Skills and Topics for TeenCoder: Game Programming

Our Self-Study Approach

Our courses are **self-study** and can be completed on the student's own computer, at their own pace. You can steer your student in the right direction with no prior programming knowledge. Students only need typical computer usage skills to start; we will teach them programming from the ground up!

Each course comes with student activity starters, supplementary instructional documents, a **Solution Guide**, fully coded solutions for all activities, tests and answer keys, and guidance on evaluating projects.

Most questions about how to code individual activities are easily answered by referring to the Solution Guide (with or without parental involvement). We also provide **free technical support** to assist with any aspect of the courses!

Teachers who wish to closely monitor and grade student progress for credit purposes can administer chapter tests which are provided (with answer keys). We also provide advice and guidelines for evaluating student activities.

What Skills do Students Need to Begin?

All of our courses assume the student is already familiar with using a keyboard and mouse to select and run software, navigate the menus in a typical software program, and generally interact with their computer.

Students should understand how to use the built-in operating system software (like Windows Explorer) to find, save and retrieve files on their computer. It may also be helpful to have some familiarity with text editors (like Notepad or WordPad) and some experience using web browsers to find information on the Internet. We teach students how to program a computer from the ground up, but they should already know the basics about using one!

This course requires a **Windows** computer with CD-ROM or DVD-ROM.

TeenCoder: Game Programming is a second-semester course. **Students must have successfully completed the first-semester TeenCoder: Windows Programming course prior to starting TeenCoder: Game Programming!**

Topics Covered In This Course

The following are some of the computer programming topics that are covered in this course. For a full list of topics and sections, please see the Table of Contents for this course.

- Introduction to the XNA Game Studio
- Game design, game engines, and timer loops
- Screen coordinates and color concepts
- Drawing, scaling, and rotating images
- Handling keyboard, mouse, and XBox 360 Gamepad controller inputs
- Creating Sprite objects
- Collision detection
- 2D animation techniques
- Playing music and sound effects
- Game physics
- Maze generation and solution algorithms
- Menus, overlays, and deployment models
- Multi-player scrolling games
- Game artificial intelligence (AI)

TeenCoder™ Series

Game Programming

A hands-on introduction to the field of Game Programming for high school students.

Student Textbook

Third Edition

TeenCoder™ Series

TeenCoder™: Game Programming

Student Textbook

Third Edition

Homeschool Programming, Inc.

TeenCoder™: Game Programming

Third Edition

980 Birmingham Rd, Suite 501-128
Alpharetta, GA 30004

ISBN: **978-0-9887033-2-2**

Terms of Use

Disclaimer

Homeschool Programming, Inc, and their officers and shareholders, assume no liability for damage to personal computers or loss of data residing on personal computers arising due to the use or misuse of this course material. Always follow instructions provided by the manufacturer of 3rd party programs that may be included or referenced by this course.

Contact Us

You may contact Homeschool Programming, Inc. through the information and links provided on our website: http://www.HomeschoolProgramming.com. We welcome your comments and questions regarding this course or other related programming courses you would like to study!

Other Courses

Homeschool Programming, Inc. currently has two product lines for students: KidCoder™ and TeenCoder™. Our KidCoder™ Series provides easy, step-by-step programming curriculum for 4th through 12th graders. The Visual Basic series teaches introductory programming concepts in a fun, graphical manner. The Web Design series lets students create their own websites in HTML. Our TeenCoder™ Series provides introductory programming curriculum for high-school students. These courses are college-preparatory material designed for the student who may wish to pursue a career in Computer Science or enhance their transcript with a technical elective. Students can learn C#, Java, game programming, and Android application development.

3rd Party Copyrights

This course uses Microsoft's Visual C# 2010 Express as the programming platform. Visual Studio, Visual Studio Express, Windows, and all related products are copyright Microsoft Corporation. Please see http://www.microsoft.com/visualstudio/eng/products/visual-studio-2010-express for more details.

Instructional Videos

This course may be accompanied by optional Instructional Videos! These Flash-based videos will play directly from a DVD drive on the student's computer. Instructional Videos are supplements to the Student Textbook, covering every chapter and lesson with fun, animated re-enforcement of the main topics.

