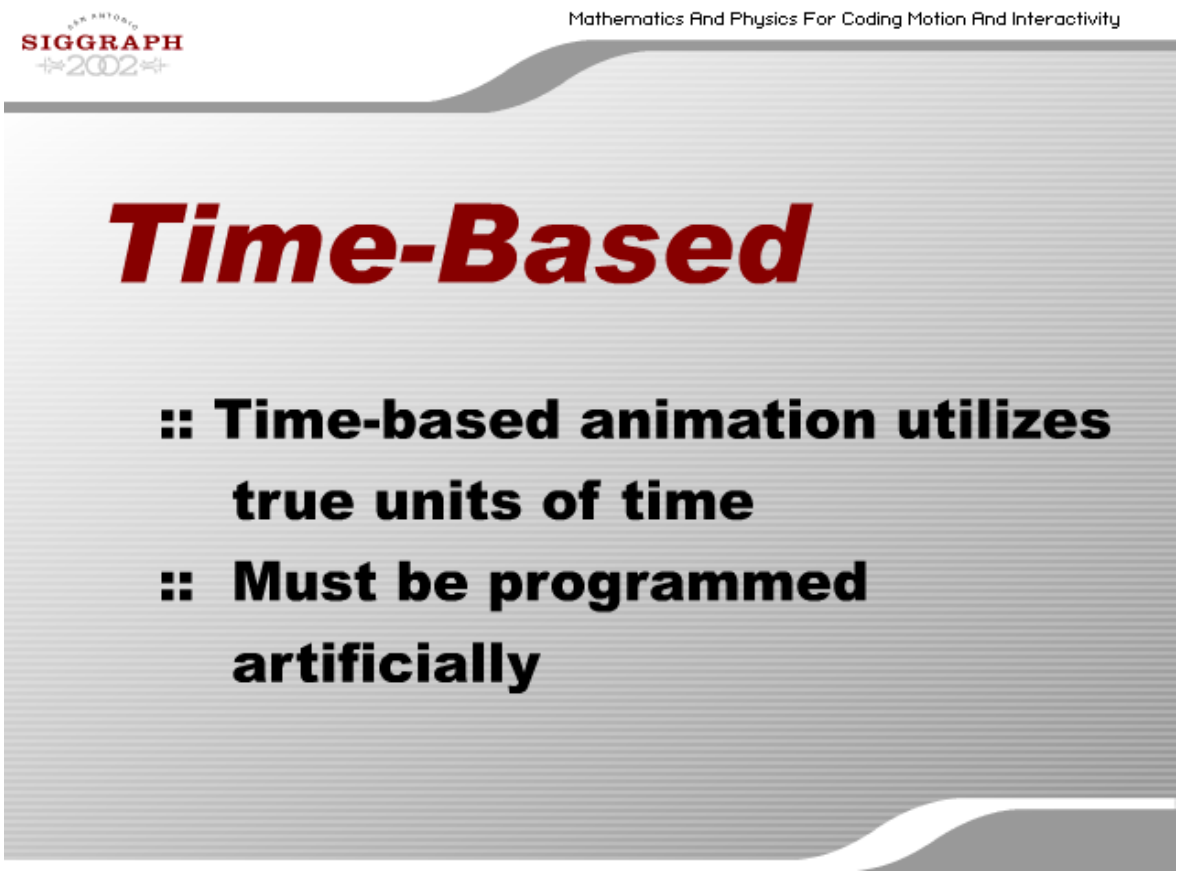

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
essential
MATHEMATICS
multimedia authoring trends

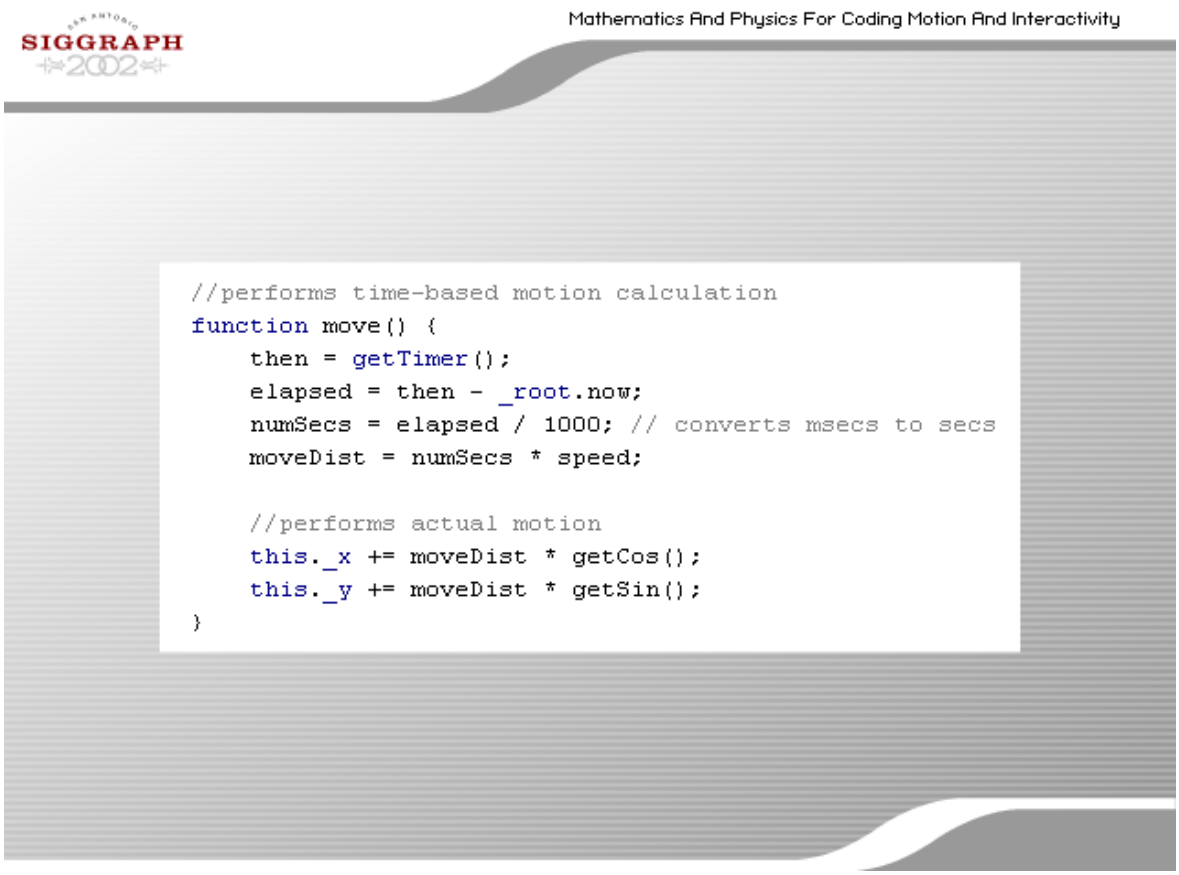

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Frame-Based
:: Frames are a function of time
linear
uniform

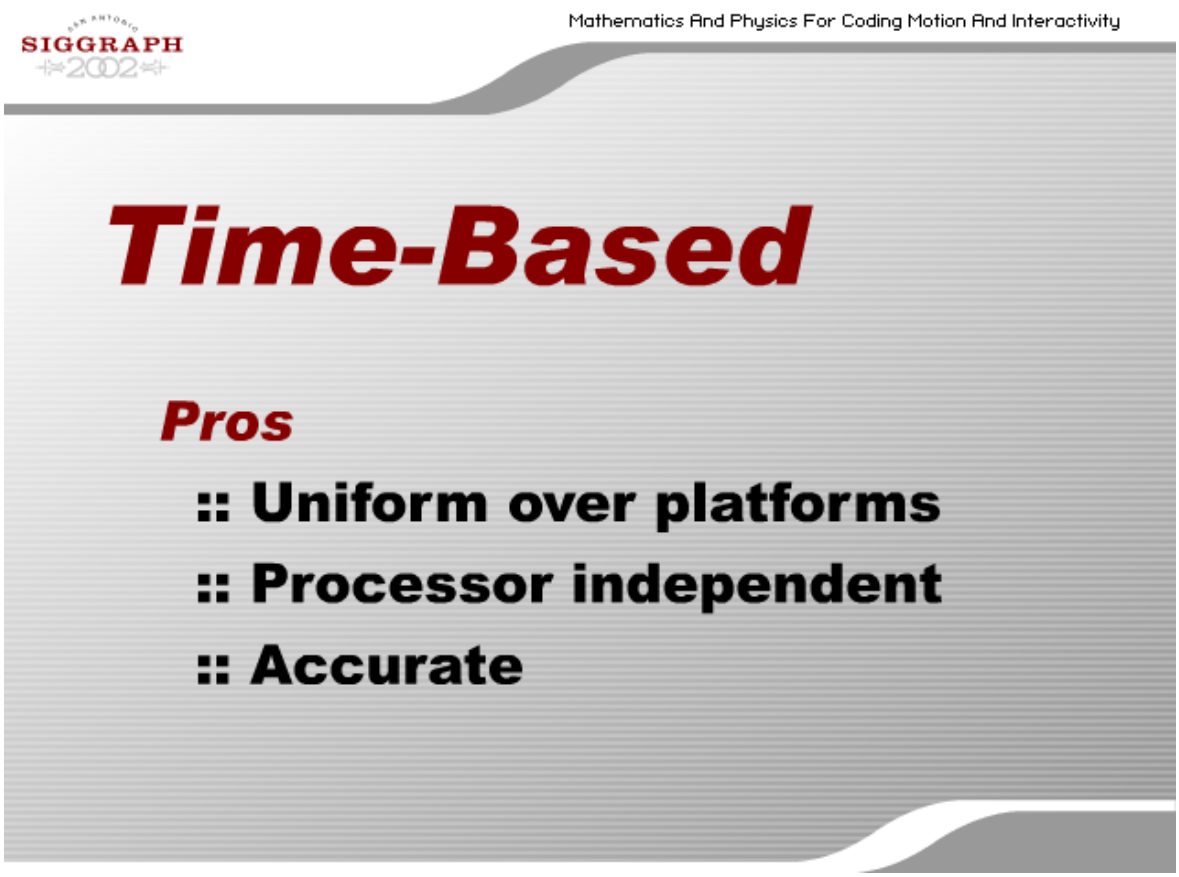

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Time
Frames
1
5
10
15
20
25
30
30
12.0 fps
2.4s


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Frame-Based
Pros
:: Lesser processor utilization
:: Popular and customary
:: Preferred method


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Frame-Based
Cons
:: User-device dependent
:: Tricky to implement
:: Not uniform over platforms

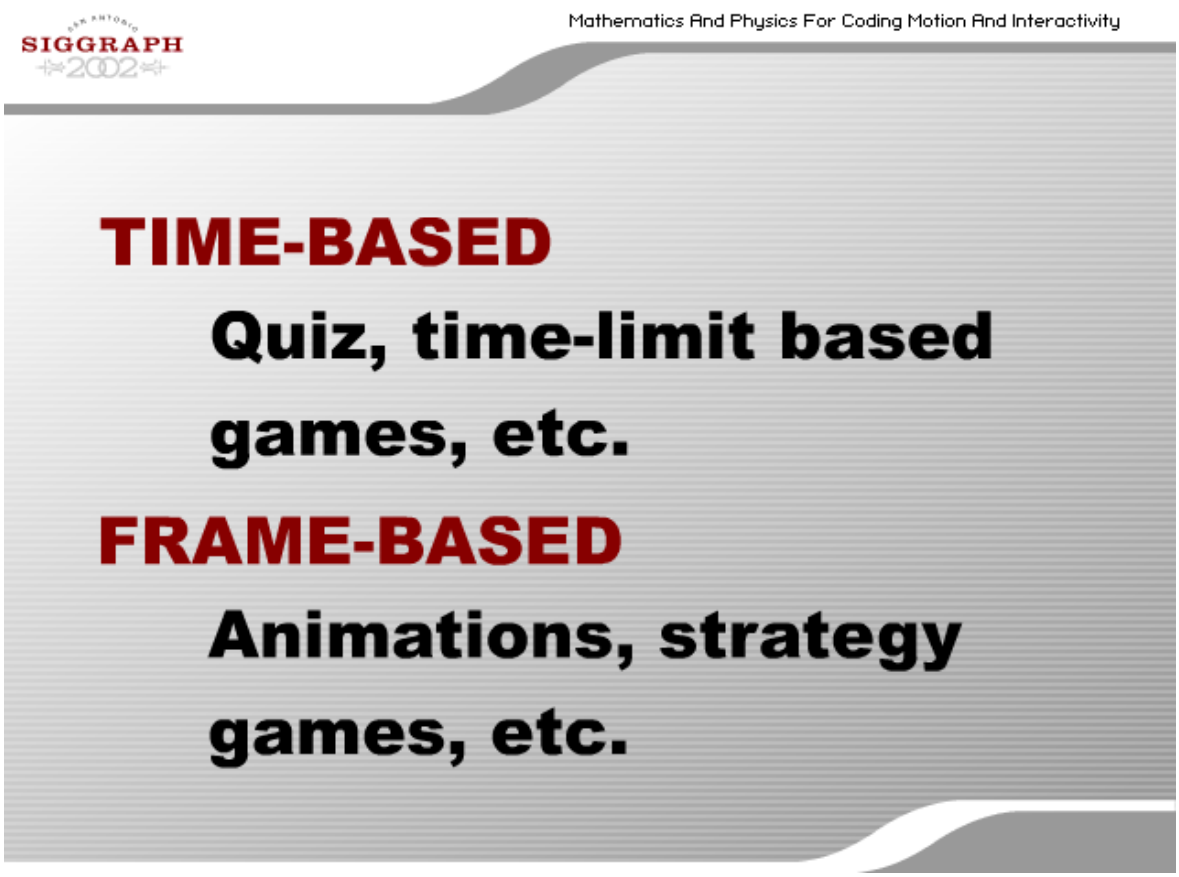

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Time-Based
:: Time-based animation utilizes true units of time
:: Must be programmed artificially


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
//performs time-based motion calculation
function move() {
then = getTimer();
elapsed = then - _root.now;
numSecs = elapsed / 1000; // converts msecs to secs
moveDist = numSecs * speed;
//performs actual motion
this._x += moveDist * getCos();
this._y += moveDist * getSin();
}

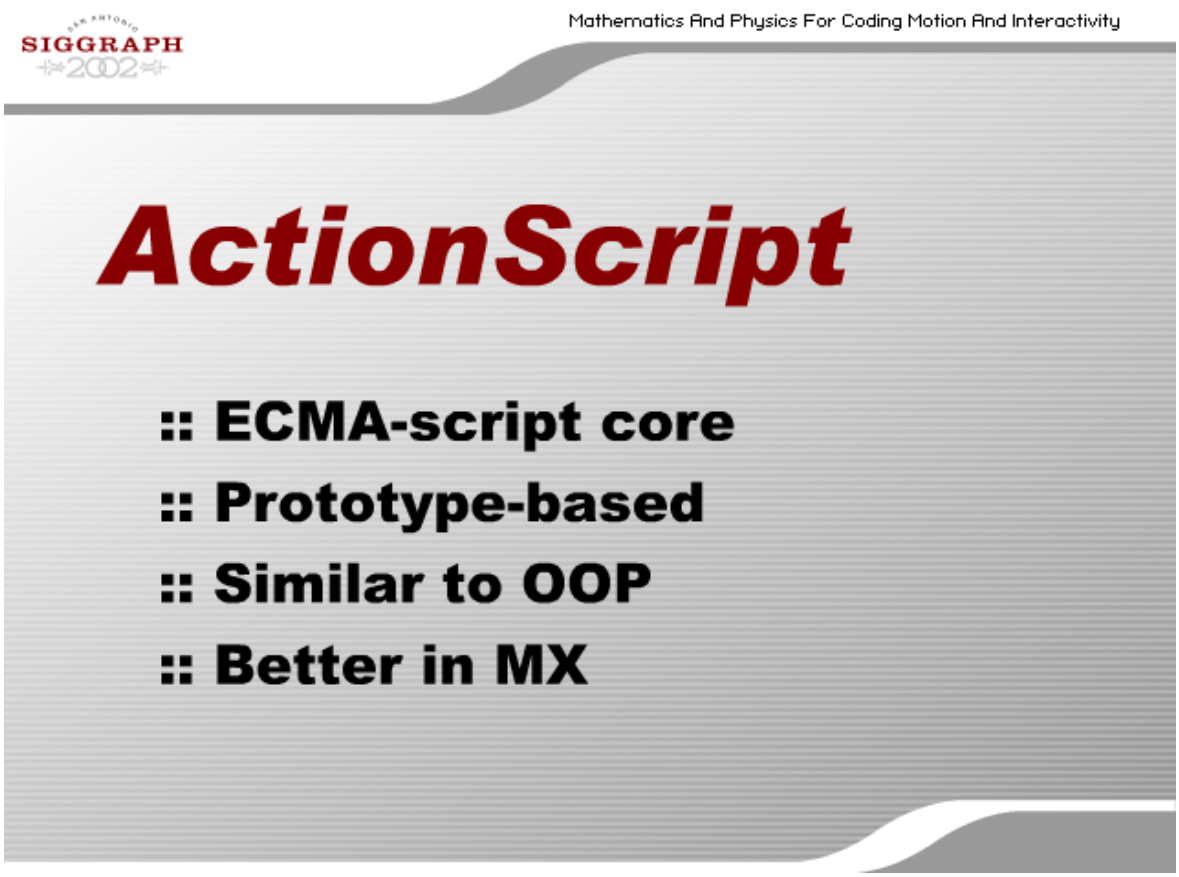

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Time-Based
Pros
:: Uniform over platforms
:: Processor independent
:: Accurate

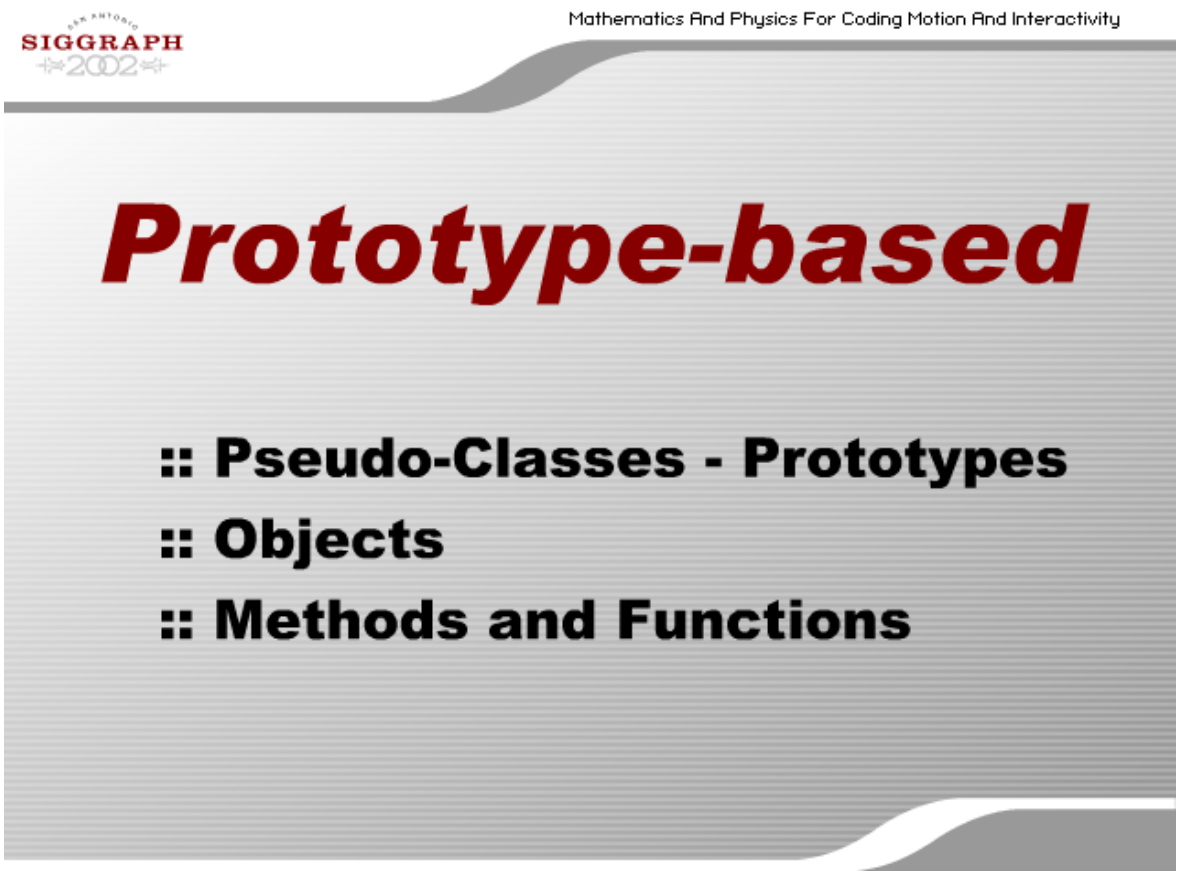

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Time-Based
Cons
:: Processor intensive
:: Not applicable to everything


SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity

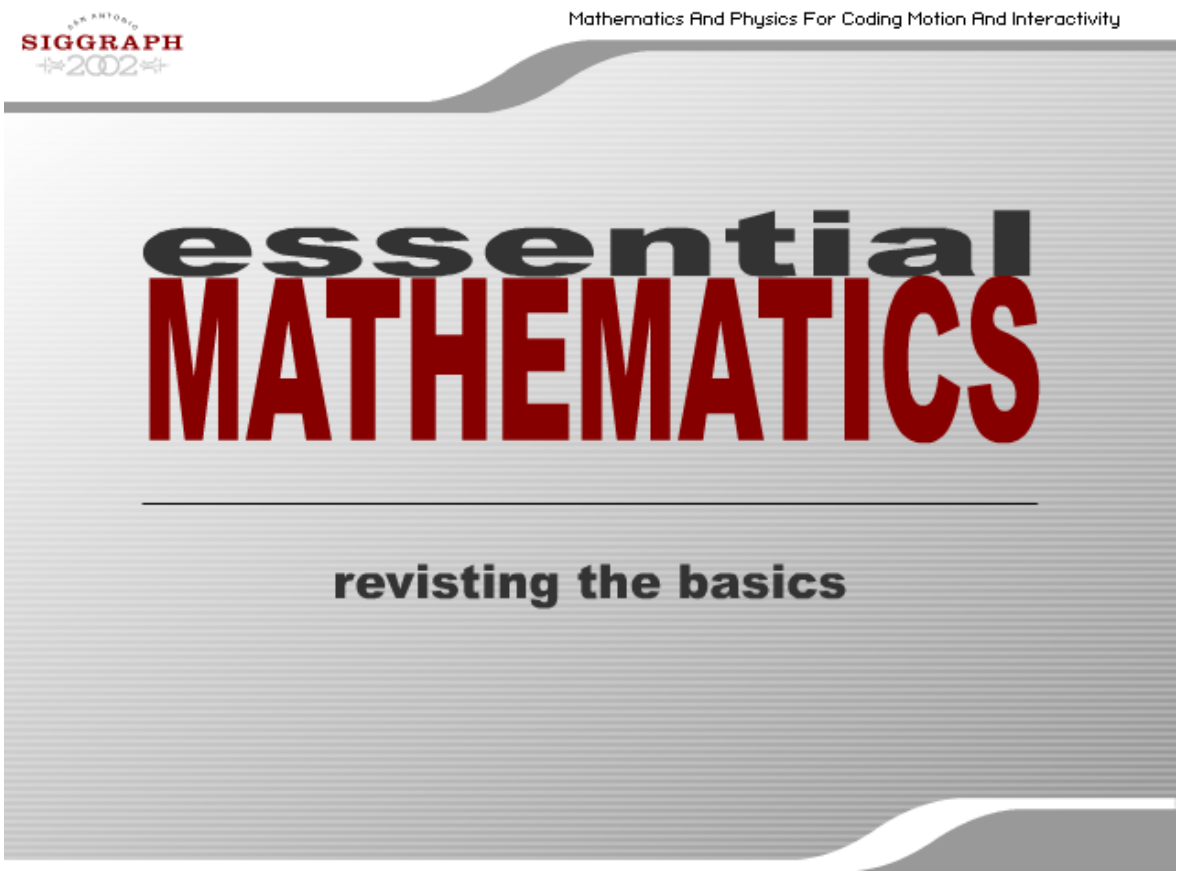

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
TIME-BASED
Quiz, time-limit based games, etc.
FRAME-BASED
Animations, strategy games, etc.

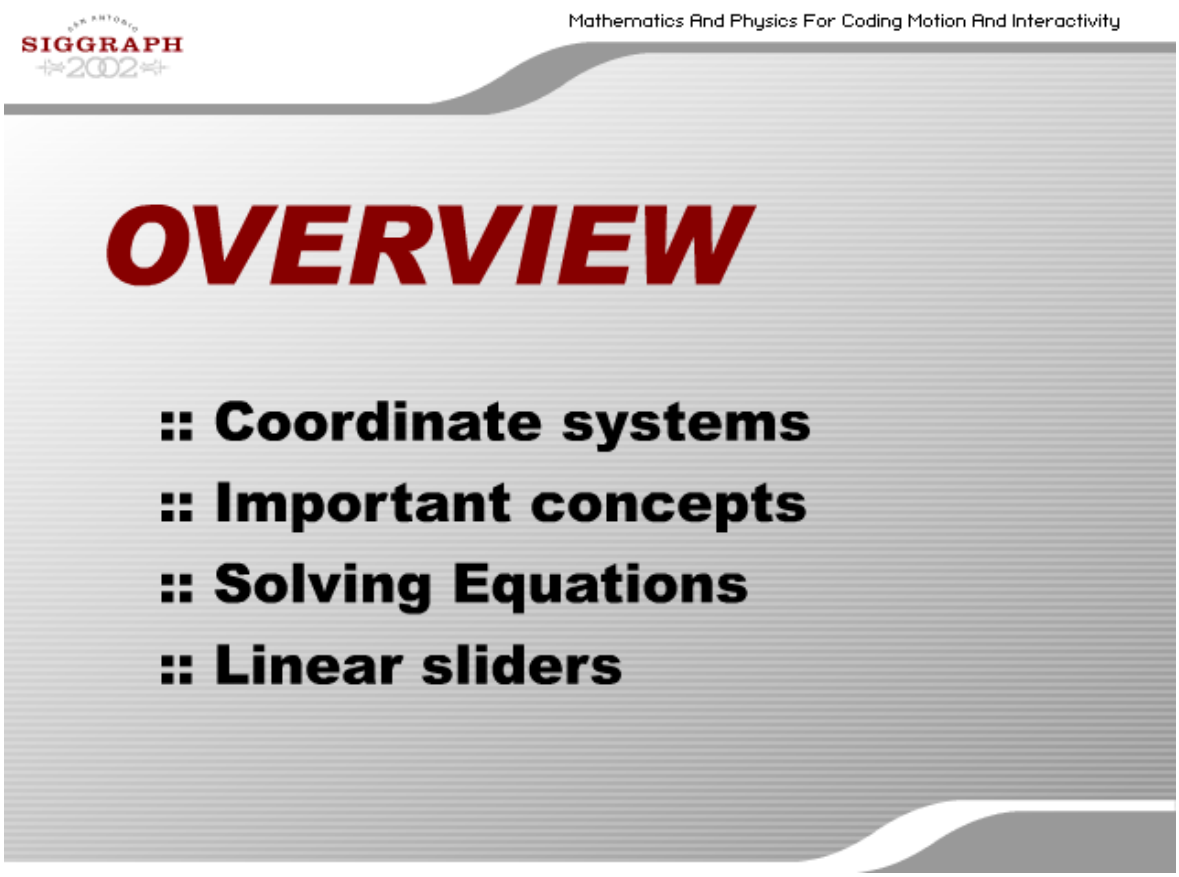

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
OOP
OBJECT
ORIENTED
PROGRAMMING

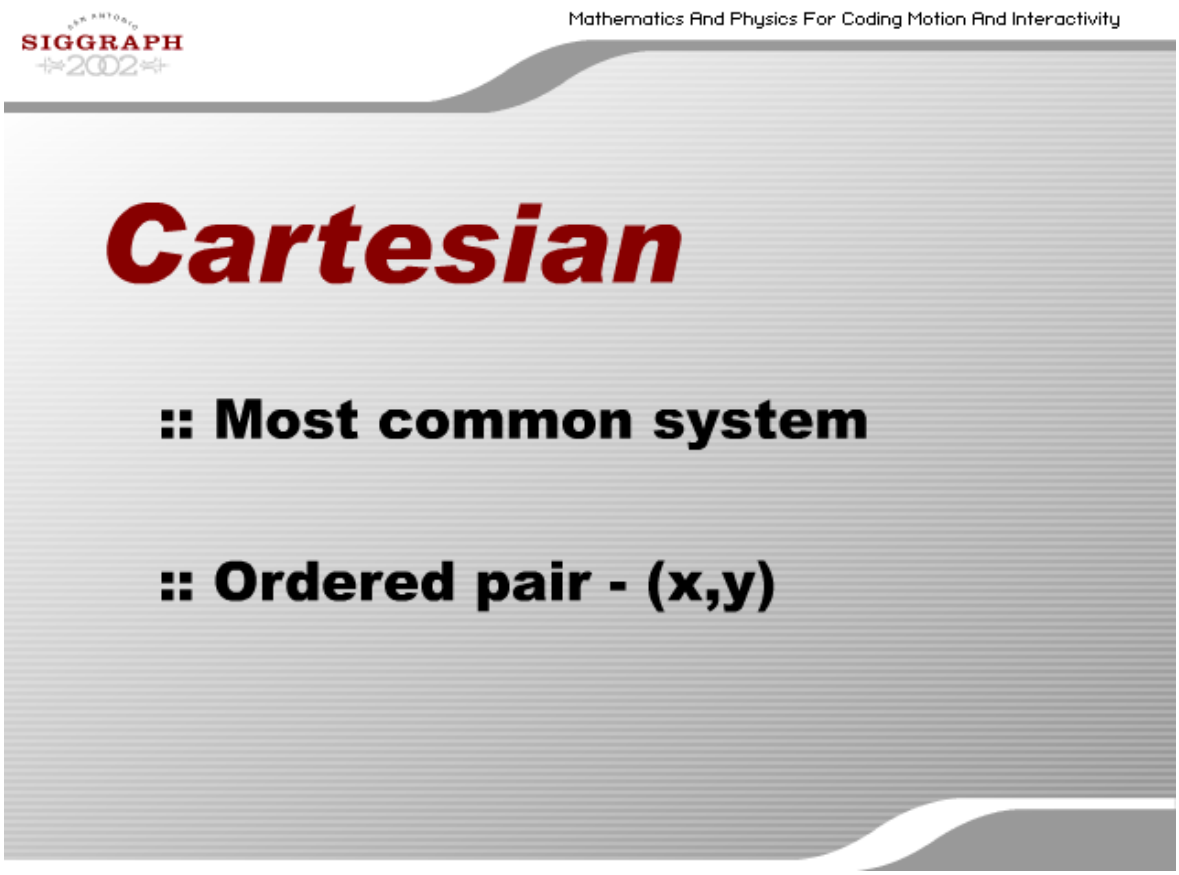

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
ActionScript
:: ECMA-script core
:: Prototype-based
:: Similar to OOP
:: Better in MX

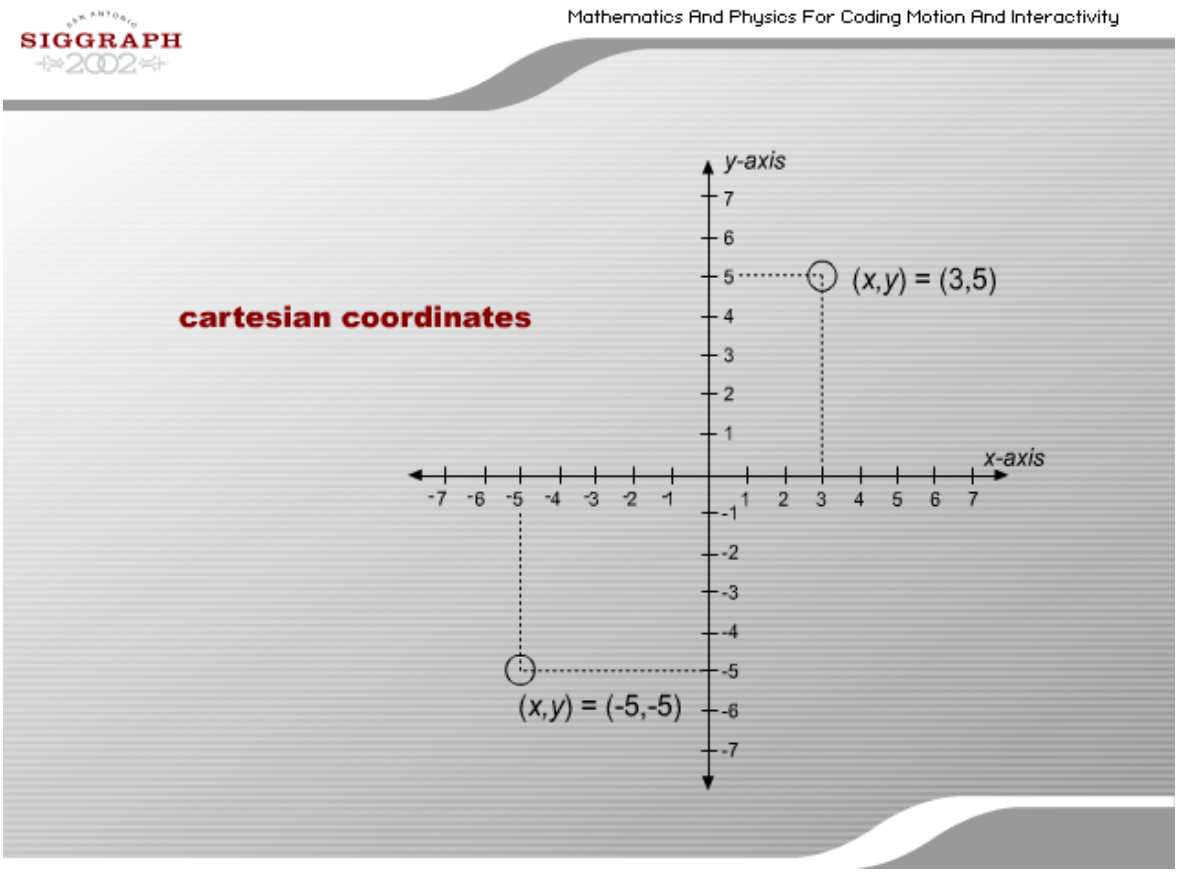

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Prototype-based
:: Pseudo-Classes - Prototypes
:: Objects
:: Methods and Functions

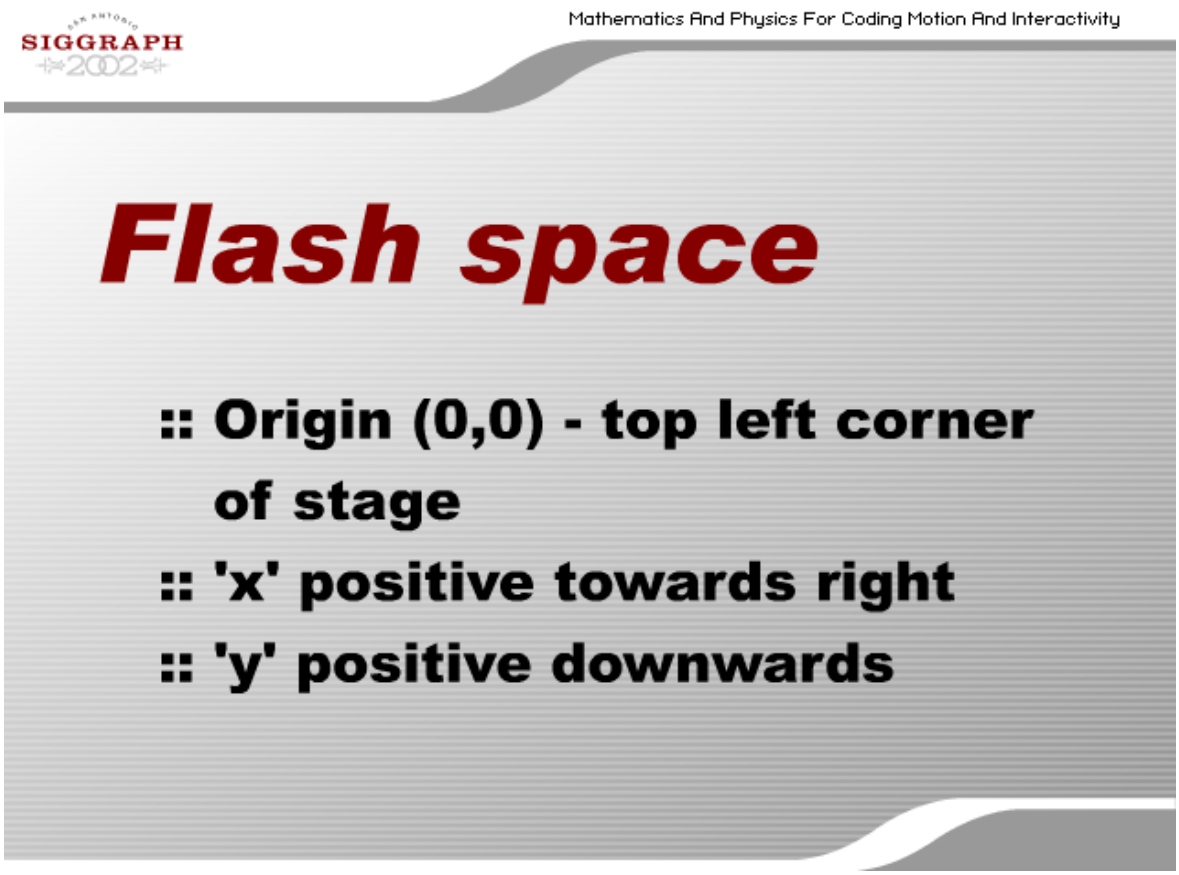

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Anti-Events
:: Events are good for simple actions
:: For rich applications, you need to learn OOP

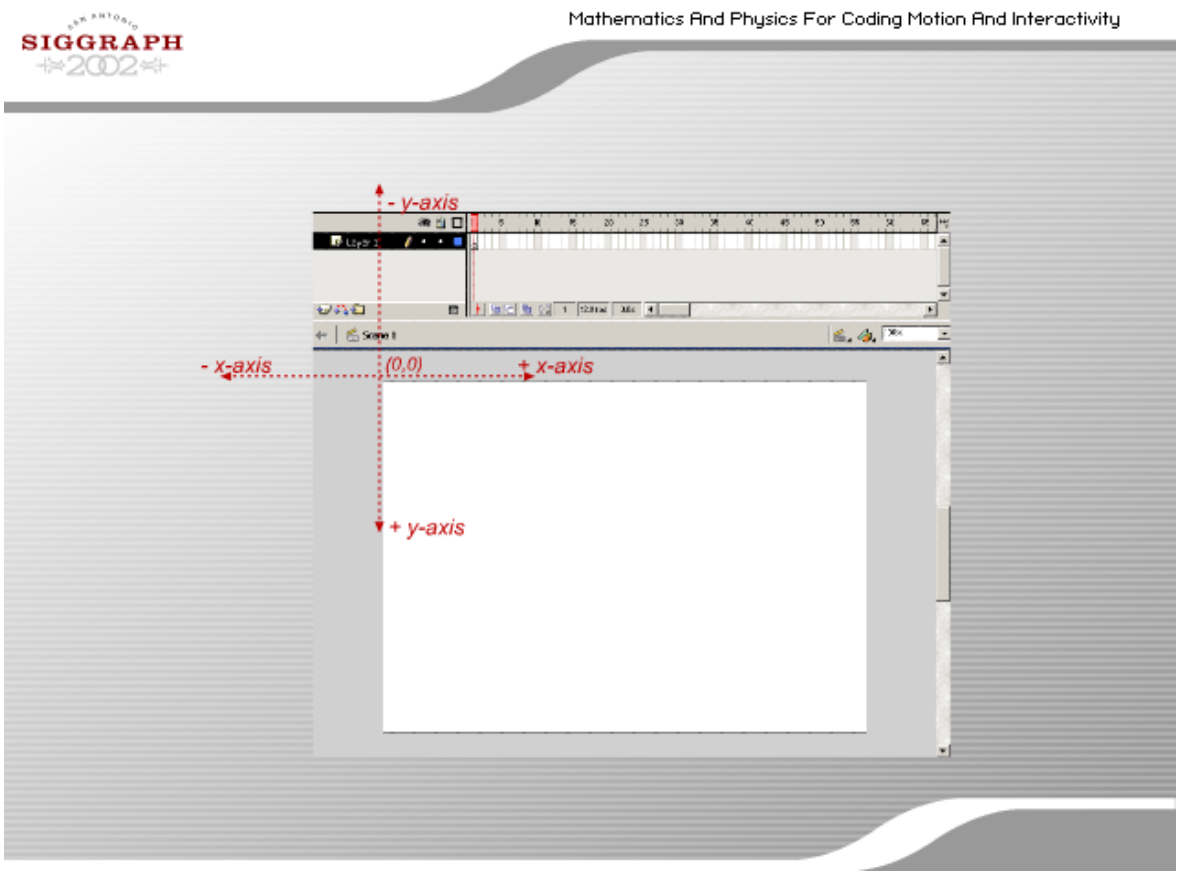

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
essential
MATHEMATICS
revisting the basics


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
OVERVIEW
:: Coordinate systems
:: Important concepts
:: Solving Equations
:: Linear sliders


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Cartesian
:: Most common system
:: Ordered pair - (x,y)

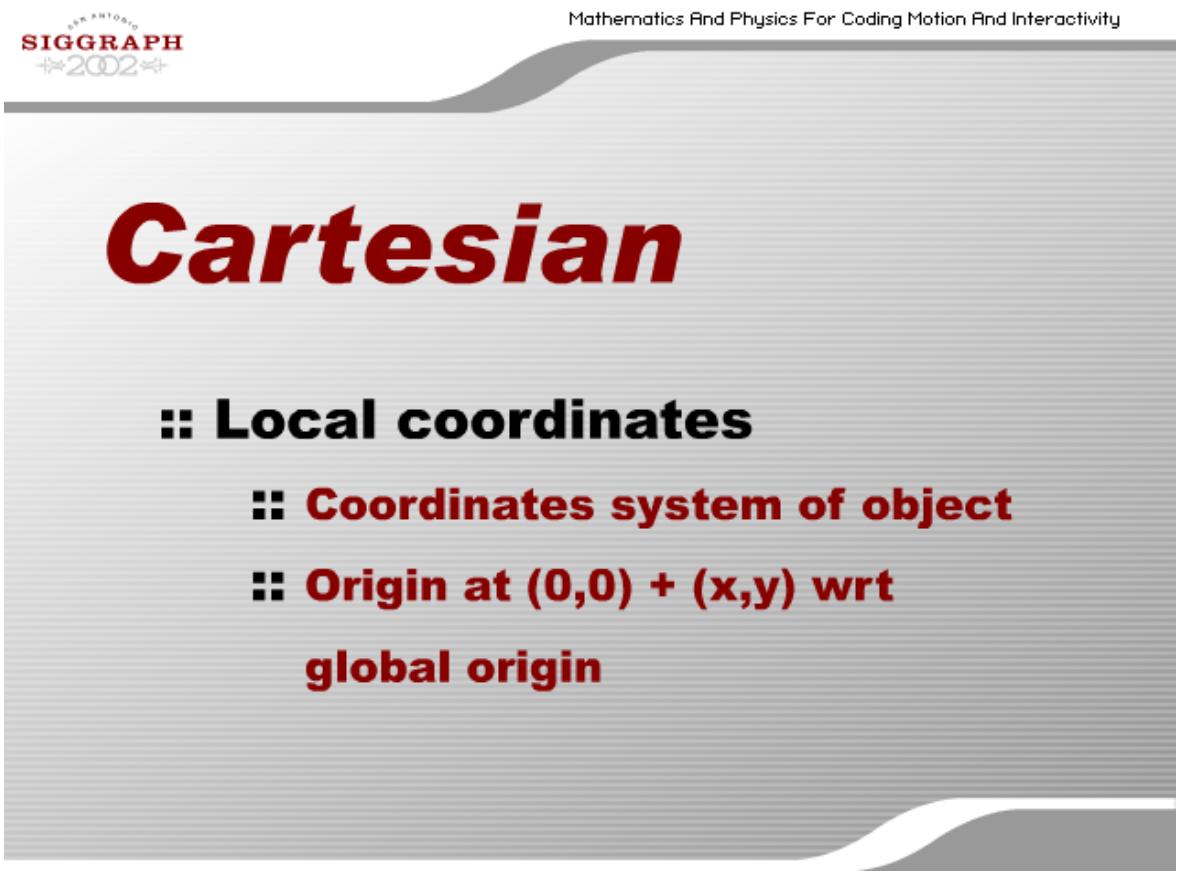

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
cartesian coordinates
y-axis
x-axis
(x,y) = (3,5)
(x,y) = (-5,-5)

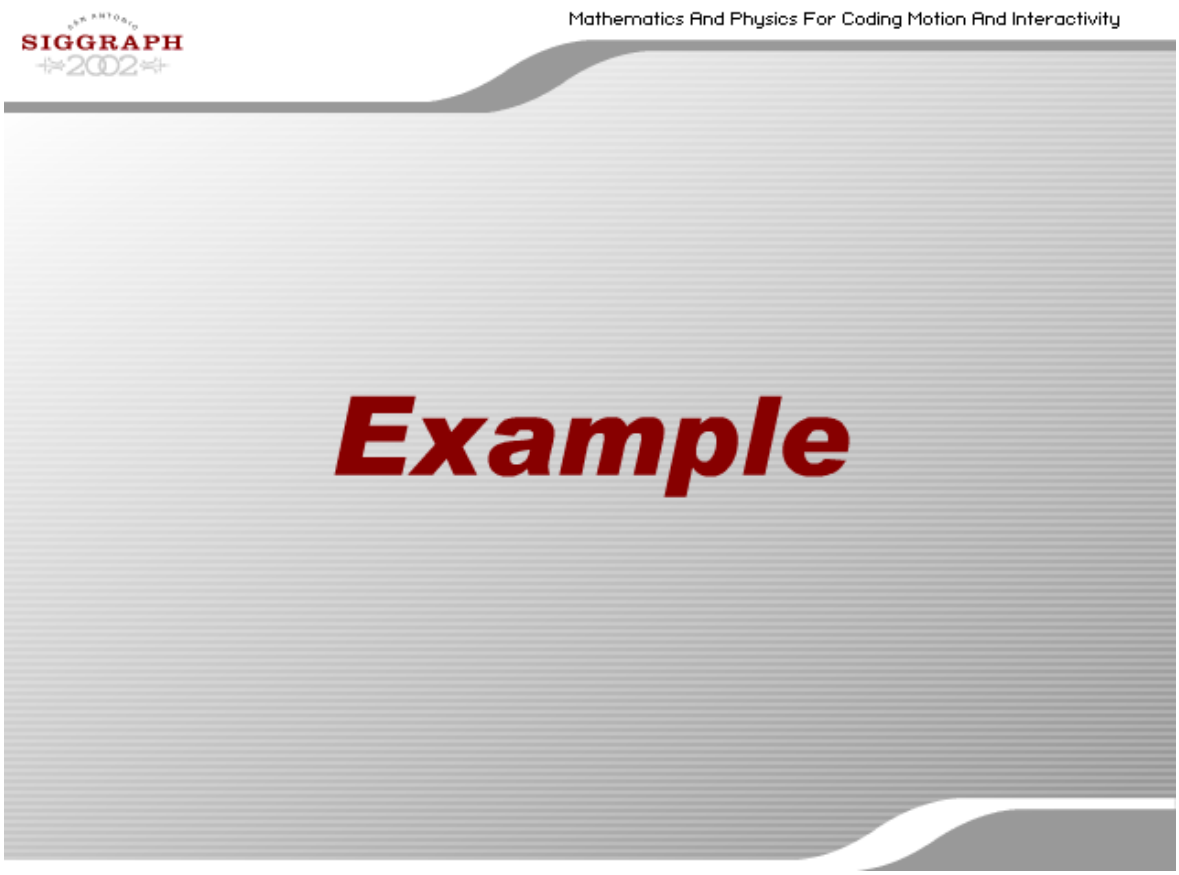

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Flash space
:: Origin (0,0) - top left corner of stage
:: 'x' positive towards right
:: 'y' positive downwards

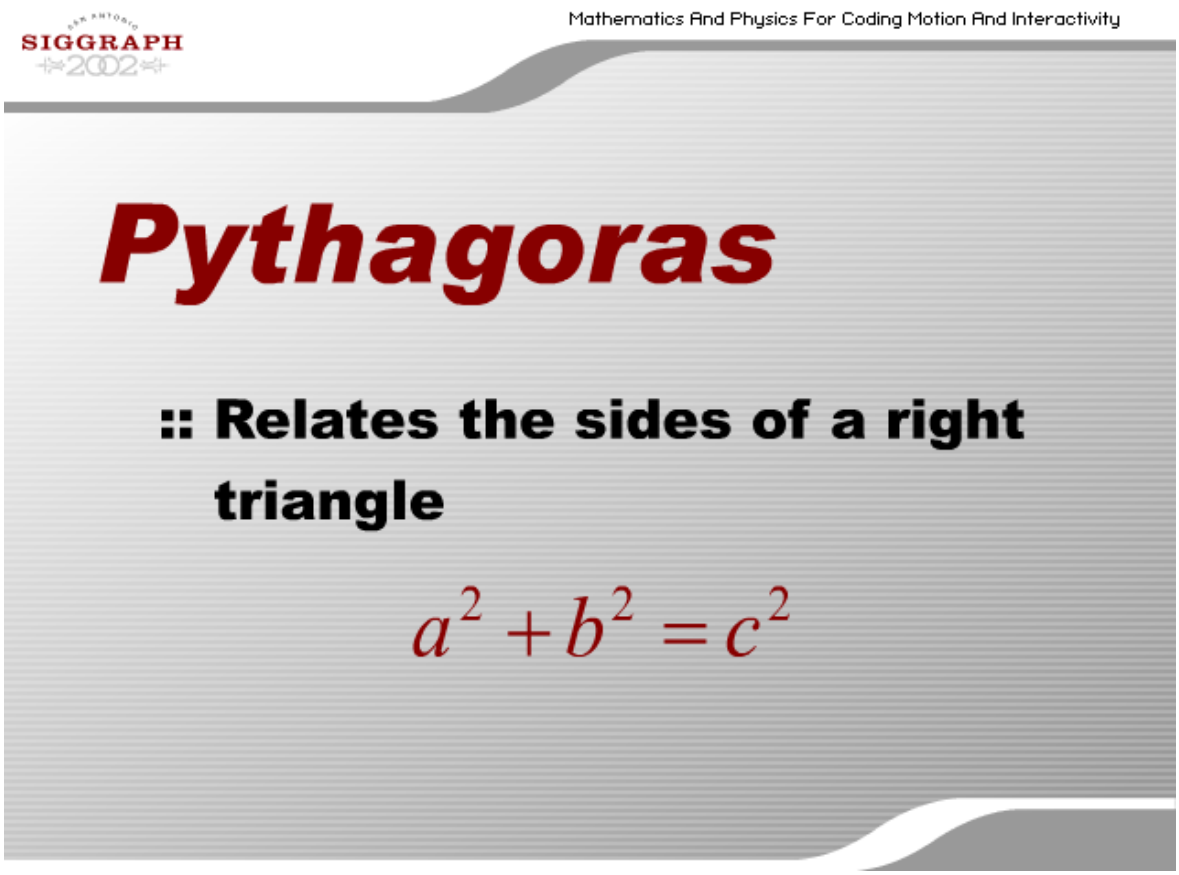

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
- y-axis
- x-axis
(0,0)
+ x-axis
+ y-axis

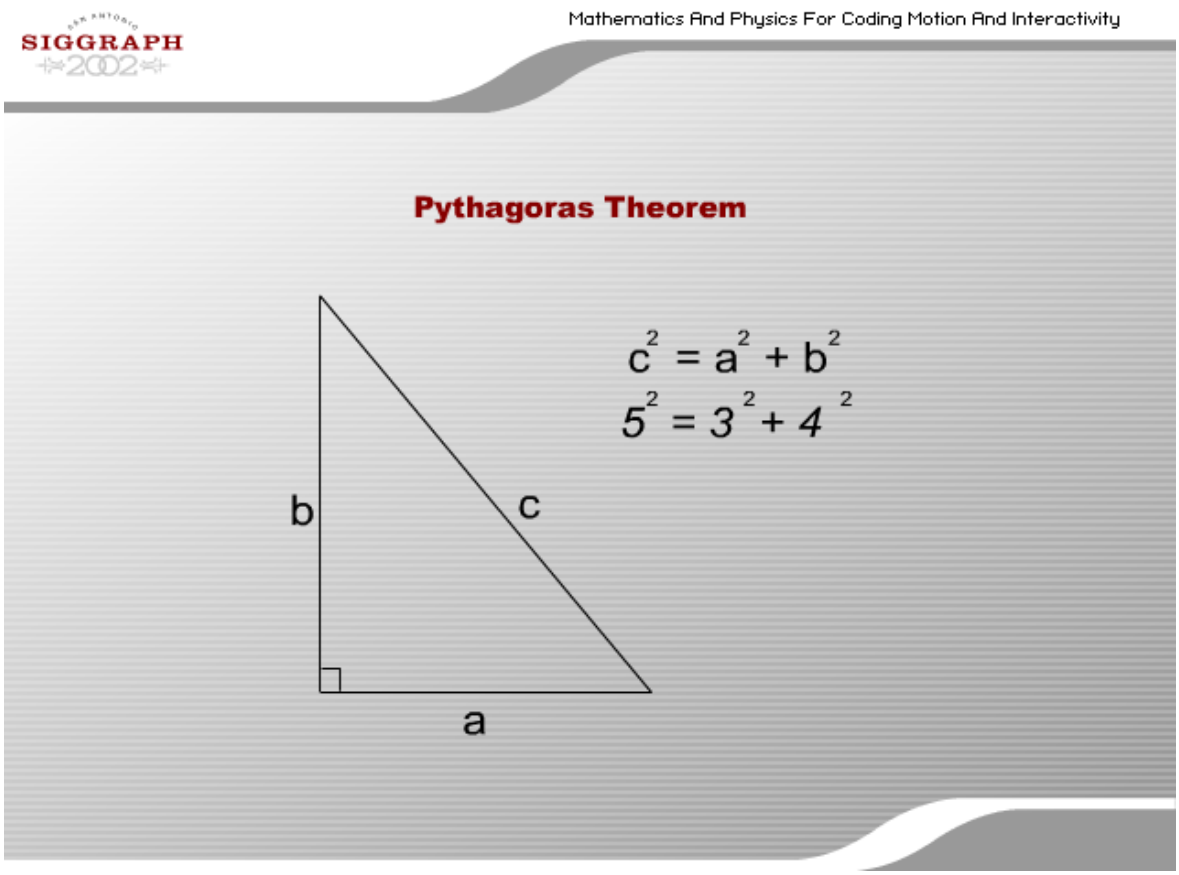

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Cartesian
:: Global Coordinates
:: Local Coordinates

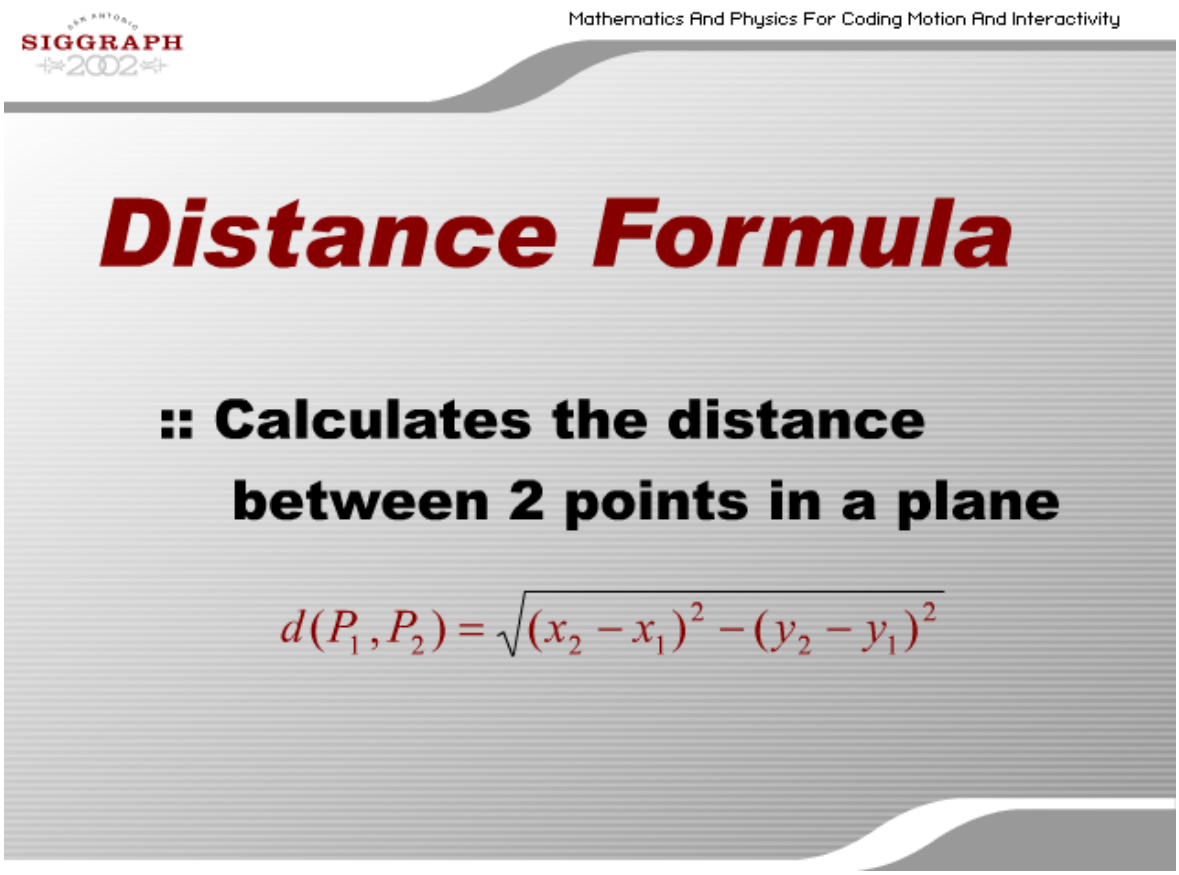

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Cartesian
:: Global Coordinates
:: World coordinate system
:: Origin at (0,0) on global axes

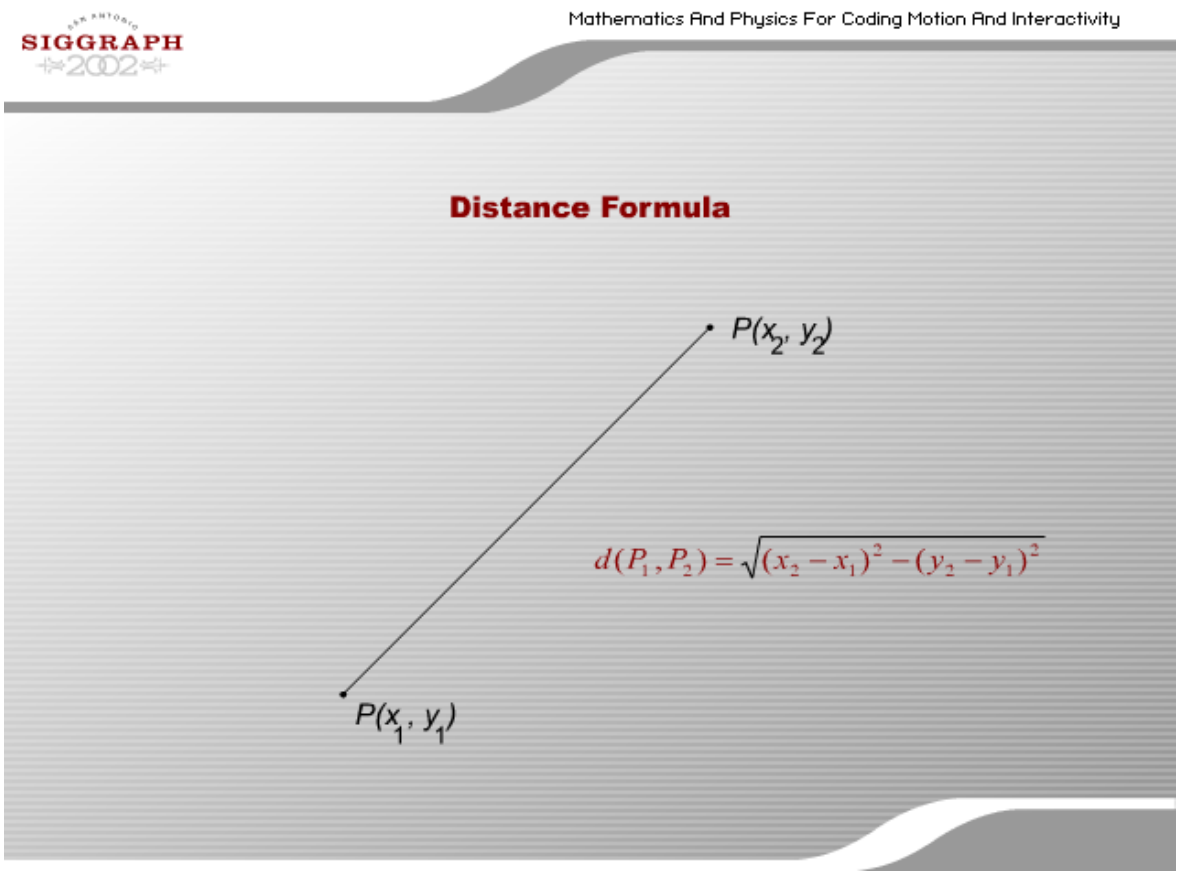

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Cartesian
:: Local coordinates
:: Coordinates system of object
:: Origin at (0,0) + (x,y) wrt
global origin

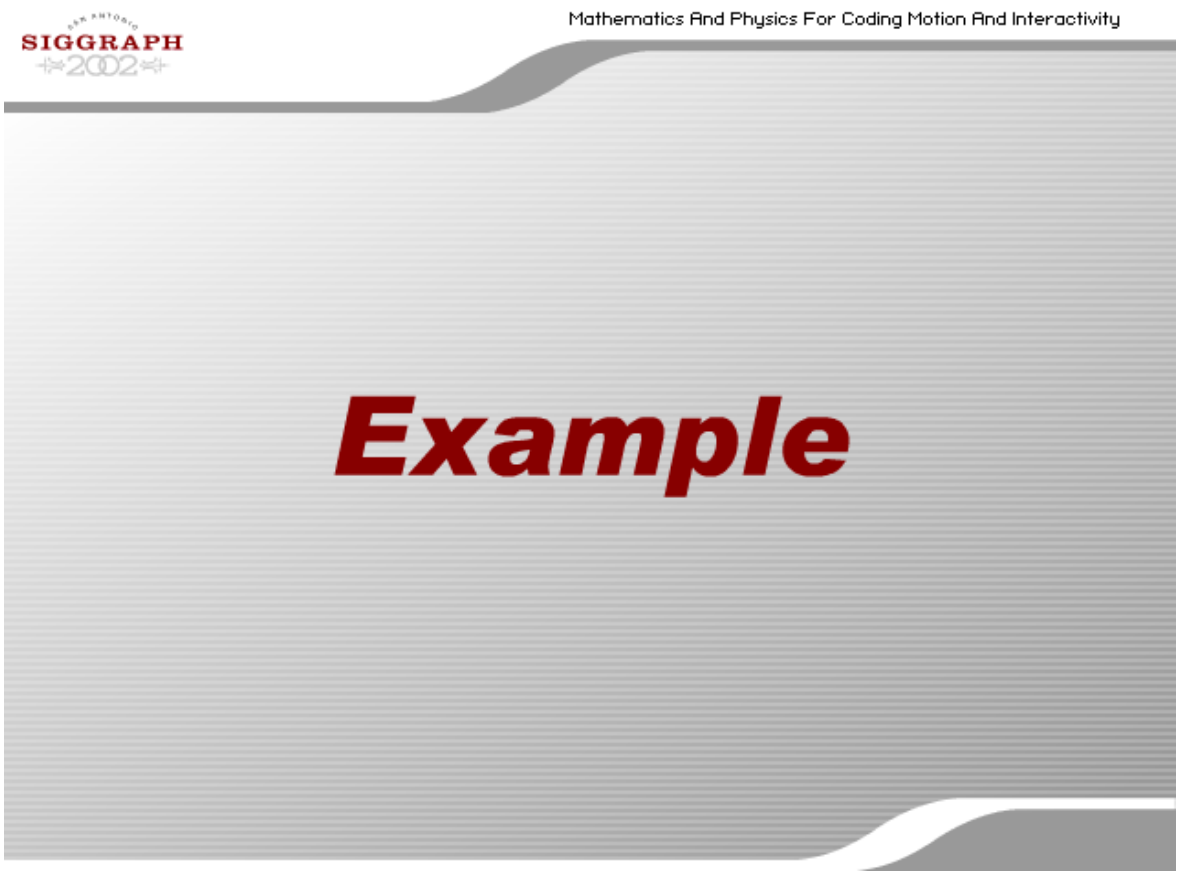

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

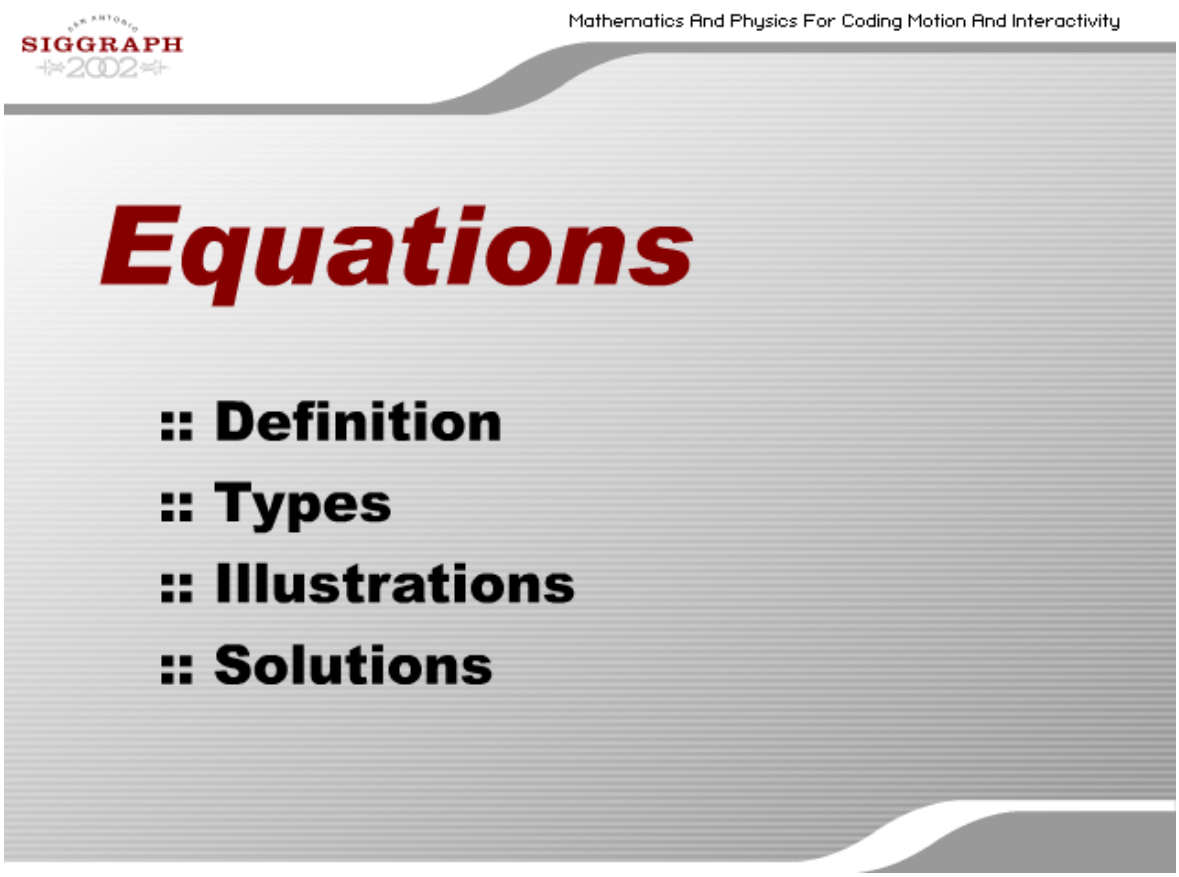

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Pythagoras
:: Relates the sides of a right triangle
$a^2 + b^2 = c^2$

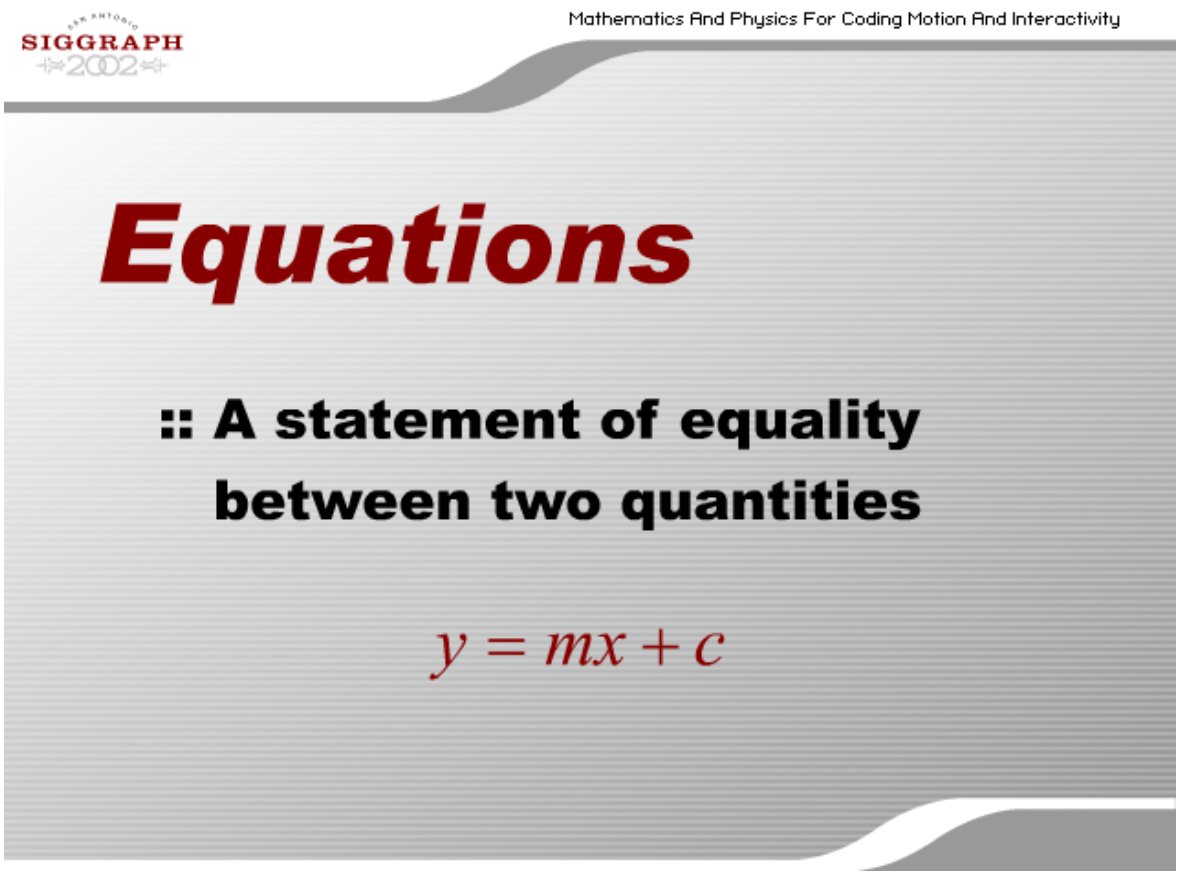

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Pythagoras Theorem
$c^2 = a^2 + b^2$
$5^2 = 3^2 + 4^2$
b
c
a

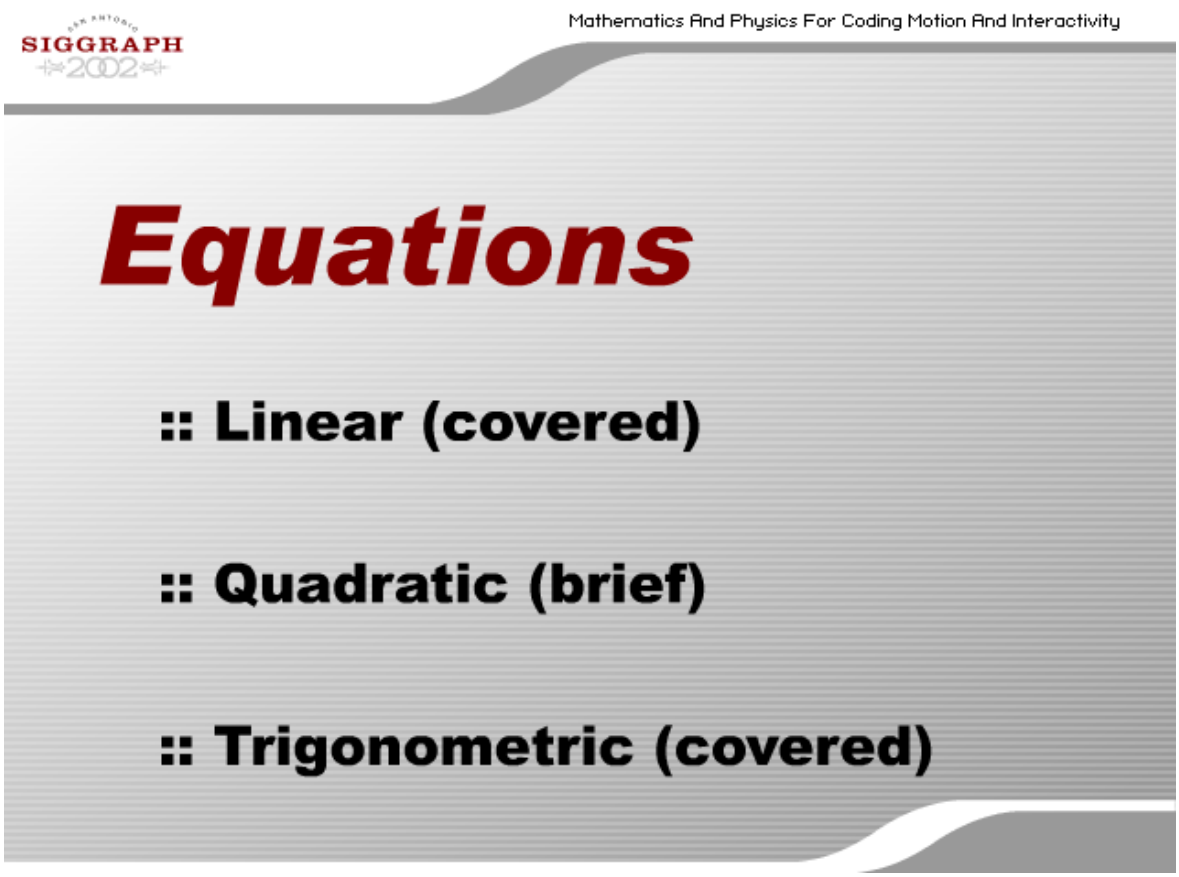

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Distance Formula
:: Calculates the distance between 2 points in a plane
$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2}$

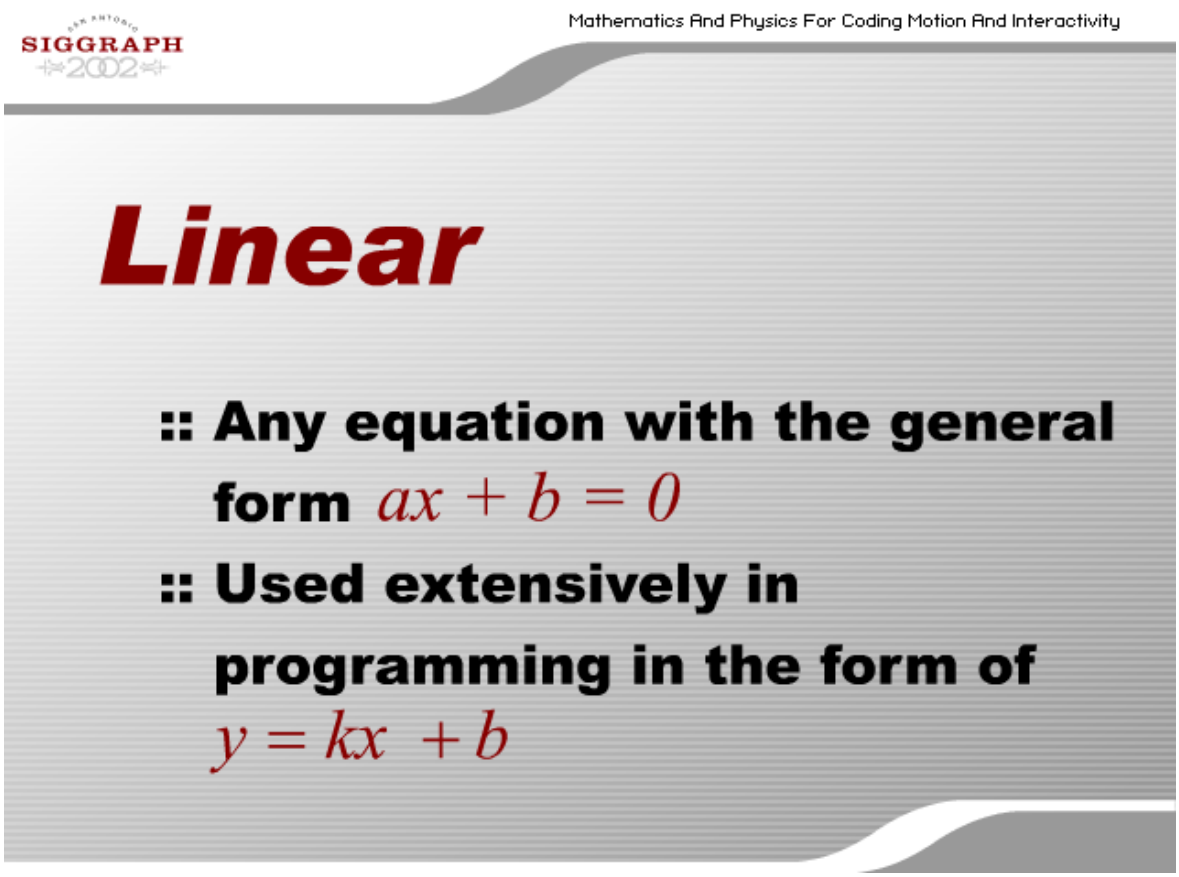

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Distance Formula
$P(x_2, y_2)$
$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2}$
$P(x_1, y_1)$

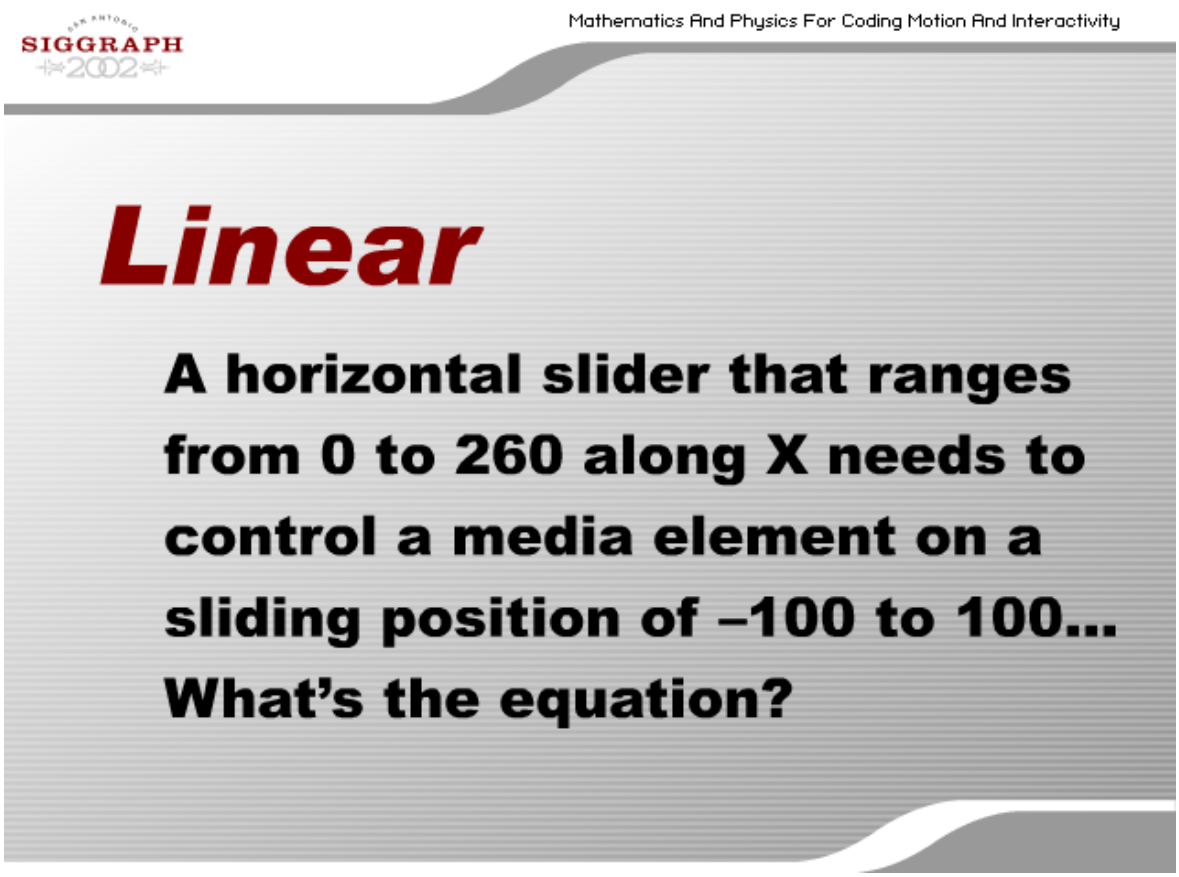

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

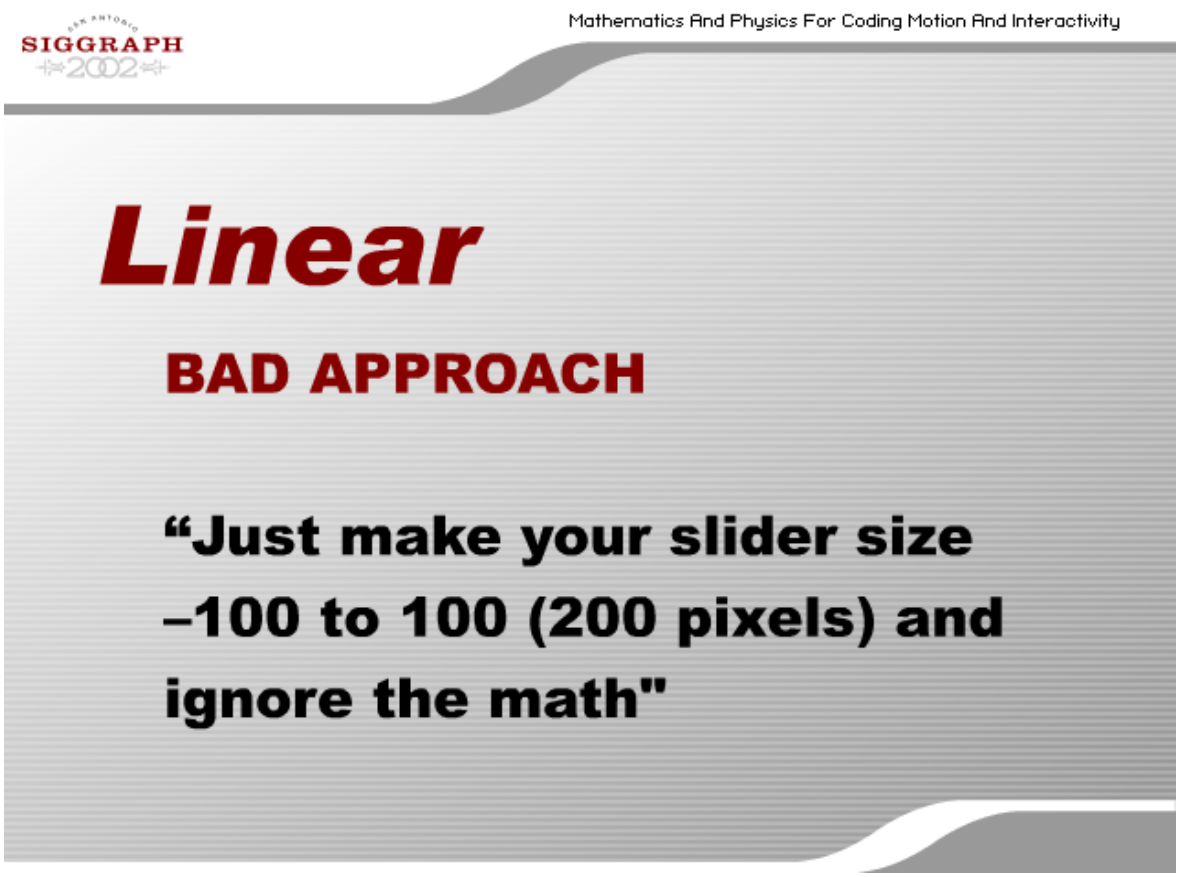

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Equations
:: Definition
:: Types
:: Illustrations
:: Solutions

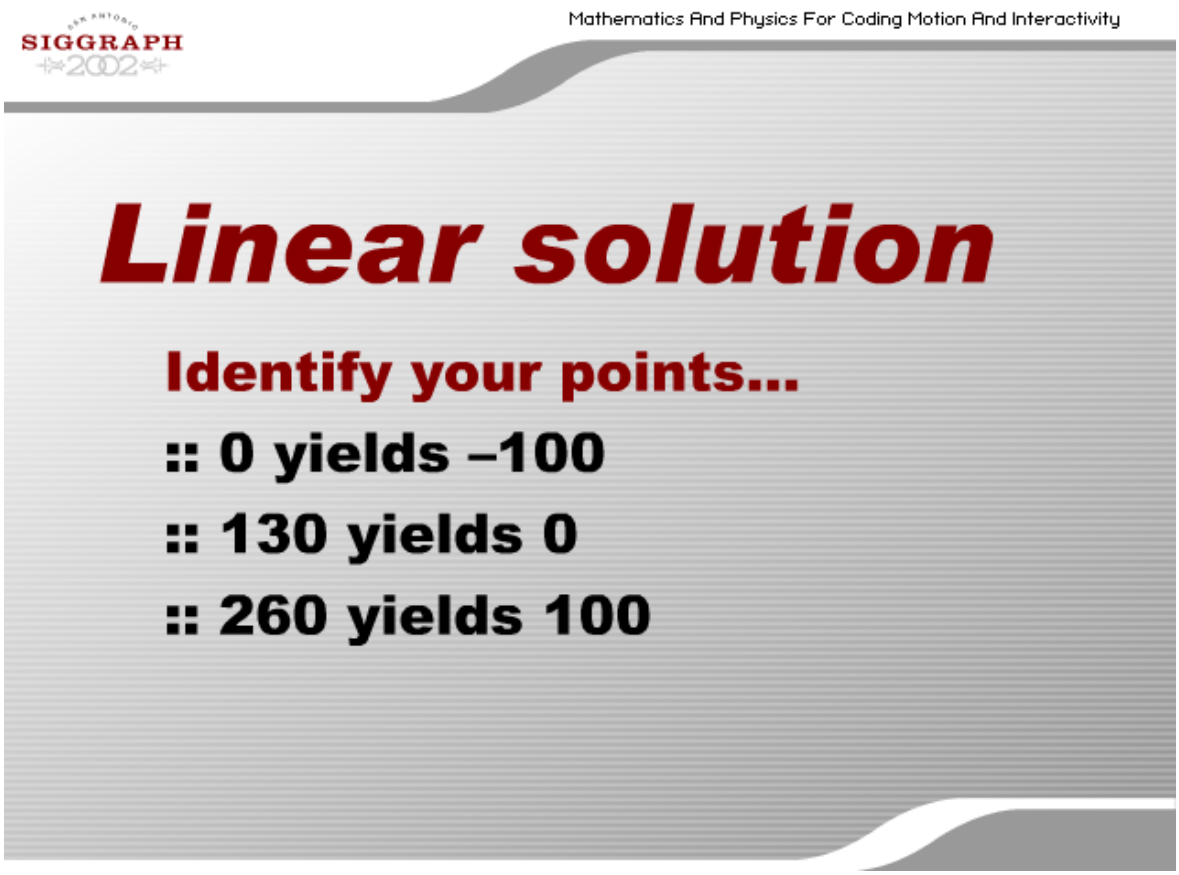

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Equations
:: A statement of equality between two quantities
$y = mx + c$

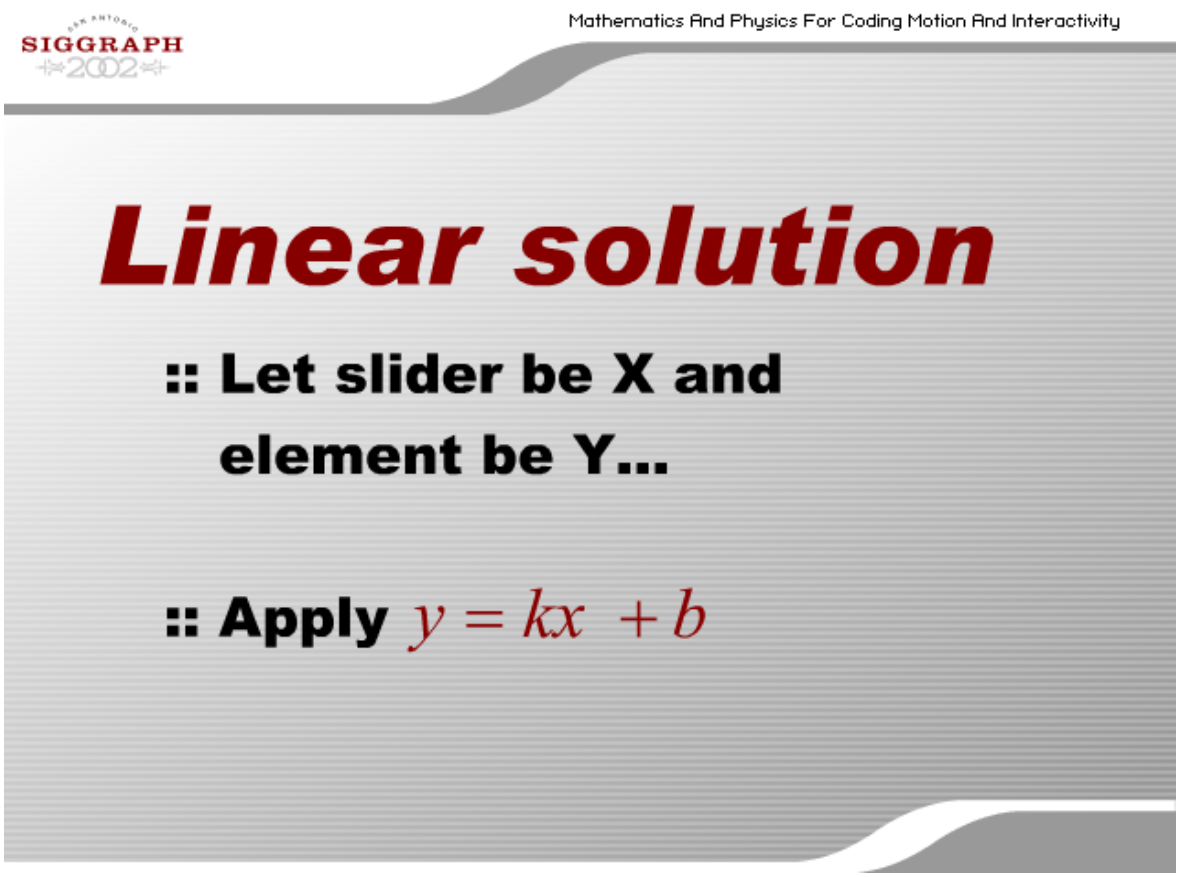

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Equations
:: Linear (covered)
:: Quadratic (brief)
:: Trigonometric (covered)

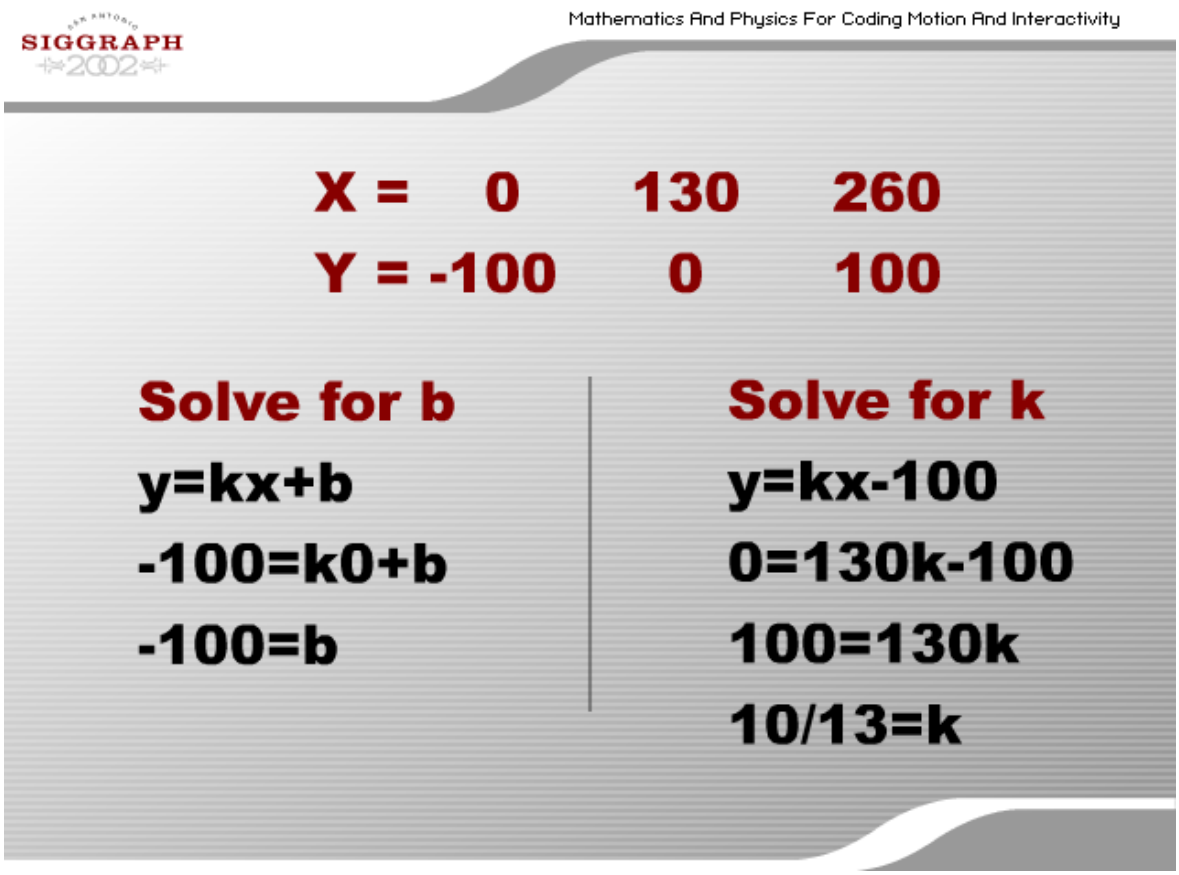

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Linear
:: Any equation with the general form $ax + b = 0$
:: Used extensively in programming in the form of $y = kx + b$

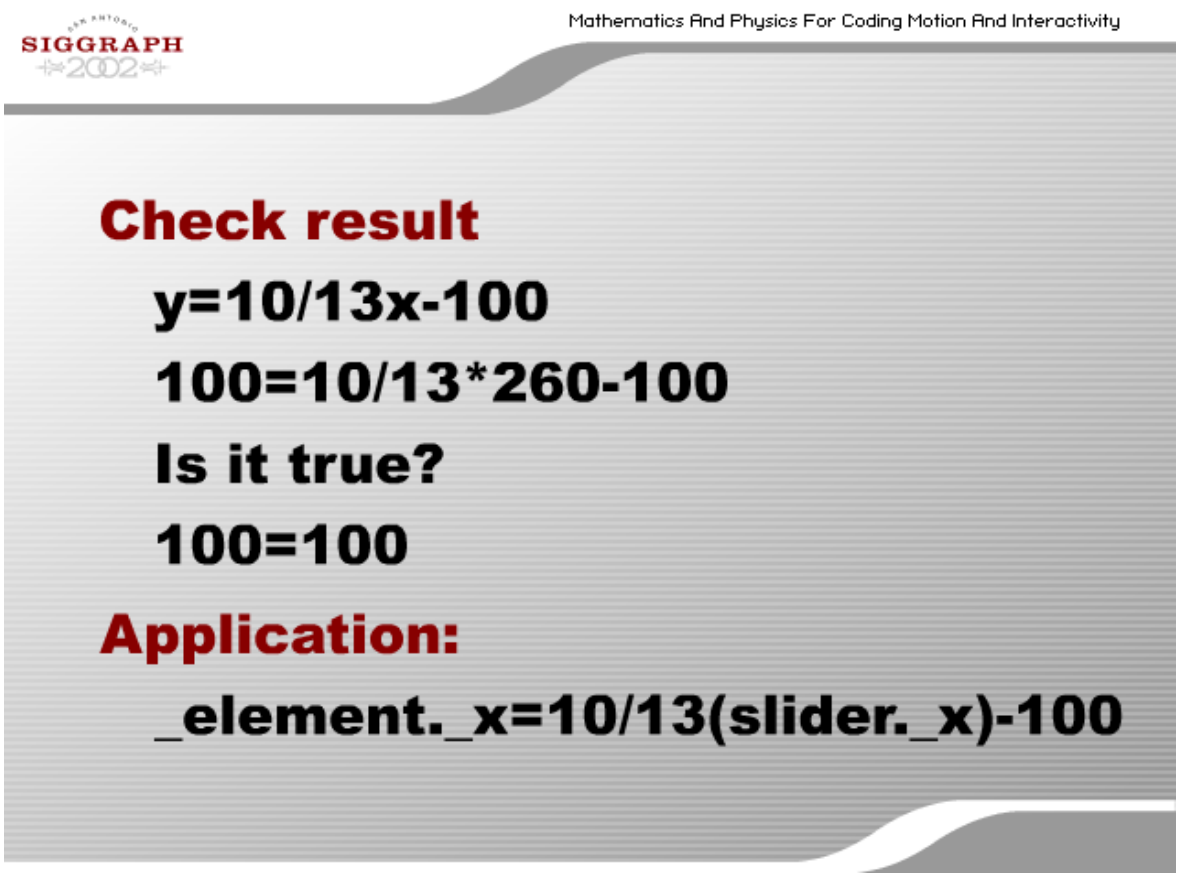

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Linear
A horizontal slider that ranges from 0 to 260 along X needs to control a media element on a sliding position of –100 to 100... What's the equation?

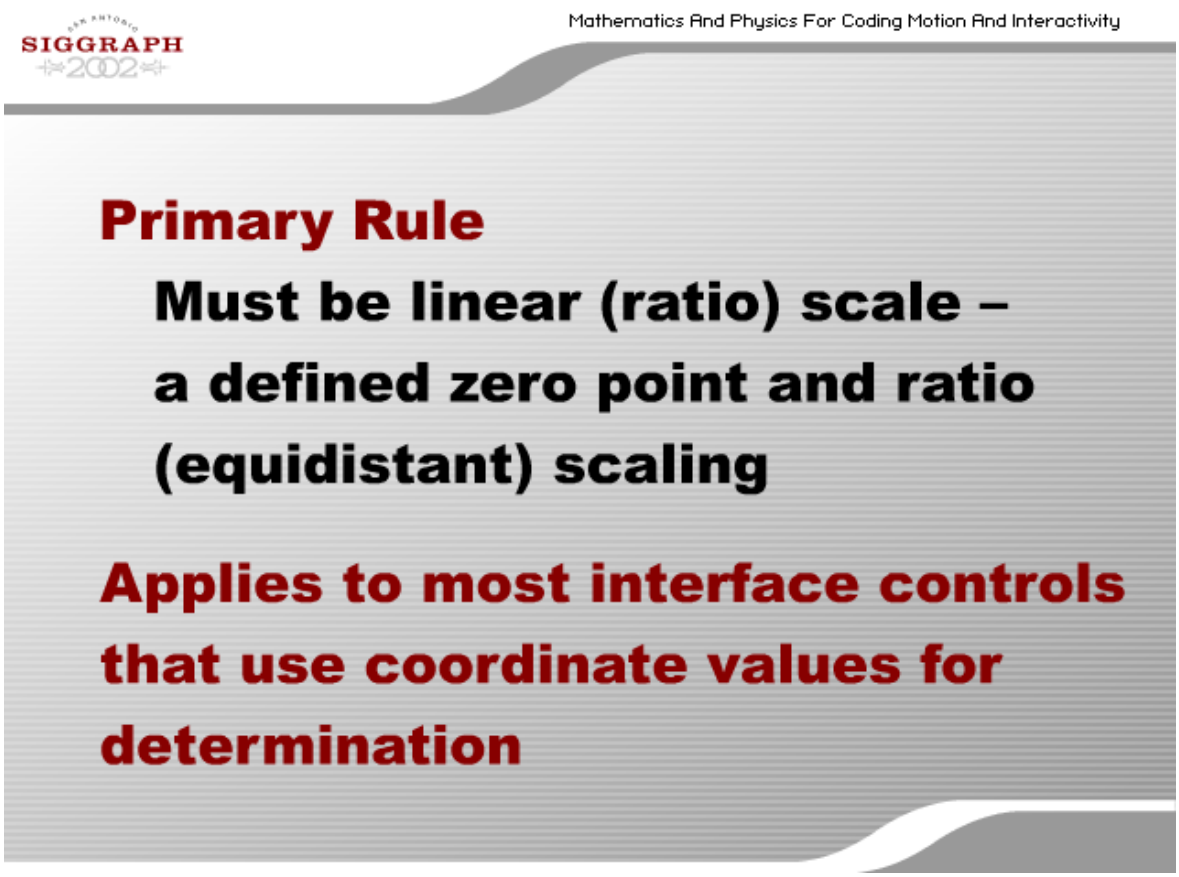

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Linear
BAD APPROACH
“Just make your slider size –100 to 100 (200 pixels) and ignore the math"

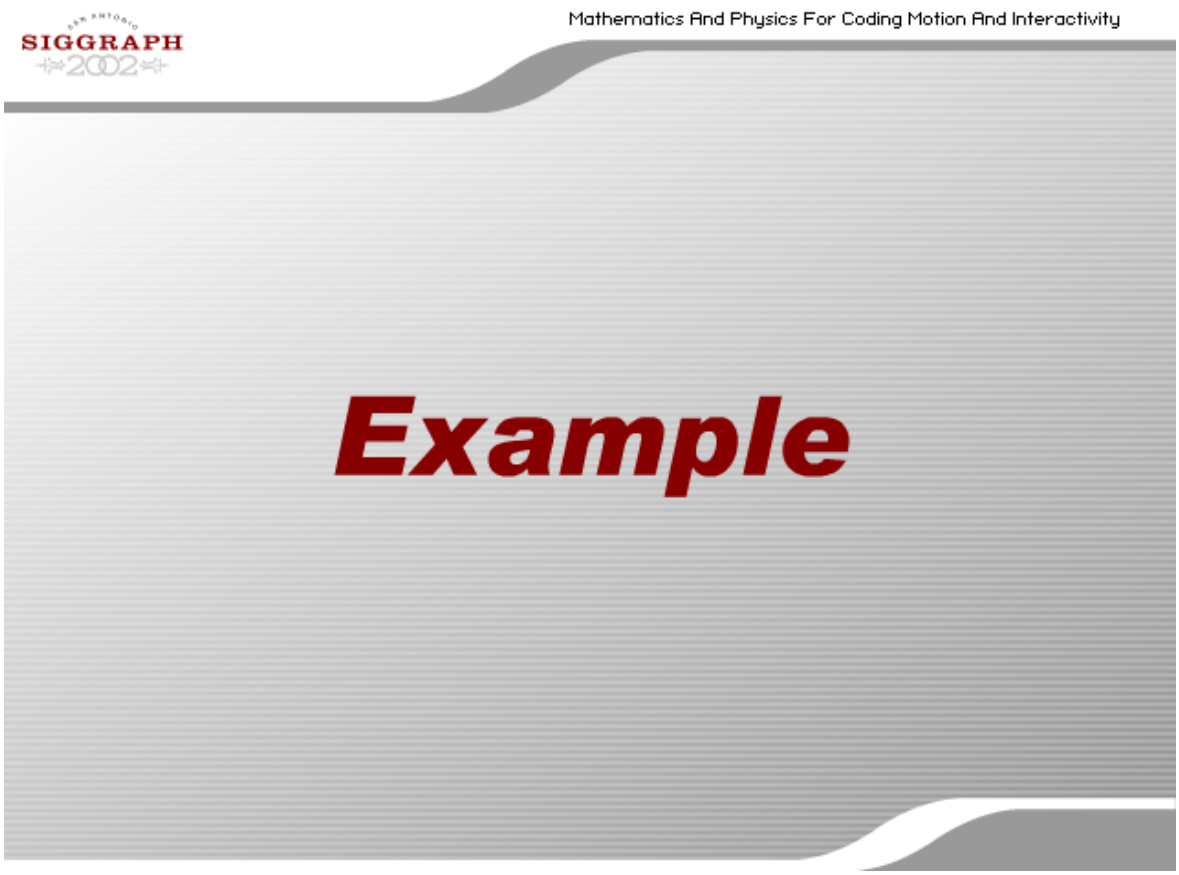

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Linear solution
Identify your points...
:: 0 yields –100
:: 130 yields 0
:: 260 yields 100

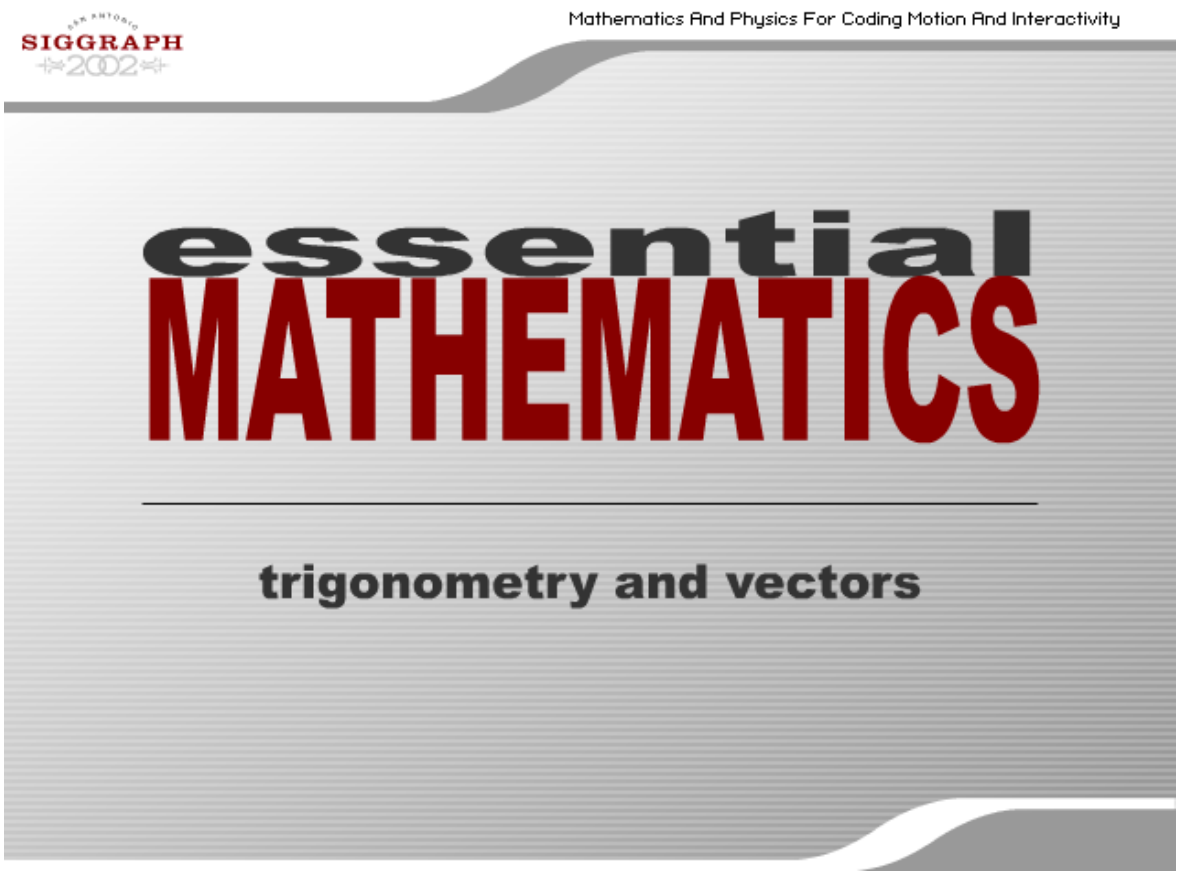

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Linear solution
:: Let slider be X and element be Y...
:: Apply $y = kx + b$

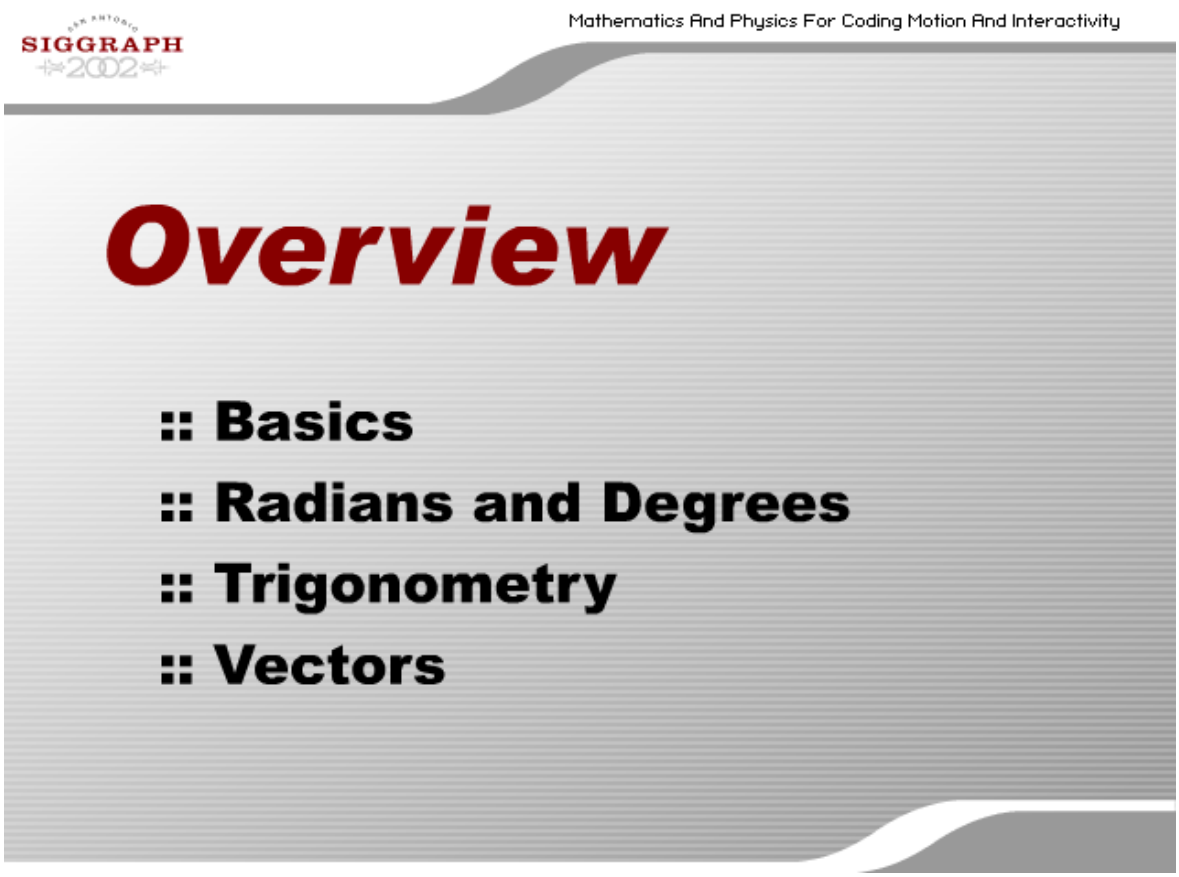

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
X = 0 130 260
Y = -100 0 100
Solve for b
y=kx+b
-100=k0+b
-100=b
Solve for k
y=kx-100
0=130k-100
100=130k
10/13=k

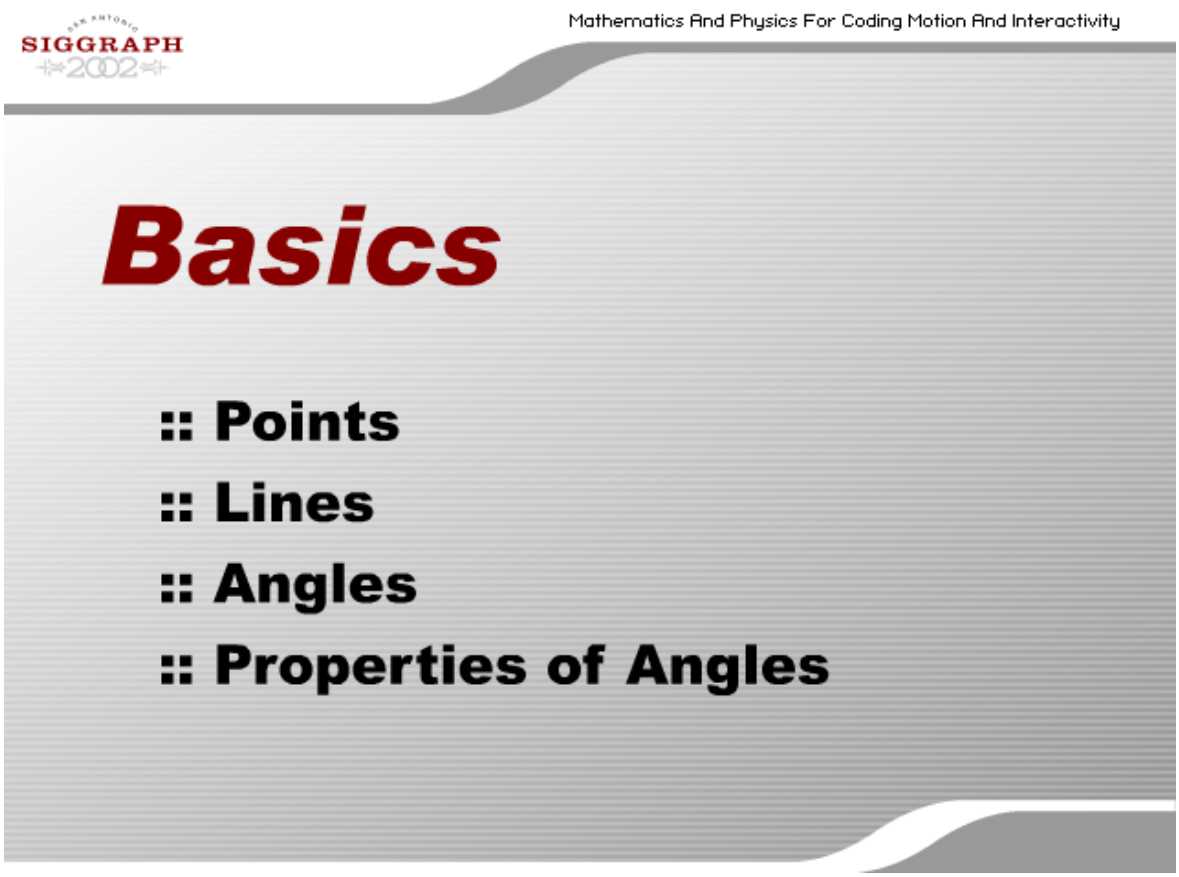

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Check result
y=10/13x-100
100=10/13*260-100
Is it true?
100=100
Application:
_element._x=10/13(slider._x)-100

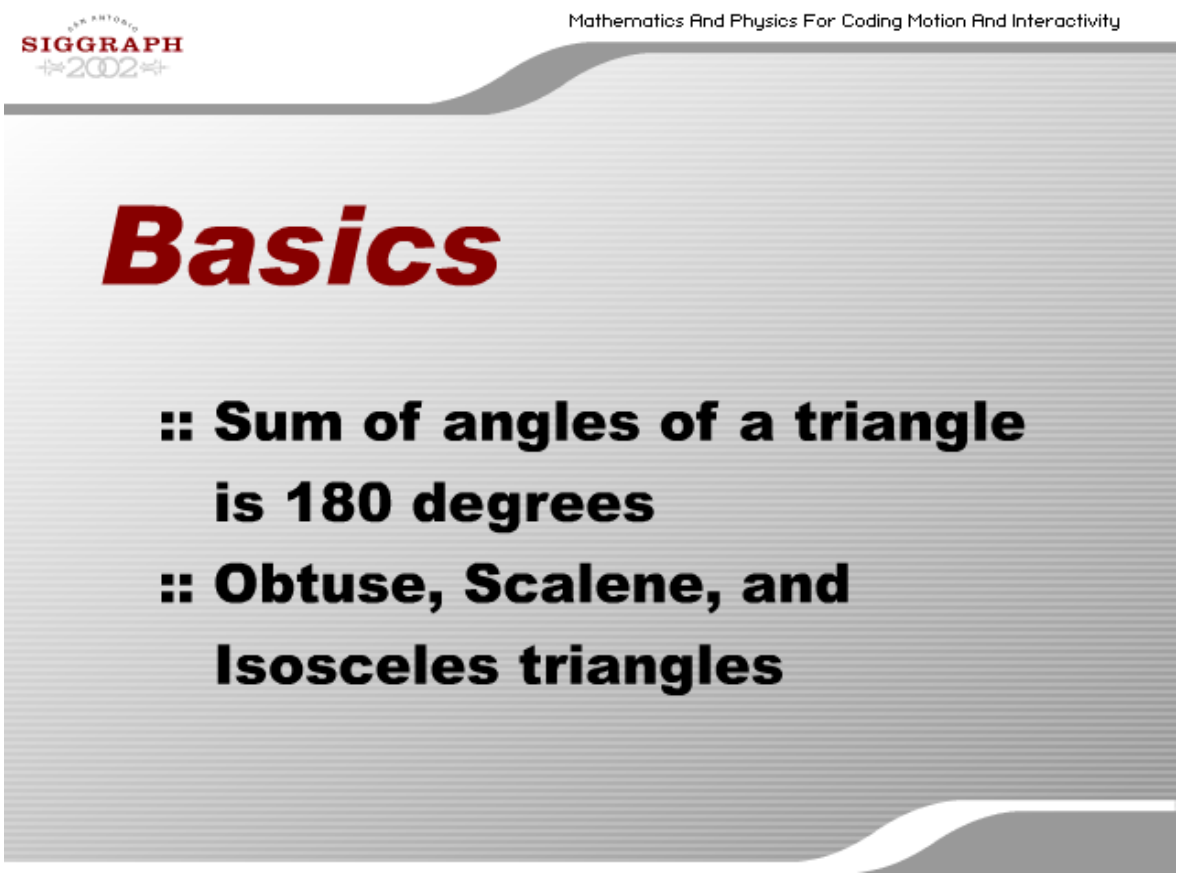

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Primary Rule
Must be linear (ratio) scale –
a defined zero point and ratio
(equidistant) scaling
Applies to most interface controls
that use coordinate values for
determination

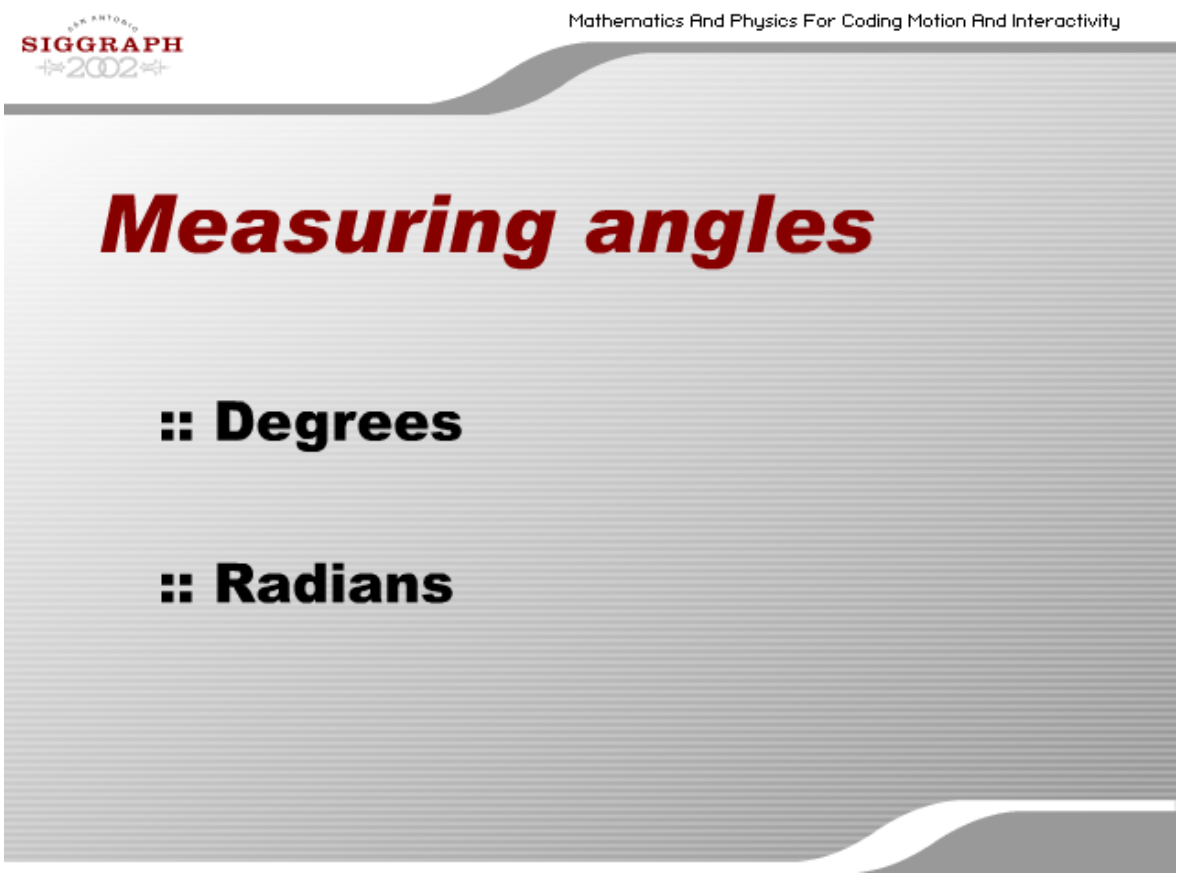

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

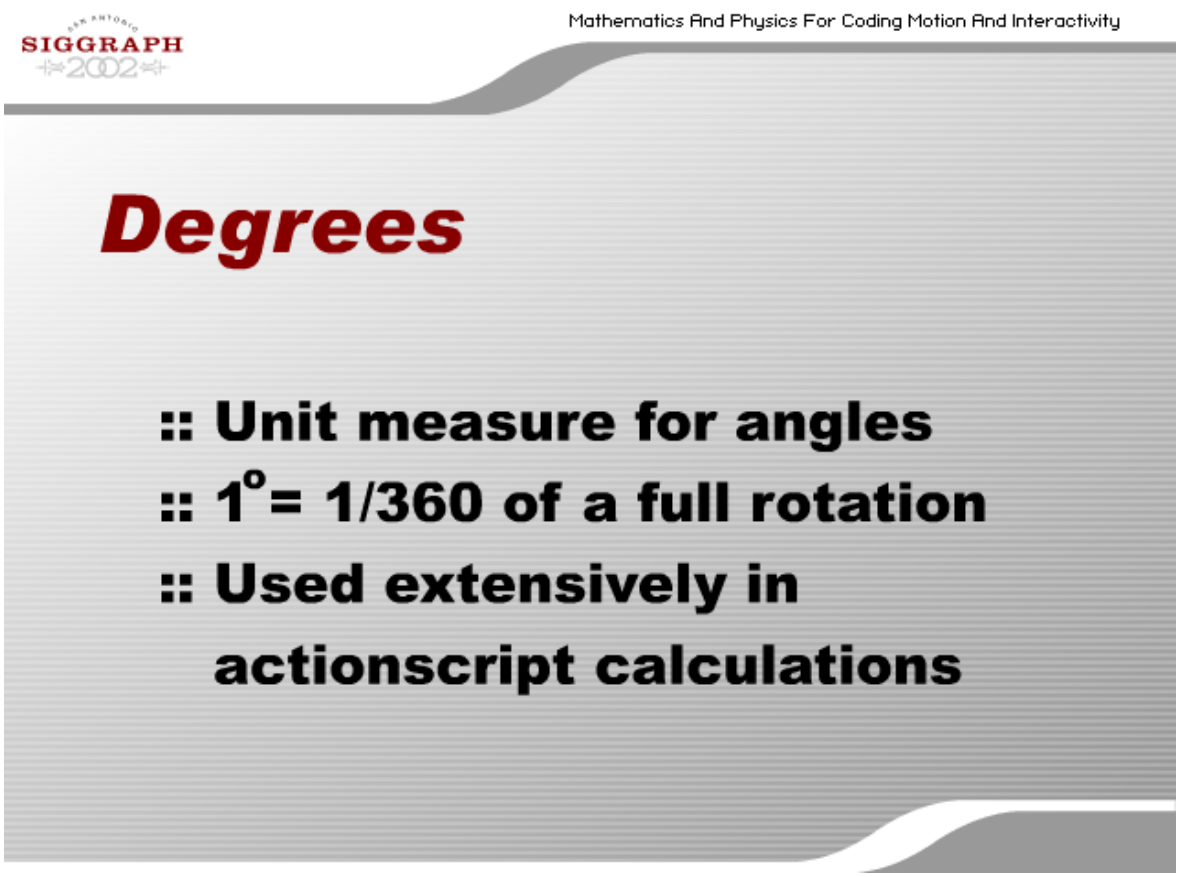

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
essential
MATHEMATICS
trigonometry and vectors

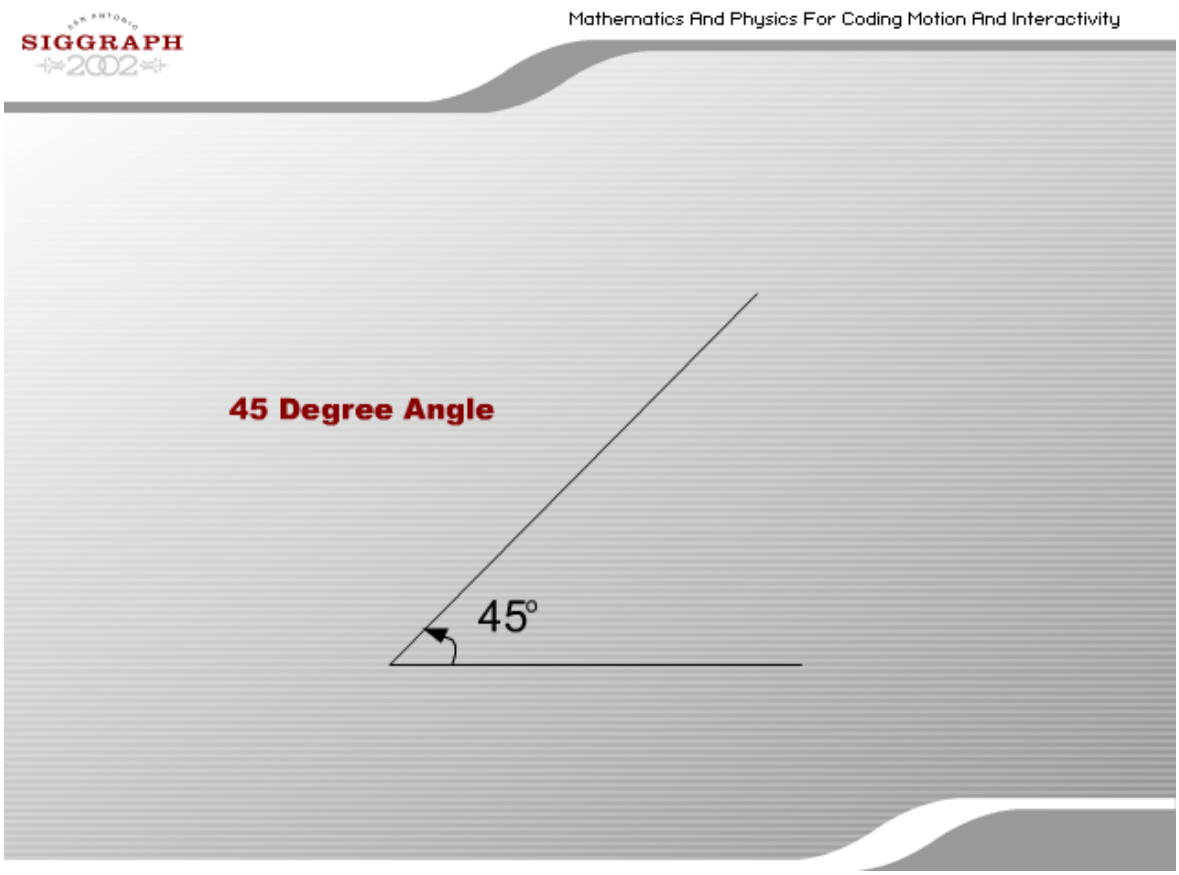

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Overview
:: Basics
:: Radians and Degrees
:: Trigonometry
:: Vectors

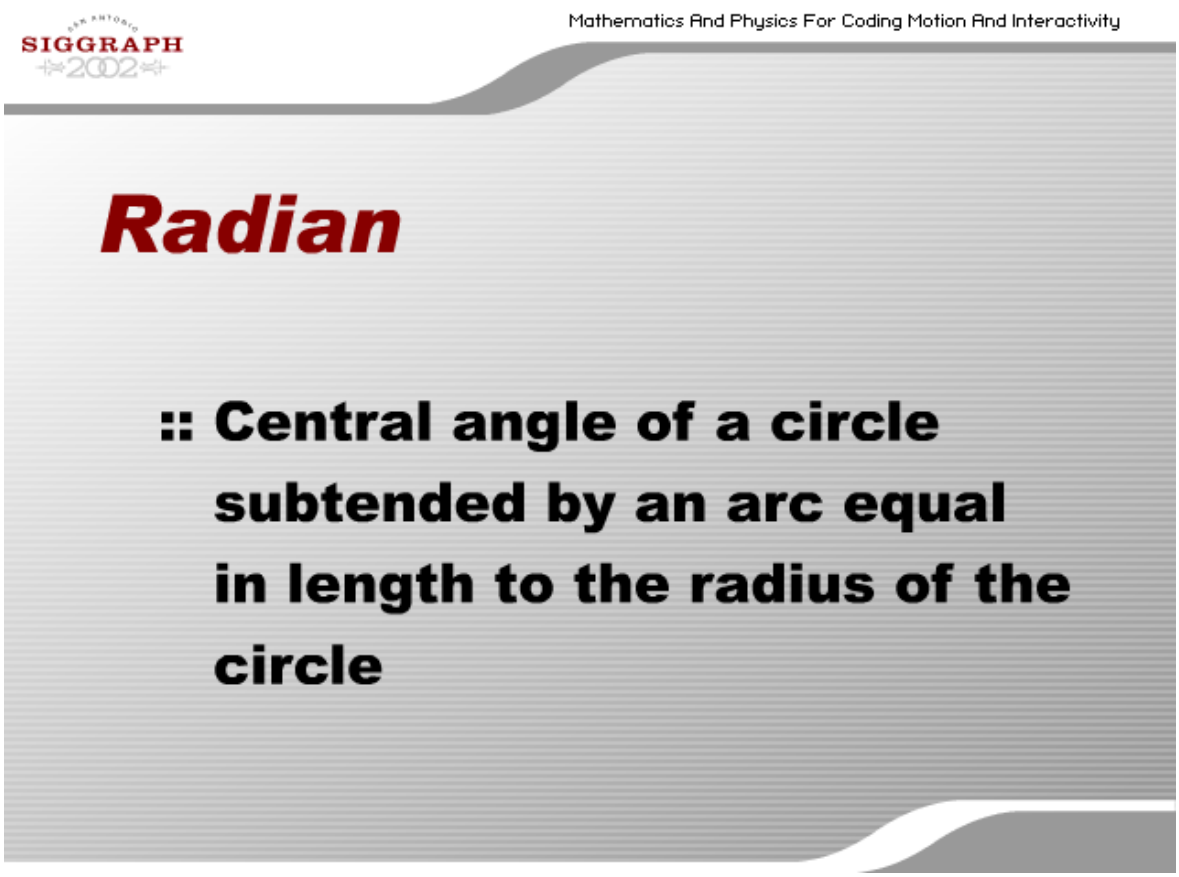

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Basics
:: Points
:: Lines
:: Angles
:: Properties of Angles

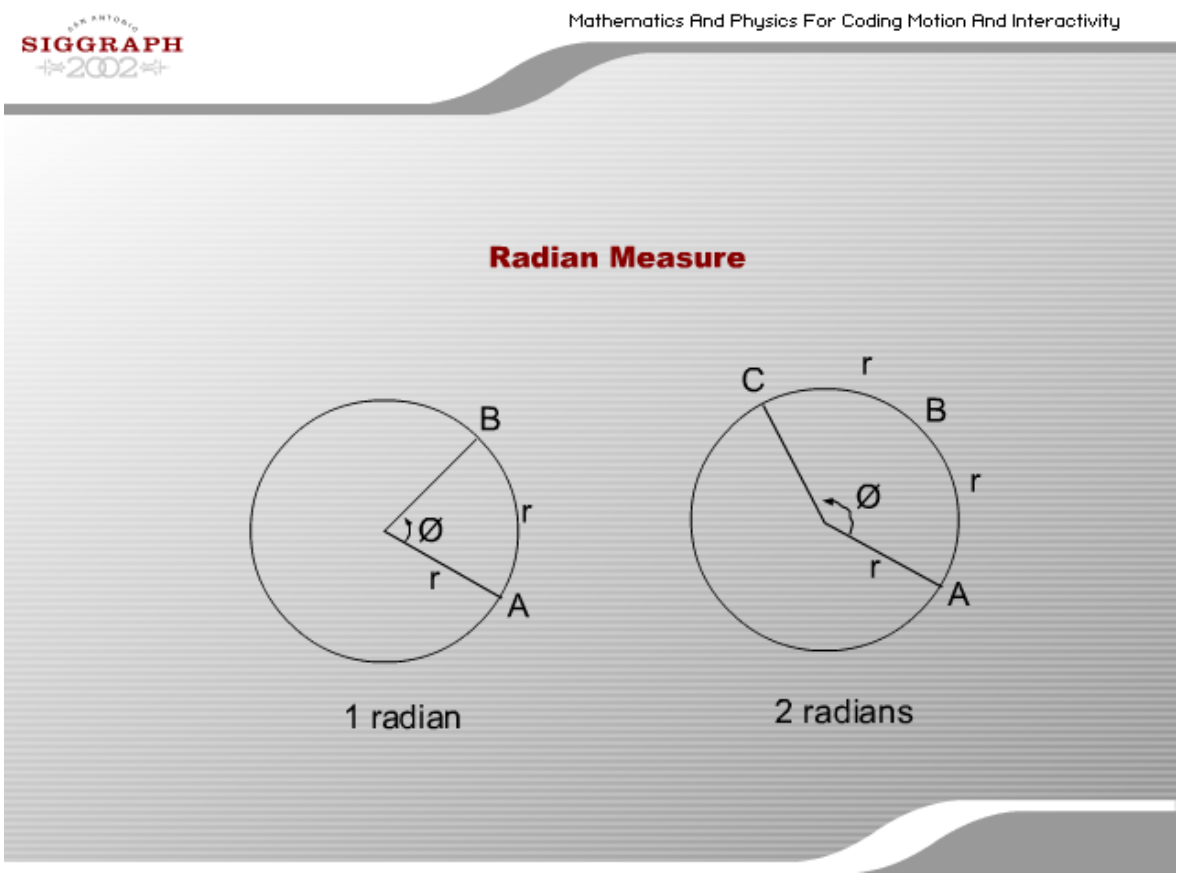

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Basics
:: Sum of angles of a triangle is 180 degrees
:: Obtuse, Scalene, and Isosceles triangles

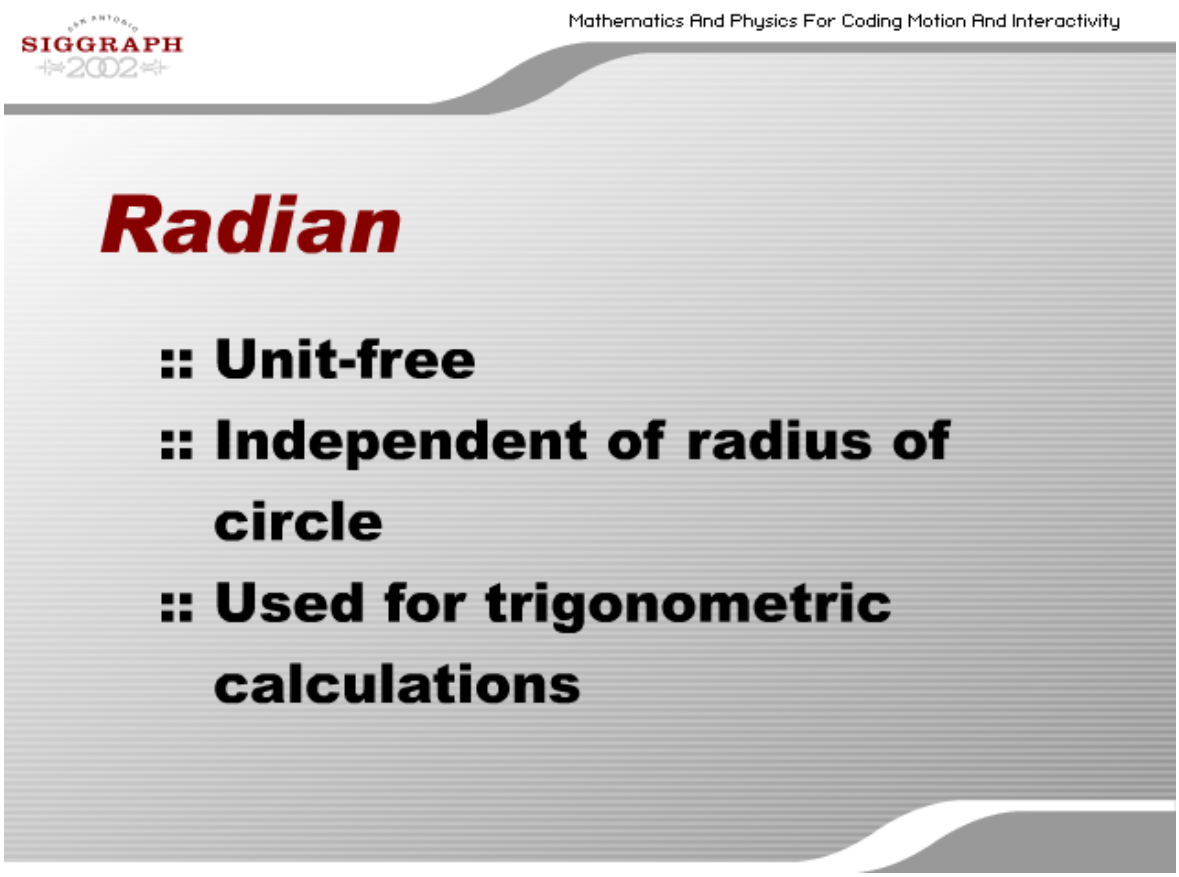

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Measuring angles
:: Degrees
:: Radians