Instructional Videos are intended for students who enjoy a more audio-visual style of learning. They are not replacements for the Student Textbook which is still required to complete this course! However by watching the Instructional Videos first, students may begin each textbook chapter and lesson already having some grasp of the material to be read. Where applicable, the videos will also show "screencasts" of a real programmer demonstrating some concept or activity within the software development environment.

This Student Textbook and accompanying material are entirely sufficient to complete the course successfully! Instructional Videos are optional for students who would benefit from the alternate presentation of the material. For more information or to purchase the videos separately, please refer to the product descriptions on our website: http://www.HomeschoolProgramming.com.

Table of Contents

Before You Begin

Please read the following topics before you begin the course.

Minimum Hardware and Software Requirements

This is a hands-on programming course! You will be installing Microsoft's Visual C# 2010 Express on your computer, which must meet the following minimum requirements:

Computer Hardware

Your computer hardware must meet the following minimum specifications:

	Minimum
CPU	1.6GHz or faster processor
RAM	1024 MB
Display	1024 x 768 or higher resolution
Graphics Card	Supports DirectX-10 or later
Hard Disk Size	3GB available space
DVD Drive	DVD-ROM drive

Xbox 360 Gamepad controllers are optional input devices that may be used if available, but they are not required for any activity.

Operating Systems

Your computer operating system must match one of the following:

Windows XP (x86) with Service Pack 3 or above (except Starter Edition)
Windows Vista (x86 and x64) with Service Pack 2 or above (except Starter Edition)
Windows 7 (x86 and x64)
Windows 8 or Windows 8 Pro (excluding Windows 8 RT)

Conventions Used in This Text

This course will use certain styles (fonts, borders, etc) to highlight text of special interest.

```
Source code will be in 11-point Consolas font, in a single box like this.
```

Variable names will be in **`12-point Consolas bold`** text, similar to the way they will look in your development environment. For example: **`myVariable`**.

Function names, properties and keywords will be in **bold face** type, so that they are easily readable.

This picture highlights important concepts within a lesson.

Sidebars may contain additional information, tips, or background material.

Chapter Review section will highlight key elements from each chapter.

Each chapter includes one or more activities that allow you to practice the concepts you have learned.

What You Will Learn and Do In This Course

TeenCoder™: Game Programming will teach you how to write simple games on your own computer! You will be using Microsoft's C# programming language, the Visual C# 2010 Express development environment, and Microsoft's XNA Game Studio 4.0 development kit. This course is geared for high-school students who are already comfortable with C# and object-oriented programming concepts.

This course will not teach you how to write games by stitching together a few pre-built widgets in a predefined game environment or how to create complex animation or 3D environments. To do those things you are using *someone else's* platform to hide many of the interesting and fundamental concepts required to build games from scratch. Upon completion of this course you will understand many of the techniques used to create a wide range of game types!

What You Need to Know Before Starting

You must have completed the *TeenCoder™: Windows Programming* course prior to starting this course. The Visual C# and object-oriented concepts learned in the first course are prerequisites to learning and enjoying this game programming material.

You are also expected to already know the basics of computer use before beginning this course. You need to know how to use the keyboard and mouse to select and run programs, use application menu systems, and work with the Windows operating system. You should understand how to store and load files on your hard disk, and how to use the Windows Explorer to walk through your file system and directory structures. You should also have some experience using text editors and finding helpful information on the Internet.

Software Versions

You will be using the *Microsoft Visual C# 2010 Express* software and the *XNA Game Studio 4.0* to complete this course. These programs can be freely downloaded from Microsoft's website. Your course will contain links to download and install instructions on our website, http://www.HomeschoolProgramming.com. Microsoft may from time to time change their website or download process, or release newer versions of the product. Our website will contain updated versions of the instructions as needed.

Getting Help & Course Errata

All courses come with a Solution Guide PDF and fully coded solutions for all activities. Simply install the "Solution Files" from your course setup program and you will be able to refer to the solutions as needed from the "Solution Menu". If you are confused about any activity you can see how we solved the problem!

We welcome your feedback regarding any course details that are unclear or that may need correction. You can find a list of course errata for this edition on our website.