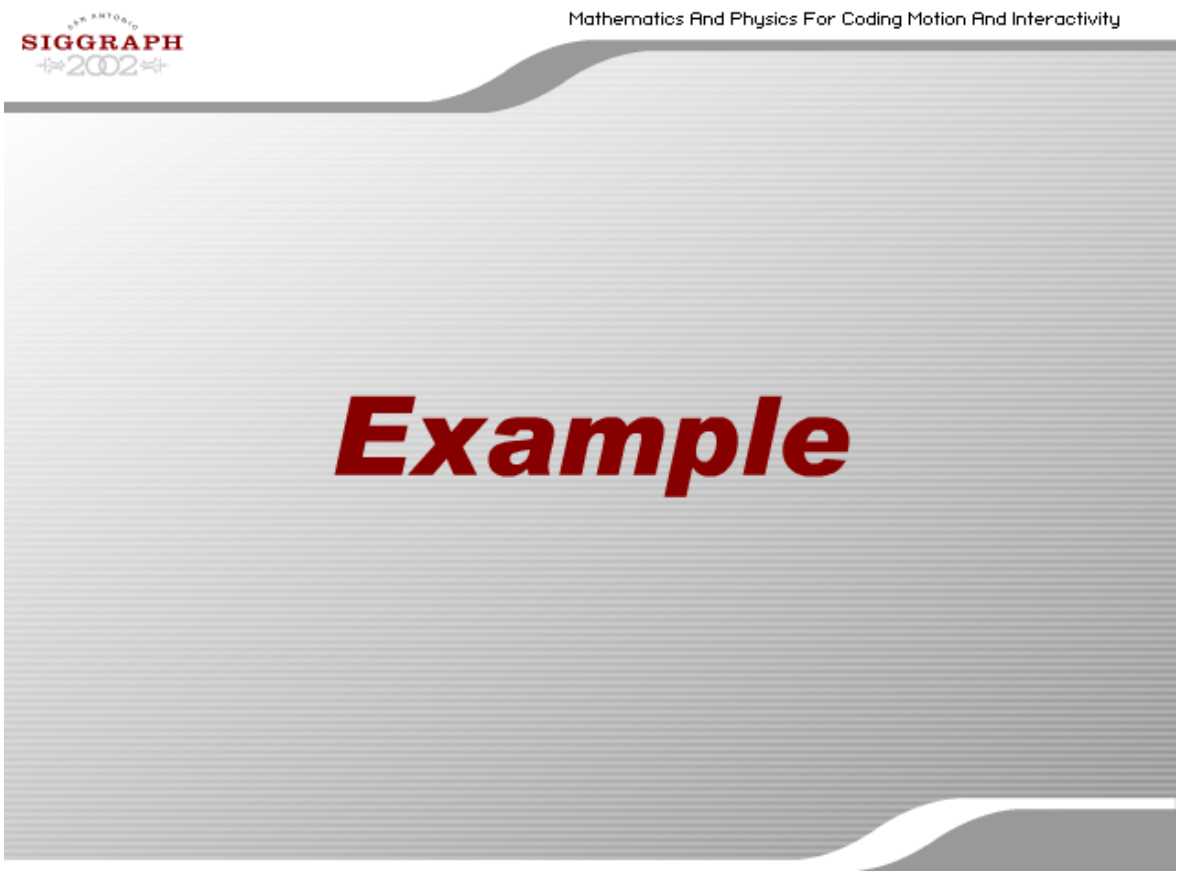

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Degrees
:: Unit measure for angles
:: 1°= 1/360 of a full rotation
:: Used extensively in actionscript calculations

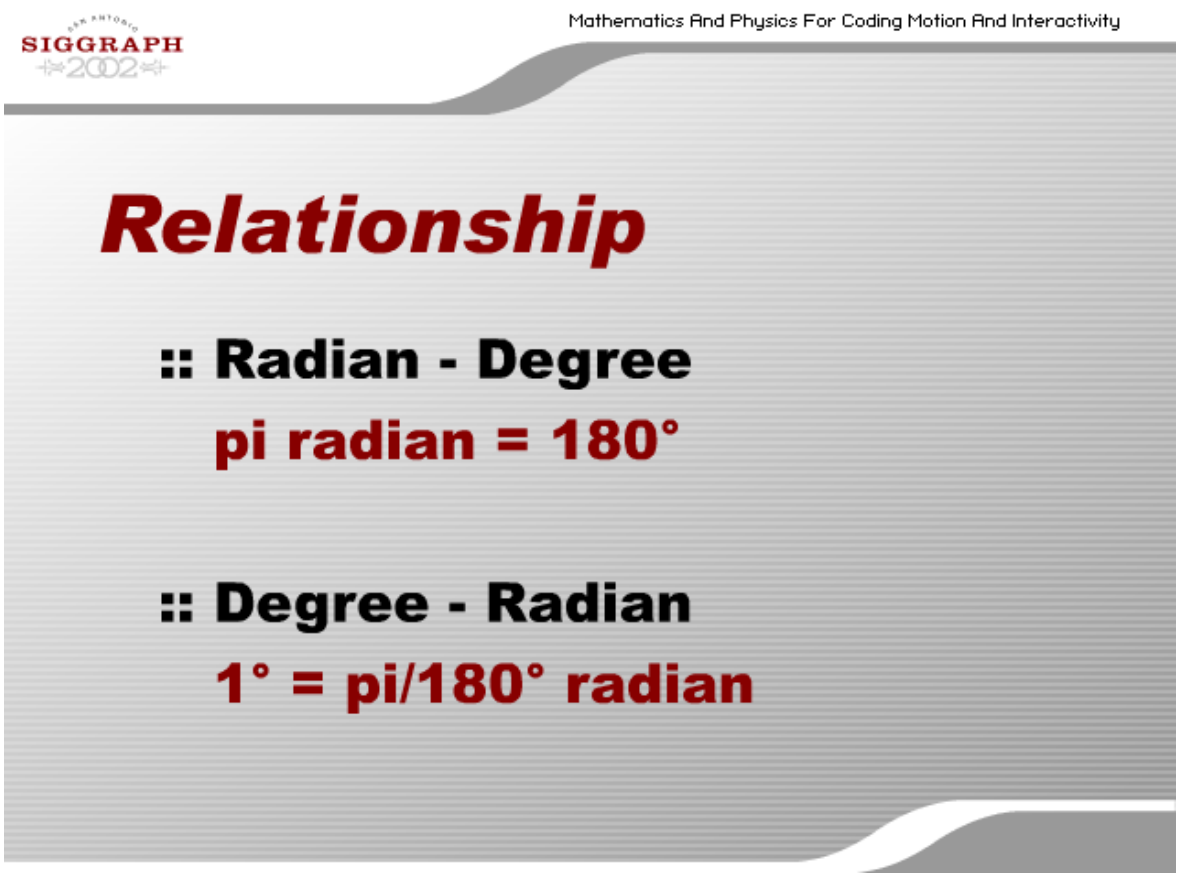

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
45 Degree Angle
45°

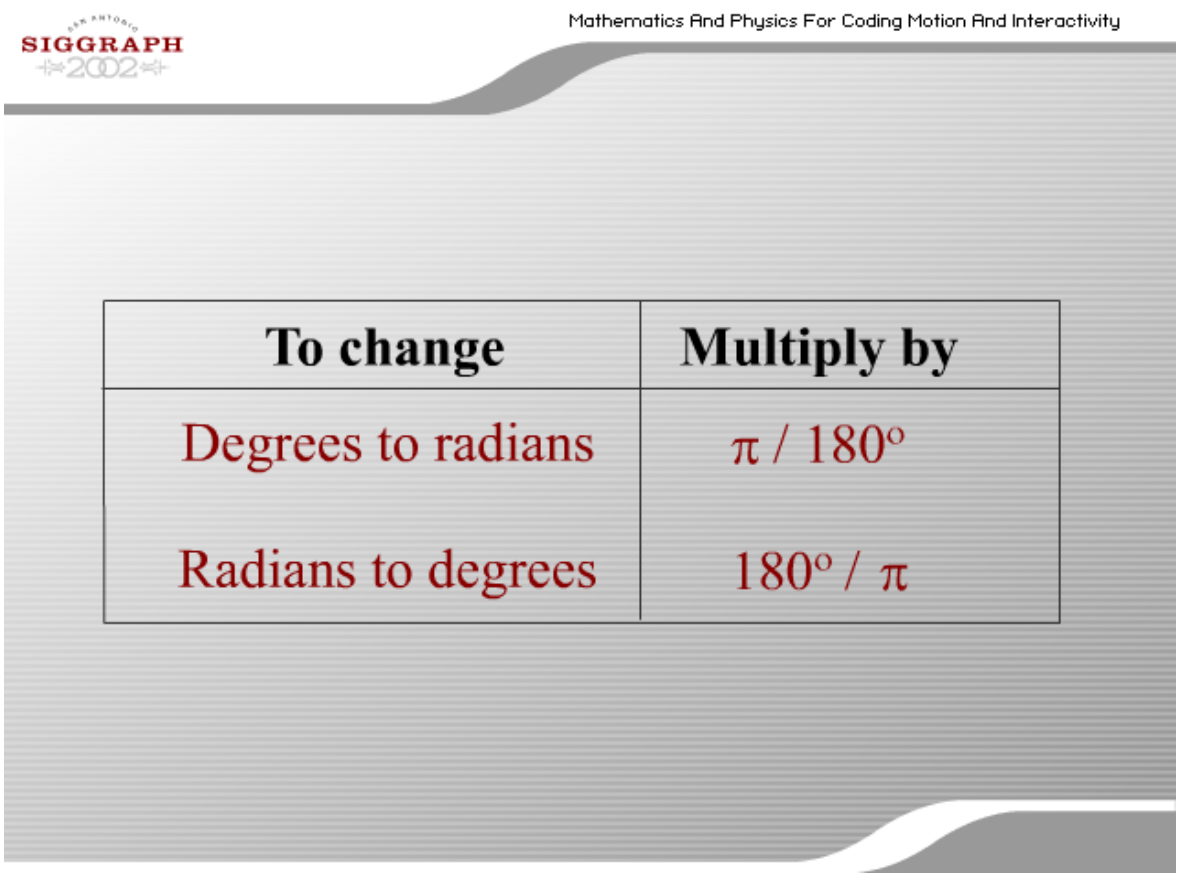

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Radian
:: Central angle of a circle subtended by an arc equal in length to the radius of the circle

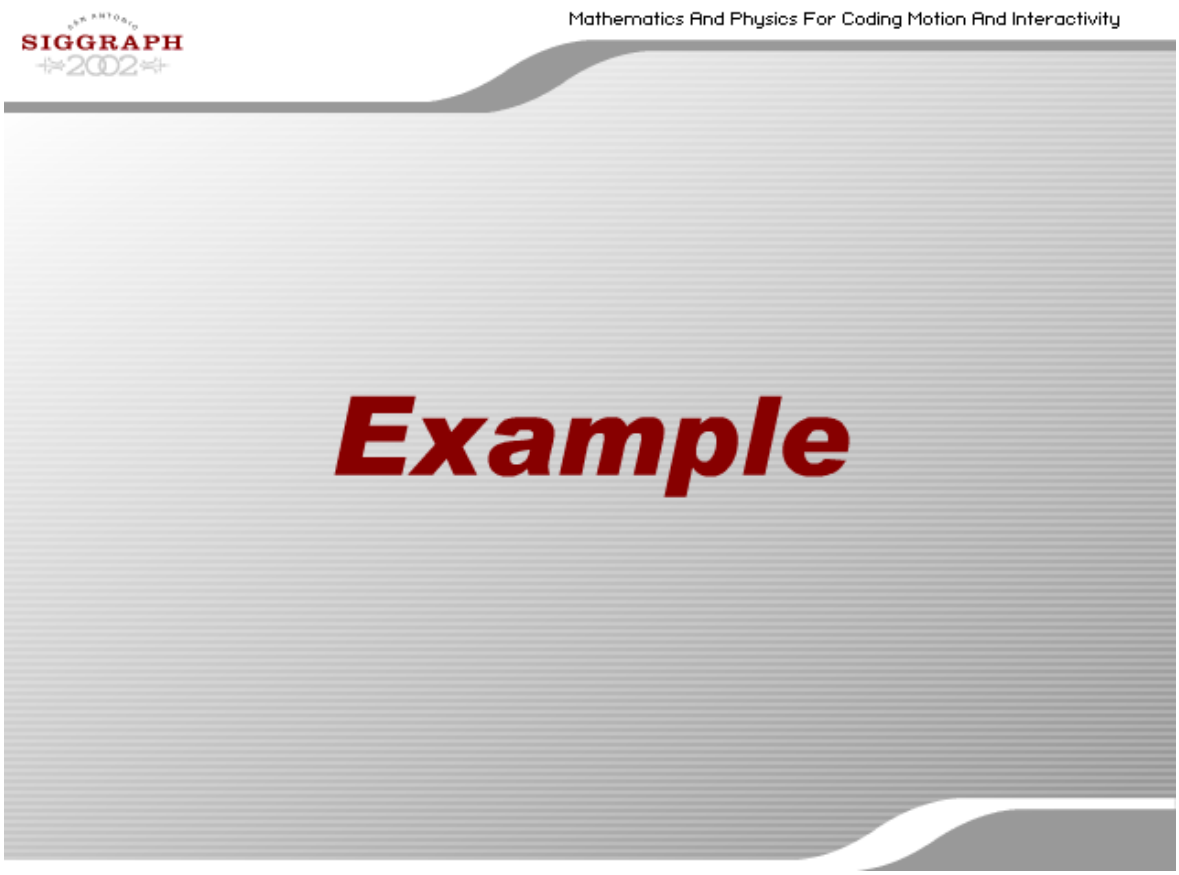

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Radian Measure
B
Ø
r
r
A
1 radian
C
r
B
Ø
r
r
A
2 radians

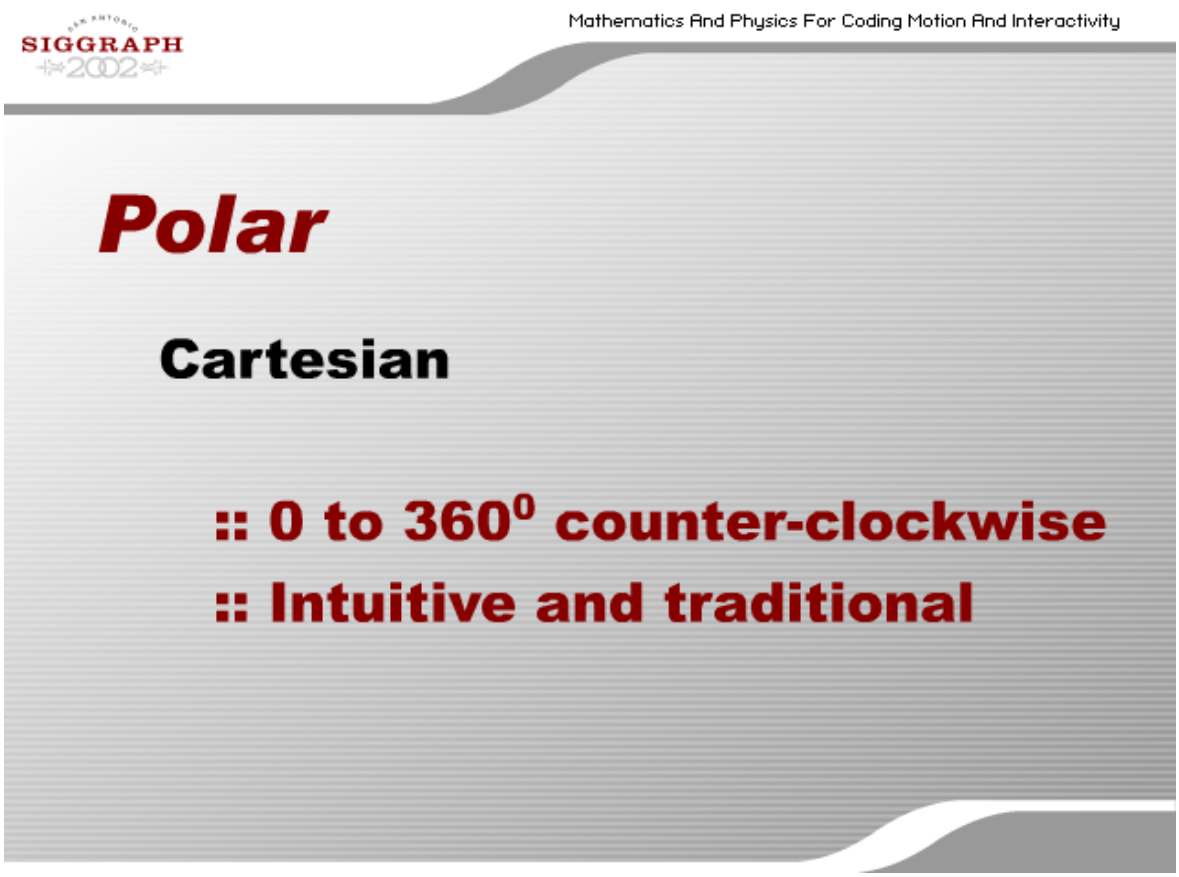

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Radian
:: Unit-free
:: Independent of radius of circle
:: Used for trigonometric calculations

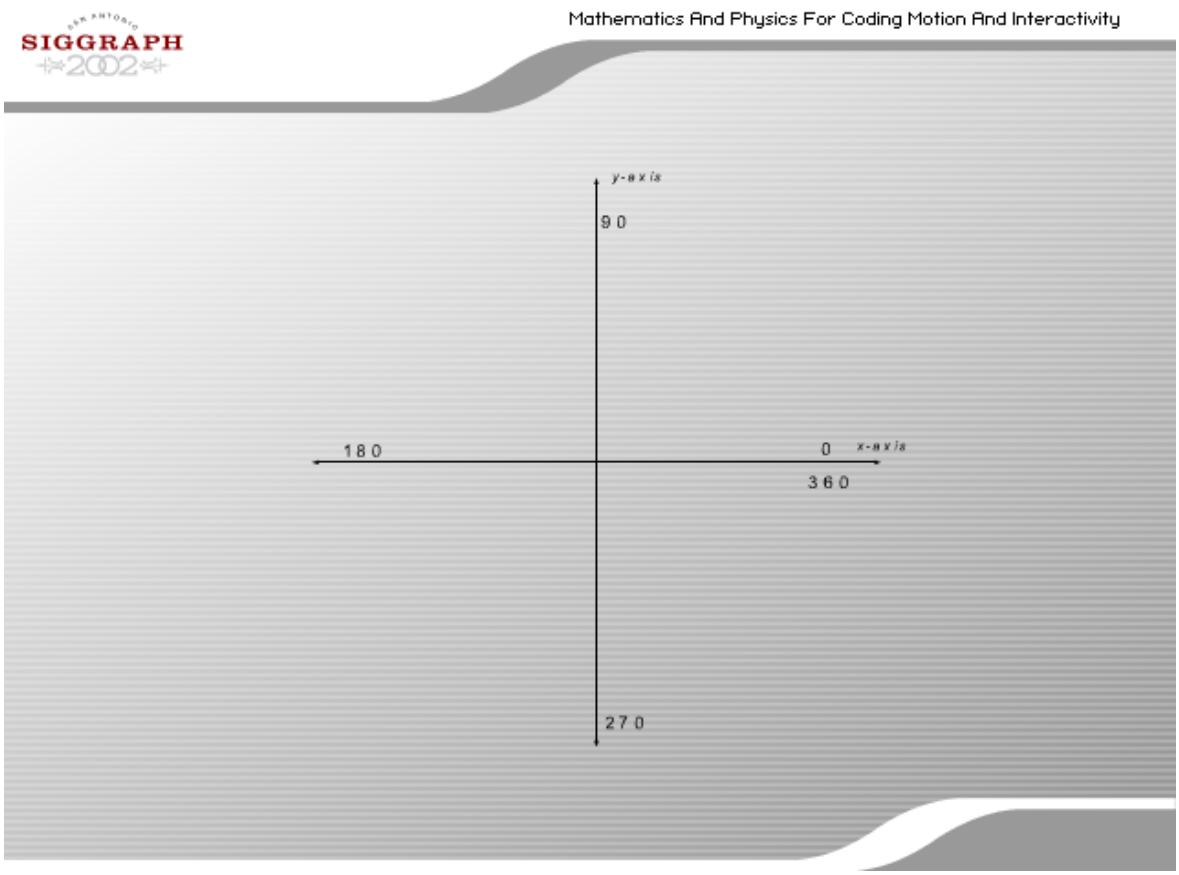

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

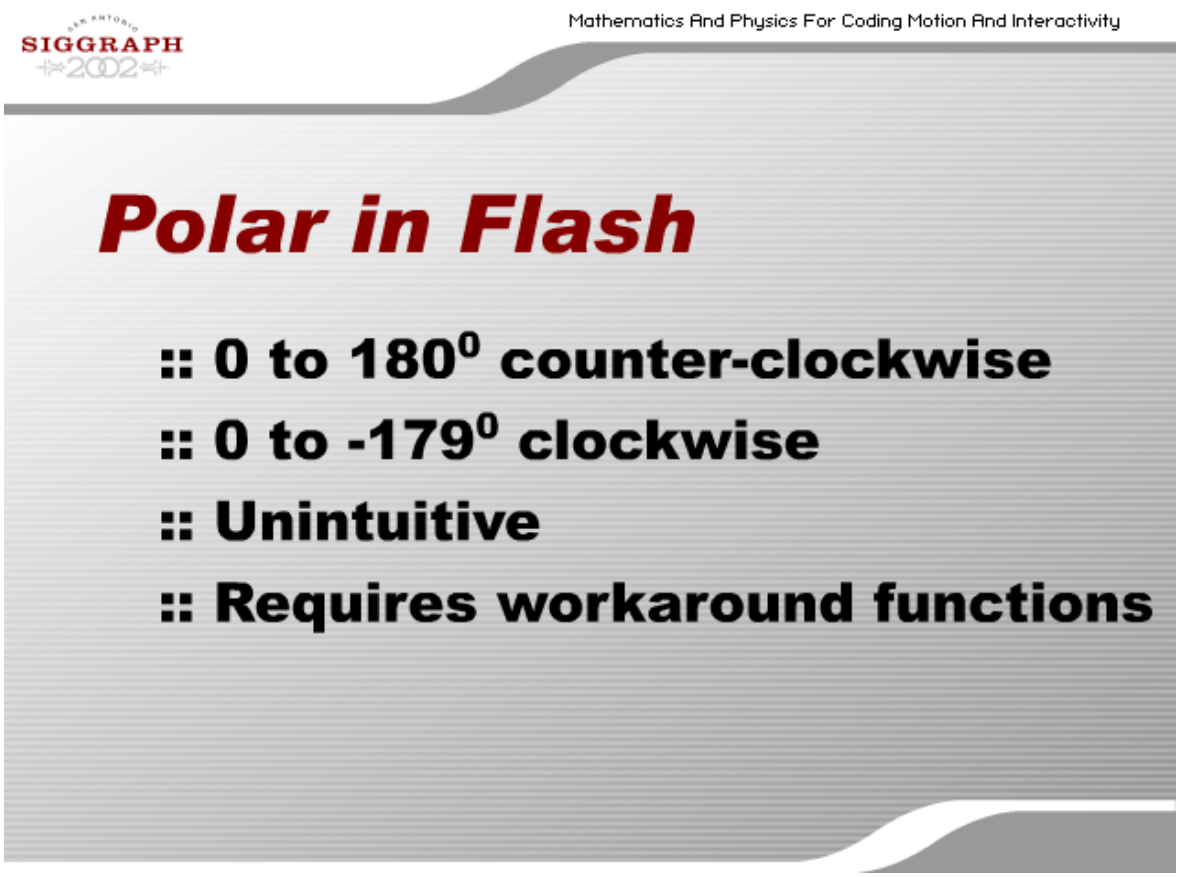

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Relationship
:: Radian - Degree
pi radian = 180°
:: Degree - Radian
1° = pi/180° radian

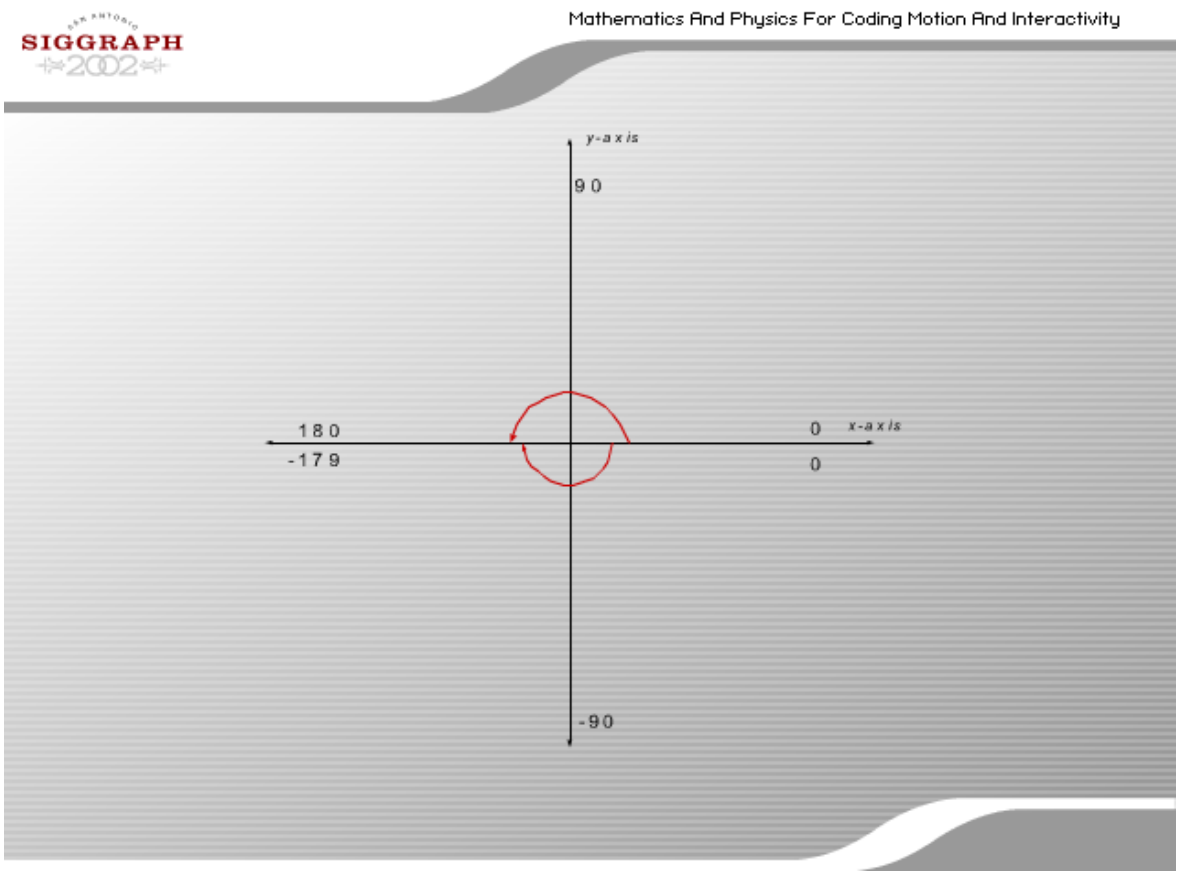

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
To change
Multiply by
Degrees to radians
π / 180°
Radians to degrees
180° / π

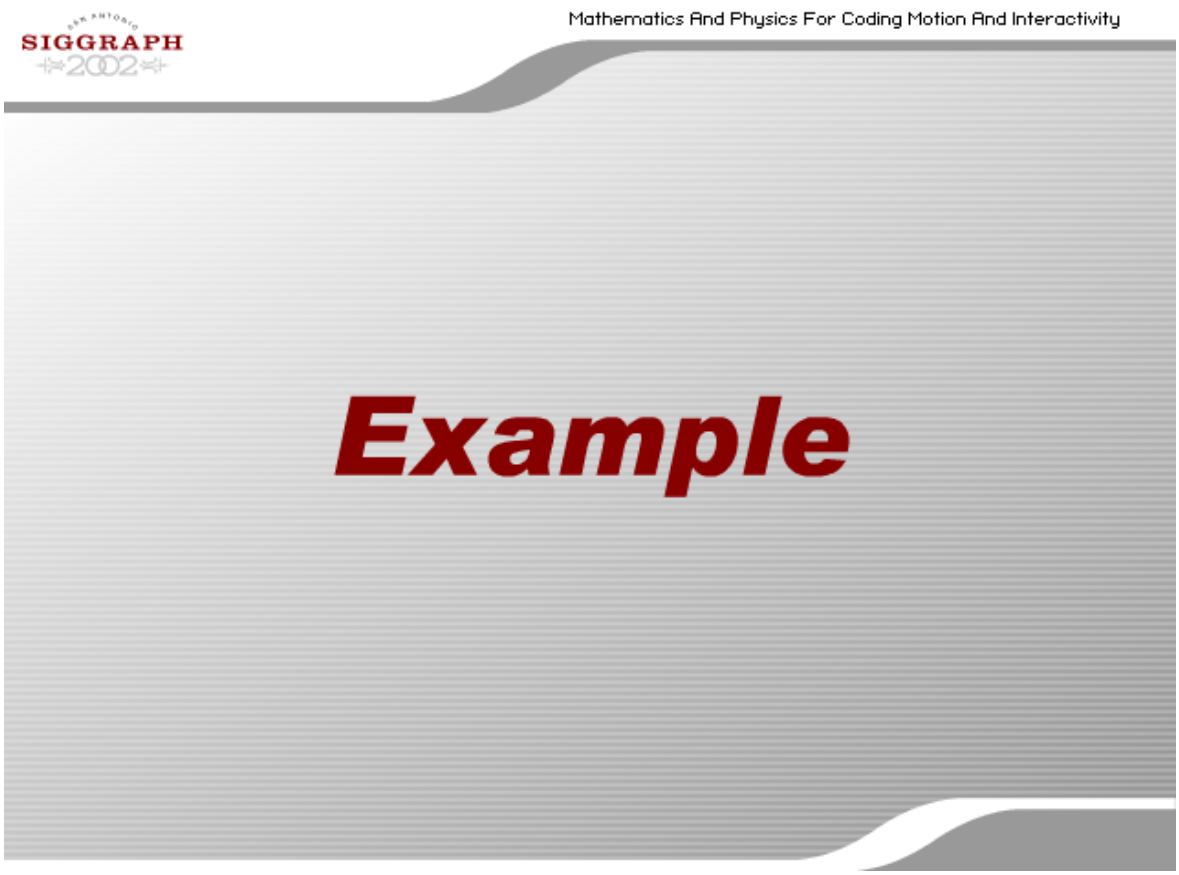

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

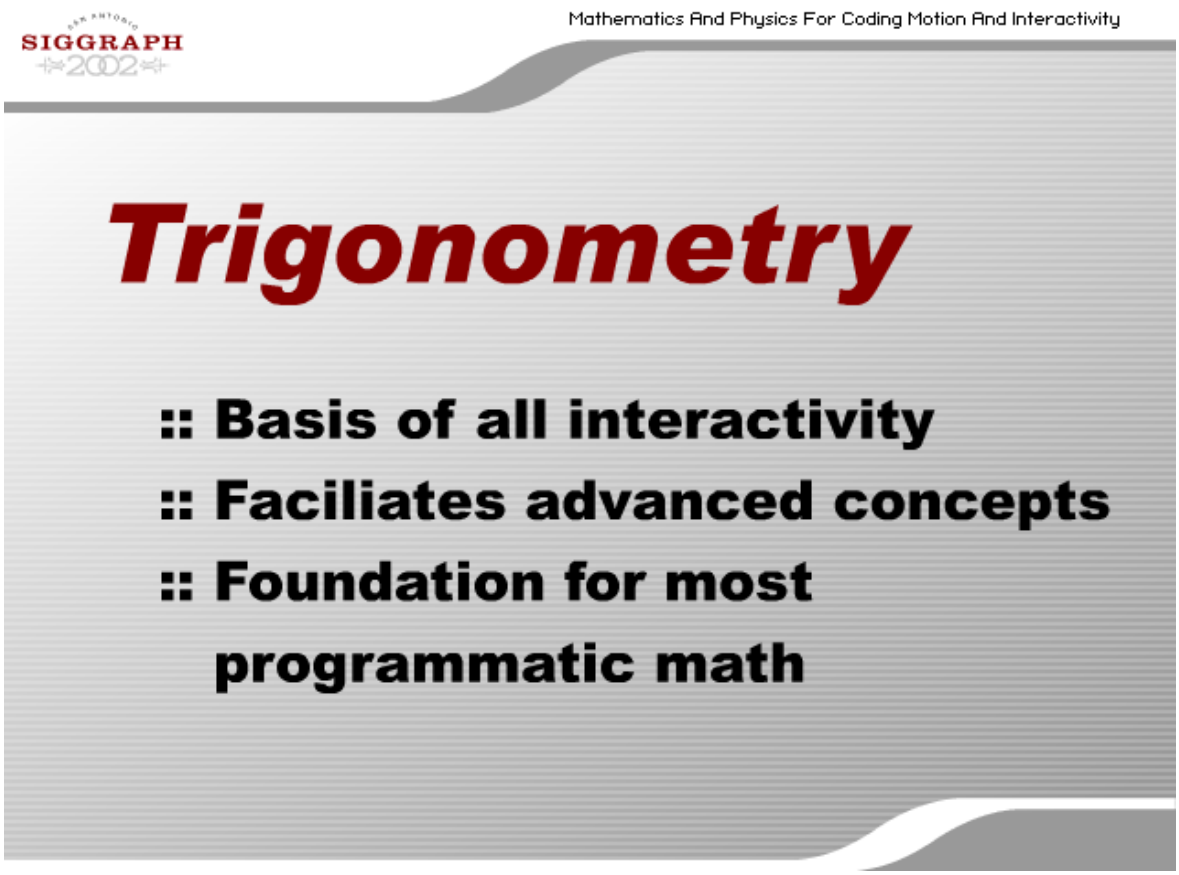

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Polar
Cartesian
:: 0 to 360⁰ counter-clockwise
:: Intuitive and traditional

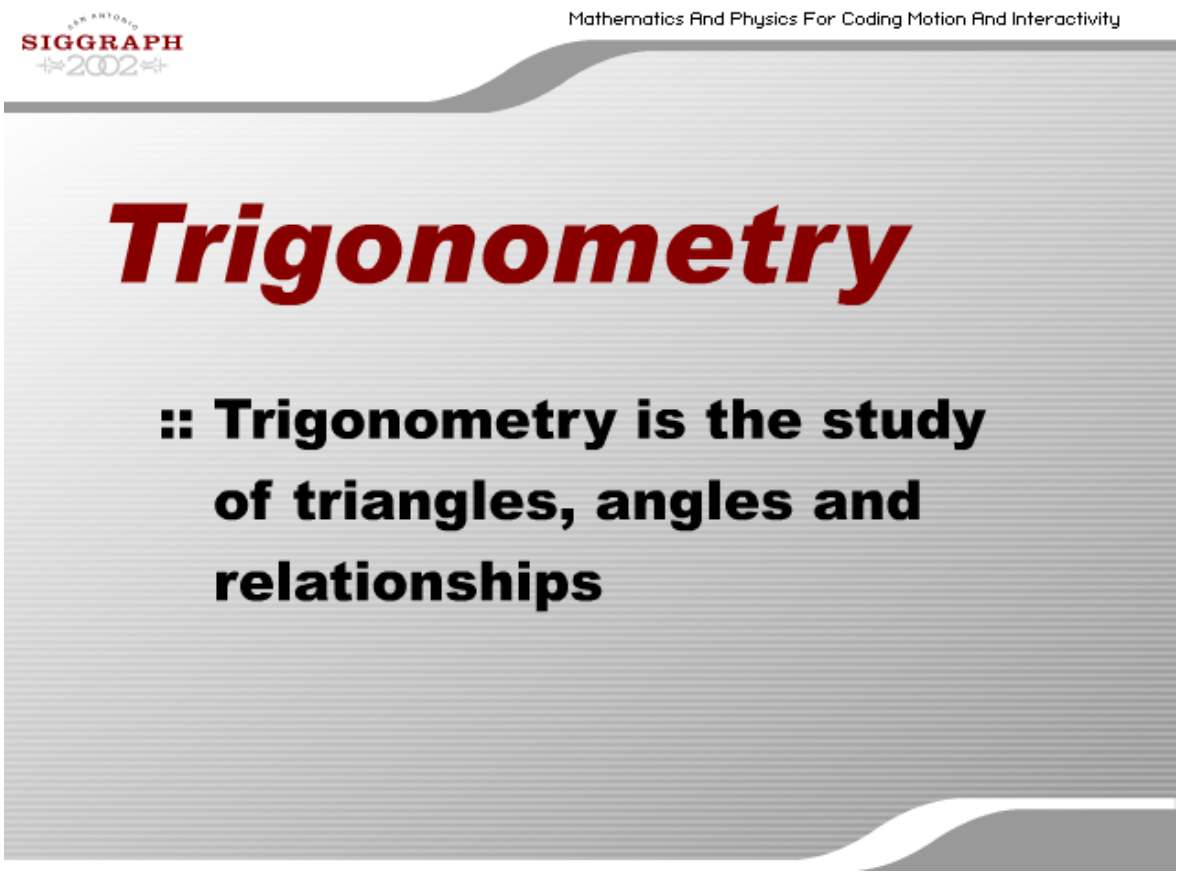

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
y-axis
90
180
0
x-axis
360
270

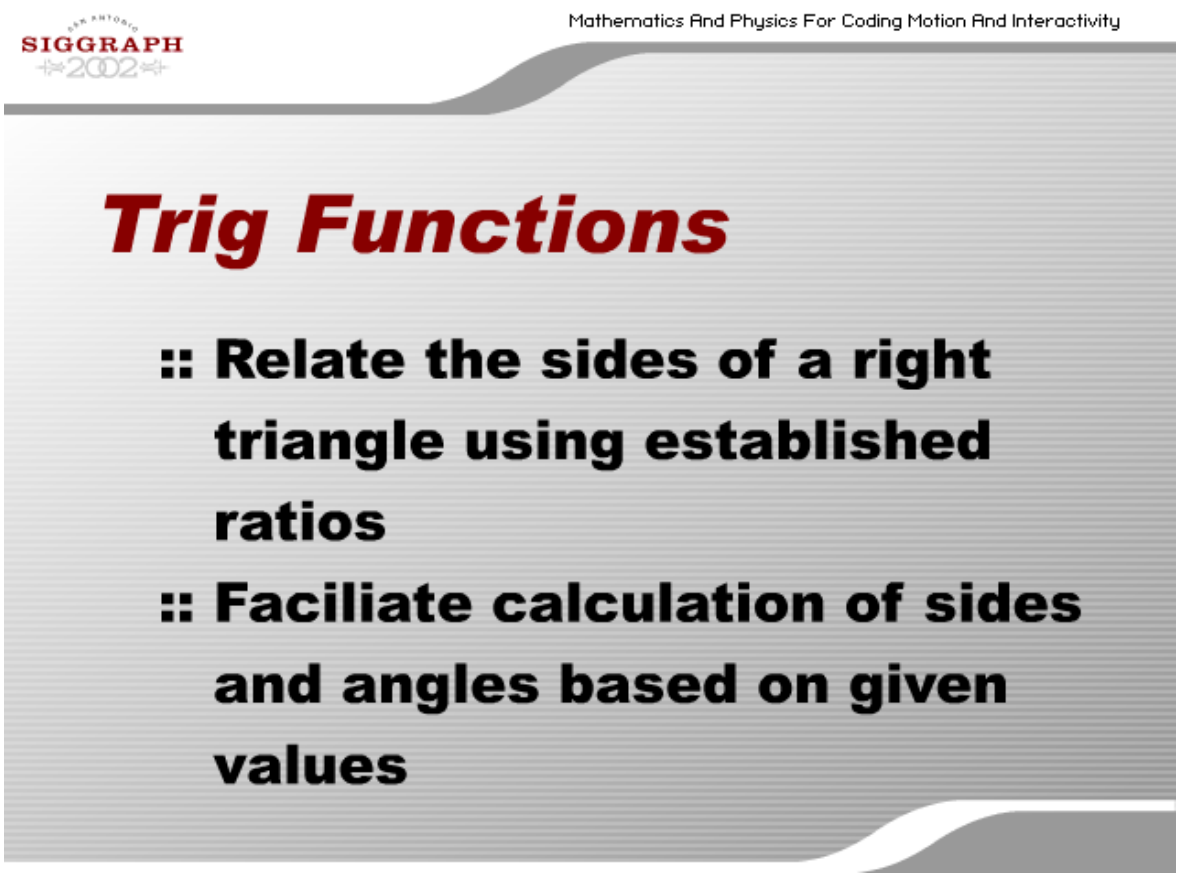

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Polar in Flash
:: 0 to 180⁰ counter-clockwise
:: 0 to -179⁰ clockwise
:: Unintuitive
:: Requires workaround functions


SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
y-axis
90
180
-179
0
0
x-axis
-90


SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Trigonometry
:: Basis of all interactivity
:: Faciliates advanced concepts
:: Foundation for most programmatic math


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Trigonometry
:: Trigonometry is the study of triangles, angles and relationships


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Trig Functions
:: Relate the sides of a right triangle using established ratios
:: Faciliate calculation of sides and angles based on given values

Trigonometric functions

$$\sin\theta = \frac{\text{opp}}{\text{hyp}} = \frac{b}{c}$$

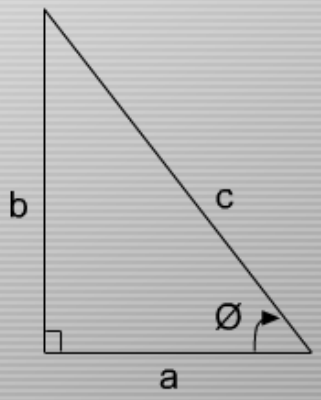
$$\cos\theta = \frac{\text{adj}}{\text{hyp}} = \frac{a}{c}$$

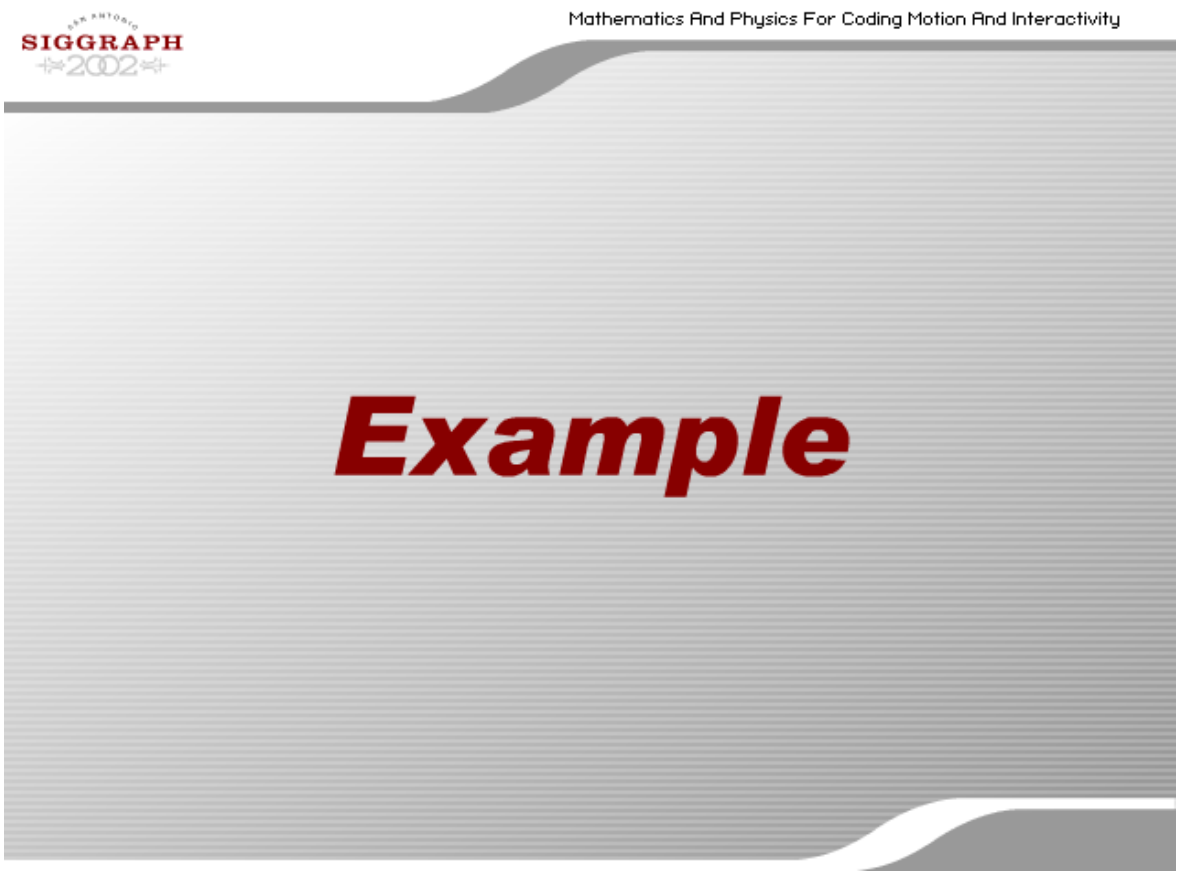
$$\tan\theta = \frac{\text{opp}}{\text{adj}} = \frac{b}{a}$$

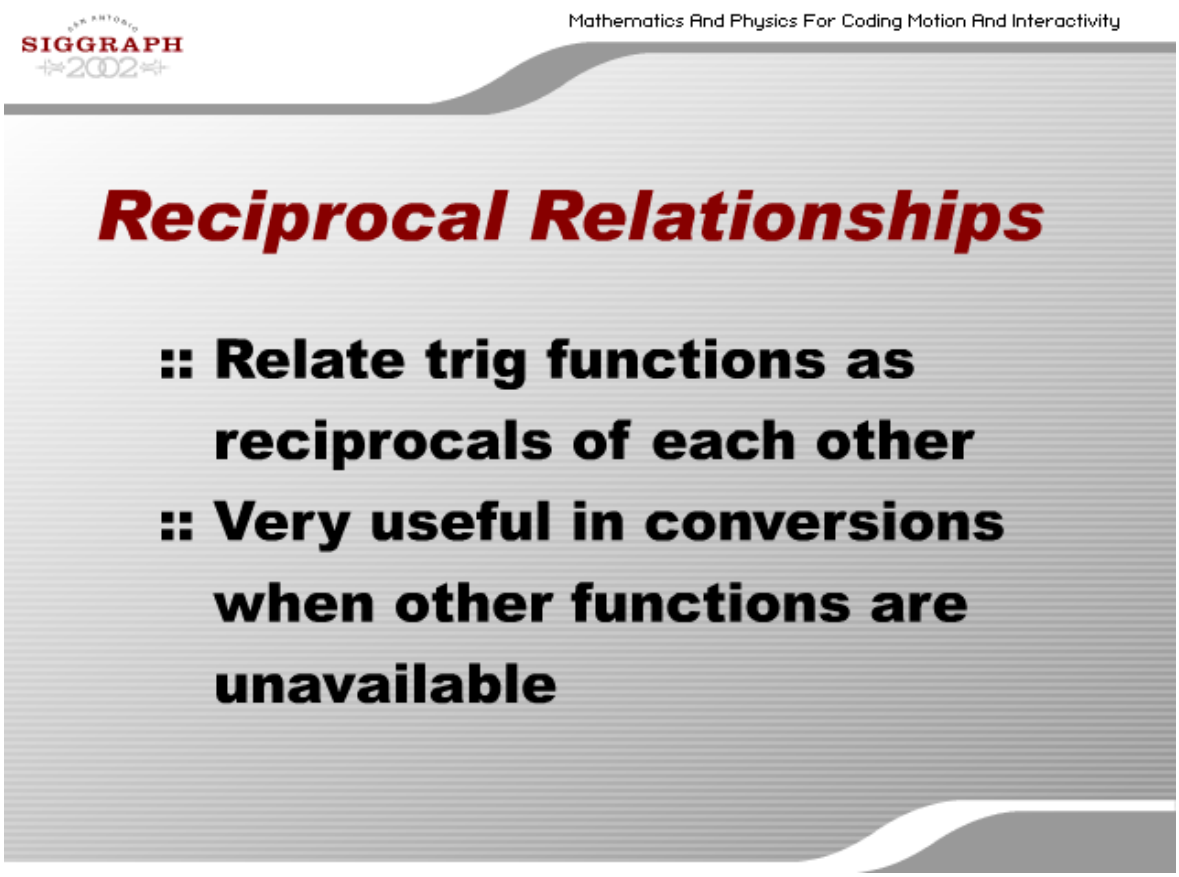
$$\cot\theta = \frac{\text{adj}}{\text{opp}} = \frac{a}{b}$$

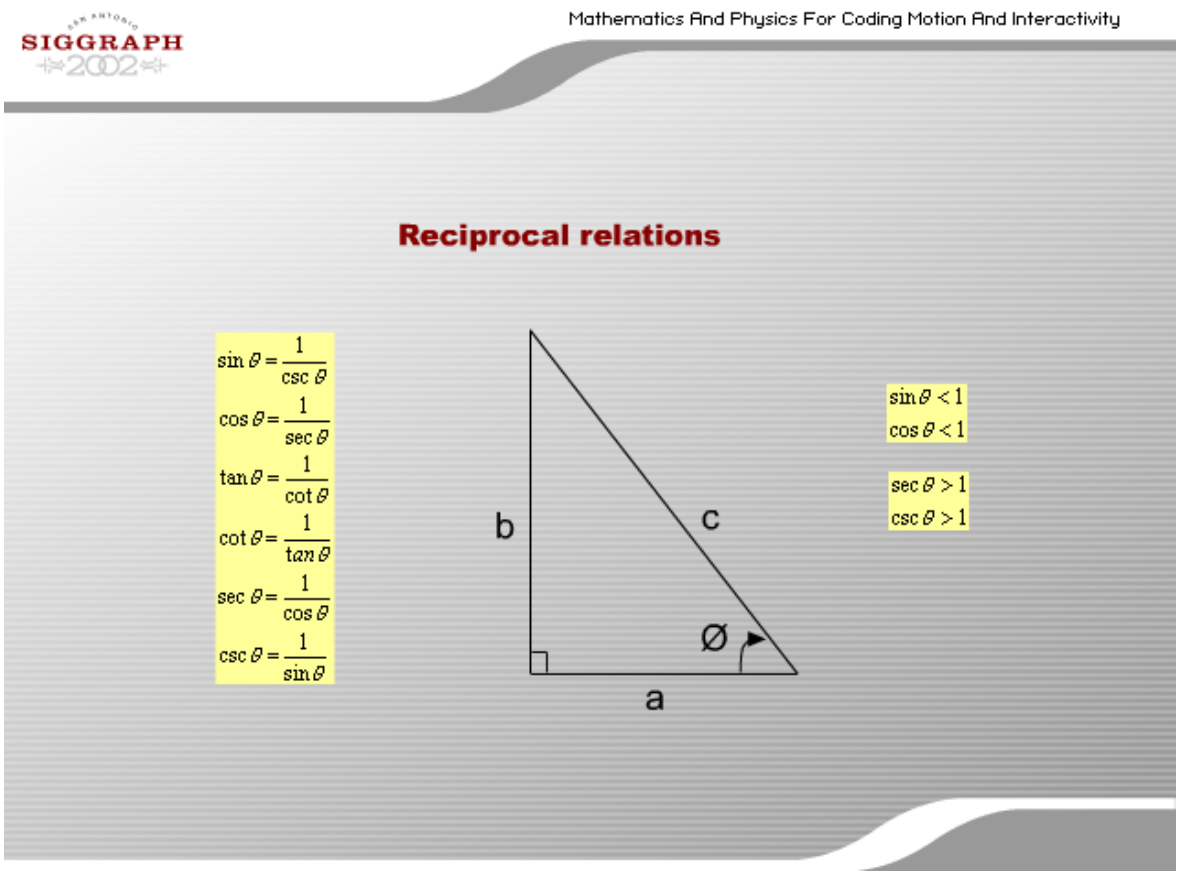
$$\sec\theta = \frac{\text{hyp}}{\text{adj}} = \frac{c}{a}$$

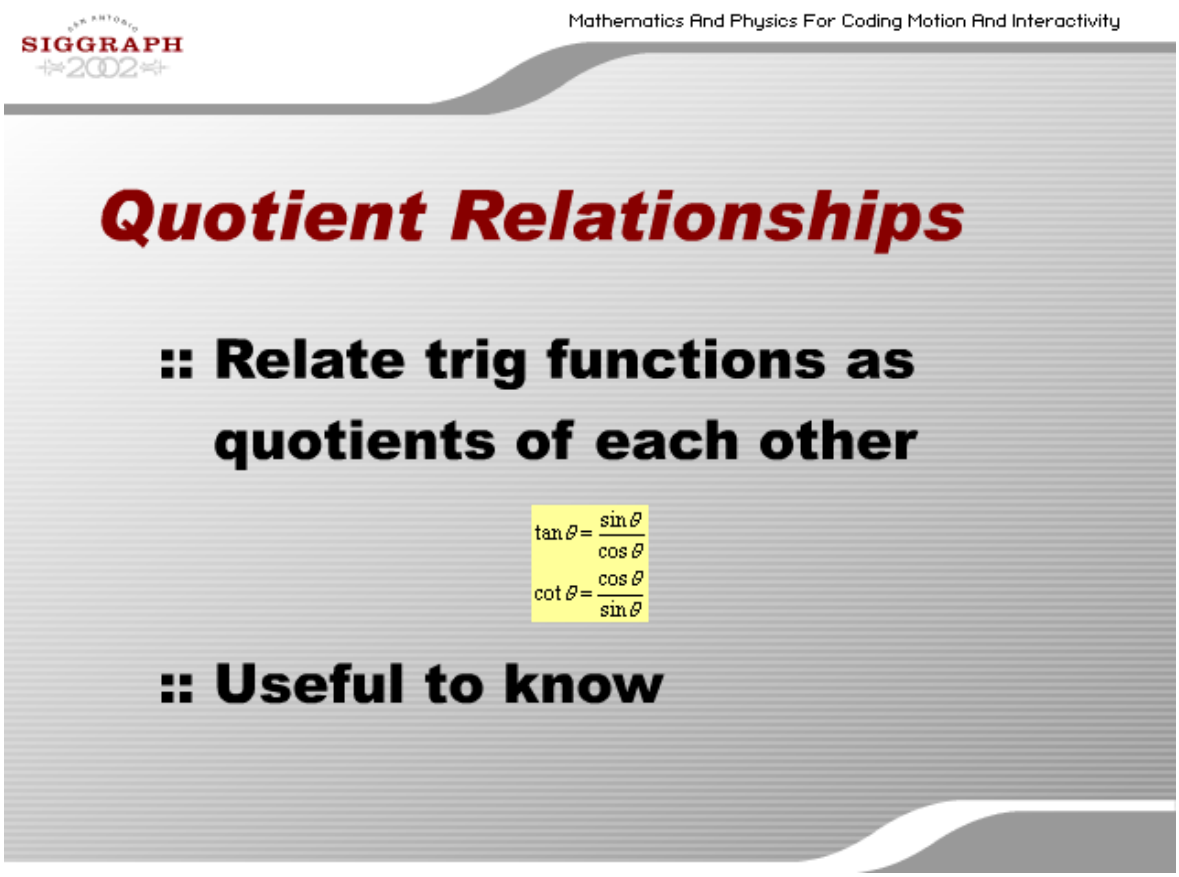
$$\csc\theta = \frac{\text{hyp}}{\text{opp}} = \frac{c}{b}$$

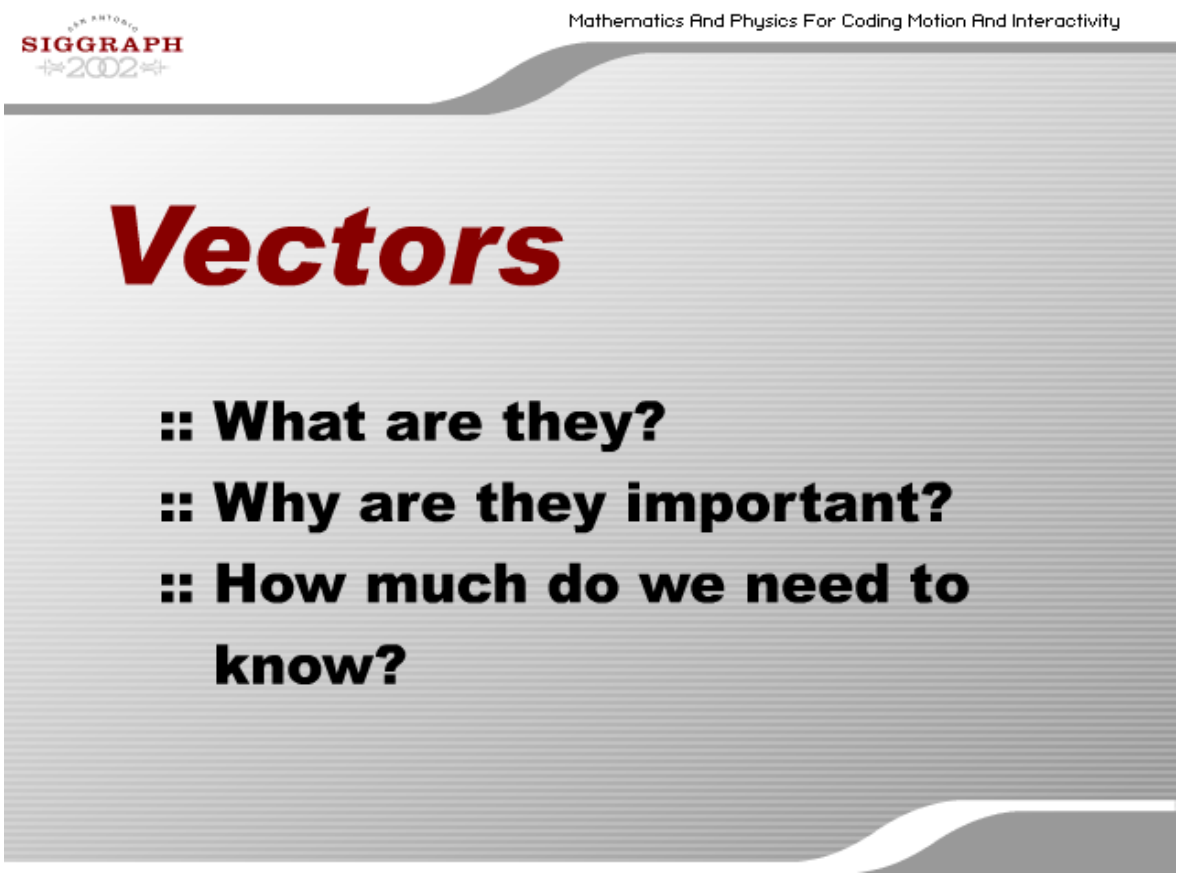


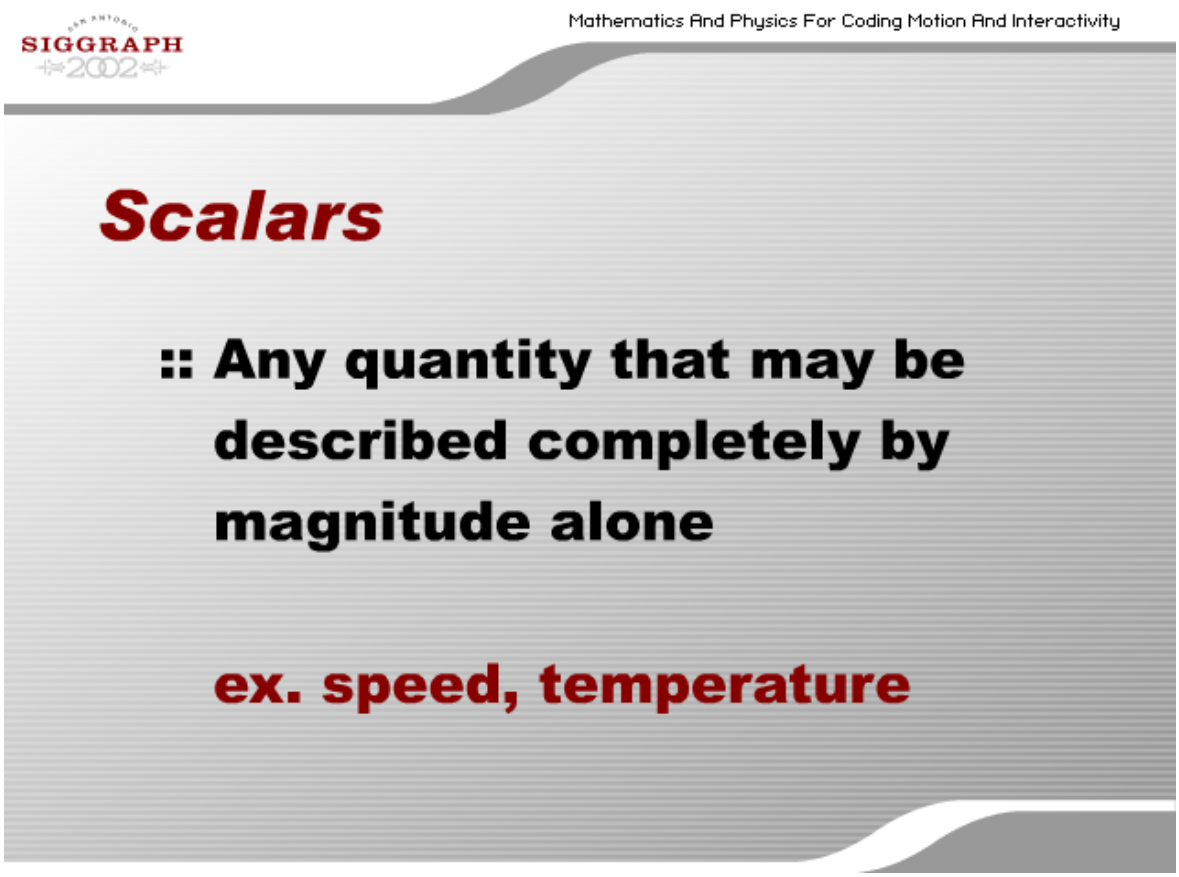

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Reciprocal Relationships
:: Relate trig functions as reciprocals of each other
:: Very useful in conversions when other functions are unavailable

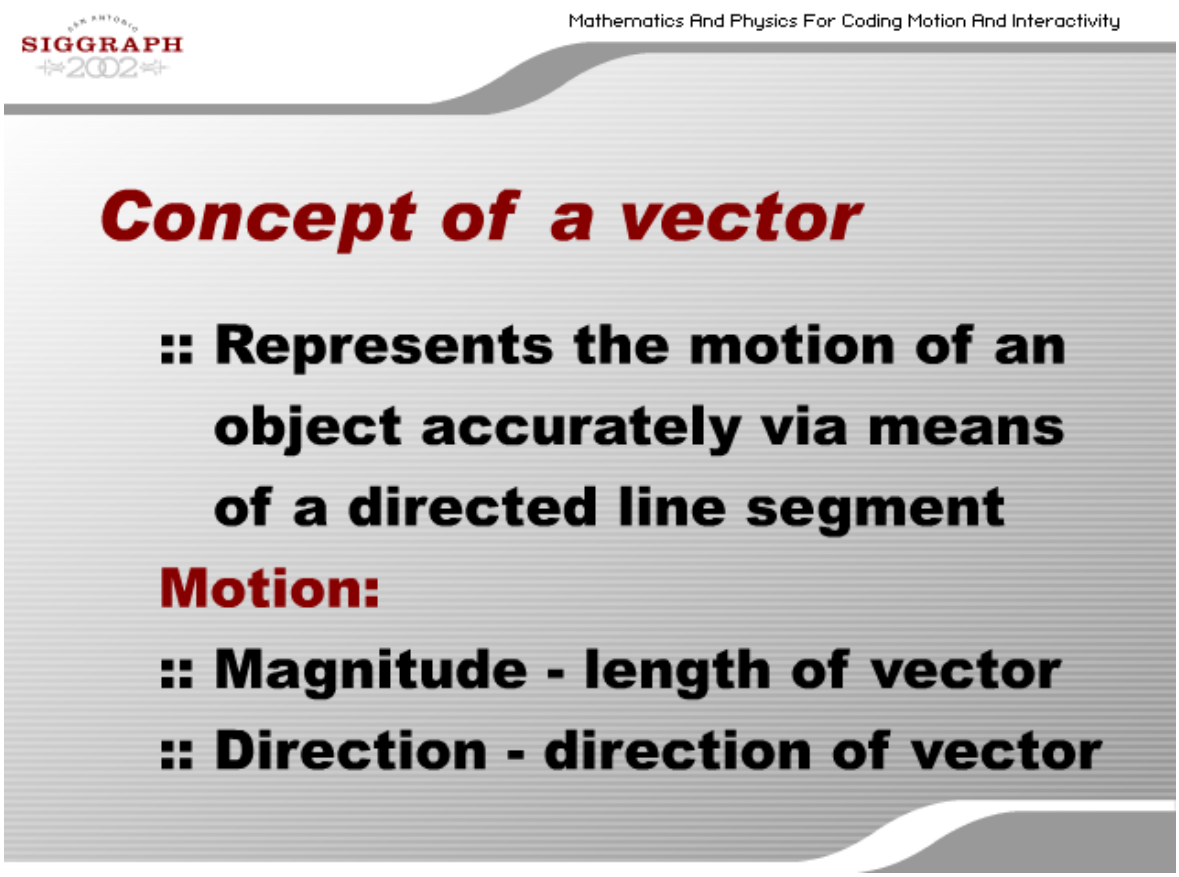

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Reciprocal relations
$\sin\theta = \frac{1}{\csc\theta}$
$\cos\theta = \frac{1}{\sec\theta}$
$\tan\theta = \frac{1}{\cot\theta}$
$\cot\theta = \frac{1}{\tan\theta}$
$\sec\theta = \frac{1}{\cos\theta}$
$\csc\theta = \frac{1}{\sin\theta}$
b
c
Ø
a
$\sin\theta < 1$
$\cos\theta < 1$
$\sec\theta > 1$
$\csc\theta > 1$

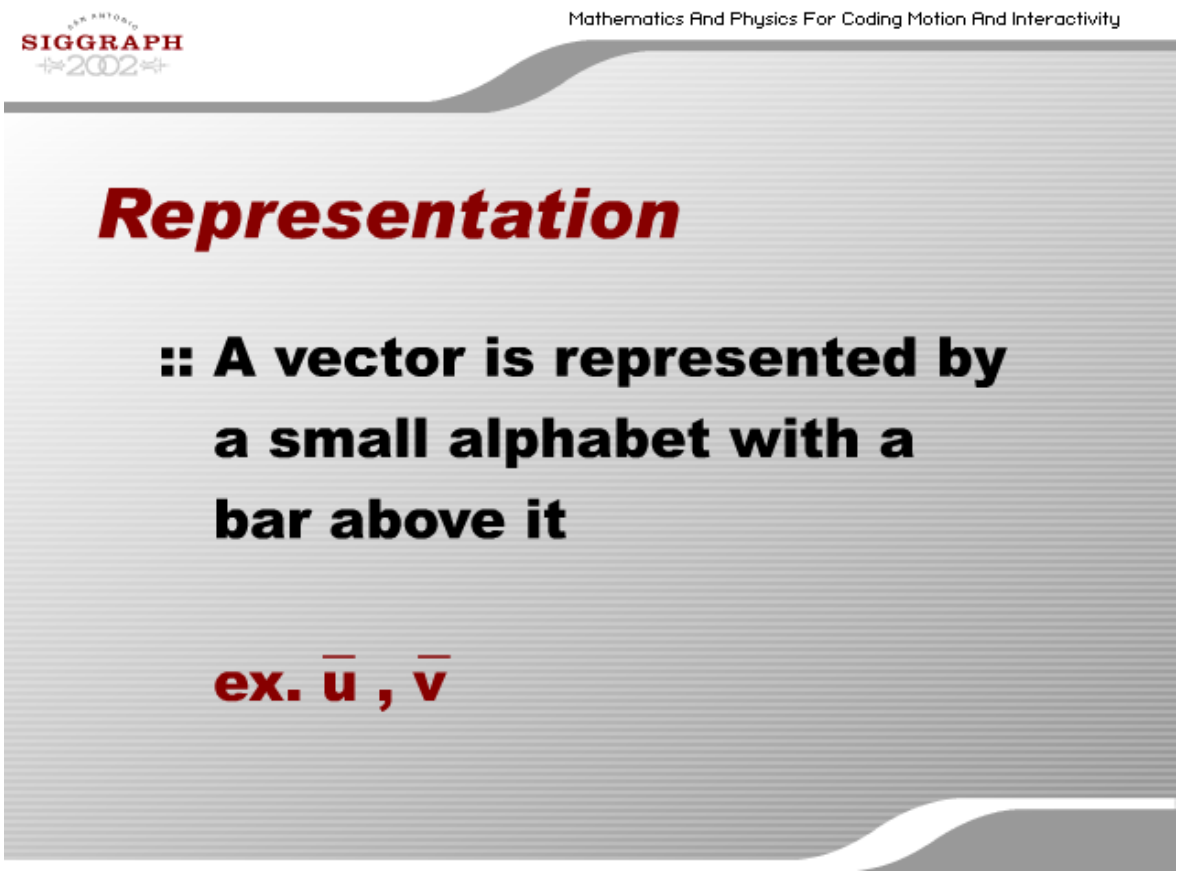

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Quotient Relationships
:: Relate trig functions as quotients of each other
$\tan\theta = \frac{\sin\theta}{\cos\theta}$
$\cot\theta = \frac{\cos\theta}{\sin\theta}$
:: Useful to know

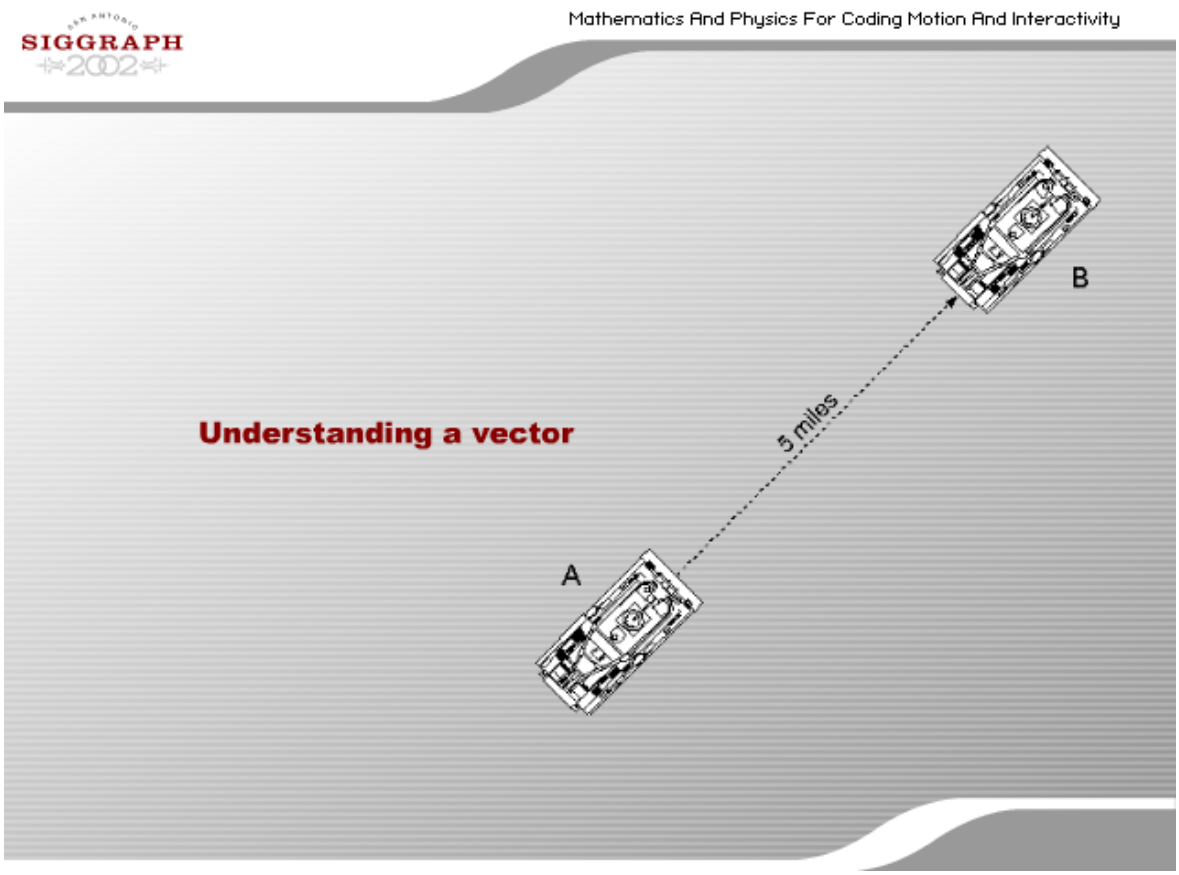

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Vectors
:: What are they?
:: Why are they important?
:: How much do we need to know?