SAMPLE STUDENT LESSON

The following pages contain a sample student lesson from the TeenCoder: Game Programming textbook.

Chapter Ten: Game Physics

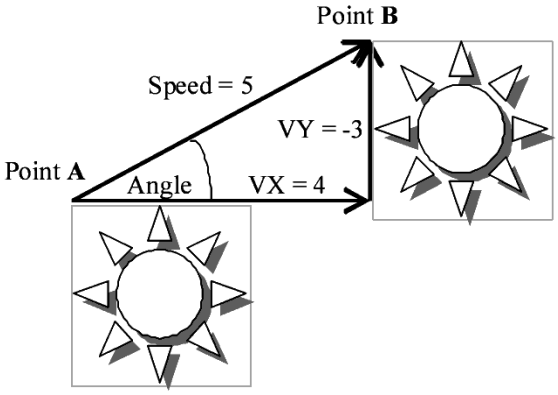
The laws of physics determine how objects move, change speeds, and bounce or reflect off hard surfaces. You will often want to simulate these laws to govern sprite movement and make your games more realistic.

Lesson One: Velocity and Acceleration

In an earlier chapter we discussed velocity in the context of sprite movement. In this lesson we will look at how *acceleration* can change velocity over time.

Velocity

Recall from Chapter Six that *velocity* is a combination of *speed* (change in position per unit time) and *direction* (angle of movement). We also know that the velocity vector can be represented equally well by components in the X and Y direction instead of a speed and angle. We can always convert velocity representation between the (speed, angle) pair and the (velocity X, velocity Y) pair using some trigonometric functions.



In trigonometry, if you know one side of a right triangle and the angle of one of its corners, you can calculate its other two sides. The diagram on the right shows how this calculation works when moving a sprite. The overall speed is 5 at an angle of 36.9 degrees. From these two values we can use some easy calculations to find the velocity in the X and Y directions:

```
Velocity X =  5 * Cosine (36.9°) = 4
Velocity Y = -5 *   Sine (36.9°) = -3
```

This means that if a sprite is moving with speed 5 at 36.9 degrees, we move the sprite by adjusting the upper-left corner position by 4 pixels in the **X** direction and -3 pixels in the **Y** direction. Remember that in computer terms the "up" direction is negative and the "down" direction is positive, which is opposite to the way that the math world works. This is why we use the negative value for our **Velocity Y** calculation.

What if we knew the velocity-X (VX) and velocity-Y (VY) components and wanted to calculate the speed and direction? The speed can be calculated by using *Pythagorean's theorem*. Pythagorean's theorem states that the

square of the hypotenuse (diagonal edge) of a right triangle equals the sum of the squares of the two other sides of the triangle: $a^2 + b^2 = c^2$. In our case, the calculation is: $VX^2 + VY^2 =$ (overall speed of the sprite)^2. The square root of this result is our overall speed.

```
Speed = Sqrt(VX² + VY²)
```

Given the VX and VY components you can also determine the direction angle based on the tangent function. We know that for a right triangle, the opposite side (VY) divided by the near side (VX) equals the tangent of the angle. Therefore the inverse tangent (arctan) of the VY divided by VX equals the angle:

```
Angle = ArcTan( VY / VX )
```

This formula unfortunately isn't specific enough to determine which quadrant the angle actually resides in, since the input parameter can be positive or negative and there are two combinations of positive or negative VX and VY that will produce a positive or negative. Fortunately the .NET framework has another method called **Math.Atan2()** which handles this nicely!

```
double Math.Atan2(double y, double x)
```

The resulting value (in radians) has been assigned to the correct quadrant based on the positive and negative combination of the individual input parameters.