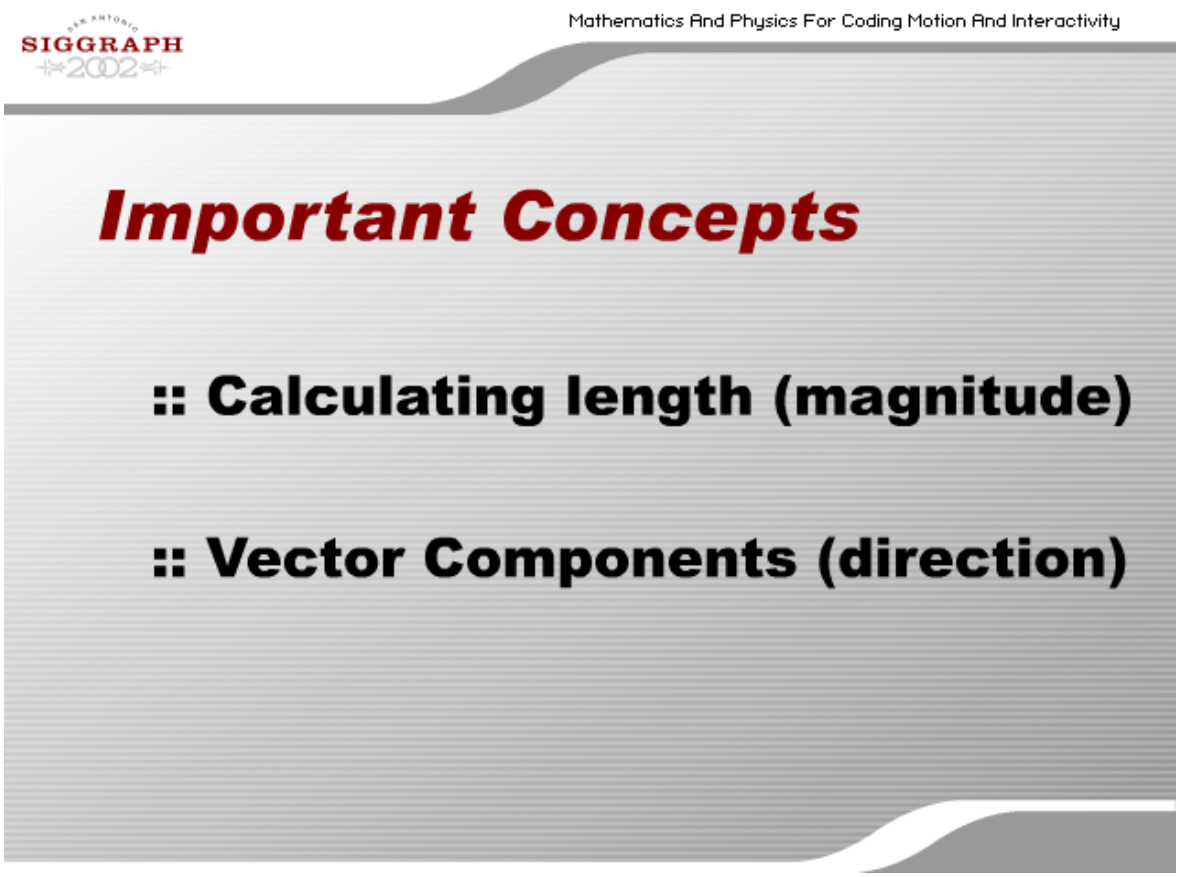

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Scalars
:: Any quantity that may be described completely by magnitude alone
ex. speed, temperature

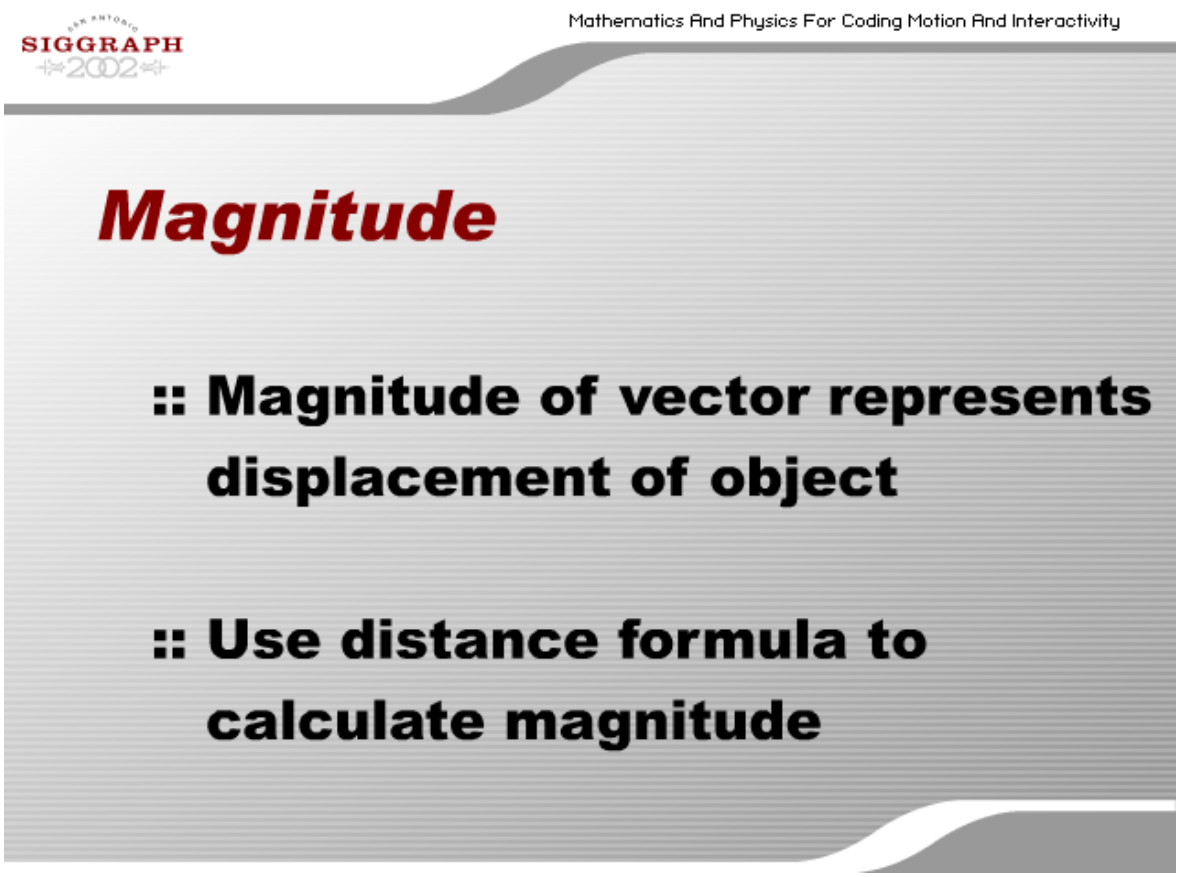

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Vectors
:: Any quantity that needs magnitude and direction to describe it.
ex. velocity, displacement

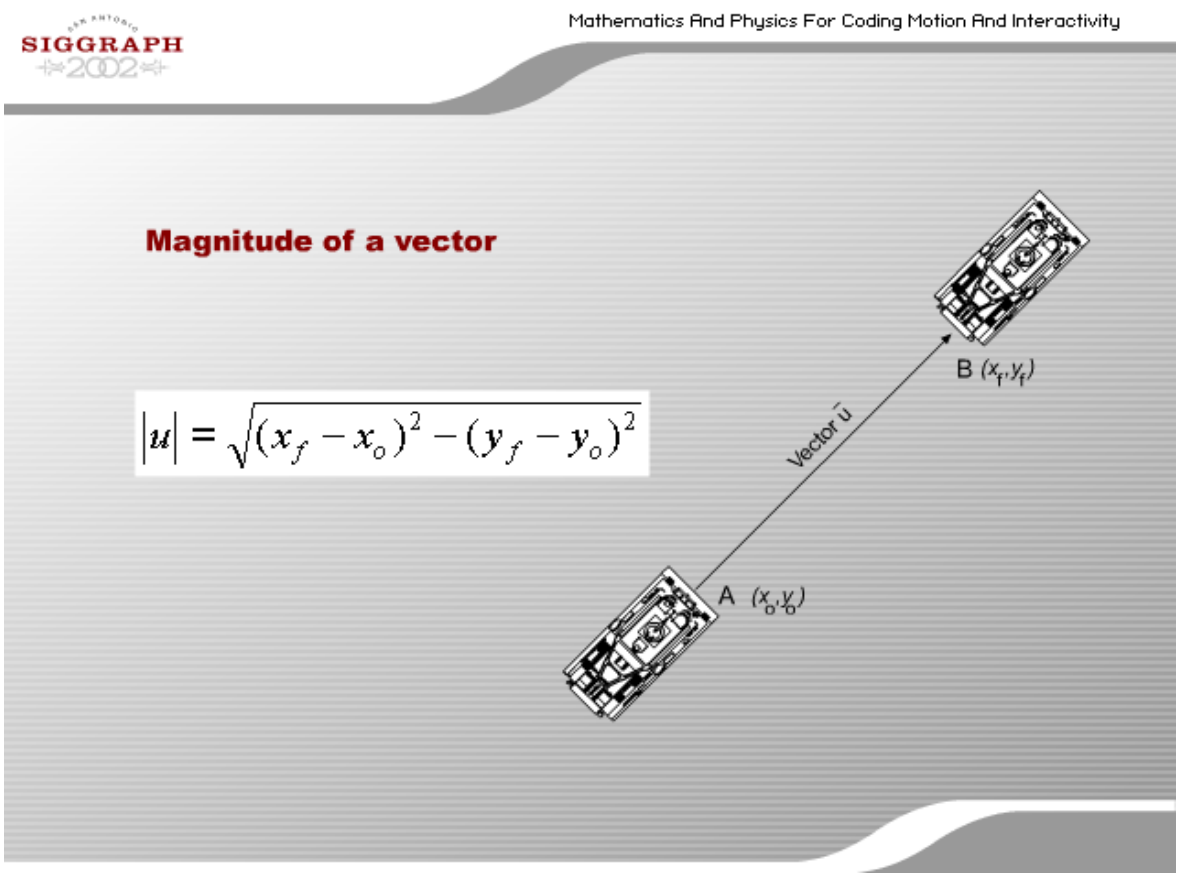

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Concept of a vector
:: Represents the motion of an object accurately via means of a directed line segment
Motion:
:: Magnitude - length of vector
:: Direction - direction of vector

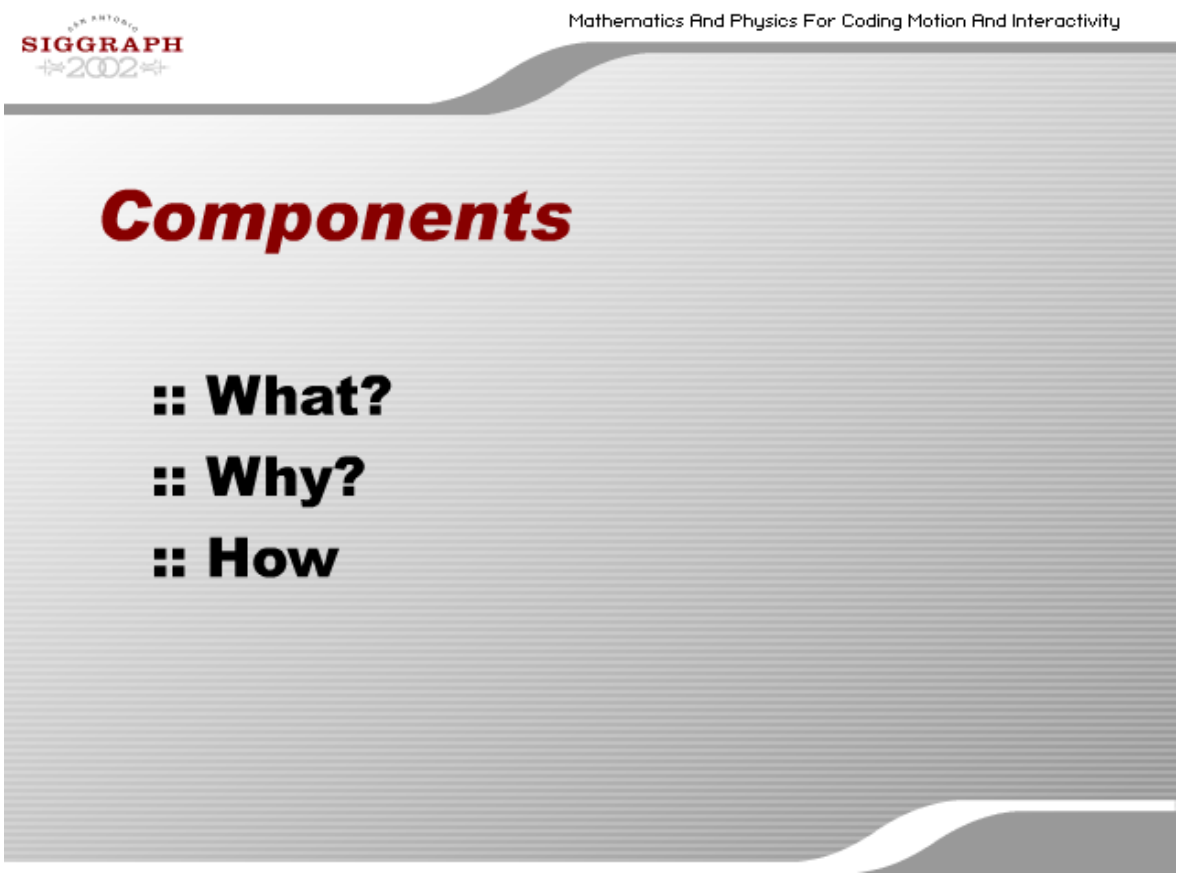

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Representation
:: A vector is represented by a small alphabet with a bar above it
ex. $\bar{u}$, $\bar{v}$

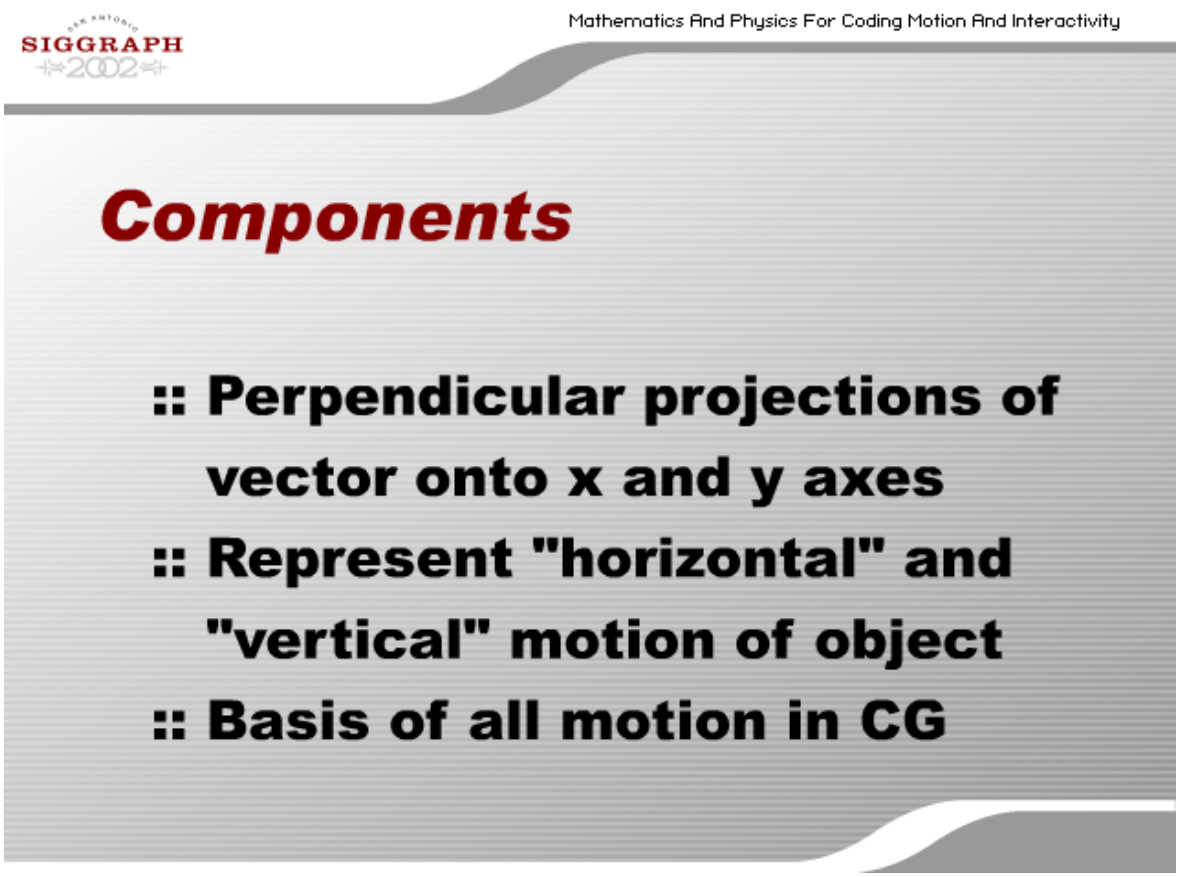

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Understanding a vector
5 miles
A
B

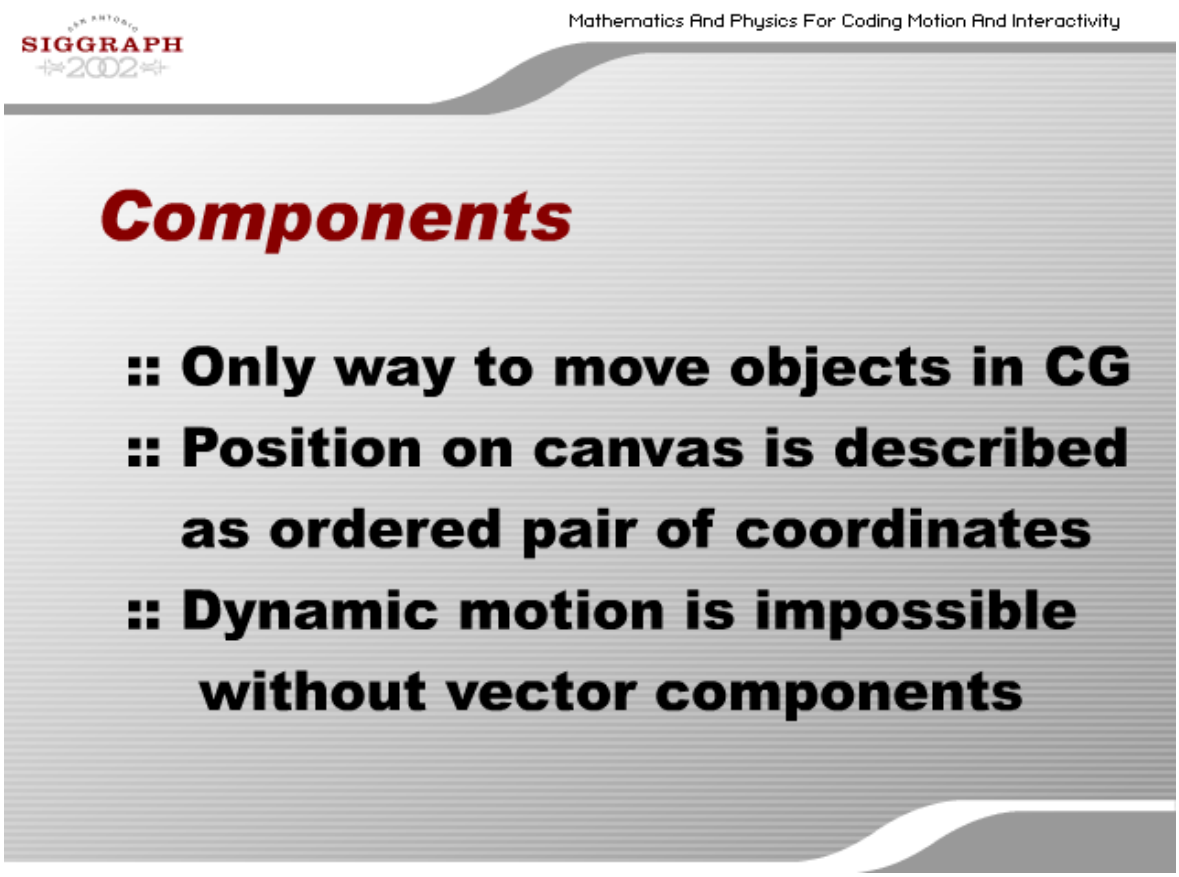

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Important Concepts
:: Calculating length (magnitude)
:: Vector Components (direction)

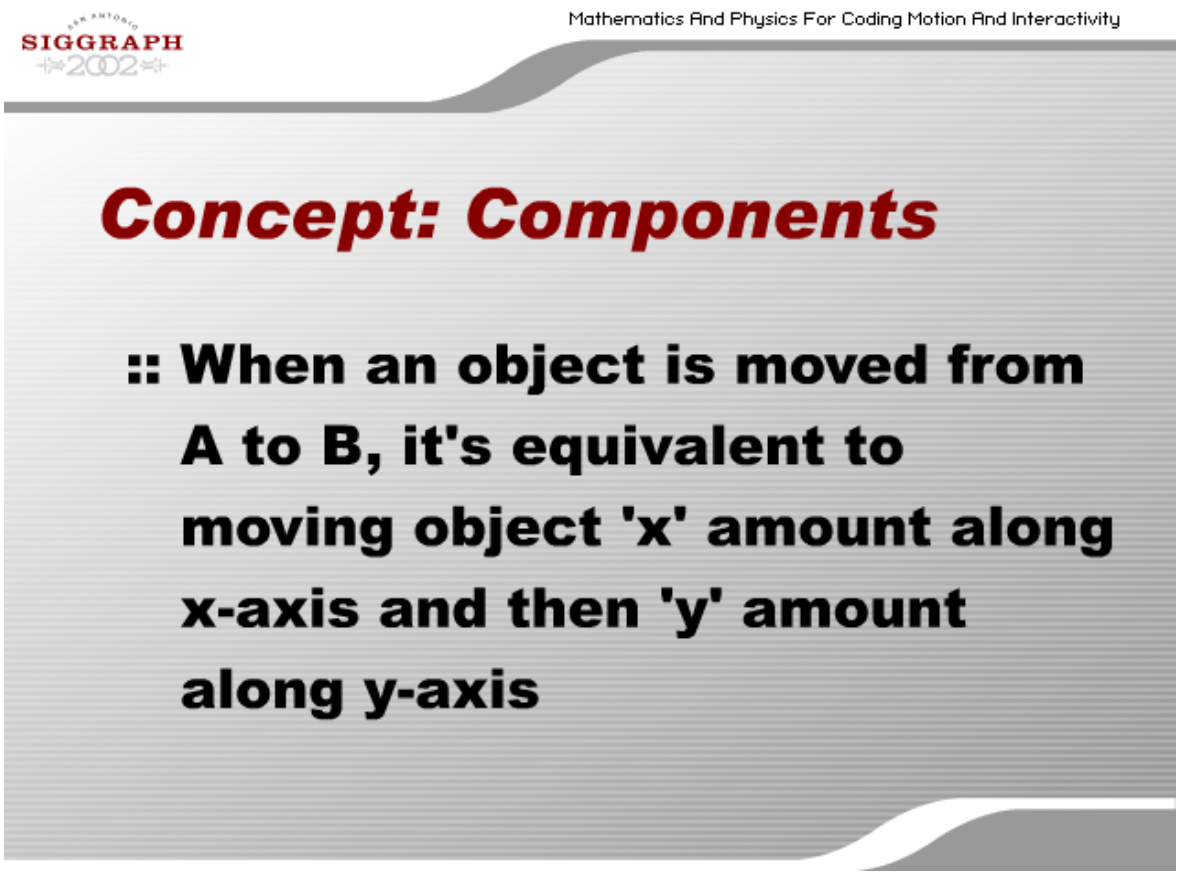

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Magnitude
:: Magnitude of vector represents displacement of object
:: Use distance formula to calculate magnitude

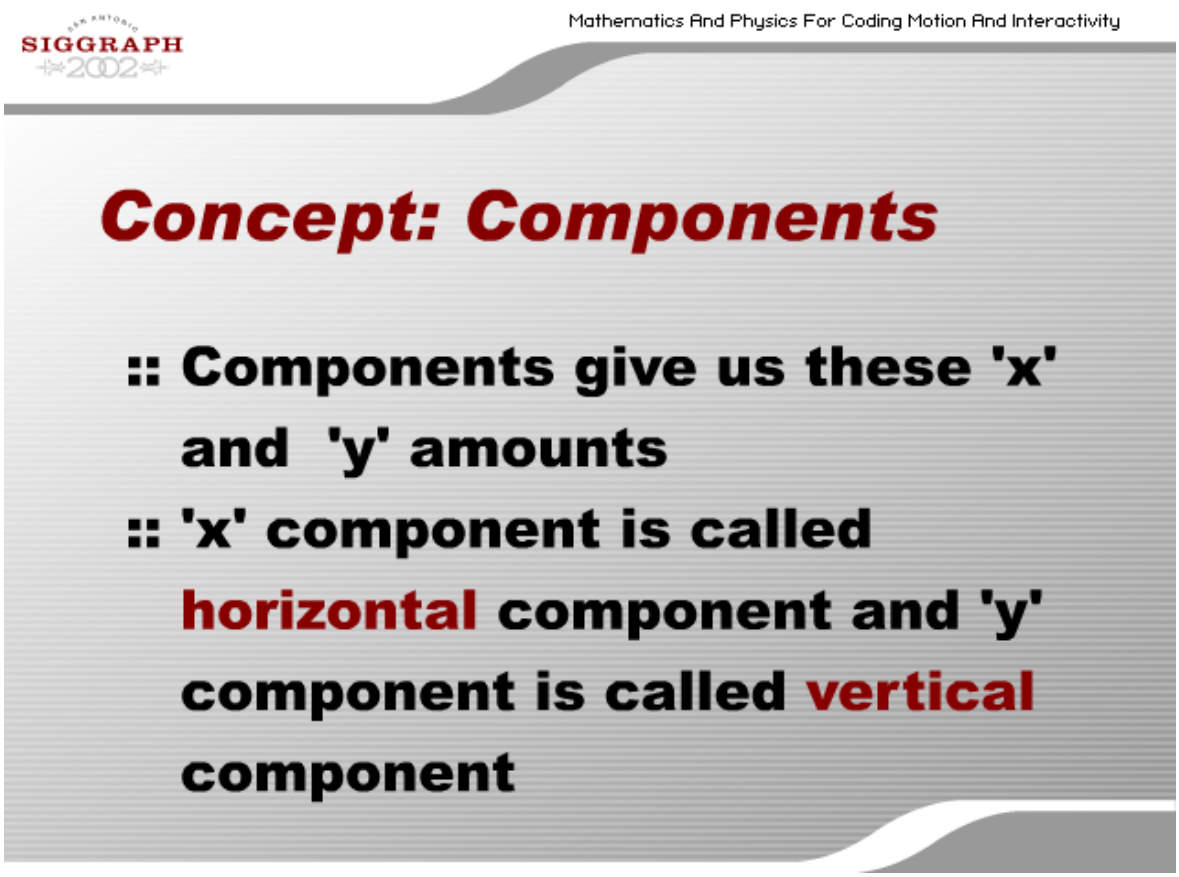

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Magnitude of a vector
$|u| = \sqrt{(x_f - x_o)^2 - (y_f - y_o)^2}$
Vector ū
B (x_f, y_f)
A (x_o, y_o)

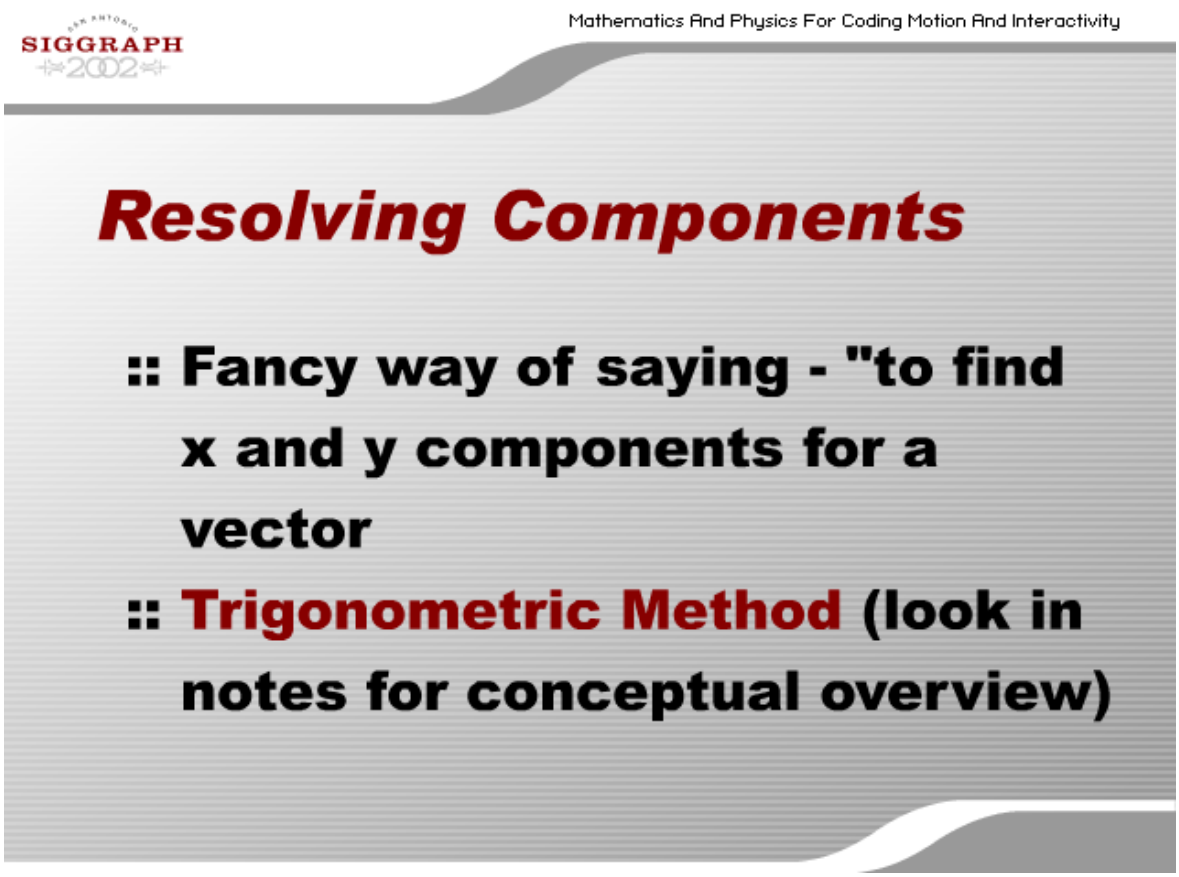

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Components
:: What?
:: Why?
:: How

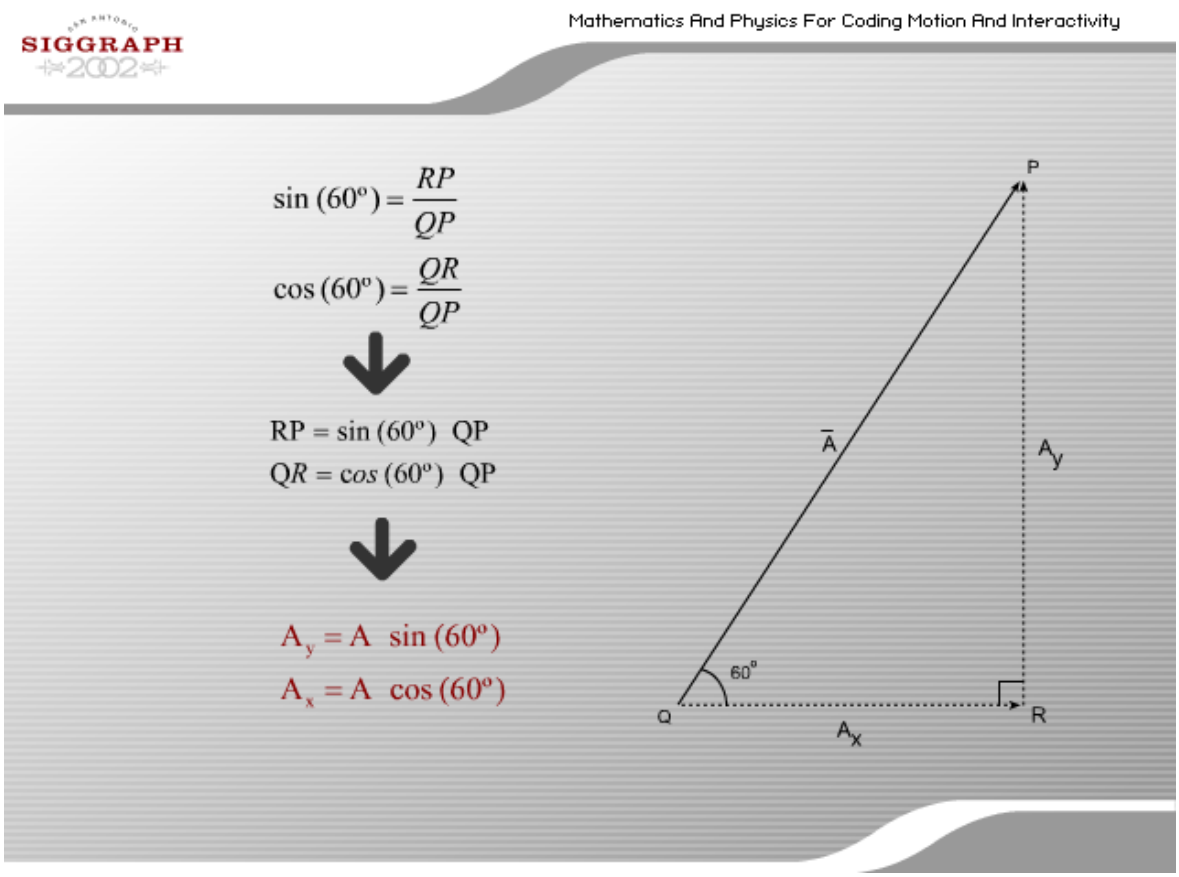

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Components
:: Perpendicular projections of vector onto x and y axes
:: Represent "horizontal" and "vertical" motion of object
:: Basis of all motion in CG


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Components
:: Only way to move objects in CG
:: Position on canvas is described as ordered pair of coordinates
:: Dynamic motion is impossible without vector components

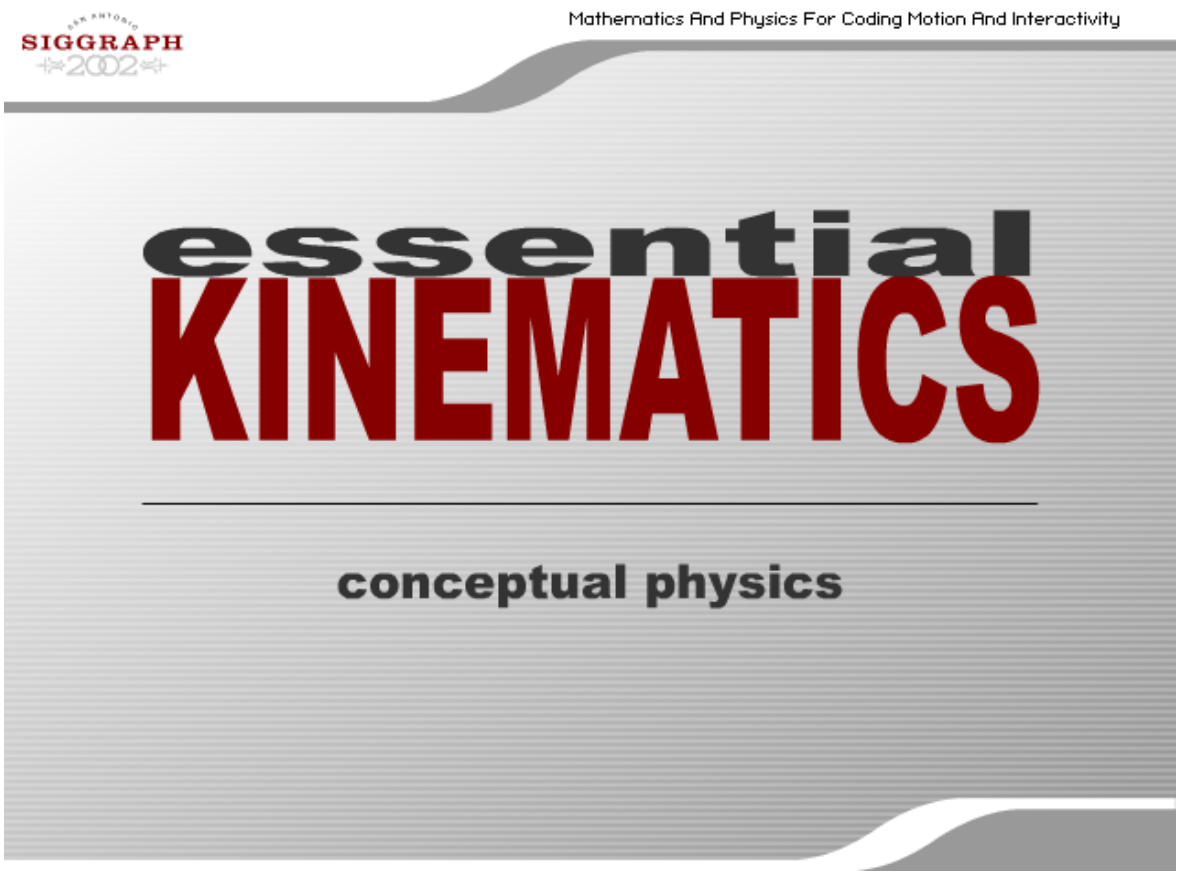

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Concept: Components
:: When an object is moved from A to B, it's equivalent to moving object 'x' amount along x-axis and then 'y' amount along y-axis

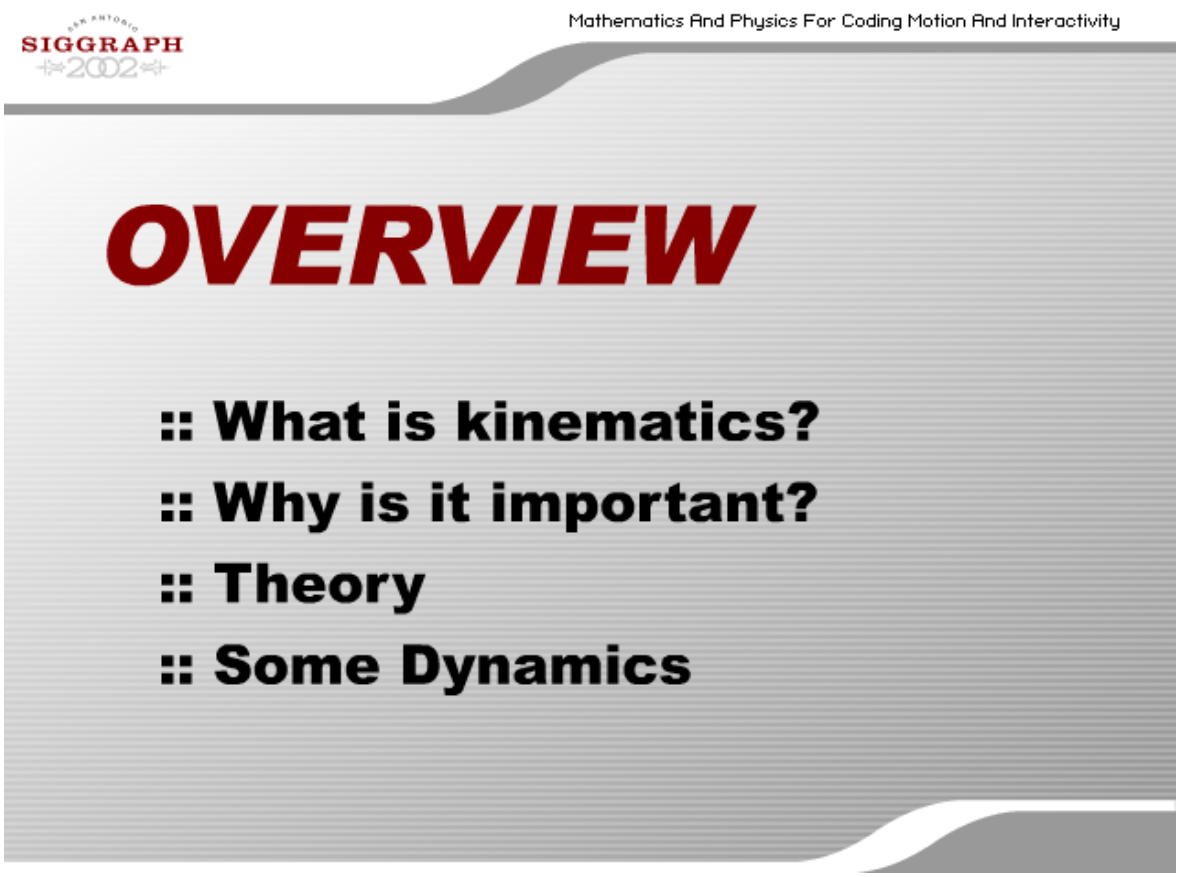

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Concept: Components
:: Components give us these 'x' and 'y' amounts
:: 'x' component is called horizontal component and 'y' component is called vertical component

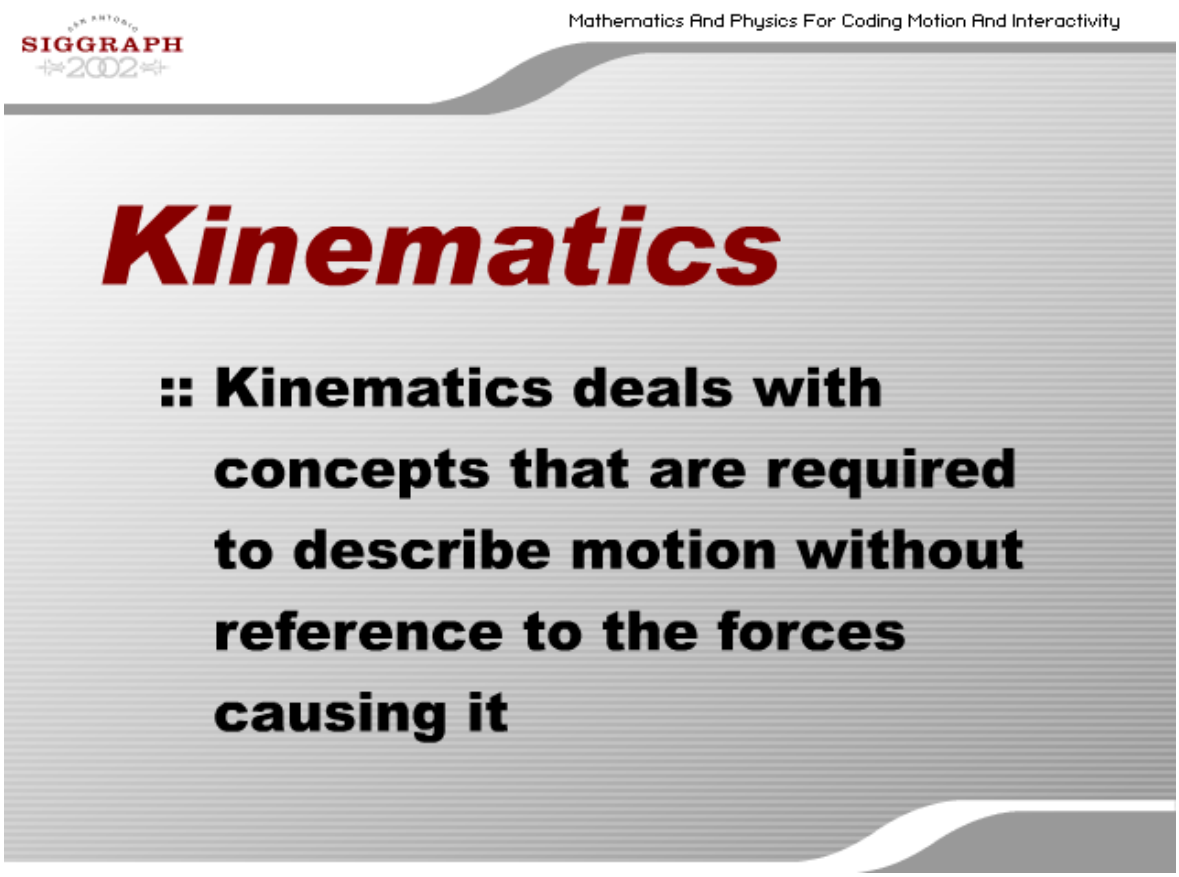

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Resolving Components
:: Fancy way of saying - "to find x and y components for a vector
:: Trigonometric Method (look in notes for conceptual overview)

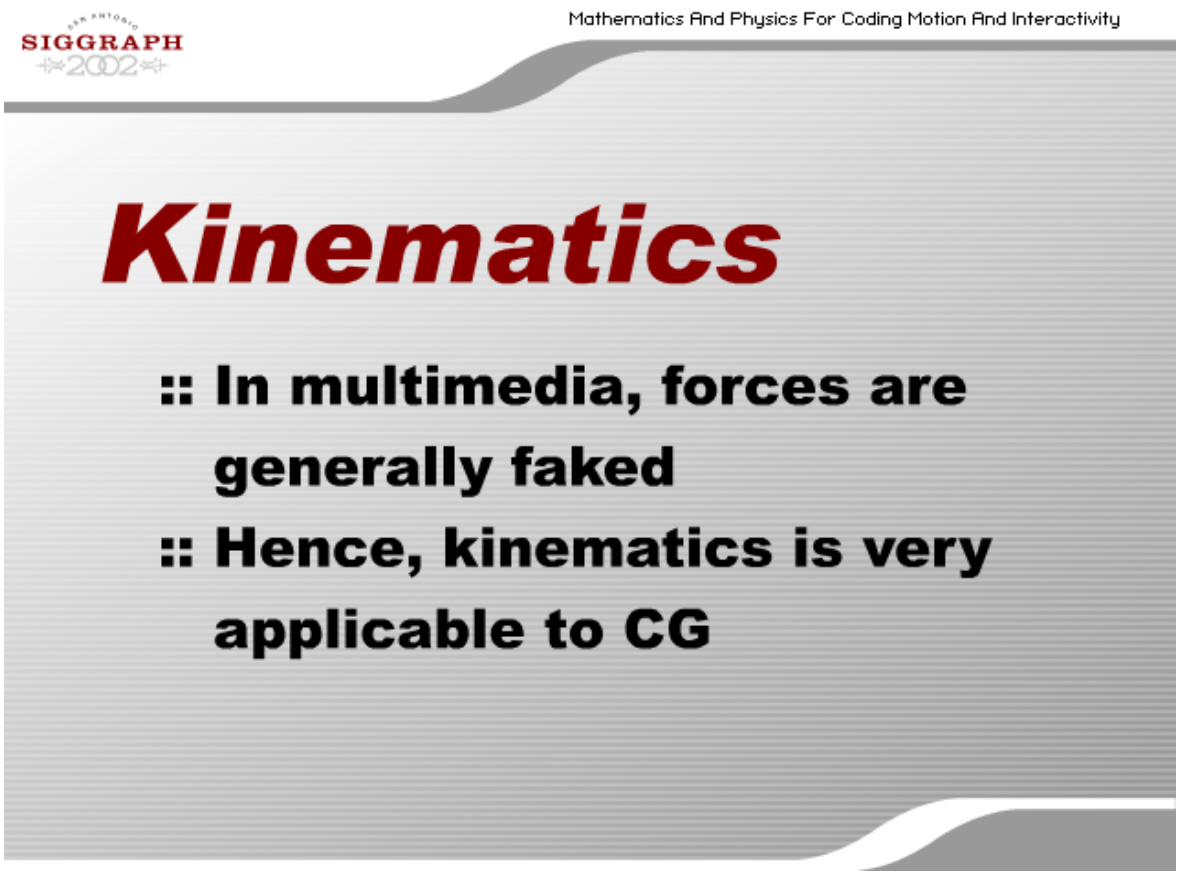

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
sin (60°) = RP / QP
cos (60°) = QR / QP
RP = sin (60°) QP
QR = cos (60°) QP
A_y = A sin (60°)
A_x = A cos (60°)
P
A
A_y
60°
Q
A_X
R

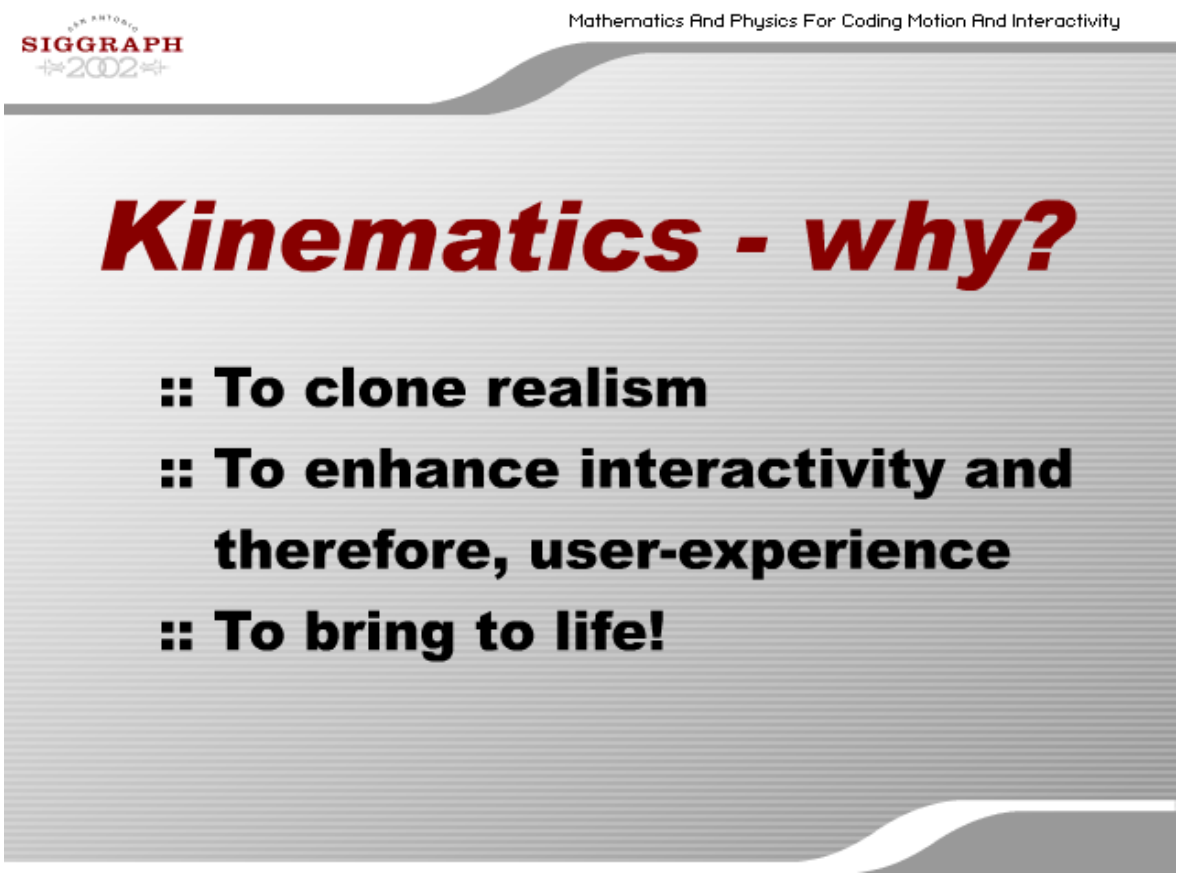

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
BREAK

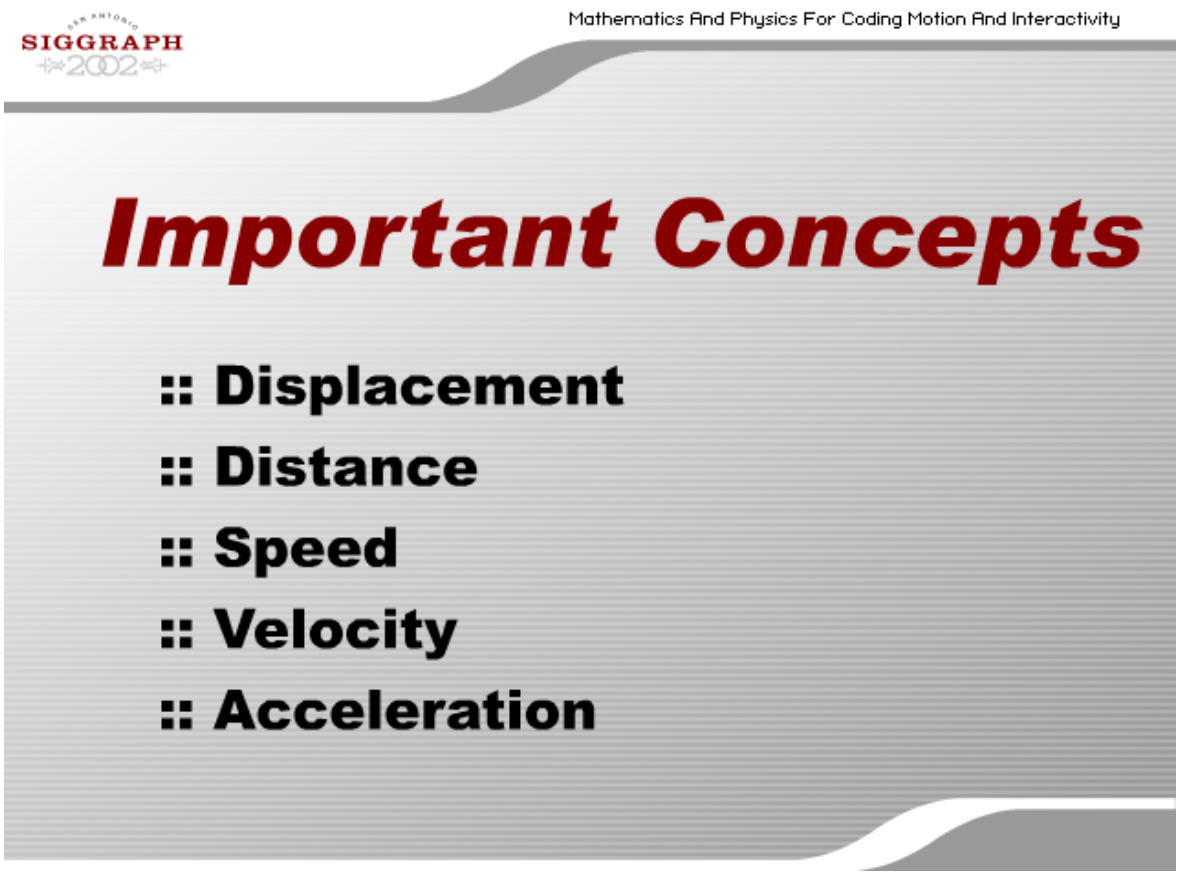

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
essential
KINEMATICS
conceptual physics

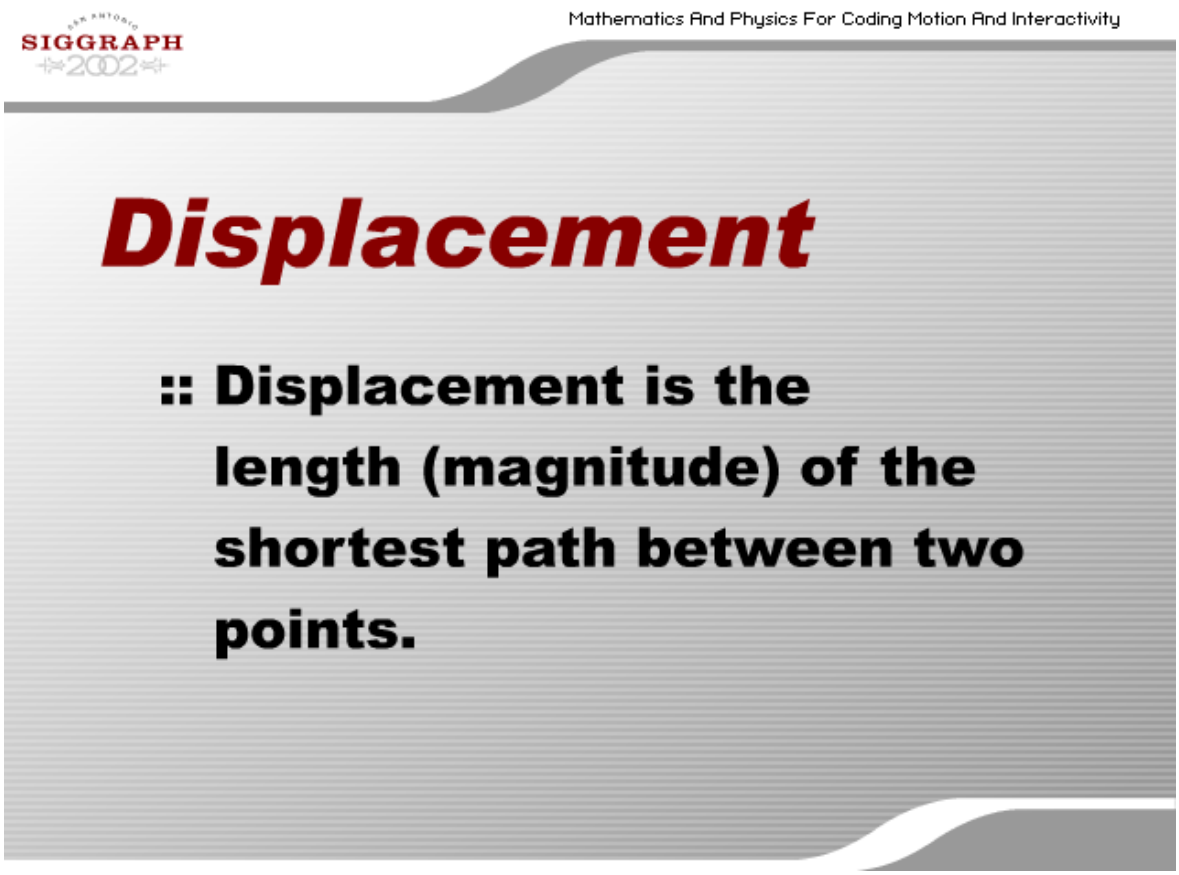

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
OVERVIEW
:: What is kinematics?
:: Why is it important?
:: Theory
:: Some Dynamics

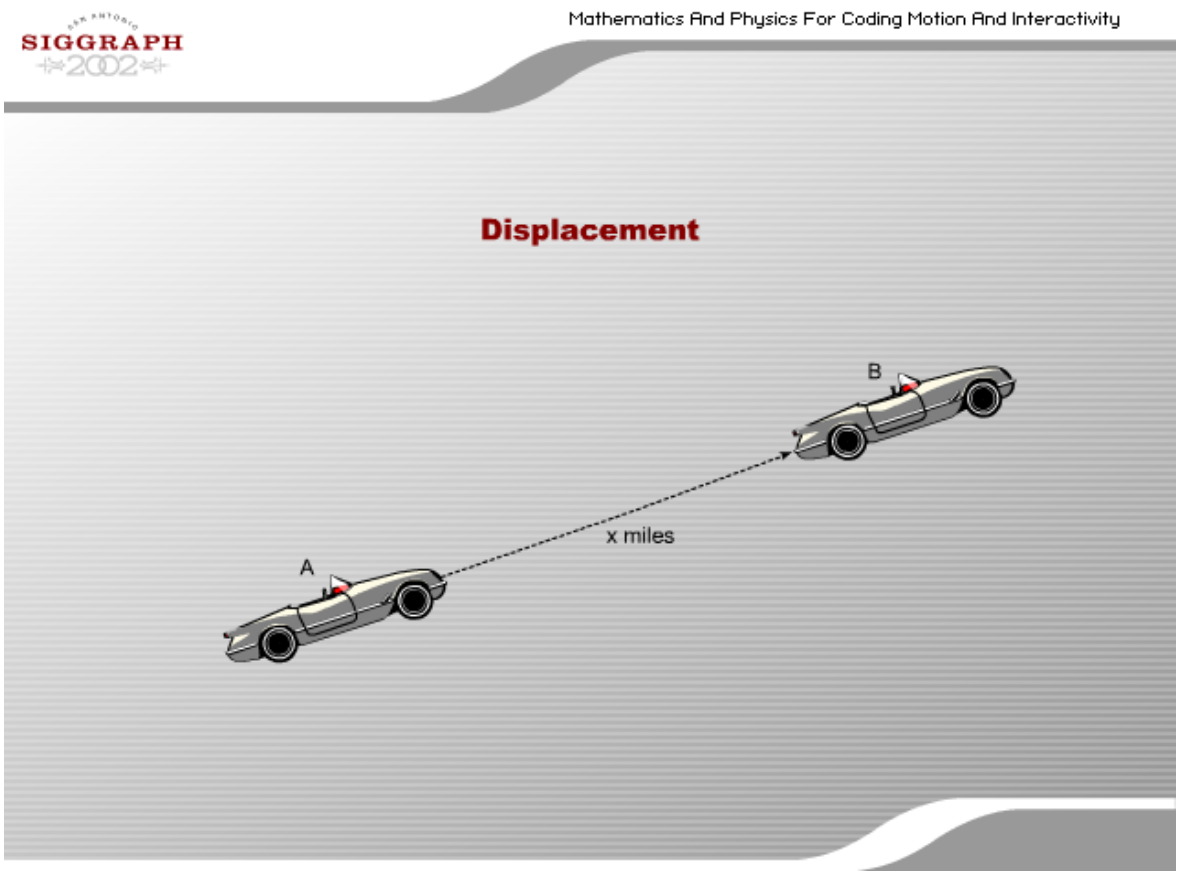

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Kinematics
:: Kinematics deals with concepts that are required to describe motion without reference to the forces causing it

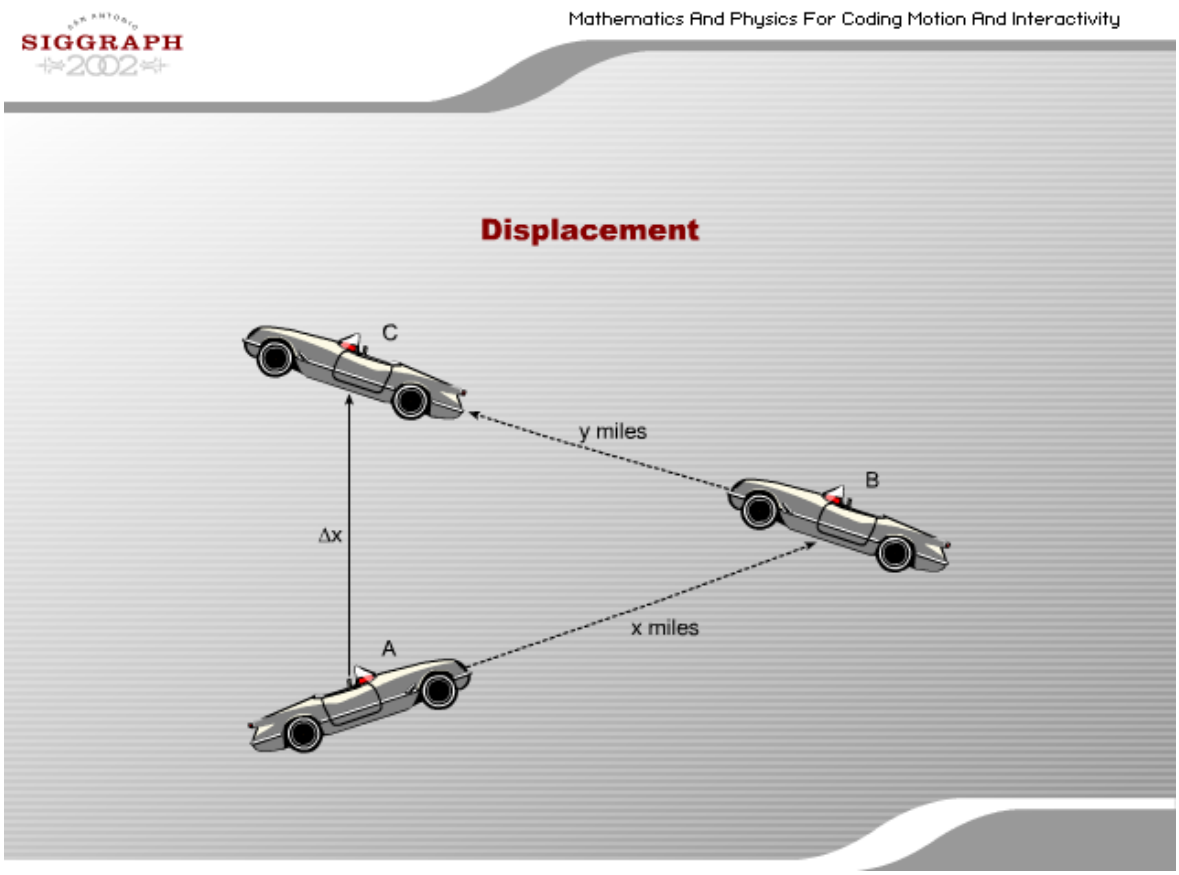

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Kinematics
:: In multimedia, forces are generally faked
:: Hence, kinematics is very applicable to CG

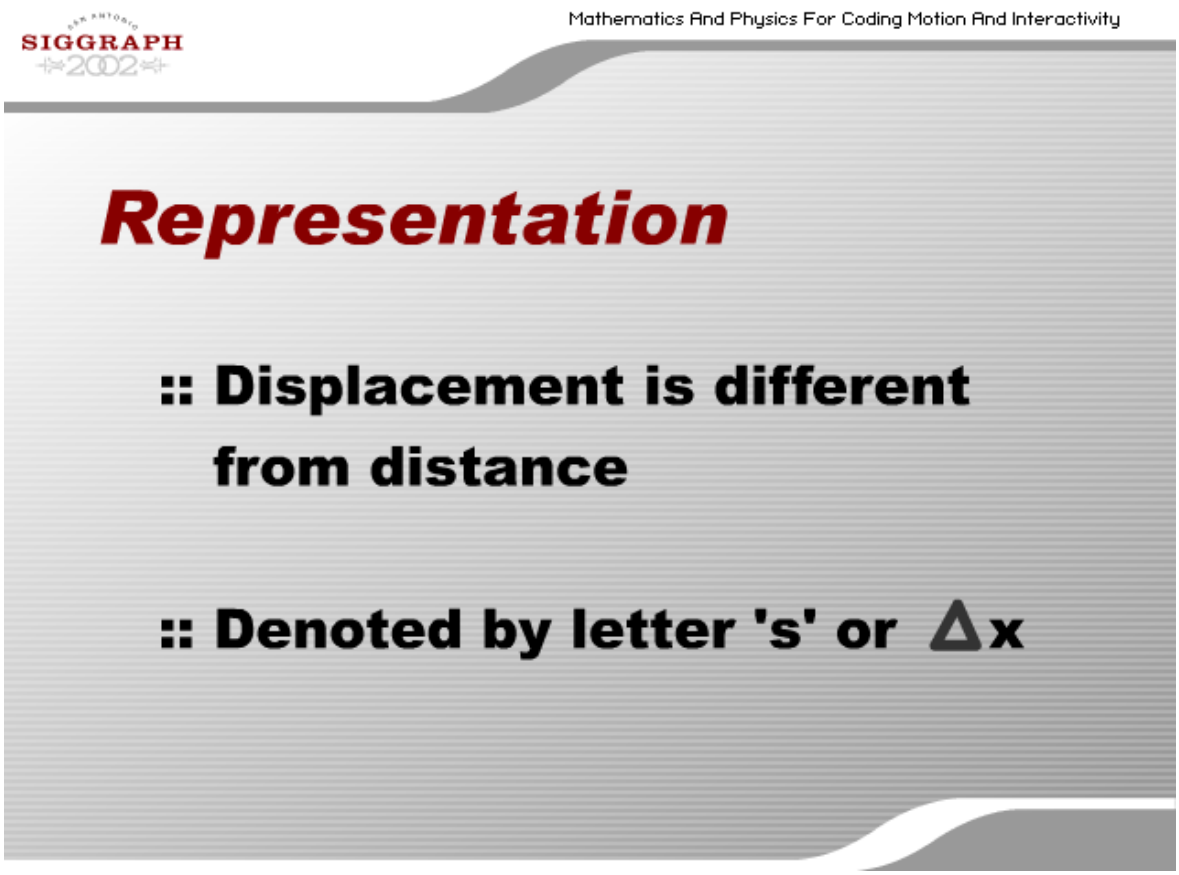

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Kinematics - why?
:: To clone realism
:: To enhance interactivity and therefore, user-experience
:: To bring to life!