The **Sprite** has a static utility method that wraps up this calculation for you, accepting an input **Vector2** and returning the corresponding angle in degrees:

```
static public double CalculateDirectionAngle(Vector2 vect)
```

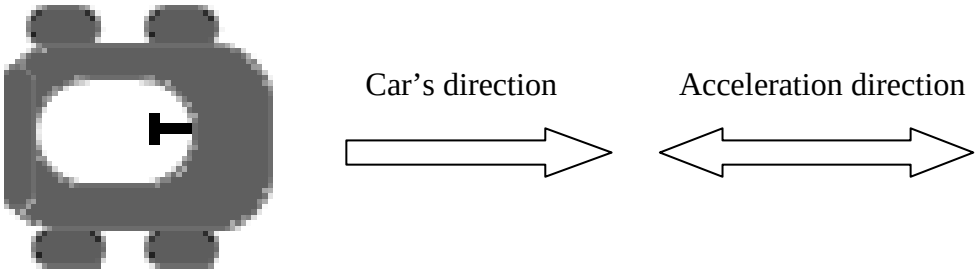
Acceleration

The speed of an object is the distance that the object can travel over a certain amount of time. For example, the speed of a car may be measured in the number of miles or kilometers it travels in an hour. *Acceleration* is a measure of how fast you are speeding up or slowing down. If you step on the gas pedal in a car, the car will increase its speed, or *accelerate*. If you step on the brake pedal, the car will slow down, or *decelerate*. The rate that the car speeds up or slows down is its *acceleration* or *deceleration*. So, for instance, if your car is traveling at 20 miles per hour and you accelerate 5 miles per hour, the car will end up traveling at 25 miles per hour (the original speed plus the acceleration).

Graphical objects on the screen can also accelerate and decelerate. Let's say you have a simple program in which a spaceship is flying through outer space. When the program starts the spaceship is moving at a speed of 5 pixels a second. Now let's say the ship is being chased by aliens. The player can hit the plus key (+) to

accelerate the ship by 2 pixels a second for each key press. If the user hits the key 3 times then the ship's speed is accelerated to 5 + 2 + 2 + 2 = 11 pixels a second! Voila! The spaceship can escape the evil aliens.

You will typically want to accelerate an object in one of two ways. First, in many cases the acceleration is working along the same angle as the sprite is moving. The car example falls into this case, where acceleration (or breaking, which is just negative acceleration) always happens in whatever direction the car is heading. You can specify a single acceleration value that should impact the sprite's overall speed without changing the angle.



The **Sprite** class contains an **Accelerate**() method to accelerate (with a positive parameter) or decelerate (with a negative parameter) the sprite's speed along the current angle of travel.

```
void Accelerate(double acceleration)
```

Using trigonometry, the method calculates the X and Y components for the input acceleration based on the current direction angle and then adds these values to the current **Velocity.X** and **Velocity.Y** properties.

Just like velocity, acceleration can be represented as an acceleration magnitude and direction or as acceleration components in the X and Y directions (AX, AY). In order to adjust velocity by some acceleration, you can convert both velocity and acceleration to their X and Y components and then simply add the acceleration X to the velocity X and the acceleration Y to the velocity Y values.

```
Velocity.X = Velocity.X + acceleration.X
Velocity.Y = Velocity.Y + acceleration.Y
```

If you wish to accelerate a **Sprite** using the AX and AY components individually, of course there is method to accomplish this too!

```
void Accelerate(double AX, double AY)
```

This method will directly add the specified acceleration in the X and Y directions to the current velocity X and Y components.

Lesson Two: Gravity and Wind

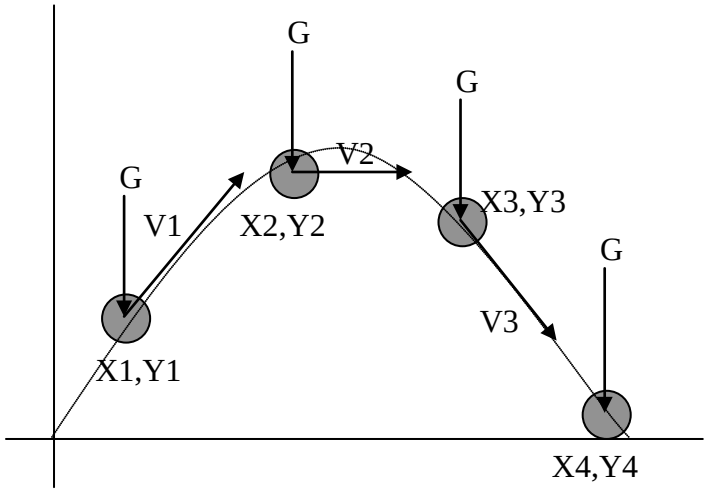
Many computer games involve the use of objects that are either thrown or shot into the air. The movement of these objects may be influenced by gravity and wind effects.