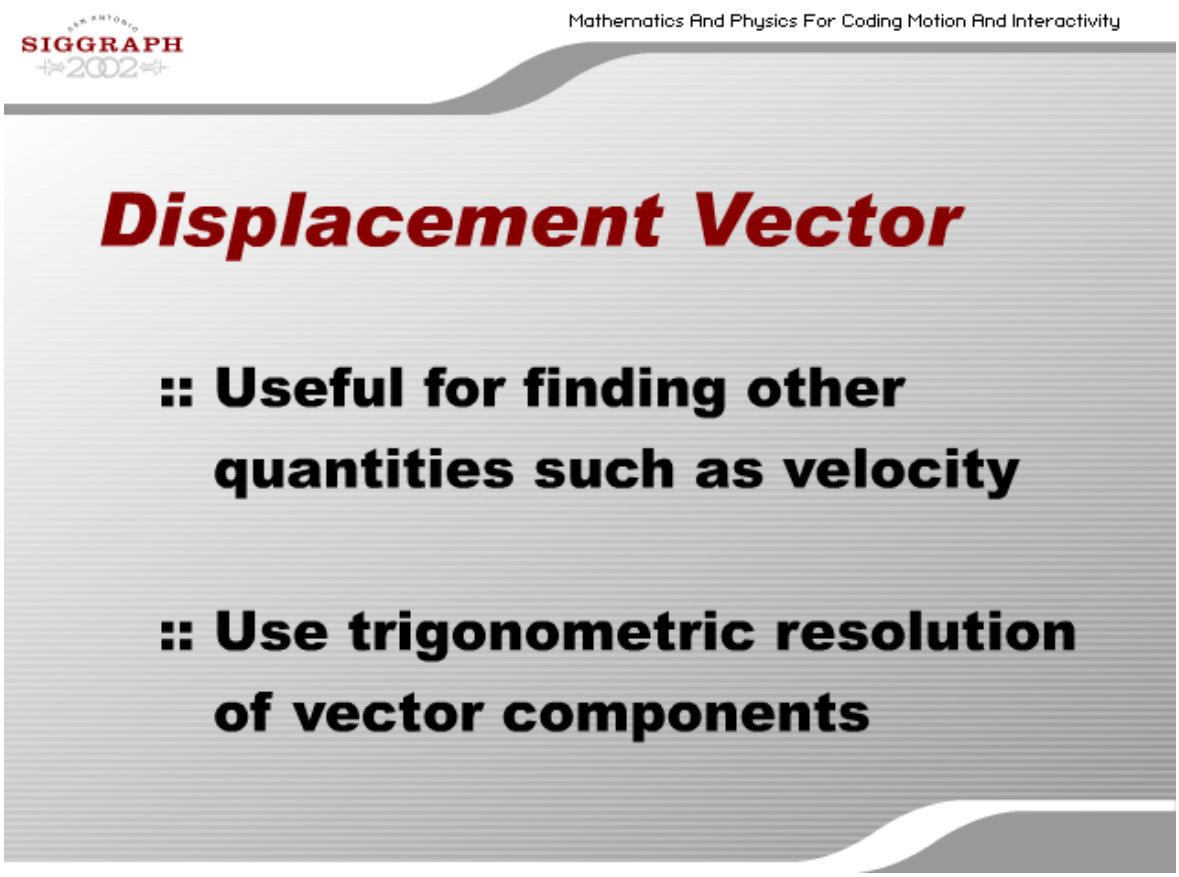

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Important Concepts
:: Displacement
:: Distance
:: Speed
:: Velocity
:: Acceleration

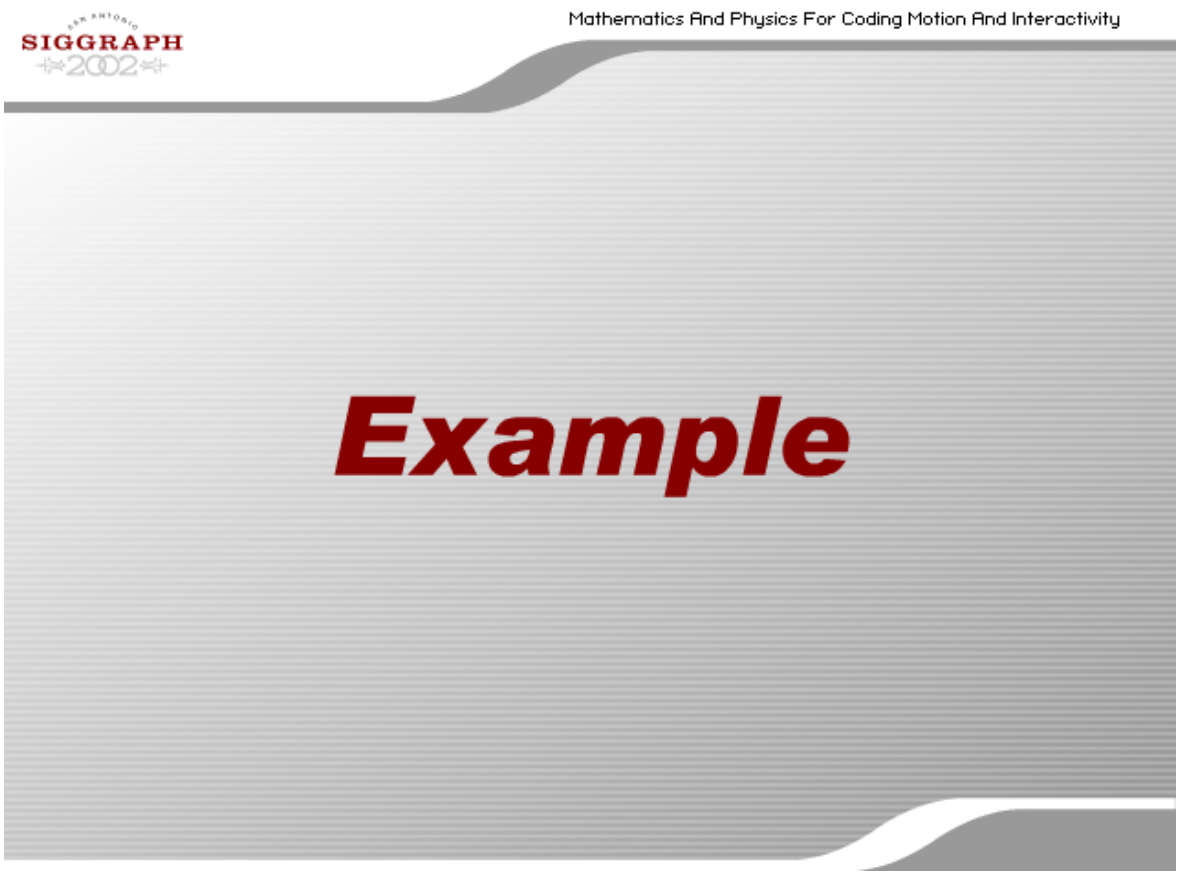

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Displacement
:: Displacement is the length (magnitude) of the shortest path between two points.

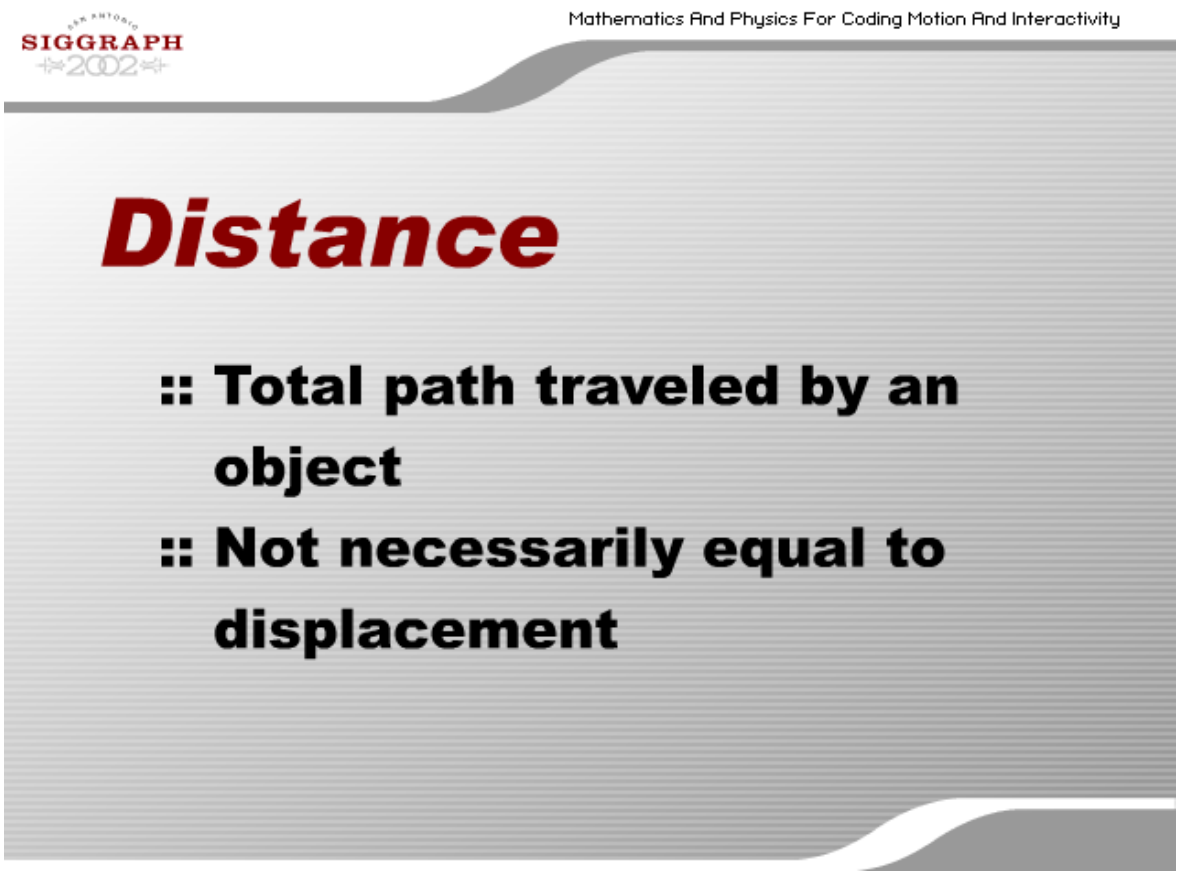

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Displacement
B
x miles
A

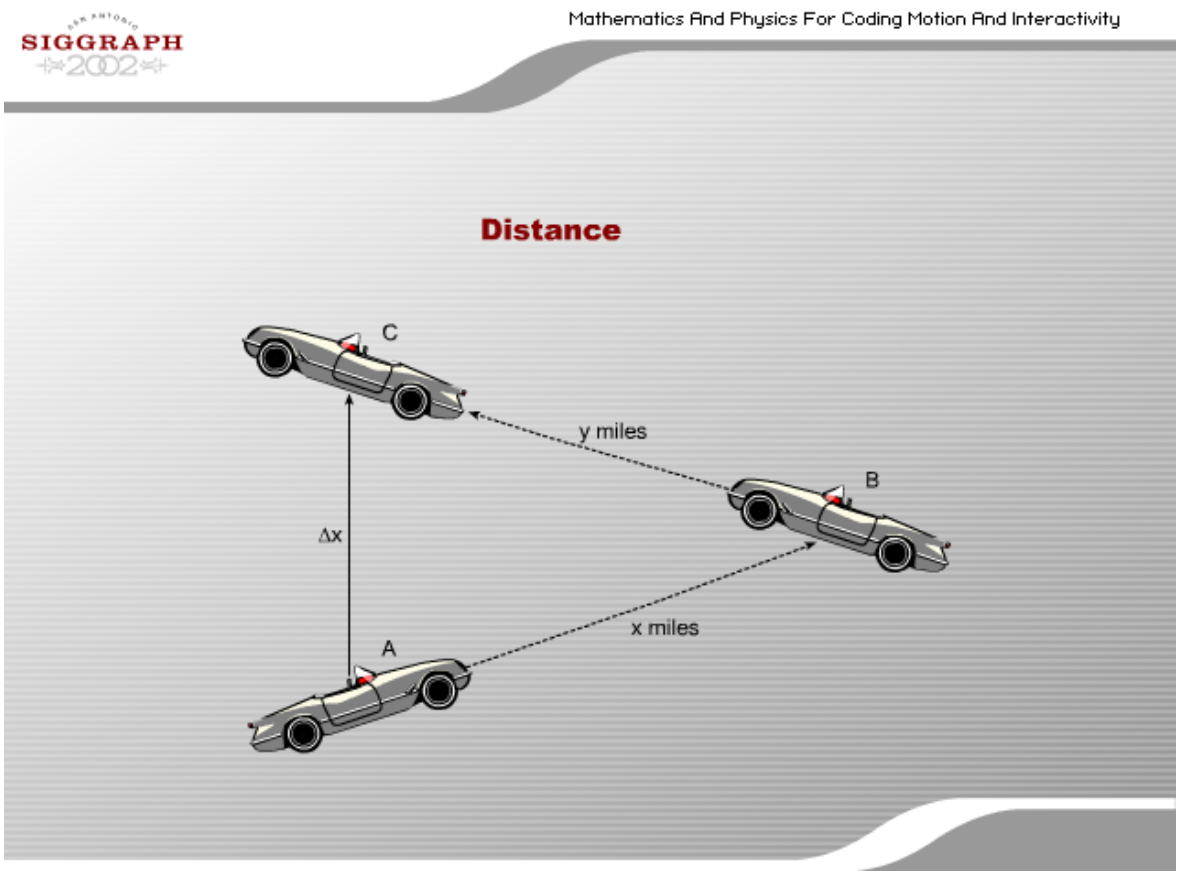

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Displacement
C
y miles
B
Δx
x miles
A

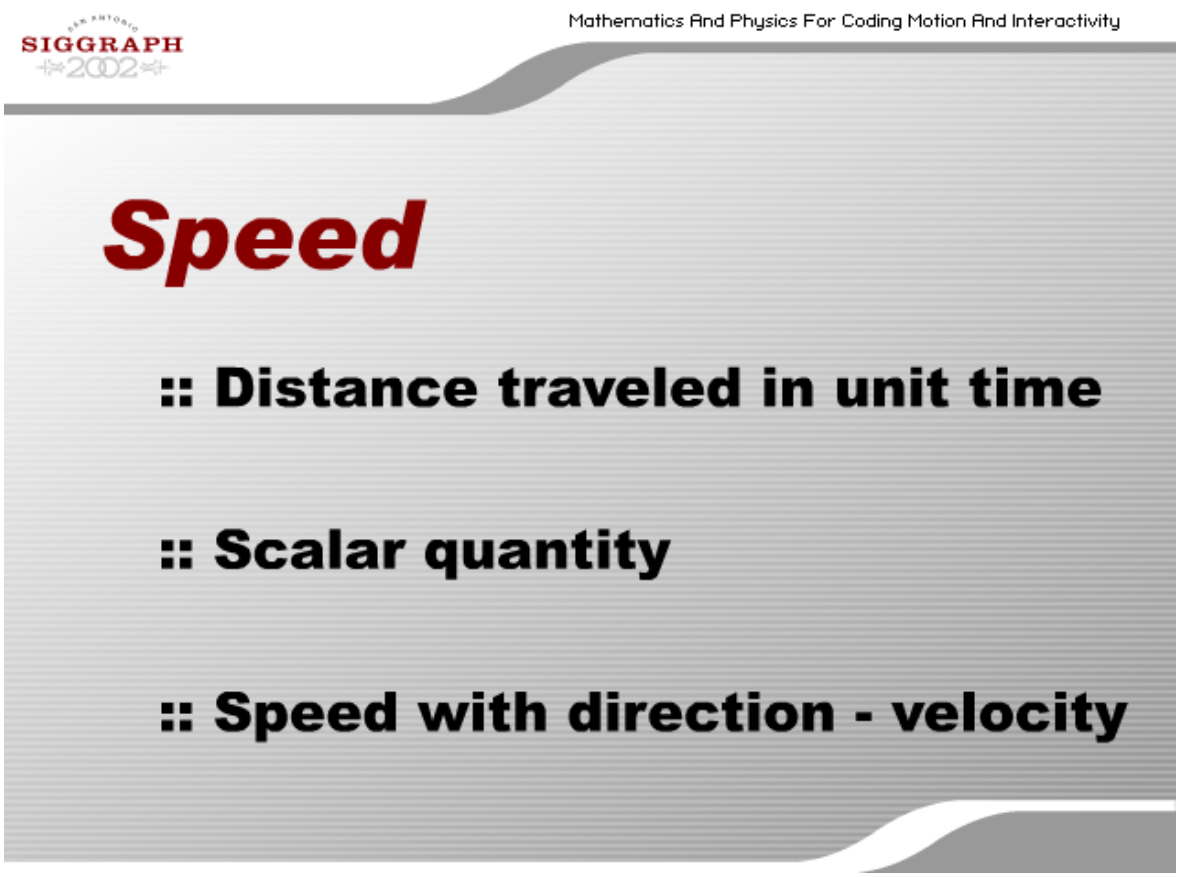

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Representation
:: Displacement is different from distance
:: Denoted by letter 's' or Δx

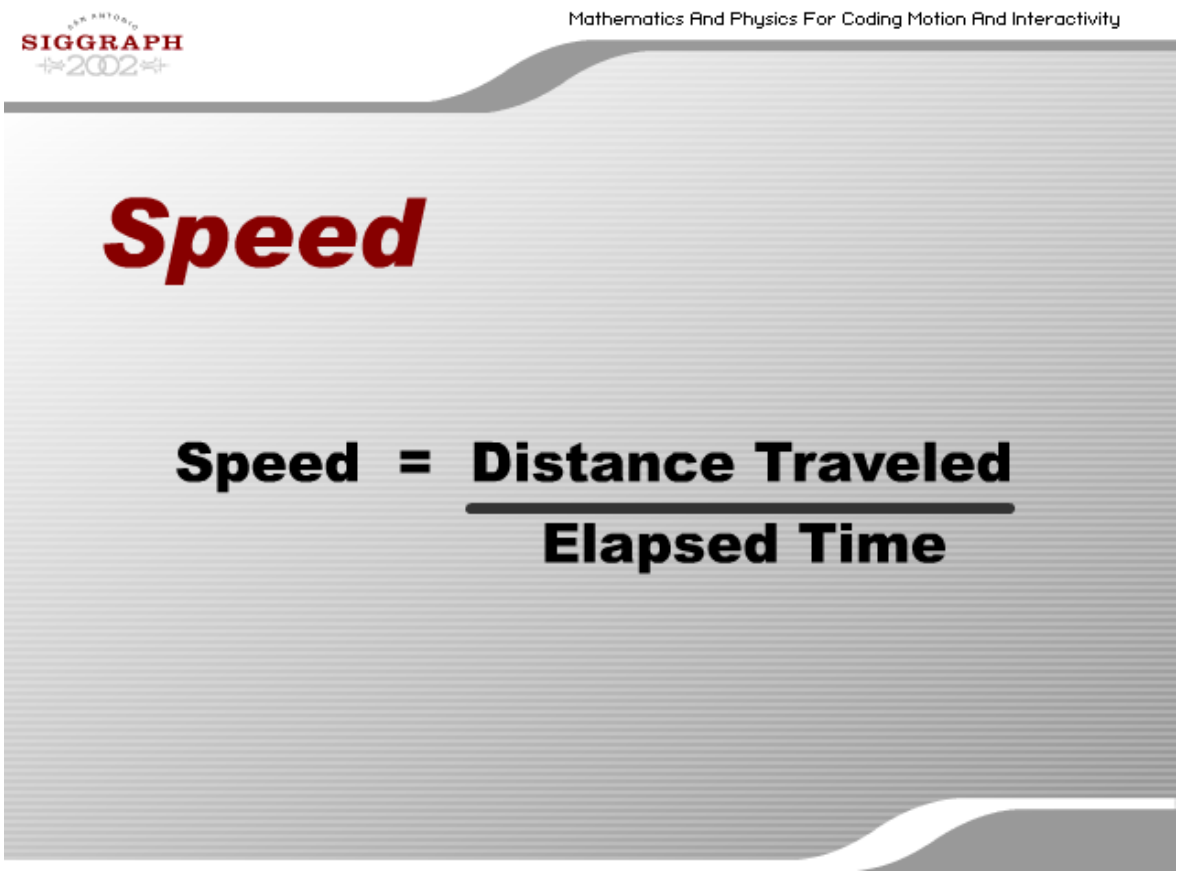

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Displacement Vector
:: Useful for finding other quantities such as velocity
:: Use trigonometric resolution of vector components

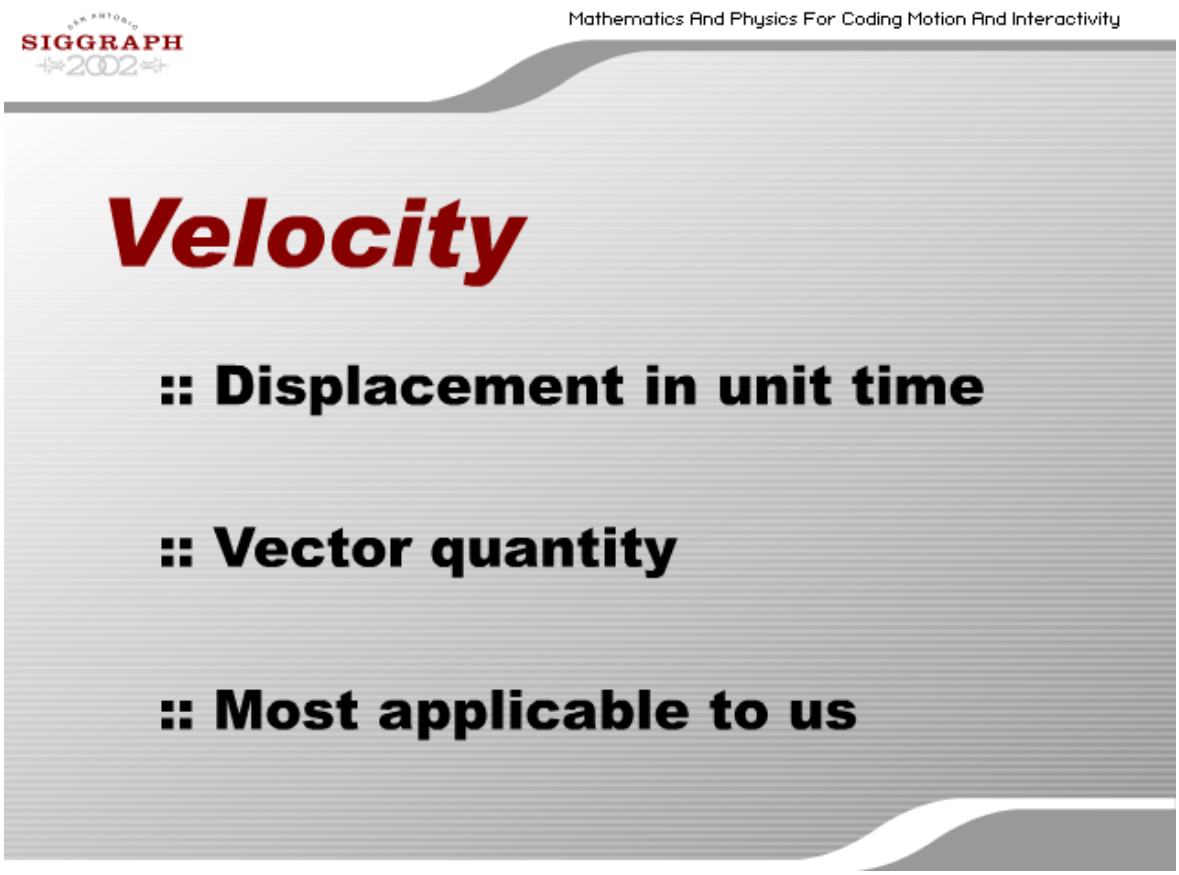

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

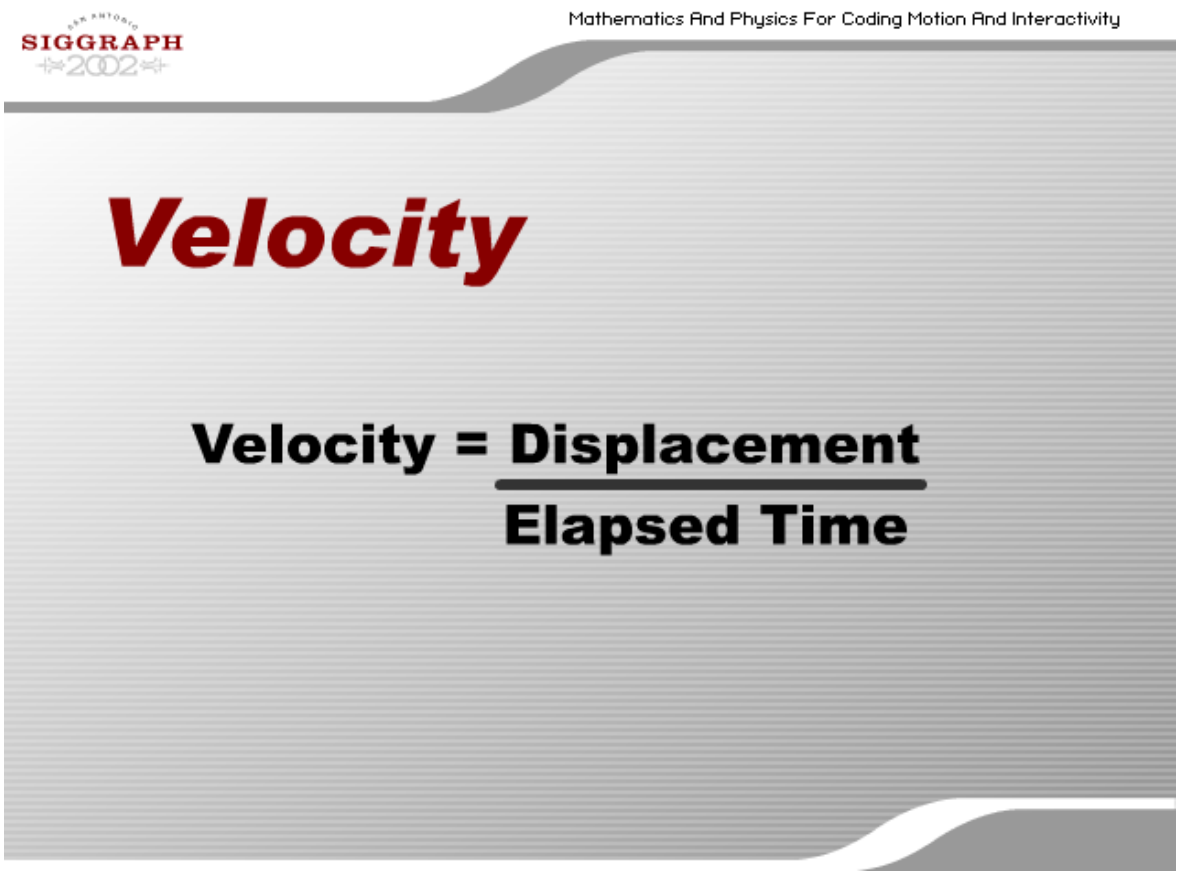

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Distance
:: Total path traveled by an object
:: Not necessarily equal to displacement

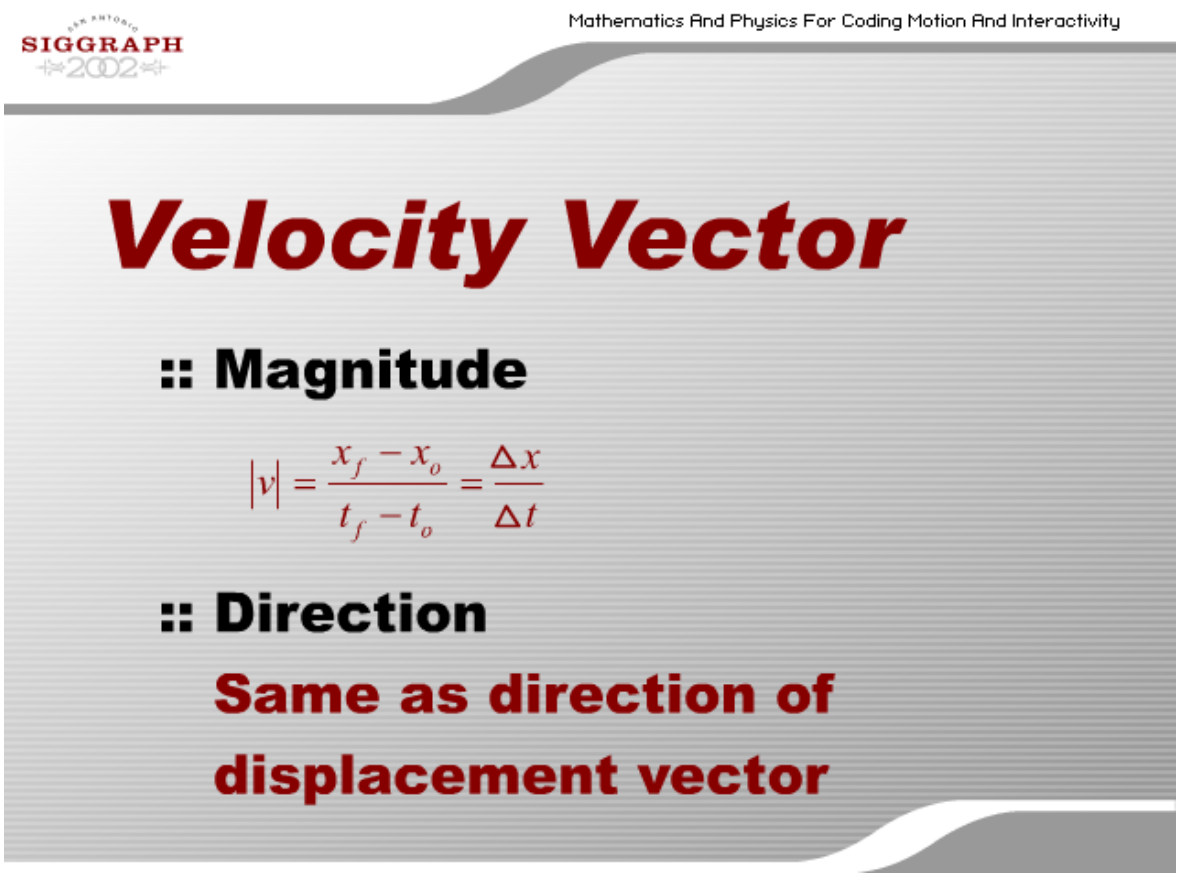

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Distance
C
y miles
B
Δx
x miles
A

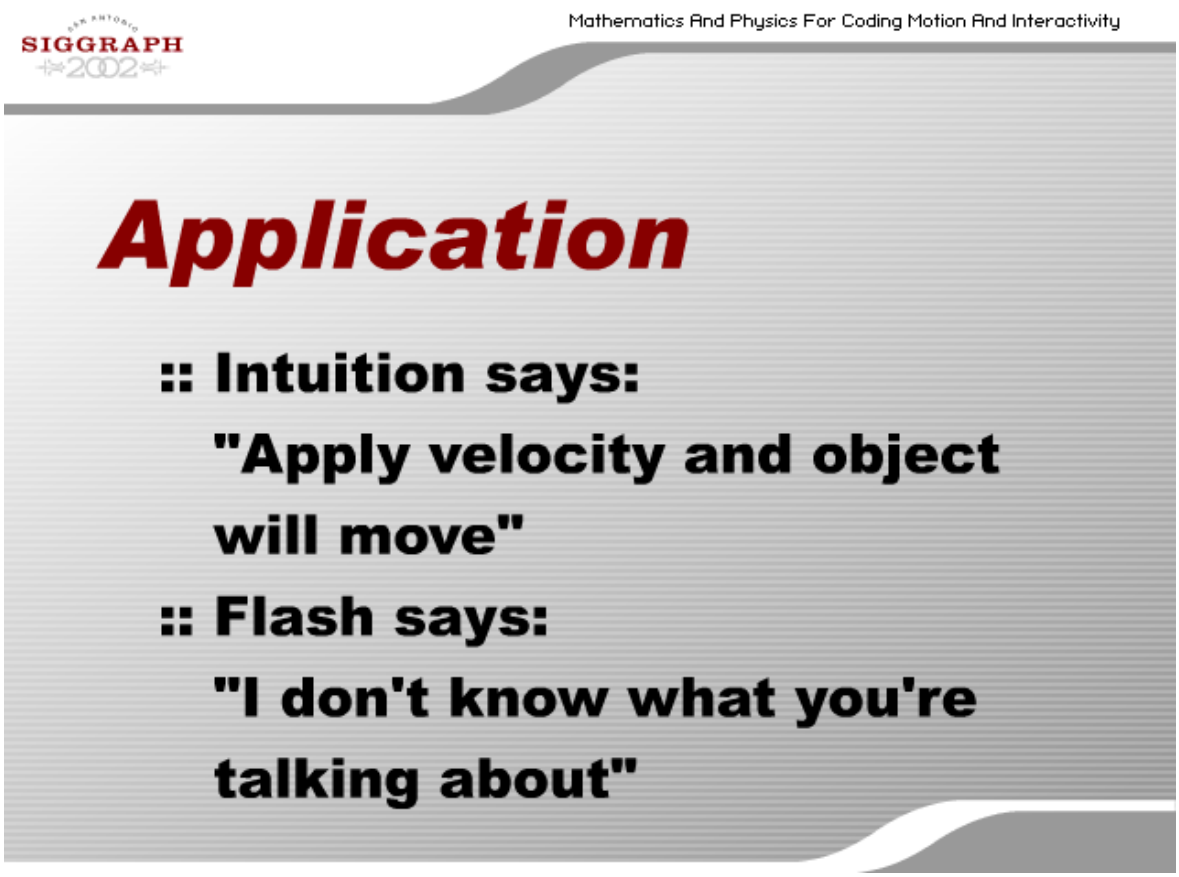

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Speed
:: Distance traveled in unit time
:: Scalar quantity
:: Speed with direction - velocity

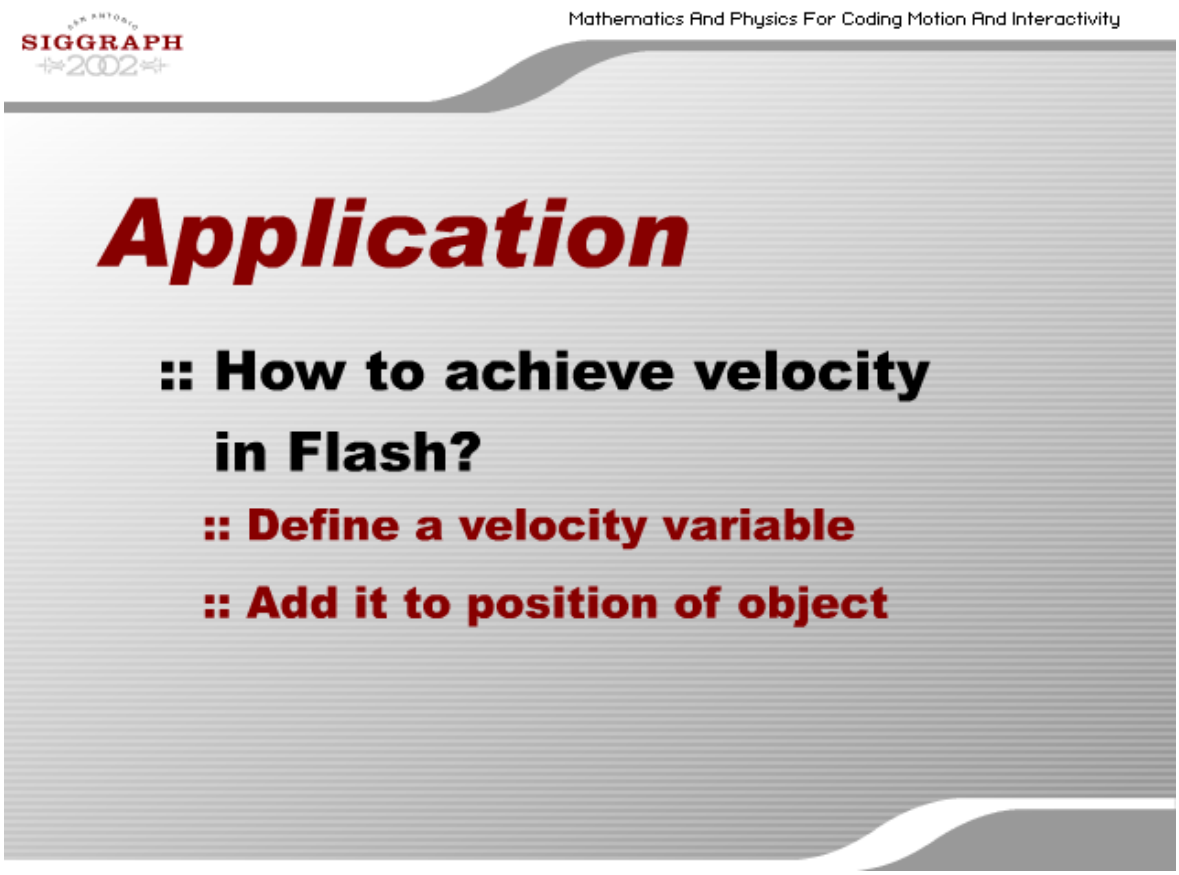

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Speed
Speed = Distance Traveled / Elapsed Time

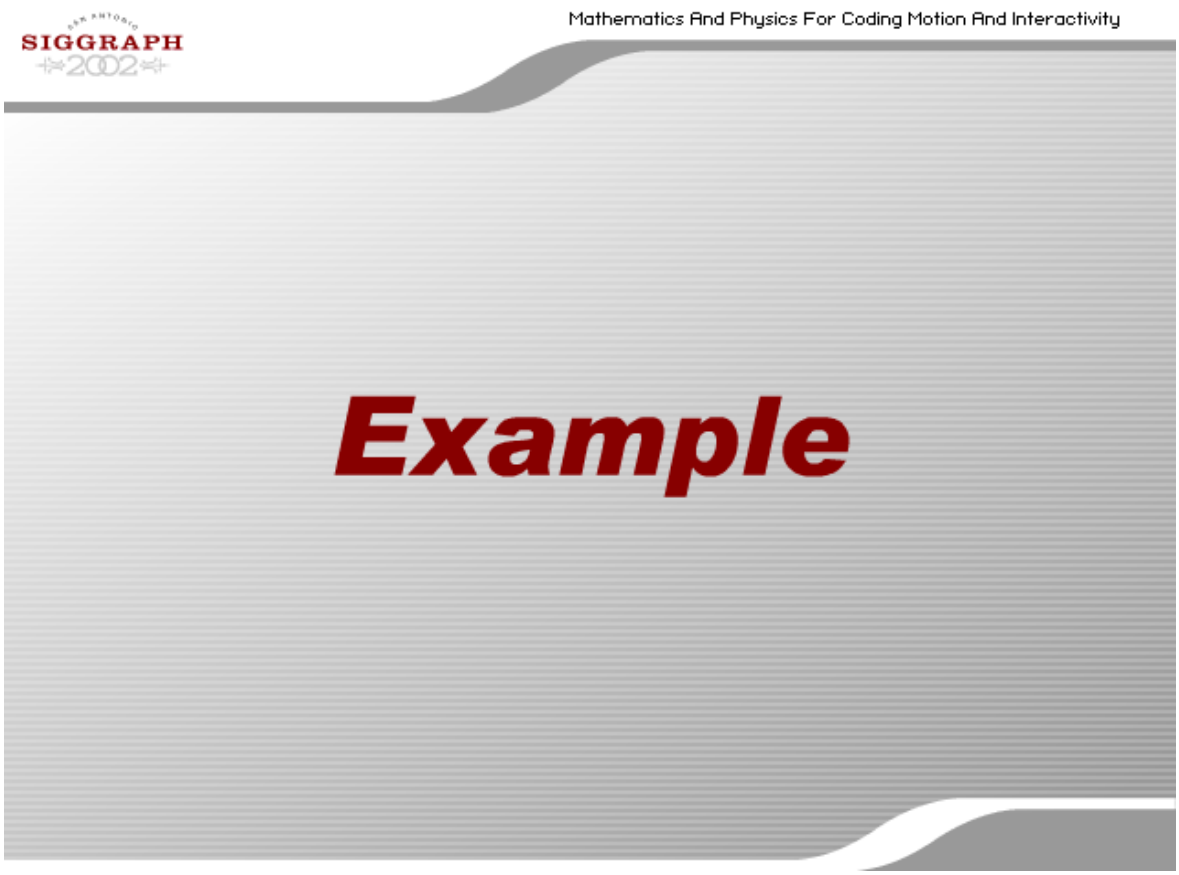

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Velocity
:: Displacement in unit time
:: Vector quantity
:: Most applicable to us

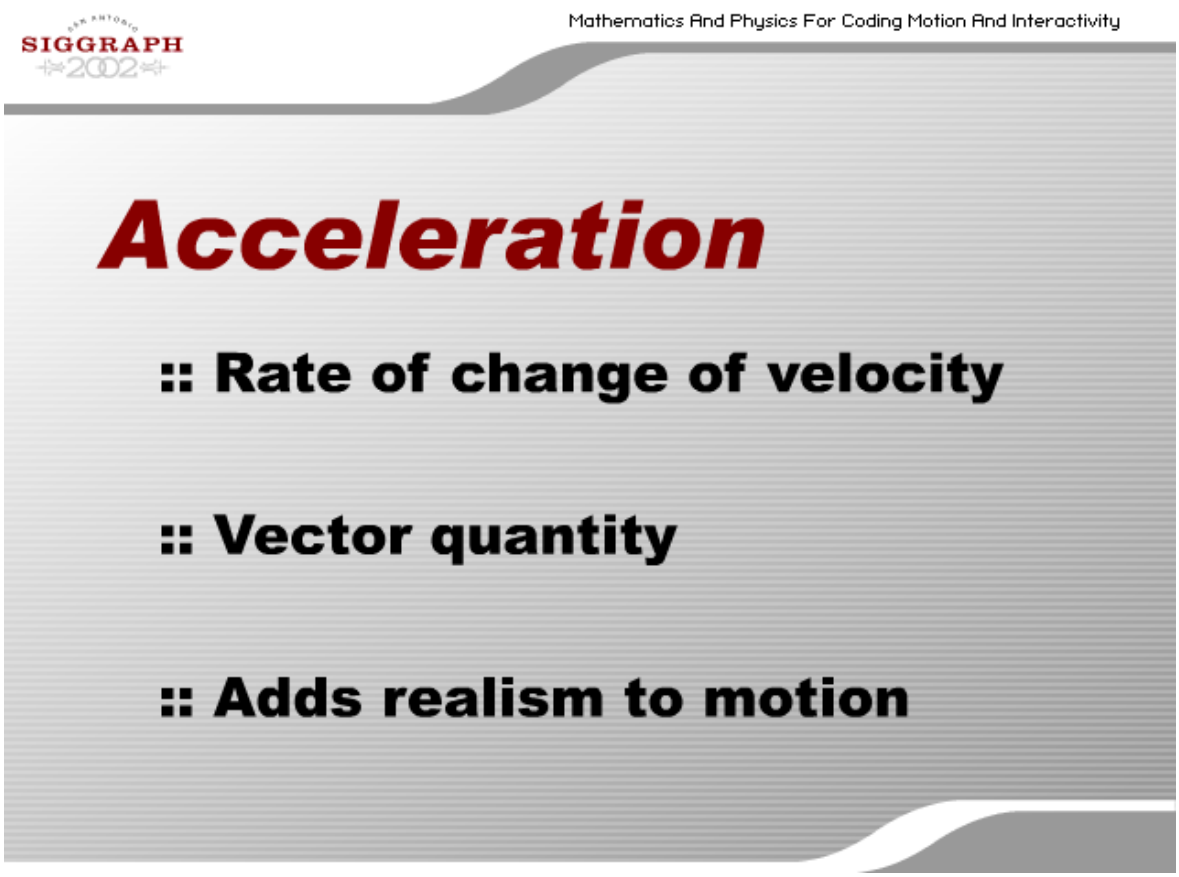

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Velocity
Velocity = Displacement
Elapsed Time

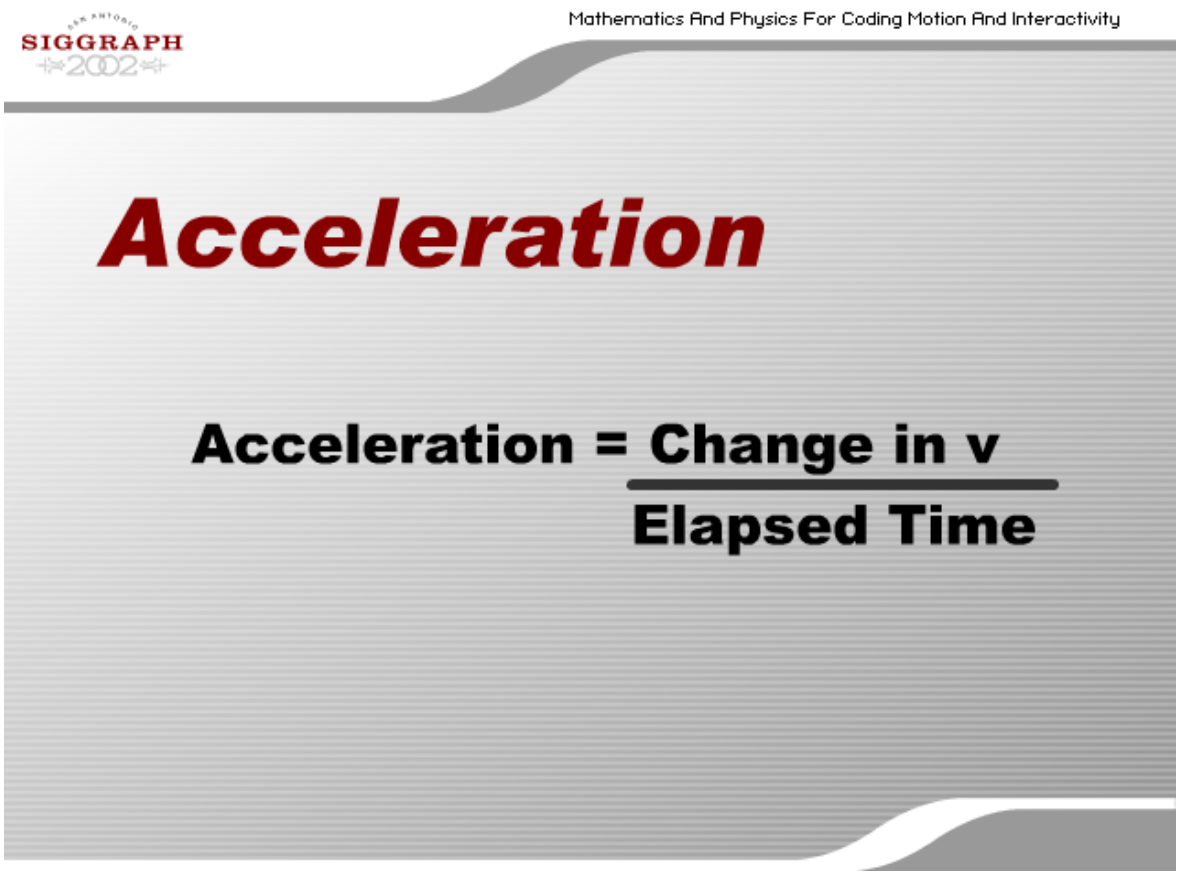

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Velocity Vector
:: Magnitude
$|v| = \frac{x_f - x_o}{t_f - t_o} = \frac{\Delta x}{\Delta t}$
:: Direction
Same as direction of displacement vector

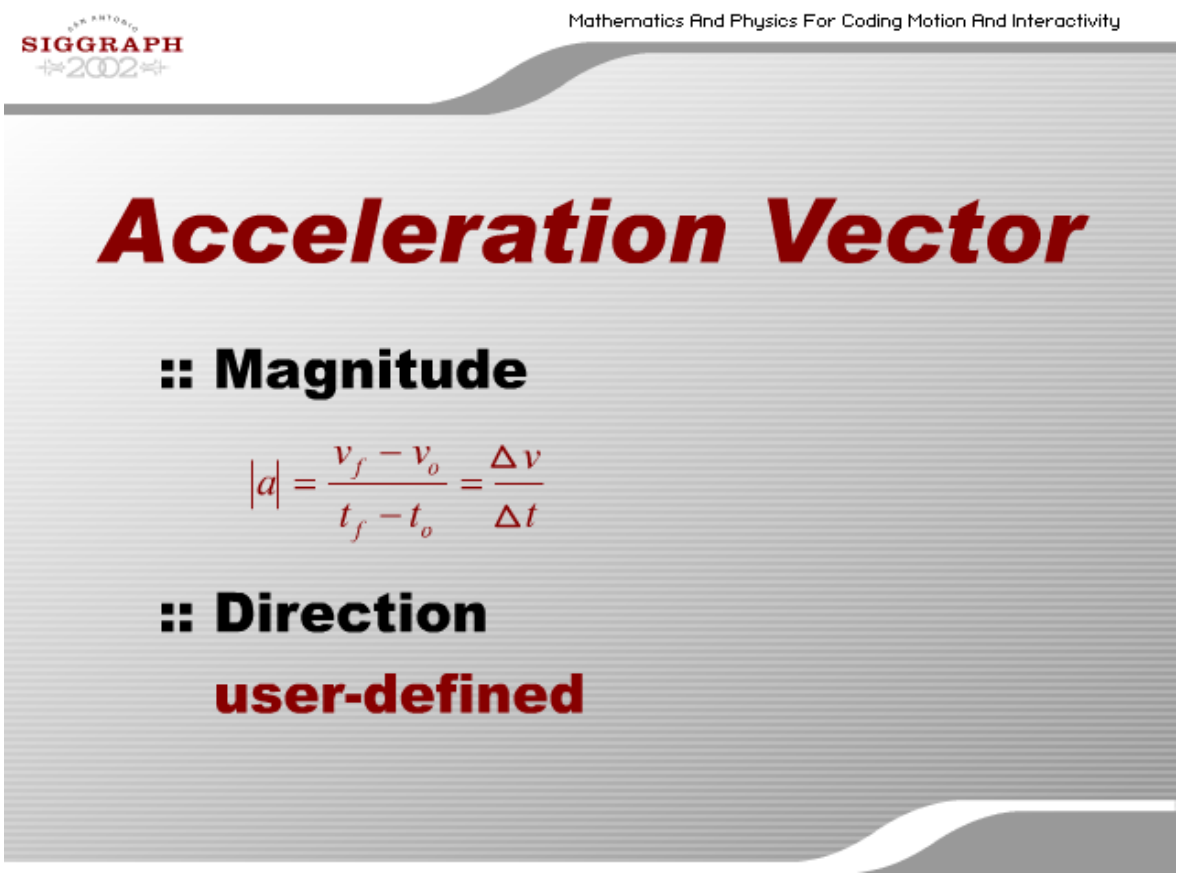

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Application
:: Intuition says:
"Apply velocity and object will move"
:: Flash says:
"I don't know what you're talking about"

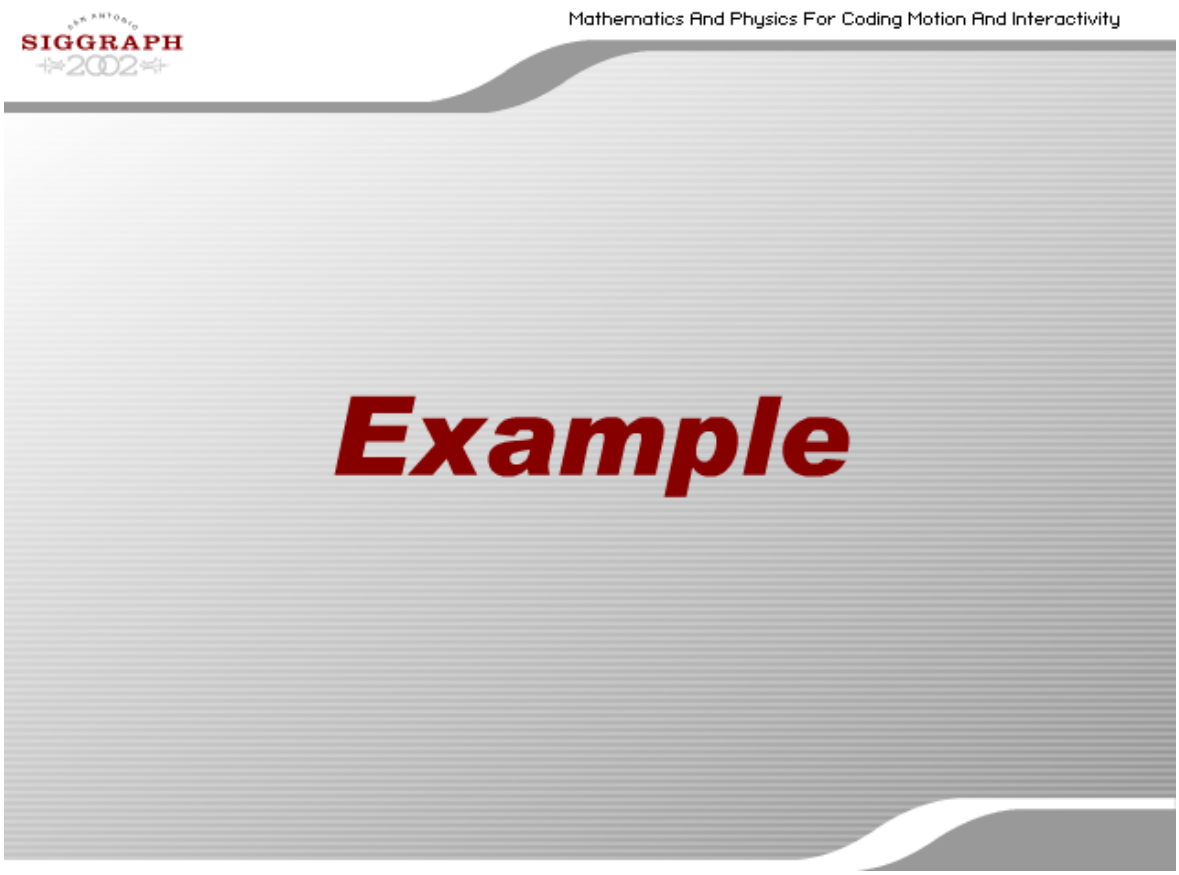

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Application
:: How to achieve velocity in Flash?
:: Define a velocity variable
:: Add it to position of object

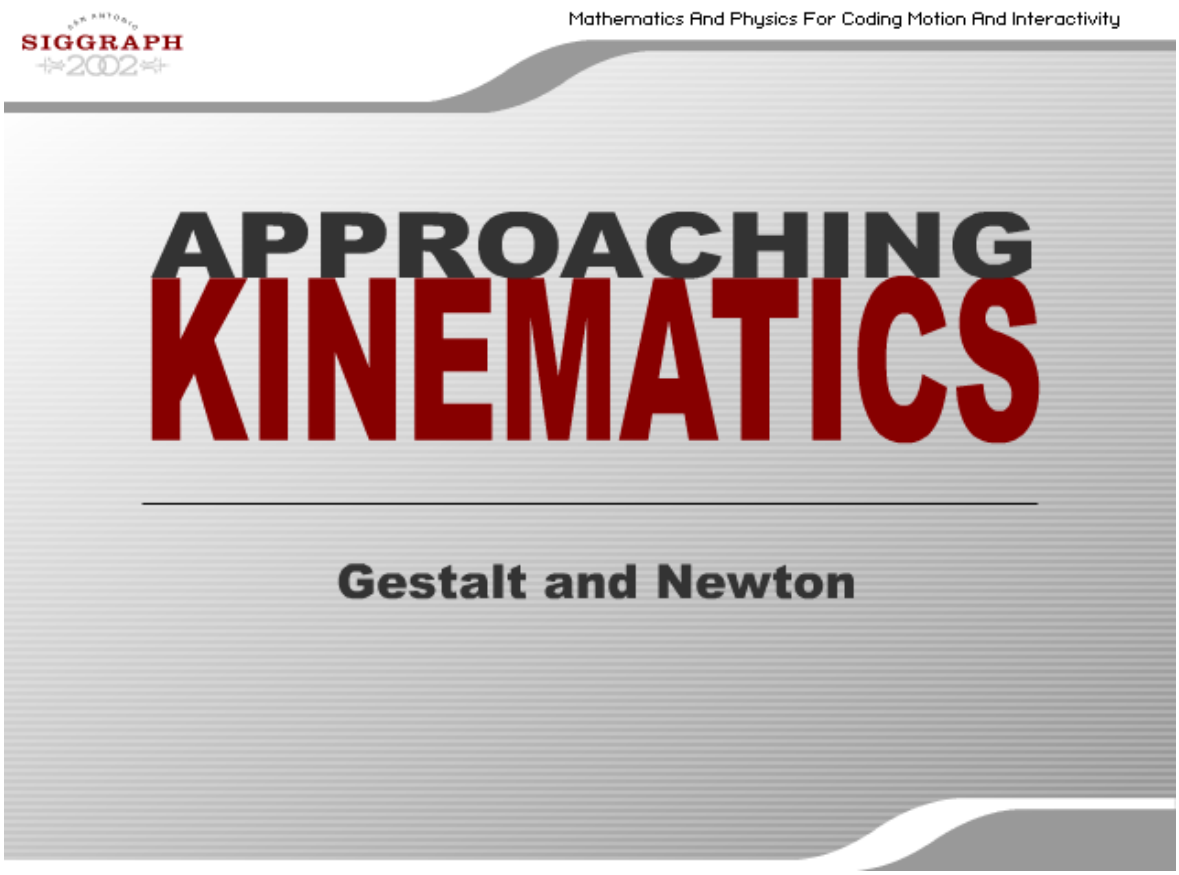

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

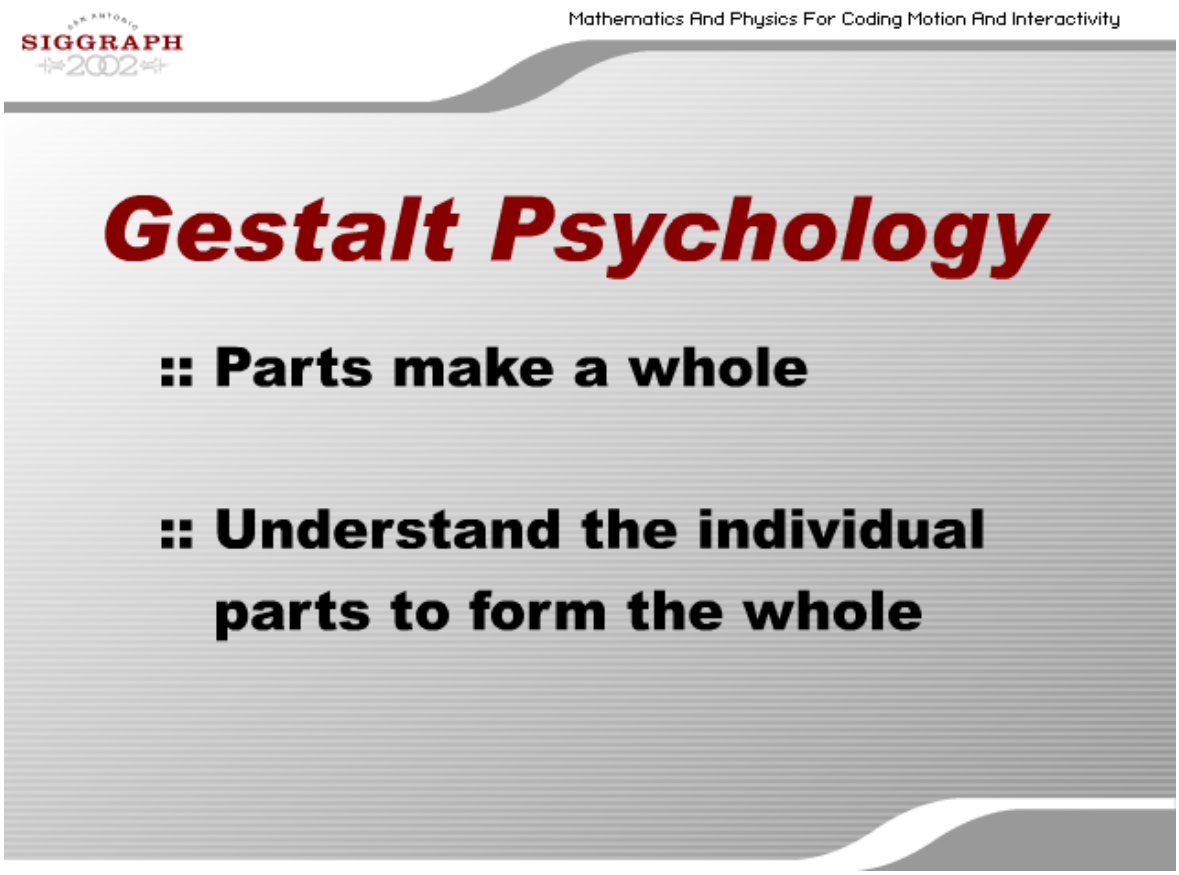

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Acceleration
:: Rate of change of velocity
:: Vector quantity
:: Adds realism to motion

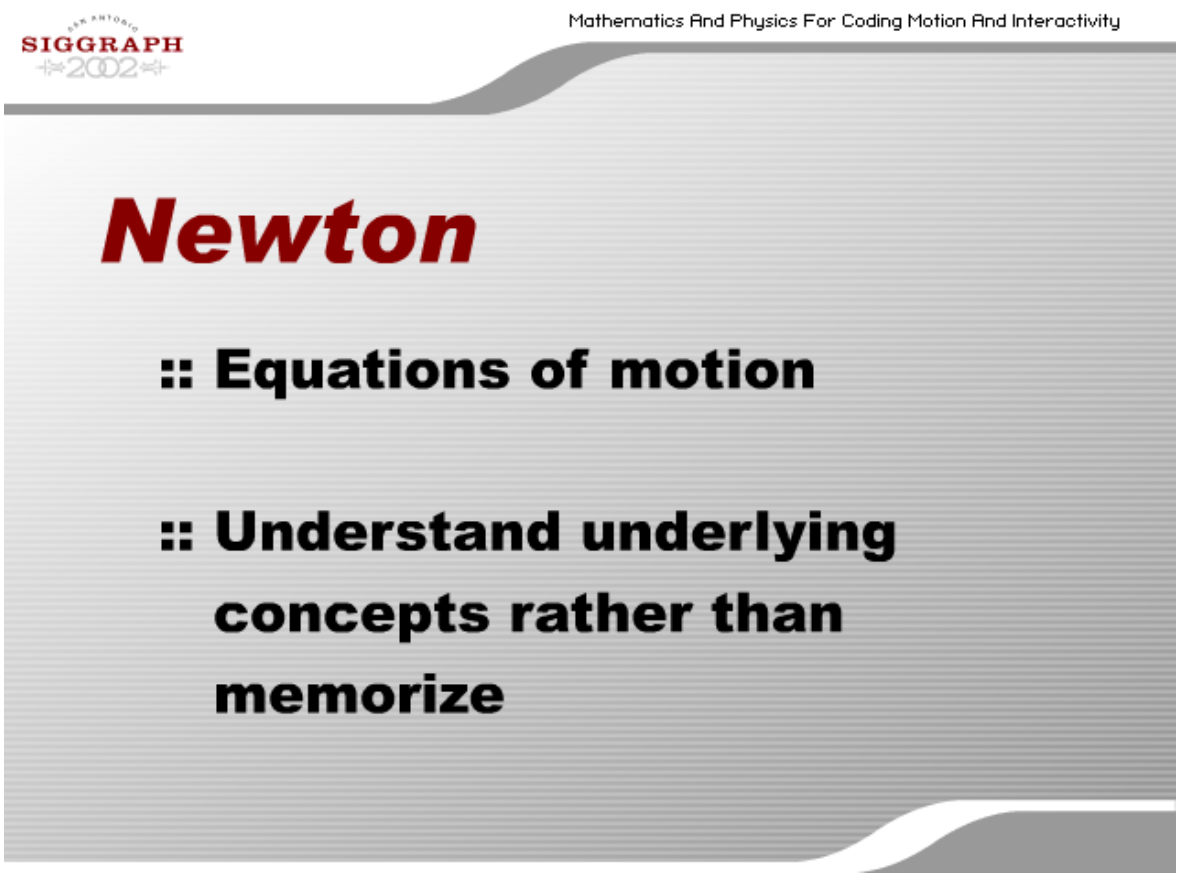

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Acceleration
Acceleration = Change in v
Elapsed Time

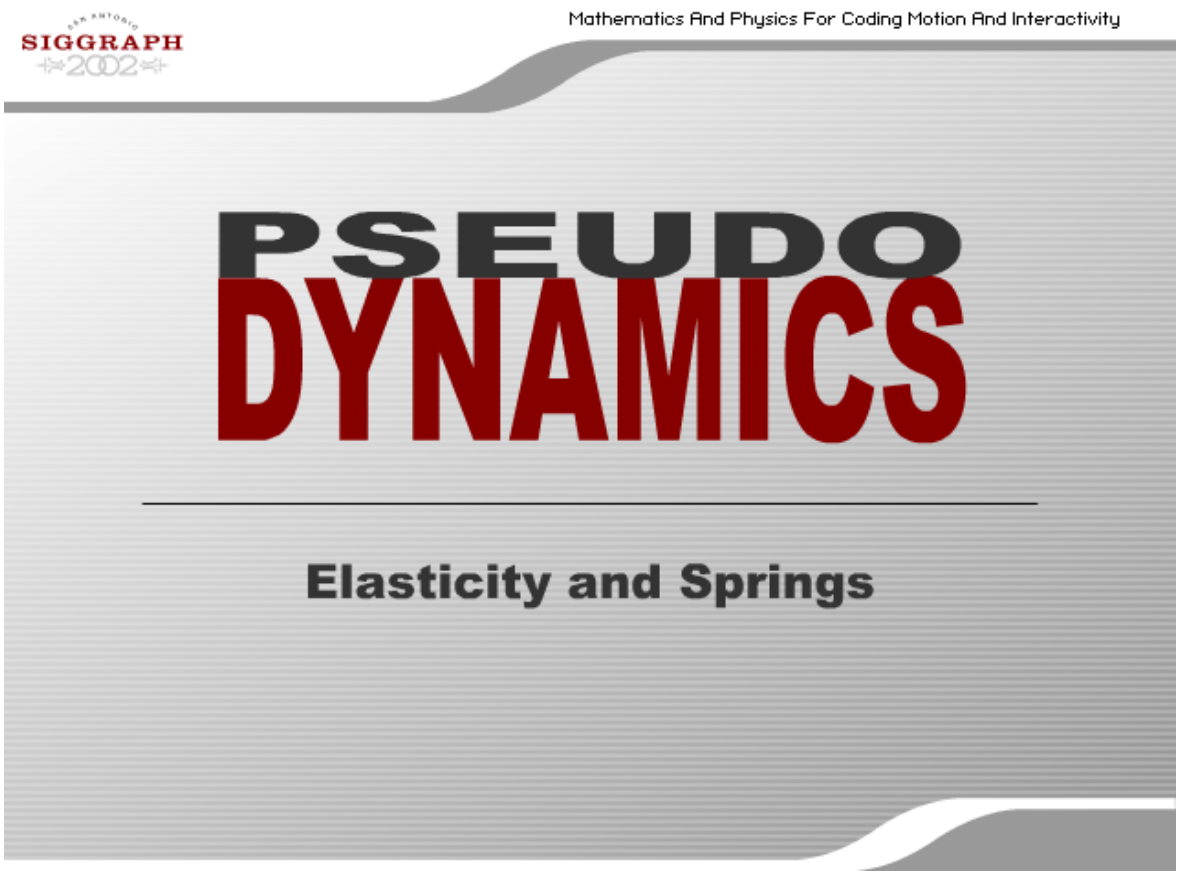

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Acceleration Vector
:: Magnitude
$|a| = \frac{v_f - v_o}{t_f - t_o} = \frac{\Delta v}{\Delta t}$
:: Direction
user-defined

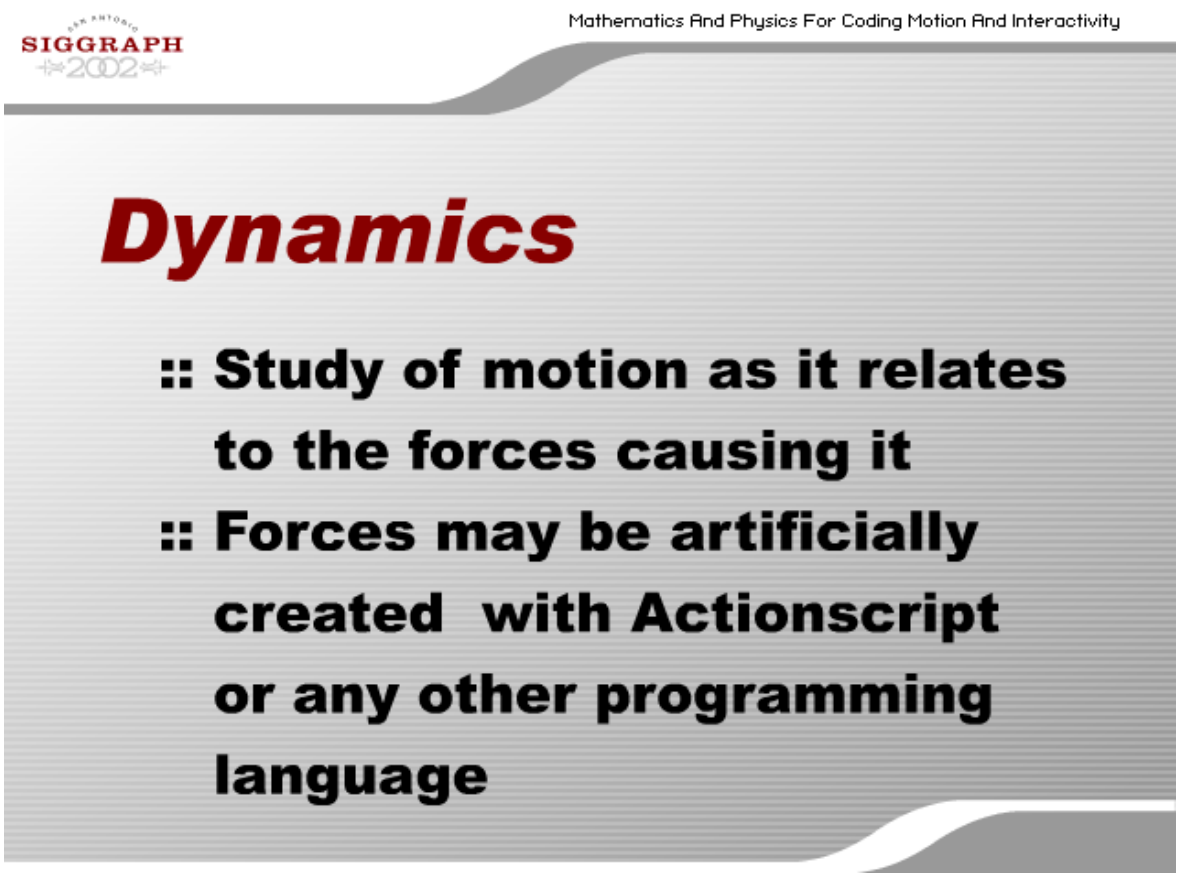

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

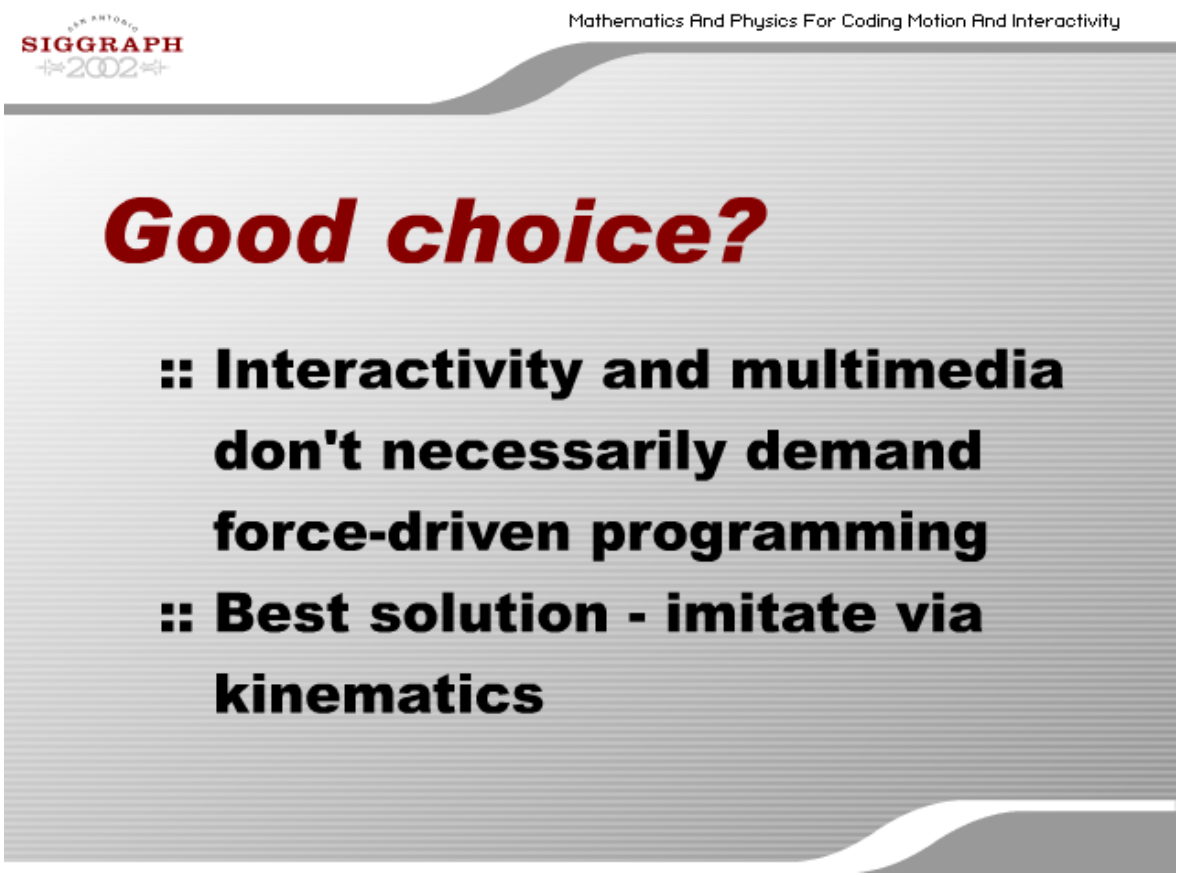

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
APPROACHING
KINEMATICS
Gestalt and Newton

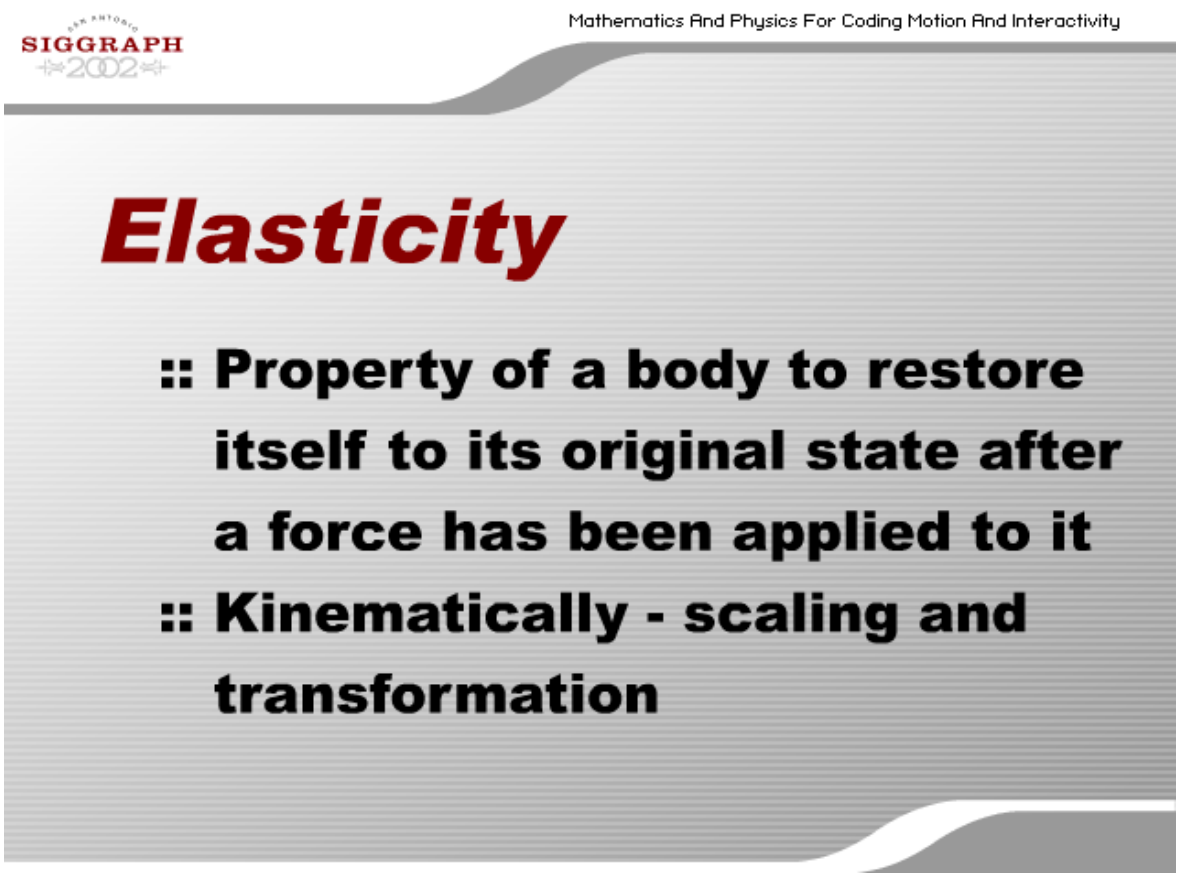

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Gestalt Psychology
:: Parts make a whole
:: Understand the individual parts to form the whole

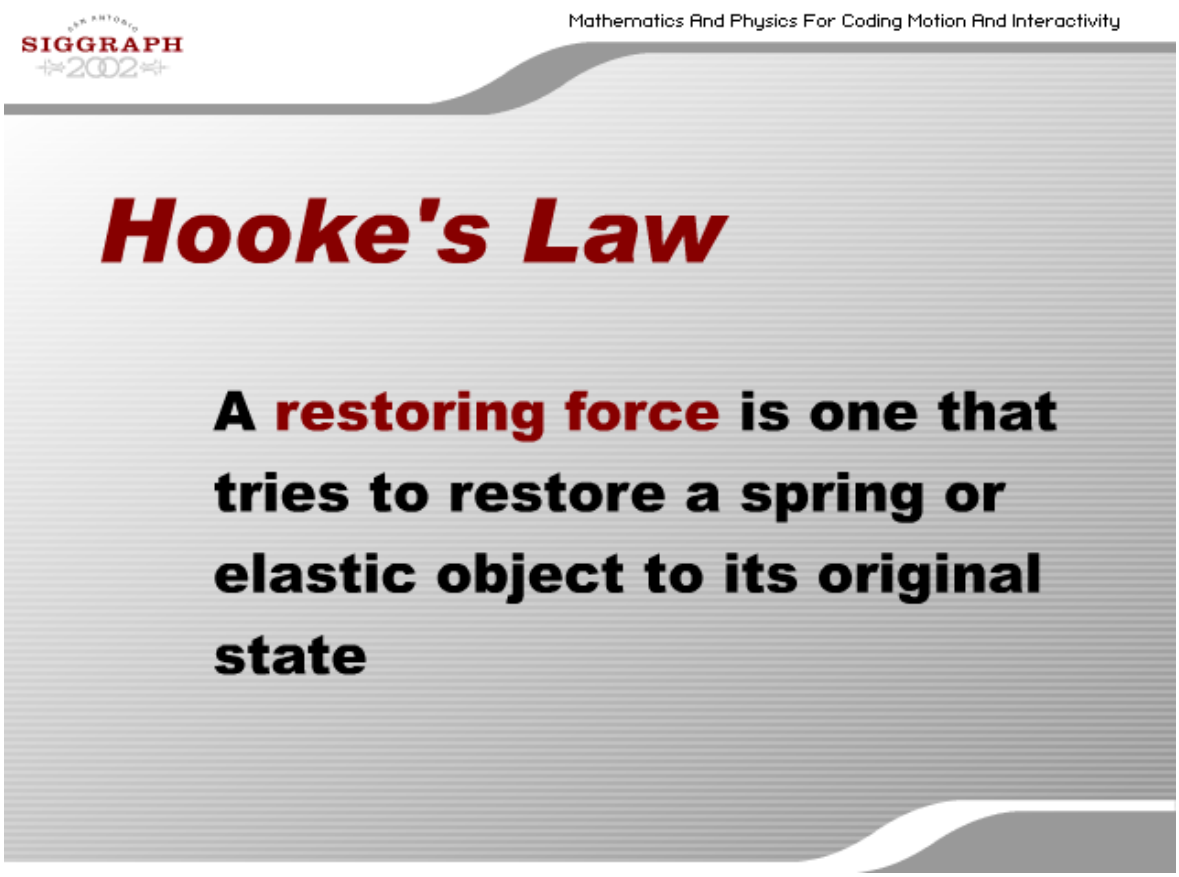

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Newton
:: Equations of motion
:: Understand underlying concepts rather than memorize

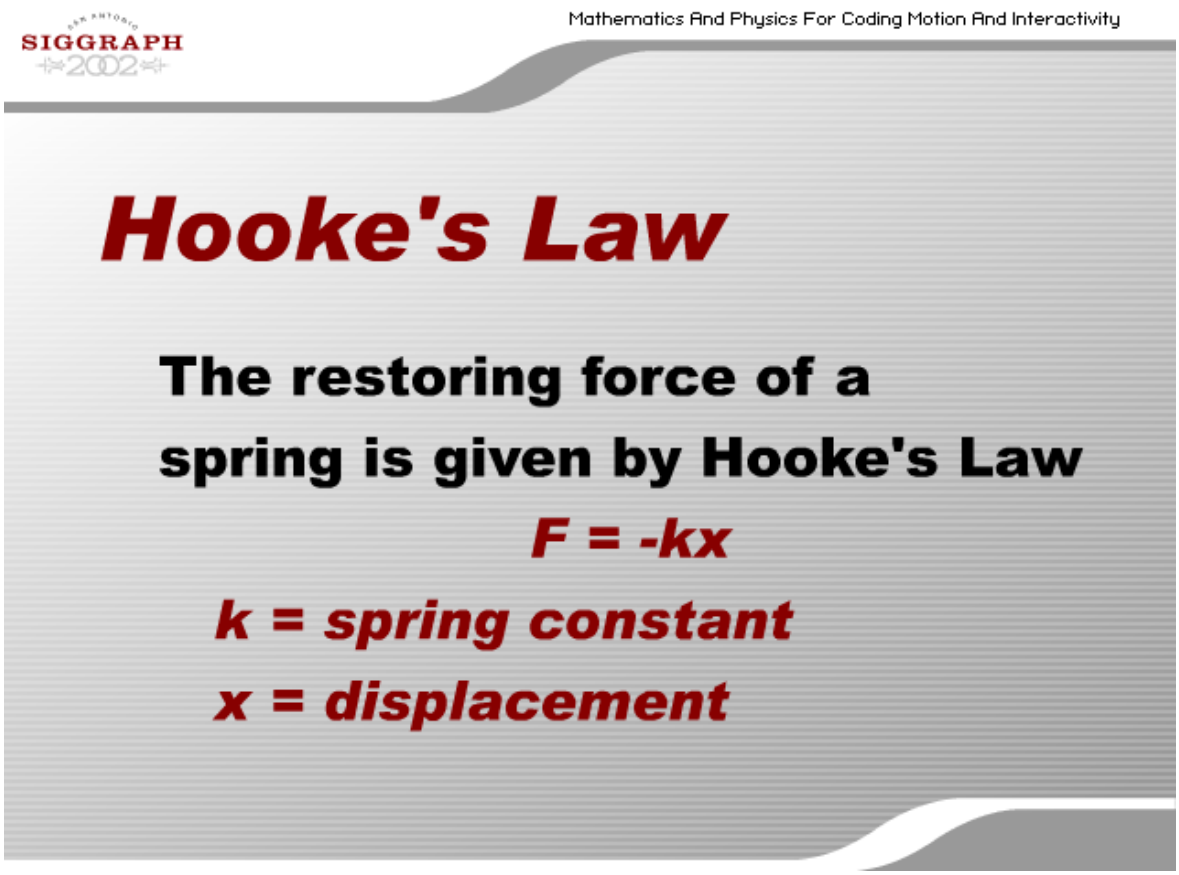

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
PSEUDO
DYNAMICS
Elasticity and Springs

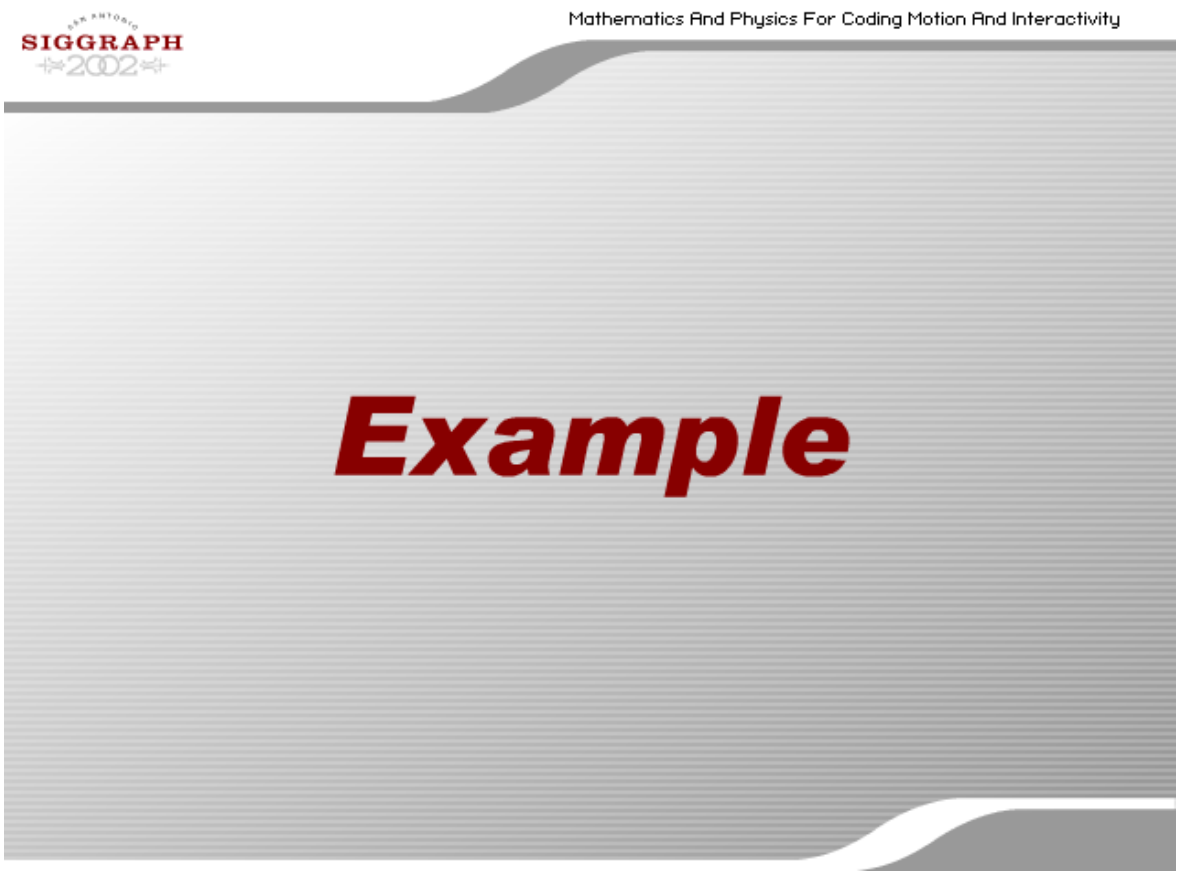

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Dynamics
:: Study of motion as it relates to the forces causing it
:: Forces may be artificially created with Actionscript or any other programming language

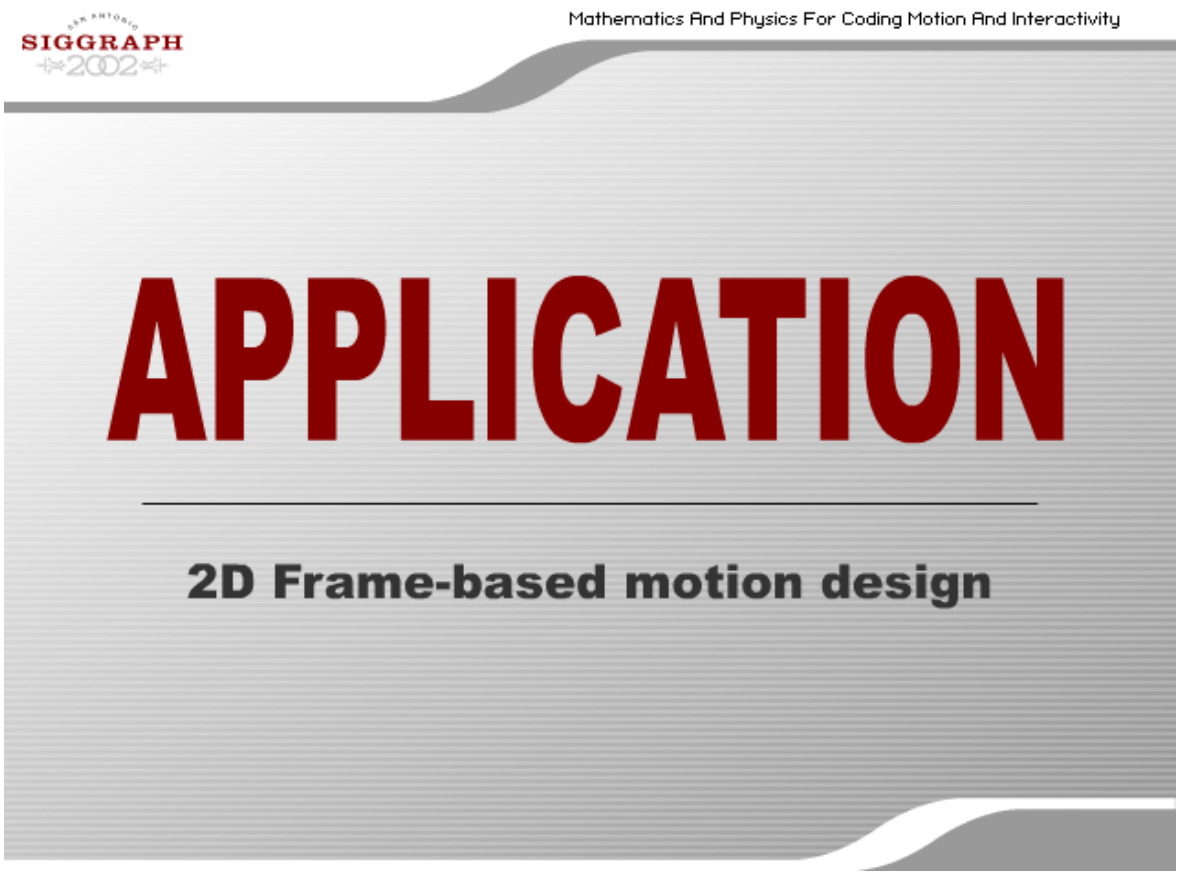

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Good choice?
:: Interactivity and multimedia don't necessarily demand force-driven programming
:: Best solution - imitate via kinematics

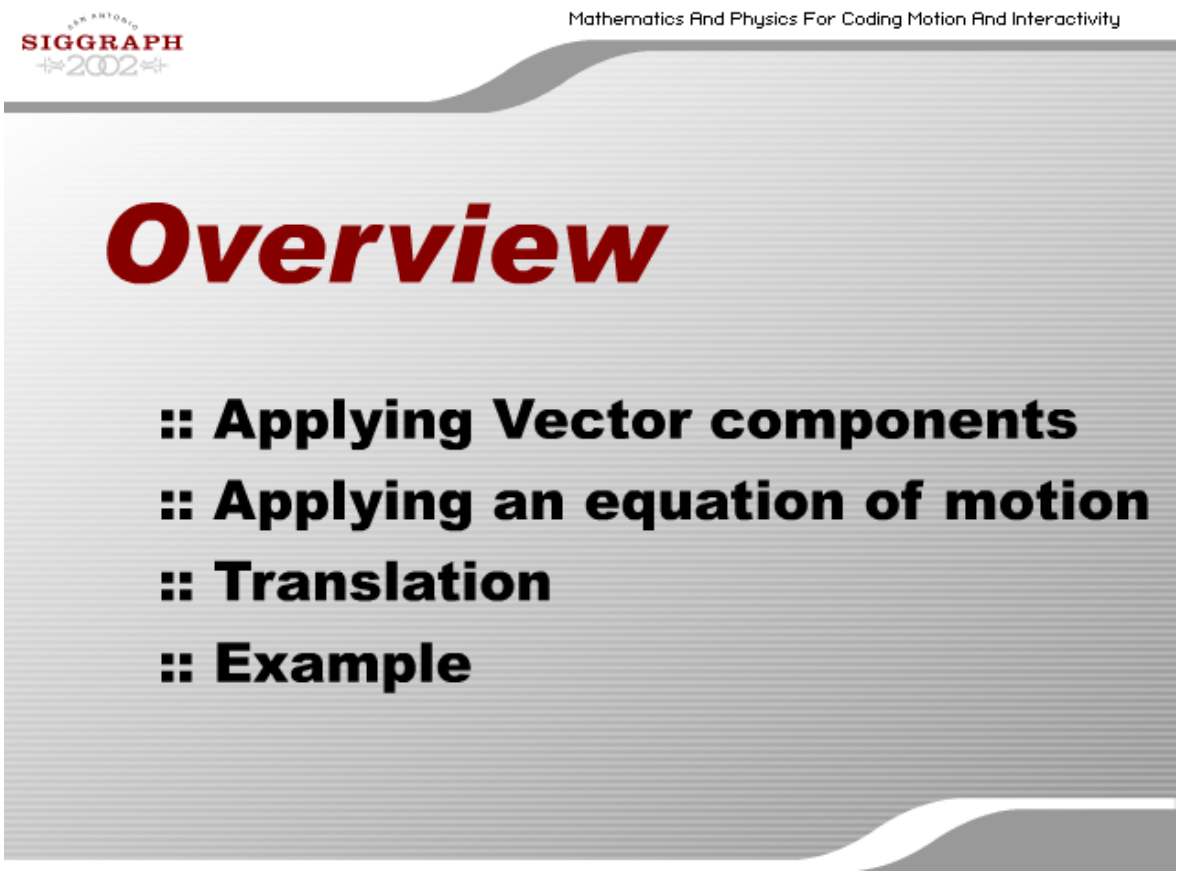

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Elasticity
:: Property of a body to restore itself to its original state after a force has been applied to it
:: Kinematically - scaling and transformation

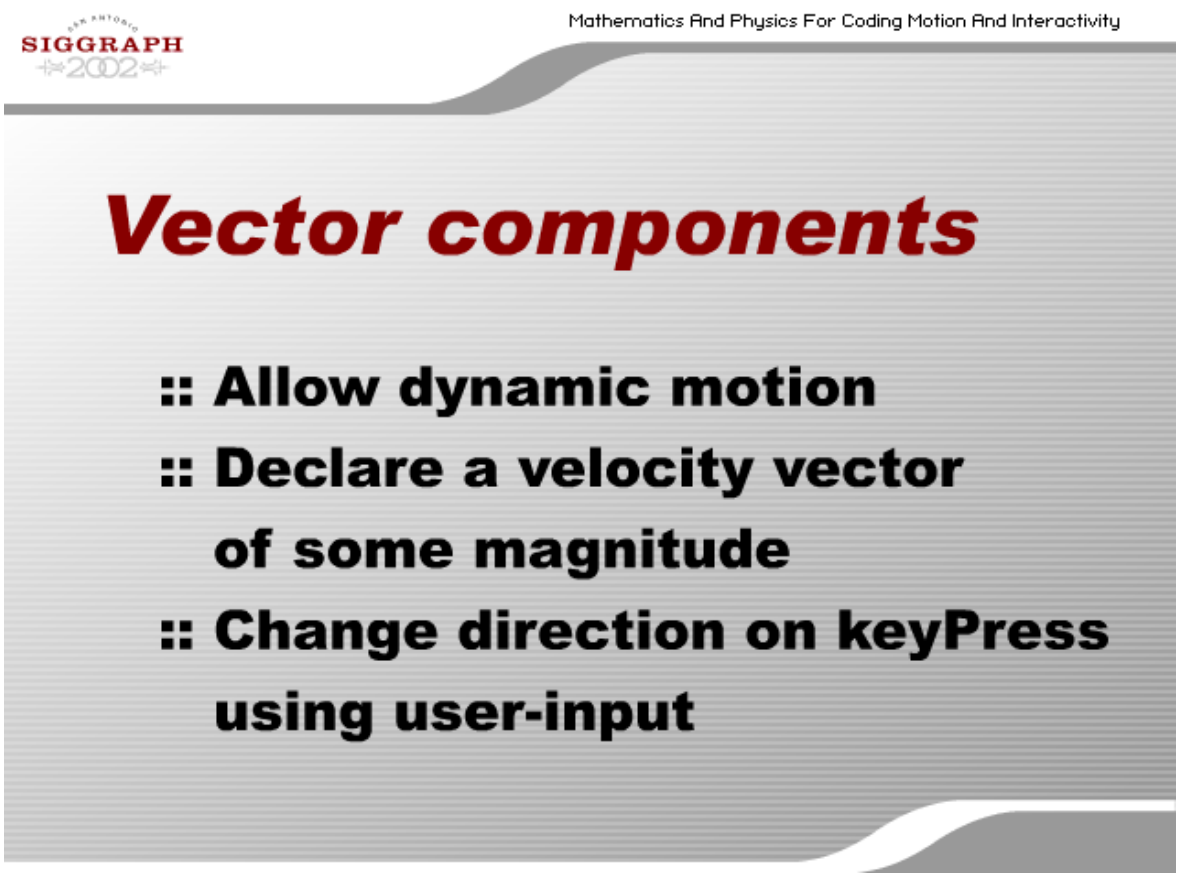

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Hooke's Law
A restoring force is one that tries to restore a spring or elastic object to its original state

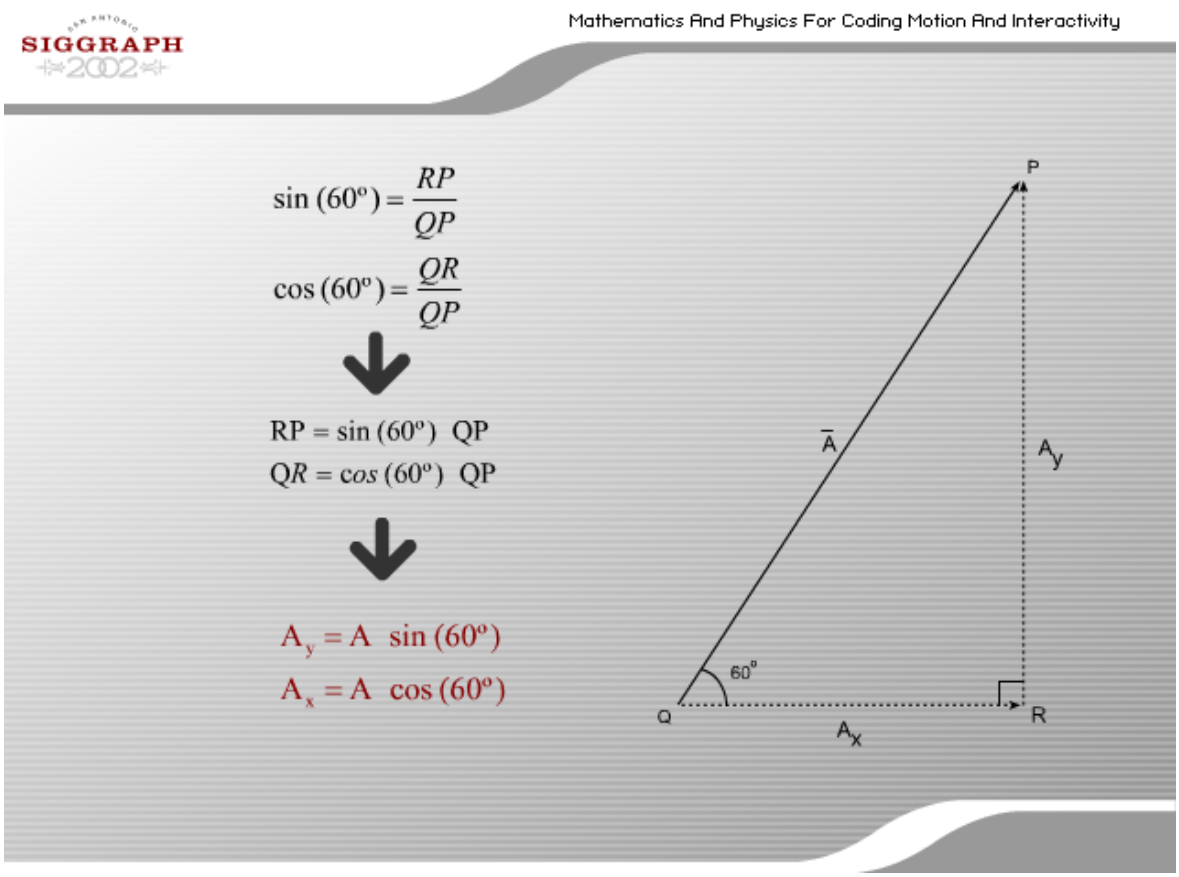

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Hooke's Law
The restoring force of a spring is given by Hooke's Law
F = -kx
k = spring constant
x = displacement

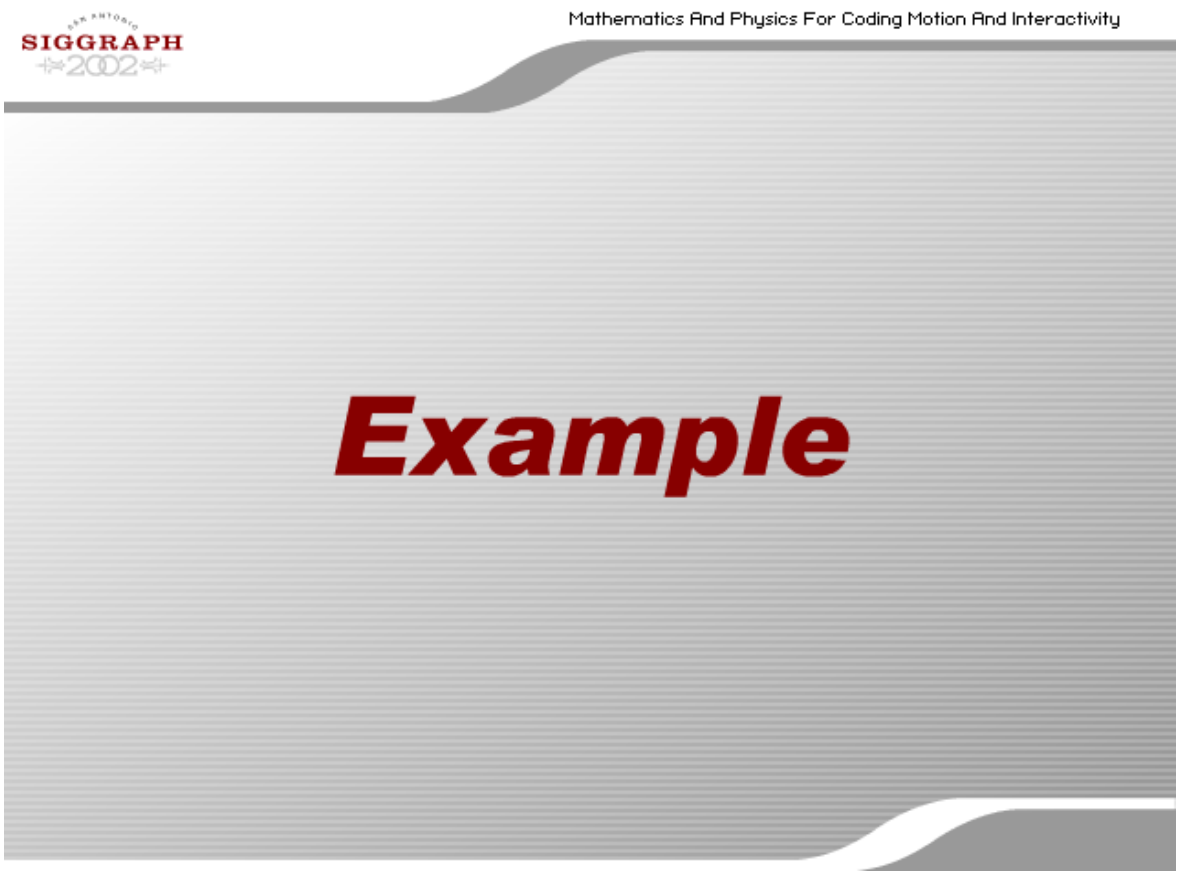

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

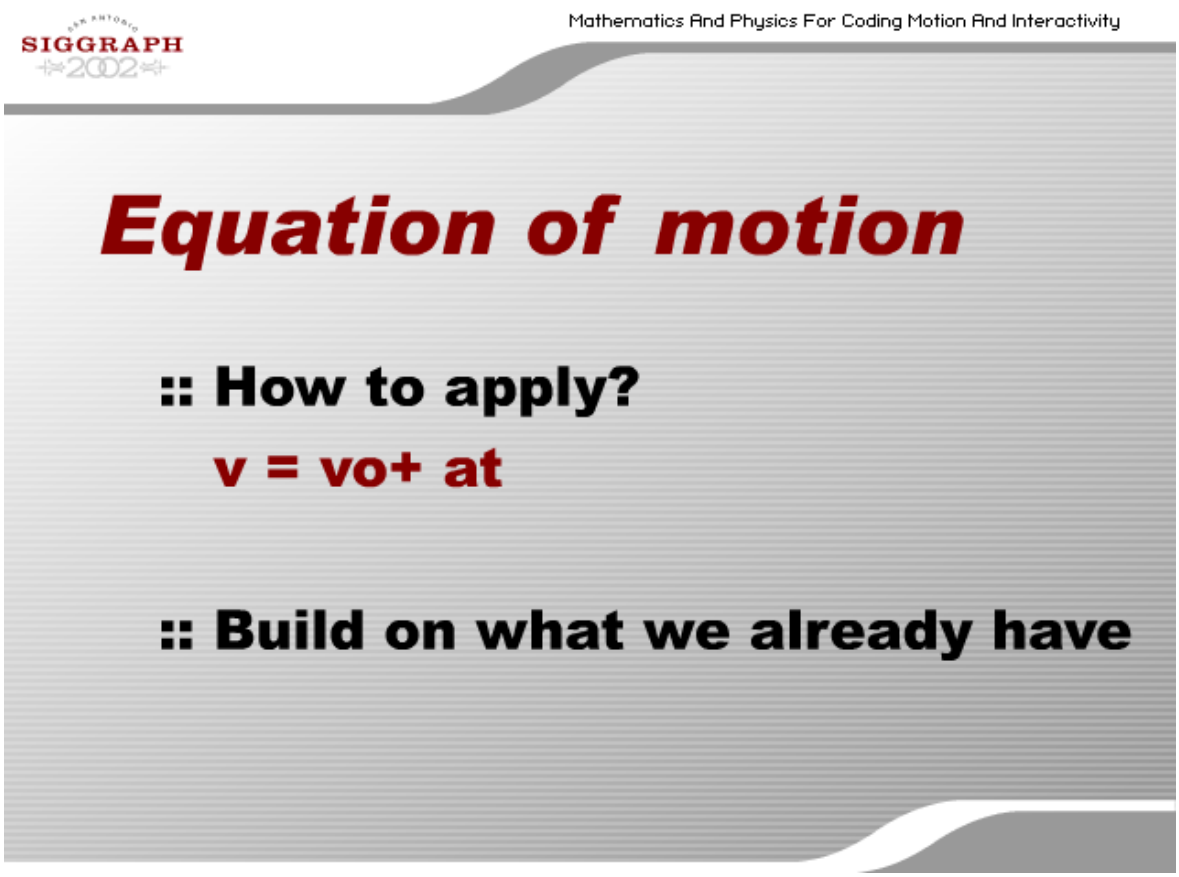

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
APPLICATION
2D Frame-based motion design

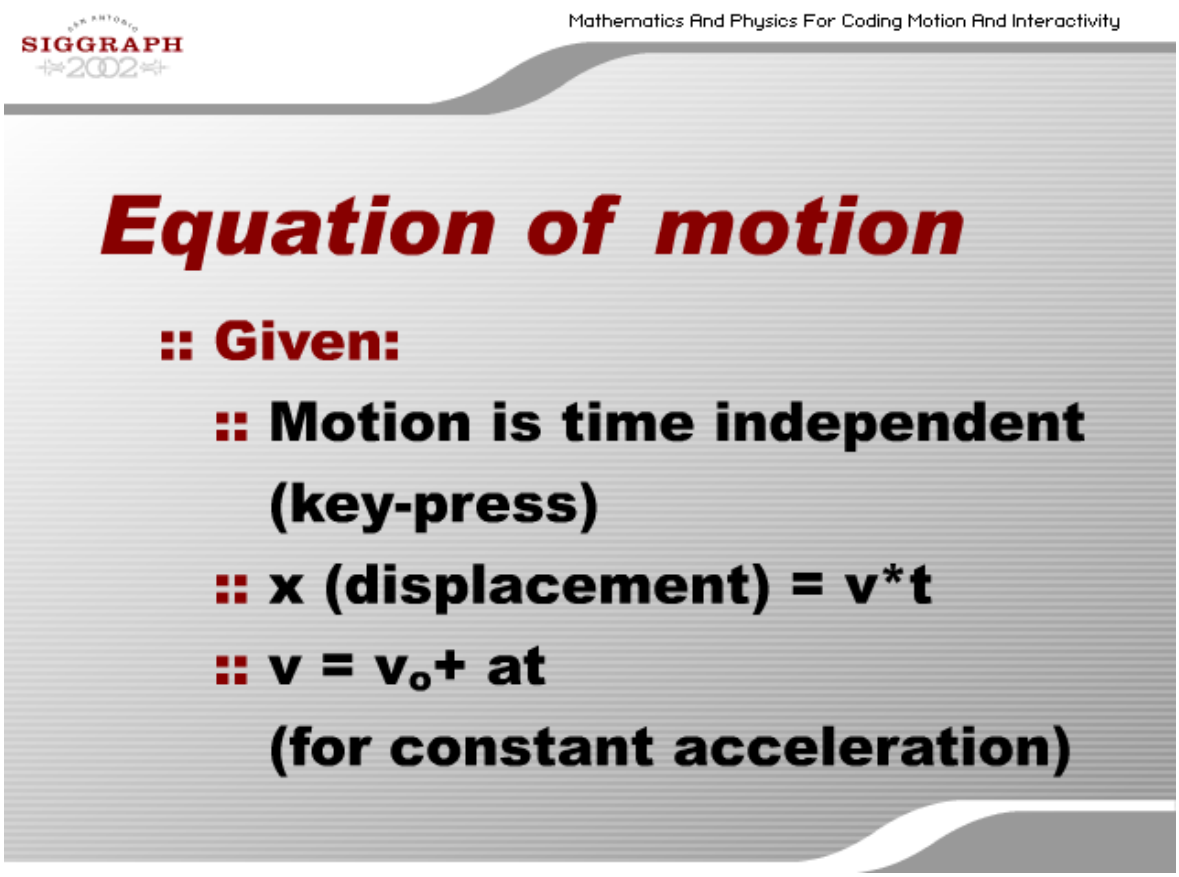

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Overview
:: Applying Vector components
:: Applying an equation of motion
:: Translation
:: Example

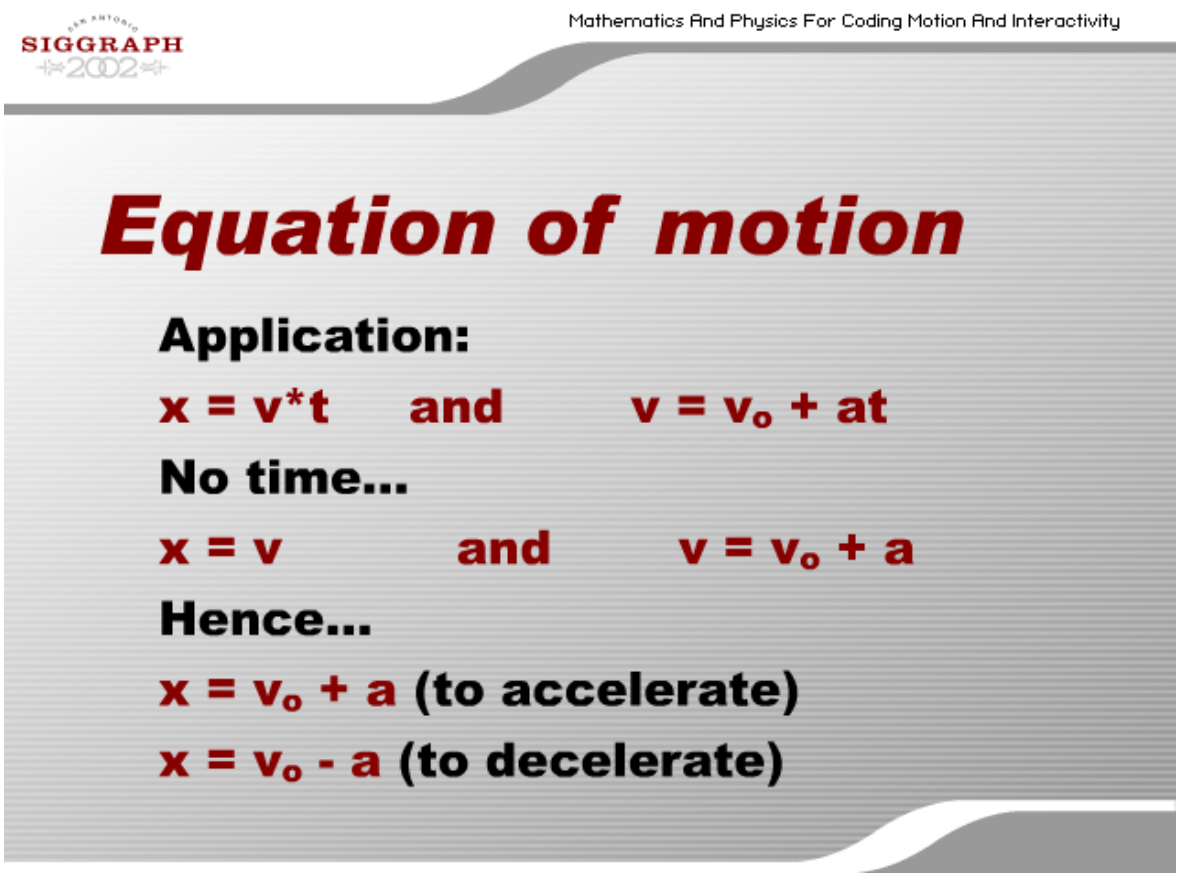

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Vector components
:: Allow dynamic motion
:: Declare a velocity vector of some magnitude
:: Change direction on keyPress using user-input

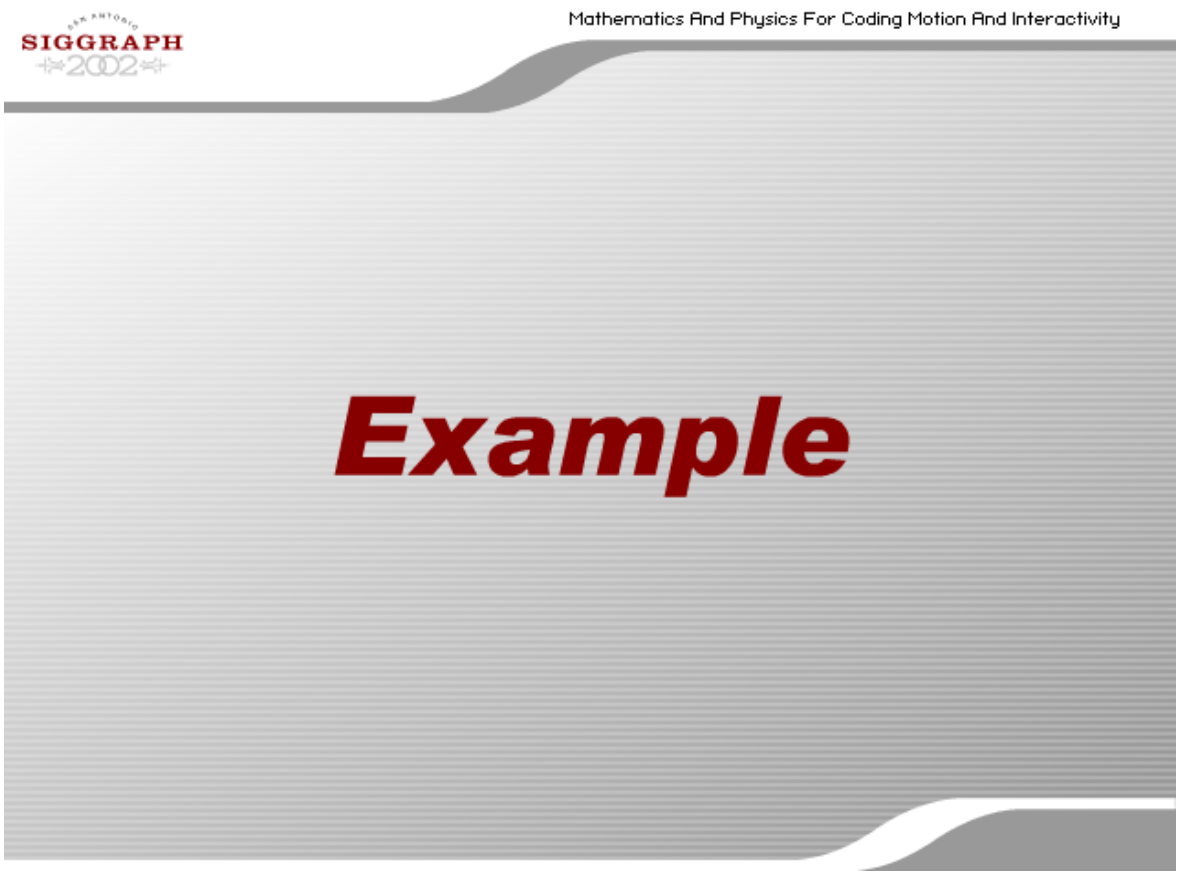

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
$\sin(60^\circ) = \frac{RP}{QP}$
$\cos(60^\circ) = \frac{QR}{QP}$
RP = sin (60°) QP
QR = cos (60°) QP
$A_y = A \sin(60^\circ)$
$A_x = A \cos(60^\circ)$
P
$\bar{A}$
A_y
60°
Q
A_X
R

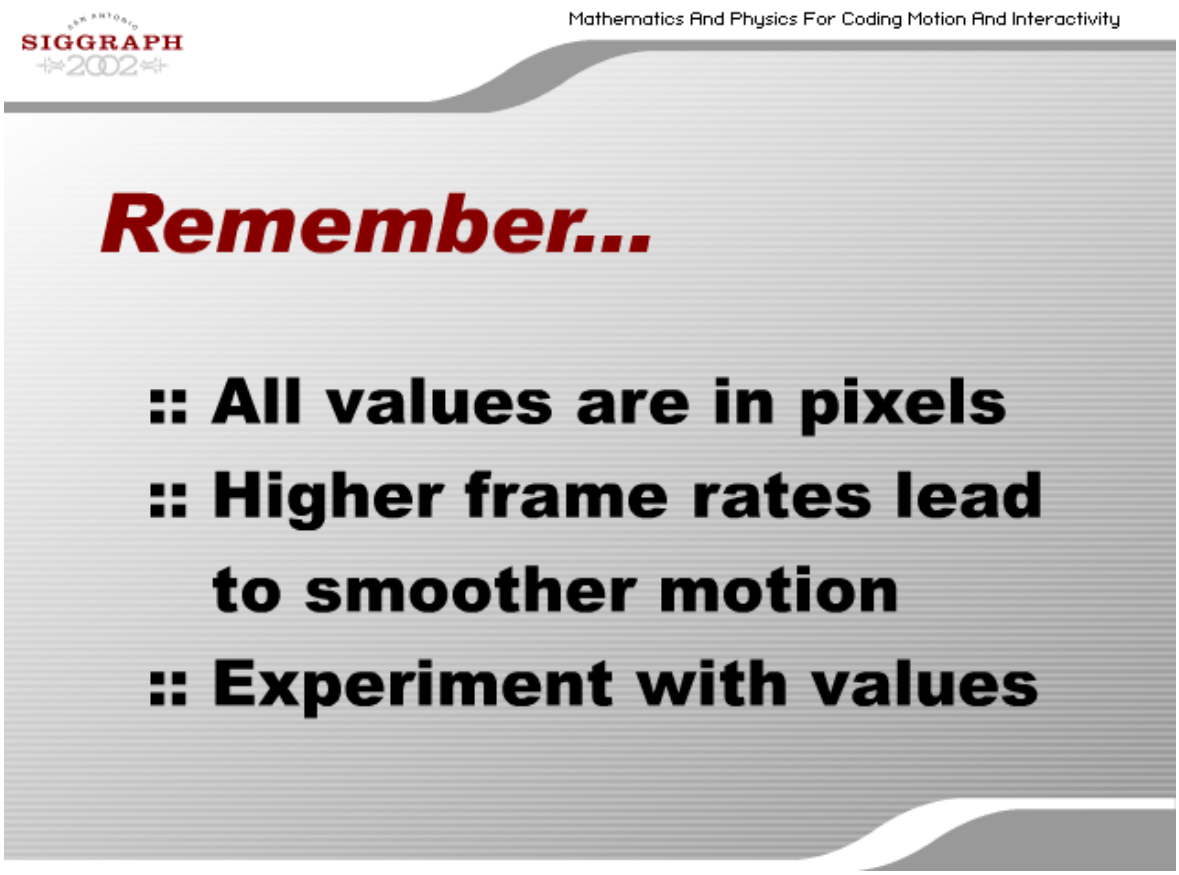

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

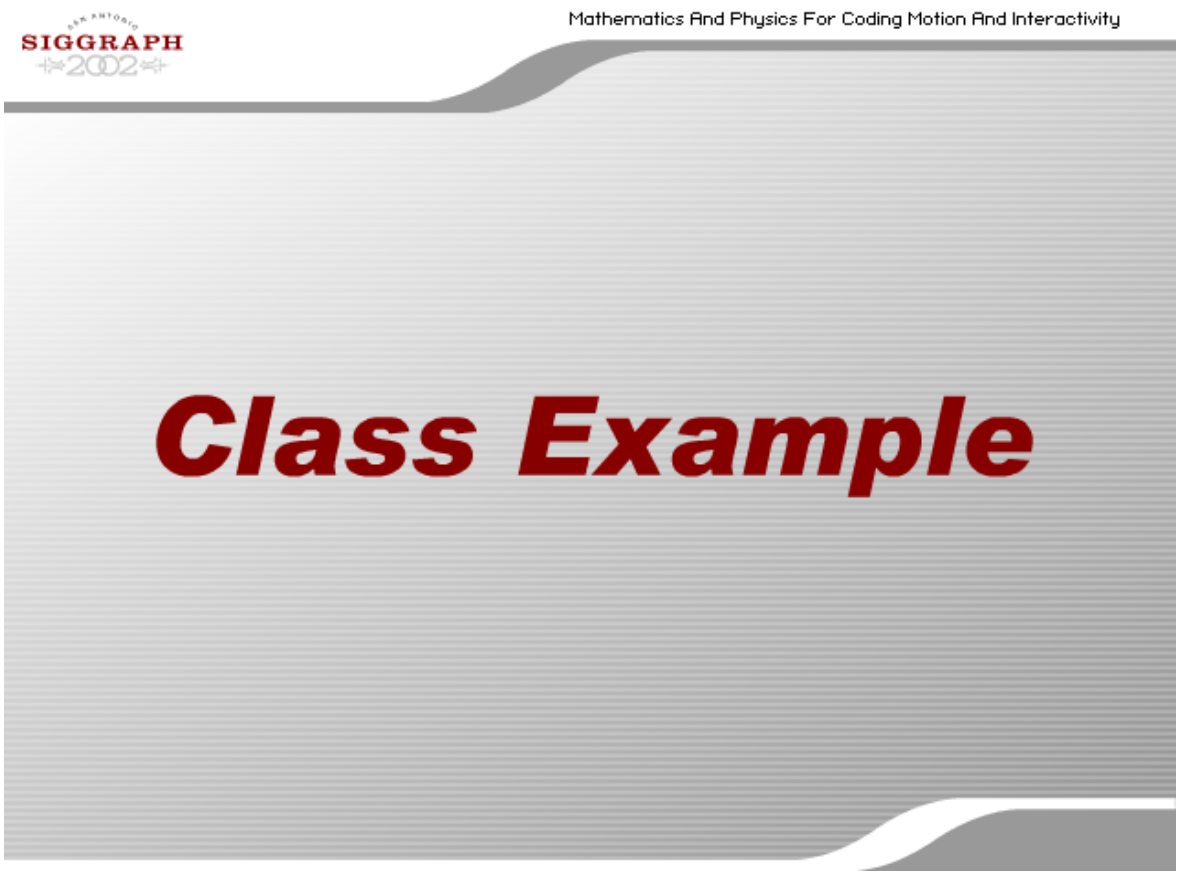

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Equation of motion
:: How to apply?
v = vo+ at
:: Build on what we already have

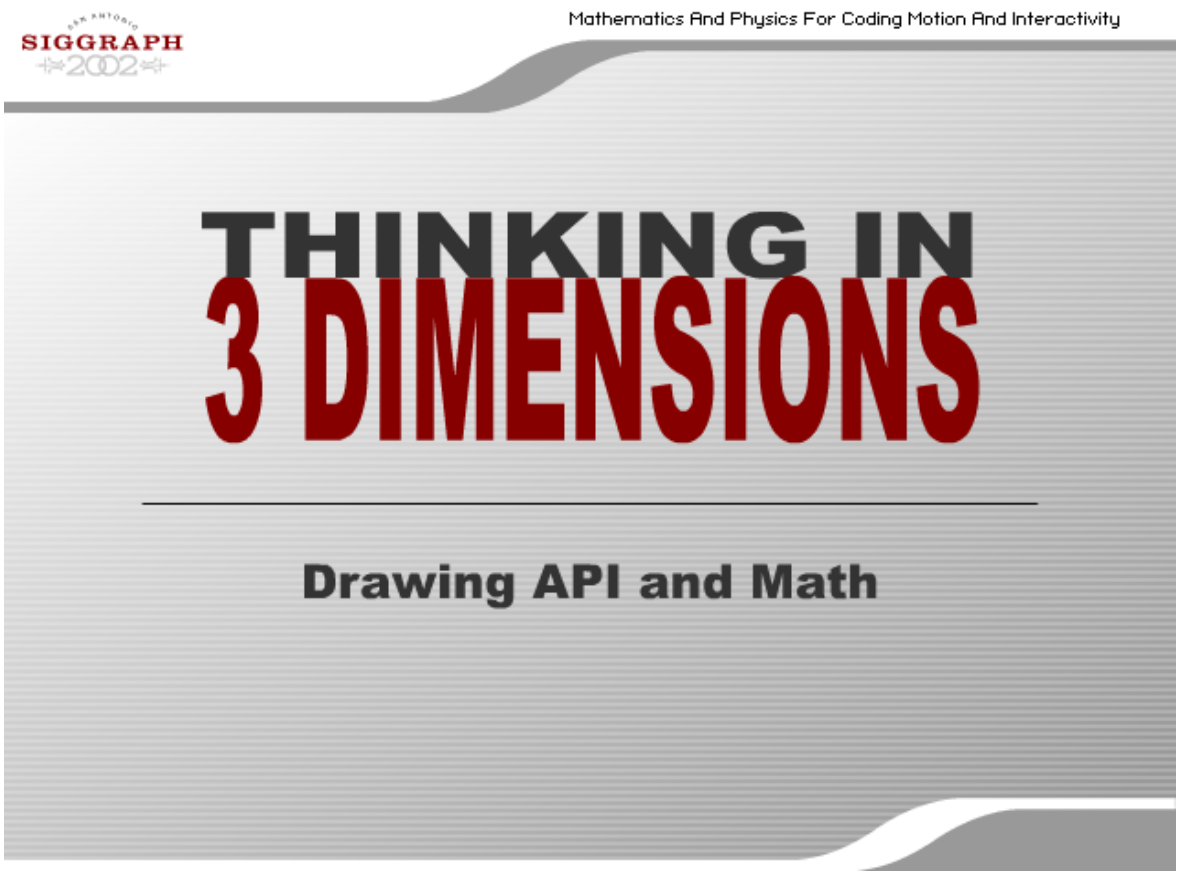

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Equation of motion
:: Given:
:: Motion is time independent
(key-press)
:: x (displacement) = v*t
:: v = v$_o$+ at
(for constant acceleration)

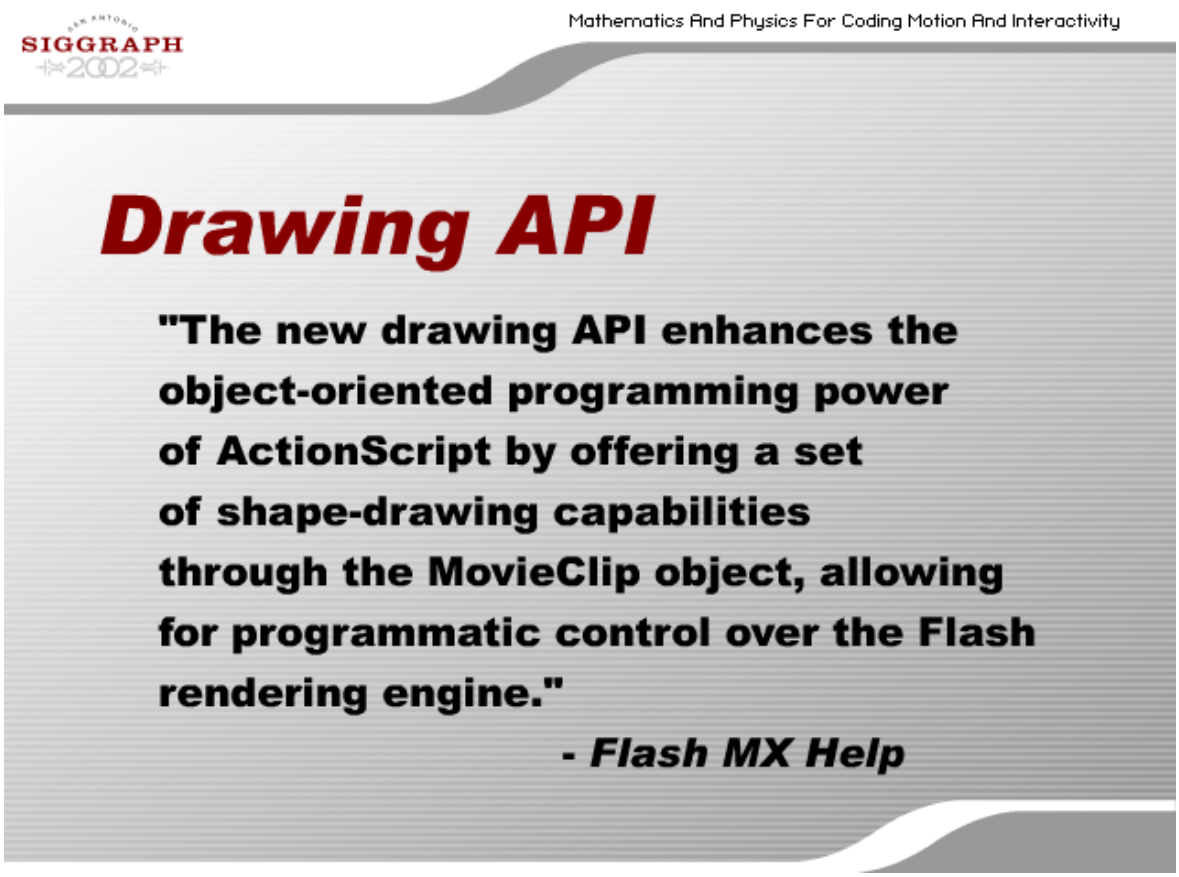

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Equation of motion
Application:
x = v*t and v = v0 + at
No time...
x = v and v = v0 + a
Hence...
x = v0 + a (to accelerate)
x = v0 - a (to decelerate)