Gravity

In the real world, every object that is launched or thrown into the air is influenced by the force of gravity. The Earth's gravity is a force which is constantly pulling everything towards the center of the earth (or, in terms of the computer screen, straight down in the positive Y direction).

When you throw a ball into the air it will travel up and away from you. As soon as it leaves your hands, gravity begins to work on the ball. No matter how hard you throw, the ball will eventually stop going in the upwards direction and will start moving down towards the ground. The arc that the ball follows from the time you throw it until the time it hits the ground is called a parabolic arc, which is shaped like an upside-down 'U'.

Let's take a look at a diagram that shows a ball moving under the force of gravity on the screen:



The original position of the ball is shown as point (X1, Y1). The ball has an initial velocity of V1, which is moving the ball in the negative Y direction (up), and positive X direction (to the right). We also have the force of gravity "G", which is pushing down on the ball. The effect is to reduce the upward velocity in the Y direction and leave the X velocity untouched.

The next position, point (X2, Y2), shows where the ball is after one or more iterations of the **Update**() method. At this point, gravity has slowed the Y velocity to near zero, and our ball is mostly moving in the positive X direction (still to the right).

The next position, point (X3, Y3) again shows the position of the ball after another timer tick or two. Now, gravity has reversed the Y velocity so that the direction is now positive (moving down on the screen). Finally, the last position of the ball (X4, Y4) shows the ball where it has landed on the "ground".

The curved line in the diagram shows the parabolic arc the ball would complete in the real world. Notice that the speed of the ball from left to right in the X direction is completely unaffected by the force of gravity. Gravity only acts to change the velocity in the Y direction. Also notice that the force of gravity "G" is a constant that affects the ball equally at each position.

We now know the values for both the X and Y acceleration components! There is zero (0.0) acceleration in the X direction because the X velocity does not change. The Y acceleration is a constant "G" representing the force of gravity. The second version of **Sprite.Accelerate**() accepting the component AX and AY values is ideal for our use in this situation.

So, to add the effects of gravity to a sprite flying through the air, we could use the following code:

```
    mySprite.Accelerate(0.0f, 0.04f);
```

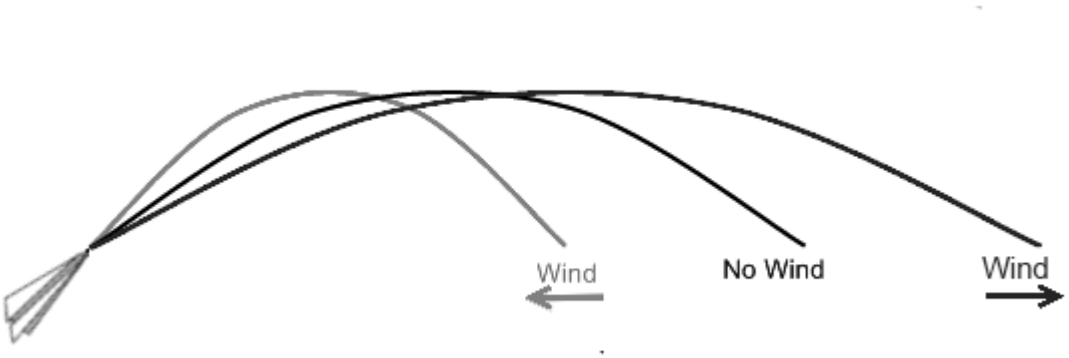
While gravity may be known as 9.8 m/s^2 in the real world, we don't have the concept of a "meter" on the computer screen. We are also applying this force 60 times a second (assuming default timing for the **Update**() function call). You'll have to play with several different AY values to find the strength of gravity value that "feels" right for your game environment. We picked 0.04f above. The size of the AY parameter will determine how quickly the object will fall back down to the ground.

Wind

Up until now we have only considered projectiles that are travelling through the air without any impact from the wind. Wind affects almost every projectile in the air. Golfers often check the wind before lining up their shots. Paper airplanes fly further if they are propelled by the wind, and baseball players may hit more home runs with the wind at their back than if the wind is blowing in their face. Adding the effects of wind for flying objects in a game may be a nice touch of realism or even be crucial to game-play!

So how do we add wind to a game? In the last section, we learned that gravity is a constant acceleration in the **Y** direction but did not affect the **X** velocity. The force of gravity caused our projectile to move in an arc on the screen. Wind, on the other hand, can be considered a constant acceleration in the **X** direction (assuming it is simply blowing left or right with no up or down-drafts). A wind blowing to the left would be a negative **X** acceleration, and wind blowing to the right would be a positive **X** acceleration.

The wind can have a powerful affect on the velocity and trajectory of a projectile. Consider the following illustration of a paper airplane in flight:



The paper airplane is thrown three different times with the same initial starting location, direction, and speed. The middle end-point represents the path that the airplane will take if there is no wind at all. With no wind, only the force of gravity affects the plane's path.

The right end-point shows the path the airplane will take if the wind is heading in the same direction as the airplane. This type of wind is called a *tailwind*, which is any wind travelling in the same direction as the projectile. The airplane will travel much further with a tailwind!

The left end-point shows the path the airplane will take when the wind is travelling in the opposite direction. This type of wind is called a *headwind*. With a headwind the airplane's flight will be much shorter!

How do you implement the effects of wind in your game? Easy! You already have an acceleration function that will split apart acceleration in either the **X** or **Y** direction. You added gravity to a projectile by specifying acceleration in the **Y** direction. The wind can just be thought of as acceleration in the **X** direction. A positive **X** acceleration will simulate a wind that travels from left to right on the screen. A negative **X** acceleration will simulate a wind that travels from right to left on the screen.

To add both gravity and wind effects to a sprite flying through the air, we could use the following code:

```
    mySprite.Accelerate(0.03f, 0.04f);
```

A positive AX parameter (like 0.03) will give the effect of wind blowing from left to right on the screen. To add a leftward wind instead, just use a negative value like -0.03. Again, just like gravity, you'll have to play with the magnitude of this number to see what feels right in your game. Pick a value that is too high and you've just created a tornado that will overwhelm every projectile motion in your game!

SAMPLE SOLUTION GUIDE

The following pages contain sample solution material for an activity in the TeenCoder: Game Programming textbook.

Chapter Two Activity (Looping Colors)

In this activity, the student will create a new game project that will cause the game screen to change colors every second. This program will teach the student how to update a game window at a specific interval.

Game Screen

The game screen should show a solid color that changes once a second. Students may choose the specific colors.

Code Required to Complete this Activity

The student will begin by adding two new variables at the top of the **LoopingColor** class definition. Students will choose the variable names, so they do not need to match our examples below:

```
public class LoopingColor : Microsoft.Xna.Framework.Game
{
    // declare an array of colors
    private Color[] backgroundColors;

    // declare an integer index into the array
    int currentColor;
```

Next, in the **Initialize**() method, the student will set the frequency of the **Update**() call to one second:

```
protected override void Initialize()
{
      this.TargetElapsedTime = TimeSpan.FromSeconds(1.0f);
```

Also in the **Initialize**() method, the student will fill out the array of background colors with any **Colors** of their choosing and initialize their current color index variable to 0:

```
    // initialize an array of length 5 with a variety of colors
    backgroundColors = new Color[5]
         {Color.Red, Color.Blue, Color.Yellow, Color.Violet, Color.PapayaWhip};

    // set the current color index to 0
    currentColor = 0;
```

The student will then add some code to the **Update**() method to change the current color index:

```
protected override void Update(GameTime gameTime)
{
    // move on to the next color
    currentColor++;

    // if the index is past the end of the array
    if (currentColor >= backgroundColors.Length)
    {
          currentColor = 0;   // reset back to the beginning of the array!
    }
```

Finally, the student will change one line in the **Draw**() method to the following:

```
protected override void Draw(GameTime gameTime)
{
      GraphicsDevice.Clear(backgroundColors[currentColor]);
```

When run, the program should cycle through the array of colors once per second, showing that color as a solid background in the program window.

The completed project for this activity is located in the "Activity Solutions\Looping Colors" folder underneath the Solution Files installation directory.