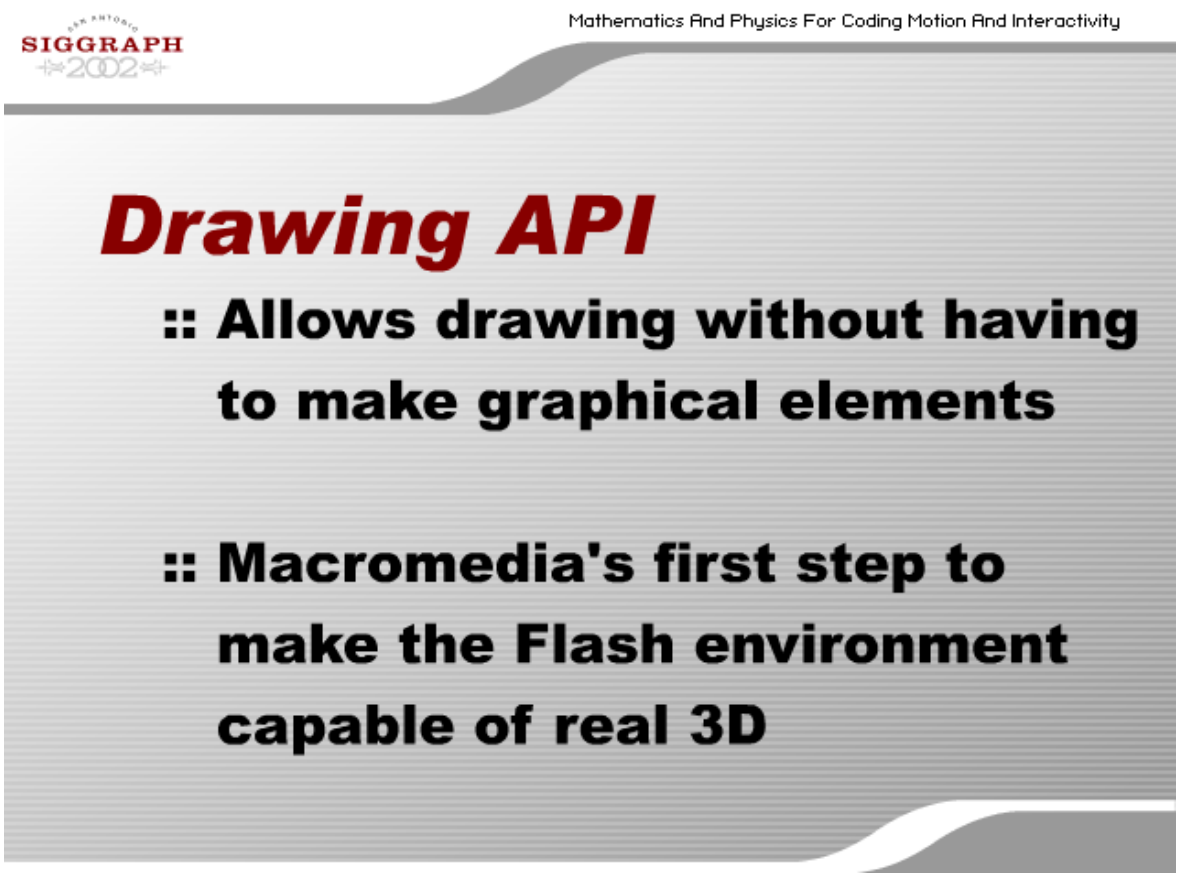

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Example

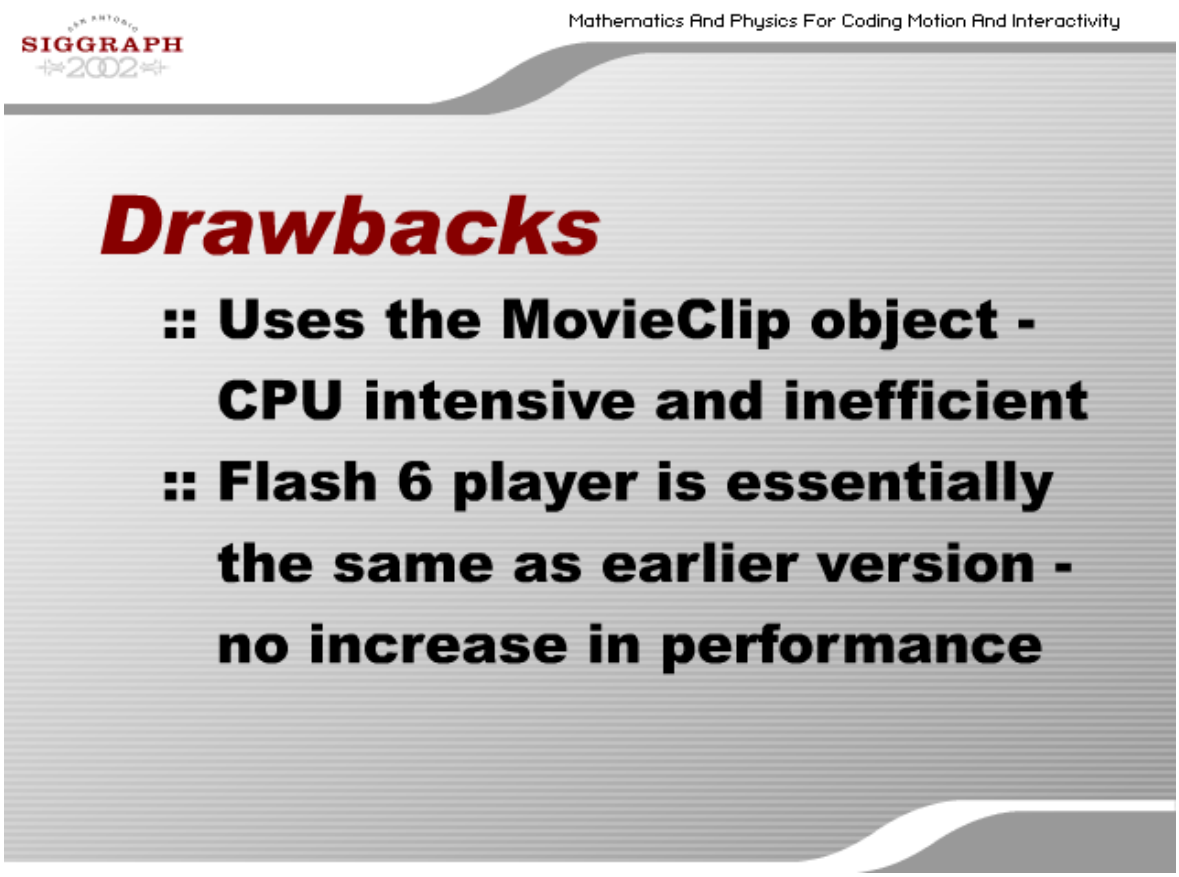

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Remember...
:: All values are in pixels
:: Higher frame rates lead to smoother motion
:: Experiment with values


SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Class Example

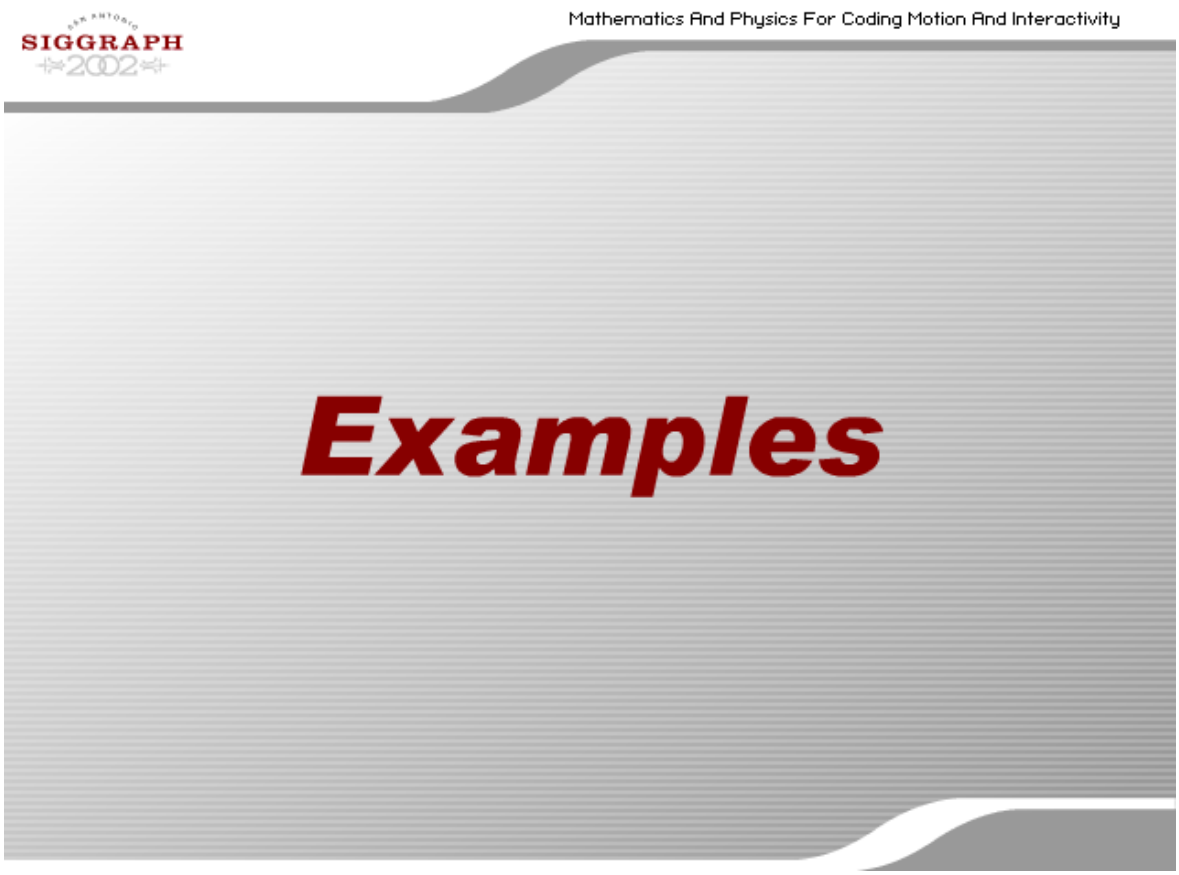

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
THINKING IN
3 DIMENSIONS
Drawing API and Math

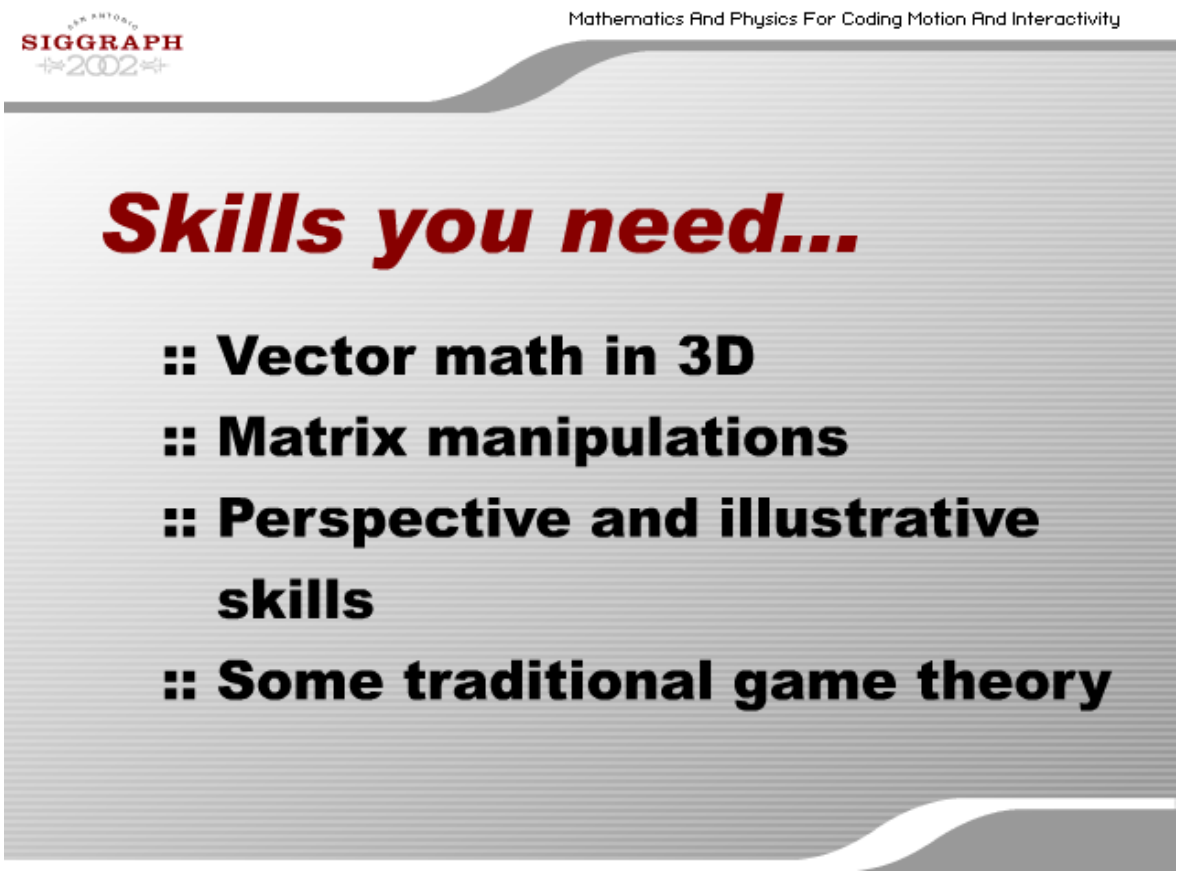

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Drawing API
"The new drawing API enhances the object-oriented programming power of ActionScript by offering a set of shape-drawing capabilities through the MovieClip object, allowing for programmatic control over the Flash rendering engine."
- Flash MX Help


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Drawing API
:: Allows drawing without having to make graphical elements
:: Macromedia's first step to make the Flash environment capable of real 3D


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Drawbacks
:: Uses the MovieClip object - CPU intensive and inefficient
:: Flash 6 player is essentially the same as earlier version - no increase in performance


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Possibilities
:: Simple 3D games
:: Interesting and complex programmatic graphics
:: Influx of programmers into Flash developement arena

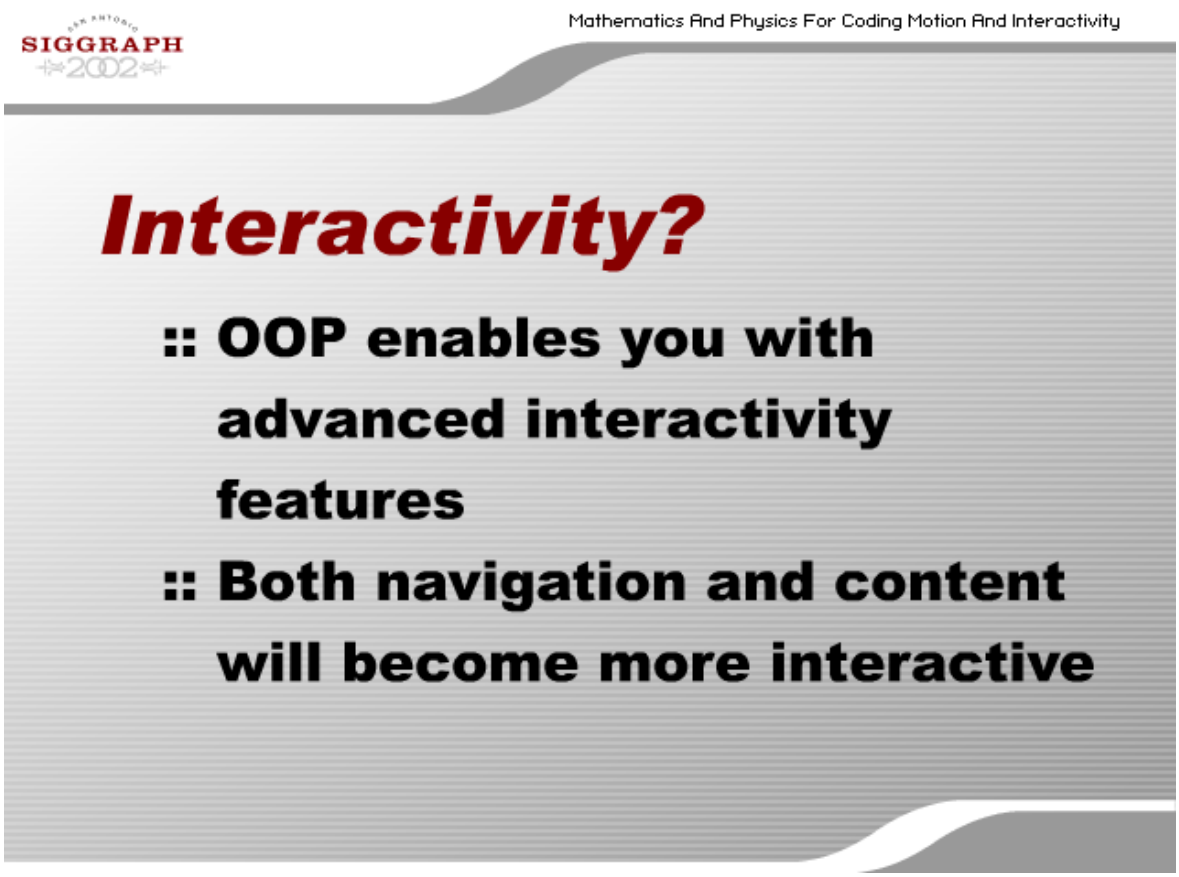

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
Examples

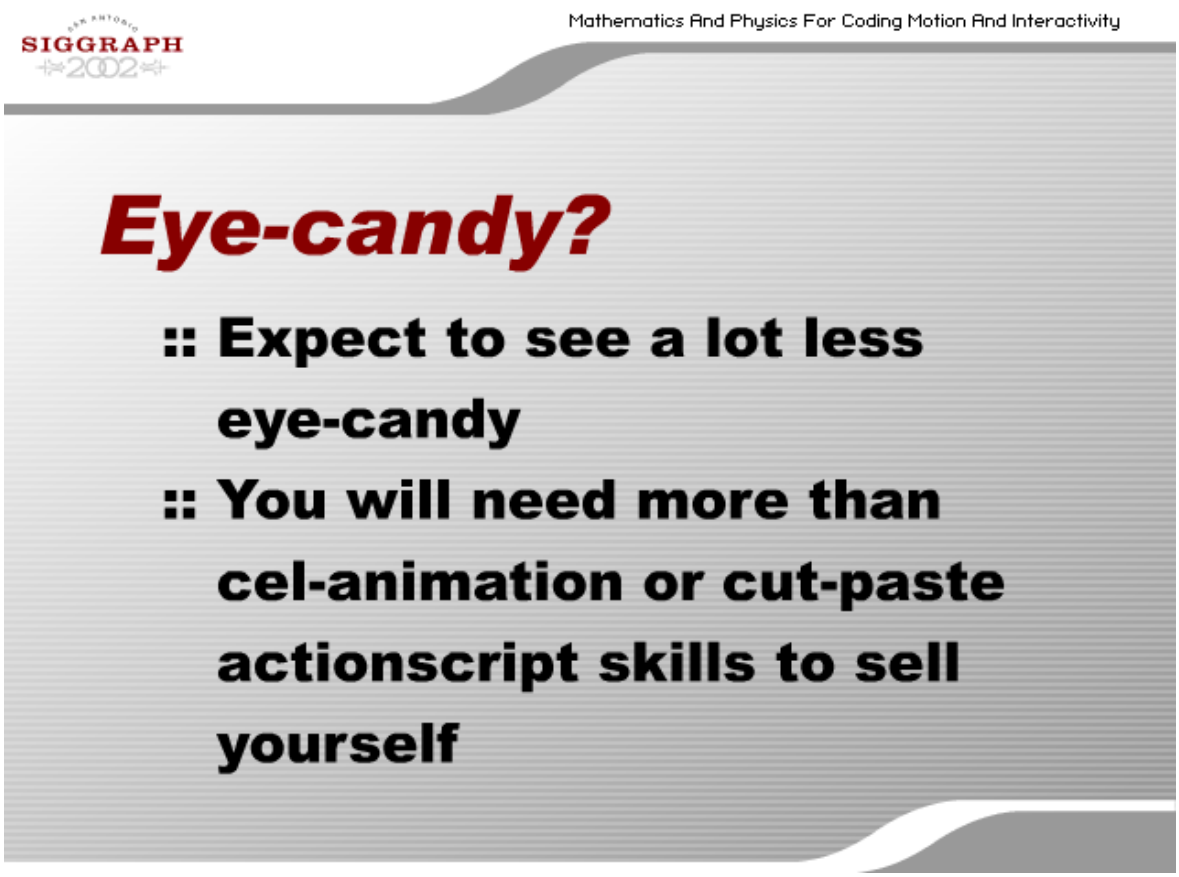

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Skills you need...
:: Vector math in 3D
:: Matrix manipulations
:: Perspective and illustrative skills
:: Some traditional game theory

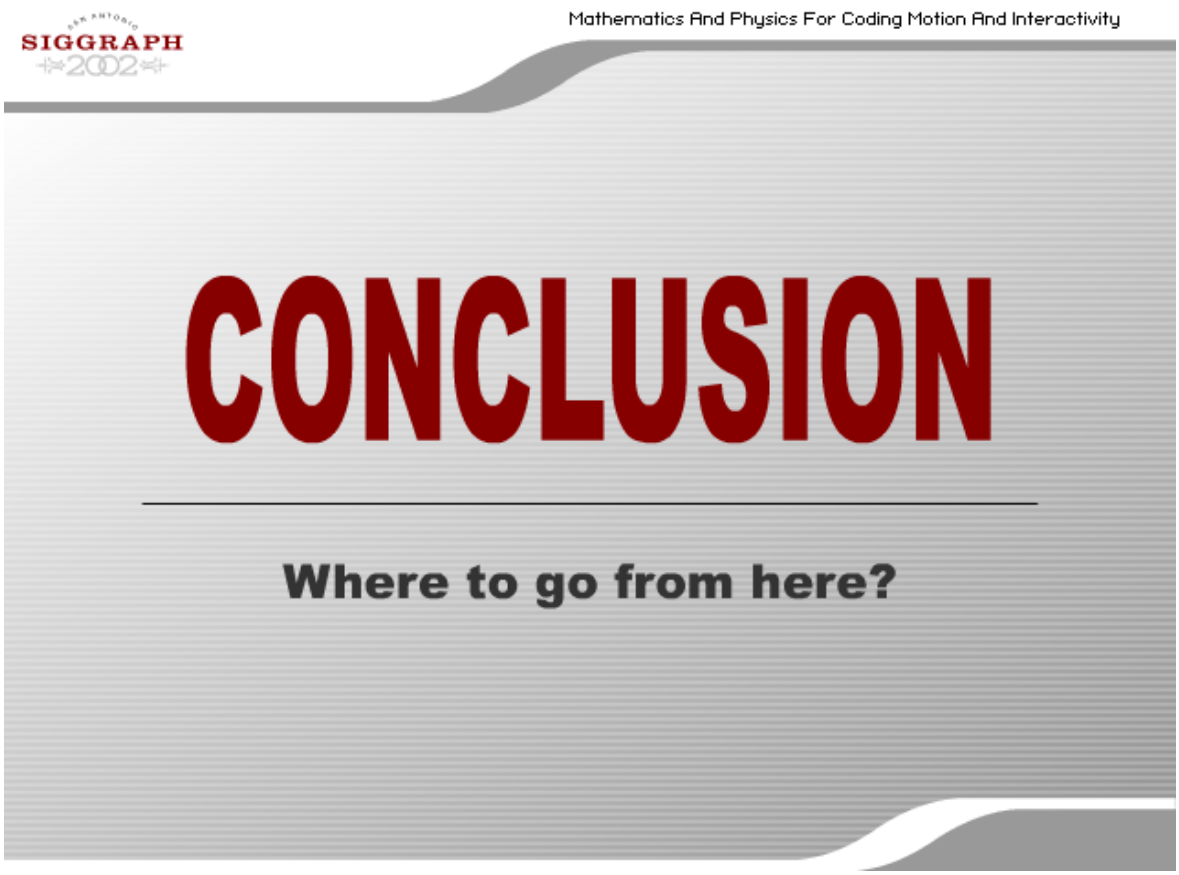

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
NEAR
FUTURE
What to expect

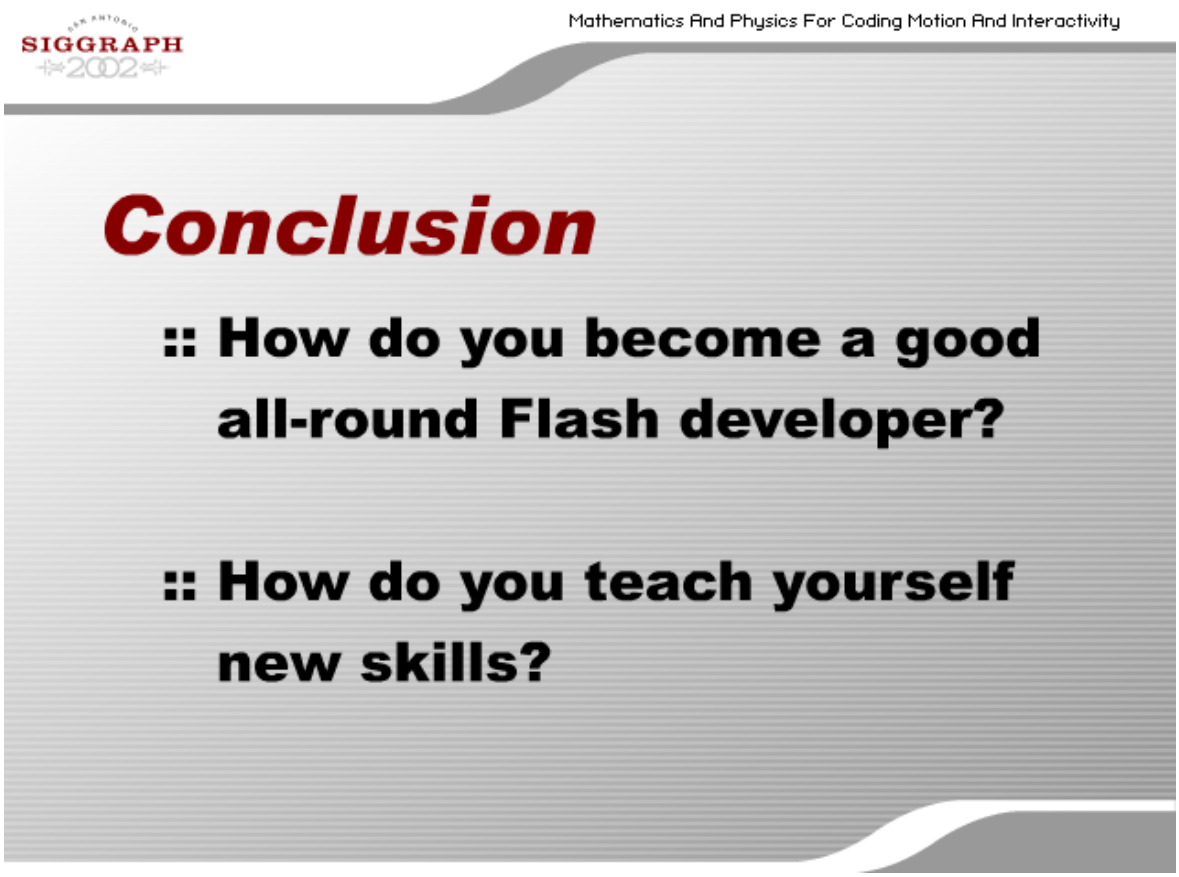

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Where Flash is going
:: Macromedia is aiming to make Flash an application development tool for the web
:: Macromedia also wants to attract the Java developers

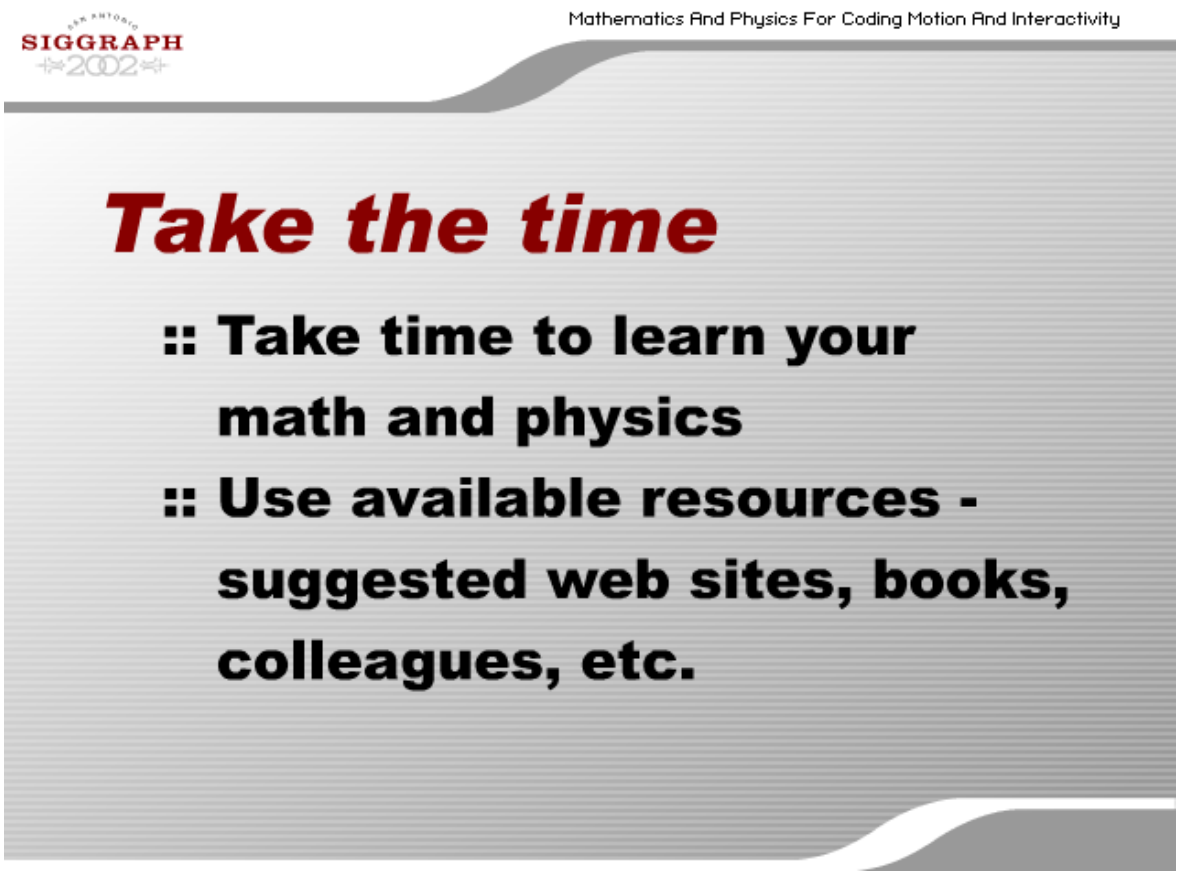

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Where Flash is going
:: Expect to see very complex applications
:: Expect to see Inverse Kinematics
:: Expect to see basic first person games

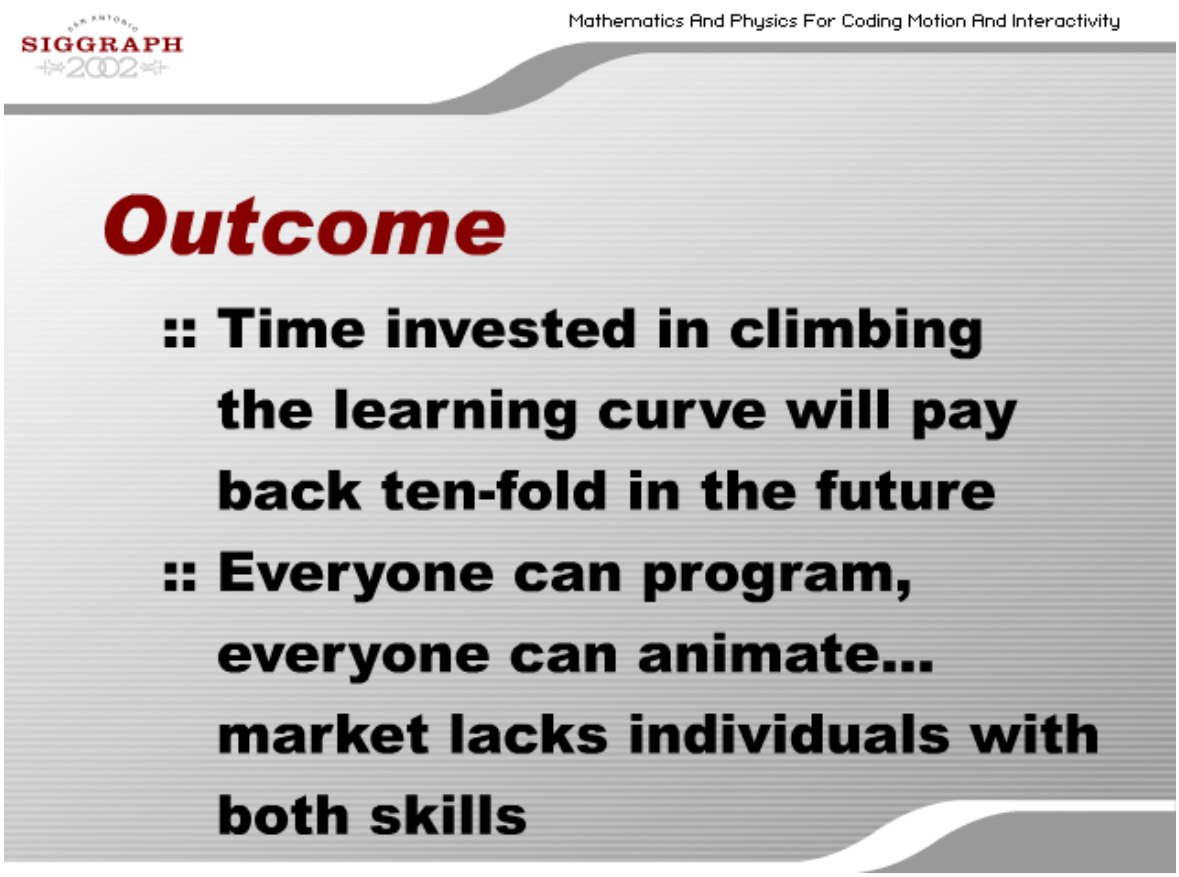

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Interactivity?
:: OOP enables you with advanced interactivity features
:: Both navigation and content will become more interactive

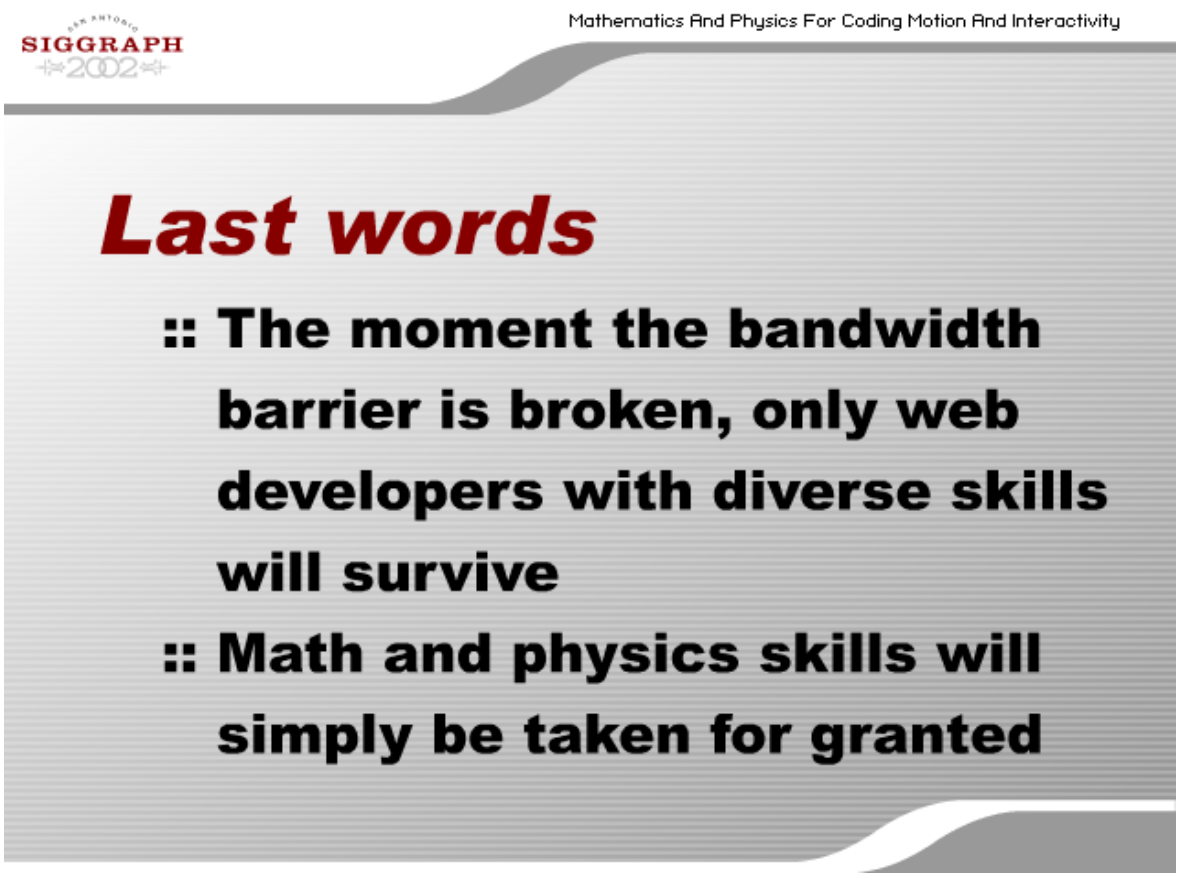

SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Eye-candy?
:: Expect to see a lot less eye-candy
:: You will need more than cel-animation or cut-paste actionscript skills to sell yourself

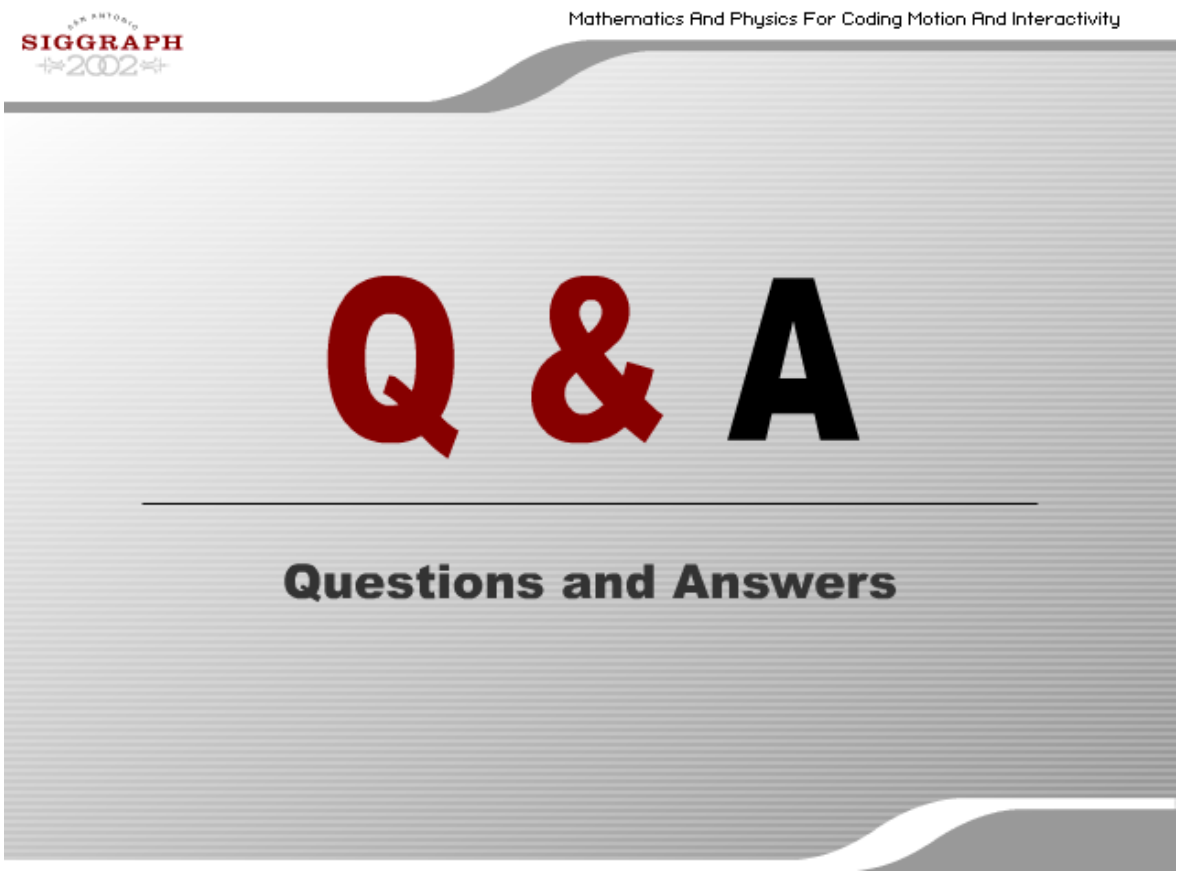

SIGGRAPH
2002
Mathematics And Physics For Coding Motion And Interactivity
CONCLUSION
Where to go from here?


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Conclusion
:: How do you become a good all-round Flash developer?
:: How do you teach yourself new skills?


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Take the time
:: Take time to learn your math and physics
:: Use available resources - suggested web sites, books, colleagues, etc.


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Outcome
:: Time invested in climbing the learning curve will pay back ten-fold in the future
:: Everyone can program, everyone can animate... market lacks individuals with both skills


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Last words
:: The moment the bandwidth barrier is broken, only web developers with diverse skills will survive
:: Math and physics skills will simply be taken for granted


SIGGRAPH 2002
Mathematics And Physics For Coding Motion And Interactivity
Q & A
Questions and Answers

Supplementary Materials

MATHEMATICS

Introduction

For some reason, the mention of the word “math” intimidates most people. It’s much like the phenomenon of *Classical Conditioning*, a technique first developed by Ivan Pavlov, a prominent Russian psychologist back in the day. For the benefit of those who have never been exposed to psychology, let us elaborate on Pavlov’s interesting work.

Pavlov was very interested in digestive processes, namely how dogs salivated in the presence of different foods. People salivate when food is placed in their mouths, and the same is true for dogs. Pavlov's studies lead him to formulate his theories of Classical Conditioning. According to Pavlov, the *unconditioned response* of drooling was a direct result of the *unconditioned stimulus*, namely, the placement of food in the mouth. Unconditioned responses are not learned responses, but governed by reflex actions. However, Pavlov noticed that dogs often begin to drool before the food is actually placed in their mouths, merely by the sight or smell of it, or even hearing the footsteps of the bearer. This led him to coin the term *Conditioned Response*. The dogs learnt that if they saw the food, smelled it, or even heard somebody’s footsteps approaching with the food dish, they were probably going to be fed. Hence, they began to salivate in anticipation on a meal. This response is what Pavlov called *Conditioned Response*, as it was learnt over a span of time.

Seeing this, Pavlov hypothesized that such associations could easily be manipulated. His studies show that the process of learning is based on associations. Undesirable stimuli typically cause negative responses. For example, if math was always presented to you in an uninteresting manner and you were forced to complete difficult homework problems all through school, then it would be natural for you to have developed a bad taste for mathematics. In its natural form however, mathematics is much like the anticipated meal fed to Pavlov’s dog – desirable, tasty, satisfying and last but not the least, necessary. We have been conditioned into believing that math is not fun and hopefully, this course should at least recondition you to reverse some potentially bad memories.

This section deals with basic math concepts that you MUST know. They will be invaluable to you if you decided to take this course a step further by taking some traditional game design and theory courses, or simply plan to indulge in some advanced programming. This section covers the bare essentials. It does not offer you programmatic solutions to problems. We thought best to separate theory from application, for theory is forever, whereas application is forever changing.

2D Space

This is a topic that needs to be addressed for the benefit of those who haven't revisited math since high school.

Cartesian Coordinate System

To define an object in a plane, the 2D Cartesian coordinate system is used. This consists of 2 perpendicular axes that intersect at the origin. Every point in the plane is assigned an ordered pair (x, y) to determine its location as shown in figure 1.1.

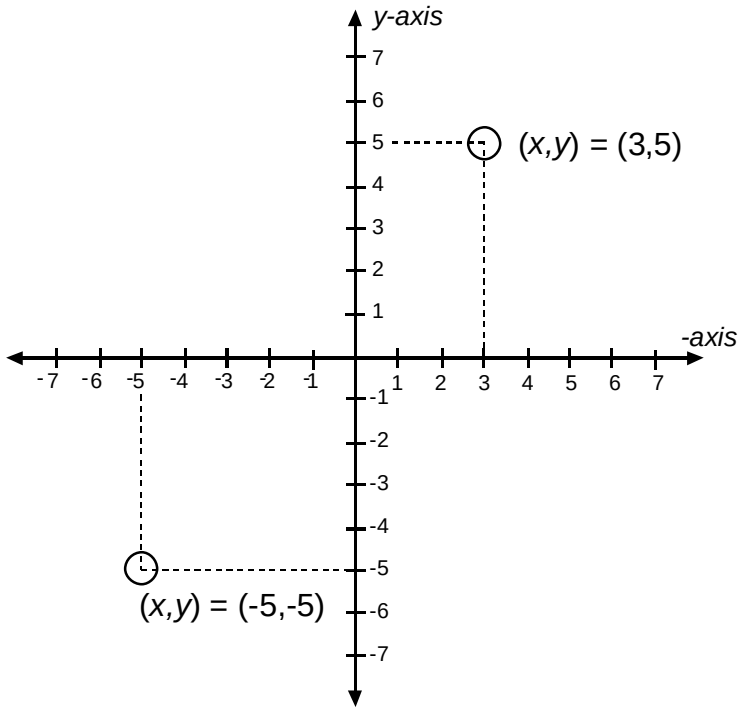


Fig. 1.1 The Cartesian Coordinate System

In fig. 1.1, note that:

x = perpendicular distance of object from y-axis
y = perpendicular distance of object from x-axis

Coordinate system used in Flash and similar products

Now, you're probably wondering where the origin is located on the main stage in your Flash movie. The top left corner of the main stage is the origin (as shown in fig. 1.2). This means:

- *The top edge of the stage is the x-axis*
- *The left edge of the stage is the y-axis*

Hence, we can extrapolate the following:

- A positive x coordinate refers to a point on the **right** of the y-axis (similar to the Cartesian system.)

- A positive y coordinate refers to a point **below** the x-axis (*contrary* to what the Cartesian coordinate system follows.)

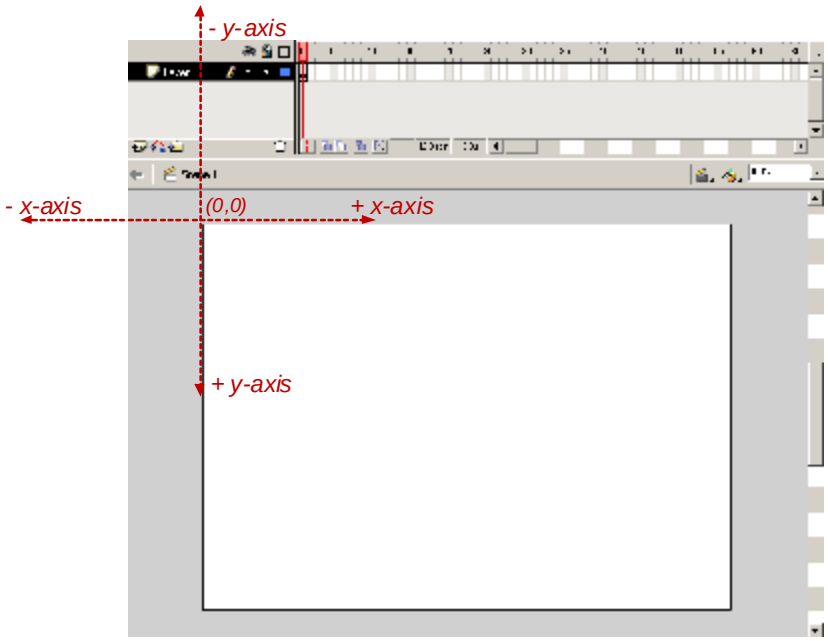


Figure 1.2 Flash's coordinate system.

You can still use the Cartesian coordinate system to calculate the position of your object, etc. by flipping the y-axis.

Pythagorean Theorem

Pythagoras was probably one of the most influential mathematicians that lived. However, he is was also one of the most mysterious and least documented of all. Most of you probably know the *Pythagorean theorem*. It is beyond the scope of this course to provide the proof for the theorem. However, it is important to mention as it is foundational to many tasks.

The theorem, in its simplest form, relates the length of the three sides of a right triangle with the following identity:

$$a^2 + b^2 = c^2$$

where, *c* is the side opposite to the right angle (called the hypotenuse,) and *a* and *b* are the sides adjacent to the right angle. Figure 1.3 shows a graphical example.

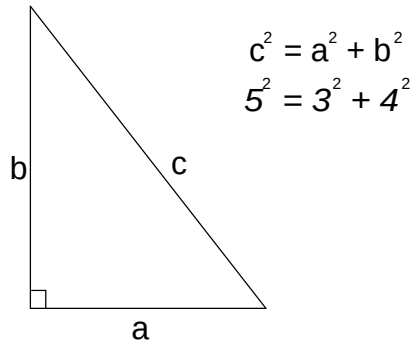


Figure 1.3 The Pythagorean theorem

The *Pythagorean theorem* is one of the most useful identities in mathematics, as you will observe in the sections to follow. One very useful application of this theorem is the *distance formula*, used to calculate the distance between two points.

Distance Formula

The distance formula states that the distance d (P_1, P_2) between point P_1 (x_1, y_1) and point P_2 (x_2, y_2) in a plane, is given by the relation:

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2}$$

This can be presented graphically as shown in Figure 1.4.

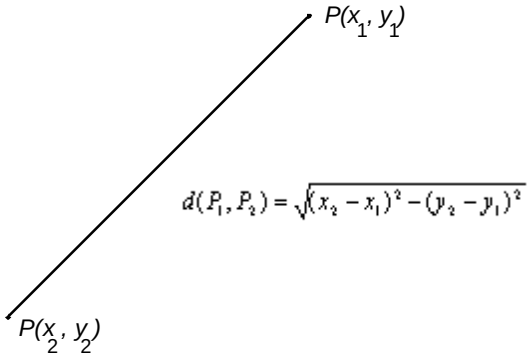


Figure 1.4 This distance between two points.

Keep in mind that d (P_1, P_2) = d (P_2, P_1) and hence the order in which the x and y coordinates are subtracted is immaterial. In essence, the line can be calculated beginning at either point.

BODMAS

Having studied in a Catholic school following a British curriculum, I've been through some rigorous math in my school days. We were never allowed to use a calculator, and if you were ever caught with one, the teacher would march you off to the Principal's office. And believe me that was never good!

In 2nd grade I had to take 3 math classes – arithmetic, geometry and lastly, mental mathematics. Mental mathematics, now that I look back, was really the most helpful. The idea was to make the kids learn their multiplication tables from one to twenty really well. The tests and quizzes consisted of several simple problems like:

$$4 + 5 * 6 / 3$$

Here's the twist – we'd get 100 problems on each test and about an hour to complete them. The problems got progressively tougher towards the end of the tests. We were allowed to use nothing but a pencil, an eraser and our heads. It was illegal to have a sheet to do rough work on. The only way to do well was to know your tables, and your math rules really well. I cannot emphasize how unpopular that class was amongst the students! Now that I look back I'm glad we had to take it because it's quite convenient to know what 19 times 12 is off the top of your head.

This class was where I was first introduced to a technique that can be abbreviated BODMAS. BODMAS stands for "Brackets Order Division, Multiplication, Addition, and Subtraction." BODMAS assigns priority to mathematical operators. "Order" is an old English way to say "raised to" or "to the power."

Let's see how this works. Take the above example 4 + 5 * 6 / 3. There are several ways to solve this problem, and each one would give different answers. A few permutations are:

$$4 + 5 = 9,\ 9 * 6 = 54,\ 54 / 3 = \mathbf{18}$$

$$5 * 6 = 30,\ 30 + 4 = 34,\ 34 / 3 = \mathbf{11.34}$$

As you can see, the process could go on and on and each time it could give you a different answer. So, what is really correct? This is where BODMAS can save the day. If you follow the order of priority i.e. DMAS, you can never go wrong. According to BODMAS, the above problem would work out as follows:

$$6 / 3 = 2,\ 2 * 5 = 10,\ 10 + 4 = \mathbf{14}$$

BODMAS is a handy tool when you don't have a calculator around. The application of this rule is seen in our day-to-day lives. And sometimes, you get a problem that you can't punch into your calculator, so you can apply BODMAS quite easily. It works for any type of math. Many computer languages follow a similar pattern of "operator precedence."

Basic Trigonometry

The word *trigonometry* is derived from the two Greek words *trigonon* (triangle) and *metria* (measurement). The applications of trigonometry are found in our day-to-day lives, forming the basis for many physics and math topics. We will be looking at some basics to help us understand supplementary concepts in kinematics.

Radian Measure

One radian is the measure of the central angle of a circle subtended by an arc equal in length to the radius of the circle. The radian measure is independent of the size of the circle. It is found by determining how many times the length of the radius of the circle is contained in the length of the subtended arc. If we consider a circle of radius *r*, then an angle O with a measure of 1 radian intercepts an arc *AB* having length *r*, as illustrated in figure 1.5.

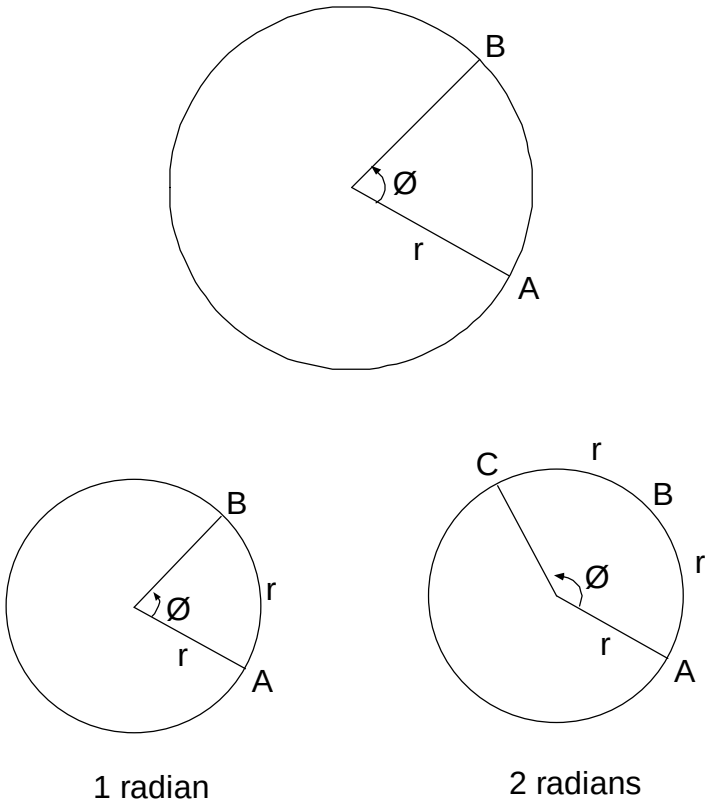


Figure 1.5 Understanding the radian

One important relationship you must remember is that between radians and degrees. Table 1.1 shows this basic relationship.

$180° = \pi$ radian
$1° = \pi / 180°$ radian $\cong 0.0175$ radian

Table 1.1 The relationship between radians and degrees.

Radian measure has no units in representation. So, if an angle has radian measure 22, we write O = 22.

Converting Radians to Degrees and Vice Versa

This is a very important section because of its direct application in ActionScript, and programming in general. Several functions used in ActionScript return the degree measure for rotation. Due to familiarity (if for no other reason), most of the time we use degrees as our base unit for manipulations. However, Flash's trigonometric functions (sin, cosine, and so on) require radian values. You can obviously see the implications of mistaking a radian for a degree. A possible scenario would be your spaceship turning left when you intended to make it go higher. Not good!

Table 1.2 illustrates the conversion between the two angle measurements, while table 1.3 shows some commonly occurring degree to radian conversions:

To change	Multiply by
Degrees to radians	$\pi / 180°$
Radians to degrees	$180° / \pi$

Table 1.2 Converting between radians and degrees.

Degrees	Radians
0	0
30	$\pi / 6$
45	$\pi / 4$
60	$\pi / 3$
90	$\pi / 2$
120	$2\pi / 3$
135	$3\pi / 4$
150	$5\pi / 6$
180	π
270	$3\pi / 2$
360	2π

Table 1.3 Common degree to radian conversions.

Flash and Radians

It is important to know how to determine angles in Flash and other programs. Flash's *polar* setup is similar in peculiarity to its coordinate system setup. Figure 1.6 illustrates a regular coordinate axes with respective angles of rotation.

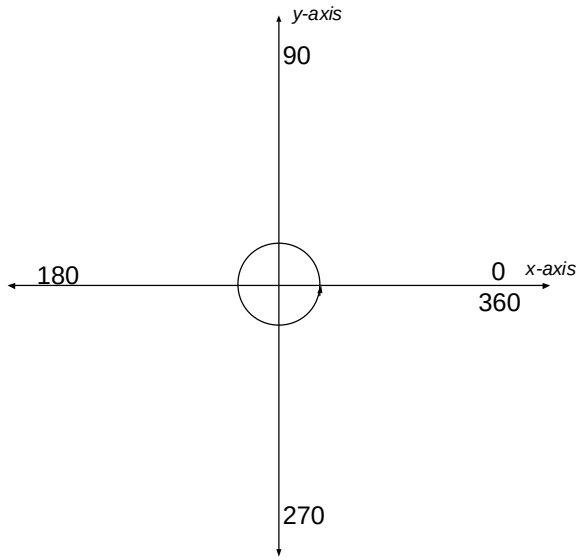


Figure 1.6 Regular coordinate axes.

This should be familiar to most of you. Traditionally, the positive x-axis is considered to be at 0°, with the angle increasing in the counter-clockwise direction. One whole revolution (or rotation, depending on how you are looking at it) accounts for 360 degrees of rotation in natural numbers. In contrast, examine the Flash coordinate system, as illustrated in Figure 1.7.

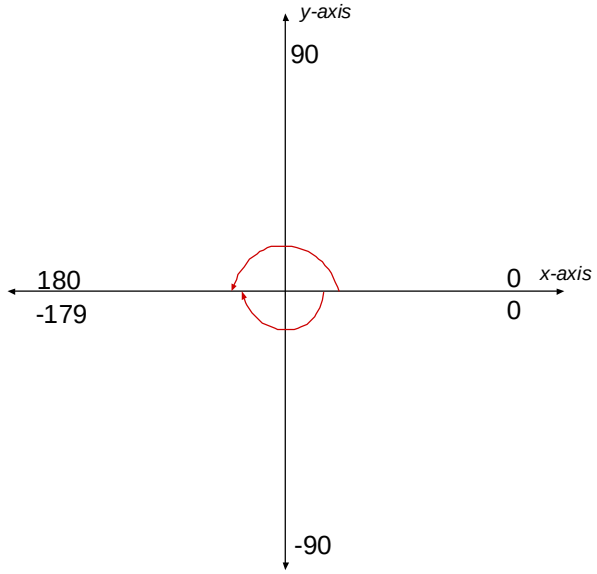


Figure 1.7 Flash's coordinate system.

As you can see, Flash has a peculiar setup for rotation angles. Though the starting point is the same (positive *x*-axis,) everything else is not quite intuitive. The angle increases in the clockwise direction until it reaches 180°. However, once you get back in the second quadrant, it starts increasing from a value of -179° (mind you, increasing in negative integers means getting closer to 0.) Hence, the first and second quadrants are simply negative reflections of the third and fourth. Although this may be unclear at this point, once you start using Flash's *_rotation* property, as well as its transformation tools, this will be more evident.

A simple way to overcome this peculiarity is to add 2π radians to a negative rotation result as follows:

```
if ( dy < 0 ) {
        radians = Math.atan2( dy, dx ) + ( 2 * Math.PI );
}
```

In several examples you will see this applied. The key is to remember that Flash uses negative numbers in certain rotation measurements.

Trigonometric Functions

Trigonometric functions relate the sides of a right triangle using established ratios. Any triangle is a right triangle if one of its angles equals 90°. Consider figure 1.8.

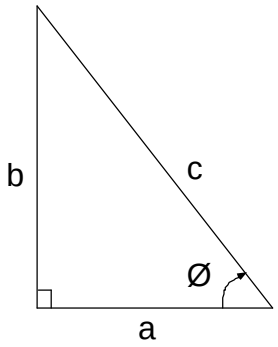


Figure 1.8 A basic right triangle.

If ? is any acute angle of right triangle abc, then the most vital ratios to be considered are:

$$\frac{b}{c}, \frac{a}{c}, \frac{b}{a}, \frac{a}{b}, \frac{c}{a}, \frac{c}{b}$$

Trigonometric functions establish the fact that these ratios are dependent only on ?, and not the actual size of the triangle. For each ?, the six ratios

have unique values. Since the ratios are dependent on ?, they are called *functions of ?*. These are termed **trigonometric functions**. As we go on, the importance of trigonometric functions will become all to clear to you. However, for the time being, we need to get the technical aspects out of the way.

Different Trigonometric Identities

The six important trigonometric functions are shown in table 1.4. These become the basic for many operations on graphic objects in Flash.

$$\sin q = \frac{\text{opp}}{\text{hyp}} = \frac{b}{c}$$
$$\cos q = \frac{\text{adj}}{\text{hyp}} = \frac{a}{c}$$
$$\tan q = \frac{\text{opp}}{\text{adj}} = \frac{b}{a}$$
$$\cot q = \frac{\text{adj}}{\text{opp}} = \frac{a}{b}$$
$$\sec q = \frac{\text{hyp}}{\text{adj}} = \frac{c}{a}$$
$$\csc q = \frac{\text{hyp}}{\text{opp}} = \frac{c}{b}$$

Table 1.4 The six important trigonometric functions.

Reciprocal Relations

It is often useful to define trigonometric identities in terms of reciprocals of each other as shown in table 1.5.

$$\sin q = \frac{1}{\csc q}$$
$$\cos q = \frac{1}{\sec q}$$
$$\tan q = \frac{1}{\cot q}$$
$$\cot q = \frac{1}{\tan q}$$
$$\sec q = \frac{1}{\cos q}$$
$$\csc q = \frac{1}{\sin q}$$

Table 1.5 Reciprocal relations or trigonometric identities.

So what does all this mean? It's very simple. The best way to remember these relations is to remember the underlying diagram behind the idea. So figure 1.8 shows it once again.

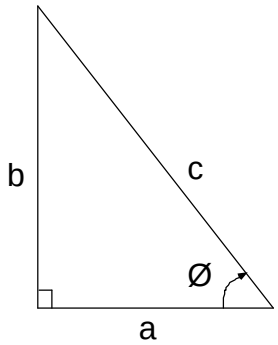


Figure 1.8 Remembering this figure is important!

In terms of the figure 1.8,

side c = hyp (hypotenuse)
side b = opp (side 'opposite' to ?)
side a = adj (side 'adjacent' to ?)

Since the hypotenuse of a triangle is always greater than its other sides, you can deduce certain relationships of trigonometric identities, as shown in table 1.6.

$\sin q < 1$	$\sec q > 1$
$\cos q < 1$	$\csc q > 1$

Table 1.6 Deductions concerning trigonometric identities.

Quotient Relations

The following relations are called *quotient relations* since the right side is equated to the left side as a quotient, as shown in table 1.7. Quotient relations are simply tools to interrelate trigonometric functions.

$$\tan q = \frac{\sin q}{\cos q}$$
$$\cot q = \frac{\cos q}{\sin q}$$

Table 1.7 Quotient relations

Quadrants and signs

Once you get into some development, you will realize that trigonometric values aren't always positive in sign. Quite to the contrary, most of the trigonometric values you will have to deal with will be negative. The signs

of the values depend on the positions of the arms of the triangle with respect to *quadrants*. You may refer to the table 1.8 to determine the sign of a trigonometric value. Figure 1.9 shows the quadrant labeling order.

Quadrant	Sin	Cos	Tan	Cot	Sec	Csc
First	+	+	+	+	+	+
Second	+	-	-	-	-	+
Third	-	-	+	+	-	-
Fourth	-	+	-	-	+	-

Table 1.8 Signs of trigonometric values with respect to quadrants.

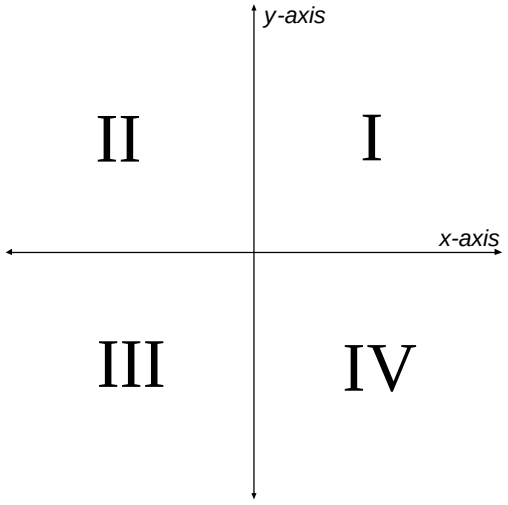
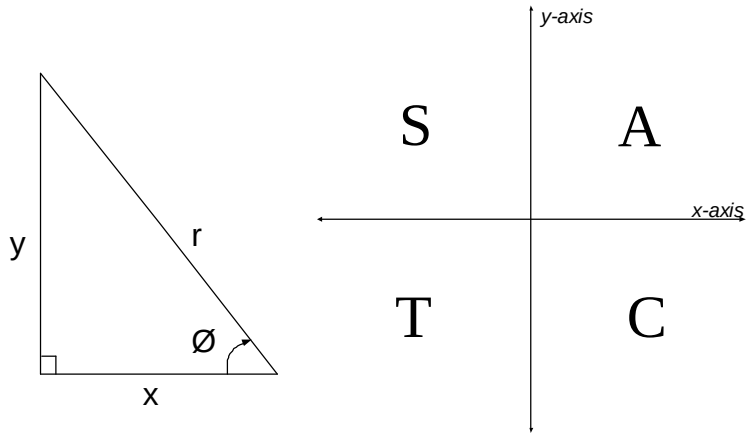


Figure 1.9 Quadrant Labeling.

There is a very simple way to remember the signs related to quadrants—actually, there are a couple of simple ways. You could remember the acronym "ASTC" which stands for "All - Sine - Tangent - Cosine" respectively. This denotes the positive values in order of quadrants.

For instance, all values are positive in the first quadrant, only sine values are positive in the second quadrant, and so on. If you remember your reciprocal identities (see table 1.5), you can easily figure out the signs for the remaining three functions. Always remember, sin is the reciprocal of csc, cos is reciprocal of sec, and tan is the reciprocal of cot. Hence, they have the same signs.

Another way to remember the table is to think of it logically. In the example triangle shown in figure 1.10(a), the adjacent side is arm x, and the opposite side is the arm y.



(a) (b)

Figure 1.10 Comparing the right triangle (a) to the ASTC idea (b).

The sine function is the ratio of *y* to *r*. *y* is always positive above the *x*-axis, and hence, the sine function always has positive values in the first and second quadrants.

The cosine function is the ratio of *x* to *r*. *x* is always positive on the right of the *y*-axis and hence, the cosine function has positive values in the first and fourth quadrants.

The tangent function is the ratio of *y* to *x*. *y* and *x* have the same signs in the first and third quadrants, and hence, the tangent function has positive values in these quadrants.

Similarly, the remaining signs may be derived using the reciprocal identities table. All reciprocal functions have the same signs.

Constructing A Special Value Trigonometric Table

We decided to put this section in at the last minute. It's very useful to know some common trigonometric values without having to use a calculator, especially when you are debugging code. If you know the sine values that correspond to the common angles like 0°, 30°, 45°, 60°, and 90°, you can extract all the others from those.

Let's construct the table. The first thing you need to do is draw a barebones table similar to the one in table 1.9.

	0°	30°	45°	60°	90°
sin					
cos					
tan					
cot					
sec					
csc					

Table 1.9 The basic table.

Now, fill in each column in the first row with 0, 1, 2, 3, and 4, respectively:

	0°	30°	45°	60°	90°
sin	0	1	2	3	4
cos					
tan					
cot					
sec					
csc					

Table 1.10 Number the cells.

Divide all the cells you filled in by 4. You get:

	0°	30°	45°	60°	90°
sin	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1
cos					
tan					
cot					
sec					
csc					

Table 1.11 Divide each cell by 4.

Calculate the square root of each cell:

	0°	30°	45°	60°	90°
sin	0	$\frac{1}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{\sqrt{3}}{2}$	1
cos					
tan					
cot					
sec					
csc					

Table 1.12 Determine the square root.

Now, let's fill in the remaining cells starting with cosine. The graph of the cosine function looks identical to the sine function graph with one major difference - it is shifted. This results in a shifted set of values. Simply copy the sine values backwards, by filling them in descending order by angle as shown:

	0°	30°	45°	60°	90°
sin	0	$\frac{1}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{\sqrt{3}}{2}$	1
cos	1	$\frac{\sqrt{3}}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{2}$	0
tan					
cot					
sec					
csc					

Table 1.13

Here's the real test to whether you learnt your tables of identities. We established in table 1.7 that:

$$\tan q = \frac{\sin q}{\cos q}$$

Hence, filling up the tan table is simply a matter of dividing the corresponding sin and cos values, and filling in the result. The values you should end up with are shown in table 1.14.

	0°	30°	45°	60°	90°
sin	0	$\frac{1}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{\sqrt{3}}{2}$	1
cos	1	$\frac{\sqrt{3}}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{2}$	0
tan	0	$\frac{1}{\sqrt{3}}$	1	$\sqrt{3}$	∞
cot					
sec					
csc					

Table 1.14 Resulting values for tan.

You could fill in the last three identities in a similar manner. Use Table1.7 for reference. Table 1.15 shows the completed table.

	0°	30°	45°	60°	90°
sin	0	$\frac{1}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{\sqrt{3}}{2}$	1
cos	1	$\frac{\sqrt{3}}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{2}$	0
tan	0	$\frac{1}{\sqrt{3}}$	1	$\sqrt{3}$	∞
cot	∞	$\sqrt{3}$	1	$\frac{1}{\sqrt{3}}$	0
sec	1	$\frac{2}{\sqrt{3}}$	$\sqrt{2}$	2	∞
csc	∞	2	$\sqrt{2}$	$\frac{2}{\sqrt{3}}$	1

Table 1.15 The completed table.

The purpose of this section was not just to help you memorize the table. We hope that the method employed here showed you the relationships between different trigonometric identities. This may seem monotonous at first, but as you move on to more complex sections, it will become crucial for you to know your trigonometric identities off the top of your head. We strongly encourage you to spend some time learning how to construct the table, simply to strengthen your grasp of trigonometric identities and their relationships with each other.

Vectors

Vectors are a programmer's best friends. They represent concepts like displacement, velocity, acceleration, or whatever else requires depiction of motion. In most books, you'll find that vectors are defined as physical quantities that have magnitude, as well as direction, and are represented by a line with a starting point and an ending point. Well, what does that mean?

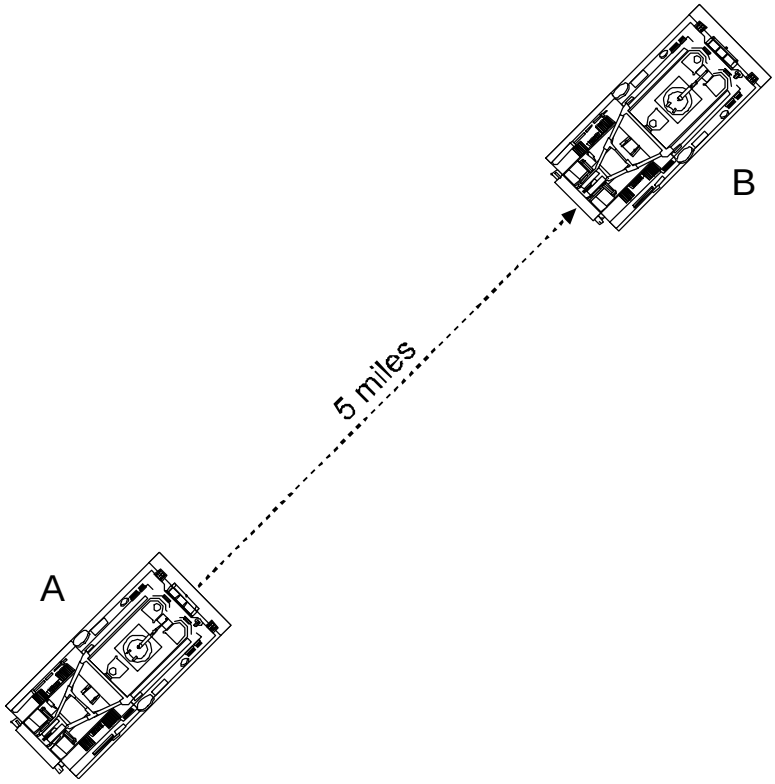


Figure 1.11 Understanding vectors.

In the figure, the tank has moved 5 miles from A to B. Now, if you were asked to describe the displacement of the tank, you might respond by saying, "The tank has moved 5 miles." But *all* this would tell someone is that the tank is 5 miles from its starting point. This could mean that the tank could lie anywhere on the circumference of a circle with a 5 mile radius with A as origin.

However, if you used a vector like the one in the figure, you could pinpoint the exact location of the displaced tank. Your answer would look something like; "The tank has moved a distance of 5 miles (*magnitude* of the vector) in a direction 45? North of East (*direction* of the vector.) This may seem like a lot to say when it comes to our little tank, but it makes life a lot easier when we start dealing with a tank moving around in the 2D plane of a computer

game, and representing a point in 3D space simply becomes close to impossible without using a vector.

As we move further on you will slowly grasp the concept of vectors if you haven't already. It's a concept that isn't something you can visualize immediately because of its abstract nature. Keep reading on and it will make more sense in the context of real examples. But before we get our feet wet with some *kinematics,* we need to dig up a few more concepts.

Properties of vectors

Vectors are very useful to a programmer. You can add them, and subtract them and do a bunch of other neat things with them. We will briefly describe some of the important properties, and for the really motivated ones, you'll find additional links to resources that go deeper into the subject.

Representation

We use PQ to denote a vector with initial point P and terminal point Q. We use letters like u and v to denote vectors with no terminal point.

Length or Magnitude

The Length of a vector is simply the distance from the origin to the tip of the vector. It is denoted by $|\boldsymbol{u}|$ and read as "the length of $\boldsymbol{u}$." As you can see in figure 1.12, you can use the *Distance Formula* to find the length of a vector.

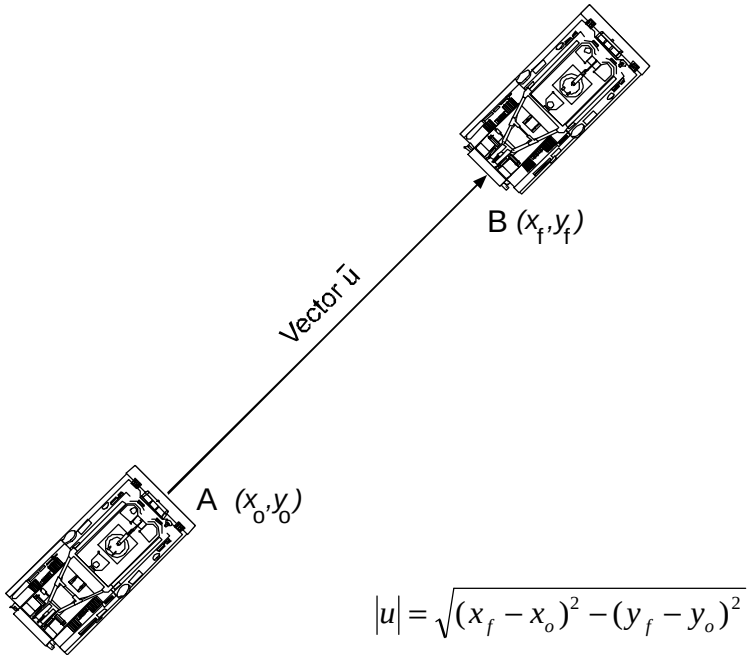


Figure 1.12 Applying the distance formula.

Vector Components

Vector components are the most useful tools that vectors provide us with. They facilitate the creation of very simple to extremely sophisticated motion based graphics programming. Understanding this concept itself is half the battle, as the rest that follows is simply application of these rules.

By definition, the perpendicular projections of the vector onto the x- and y-axes are called the *components of the vector*. The components of a vector u are simply two mutually perpendicular vectors, u_x and u_y that are parallel to the x and y axes respectively such that $u = u_x + u_y$. The component parallel to the *x*-axis is called the *horizontal component*, and the component parallel to the *y*-axis is called the *vertical component.* Take a look at the figure and it will make more sense.

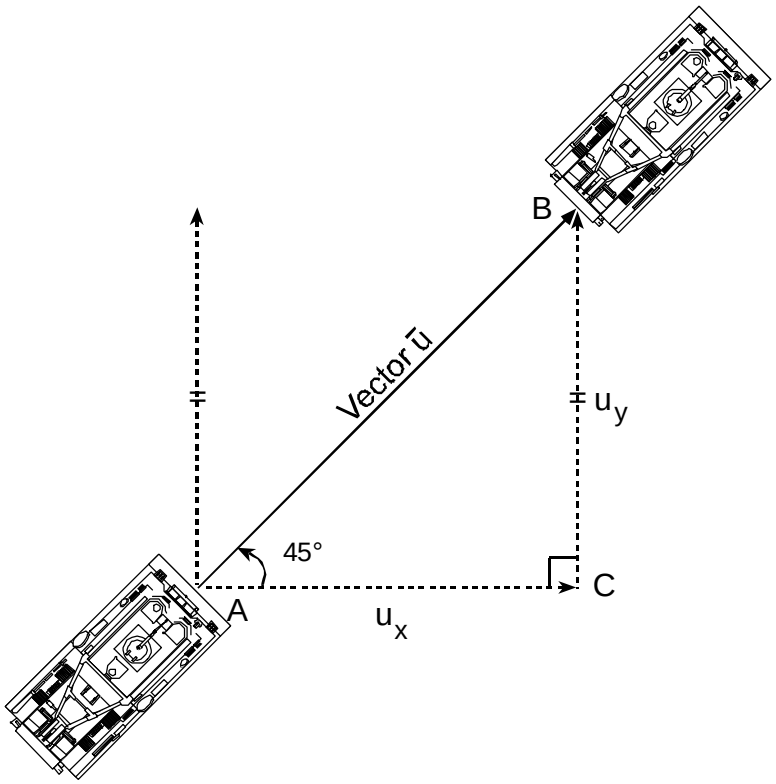


Figure 1.13 Components of the vector.

Let us explain with an illustration using our tank example. When our tank moves from A to B, its displacement vector is u. Now, if you draw a vector, u_x from A to C (parallel to the x-axis) and u_y from C to B (parallel to the Y axis, to form a right triangle of vectors ABC, then we call the vectors u_x and u_y the components of vector u. As you can see in the figure, u_x and u_y are mutually perpendicular. This opens up a world of opportunities for us to manipulate, and also supplies us with some invaluable relationships to make different calculations and transformations.

Concept: Resolving a vector

Physics and math books provide some of the most comprehensive explanations about resolving vector components (that's the techical way to say that you're going to figure out a vector's components.) But what does that mean in simple English? As far as we're concerned, we have a vector and we need to figure out its components.

Here's an unconventional, but extremely simple method you may use to resolve a vector.

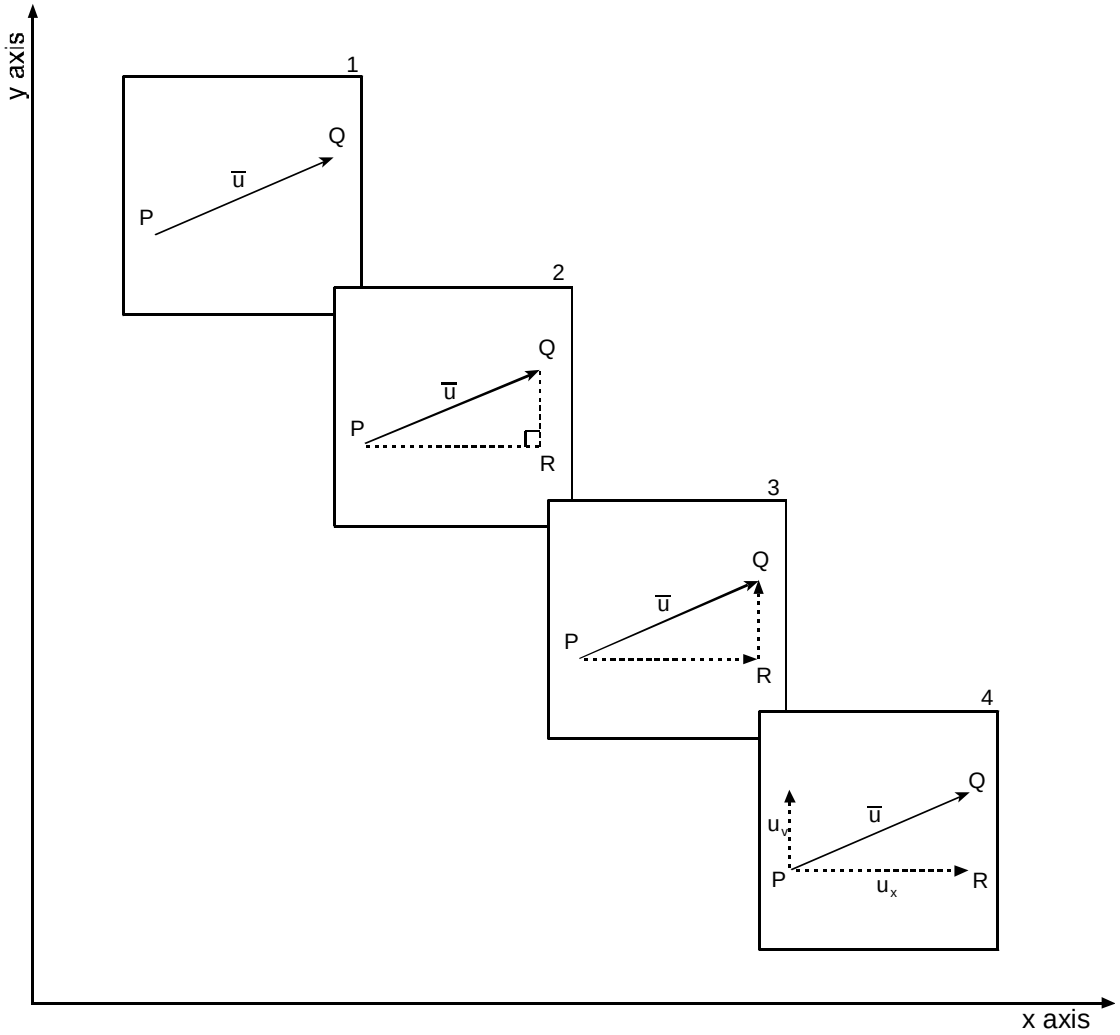


Figure 1.14 Resolving a vector

Step 1: Locate your problem vector i.e. vector **u** in this case.

Step 2: Draw PR and QR as shown to form a right angle triangle.

Step 3: Draw arrows as shown. Keep in mind, the arrows of each component trace a path from the initial point to the terminal point of the parent vector.

Step 4: Your vector has been resolved into the components u_x and u_y.

You're probably wondering why we moved the y component to the origin of **u**. That is how it's done conventionally. The idea is that each component translates the motion of the object assuming the absence of the other component. The x component tells you how the object would move as if there wasn't any movement in the y direction and vice versa. It's just simpler to think in terms of the right triangle and then move the vectors to the origin keeping the magnitude and direction the same.

Resolving a vector's components using trigonometry

In the previous section we noticed that a right triangle's arms play an important role in the formation of vector components. That should scream out one word to you – "Trigonometry!"

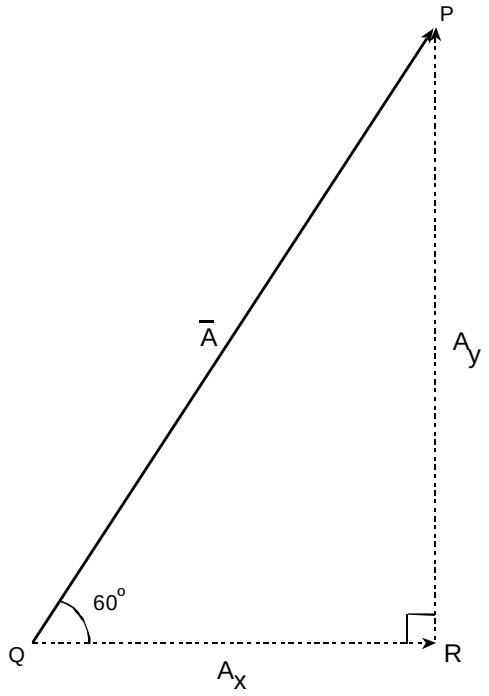


Figure 1.15 A closer look

Take a look at the figure 1.15. It's a vector diagram similar to the ones we looked at a while ago. Now, just for a few minutes, think of it as a simple right triangle PQR with angle PQR equal to 60?.

By definition:

$$\sin(60^o) = \frac{RP}{QP}$$
$$\cos(60^o) = \frac{QR}{QP}$$

Simplifying it:

$$\mathrm{RP} = \sin(60^o) \cdot \mathrm{QP}$$
$$\mathrm{QR} = \cos(60^o) \cdot \mathrm{QP}$$

If we think of this in terms of our vectors, we get,

$$\mathrm{A_y} = \mathrm{A} \cdot \sin(60^o)$$
$$\mathrm{A_x} = \mathrm{A} \cdot \cos(60^o)$$

The beauty of this is that if you know a vector's magnitude and it's direction, you can easily compute its components using basic trigonometry. Here's the definition of the above process:

Given magnitude (r) of a vector A and its direction (?) with respect to the x-axis, its components are given by the equations shown in table 1.16:

$$\mathrm{A_y} = \mathrm{r} \cdot \sin(q)$$
$$\mathrm{A_x} = \mathrm{r} \cdot \cos(q)$$

Table 1.16 Determining the components of a vector

Function Graphs

When dealing with animation and movement, as a computer graphics developer, you need to concern yourself with several kinds of functions. Some of the most important ones are:

- **Linear:** $y = mx + c$
- **Quadratic:** $y = ax^2 + bx + c$
- **Cubic:** $y = ax^3 + bx^2 + cx + d$
- **Trigonometric:** $y = a\sin(x)$

Getting into these functions in detail is beyond the scope of this course. However, we would like to highlight one important use of these functions. When graphed, most of these functions depict a relationship between two variables. For instance, the graph shown in figure 1.16 depicts a linear relationship between Velocity and Time. Hence, as time progresses, the velocity of the object increases in a linear fashion.

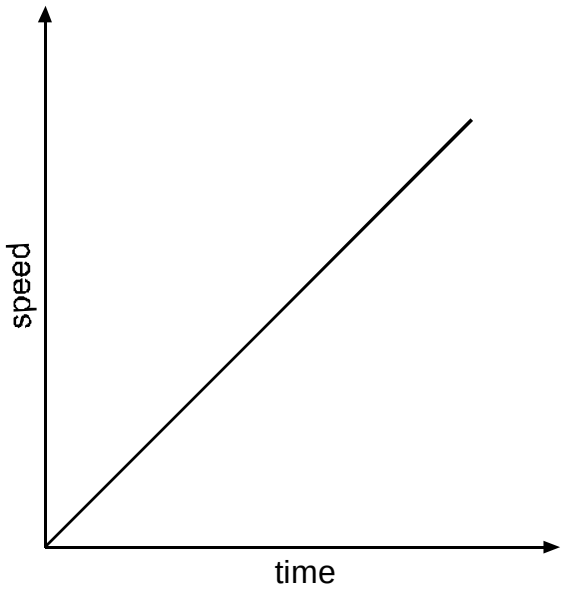


Figure 1.16 A linear relationship between velocity and time.

It is important to be able to plot functions. Most of the times, when you are trying to achieve some sort of complex motion, it can easily be plotted on a scientific calculator such as the TI-89, because motion is governed by functions and equations. If you can understand the graph of a function, you don't need to predict the motion it will produce. This makes for some extremely efficient conceptualization. If you plan to devise your own tweening functions with custom easing parameters, it will be crucial for you to have a strong base in Liner algebra and polynomial functions. Furthermore, if you can master plotting functions on a calculator, coding motion will become a lot simpler for you, and probably a lot more effective.

PHYSICS

Introduction

When you think of physics as it is related to multimedia, it's quite tough to pin down only those concepts that are important. Physics is a very broad science, and its implications to man are virtually endless. Similarly, the importance of physics in interactive multimedia, game programming, and motion-rich presentations is immeasurable. New technologies such as Flash provide developers with a blank canvas and a wide array of programming tools to incorporate real world kinematics and dynamics into their creations.

Unfortunately, Flash is in its teenage years on the maturity continuum. The general population of developers is content with key-frame based animation and cut-and-paste functions. While these definitely enhance presentations and applications, they fall short of actually harnessing the available infrastructure provided by the scripting language. As discussed earlier, the power of Flash and any other product similar to it, lies in an inter-disciplinary knowledge of incorporating mathematics, physics, and computer science in the final product.

One does not need to be a rocket scientist to achieve a high level of inter-disciplinary skills. Quite to the contrary, an individual with basic math and physics skills, some level of programming experience and a considerable amount of common sense, may easily master the art of blending these different variables into the final equation of a presentation, web site, game, or any type of media-rich experience. You must simply develop a wide knowledge base in these various areas and learn how to maximize on what you know and what is available to you.

The following sections were intended to serve as a review and reference of essential physics concepts. However, they are far from complete. We have simply documented the most basic concepts for you. Also, we do not intend to give you a series of cut-and-paste scripts in these course notes, for those are available in plenty over the Internet. We simply hope to give you a goal, direction and a head start to achieving that goal.

Mechanics

Motion, in a broad sense, has two main aspects. One aspect is the movement itself. For instance, is it fast or slow? The other aspect deals with the cause or origin of the motion. Both aspects are equally important to multimedia. However, the representation of both aspects is quite different in multimedia.

Mechanics is the branch of physics that focuses on the motion of objects and the underlying causes (forces) for the motion. Mechanics may be classified into two broad categories - kinematics, and dynamics. *Kinematics* deals with concepts that are required to describe motion without reference to the forces causing it. *Dynamics*, on the other hand, deals with how forces cause motion.

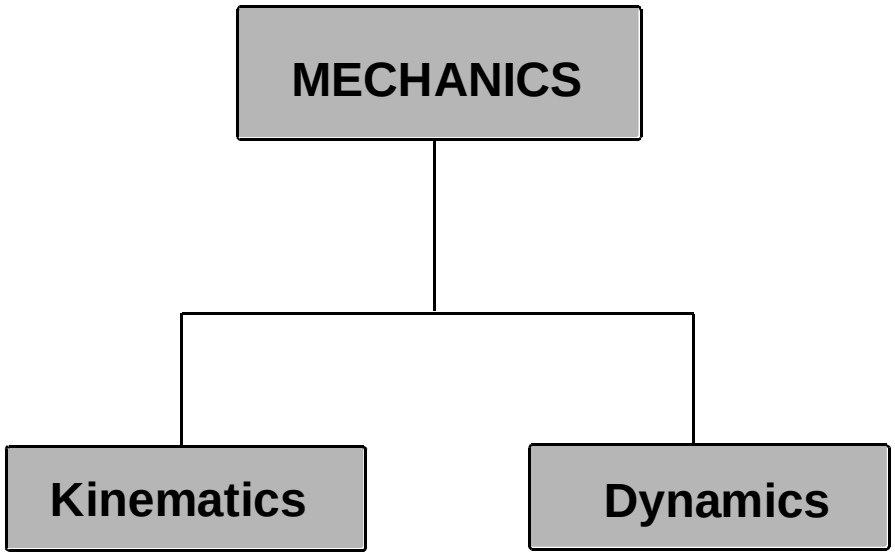


Figure 2.1 An overview of mechanics.

For the large part, we do not need to worry about dynamics for development at this level. Representation of forces is an abstract and advanced concept, best left to professional game developers. For our purposes, most any kind of motion, may it be linear, circular, hyperbolic, oscillatory, projectile, and so on, can be produced by manipulating variables.

The best part is that all the work has already been done for us. Each and every equation and concept of kinematics applies to the world of programming and multimedia. The key lies in knowing how to manipulate existing equations and concepts. Most people give up even if they know the concepts and have a sufficient amount of programming experience simply because they can't tie the two together. Our goal is to be able to develop a

problem-solving model that can help us tie these skills together to produce effective applications, games, or presentations.

Conceptual Kinematics

Kinematics deals with concepts that are required to describe motion without reference to the forces causing it. We will cover the following concepts:

- Displacement – shortest path of travel *or* final position minus initial position
- Distance – length of path traveled
- Speed – change in distance divided by the change in time
- Velocity – magnitude and direction of speed.
- Acceleration – change in velocity divided by change in time

After we understand what the above concepts are, we will learn how to apply them in simple game programming techniques and interactive multimedia to produce realistic results. Again, our main aim is to provide you with a firm idea of how things work, not to give you code snippets. Of course, there will be a few snippets to help conceptualize various routines, but we believe that with a strong understanding of kinematics, you will be able to use the rest of the techniques you've learned to produce your own code.

The mathematical quantities used to describe the above keywords of kinematics may be classified into two broad categories:

- Scalar Quantities - These are fully described by their magnitude alone.
- Vector Quantities - These are described by their magnitude as well as their direction.

Examples of scalars and vectors

The following examples should clarify the difference between a scalar and a vector:

- 22 km/hr - Scalar. There is no direction specified
- 22 km/hr, NE - Vector. The direction is specified.
- 1024 KB - Scalar. There is no direction specified
- 520 Amps - Scalar. There is no direction specified
- 5 pixels per second, 45degrees - Vector. The direction is specified.

Displacement

Consider a car in motion a shown in figure 2.2. The car moves from point A to point B. The difference '*x*' between points A and B gives the value of the *displacement* of the car. Simply put, *displacement is the length (magnitude) of the shortest path between two points.*

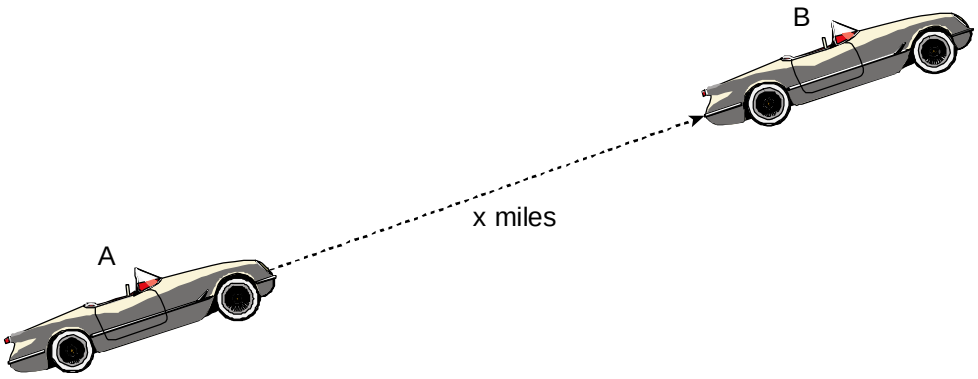


Figure 2.2 A car in motion.

Now, let's say the same car moves from point A to point C, but via point B, as shown in figure 2.3.

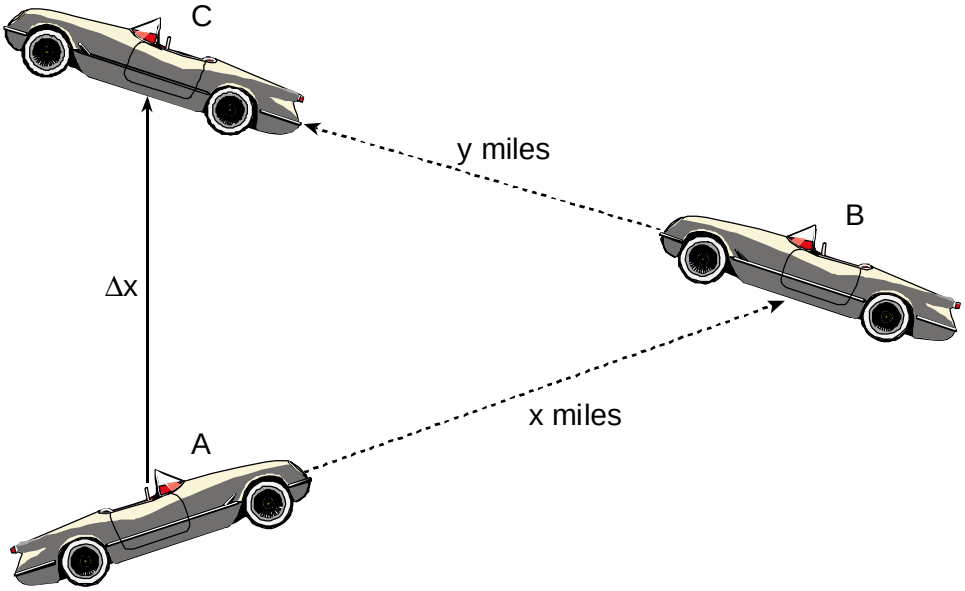


Figure 2.3 Adding a point to the movement.

Our first instinct would be to say that the displacement is the sum of the distances AC (x) and BC (y). This is INCORRECT. The displacement in this case is simply the magnitude of AC (? x). Always remember, the path

of the motion is not taken into consideration, only the initial and final positions of the object.

We shall denote displacement by '? x'

Displacement Vector

The displacement vector is simply a vector that points from an object's initial position to its final position. The distance between the initial and final points gives its magnitude.

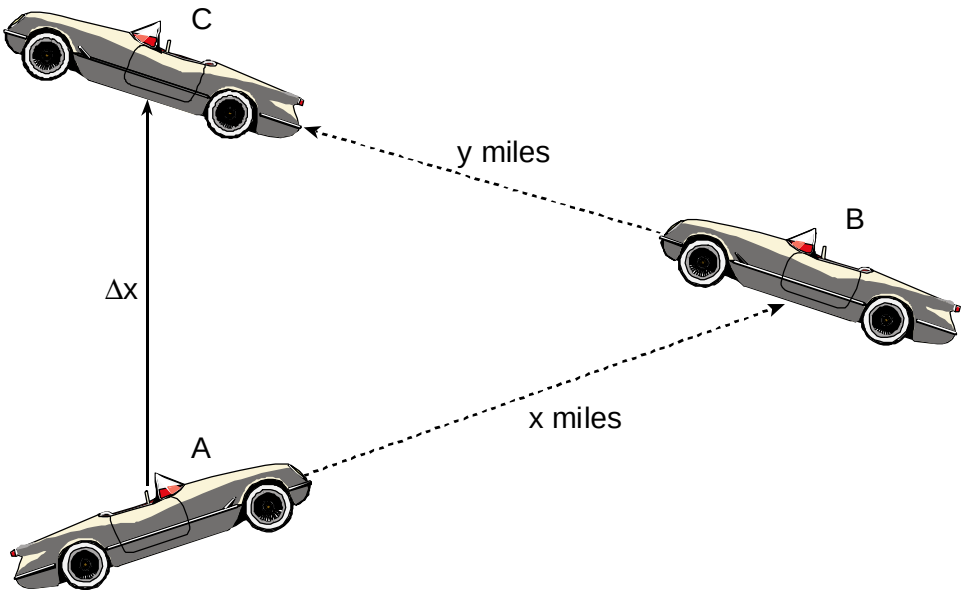


Figure 2.4 The displacement vector.

In the above figure, note that:

Displacement (? x) = Final position (x_f) - Initial position (x_o)

'? x' simply means "change in position." '? ' is the nickname for "difference between" or "change in."

Quick Review

Given,

x_o = initial position of object
x_f = final position of object
? x = displacement of object (change in position of object)

Then,

? x = x_f - x_o

Distance

Distance is a scalar quantity that represents the total path traveled by an object. If we take the above example, the distance traveled would be:

$$Distance\ (d) = x + y$$

It is important to be able to make a distinction between distance and displacement. Several of the following sections will require you to use either distance or displacement in their definitions. If they are interchanged, you will yield an incorrect solution.

Speed

Speed is the amount of time it takes to do something. When dealing with kinematics, *speed is defined as the distance traveled in unit time.*

$$Speed = \frac{Distance\ Traveled}{Elapsed\ Time}$$

The key point to remember is that we are dealing with "distance" and not "displacement." Speed is a very useful concept in understanding velocity, because like distance and displacement, speed and velocity are very similar, and often are terms used interchangeably. However, speed is *not* velocity and vice versa.

Speed is a scalar quantity *i.e.* it has magnitude, but no direction. Hence, speed does not require a vector for its representation. Speed can be represented completely by a number and a unit. For example, one can say that the speed of the plane is 200 mph. But what if we need to find out how fast the plane is moving as well as the direction it is moving in? This is where we are introduced to the concept of *velocity.*

Velocity

Velocity is defined as *the displacement of an object in unit time.* Notice, the word "displacement."

$$Velocity = \frac{Displacement}{Elapsed\ Time}$$

Due to the presence of displacement (? x) in the formula, velocity becomes a vector quantity. Recall, a vector quantity is one that has magnitude, as well as direction.

Velocity Vector

The magnitude of the velocity vector is given by the formula:

$$\textit{Velocity} = \frac{\textit{Displacement}}{\textit{Elapsed Time}} \qquad (1)$$

We already know that,

$$\text{Displacement } ?x = x_f - x_o \qquad (2)$$

Now let:

t_o = time when motion started (time at x_o)
t_f = time when motion ended (time at x_f)

Then:

$$\text{Elapsed time } (?t) = t_f - t_o \qquad (3)$$

Substituting (3) and (2) in (1), we get,

$$|v| = \frac{x_f - x_o}{t_f - t_o} = \frac{\Delta x}{\Delta t}$$

The direction of the velocity vector is the same as the direction of its displacement vector. We already learned how resolve a vector's components in an earlier section.

Quick Review

$$\textit{Speed} = \frac{\textit{Distance Traveled}}{\textit{Elapsed Time}}$$

Given,

x_o = initial position of object
x_f = final position of object
$?x$ = displacement of object
t_o = initial time of object before motion
t_f = final time of object after motion
$?t$ = elapsed time

Magnitude of velocity vector => $|v| = \frac{x_f - x_o}{t_f - t_o} = \frac{\Delta x}{\Delta t}$

The velocity vector points in the same direction as its displacement vector.

Acceleration

Constant velocity is not a practical phenomenon. It works well in theory, but in reality a moving object's velocity is bound to change over time. For example, the velocity of a car steadily increases when its driver steps on the gas pedal. Varying velocity is something that can be explained by *acceleration.*

Acceleration is defined as *the change in velocity of an object in unit time.*

$$Acceleration = \frac{Change\ in\ Velocity}{Elapsed\ Time}$$

This concept is best explained by an example. Examine figure 2.5.

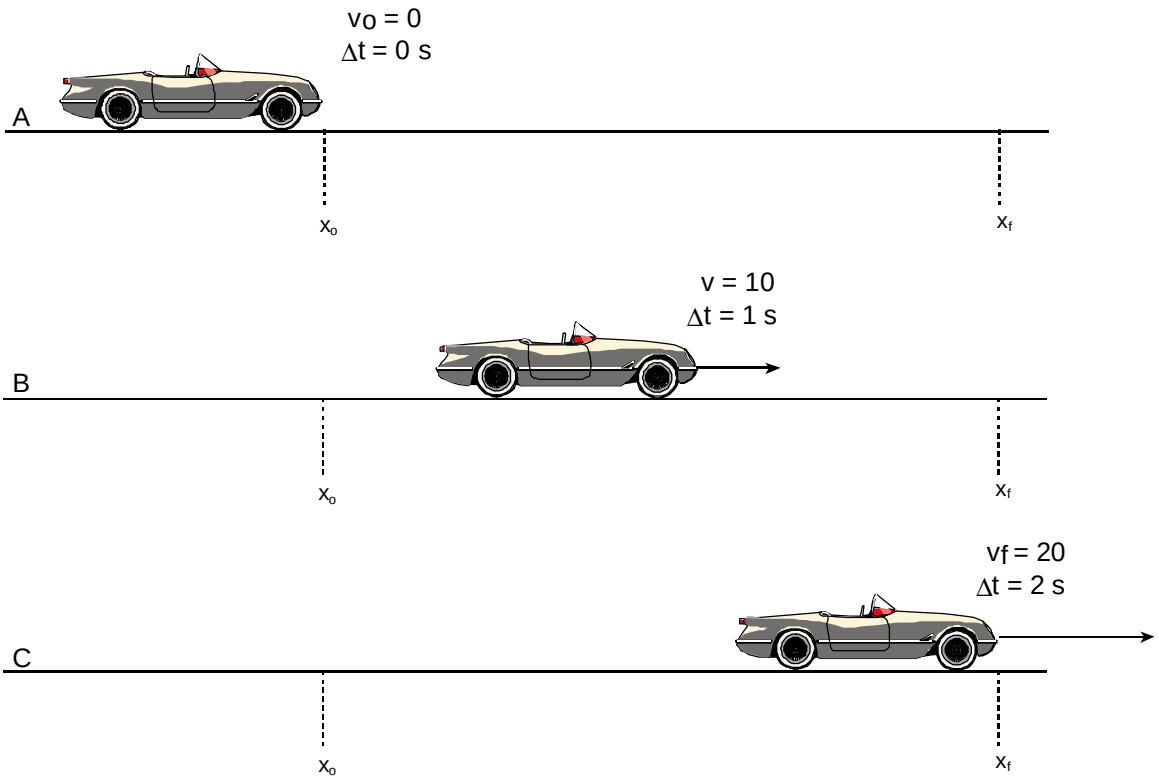


Figure 2.5 Understanding acceleration.

A car is at rest as shown in the figure 2.5(A). At some point the driver steps on the gas pedal and the car begins to move with an acceleration factor of 10 km/hr/sec. This means that the car's velocity increases by 10 km/hr every second. So, as shown in figure 2.5(B), after 1 second, it's velocity is 10 km/hr and after 2 seconds, it is 20 km/hr. That's all there is to acceleration.

Acceleration Vector

We know that,

$$Acceleration = \frac{Change\ in\ Velocity}{Elapsed\ Time} \quad (1)$$

Given,

v_o = Initial velocity
v_f = Final velocity

Then,

$$\text{Change in velocity } (?\,v) = v_f - v_o \quad (2)$$

We already know that,

$$\text{Elapsed Time } (?\,t) = t_f - t_o \quad (3)$$

Substituting (3) and (2) in (1)

$$|a| = \frac{v_f - v_o}{t_f - t_o} = \frac{\Delta v}{\Delta t}$$

The acceleration vector points in the direction of the change in velocity and has a magnitude equal to the change in velocity.

Negative Acceleration (Deceleration)

At times, the acceleration of an object is opposite to the direction of its motion *i.e.* the direction of velocity. An example of this would be when a car is braking. It moves ahead, but with reducing velocity. If you recall, change is velocity over time is known as acceleration. Hence, the braking car satisfies this condition and is actually accelerating, even though it's coming to a halt. This sounds peculiar because we tend to pair acceleration with increase in velocity. This type of acceleration is called *deceleration.* Simple enough? Deceleration is also known as *negative acceleration*. The following example (figure 2.6) should make it clear.

Figure 2.6 Negative acceleration.

In the above figure, the car is moving at 20m/s. Its brakes are applied with a deceleration factor of –10 m/s^2. This means that the car's velocity decreases by 10 m/s per second (hence, s^2 in the denominator of the unit.) At that rate the car will come to a complete halt in 2 seconds.

It is important to note that the '–' sign does not actually imply a negative number. It simply denotes that the acceleration is in the opposite direction of velocity.

Equations of Kinematics

There are four important equations of kinematics. We are not going to derive these equations because it's beyond the scope of this course. However, we are simply going to give you the equations and you'll have to take our word for it that they're correct. At this point it is not important to understand where they came from, but to know that they are very useful in finding unknowns like velocity, time, displacement, and acceleration when the acceleration is kept constant. If you feel really compelled, and are one of those people, go ahead and memorize them! I assure you that it will help.

$$v_f = v_o + at$$

$$x = \frac{1}{2}(v_o + v_f) \cdot t$$

$$x = v_o t + \frac{1}{2} a t^2$$

$$v_f{}^2 = v_o{}^2 + 2ax$$

Where,

- x = displacement
- v_o = initial velocity
- v_f = final velocity
- a = acceleration
- t = time in seconds

Projectiles and Trajectories

Free Falling Bodies

To understand projectile motion, we first need to understand the term *gravity*. Gravity, as we all know, is the natural phenomenon that causes all bodies to fall downward (or towards each other, depending on the perspective.) An important thing to remember about gravity is that it has the

same effect on all bodies, regardless of their size or mass. This would imply that a rock and a feather, if dropped from the same height would fall downward with the same acceleration. This probably won't happen if you go out and try it because of the effects of air resistance on the objects. However, in the absence of other factors like air resistance, all bodies fall with the same acceleration value despite size and mass differences.

Bodies that fall under the influence of gravity are called *free- falling bodies.* For practical purposes, the value of acceleration due to gravity (denoted by the letter 'g') is 9.8 m/s^2. That number doesn't mean anything to us yet. We will take a look at the concept of gravity in the examples section.

All the equations of kinematics may be used for free falling bodies by replacing the *x* with *y*. This is because essentially the body moves only in the y direction.

Projectile motion

A *projectile* is any body that is thrown or projected by some force (for instance, an impulse) and continues in motion for a short period of time by its own inertia. The path traced by a projectile is called a *trajectory.* Ideally, in the absence of all external factors such as gravity and air resistance, a projectile would move with constant velocity. However, we can't avoid gravity, and due to this most projectiles undergo acceleration.

The motion of a projectile is a little complicated, but if you break it up into logical parts, it's just like resolving a vector. In fact, it *is* just resolving a vector. We must look at the vertical and the horizontal motions individually.

To understand a projectile, let's look at a simple example. In figure 2.7 (A), a ball is shown rolling off a cliff in the absence of gravity and hence, it continues it's motion in the horizontal direction. You notice that the ball covers equal distances in equal intervals of time. In figure 2.7 (B), the same ball is dropped off the edge of the cliff with gravity present. The ball falls vertically under gravitational acceleration. This time the ball covers successively larger distances in equal intervals of time (because it is accelerating due to gravity.)

Now, what would happen if we assumed that gravity was present and we rolled the ball of the cliff with the same speed as in the first case? As shown in figure 2.7(C), the ball covers the same horizontal distance as it covered in the first case, while falling at the same rate as it fell in the previous case. This makes life simple for us. All we have to do in the case of programming a projectile is approach the horizontal and vertical components of motion individually (as if the other were absent), and displace the ball an amount equal to each component at instantaneous time/frame intervals.

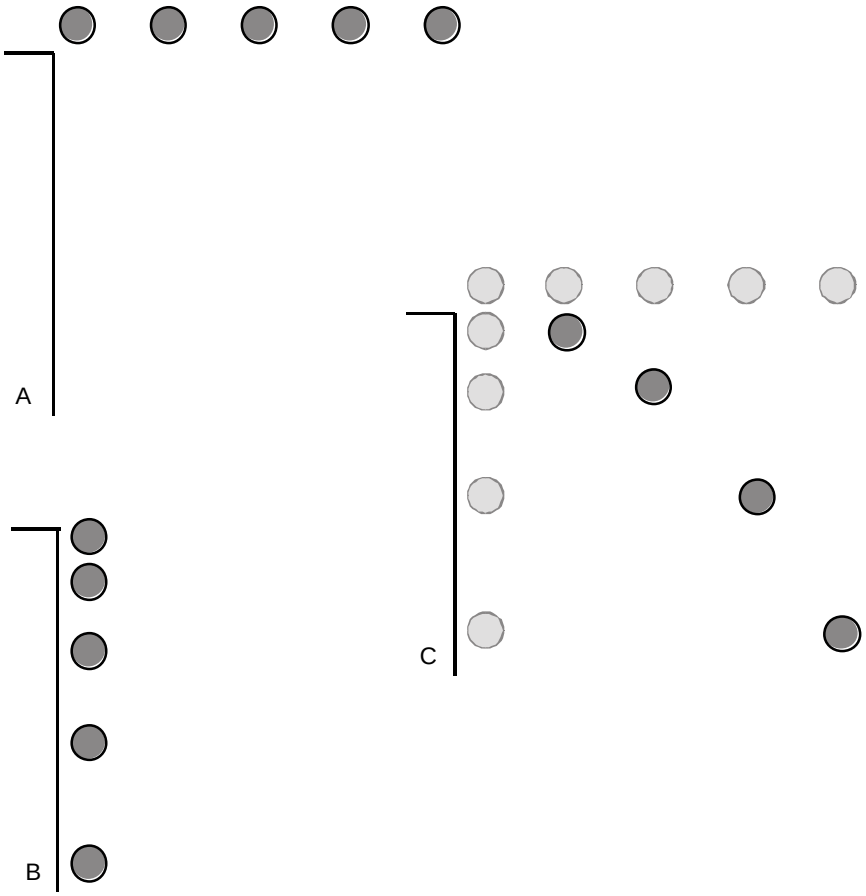


Figure 2.7 Understanding projectile motion

To summarize projectile motion, here are a few important points:

- Trajectories are always parabolic.
- A projectile achieves its maximum range when its launch angle is 45°.
- The horizontal component of velocity is always constant for the trajectory.
- The vertical component of velocity for the trajectory is variable based on the gravitational pull on the projectile.
- At its highest point, a projectile's vertical component of velocity is 0.

For our purposes, we may resolve the components of projectile motion using the formulae shown in table 2.1.

To Find (instantaneous)	Formula
x (t) - horizontal displacement	$(v_o \cos q)t$
y (t) - vertical displacement	$(v_o \sin q)t - \frac{gt^2}{2}$
v_x (t) - horizontal velocity	$v_o \cos q$
v_y (t) - vertical velocity	$v_o \sin q - gt$

Table 2.1 Components of projectile motion.

Elasticity and Spring

Motion is incomplete without the effects of elasticity and spring. To understand elasticity, we need to understand force. *Force is simply anything that brings about a change of state in an object*. We cannot really delve into force too much, but it is an important concept for you to understand. We recommend you a good physics book and just understand the concept, rather than get tied down by dynamics equations.

Elasticity is the property possessed by a body to restore itself after a deformation or force has acted upon it. There are other definitions, but this is the one that applies to us the most. Elasticity is characterized by something called a *restoring force*. Just as its name indicates, a restoring force is one that tries to restore a spring or elastic object to its original state.

The restoring force of a spring is given by Hooke's Law.

$$F = -kx$$

where k is the spring constant and x is the displacement of the spring from its original position. The spring constant is simply the strength with which the spring recoils. The minus sign indicates that the restoring force always points in the direction opposite to the displacement of the spring.

Ideally, in a frictionless environment, a spring would never stop oscillating after pulling and releasing it once. However, we can work around this. In

computer graphics, Hooke's law can be manipulated into the following form:

$$a = -k(dx) + cv$$

We will elaborate on this formula in our examples.

APPENDIX A

Flash Resources on the Web

The Internet is filled with loads of great Flash resources. We've tried to identify some of the most prominent and useful ones in this appendix:

Flash Kit

URL: http://www.flashkit.com/
Flash Kit is a collective of ideas and resources, and has established itself as the largest resource for Flash movies over the past year. It is a place where the Flash community shares new ideas, techniques, and creations in open source format. The web site has a large array of archives ranging from animation to 3D programming, with great tutorials if you know how to search its database. It also offers free sound loops, and a wide variety of other resources.

Flash Coders

URL: http://chattyfig.figleaf.com/flashcoders-wiki/
FlashCoders is a mailing list that is focused on programming with Macromedia Flash. The list is run by Branden Hall and is hosted by his employer, Fig Leaf Software. This is probably the best resource for extremely technical and accurate information on advanced ActionScripting, with topics ranging from OOP to inverse kinematics. If you are serious about ActionScript, this is the place to go to.

Debreuil.com OOP pages

URL: http://www.debreuil.com/docs/
If you are new to OOP and don't understand how prototypes work in Flash, then this resource will answer all your questions. Hosted by Robin Debreuil, this is a very refreshing approach to understanding OOP in Flash. Robin explains the concepts in simple English, unlike most other resources. It is a must for beginners, and experts, alike.

Motionculture.org

URL: http://www.motionculture.org/
" By definition, MotionCulture can best be described as; a community comprised of artists, programmers, enthusiasts and professionals that share a common philosophy about Web growth, development and the open exchange of ideas, techniques and source files were as enriching our community through a cooperative effort." Just as quoted, Motionculture makes up for the shortcomings of other sites. Their collection is smaller, yet very helpful.

Layer51

URL: http://www.layer51.com/

Aptly titled, this is one of our favorite cut-and-paste resources for Flash prototypes. If you are anything like us, then it probably helps you to see a lot of examples before you completely grasp a concept. This is the best resource to grasp the concept of a "prototype" object if you're such a person. The site offers a wide array of prototype objects arranged very efficiently by Flash object categories.

Levitated

URL: http://www.levitated.net/

Levitated is a combined effort of Jared Tarbell and Dr. Lola Brine. Every movie in the archive is open source (no technical support, though.) The developers present some very unique and creative examples that deal with isometric and three-dimensional Flash utilizing complex algorithms. We recommend this site both for its technical and aesthetic values.

Were-Here Forums

URL: http://www.were-here.com/

Yet another great Flash community portal, were-here hosts a never-ending string of useful threads where experts and amateurs discuss the best and worst of Flash. If you have a question and time to search for it, you will probably find an answer on were-here. If you don't, you'll have a good reason to start a new thread on their web site.

Ultrashock

URL: http://www.ultrashock.com

Ultrashock is a fairly new resource, but is growing fast. It offers a fairly decent range of example files and forums. Several well-known Flash developers serve as moderators on the Ultrashock forums.

FlashGuru

URL: http://www.flashguru.co.uk

Started by Guy Watson, FlashGuru.com is one of the better Flash community sites on the Internet right now. Watson, who served as a beta-tester for Flash MX, accumulates and shares the latest information about Flash on FlashGuru. FlashGuru is much like FlashCoders with a slight touch of casualness. You should bookmark this site as it very frequently updated and keeps in touch with the latest in the Flash world.

Moock.org

URL: http://www.mock.org

Colin Moock, the author of the infamous "Actionscript: The definitive guide", hosts this great web site. Moock's web site has a great knowledgebase of cross-disciplinary resources such as programmatic

physics, trigonometry and database integration. This is a resource filled with great resources, and hence, another must-see on our list.

Macromedia Flash MX Application Development Center

URL: http://www.macromedia.com/desdev/mx/flash/

Straight from the guys who made Flash, this resource has a great array of tutorials, articles and free stuff. A lot of tips and tricks, visionary articles and intermediate to advanced tutorials make this resource an invaluable one to the Flash community.

APPENDIX B

Math and Physics Resources on the Web

The Physics Classroom

URL: http://www.glenbrook.k12.il.us/gbssci/phys/Class/BBoard.html
This is an invaluable resource for theoretical explanations with lots of animated examples. Sections include 1-D Kinematics, 2-D Kinematics, vector math, and a list of other important concepts that this course could not cover. If you found the concepts covered in the course useful, then you will love this resource

GameDev.net

URL: http://www.gamedev.net/reference/list.asp?categoryid=44
This web site contains articles dealing with traditional isometric game design using different tiling systems. We simply managed to brush the tip of the iceberg with this topic in our course, but it is nonetheless a very interesting and important topic. GameDev is not Flash-based, but all the concepts are truly mathematical in nature and may be easily applied into Flash.

Inverse Kinematics

URL: http://freespace.virgin.net/hugo.elias/models/m_ik.htm
If you need an easy start on inverse kinematics, then this is a great resource for you. Inverse kinematics is becoming a reality in Flash, especially since the introduction of Flash MX and its drawing API. This topic is fairly complex and it would help for you to have some back ground in animation to get better at it. Nevertheless, this is a very valuable article.

Basic 2D and 3D Math

URL: http://www.geocities.com/SiliconValley/2151/math2d.html
http://www.geocities.com/SiliconValley/2151/math3d.html
Written by Mark Feldmen, these are great excerpts to understand the basics of 2D and 3D math as they relate to game programming. They explain the relationships between vectors and programming very well, and in simple English.

MathTV

URL: http://www.mathtv.com/Trig/pages/toc.htm
This is the great resource for whose of you who like a classroom approach. You simply browse through the sections of trigonometry concepts and click on tutorial video clips. The site lacks in theory but has great visualization examples. If you have a few hours to spare and want to reinforce the concepts that you already know without really working with a book or on paper, then login to MathTV, sit back, and simply listen.

ExploreMath.com

URL: http://www.exploremath.com/

Providing comprehensive shockwave examples, this site is complete and thorough. The nice thing about this site is that it offers completely free services conditioned for an educator. All the lectures have been designed for an audience of professors and teachers, explaining how the topic should be taught. All you have to do is register and go through the lectures, labs and examples. We took the liberty to set up a default account for this course if you would just like to get a feel for the site:

Login	: **siggraph**
Password	: **siggraph**

Flash Kit Physics and Math Forum

URL: http://board.flashkit.com/board/forumdisplay.php?forumid=63

A new forum that sprung up on Flash Kit just recently (we hope it will around by the time these course notes are released), it has some pretty neat solutions and ideas. The search capabilities are weak, but if you have time to browse through the threads, you will probably find your answer. If you don't, you'll still learn something new every time. We did.

APPENDIX C

OOP Primer for Flash MX

Introduction

Flash MX truly models itself after ECMAScript's Prototype and object model. Contrary to what you may have heard, Flash is not a true object-oriented language in the real sense of the term. This is because Flash doesn't have "Classes", rather it uses something called a "Prototype." We will get into the nitty-gritty of prototypes later in this section. First we need to understand the prototype-based programming paradigm. For the remainder of the section, prototype-based programming will simply be referred to as OOP.

OOP: Understanding the concept

Object oriented programming, or OOP, is not really something that you can get a tutorial or a solid example on. This is because OOP is not really a programming language, or something that we can associate a solid object with, but simply a very logical way of thinking about programming. In fact, object oriented programming is so logical that it applies almost to every aspect of computing. Without thinking about it, each one of you really uses OOP in your daily lives.

In its simplest form, OOP is simply a set of logical rules that facilitates the re-use of programming code. In the good old days, programmers had only one way to approach their programs known as the "Procedural Model." Procedurally -written code simply allows a programmer linear freedom with his logic structure. We will leave it to you to read up about procedural programming and its shortcomings, but to summarize them, here's why there was a need for a new paradigm in programming:

- Procedural code can be interpreted only by its own programmer. Due to the length a program could attain by completion, it is virtually impossible for someone else to go look at the code and know what's going on. This brought up several inconveniences.

- Procedural code is not portable. It cannot be re-used or shared. Also, it is a lot slower than OOP code.

There were several other reasons for the need of a paradigm shift, but these were the main ones. With that said, let's understand OOP.

Objects

If you hadn't guessed it already, the heart of OOP lies in the "Object." An object is very similar to objects we see in our lives everyday, like chairs, tables, lamps, cats, etc. Objects in programming (and in real life) are characterized by two main attributes:

- State
- Behavior *or* Function

The *State* of an object is simply something that describes one of its physical characteristics. For example, cats have state (name, weight, color, cranky, sleepy, etc.) The *Behavior* of an object is something that describes an action performed by it. For instance, cats have behaviors (yawn, purr, scratch, etc.)

In programming, objects are characterized by *state* and *behavior* too. Objects store their state in variables, while their behavior is stored in methods and functions. If this seems too abstract, it'll become easier as we move along and examine solid examples.

Classes and Instances

A class is simply a template for an object. A class defines what *state* and *behavior* that object will possess. It isn't really something that you can feel or touch. For instance, think of the word dog as a class. You cannot really touch the 'idea' of a dog because it is simply a set of *states* and *behaviors* that define a real dog. Still don't get it? Read this paragraph again, and then continue. The tough thing about OOP is that you have to change the way you think. Once you can see things in an OOP way, it will open up boundless opportunities for you as far as application development is concerned.

Let's use a simple example to illustrate what a class, object and an instance means. We haven't talked about instances yet, but they are better explained with an example:

```
Dog = function (name) {
     this.name = name;
     this.legs = "4";
}

myDog1 = new Dog ("Bonzo");
myDog2 = new Dog ("Oliver");

trace (myDog1.name); //Bonzo
trace (myDog2.name); //Oliver
trace (myDog1.legs); //4
trace (myDog2.legs); //4
```

The above example demonstrates some key OOP principles. Let's look at them one by one. First, we declared a `Dog` class. The `Dog` class defined the following states:

- Every dog will have a name. This name will be decided by the creator of the dog.
- *This* particular dog will have four legs.

After defining a class, we used some simple code to make an *instance* of the `Dog` class. Instances are simply objects. Every instance may have properties, variables and methods associated with it. In the above example, we used made an instance of a Dog class called "Bonzo" and stored it in the object myDog1. We made another instance of Dog class called "Oliver" and stored it in the object myDog2. Now myDog1 and myDog2 are completely different objects (instances) of the kind Dog.

For those who come from an OOP background, you must have already noticed a major flaw in this code. It is almost right… but not just quite yet. To completely capitalize on the OOP programming structure we need to look at a few more concepts.

The Flaw

Let's look at the above code again.

```
Dog = function (name) {
     this.name = name;
     this.legs = "4";
}

myDog1 = new Dog ("Bonzo");
myDog2 = new Dog ("Oliver");
```

Let's stop to think about our dog class. We want to be able to name our dog and we already provided a mechanism for that. However, we know that all our dogs will have 4 legs. With the above approach, every time we make a `Dog` object, we are creating a new *instance* for that `Dog` object's `legs` property. If we made a hundred dogs using the above code, we'd end up using a lot of memory to make a hundred instances of the legs property. That simply makes for waste of space and slows your movie down.

Fortunately, there is a solution to this problem. And the solution is called "Prototype."

Prototype and Inheritance

Prototypes confuse even the best of OOP programmers. To understand a prototype, we need to further understand a class. We already defined a class as a template that objects use to *instantiate* (make an instance) themselves. However, because of how a prototype-based language is set up, there is no way to access the properties of a class. For instance, in the above example, you couldn't just do the following.

```
Trace (Dog.legs); //undefined
```

So, what would we do if we needed to change the legs property for all dogs to "5"? This is where the prototype comes into the picture. The prototype object is simply a place where a class stores its properties. Everything that is owned by a class is stored in the prototype object. This makes for efficient coding because it uses the concept of inheritance. Let's re-write the above code to understand the concepts of inheritance and prototypes:

```
Dog = function (name) {
     this.name = name;
}

Dog.prototype.legs = "4";

myDog1 = new Dog ("Bonzo");
myDog2 = new Dog ("Oliver");

trace (myDog1.name);          //Bonzo
trace (myDog2.name);          //Oliver
trace (myDog1.legs);          //4
trace (myDog2.legs);          //4
trace (Dog.prototype.legs);   //4
```

Let's think about what we just did. As earlier, we made a Dog class. But this time, all our class said:

- Every dog will have a name. This name will be decided by the creator of the dog.

Then we used the prototype object of the Dog class. We created a property in the prototype object of a Dog class called "legs" and assigned it a value of "4." By doing this, we created one master copy of the legs property for all Dogs. Now, all dogs would have 4 legs, but the neat thing is that we do not end up making a copy of the legs property for every new instance of a `Dog`. Instead, we let every `Dog` *inherit* the legs property of its Class. So

every time we call the legs property for an instance, it looks in the object's prototype to see if legs exist and if it does, it returns it.

It's tough to visualize a prototype. The simplest way to think of a prototype is as a window that lets you look into a Class. Think of a prototype as an object that contains all the properties that a class possesses (that is, the ones that you assigned to the prototype) and lets you manipulate those properties. Another way to look at it - a prototype is a ticket that lets you enter into a Class and look at, add, delete, or manipulate stuff in the Class.

When should a prototype be used?

Even though the prototype is so powerful, it is best used in the following circumstances:

- When you would like to make something unique that will remain constant for all instances of a Class ex. legs
- When you need to define a static *method* available to all instances of a Class ex. bark()

Inherited Methods

To take inheritance a step further, we can even assign methods to the prototype object of a Class. For instance, we can do something like:

```
Dog = function (name) {
     this.name = name;
}

Dog.prototype.legs = "4";

//Inherited Method
Dog.prototype.bark = function () {
     return (this.name + " is barking!");
}

myDog1 = new Dog ("Bonzo");
trace (myDog1.bark()); //Bonzo is barking!
```

What is the benefit of doing it this way? This way we avoid making a bark method for every instance of the Dog. Instead, by making a bark method in the prototype, we create the method universal to all Dogs and let a Dog borrow it every time it needs to bark. In fact, this is the preferred way to do application programming.

Overriding and Overwriting

These two terms are very common to programming, but are used quite loosely. However, it is important to make a distinction between them so as to understand more advanced concepts.

Overriding

So far, all our Dogs have their own names and four legs. Let's say that Bonzo was a gifted Dog and got one extra leg. How do we achieve this in our code? We have already defined that all dogs have four legs, right? We could do something like:

```
Dog.prototype.legs = "5";
```

But this would make all dogs five-legged. That's not what we wanted to do. So how does OOP let us give Bonzo five legs without changing the number of legs for every Dog? Here's how:

```
Dog = function (name) {
     this.name = name;
}

Dog.prototype.legs = "4";

myDog1 = new Dog ("Bonzo");
trace (myDog1.legs); //4

myDog1.legs = "5";

trace (myDog1.legs); //5
```

We gave all Dogs four legs. Then we said that Bonzo needs five legs. This process is called overriding. It's all about hierarchy. When we request the legs property, what Flash really does is look for the closest instance of legs. This implies that Flash will search for the legs property within the object itself and if it doesn't find it there, it will search in the object's parent. What if it didn't find it here? It'd look in that object's parent and so on.

The great thing about overriding is the fact that it does not write over the original legs property. In fact, all dogs other than Bonzo will still have 4 legs. For instance, if you did something like this:

```
MyDog2 = new Dog ("Oliver");
trace (myDog1.legs); //4
```

Overwriting
Overwriting on the other hand is destructive. To "overwrite" something means to change its value permanently. If we wanted to return Bonzo to his normal state (with 4 legs) we would overwrite his legs property that has five legs in the following manner:

```
myDog1.legs = "4";

trace (myDog1.legs); //4
```

Now Bonzo will have four legs for the remainder of the program, or till someone overwrites Bonzo's legs property again.

SuperClasses and SubClasses

Often you need to share the methods and properties of a class with another class. One way of doing this would be to simply declare all the methods and properties all over again for the new class. However, this uses excess memory and defeats the purpose of inheritance. Wouldn't it be nice to be able to inherit all the methods and properties of a class into a new Class, and then build on the new Class to customize it?

Let's use a new illustration for the next few concepts. We are going to use a Shape Class to serve as the *SuperClass* for all shapes. Its *SubClasses* would be of type triangle, rectangle, circle, etc. Each one of these SubClasses may have SubClasses of their own, for instance, right triangle, isosceles triangle, etc. *SuperClasses and* SubClasses share a Parent-child relationship. Hence, a child will have everything that a parent has, and more. Look at the following code:

```
//Shape Class
Shape = function() {}

//Shape Method
Shape.prototype.method = function() {
    trace("New Shape created!");
}

//Triangle Class
Triangle = function () {}

//Triangle Class inherits Shape Class
Triangle.prototype = new Shape();
```

```
//Triangle Method
Triangle.prototype.method = function() {
    super.method();
    trace("New Triangle created!");
}

//SubTriangle Class
SubTriangle = function (name) {
    this.name = name;
}

//SubTriangle Class Inherits Triangle Class
SubTriangle.prototype = new Triangle();

//SubTriangle Method
SubTriangle.prototype.method = function () {
    super.method();
    trace ("New SubTriangle created!");
}

//Return name of SubTriangle
SubTriangle.prototype.getName = function() {
    trace ("New SubTriangle is a " + this.name + "
triangle!");
}

myRightTriangle = new SubTriangle("right");
myRightTriangle.method();
myRightTriangle.getName();

// OUTPUT
// New Shape created!
// New Triangle created!
// New SubTriangle created!
// New SubTriangle is a right triangle!
```

Let's dissect the above code:

```
//Shape Class
Shape = function() {}

//Shape Method
Shape.prototype.method = function() {
     trace("New Shape created!");
}
```

First we declare a Shape class. All Shapes inherit from this Class. Then we assign the Shape Class an inheritable method called `method.` This method simply traces a string indicating that a Shape has been instantiated.

```
//Triangle Class
Triangle = function () {}
```

Then we create a Triangle Class.

```
//Triangle Class inherits Shape Class
Triangle.prototype = new Shape();
```

Since the Triangle class is a SubClass of the Shape Class it inherits all of Shape Class's properties and methods using the Class's prototype object.

```
//Triangle Method
Triangle.prototype.method = function() {
     super.method();
     trace("New Triangle created!");
}
```

Then we override (actually, this is not truly overriding, but for our purposes it's fine to call it that) the Shape Class method `method` using the super keyword. We also *extend* it by adding a trace function that indicates the instantiation of a Triangle object.

```
//SubTriangle Class
SubTriangle = function (name) {
     this.name = name;
}
```

Similarily, we create a SubTriangle Class. This Class accepts an argument called "name."

```
//SubTriangle Class Inherits Triangle Class
SubTriangle.prototype = new Triangle();
```

Consequently, the SubTriangle Class is simply a type of a Tringle Object, so it inherits the Triangle Class properties and methods using the prototype object.

```
//SubTriangle Method
SubTriangle.prototype.method = function () {
     super.method();
     trace ("New SubTriangle created!");
}
```

Now, we override the Tringle method `method` in a similar fashion to what we did earlier.

```
//Return name of SubTriangle
SubTriangle.prototype.getName = function() {
     trace ("New SubTriangle is a " + this.name +
" triangle!");
}
```

We add a new method to the SubTriangle prototype object called `getName.` This method returns the name of the SubTriangle. We are now done with our coding. Let's test it.

```
myRightTriangle = new SubTriangle("right");
myRightTriangle.method();
myRightTriangle.getName();
```

We create a right angle triangle of type SubTriangle and pass it a name. Notice the output for the above code.

```
// OUTPUT
// New Shape created!
// New Triangle created!
// New SubTriangle created!
// New SubTriangle is a right triangle!
```

The SubTriangle Class inherits the Triangle Class that in turn inherits the Shape Class. Further, the SubTriangle `method` overrides the Triangle method and that in turn overrides the Shape `method.` This is how the chain of objects works in OOP. It's very logical, and efficient.

Building your own custom prototypes

Most of you must already be excited about how OOP works in Flash because it gives you a chance to create extremely powerful and reusable Classes. Not only that, you can even expand on the available Flash objects such as Math, Array, etc. There are a few intricacies involved, but on the whole, it's very simple.

Let's create a simple custom method for the Math object that converts degrees to radians. Here's the code to do it:

```
Math.radian = function (degrees) {
     return (degrees * Math.PI/180);
};
```

This method would be called in the following manner:

```
Math.radian(180); //result - 3.14159265358979
```

However, this won't work with, say the MovieClip object. Let's make a custom method for the MovieClip Object that finds the distance between two clips on stage. Here is what the code would look like:

```
MovieClip.prototype.distance = function (tc){

     var dx = Math.floor (Math.sqrt(((this._x -
tc._x) * (this._x - tc._x))));
     var dy = Math.floor (Math.sqrt(((this._y -
tc._y) * (this._y - tc._y))));
     var distance = dx + dy;
     return distance;
}
```

The only thing different about this method from the last one is the manner in which it is declared. Here's where we need to make the distinction. While some objects require instantiation in Flash before they are used, others don't. For instance, the Array object is used in the following manner:

```
MyArray = new Array ();
```

Now, if we need to make a custom method for the array object, we must declare it in the following manner:

```
Array.prototype.myMethod = function ([args]) {
     //code
}
```

On the other hand, if you are dealing with an object such a Math, where you call methods simply by doing something like Math.sin(), then you need to declare the method as follows:

```
Math.myMethod = function ([args]) {
     //code
};
```

Remember to include the custom method in any movie that you plan to use it in. It's best to include it in the first frame of the movie using an include statement such as

```
#include myMethods.as
```

Conclusion

This was a quick and dirty introduction to OOP in Flash. We strongly recommend that you get yourself some books and go through the recommended web sites to get better with OOP. Good OOP skills come from a lot of experience and practice. It's not something that you can develop overnight. However, once you get the hang of it, you will notice a great increase of efficiency and portability in your code. If you are serious about application development in Flash, you definitely need to be very proficient with OOP techniques.

APPENDIX D

Additional Readings and References

Nairne, James S. (1999). *Psychology. The Adaptive Mind* (2nd ed.). California: Wadsworth/Thomson Learning.

Hewitt, Paul G. (1985). *Conceptual Physics - A New Introduction.* Boston: Little, Brown and Company.

Swokowski, Jeffery A. Cole & Earl, W. (1997). *Algebra and Trigonometry with Analytic Geometry*. California: Brooks/Cole Publishing Company.

Downing, Douglas. (2001). *Trigonometry: the easy way* (3rd ed.). New York: Barron's Educational Series, Inc.

Lehrman, Robert L. (1998). *Physics: the easy way* (3rd ed.). New York: Barron's Educational Series, Inc.

Hearn, Donald, & Baker, Pauline M. (1997). *Computer Graphics, C version* (2nd ed.). New Jersey: Prentice Hall.

LaMothe, Andre. (1999). *Tricks of the Windows Game Programming Gurus - Fundamentals of 2D and 3D Game Programming.* Indiana: Sams.

Mohler, James L. (2002). *Flash MX: Graphics, Animation and Interactivity*. New York: OnWord Press.

Hudson, Ralph G. (1939). *The Engineers' Manual* (2nd ed.). NewYork: John Wiley and Sons.

Gibilisco, Stan., & Crowhurst, Norman H. (1999). *Mastering Technical Mathematics* (2nd ed.). New York: McGraw-Hill.

Hodgman, Charles D. (1951). *Mathematical Tables* (9th ed.). Cleveland, Ohio: Chemical Rubber Publishing Co.

Downing, Douglas. (1995). *Dictionary of Mathematics Terms* (2nd ed.). New York: Barron's Educational Series, Inc.

Cutnell, John D., & Johnson, Kenneth W. (2001). *Physics* (Vol. 1). NewYork: John Wiley and Sons.

Vince, John. (2001). *Essential mathematics for computer graphics fast*. New York: Springer.

Lafore, Robert. (1999). *Object Oriented Programming in C++* (3r ed.). Indianapolis, Indiana: Sams.

Bourg, David M. (2001). *Physics for Game Developers*. California: O'Reily and Associates.

Rice, Bernard J. (1986). *Plane Trigonometry* (4th ed.). Boston: Prindle, Weber & Schmidt.

Lewis, John & Loftus, William. (1998). *Java software solutions*: foundations of program design. Massachusetts: Addison-Wesley.

Web sites:

Just What is OO Programming. *Debreuil.* Retrieved February 18, 2002, from http://www.debreuil.com/docs/ch01_Intro.htm

[PROTO]type. *Layer51.* Retrieved February 22, 2002, from http://www.layer51.com/proto/

Simple Object Oriented Glossary. *FlashCoders.* Retrieved March 10, 2002, from http://chattyfig.figleaf.com/flashcoders-wiki/index.php?Simple%20Object-Oriented%20Glossary

Object Oriented Programming Part I. *ProFlasher.* Retrieved April 1, 2002, from http://www.proflasher.com/index.php?sid=articles&aid=1

Object Oriented Programming Part II. *ProFlasher.* Retrieved April 1, 2002, from http://www.proflasher.com/index.php?sid=articles&aid=1

FlashGuru's Knowledge_base. *FlashGuru.* Retrieved March 3, 2002, from http://www.flashguru.co.uk/

Object-Oriented Programming Concepts. *Java.sun.com.* Retrieved March 14, 2002, from http://java.sun.com/docs/books/tutorial/java/concepts/

Vectors - Motion and Forces in Two Dimensions. *The Physics Classroom.* Retrieved February 17, 2002, from http://www.physicsclassroom.com/Class/vectors/vectoc.html

Weisstein, Eric. (2002). *Eric Weisstein's World of Mathematics*. Retrieved March 11, 2002, from http://mathworld.wolfram.com/